PHP 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

Build a Web application using PHP to access the Apache Derby database.

- Introduce the PDO_ODBC driver in PHP 5.x
- Set up PHP 5 with the PDO_ODBC extension compiled with Apache Derby support
- Display data from the database in a Web page
- Insert data into the database

Quick facts

Created by Rasmus Lerdorf in mid 1990's as an HTML template language, PHP's ease of extensibility and close connection to databases launched its popularity skywards. The release of PHP 5 in 2004 introduced "real" object-oriented support and focused firmly on integrating XML and Web Service support into the core of the language.

1. Introducing PDO and PDO_ODBC

PHP's initial popularity was built largely on the tight linkage between PHP and MySQL; earlier versions of PHP bundled the MySQL client libraries so that application deployment was painless. However, in the run-up to the release of PHP 5, MySQL AB, the commercial enterprise behind the MySQL database, changed the license of the client libraries to a license incompatible with the PHP license. Until the introduction of the PHP Data Objects (PDO) extension and database-specific drivers that implemented that interface in 2005, there was no common interface for accessing databases in PHP.

PDO is an object oriented interface for database access designed to simplify the task of writing portable PHP applications by providing a common set of database access objects and methods.

PDO_ODBC is a database driver written to the ODBC version 3 specification. This specification is very similar to the Call Level Interface (CLI) library used to access Apache Derby databases by the DB2 Application Development Client, and PDO_ODBC can be compiled directly against the DB2 Application Development Client to enable you to directly access Apache Derby databases without requiring the overhead of an ODBC Driver Manager.

Both PDO and PDO_ODBC were written by Wez Furlong, a developer responsible for core features of PHP such as streams, infrastructure such as the PHP mailing lists, and overseeing the PHP Extension Community Library (PECL) project.

2. Setting up PDO_ODBC support for Apache Derby

Prerequisites:

- Installed DB2 Application Development Client with application development support
- (Linux): Download PHP 5.1.x source from http://www.php.net/downloads.php
- (Linux): Application development tools such as apache-devel, gcc, autoconf, automake, bison, flex, libxml2

Configuring and compiling

On Windows, PDO_ODBC support is compiled into the PHP 5.1.x binary, and is compiled against the Windows ODBC Driver Manager.

On Linux, you can compile PDO_ODBC support against the DB2 Application Development Client libraries by following these steps:

```
bash$ tar xzf php-5.1.tar.gz
bash$ cd php-5.1
bash$ ./configure --with-pdo-odbc=ibm-db2 --enable-cli --disable-cgi \
--with-apxs2=/usr/sbin/apxs
bash$ make && su -c 'make install'
```

This compiles PHP and installs its libraries in the `/usr/local/lib/` directory, with the **php** executable in `/usr/local/bin/`. The installer also tries to set up Apache with the appropriate settings for PHP support.

Test that you have successfully installed PHP 5.1 by issuing the following command:

```
bash$ php -v
```

This should produce the following output:

```
PHP 5.1.0-dev (cli) (built: Jul 10 2005 20:58:07)
Copyright (c) 1997-2005 The PHP Group
Zend Engine v2.1.0-dev, Copyright (c) 1998-2004 Zend Technologies
```

Testing the installation

Test that you have successfully built PDO and PDO_ODBC support into the executable by issuing the following command:

```
bash$ php -m
```

This should produce the output similar to the following:

```
[PHP Modules]
ctype
date
dom
iconv
libxml
pcre
PDO
PDO_ODBC
pdo_sqlite
posix
session
SimpleXML
SPL
SQLite
standard
tokenizer
xml

[Zend Modules]
```

3. Displaying data in a Web page

We'll start by creating a PHP script that simply creates an uncataloged connection to your Apache Derby database and either confirms the connection, or explains why the connection failed.

3a: Creating a function library

Create a new script named **`menu_lib.php`** containing the following code:

```
<?php
function menu_connect($database='MYDB', $hostname='localhost', $port=1527,
$user='lynn', $password='5tuff') {
    // Create the uncataloged connection string
    $DSN = "DRIVER={IBM DB2 ODBC DRIVER};
PROTOCOL=TCPIP;DATABASE=$database;HOSTNAME=$hostname;PORT=$port;UID=$user;PWD
=$password;";
    $dbh = new PDO("odbc:$DSN");
    return $dbh;
}
?>
```

3b: Connecting to the database through PDO

Create a new script named **`connect.php`** containing the following code:

```
<?php

// include the menu_lib.php file within this script
require('menu_lib.php');

// Create the connection, catching any exceptions
try {
    // connect with default parameter values
    $conn = menu_connect();
}
catch (PDOException $e) {
    print "Failed to connect: " . $e->getMessage();
    exit;
}

// Here is where we would do real work; just print connection status
print "Connection succeeded!";

// Clean up the connection
$conn = null;
?>
```

Test the script from the command line to ensure the connection succeeds:

```
bash$ php connect.php
```

We will use the **`connect.php`** script as the basis for the rest of our PHP scripts by fleshing it out with real PHP code and HTML output.

4: Retrieving data from the database

To demonstrate how to retrieve information from an Apache Derby database using PDO, we will search for menu items by preparing and executing an SQL statement (PDOStatement object) with a parameter marker. The search string will be passed from an HTML form.

Create the following HTML search form named **`search.html`**:

```
<html>
<head><title>Search for a menu item</title></head>
<body>
<h1>Search for a menu item</h1>
<form action='search.php' method='POST'>
<label for='sinput'>Please enter the search string:</label>
<input type='text' name='search' id='sinput' size='30'/><br />
<input type='submit'/><input type='reset'/>
</form>
```

Create the following PHP script named **`search.php`**:

```
<?php

// include the menu_lib.php file within this script
require('menu_lib.php');

function menu_footer() {
    print '</body></html>';
}

// display the search form
readfile('search.html');

// assign the user-supplied input to a local variable
if (!isset($_POST['search'])) {
    print "Please supply a search string.";
    print menu_footer();
    exit;
}
// you can filter the input here
$search = preg_match('#^[^\W_]+$#', $_POST['search']);
if (!$search) {
    print "Your search string may only contain letters and numbers.";
    print menu_footer();
    exit;
}
else {
    $search = $_POST['search'];
}

// Create the connection, catching any exceptions
try {
    // connect with default parameter values
    $conn = menu_connect();
}
catch (PDOException $e) {
    print "Failed to connect: " . $e->getMessage();
    exit;
```

```
}

// Use a parameter marker to support variable input
$sql = 'SELECT name, description
        FROM menu.food
        WHERE menu.name LIKE ?';

// Prepare the statement
$stmt = $conn->prepare($sql);

// Execute the statement, passing in an array of input variables
$stmt->execute(array($search));

while ($row = $stmt->fetch()) {
    // Retrieve column by index number
    print "<p>Name: {$row[0]}\n";
    // Retrieve column by column name
    print "Description: {$row['DESCRIPTION']}</p>\n";
}

$stmt = null;
$conn = null;

print menu_footer();

?>
```

Copy the **`search.html`** and **`search.php`** files to your HTML document root, open **`search.html`** in your Web browser, and try searching for some exact matches of menu items.

4a. Search for a substring

Alter the **`search.php`** script so that instead of searching for an exact string, it searches for the substring in the menu item name. (Hint: the SQL wildcard character for multiple characters is `'%'`.)

4b. Search for a case-insensitive substring

Alter the **`search.php`** script so that instead of matching the exact case of the string, it performs a case-insensitive search. (Hints: the UPPER() and LOWER() SQL column functions force values returned from the database to be either uppercase or lowercase, respectively; the `strtoupper()` and `strtolower()` PHP functions force strings to be uppercase or lowercase, respectively.)

4c. Display the last search string in the search text field

Alter the **`search.html`** page so that it displays the last search string entered by a user in the search text field by default.

5. Inserting data into the database from form input

We can insert data into the database in the same way that we retrieved data: preparing and executing an SQL statement through a PDOStatement. This time, we will demonstrate the use of named parameters in the SQL statement.

5a. Create an HTML form for adding new menu items

Create a new HTML form named **`input.html`** consisting of three text fields: itemName, itemDescription, and price.

5b. Create a PHP script that inserts the contents of the HTML form

Create a new PHP script that inserts the values submitted from the HTML form into the database. Filter the input data to ensure that no tainted input values reach the database.

Here is the template SQL for the solution:

```
INSERT INTO menu.food(name, description) VALUES (:name, :description);
INSERT INTO menu.prices (id, price) VALUES (IDENTITY_VAL_LOCAL(), :price);
```

For named parameters in a PDOStatement object, PDO binds values associated with the parameter names passed in an associative array. For example:

```
$stmt->execute(array(':name' => 'Big burger');
```

IDENTITY_VAL_LOCAL() returns the last IDENTITY value generated by an INSERT statement on the same connection to Apache Derby.

5c. Ensuring that the INSERT statements are atomic

To prevent a situation where one of the INSERT statements succeeds, but the other statement fails, you should use a database transaction. PDO starts every connection in auto-commit mode by default, but you can begin a new transaction by calling the `beginTransaction()` method on the PDO connection object. You then have the option of ending the transaction by explicitly committing or rolling back the transaction by calling the `commit()` or `rollBack()` methods on the PDO connection handle, respectively. This returns the database connection to auto-commit mode.

If you begin a transaction, but do not commit the work before the script ends, PHP will automatically roll back the transaction during request clean-up.

Example of a transaction in PDO

```
try {
    // Begin a transaction
    $conn->beginTransaction();

    // Execute the statement, passing in an array of input variables
    $stmt_food->execute(array(':name' => $name,
        ':description' => $description));

    // Execute the statement, passing in an array of input variables
    $stmt_price->execute(array(':price' => $price));
}
catch (PDOException $e) {
    // Uh-oh -- roll back the transaction
    $conn->rollBack();
    print menu_footer();
    exit();
```

```
}

// Commit the transaction
$conn->commit();
```

PHP Syntax Primer

Code blocks

A PHP section begins with `<?php` and ends with `?>`. Any text before or after a PHP section will be output exactly as is to the browser or command line.

Types

PHP is a loosely typed language that will happily interpret the values of variables in whatever way is necessary to provide a reasonably valid comparison. For example, you can compare the integer value `6` with the string value `"7"` to determine which value is greater.

Boolean

A simple TRUE or FALSE value, where FALSE == 0, null, false, empty arrays, or empty strings.

Numeric

PHP supports signed integer and float (decimal) values. As PHP float values lack precision, most PDO drivers return decimal values as strings.

Strings

Strings can be single-quoted, non-interpolated strings; double-quoted, interpolated strings; or HEREDOC notation interpolated strings. For example:

```
<?php
$it = 'nothing';
$looks = "something from {$it}";
$single = 'This prints as $it $looks.';
$heredoc = <<<HEREDOC
  All of this whitespace is preserved, and HEREDOC
  notation supports interpolated strings, so you can
  get $looks if you try really hard.
HEREDOC;
?>
```

Arrays

Arrays are an extremely flexible data type that consist of a set of keys that map to PHP values. In PHP, arrays enable you to access data by keys that are either integers or strings. By default, an array is a zero-indexed structure that increments the key by one for every new item that is pushed onto the array; however, you can explicitly assign keys when you populate the array. You can also create arrays of arrays if you require more complex structures.

PDO returns rows as arrays that, by default, are indexed by both column number and name.

```
<?php
$array = new array();
$array[] = "item one";
$array[] = "item two";
$array[64] = "when I'm ";
$array['Beatles'] = "Rock 'n' Roll";
```

```
?>
```

Objects

PHP supports objects with inheritance and encapsulation and all kinds of complexity, but for the purposes of this tutorial here's what you need to know to use PDO (an object-oriented interface):

Constructor

PHP objects use a constructor to create a new instance of a class. The constructor is invoked by the `new` operator, and you are responsible for passing whatever arguments the constructor requires to create the class. For example, the constructor syntax for a PDO object is:

```
<?php
$conn = new PDO($DSN, $username, $password, $driver_arguments);
?>
```

Method invocation

To invoke a dynamic method on an object, use the `->` operator to bind the method name to the object name. To invoke a static method on a class, use the `::` operator. For example:

```
<?php
$stmt = $conn->prepare('SELECT name FROM menu.food');
$stmt->execute();
?>
```

Control structures

Code blocks in PHP are delimited by `{ }` characters.

if, while, do

The `if` and `while` control structures test a particular condition and execute a code block if the condition is true. The `do` control structure always executes its code block once, and then loops again if the closing `while` test is true.

```
<?php
$result = $stmt->execute();

if (!$result) {
  print "DEBUG: Statement failed: {$stmt->errorInfo();}\n";
}

while ($row = $stmt->fetch()) {
  print "Name: {$row['NAME']}\n";
}
?>
```

for

The `for` loop is very similar to C's `for` control structure. `for` syntax contains three separate expressions:

1. the first is executed only once, at the beginning of the `for` loop -- this is typically used to initialize a local variable

2. the second is evaluated before every iteration of the loop, and which terminates the `for` control structure if the value is FALSE
3. the third is evaluated at the end of every iteration of the loop, and is typically used to increment or decrement the local variable

foreach

The `foreach` control structure iterates over every element of an array or object that implements an iterator interface. This is an extremely handy way of handling dynamic data in your application; for example, if your data structure changes and you need to print every column of every row that is returned, you could use the following syntax:

```
<?php
$row = $stmt->fetch(PDO_FETCH_ASSOC);
foreach ($row as $key => $value) {
  print "Column name: [$key]. Column value: [$value]\n";
}
?>
```

Resources

- PDO documentation: http://www.php.net/PDO
- PHP Web site: http://www.php.net
- PHP Cheat Sheet: http://www.ilovejackdaniels.com/php/php-cheat-sheet/
- PHP Extension and Application Repository (PEAR): http://pear.php.net
- PHP Extension Community Library (PECL): http://pecl.php.net
- Zend Core for IBM: http://www.ibm.com/software/data/info/zendcore/