Using the Ruby DBI Module

Paul DuBois
paul@kitebird.com

Document revision: 1.03
Last update: 2006-11-28

Introduction

The Ruby DBI module provides a database-independent interface for Ruby scripts similar to that of the Perl DBI module. This document describes how to write Ruby DBI-based scripts. It is an adjunct to and not a substitute for the Ruby DBI specification documents. See the ‘‘Resources’’ section for a pointer to the specifications and also for information about downloading the example scripts used here.

The general architecture for Ruby DBI uses two layers:

- The database interface (DBI) layer. This layer is database independent and provides a set of common access methods that are used the same way regardless of the type of database server with which you’re communicating.
- The database driver (DBD) layer. This layer is database dependent; different drivers provide access to different database engines. There is one driver for MySQL, another for PostgreSQL, another for InterBase, another for Oracle, and so forth. Each driver interprets requests from the DBI layer and maps them onto requests appropriate for a given type of database server.

The examples in this document use the MySQL database driver, but for many of them you should be able to substitute other drivers. Some of the MySQL-specific features require Ruby DBI 0.1.1 or higher.

Prerequisites

The Ruby DBI module includes the code that implements the general DBI layer, as well as a set of DBD-level drivers. Many of these drivers require that you have additional software installed. For example, the database driver for MySQL is written in Ruby and provides a binding to the Ruby MySQL module, which itself is written in C and provides a binding to the MySQL C client API. This means that if you want to write DBI scripts to access MySQL databases, you’ll need to have both the Ruby MySQL module and the C API installed. For further information on the Ruby MySQL module, see the document referenced in the ‘‘Resources’’ section. Here, I assume that the MySQL module is installed and available for use by DBI.

Installation

After you have satisfied the prerequisites described in the previous section, you can install the Ruby DBI module, which can be obtained from the following site:

```
http://rubyforge.org/projects/ruby-dbi/
```

The DBI module is distributed as a compressed *tar* file, which you should unpack after downloading it. For example, if the current version is 0.1.1, the distribution file can be unpacked using either of the following commands:

```
% tar zxf dbi-0.1.1.tar.gz
```

```
% gunzip < dbi-0.1.1.tar.gz | tar xf -
```

After unpacking the distribution, change location into its top-level directory and configure it using the *setup.rb* script in that directory. The most general configuration command looks like this, with no arguments following the `config` argument:

```
% ruby setup.rb config
```

That command configures the distribution to install all drivers by default. To be more specific, provide a *--with* option that lists the particular parts of the distribution you want to use. For example, to configure only the main DBI module and the MySQL DBD-level driver, issue the following command:

```
% ruby setup.rb config --with=dbi,dbd_mysql
```

After configuring the distribution, build and install it:

```
% ruby setup.rb setup
% ruby setup.rb install
```

You might need to run the installation command as `root`.

The rest of this document uses the following notational conventions:

- ‘‘DBI module’’ refers collectively to the DBI layer as well as the DBD-level drivers, unless context indicates that only the database independent layer is meant.
- ‘‘`DBD::Mysql`’’ refers to the MySQL-specific database driver for DBI.
- ‘‘Ruby MySQL module’’ refers to the module on which `DBD::Mysql` is built (that is, the module that provides the bindings to the MySQL C client library).

A Simple DBI Script

With the Ruby DBI module installed, you should be able to access your MySQL server from within Ruby programs. Assume for purposes of this article that the server is running on the local host and that you have access to a database named `test` by connecting using an account that has a username and password of `testuser` and `testpass`. You can set up this account by using the *mysql* program to connect to the server as the MySQL `root` user and issuing the following statement:

```
mysql> GRANT ALL ON test.* TO 'testuser'@'localhost' IDENTIFIED BY 'testpass';
```

If the `test` database does not exist, create it with this statement:

```
mysql> CREATE DATABASE test;
```

If you want to use a different server host, username, password, or database name, just substitute the appropriate values in each of the scripts discussed in the remainder of this article.

The following script, *simple.rb*, is a short DBI program that just connects to the server, retrieves and displays the server version, and disconnects. You can download the script from the link listed in the ‘‘Resources’’ section, or use a text editor to create it directly:

```
#!/usr/bin/ruby -w
# simple.rb - simple MySQL script using Ruby DBI module

require "dbi"

begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:test:localhost", "testuser", "testpass")
  # get server version string and display it
  row = dbh.select_one("SELECT VERSION()")
```

```
  puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code: #{e.err}"
  puts "Error message: #{e.errstr}"
ensure
  # disconnect from server
  dbh.disconnect if dbh
end
```

The *simple.rb* script provides a very basic general overview of DBI concepts. The immediately following discussion explains how it works, and later sections show other examples that provide additional detail on specific aspects of DBI programming.

simple.rb begins with a `require` line that pulls in the DBI module; without that line, DBI methods will fail. The rest of the script is placed within a `begin/rescue/ensure` construct:

- The `begin` block handles all the database processing.
- The `rescue` clause handles any exceptions that occur; it obtains and displays error information.
- The `ensure` clause make sure that the script closes any open connection to the database server.

The `connect` method establishes a connection to the database server and returns a database handle to use for further communication with the server. The first argument is a data source name (DSN) that indicates the driver name (which for MySQL is `Mysql`), the default database name, and the hostname of the server. The second and third arguments are the username and password of the MySQL account to use. Other ways to write DSN values are covered later in the section ‘‘More on Connecting to the Server.’’

simple.rb uses the database handle to call `select_one`, a method that sends a statement to the server and returns the first row of the result as an array. The `SELECT VERSION()` statement returns a single value, so the version string is available as `row[0]`, the first (and only) element of the array. When you run the script, the result looks something like this:

```
% ruby simple.rb
Server version: 5.1.14-beta-log
```

Errors cause exceptions to be raised. Various kinds of exceptions can occur, but most of those related to database operations cause a `DatabaseError` exception to be raised. Objects of this exception class have `err`, `errstr`, and `state` attributes that represent the error number, descriptive error message, and SQLSTATE value. *simple.rb* prints the error number and message for database exceptions, and ignores other types of exceptions. (Should another exception occur, it is passed back to Ruby itself for processing.)

simple.rb terminates the connection to the server by invoking the `disconnect` method. This is done in an `ensure` clause to make sure that connection termination occurs even if an error occurs during statement processing.

Processing SQL Statements

Ruby DBI provides many ways to execute statements. This section discusses a few of them, but there are others.

Many of the examples use a table named `people` that has the following structure:

```
CREATE TABLE people
(
  id     INT UNSIGNED NOT NULL AUTO_INCREMENT, # ID number
  name   CHAR(20) NOT NULL,                    # name
  height FLOAT,                                # height in inches
```

```
  PRIMARY KEY (id)
);
```

Processing Statements that Return No Result Set

Statements that do not return rows can be issued by invoking the `do` database handle method. This method takes a statement string argument and returns a count of the number of rows affected by the statement. The following example uses `do` several times to create the `people` table and populate it with a small data set:

```
dbh.do("DROP TABLE IF EXISTS people")
dbh.do("CREATE TABLE people (
          id INT UNSIGNED NOT NULL AUTO_INCREMENT,
          name CHAR(20) NOT NULL,
          height FLOAT,
          PRIMARY KEY (id))")
rows = dbh.do("INSERT INTO people (name,height)
               VALUES
                 ('Wanda',62.5),
                 ('Robert',75),
                 ('Phillip',71.5),
                 ('Sarah',68)")
puts "Number of rows inserted: #{rows}"
```

For the `INSERT` statement, this script obtains the row count and displays it to indicate how many rows were added to the table.

Processing Statements that Return a Result Set

Statements such as `SELECT` or `SHOW` return rows. To process such a statement, send it to the server for execution, retrieve any rows in the result set that it generates, and dispose of the result set.

One way to do this is to call `prepare` to generate a statement handle. Use that handle to execute the statement and fetch its results, and then call `finish` to dispose of the result set:

```
sth = dbh.prepare(statement)
sth.execute
... fetch rows ...
sth.finish
```

It's also possible to pass the statement directly to `execute` and skip the call to `prepare`:

```
sth = dbh.execute(statement)
... fetch rows ...
sth.finish
```

There are many ways to fetch results after executing a statement. You can call `fetch` as a standalone method in a loop until it returns `nil`:

```
sth = dbh.execute("SELECT * FROM people")
while row = sth.fetch do
  printf "ID: %d, Name: %s, Height: %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

`fetch` can also be used as an iterator, in which case it is the same as `each`. The following two row-fetching loops are equivalent:

```
sth = dbh.execute("SELECT * FROM people")
sth.fetch do |row|
  printf "ID: %d, Name: %s, Height: %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

```
sth = dbh.execute("SELECT * FROM people")
sth.each do |row|
  printf "ID: %d, Name: %s, Height: %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

`fetch` and `each` produce `DBI::Row` objects, which have several methods for accessing their contents:

- Column values can be accessed by index or name using array notation:

```
val = row[2]
val = row["height"]
```

- You can use a row object with `by_index` or `by_field` to access column values by number or by name:

```
val = row.by_index(2)
val = row.by_field("height")
```

- An iterator method, `each_with_name`, produces each column value along with the column name:

```
row.each_with_name do |val, name|
  printf "%s: %s, ", name, val.to_s
end
print "\n"
```

- `DBI::Row` objects have a `column_names` method that returns an array containing the names for each column. `field_names` is an alias for `column_names`.

Other row-fetching methods include `fetch_array` and `fetch_hash`. These do not return `DBI::Row` objects. Instead, they return the next row as an array or a hash, or `nil` if there are no more rows. Hashes returned by `fetch_hash` are keyed by column name with column values as hash values. Either method can be invoked in standalone fashion or as an iterator. The following examples demonstrate this for `fetch_hash`:

```
sth = dbh.execute("SELECT * FROM people")
while row = sth.fetch_hash do
  printf "ID: %d, Name: %s, Height: %.1f\n",
         row["id"], row["name"], row["height"]
end
sth.finish

sth = dbh.execute("SELECT * FROM people")
sth.fetch_hash do |row|
  printf "ID: %d, Name: %s, Height: %.1f\n",
         row["id"], row["name"], row["height"]
end
sth.finish
```

You can avoid the execute-fetch-finish sequence by using database handle methods that do all the work for you and return the results:

```
row = dbh.select_one(statement)
rows = dbh.select_all(statement)
```

`select_one` executes a statement and returns the first row as an array, or `nil` if the statement returns no rows. `select_all` returns an array of `DBI::Row` objects. You can access the contents of these objects as discussed earlier. The array is empty if the statement returns no rows.

The MySQL driver examines the metadata for the result set and uses it to coerce row values to the corresponding Ruby data type. (This means, for example,, that `id`, `name`, and `height` values retrieved from the `people` table are returned as `Fixnum`, `String`, and `Float` objects.) However, be aware that if a column value is `NULL`, it is represented as `nil` in the result set and has a type of `NilClass`. Also note

that this coercion behavior does not appear to be mandated by the DBI specification and might not be performed by all drivers.

Quoting, Placeholders, and Parameter Binding

Ruby DBI provides a placeholder mechanism that enables you to avoid including data values literally in a statement string. Instead, you use special '?' placeholder markers within the statement to indicate where the data values go. When you execute the statement, you provide values to be bound to the placeholders. DBI substitutes the values into the statement where the placeholders appear, performing any quoting of string values and escaping of special characters as necessary. This makes it easy to construct statements without having to know whether the values contain special characters, and without having to do any quote processing yourself. The placeholder mechanism also properly handles `NULL` values; provide `nil` as a data value and it is placed into the statement as an unquoted `NULL` value.

The following example illustrates how this works. Suppose you want to add a new row to the `people` table for someone named Na'il (a name that includes a quote), who is 76 inches tall. To indicate where the data values go in the `INSERT` statement, use '?' placeholder markers (without any surrounding quotes), and provide the data values as additional arguments to `do` following the statement:

```
dbh.do("INSERT INTO people (id, name, height) VALUES(?, ?, ?)",
       nil, "Na'il", 76)
```

The resulting statement produced by `do` and sent to the server looks like this:

```
INSERT INTO people (id,name,height) VALUES(NULL,'Na\'il',76)
```

If you plan to execute a statement multiple times, you can prepare it first to obtain a statement handle, and then execute it with the data values as arguments. Assume that a data file named *people.txt* contains lines of tab-delimited name/height pairs to be inserted into the `people` table. The following example reads the file to obtain row data, executing a prepared `INSERT` statement once for each row:

```
# prepare statement for use within insert loop
sth = dbh.prepare("INSERT INTO people (id, name, height) VALUES(?, ?, ?)")

# read each line from file, split into values, and insert into database
File.open("people.txt", "r") do |f|
  f.each_line do |line|
    name, height = line.chomp.split("\t")
    sth.execute(nil, name, height)
  end
end
```

Preparing a statement first and then executing it multiple times within a loop is more efficient than invoking `do` each time through the loop (which in effect calls both `prepare` and `execute` for each iteration). The difference is most significant for database engines that prepare a query execution plan and reuse it for each call to `execute`. MySQL doesn't do this; Oracle does.

To use placeholders for `SELECT` statements, the proper strategy depends on whether you prepare the statement first:

- If you invoke `prepare` to obtain a statement handle, use that handle to call `execute` and pass it the data values to be bound to the placeholders:

  ```
  sth = dbh.prepare("SELECT * FROM people WHERE name = ?")
  sth.execute("Na'il")
  sth.fetch do |row|
    printf "ID: %d, Name: %s, Height: %.1f\n", row[0], row[1], row[2]
  end
  sth.finish
  ```

- If you don't use `prepare`, the first argument to `execute` is the statement and the following

arguments are the data values:

```
sth = dbh.execute("SELECT * FROM people WHERE name = ?", "Na'il")
sth.fetch do |row|
  printf "ID: %d, Name: %s, Height: %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

Other drivers might allow or require that you represent placeholders differently. For example, you might write placeholders as `:name` or `:n` to specify them in named or numbered form. Consult the documentation for the driver that you want to use.

The `quote` method performs quoting and escaping of a data value and returns the result. This can be useful for constructing statements to be executed by other programs. For example, if you want to read the data file *people.txt* and convert it to a set of `INSERT` statements that can be processed by a program such as the *mysql* command-line client, do this:

```
# read each line from file, split into values, and write INSERT statement
File.open("people.txt", "r") do |f|
  f.each_line do |line|
    name, height = line.chomp.split("\t")
    printf "INSERT INTO people (id, name, height) VALUES(%s, %s, %s);\n",
           dbh.quote(nil), dbh.quote(name), dbh.quote(height)
  end
end
```

Statement Result Metadata

For statements that return no result set, such as `INSERT` or `DELETE`, the `do` method returns a count of the number of rows processed.

For statements that return rows, such as `SELECT`, you can use the statement handle after invoking `execute` to get row and column counts or information about each of the columns in the result set:

- The row and column counts are not available directly. To get the row count, either count the rows as you fetch them, or fetch them into a data structure and see how many elements it contains. To get the column count, you can determine it from the number of column names, available as `sth.column_names.size`.
- The `column_info` method returns information about each column.

This script shows how to obtain metadata for a statement:

```
sth = dbh.execute(stmt)

puts "Statement: #{stmt}"
if sth.column_names.size == 0 then
  puts "Statement has no result set"
  printf "Number of rows affected: %d\n", sth.rows
else
  puts "Statement has a result set"
  rows = sth.fetch_all
  printf "Number of rows: %d\n", rows.size
  printf "Number of columns: %d\n", sth.column_names.size
  sth.column_info.each_with_index do |info, i|
    printf "--- Column %d (%s) ---\n", i, info["name"]
    printf "sql_type:         %s\n", info["sql_type"]
    printf "type_name:        %s\n", info["type_name"]
    printf "precision:        %s\n", info["precision"]
    printf "scale:            %s\n", info["scale"]
    printf "nullable:         %s\n", info["nullable"]
    printf "indexed:          %s\n", info["indexed"]
```

```
    printf "primary:          %s\n", info["primary"]
    printf "unique:           %s\n", info["unique"]
    printf "mysql_type:       %s\n", info["mysql_type"]
    printf "mysql_type_name:  %s\n", info["mysql_type_name"]
    printf "mysql_length:     %s\n", info["mysql_length"]
    printf "mysql_max_length: %s\n", info["mysql_max_length"]
    printf "mysql_flags:      %s\n", info["mysql_flags"]
  end
end
sth.finish
```

Members of `column_info` objects are accessible two ways. The script just shown accesses them using hash member notation, but you can also access them using `info.`*`member_name`* notation. For example, you can get the column name using either of these values:

```
info.["name"]
info.name
```

Note: Older versions of this document stated that you can get the row count for a `SELECT` result as `sth.rows`. That is not supported. (It currently does happen to work for the MySQL driver, but you should not rely on this behavior.)

Methods That Take Code Blocks

Some handle-creating methods can be invoked with a code block. When executed this way, they provide the handle to the code block as its parameter, and automatically clean up the handle when the block terminates:

- `DBI.connect` generates a database handle, for which it calls `disconnect` if necessary at the end of the block.
- `dbh.prepare` generates a statement handle, for which it calls `finish` at the end of the block. Within the block, you must invoke `execute` to execute the statement.
- `dbh.execute` is similar except you don't invoke `execute` within the block; the statement handle is automatically executed.

The following example illustrates use of a code block with each of those handle-creating methods:

```
# connect can take a code block, passes the database handle to it,
# and automatically disconnects the handle at the end of the block

DBI.connect("DBI:Mysql:test:localhost", "testuser", "testpass") do |dbh|

  # prepare can take a code block, passes the statement handle
  # to it, and automatically calls finish at the end of the block

  dbh.prepare("SHOW DATABASES") do |sth|
    sth.execute
    puts "Databases: " + sth.fetch_all.join(", ")
  end

  # execute can take a code block, passes the statement handle
  # to it, and automatically calls finish at the end of the block

  dbh.execute("SHOW DATABASES") do |sth|
    puts "Databases: " + sth.fetch_all.join(", ")
  end
end
```

There is also a `transaction` method that takes a code block. It is described in ''Transaction Support.''

More on Connecting to the Server

The *simple.rb* script shown earlier connects to the server using the DBI `connect` method as follows:

```
dbh = DBI.connect("DBI:Mysql:test:localhost", "testuser", "testpass")
```

The first argument to `connect` is the data source name (DSN); it identifies the type of connection to make. The other two arguments are the username and password of your MySQL account.

The DSN can be given in any of the following formats:

```
DBI:driver_name
DBI:driver_name:db_name:host_name
DBI:driver_name:param=val;param=val...
```

The DSN always begins with `DBI` or `dbi` (in uppercase or lowercase, but not in mixed case) and the driver name. For MySQL, the driver name is `Mysql`, and it's best to use exactly that capitalization. (There is some indication in the DBI specification that lettercase of the driver name should not matter, but that is not always true up through DBI versions as recent as 0.0.18.) For other drivers, you'll need to use the appropriate driver name.

`DBI` (or `dbi`) and the driver name must always be given in the DSN. If nothing follows the driver name, the driver may (I think) attempt to connect using a default database and host name. The second format requires two values, a database name and hostname separated by a colon. The third format allows a list of parameter assignments to be specified following the second colon (which is required), in *`param=value`* format separated by semicolons. The following DSNs are all equivalent:

```
DBI:Mysql:test:localhost
DBI:Mysql:host=localhost;database=test
DBI:Mysql:database=test;host=localhost
```

The DSN syntax that uses *`param=value`* format is the most flexible because it allows the parameters to be specified in any order. It also allows for the possibility of driver-specific parameters, which means that drivers can be extensible in the connection parameters they accept. For MySQL, several of the parameters correspond to arguments of the `mysql_real_connect()` C API function:

- `host=`*`host_name`*

  The host where the MySQL server runs

- `database=`*`db_name`*

  The database name

- `port=`*`port_num`*

  The TCP/IP port number, for non-`localhost` connections

- `socket=`*`path_name`*

  The pathname of the Unix socket file, for `localhost` connections

- `flag=`*`num`*

  Flags to enable

MySQL programs can read options from option files, as described in the MySQL Reference Manual. Two DSN parameters enable Ruby DBI scripts to use this capability:

- `mysql_read_default_file=`*`file_name`*

  Read options only from the named option file.

- `mysql_read_default_group=`*`group_name`*

  Read options from the `[`*`group_name`*`]` option group (and from the `[client]` group, if *`group_name`* differs from `client`).

If neither option is given, option files are not used. If only `mysql_read_default_group` is given, options are read from the standard option files (such as *.my.cnf* in your home directory and */etc/my.cnf* on Unix). The following example shows how to connect using any `[client]` group options in the standard option files:

```
dsn = "DBI:Mysql:mysql_read_default_group=client"
dbh = DBI.connect(dsn,nil,nil)
```

Other DSN options:

- `mysql_compression={0|1}`

  Disable or enable compression in the client/server protocol. The default is not to use it.

- `mysql_client_found_rows={0|1}`

  For statements that modify rows, MySQL by default returns a row count of the number of rows actually changed. You can use `mysql_client_found_rows=1` to tell the server to return a count of the rows matched by the statement, regardless of whether they were changed. For example, by default the following statement causes MySQL to return a row count of 0 because no value in any row changes:

  ```
  UPDATE t SET id = id;
  ```

  With `mysql_client_found_rows=1`, the row count will be equal to the number of rows in the table.

Error Handling and Debugging

If a DBI method fails, DBI raises an exception. DBI methods may raise any of several types of exception, but for database-related operations, the relevant exception class is `DatabaseError`. Exception objects of this class have three attributes named `err`, `errstr`, and `state`, which represent the error number, a descriptive error string, and a ‘‘standard’’ error code. For MySQL, these values correspond to the return values of the `mysql_errno()`, `mysql_error()`, and `mysql_sqlsate()` C API functions. You can obtain these values when an exception occurs as follows:

```
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code: #{e.err}"
  puts "Error message: #{e.errstr}"
  puts "Error SQLSTATE: #{e.state}"
```

If your version of the MySQL Ruby module is old and does not provide SQLSTATE information, `e.state` is `nil`.

To get debugging information about what your script is doing as it executes, you can enable tracing. To do this, you must first load the `dbi/trace` module:

```
require "dbi/trace"
```

The `dbi/trace` module is not loaded automatically by the `dbi` module because it is dependent on version 0.3.3 or newer of the AspectR module, which may not be present on your machine.

The `dbi/trace` module provides a `trace` method that controls the trace mode and output destination:

```
trace(mode, destination)
```

The *mode* value may be 0 (off), 1, 2, or 3, and the *destination* should be an IO object. The default values are 2 and STDERR, respectively.

trace can be invoked as a class method to affect all subsequently created handles, or as an object method for individual driver, database, or statement handles. When invoked as an object method, any other objects subsequently derived from that object also inherit the trace setting. For example, if you enable tracing on a database handle, statement handles created from it from that point on are given the same trace setting.

Transaction Support

DBI provides a transaction abstraction. However, availability of the abstraction depends on transaction support in your database engine, and on a DBD-level implementation of the abstraction in your driver. For the MySQL driver, this abstraction is not functional prior to DBI 0.0.19, so you must perform transactions by explicitly using statements that control the auto-commit level, commits, and rollbacks. For example:

```
dbh.do("SET AUTOCOMMIT=0")
dbh.do("BEGIN")
... statements that make up the transaction ...
dbh.do("COMMIT")
```

For DBI 0.0.19 and up, you can use the transaction abstraction with MySQL. One aspect of the abstraction enables the auto-commit level to be set by assigning to the database handle AutoCommit attribute:

```
dbh['AutoCommit'] = true
dbh['AutoCommit'] = false
```

While auto-commit is disabled (set to false), you can perform transactions two ways. The following examples illustrate the two approaches, using an account table for which funds are transferred from one person to another.

- The first approach uses DBI's commit and rollback methods to explicitly commit or cancel the transaction:

```
dbh['AutoCommit'] = false
begin
  dbh.do("UPDATE account SET balance = balance - 50 WHERE name = 'bill'")
  dbh.do("UPDATE account SET balance = balance + 50 WHERE name = 'bob'")
  dbh.commit
rescue
  puts "transaction failed"
  dbh.rollback
end
dbh['AutoCommit'] = true
```

- The second approach uses the transaction method. This is simpler, because it takes a code block containing the statements that make up the transaction. The transaction method executes the block, then invokes commit or rollback automatically, depending on whether the block succeeds or fails:

```
dbh['AutoCommit'] = false
dbh.transaction do |dbh|
  dbh.do("UPDATE account SET balance = balance - 50 WHERE name = 'bill'")
  dbh.do("UPDATE account SET balance = balance + 50 WHERE name = 'bob'")
end
dbh['AutoCommit'] = true
```

Accessing Driver-Specific Capabilities

DBI provides a `func` database-handle method that drivers can use to make database-dependent features available. For example, the MySQL C API provides a `mysql_insert_id()` function that returns the most recent `AUTO_INCREMENT` value for a connection. The Ruby MySQL module provides a binding to this function via its `insert_id` database-handle method, and `DBD::Mysql` in turn provides access to `insert_id` via the DBI `func` mechanism.

The first argument to `func` is the name of the database-specific method that you want to use; any following arguments are those required by the method. The `insert_id` method requires no additional arguments, so to access the most recent `AUTO_INCREMENT` value, do this:

```
dbh.do("INSERT INTO people (name,height) VALUES('Mike',70.5)")
id = dbh.func(:insert_id)
puts "ID for new record: #{id}"
```

Other driver-specific methods supported by `DBD::Mysql` are:

`dbh.func(:createdb, db_name)`	Create a new database
`dbh.func(:dropdb, db_name)`	Drop a database
`dbh.func(:reload)`	Perform a reload operation
`dbh.func(:shutdown)`	Shut down the server

The `createdb` and `dropdb` methods are unavailable unless your MySQL client library comes from a version older than MySQL 4 (they correspond to deprecated functions that the Ruby MySQL module does not support as of MySQL 4).

As of DBI 0.1.1, a number of other `func` methods are available. They correspond to several functions in the MySQL C API:

```
String = dbh.func(:client_info)
Fixnum = dbh.func(:client_version)
String = dbh.func(:host_info)
String = dbh.func(:info)
Fixnum = dbh.func(:proto_info)
String = dbh.func(:server_info)
String = dbh.func(:stat)
Fixnum = dbh.func(:thread_id)
```

In some cases, use of a driver-specific function may offer specific advantages, even if there is another way to accomplish the same thing. For example, the value returned by the `insert_id` function of `DBD::Mysql` can be obtained by issuing a `SELECT LAST_INSERT_ID()` statement. Both return the same value in most cases. However, the `insert_id` function is more efficient because it returns a value that is stored on the client side and can be accessed without issuing another statement. This efficiency benefit comes at the cost of greater care in how you use the function. Its value is reset for each statement executed, so you must access it after issuing the statement that generates an `AUTO_INCREMENT` value but before issuing any other statement. By contrast, the value of `LAST_INSERT_ID()` is stored on the server side and is more persistent; it is not reset by other statements except those that also generate `AUTO_INCREMENT` values.

Other DBI Goodies

The `DBI::Utils` module includes a few interesting methods:

- `DBI::Utils::measure` takes a code block and measures how long it takes to execute the code within the block. You can use this method to measure the wallclock time for execution of a statement as follows:

```
elapsed = DBI::Utils::measure do
```

```
  dbh.do(stmt)
end
puts "Statement: #{stmt}"
puts "Elapsed time: #{elapsed}"
```

- The `DBI::Utils::TableFormatter` module has an `ascii` method for displaying the contents of a result set. The first argument is an array of column names, and the second is an array of row objects. To display the contents of the `people` table, do this:

```
sth = dbh.execute("SELECT * FROM people")
rows = sth.fetch_all
col_names = sth.column_names
sth.finish
DBI::Utils::TableFormatter.ascii(col_names, rows)
```

  The resulting output is:

```
+----+---------+--------+
| id | name    | height |
+----+---------+--------+
| 1  | Wanda   | 62.5   |
| 2  | Robert  | 75.0   |
| 3  | Phillip | 71.5   |
| 4  | Sarah   | 68.0   |
+----+---------+--------+
```

- The `DBI::Utils::XMLFormatter` module has `row` and `table` methods for displaying individual result set rows or an entire result set as XML. This makes it easy to generate XML output from query results. The following example demonstrates the `table` method:

```
DBI::Utils::XMLFormatter.table(dbh.select_all("SELECT * FROM people"))
```

  The resulting output is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rows>
<row>
  <id>1</id>
  <name>Wanda</name>
  <height>62.5</height>
</row>
<row>
  <id>2</id>
  <name>Robert</name>
  <height>75.0</height>
</row>
<row>
  <id>3</id>
  <name>Phillip</name>
  <height>71.5</height>
</row>
<row>
  <id>4</id>
  <name>Sarah</name>
  <height>68.0</height>
</row>
</rows>
```

The `ascii` and `table` methods support a number of optional arguments that provide greater control over the output format and destination. See the module source code for more information.

Resources

The scripts that are used for examples in this document can be downloaded from the following location:

```
http://www.kitebird.com/articles/
```

That location also provides access to a document, ‘‘Using the Ruby MySQL Module,’’ that discusses the module that forms the basis for `DBD::Mysql`, the DBD-level MySQL driver for DBI.

You may find the following additional resources helpful for using Ruby DBI:

- You can get the Ruby DBI module and specification documents from the DBI RubyForge site:

  ```
  http://rubyforge.org/projects/ruby-dbi/
  ```

- The second edition of *MySQL Cookbook*, (O’Reilly, 2006) adds Ruby DBI to the MySQL-access interfaces that it covers. The book shows many Ruby DBI examples:

  ```
  http://www.kitebird.com/mysql-cookbook/
  ```

- The AspectR Ruby module must be installed if you want to use the `dbi/trace` module that provides DBI execution tracing. You can get AspectR at its SourceForge site:

  ```
  http://aspectr.sourceforge.net/
  ```

- The Ruby home page provides general information about Ruby itself:

  ```
  http://www.ruby-lang.org/
  ```

- MySQL can be obtained at:

  ```
  http://www.mysql.com/
  ```

Revision History

- 1.00, 2003-01-19—Original version.
- 1.01, 2003-01-30—More info about `insert_id` driver-specific function. Clarify efficiency benefits of `prepare`.
- 1.02, 2003-05-27—Correct/clarify some comments in `block.rb` script. Correct misconception about `rows` method for `SELECT` statements. Describe transaction abstraction. Other minor updates.
- 1.03, 2006-11-28—Updated for Ruby DBI 0.1.1: Describe how to use the `mysql_`*`xxx`* DSN parameters. Describe how to get SQLSTATE error values. Metadata example script lists the additional `column_info` values that now are available. Describe new `func` methods. Other minor revisions.