PHP and PostgreSQL

By Vikram Vaswani

Table of Contents

A Matter Of Choice

There's something patently unfair going on here. For some reason, almost every PHP tutorial on the planet makes the implicit assumption that if you're using PHP with a database, that database is going to be MySQL.

Now, I have absolutely nothing against MySQL − I think it's a great product, and I use it fairly often in my development activities. However, it's not the only good open−source RDBMS out there − most developers have been playing with PostgreSQL for quite a while now, and quite a few of them would love to integrate their PostgreSQL backend with PHP. The only problem is, they have no idea where to start − the functions used to communicate with a PostgreSQL database are different from those used in a MySQL environment, and the documentation out there could do with some updating.

Well, it's time to bring some balance back to the universe. Which is why this article looks at PHP from the PostgreSQL developer's point of view, explaining how PHP can be used with the PostgreSQL database system. If you're a MySQL user, you probably don't need to know any of this; if you're a PostgreSQL fan, on the other hand, you can breathe a sigh of relief and flip the page.

Getting Started

Before we get started, you need to make sure that you have everything you need to successfully use PHP with PostgreSQL. Here's your cheat sheet:

1. The latest version of PostgreSQL, available from http://www.postgresql.org (this article uses version 7.1)

2. A Web server which supports PHP, available from http://www.php.net/ (this article uses Apache 1.3.24 with PHP 4.2.0)

Note that your PHP build must support PostgreSQL in order for the examples in this article to work correctly. You can include PostgreSQL support in your PHP build by adding the "−−with−pgsql" configuration parameter when compiling the package. Note also that PHP 4.2.0 and better contains fairly extensive changes to the function names used in the language's PostgreSQL module; this article uses the new function names and assumes that you're running PHP 4.2.x. Drop by http://www.php.net/manual/en/ref.pgsql.php for more information on the changes, and the corresponding function names for older versions.

I'm not going to get into the details of configuring and installing either PostgreSQL or PHP here − the documentation included with both those packages has more than enough information to get you started, and the accompanying Web sites contain lots of troubleshooting information should you encounter problems. In case you don't already have these packages installed on your development system, drop by the Web sites listed above, get yourself set up and come back once you're done.

Assuming that you have a properly configured and installed setup, the first step is to start both the database server and the Web server.

```
[postgres@medusa] $ /usr/local/pgsql/bin/postmaster −i −D
/usr/local/pgsql/data &
DEBUG: database system was shut down at 2002−04−12 19:18:12
IST
DEBUG: CheckPoint record at (0, 1694744)
DEBUG: Redo record at (0, 1694744); Undo record at (0, 0);
Shutdown
TRUE
DEBUG: NextTransactionId: 643; NextOid: 18778
DEBUG: database system is in production state
[postgres@medusa] $ su −
root [root@medusa] $ /usr/local/apache/bin/apachectl start
Starting
httpd [OK] [root@medusa] $
```

Note that the PostgreSQL server must be started as the special "postgres" user created during the installation process, and the startup invocation must include the additional "−i" parameter to allow TCP/IP connections to the server.

The next step is to create an example database table that can be used for the code listings in this article. Here's what the SQL dump file looks like:

```
CREATE TABLE addressbook (id serial, name varchar(255),
address text,
tel varchar(50), email varchar(255));

INSERT INTO addressbook values (nextval('addressbook_id_seq'),
'Bugs
Bunny', 'The Rabbit Hole, Looney Toons, USA', '123 4567',
'bugs@wascallywabbit.net');

INSERT INTO addressbook values (nextval('addressbook_id_seq'),
'Robin
Hood', 'Sherwood Forest', 'None',
'robin@steal.from.the.rich');

INSERT INTO addressbook values (nextval('addressbook_id_seq'),
'Sherlock
Holmes', '221B Baker Street, London 16550, England', '911
1822',
'holmes@bakerstreetirregulars.domain');
```

You can import this data into PostgreSQL by dropping to a shell and using the following command:

```
[postgres@medusa] $ /usr/local/pgsql/bin/createdb test
[postgres@medusa]
$ /usr/local/pgsql/bin/psql −d test −f
/home/postgres/addressbook.sql
```

Now check whether or not the data has been successfully imported with a SELECT query (the SELECT SQL statement is used to retrieve information from a database) via the interactive PostgreSQL monitor program "psql".

```
[postgres@medusa] $ /usr/local/pgsql/bin/psql −d test
Welcome to psql, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
\h for help with SQL commands
\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit

test=# SELECT COUNT(*) FROM addressbook;
count
```

```
−−−−−−−
3
(1 row)

test=#
```

which, in English, means "count all the records in the table addressbook and give me the total".

If your output matches what's listed above, take a celebratory sip of Jolt − you're in business!

First Steps

Now, how about doing the same thing with PHP − fire a SELECT query at the database, and display the results in an HTML page?

```
<html>
<head><basefont face="Arial"></head>
<body>

<?
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// generate and execute a query
$query = "SELECT * FROM addressbook";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection));

// get the number of rows in the resultset
$rows = pg_num_rows($result);

echo "There are currently $rows records in the database.";

// close database connection
pg_close($connection);

?>

</body>
</html>
```

And here's what's the output looks like:

```
There are currently 3 records in the database.
```

As you can see, using PHP to get data from a PostgreSQL database involves several steps, each of which is actually a pre−defined PHP function. Let's dissect each step:

1. The first thing to do is specify some important information needed to establish a connection to the database server. This information includes the server name, the username and password required to gain access to it, and the name of the database to query. These values are all set up in regular PHP variables.

```
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";
```

2. In order to begin communication with the PostgreSQL database server, you first need to open a connection to the server. All communication between PHP and the database server takes place through this connection.

In order to initialize this connection, PHP offers the pg_connect() function.

```
// open a connection to the database server
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");
```

The function requires a connection string containing one or more parameters − these could include the host name, port, database name, user name and user password. Here are some examples of valid connection strings:

```
$connection = pg_connect ("host=myhost dbname=mydb");

$connection = pg_connect ("host=myhost dbname=mydb
user=postgres
password−postgres");

$connection = pg_connect ("host=myhost port=5432 dbname=mydb
user=postgres password−postgres");
```

This function then returns a "link identifier", which is stored in the variable $connection; this identifier is used throughout the script when communicating with the database.

3. Now that you have a connection to the database, it's time to send it a query via the pg_query() function. This function needs two parameters: the link identifier for the connection and the query string.

```
// generate and execute a query
$query = "SELECT * FROM addressbook";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection));
```

The result set returned by the function above is stored in the variable $result.

4. This result set may contain, depending on your query, one or more rows or columns of data. You then need to retrieve specific sections or subsets of the result set with different PHP functions − the one used here is the pg_num_rows() function, which counts the number of rows and returns the value needed.

```
// get the number of rows in the resultset
$rows = pg_num_rows($result);
```

There are several other alternatives you can use at this point, which will be explained a little further down.

5. Finally, each database connection occupies some amount of memory − and if your system is likely to experience heavy load, it's a good idea to use the pg_close() function to close the connection and free up the used memory.

```
// close database connection
pg_close($connection);
```

Simple, ain't it?

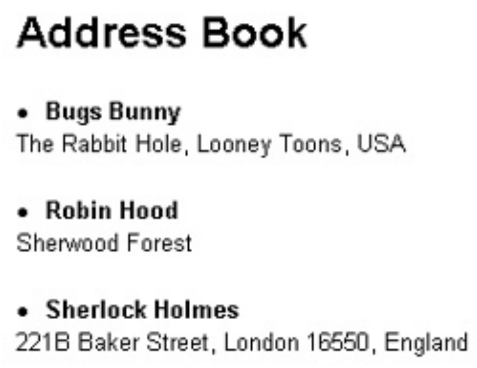
Digging Deeper

Now, that was a very basic example. How about something a little more useful?

This next example will query the database, return the list of addresses, and display them all as a neatly−formatted list.

```
<html>
<head><basefont face="Arial"></head>
<body>
<h2>Address Book</h2>
<?
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// generate and execute a query
$query = "SELECT name, address FROM addressbook ORDER BY
name"; $result
= pg_query($connection, $query) or die("Error in query:
$query. " .
pg_last_error($connection));

// get the number of rows in the resultset
// this is PG−specific
$rows = pg_num_rows($result);

// if records present
if ($rows > 0)
{
// iterate through resultset
for ($i=0; $i<$rows; $i++)
{
$row = pg_fetch_row($result, $i);
?>
<li><font size="-1"><b><? echo $row[0]; ?></b></font>
<br>
<font size="-1"><? echo $row[1]; ?></font>
<p>
<?
}
}
// if no records present
// display message
else
{
?>
<font size="-1">No data available.</font>
<?
}

// close database connection
pg_close($connection);

?>
</body>
</html>
```

Here's what the output looks like:



```
Address Book

• Bugs Bunny
The Rabbit Hole, Looney Toons, USA

• Robin Hood
Sherwood Forest

• Sherlock Holmes
221B Baker Street, London 16550, England
```

As in the previous example, the script first sets up a connection to the database. The query is formulated and the result set is returned to the browser. In this case, since I'm dealing with multiple rows of data, I've used the pg_fetch_row() function in combination with a "for" loop to iterate through the result set and print the data within each row.

The pg_fetch_row() function returns the columns within each row as array elements, making it possible to easily access the values within a record. By combining it with a "for" loop, I can easily process the entire result set, thereby displaying all returned records as list items.

Finally, in case you're wondering, the pg_last_error() function returns the last error generated by the server − combined with die(), this provides an effective, if primitive, debugging mechanism.

Different Strokes

Of course, there's more than one way of extracting records from a result set. The example on the previous page used an integer−indexed array; this one evolves it a little further so that the individual fields of each records are accessible as keys of a hash, or string−indexed array, via the pg_fetch_array() function.

```
<html>
<head><basefont face="Arial"></head>
<body>
<h2>Address Book</h2>
<?
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// generate and execute a query
$query = "SELECT name, address FROM addressbook ORDER BY
name"; $result
= pg_query($connection, $query) or die("Error in query:
$query. " .
pg_last_error($connection));

// get the number of rows in the resultset
// this is PG−specific
$rows = pg_num_rows($result);

// if records present
if ($rows > 0)
{
// iterate through resultset
for ($i=0; $i<$rows; $i++)
{
$row = pg_fetch_array($result, $i, PGSQL_ASSOC);
?>
<li><font size="−1"><b><? echo $row['name'];
?></b></font>
<br>
<font size="-1"><? echo $row['address']; ?></font>
<p>
<?
}
}
// if no records present
// display message
else
{
?>
<font size="-1">No data available.</font>
<?
}

// close database connection
pg_close($connection);

?>
</body>
</html>
```

Most of the magic here lies in the call to pg_fetch_array(),

```
$row = pg_fetch_array($result, $i, PGSQL_ASSOC);
```

which returns every row as a hash with keys corresponding to the column names.

PHP also allows you to access individual fields within a row as object properties rather than array elements, via its pg_fetch_object() function. Take a look:

```
<html>
<head><basefont face="Arial"></head>
<body>
<h2>Address Book</h2>
<?
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
```

```
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// generate and execute a query
$query = "SELECT name, address FROM addressbook ORDER BY
name"; $result
= pg_query($connection, $query) or die("Error in query:
$query. " .
pg_last_error($connection));

// get the number of rows in the resultset
// this is PG-specific
$rows = pg_num_rows($result);

// if records present
if ($rows > 0)
{
// iterate through resultset
for ($i=0; $i<$rows; $i++)
{
$row = pg_fetch_object($result, $i);
?>
<li><font size="-1"><b><? echo $row->name; ?></b></font>
<br>
<font size="-1"><? echo $row->address; ?></font>
<p>
<?
}
}
// if no records present
// display message
else
{
?>
<font size="-1">No data available.</font>
<?
}

// close database connection
pg_close($connection);

?>
</body>
</html>
```

In this case, each row is returned as a PHP object, whose properties correspond to field names; these fields can be accessed using standard object notation.

Rolling Around

One of the nice things about PostgreSQL − and one of the reasons why many developers prefer it over MySQL − is its support for transactions (in case you didn't know, this refers to the ability to group a series of SQL statements together so that they are executed either together, or not at all). You can find more information about transactions online, at http://www.postgresql.org/idocs/index.php?tutorial−transactions.html − and if you already know what they are, here's an example which demonstrates how they may be used in a PHP context with PostgreSQL.

```
<?
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// begin a transaction block
$query = "BEGIN WORK";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection));

// generate some queries
$query = "INSERT INTO addressbook values
(nextval('addressbook_id_seq'),
'Spiderman', 'The Web, Somewhere In Your Neighborhood',
'None',
'spidey@neigborhood.com')"; $result = pg_query($connection,
$query) or
die("Error in query: $query. " . pg_last_error($connection));

$query = "INSERT INTO addressbook values
(nextval('addressbook_id_seq'),
'Bruce Wayne', 'Gotham City', '64928 34585',
'bruce@batcave.org')";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection));

// now roll them back
$query = "ROLLBACK";
// if you want to commit them, comment out the line above
// and uncomment the one below
// $query = "COMMIT";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection));

// now check to see how many records are there in the table
// and print this
$query = "SELECT * FROM addressbook";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection)); $rows = pg_num_rows($result);
echo
"There are currently $rows records in the database";

// close database connection
pg_close($connection);

?>
```

Technically, there's nothing new here − this script uses the same functions you've seen in preceding examples. The difference lies in the use of multiple SQL statements to begin and end a transaction block, and in the use of COMMIT and ROLLBACK statements to commit and erase records from the database.

Catching Mistakes

All done? Nope, not quite yet − before you go out there and start building cool data−driven Web sites for your customers, you should be aware that PHP comes with some powerful error−tracking functions which can speed up development time. Take a look at the following example, which contains a deliberate error in the SELECT query string:

```
<html>
<head><basefont face="Arial"></head>
<body>

<?
// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
$connection = pg_connect ("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// generate and execute a query
$query = "SELECTA * FROM addressbook";
$result = pg_query($connection, $query) or die("Error in
query: $query.
" . pg_last_error($connection));

// get the number of rows in the resultset
// this is PG−specific
$rows = pg_num_rows($result);

echo "There are currently $rows records in the database";

// close database connection
pg_close($connection);

?>

</body>
</html>
```

And here's the output:

```
Warning: pg_query() query failed: ERROR: parser: parse error
at or near
"selecta" in /usr/local/apache/htdocs/e.php on line 23 Error
in query:
SELECTA * FROM addressbook. ERROR: parser: parse error at or
near
"selecta"
```

The pg_last_error() function displays the last error returned by PostgreSQL. Turn it on, and you'll find that it can significantly reduce the time you spend fixing bugs.

A Well−Formed Idea

Finally, how about one more example to wrap things up? This next script contains a form which can be used to enter new addresses into the table, together with a form processor that actually creates and executes the INSERT statement.

```
<html>
<head><basefont face="Arial"></head>
<body>
<h2>Address Book</h2>

<?
// form not yet submitted
// display form
if (!$submit)
{
?>
<form action="<? echo $_SERVER['PHP_SELF']; ?>" method="POST">
Name:<br>
<input name="name" type="text" size="50">
<p>
Address:<br>
<textarea name="address" rows="6" cols="40"></textarea>
<p>
Tel:<br>
<input name="tel" type="text" size="10">
<p>
Email:<br>
<input name="email" type="text" size="30">
<p>
<input type="submit" name="submit" value="Add">
</form>
<?
}
else
{
// form submitted
// prepare to insert data

// database access parameters
// alter this as per your configuration
$host = "localhost";
$user = "postgres";
$pass = "postgres";
$db = "test";

// open a connection to the database server
$connection = pg_connect("host=$host dbname=$db user=$user
password=$pass");

if (!$connection)
{
die("Could not open connection to database server");
}

// error checks on form submission go here

// generate and execute a query
$query = "INSERT INTO addressbook VALUES
(nextval('addressbook_id_seq'), '$name', '$address', '$tel',
'$email')";
$result = pg_query($connection, $query) or die("Error in
query:
$query. " . pg_last_error($connection));

echo "Data successfully added.";

// close database connection
pg_close($connection);
}
?>
</body>
</html>
```

As you can see, this script is broken up into two main sections − the appropriate section is displayed depending on whether or not the form has been submitted.

The first part merely displays an HTML form, with fields corresponding to the columns in the "addressbook" table. Once the user enters data into these fields, the same script is called again; this time, the second half will get executed. An SQL query is generated from the data entered into the form, and this query is executed using the pg_query() function you've become familiar with. A success message is displayed once the data has been successfully INSERTed.

If you're familiar with building MySQL−based Web applications with PHP, the procedure above should be familiar to you − the only difference lies in the functions used to communicate with the database server.

Surfing The Web

And that's about all for the moment. In this article, you learned how to use PHP to communicate with that other open−source RDBMS, PostgreSQL, applying PHP's built−in functions to extract and insert data into a PostgreSQL table. In addition to learning a number of different techniques to extract data, you also got a quick tour of how to use transactions with PostgreSQL, and of the error−handling functions available to help you debug your PHP/PostgreSQL scripts.

Needless to say, there's a lot more you can do with PostgreSQL − this article merely covered the basics. If you're interested in learning more about PostgreSQL features, and in the PHP functions that can be used to avail of those features, take a look at the links below.

The PHP manual page for PostgreSQL functions, at http://www.php.net/manual/en/ref.pgsql.php

The PostgreSQL documentation, at http://www.postgresql.org/idocs/

O'Reilly's article on PHP and PostgreSQL, at
http://www.onlamp.com/pub/a/onlamp/2002/01/24/postgresql.html

PHPBeginner's tutorial on PostgreSQL and PHP, at http://www.phpbeginner.com/columns/chris/postgres/

And until next time....stay healthy!

Note: All examples in this article have been tested on Linux/i586 with PostgreSQL 7.1, Apache 1.3.20 and PHP 4.2.0RC2. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!