Perl exercises

Deep Dive with Apache Derby: Perl, PHP, and Python

OSCON

August 2, 2005

Dan Scott

dan.scott@acm.org / dan.scott@ca.ibm.com

Objectives:

- Introduce the Perl Database Interface (DBI) specification
- Set up support for the Perl DBI and DBD::DB2 modules
- Connect to the Apache Derby database
- Create a Web page that pulls data from Apache Derby
- Create a Web page that inserts data into Apache Derby
- Create a script to insert binary large object (BLOB) data into Apache Derby

1. Introducing the Perl DBI specification

The standard database interface for Perl applications is the Perl DBI, currently at version 1.48, largely designed, developed, and maintained by Tim Bunce since 1994. The architecture for the Perl DBI defines the common DBI module, with database driver (DBD) modules providing connectivity for specific databases, as well as flat files, XML, and comma-separated value (CSV) files. With the approach of Perl 6, discussion on the dbi-dev mailing list is currently underway about the future of DBI.

To connect to Apache Derby, you should use the DBD::DB2 module. DBD::DB2 is written and maintained by IBM, and the source is freely available from the Comprehensive Perl Archive Network (CPAN).

2. Building and configuring the Perl DBI and DBD::DB2 modules

Prerequisites

- Installed DB2 Runtime Client with application development support
- Installed Perl 5.8.x
- (Linux): Application development tools and libraries such as perl-devel, gcc, autoconf, automake, bison, and flex.

Windows

On Windows, if you are using the ActiveState Perl distribution, you can install the Perl DBI and the DBD::DB2 modules by issuing the following command:

```
C:\> ppm install http://ftp.esoftmatic.com/outgoing/DBI/5.8.4/DBI.ppd
C:\> ppm install http://ftp.esoftmatic.com/outgoing/DBI/5.8.4/DBD-DB2.ppd
```

Linux

On Linux, you can install the Perl DBI and the DBD::DB2 modules by issuing the following commands:

```
bash$ export DB2_HOME=/home/db2inst1/sqllib/
bash$ perl -MCPAN -e 'install DBI'
bash$ perl -MCPAN -e 'install DBD-DB2'
```

3. Connecting to an Apache Derby database

In this exercise, you will create a Perl script named **`connect.pl`** that connects to the Apache Derby database named **MYDB** runnng on host **localhost** on port number 1527.

Your Perl script should begin with the following lines:

```
#!/usr/bin/perl
use strict;
use warnings;
```

The first line tells the Linux shell which interpreter should be used to process the rest of the script. The second line enforces good programming practices, like declaring lexical variables with **my**. The third line helps prevent you from committing questionable programming practices, like using a non-numeric value as one of the operands of the + operator.

You are using the Database Interface (DBI) module for all of your database access, so you must also include the following line:

```
use DBI;
```

For the purposes of simplicity, let's assign the database, user name, password, hostname, and port number values to local variables that we can use to create the database connection string:

```
my $database = 'MYDB';
my $user = 'lynn';
my $password = '5tuff';
my $hostname = 'localhost';
my $port = 1527;
```

Now we can build the complete database connection string for an uncataloged database connection (ensure that there are no spaces in this string and that it all appears on a single line):

```
my $DSN = "DRIVER={IBM DB2 ODBC DRIVER};
DATABASE=$database;HOSTNAME=$hostname;PORT=$port;PROTOCOL=TCPIP;";
```

To create a connection to a database using the Perl DBI and the DBD::DB2 driver, invoke the DBI->connect() method.

```
my $conn = DBI->connect("dbi:DB2:$DSN", $user, $password,
  { AutoCommit => 1 });

# Check to see if the connection was successful
if (!$conn) {
  print "Connection failed!\n";
}
else {
  print "Connection succeeded.\n";
  # Explicitly close our connection
  $conn->disconnect();
}
```

Now test the connection to your database.

```
bash$ perl connect.pl
```

Assuming that the Apache Derby Network Server is still running, you should receive confirmation that your Perl script has successfully connected to the database.

4. Displaying data in a Web page

Most Web applications either directly use the CGI module, or use a module that builds on top of CGI. In this section, we will use the CGI module to fetch GET and POST requests and set cookie values.

Copy the contents of **`connect.pl`** into a new script named **`search.pl`**.

Add `use CGI;` to the list of Perl modules to be imported.

4a. Creating a standard connect() function

Refactor the connection code into a Perl function that returns the connection object. One possible solution that hardcodes the connection information into the script itself follows:

```
sub db_connect() {
  my $database = 'MYDB';
  my $user = 'lynn';
  my $password = '5tuff';
  my $hostname = 'localhost';
  my $port = 1527;

  # Create the connection string for an uncataloged connection
  my $DSN = "DRIVER={IBM DB2 ODBC DRIVER};
DATABASE=$database;HOSTNAME=$hostname;PORT=$port;PROTOCOL=TCPIP;";

  my $conn = DBI->connect("dbi:DB2:$DSN", $user, $password,
    { AutoCommit => 1 });
  return $conn;
}
```

4b. Creating standard header() and footer() functions

Create a `header()` function that prints the requested title and displays the search form from the PHP exercises. Modify the header to call `/cgi-bin/search.pl`:

```
sub header {
  my ($title) = @_;
  my $header = "<html>
  <head><title>$title</title></head>
  <body><h1>$title</h1>";

  # add modified search code here

  return $header;
}

sub footer {
  return '</body></html>';
}
```

4c. Retrieving CGI POST and GET variables

The CGI module provides an object-oriented interface to generating HTML pages, accessing CGI variables, and manipulating cookies. Any GET and POST variables passed to the page are accessible through the `param()` method of the instantiated CGI object.

```
# Instantiate a new CGI object
my $query = new CGI;

# Print the HTTP header
print $query->header();

# Print our HTML header with search form
print header('Search for a menu item');

# Declare the $search variable in the global scope
my $search;

# Ensure the requested CGI parameter actually exists
if ($query->param('search')) {
  # Whitelist filter the input string to prevent naughty input
  ($search) = $query->param('search') =~ m/^([^\W_]+)$/;
}

if (!$search) {
  print "Please enter an alphanumeric search string.";
  exit;
}
```

4d. Issuing an SQL statement

To issue an SQL statement using the Perl DBI, you can either issue the statement directly using the `$dbh->do()` method, or, if the statement includes parameters or will be issued multiple times, you can `$dbh->prepare()` the statement and then `$sth->execute()` the resulting statement handle object.

```
# Connect to the database
my $conn = db_connect();

# Declare the SELECT statement
my $sql = q{SELECT name, description
      FROM menu.food
      WHERE name LIKE ?};

# Prepare the SELECT statement
# Apache Derby will return an error if the prepare fails.
my $sth = $conn->prepare($sql);

# Pass in the value for the query placeholder
my $result = $sth->execute($search);
```

4e. Retrieving rows from an SQL query

Once you have executed an SQL statement that returns a result set, such as a SELECT statement, the Perl DBI gives you many different methods for retrieving the rows. The fastest method, however, is to bind the returned columns to Perl variables and iterate through the rows as demonstrated below.

```
# Declare the variables to which the columns will be bound
my ($name, $description);

# Bind the variables to the returned columns
$result = $sth->bind_columns(\$name, \$description);

# Loop through each row
while ($sth->fetch) {
  print "<p>Name: $name <br />Description: $description</p>\n";
}
```

5. Inserting data into Apache Derby

5a. Creating a Perl module

Rather than recopying our Perl subroutines into every new script, let's factor the subroutines into a separate Perl module that we can reuse for the rest of our scripts. Create a new file called **`mydb.pm`** with the following code:

```
#!/usr/bin/perl
package mydb;
use strict;
use warnings;
use DBI;
use CGI;

BEGIN {
    use Exporter;
    our ($VERSION, @ISA, @EXPORT, @EXPORT_OK);
    $VERSION = 1.00;
    @ISA = qw{Exporter};
    # Export these functions from our module upon request
    @EXPORT_OK = qw {
        db_connect
        footer
        header
    };
}

our @EXPORT_OK;
```

Append your definitions for the `db_connect()`, `header()`, and `footer()` functions to this code.

5a. Inserting plain old data types

The Perl DBI automatically converts most plain old data types from Perl into their target data type in Apache Derby by treating input variables as VARCHAR data types and allowing Apache

Derby to convert the data accordingly. So, building on your experiences with creating SQL statements to inserting data in PHP and Python, and combining that with your knowledge of the Perl CGI module's forms-handling, build a Web form that lets us add new menu items and their corresponding prices to the database.

To re-use the `db_connect(), header(), and footer()` functions from your custom Perl module in your new script, create a new script called **`insert_data.pl`** beginning with the following lines:

```
#!/usr/bin/perl
use strict;
use warnings;
use DBI;
use CGI;
use mydb qw/ db_connect header footer /;
```

5b. Using transactions to commit or roll back a unit of work

So far, we have been connecting to Apache Derby and explicitly turning on AUTOCOMMIT as part of the DBI->connect() method parameters array. However, the Perl DBI also enables you to begin a new transaction and commit or roll back the unit of work using the `begin_work()`, `commit()`, and `rollback()` Connection methods.

Perl's version of exception handling executes all of the statements you want to try in an `eval {};` block, then checks the `$@` special variable to see if any errors were raised in the block. To raise errors from DBI connection objects, set the RaiseError attribute of the connection handle:

```
$conn->{RaiseError} = 1;
```

Modify your insert script to perform the set of INSERT statements within a transaction, and roll back all of the statements inside the transaction if just one of the statements fails.

5c. Inserting binary large object (BLOB) data types

Binary large objects (BLOBs) are chunks of binary data that, in Apache Derby, can be as large as 2 gigabytes in size. BLOB columns can be used to store documents, multimedia files like pictures or music, or other sets of binary data. While the performance of storing and retrieving BLOBs from a database typically suffers in comparison to the same files served directly from a filesystem, one of the primary advantages of storing BLOBs in the database is ensuring that your backup and restore operations keep your BLOBs in sync with the rest of the data in your database.

Create a new Perl script to insert a picture into your menu (at long last!). You can reuse the shell of your previous Perl script.

```
my $conn = db_connect();

my $sql = q{
  INSERT INTO menu.pictures(id, picture)
  VALUES (?, ?)
};

# Prepare the statement
my $stmt = $conn->prepare($sql);
```

```
# Create a variable to hold the item ID
my $menu_number = 1;

# Bind the ID as an input parameter
$stmt->bind_param(1, $menu_number);

# Create a variable with the location of our BLOB file
my $menu_pic = '/home/lynn/burger.png';

# Bind the BLOB as an input parameter, setting the db2_file attribute to true
$stmt->bind_param(2, $menu_pic, { db2_file => 1 });

# Execute the statement
my $result = $stmt->execute();

# Check the results of the statement and print success or failure message
if ($result) {
  print "Successfully inserted a picture into the menu.";
}
else {
  print "Failed with the following error: " . $conn->errstr;
}
```

To ensure that your BLOB was successfully inserted, use the LENGTH() scalar function to check the size of the picture column in the menu.pictures table to ensure that it matches the size of the file that you inserted into the database.