CHAPTER 31

Introducing PDO

While all mainstream databases adhere to the SQL standard, albeit to varying degrees, the interfaces that programmers depend upon to interact with the database can vary greatly (even if the queries are largely the same). Therefore, applications are almost invariably bound to a particular database, forcing users to also install and maintain the required database if they don't already own it or, alternatively, to choose another, possibly less capable, solution that is compatible with their present environment. For instance, suppose your organization requires an application that runs exclusively on Oracle, but your organization is standardized on MySQL. Are you prepared to invest the considerable resources required to obtain the necessary level of Oracle knowledge required to run in a mission-critical environment and then deploy and maintain that database throughout the application's lifetime?

To resolve such dilemmas, enterprising programmers began developing database abstraction layers, which serve to decouple the application logic from that used to communicate with the database. By passing all database-related commands through this generalized interface, it became possible for an application to use one of several database solutions, provided the database supported the features required by the application, and the abstraction layer offered a driver compatible with that database. A graphical depiction of this process is found in Figure 31-1.

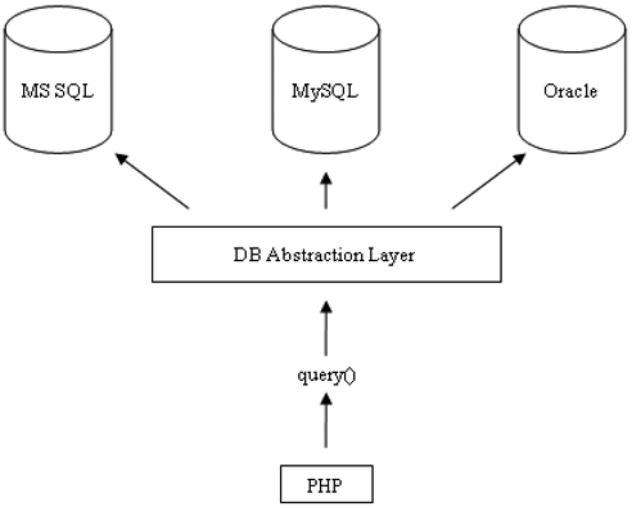

Figure 31-1. *Using a database abstraction layer to decouple the application and data layers*

It's likely you've heard of some of the more widespread implementations, a few of which are listed here:

- **MDB2:** MDB2 is a database abstraction layer written in PHP and available as a PEAR package (see Chapter 11 for more information about PEAR). It presently supports FrontBase, InterBase, Informix, Mini SQL, MySQL, Oracle, ODBC, PostgreSQL, SQLite, and Sybase.

- **JDBC:** As its name implies, the Java Database Connectivity (JDBC) standard allows Java programs to interact with any database for which a JDBC driver is available. Among others, this includes Microsoft SQL Server, MySQL, Oracle, and PostgreSQL.

- **ODBC:** The Open Database Connectivity (ODBC) interface is one of the most widespread abstraction implementations in use today, supported by a wide range of applications and languages, PHP included. ODBC drivers are offered by all mainstream databases, including those referenced in the above JDBC introduction.

- **Perl DBI:** The Perl Database Interface module is Perl's standardized means for communicating with a database, and was the inspiration behind PHP's DB package.

Because PHP offers MDB2 and supports ODBC, it seems that your database abstraction needs are resolved when developing PHP-driven applications, right? While these (and many other) solutions are readily available, an even better solution has been in development for some time, and has been officially released with PHP 5.1. This solution is known as the PHP Data Objects (PDO) abstraction layer.

Another Database Abstraction Layer?

As PDO came to fruition, it was met with no shortage of rumblings from developers either involved in the development of alternative database abstraction layers, or perhaps too focused on PDO's database abstraction features rather than the entire array of capabilities it offers. Indeed, PDO serves as an ideal replacement for the PEAR::DB package and similar solutions. However, PDO is actually much more than just a database abstraction layer, offering:

Coding consistency: Because PHP's various database extensions are written by a host of different contributors, there is no coding consistency despite the fact that all of these extensions offer basically the same features. PDO removes this inconsistency by offering a single unified interface that is used no matter the database. Furthermore, the extension is broken into two distinct components: the PDO core contains most of the PHP-specific code, leaving the various drivers to focus solely on the data. Also, the PDO developers took advantage of considerable knowledge and experience while building the database extensions over the past few years, capitalizing upon what was successful and being careful to omit what was not. Although a few inconsistencies remain, by and large the database features are nicely abstracted. Work on version 2 is currently under way, with a good chance any remaining inconsistencies will be ironed out.

Flexibility: Because PDO loads the necessary database driver at run time, there's no need to reconfigure and recompile PHP every time a different database is used. For instance, if your database needs suddenly switch from Oracle to MySQL, just load the `PDO_MYSQL` driver and go (more about how to do this later).

Object-oriented features: PDO takes advantage of PHP 5's object-oriented features, resulting in more powerful and efficient database communication.

Performance: PDO is written in C and compiled into PHP, which, all other things being equal, provides a considerable performance increase over solutions written in PHP.

Given such advantages, what's not to like? This chapter serves to fully acquaint you with PDO and the myriad of features it has to offer.

Using PDO

PDO bears a striking resemblance to all of the database extensions long supported by PHP. Therefore, for those of you who have used PHP in conjunction with a database, the material presented in this section should be quite familiar. As mentioned, PDO was built with the best features of the preceding database extensions in mind, so it makes sense that you'll see a marked similarity in its methods.

This section commences with a quick overview of the PDO installation process, and follows with a summary of its presently supported database servers. For the purposes of the examples found throughout the remainder of this chapter, the following MySQL table is used:

```
CREATE TABLE products (
   id SMALLINT NOT NULL AUTO_INCREMENT,
   sku CHAR(8) NOT NULL,
   title VARCHAR(35) NOT NULL,
   PRIMARY KEY(id)
);
```

The table has been populated with the products listed in Table 31-1.

Table 31-1. *Sample Product Data*

id	sku	title
1	ZP457321	Painless Aftershave
2	TY232278	AquaSmooth Toothpaste
3	PO988932	HeadsFree Shampoo
4	KL334899	WhiskerWrecker Razors

Installing PDO

As mentioned, PDO comes packaged with PHP 5.1 and newer by default, so if you're running this version, you do not need to take any additional steps. However, because PDO remains under active development, you may want instead to configure it as a

shared module. Consult the PHP documentation for more information about this matter.

If you're using a version older than 5.1, you can still use PDO by installing PECL; however, because PDO takes full advantage of PHP 5's object-oriented features, it's not possible to use it in conjunction with any pre-5.0 release. Therefore, to install PDO using PHP 5.0.X on Linux, execute the following:

```
%>pecl install pdo
```

Next, enable PDO by adding the following line to your `php.ini` file:

```
extension=pdo.so
```

Finally, restart Apache for the `php.ini` changes to take effect.

If you're running PHP 5.1 or newer on Linux, PDO is bundled by default, although you should specifically configure it as a shared extension in order to facilitate driver updates. A typical configuration line looks like this:

```
%>./configure --with-zlib --enable-pdo=shared --with-pdo-mysql=shared --with-
mysql
```

If you're using PHP 5.1 or newer on the Windows platform, all you need to do is add references to the PDO and driver extensions within the `php.ini` file. For example, to enable support for MySQL, add the following lines to the Windows Extensions section:

```
extension=php_pdo.dll
extension=php_pdo_mysql.dll
```

Again, don't forget to restart Apache in order for the `php.ini` changes to take effect.

PDO's Database Options

As of the time of this writing, PDO supported nine databases, in addition to any database accessible via FreeTDS and ODBC, including the following:

Firebird: Accessible via the `FIREBIRD` driver.

FreeTDS: Accessible via the `DBLIB` driver. FreeTDS is not a database but rather a set of Unix libraries that enables Unix-based programs to talk to Microsoft SQL Server and Sybase.

IBM DB2: Accessible via the `ODBC` driver.

Interbase 6: Accessible via the `FIREBIRD` driver.

Microsoft SQL Server: Accessible via the `ODBC` driver.

MySQL: Accessible via the `MYSQL` driver.

ODBC 3: Accessible via the `ODBC` driver. ODBC is not a database per se but it enables PDO to be used in conjunction with any ODBC-compatible database not found in this list.

Oracle: Accessible via the `OCI` driver. Oracle versions 8 through 11*g* are supported.

PostgreSQL: Accessible via the `PGSQL` driver.

SQLite 2.X and 3.X: Accessible via the `SQLITE` driver.

Sybase: Accessible via the `ODBC` driver.

■ **Tip** You can determine which PDO drivers are available to your environment either by loading `phpinfo()` into the browser and reviewing the list provided under the PDO section header, or by executing the `pdo_drivers()` function like so: `<?php print_r(pdo_drivers()); ?>`.

Connecting to a Database Server and Selecting a Database

Before interacting with a database using PDO, you'll need to establish a server connection and select a database. This is accomplished through PDO's constructor. Its prototype follows:

```
PDO PDO::__construct(string DSN [, string username [, string password
                    [, array driver_opts]]])
```

The Data Source Name (`DSN`) parameter consists of two items: the desired database driver name, and any necessary database connection variables such as the hostname, port, and database name. The `username` and `password` parameters specify the username and password used to connect to the database, respectively. Finally, the `driver_opts` array specifies any additional options that might be required or desired for the connection. A list of available options is offered at the conclusion of this section.

You're free to invoke the constructor in a number of fashions. These different methods are introduced next.

Embedding the Parameters into the Constructor

The easiest way to connect to a database is by simply passing the connection parameters into the constructor. For instance, the constructor can be invoked like this (MySQL-specific):

```
$dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");
```

Placing the Parameters in a File

PDO utilizes PHP's streams feature, opening the option to place the DSN string in a separate file that resides either locally or remotely, and reference it within the constructor like so:

```
$dbh = new PDO("uri:file://usr/local/mysql.dsn");
```

Make sure the file is owned by the same user responsible for executing the PHP script and possesses the necessary privileges.

Referring to the php.ini File

It's also possible to maintain the DSN information in the `php.ini` file by assigning it to a configuration parameter named `pdo.dsn.aliasname`, where `aliasname` is a chosen alias for the DSN that is subsequently supplied to the constructor. For instance, the following example aliases the DSN to `mysqlpdo`:

```
[PDO]
pdo.dsn.mysqlpdo = "mysql:dbname=chp31;host=localhost"
```

The alias can subsequently be called by the PDO constructor like so:

```
$dbh = new PDO("mysqlpdo", "webuser", "secret");
```

Unlike the previous method, this method doesn't allow for the username and password to be included in the DSN.

Using PDO's Connection-Related Options

There are several connection-related options for PDO that you might consider tweaking by passing them into the `driver_opts` array. These options are enumerated here:

- PDO_ATTR_AUTOCOMMIT: This option determines whether PDO will commit each query as it's executed, or will wait for the commit() method to be executed before effecting the changes.

- PDO_ATTR_CASE: You can force PDO to convert the retrieved column character casing to all uppercase, to convert it to all lowercase, or to use the columns exactly as they're found in the database. Such control is accomplished by setting this option to one of three values: PDO_CASE_UPPER, PDO_CASE_LOWER, or PDO_CASE_NATURAL, respectively.

- PDO_ATTR_EMULATE_PREPARES: Enabling this option makes it possible for prepared statements to take advantage of MySQL's query cache.

- PDO_ATTR_ERRMODE: PDO supports three error-reporting modes, PDO_ERRMODE_EXCEPTION, PDO_ERRMODE_SILENT, and PDO_ERRMODE_WARNING. These modes determine what circumstances cause PDO to report an error. Set this option to one of these three values to change the default behavior, which is PDO_ERRMODE_EXCEPTION. This feature is discussed in further detail in the later section "Handling Errors."

- PDO_ATTR_ORACLE_NULLS: When set to TRUE, this attribute causes empty strings to be converted to NULL when retrieved. By default this is set to FALSE.

- PDO_ATTR_PERSISTENT: This option determines whether the connection is persistent. By default this is set to FALSE.

- PDO_ATTR_PREFETCH: Prefetching is a database feature that retrieves several rows even if the client is requesting one row at a time, the reasoning being that if the client requests one row, he's likely going to want others. Doing so decreases the number of database requests and therefore increases efficiency. This option sets the prefetch size, in kilobytes, for drivers that support this feature.

- PDO_ATTR_TIMEOUT: This option sets the number of seconds to wait before timing out. MySQL currently does not support this option.

- PDO_DEFAULT_FETCH_MODE: You can use this option to set the default fetching mode (associative arrays, indexed arrays, or objects), thereby saving some typing if you consistently prefer one particular method.

Four attributes exist for helping you learn more about the client, server, and connection status. The attribute values can be retrieved using the method `getAttribute()`, introduced in the later section "Getting and Setting Attributes."

- `PDO_ATTR_SERVER_INFO`: Contains database-specific server information. In the case of MySQL, it retrieves data pertinent to server uptime, total queries, the average number of queries executed per second, and other important information.
- `PDO_ATTR_SERVER_VERSION`: Contains information pertinent to the database server's version number.
- `PDO_ATTR_CLIENT_VERSION`: Contains information pertinent to the database client's version number.
- `PDO_ATTR_CONNECTION_STATUS`: Contains database-specific information about the connection status. For instance, after a successful connection when using MySQL, the attribute contains "localhost via TCP/IP," while on PostgreSQL it contains "Connection OK; waiting to send."

Handling Connection Errors

In the case of a connection error, the script immediately terminates unless the returned `PDOException` object is properly caught. Of course, you can easily do so using the exception-handling syntax first introduced in Chapter 8. The following example shows you how to catch the exception in case of a connection problem:

```
<?php
    try {
       $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");
    } catch (PDOException $exception) {
       echo "Connection error: " . $exception->getMessage();
    }
?>
```

Once a connection has been established, it's time to begin using it. This is the topic of the rest of this chapter.

Handling Errors

PDO offers three error modes, allowing you to tweak the way in which errors are handled by the extension:

PDO_ERRMODE_EXCEPTION: Throws an exception using the PDOException class, which immediately halts script execution and offers information pertinent to the problem.

PDO_ERRMODE_SILENT: Does nothing if an error occurs, leaving it to the developer to both check for errors and determine what to do with them. This is the default setting.

PDO_ERRMODE_WARNING: Produces a PHP E_WARNING message if a PDO-related error occurs.

To set the error mode, just use the setAttribute() method (formally introduced in the later section "Getting and Setting Attributes"), like so:

```
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

There are also two methods available for retrieving error information. Both are introduced next.

Retrieving SQL Error Codes

The SQL standard offers a list of diagnostic codes used to signal the outcome of SQL queries, known as *SQLSTATE codes*. Execute a Web search for *SQLSTATE codes* to produce a list of these codes and their meanings. The errorCode() method is used to return this standard SQLSTATE code, which you might choose to store for logging purposes or even for producing your own custom error messages. Its prototype follows:

```
int PDOStatement::errorCode()
```

For instance, the following script attempts to insert a new product but mistakenly refers to the singular version of the products table:

```
<?php
    try {
        $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");
    } catch (PDOException $exception) {
        printf("Connection error: %s", $exception->getMessage());
    }
```

```
    $query = "INSERT INTO products(id, sku, title)
              VALUES(NULL, 'SS873221', 'Surly Soap') ";

    $dbh->exec($query);

    echo $dbh->errorCode();
?>
```

This should produce the code `42S02`, which corresponds to MySQL's nonexistent table message. Of course, this message alone means little, so you might be interested in the `errorInfo()` method, introduced next.

Retrieving SQL Error Messages

The `errorInfo()` method produces an array consisting of error information pertinent to the most recently executed database operation. Its prototype follows:

```
array PDOStatement::errorInfo()
```

This array consists of three values, each referenced by a numerically indexed value between `0` and `2`:

`0`: Stores the SQLSTATE code as defined in the SQL standard

`1`: Stores the database driver–specific error code

`2`: Stores the database driver–specific error message

The following script demonstrates `errorInfo()`, causing it to output error information pertinent to a missing table (in this case, the programmer mistakenly uses the singular form of the existing `products` table):

```
<?php
    try {
       $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");
    } catch (PDOException $exception) {
        printf("Failed to obtain database handle %s", $exception->getMessage());
    }

    $query = "INSERT INTO product(id, sku, title)
              VALUES(NULL, 'SS873221', 'Surly Soap') ";
```

```
    $dbh->exec($query);

    print_r($dbh->errorInfo());

?>
```

Presuming the `product` table doesn't exist, the following output is produced (formatted for readability):

```
Array (
[0] => 42S02
[1] => 1146
[2] => Table 'chp31.product' doesn't exist )
```

Getting and Setting Attributes

Quite a few attributes are available for tweaking PDO's behavior, the most complete list of which is made available in the PHP documentation. As these attributes were still in a state of flux at the time of writing, it makes the most sense to point you to the documentation rather than provide what would surely be an incomplete or incorrect summary; therefore, see `http://www.php.net/pdo` for the latest information.

Two methods are available for both setting and retrieving the values of these attributes. Both are introduced next.

Retrieving Attributes

The `getAttribute()` method retrieves the value of the attribute specified by `attribute`. Its prototype looks like this:

```
mixed PDOStatement::getAttribute(int attribute)
```

An example follows:

```
$dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");
echo $dbh->getAttribute(PDO_ATTR_CONNECTION_STATUS);
```

On my server this returns:

```
localhost via TCP/IP
```

Setting Attributes

The `setAttribute()` method assigns the value specified by `value` to the attribute specified by `attribute`. Its prototype looks like this:

```
boolean PDOStatement::setAttribute(int attribute, mixed value)
```

For example, to set PDO's error mode, you'd need to set `PDO_ATTR_ERRMODE` like so:

```
$dbh->setAttribute(PDO_ATTR_ERRMODE, PDO_ERRMODE_EXCEPTION);
```

Executing Queries

PDO offers several methods for executing queries, with each attuned to executing a specific query type in the most efficient way possible. The following list breaks down each query type:

Executing a query with no result set: When executing queries such as `INSERT`, `UPDATE`, and `DELETE`, no result set is returned. In such cases, the `exec()` method returns the number of rows affected by the query.

Executing a query a single time: When executing a query that returns a result set, or when the number of affected rows is irrelevant, you should use the `query()` method.

Executing a query multiple times: Although it's possible to execute a query numerous times using a `while` loop and the `query()` method, passing in different column values for each iteration, doing so is more efficient using a *prepared statement*.

Adding, Modifying, and Deleting Table Data

Chances are your applications will provide some way to add, modify, and delete data. To do this you would pass a query to the `exec()` method, which executes a query and returns the number of rows affected by it. Its prototype follows:

```
int PDO::exec(string query)
```

Consider the following example:

```
$query = "UPDATE products SET title='Painful Aftershave' WHERE sku='ZP457321'";
$affected = $dbh->exec($query);
echo "Total rows affected: $affected";
```

Based on the sample data introduced earlier in the chapter, this example would return the following:

```
Total rows affected: 1
```

Note that this method shouldn't be used in conjunction with `SELECT` queries; instead, the `query()` method, which is introduced next, should be used for these purposes.

Selecting Table Data

The `query()` method executes a query, returning the data as a `PDOStatement` object. Its prototype follows:

```
PDOStatement query(string query)
```

An example follows:

```
$query = "SELECT sku, title FROM products ORDER BY id";
foreach ($dbh->query($query) AS $row) {
   $sku = $row['sku'];
   $title = $row['title'];
   printf("Product: %s (%s) <br />", $title, $sku);
}
```

Based on the sample data, this example produces the following:

```
Product: AquaSmooth Toothpaste (TY232278)
Product: HeadsFree Shampoo (PO988932)
Product: Painless Aftershave (ZP457321)
Product: WhiskerWrecker Razors (KL334899)
```

■ **Tip** If you use `query()` and would like to learn more about the total number of rows affected, use the `rowCount()` method.

Introducing Prepared Statements

Each time a query is sent to the MySQL server, the query syntax must be parsed to ensure a proper structure and to ready it for execution. This is a necessary step of the process, and it does incur some overhead. Doing so once is a necessity, but what if you're repeatedly executing the same query, only changing the column values, as you might do when batch-inserting several rows? A prepared statement eliminates this additional overhead by caching the query syntax and execution process to the server, and traveling to and from the client only to retrieve the changing column value(s).

PDO offers prepared-statement capabilities for those databases supporting this feature. Because MySQL supports prepared statements, you're free to take advantage of this feature. Prepared statements are accomplished using two methods, `prepare()`, which is responsible for readying the query for execution, and `execute()`, which is used to repeatedly execute the query using a provided set of column parameters. These parameters can be provided to `execute()` either explicitly by passing them into the method as an array, or by using bound parameters assigned using the `bindParam()` method. All three of these methods are introduced next.

Using Prepared Statements

The `prepare()` method is responsible for readying a query for execution. Its prototype follows:

```
PDOStatement PDO::prepare(string query [, array driver_options])
```

A query intended for use as a prepared statement looks a bit different from those you might be used to, because placeholders must be used instead of actual column values for those that will change across execution iterations. Two syntax variations are supported, *named parameters* and *question mark parameters*. For example, a query using the former variation might look like this:

```
INSERT INTO products SET sku = :sku, name = :name;
```

The same query using the latter variation would look like this:

```
INSERT INTO products SET sku = ?, name = ?;
```

The variation you choose is entirely a matter of preference, although perhaps using named parameters is a tad more explicit. For this reason, this variation is used in relevant examples. To begin, the following example uses `prepare()` to ready a query for iterative execution:

```
// Connect to the database
$dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");

$query = "INSERT INTO products SET sku = :sku, name = :name";
$stmt = $dbh->prepare($query);
```

Once the query is prepared, it must be executed. This is accomplished by the `execute()` method, introduced next.

In addition to the query, you can also pass along database driver–specific options via the `driver_options` parameter. See the PHP manual for more information about these options.

Executing a Prepared Query

The `execute()` method is responsible for executing a prepared query. Its prototype follows:

```
boolean PDOStatement::execute([array input_parameters])
```

This method requires the input parameters that should be substituted with each iterative execution. This is accomplished in one of two ways: either pass the values into the method as an array, or bind the values to their respective variable name or positional offset in the query using the `bindParam()` method. The first option is covered next, and the second option is covered in the upcoming introduction to `bindParam()`.

The following example shows how a statement is prepared and repeatedly executed by `execute()`, each time with different parameters:

```
<?php
    // Connect to the database server
    $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");

    // Create and prepare the query
    $query = "INSERT INTO product SET sku = :sku, title = :title";
    $stmt = $dbh->prepare($query);

    // Execute the query
    $stmt->execute(array(':sku' => 'MN873213', ':title' => 'Minty Mouthwash'));

    // Execute again
    $stmt->execute(array(':sku' => 'AB223234', ':title' => 'Lovable Lipstick'));
?>
```

This example is revisited next, where you'll learn an alternative means for passing along query parameters using the `bindParam()` method.

Binding Parameters

You might have noted in the earlier introduction to the `execute()` method that the `input_parameters` parameter was optional. This is convenient because if you need to pass along numerous variables, providing an array in this manner can quickly become unwieldy. So what's the alternative? The `bindParam()` method. Its prototype follows:

```
boolean PDOStatement::bindParam(mixed parameter, mixed &variable [, int datatype
                                [, int length [, mixed driver_options]]])
```

When using named parameters, `parameter` is the name of the column value placeholder specified in the prepared statement using the syntax `:title`. When using question mark parameters, `parameter` is the index offset of the column value placeholder as located in the query. The `variable` parameter stores the value to be assigned to the placeholder. It's depicted as passed by reference because when using this method in conjunction with a prepared stored procedure, the value could be changed according to some action in the stored procedure. This feature won't be demonstrated in this section; however, after you read Chapter 32, the process should be fairly obvious. The optional `datatype` parameter explicitly sets the parameter datatype, and can be any of the following values:

- `PDO_PARAM_BOOL`: SQL `BOOLEAN` datatype
- `PDO_PARAM_INPUT_OUTPUT`: Used when the parameter is passed into a stored procedure and therefore could be changed after the procedure executes
- `PDO_PARAM_INT`: SQL `INTEGER` datatype
- `PDO_PARAM_NULL`: SQL `NULL` datatype
- `PDO_PARAM_LOB`: SQL large object datatype (not supported by MySQL)
- `PDO_PARAM_STMT`: `PDOStatement` object type; presently not operational
- `PDO_PARAM_STR`: SQL string datatypes

The optional `length` parameter specifies the datatype's length. It's only required when assigning it the `PDO_PARAM_INPUT_OUTPUT` datatype. Finally, the `driver_options` parameter is used to pass along any driver-specific options.

The following example revisits the previous example, this time using `bindParam()` to assign the column values:

```
<?php

    // Connect to the database server
    $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");

    // Create and prepare the query
    $query = "INSERT INTO products SET sku = :sku, title = :title";
    $stmt = $dbh->prepare($query);

    $sku = 'MN873213';
    $title = 'Minty Mouthwash';

    // Bind the parameters
    $stmt->bindParam(':sku', $sku);
    $stmt->bindParam(':title', $title);

    // Execute the query
    $stmt->execute();

    // Bind the parameters
    $stmt->bindParam(':sku', 'AB223234');
    $stmt->bindParam(':title', 'Lovable Lipstick');

    // Execute again
    $stmt->execute();
?>
```

If question mark parameters were used, the statement would look like this:

```
$query = "INSERT INTO products SET sku = ?, title = ?";
```

Therefore, the corresponding `bindParam()` calls would look like this:

```
$stmt->bindParam(1, 'MN873213');
$stmt->bindParam(2, 'Minty Mouthwash');
. . .
$stmt->bindParam(1, 'AB223234');
$stmt->bindParam(2, 'Lovable Lipstick');
```

Retrieving Data

PDO's data-retrieval methodology is quite similar to that found in any of the other database extensions. In fact, if you've used any of these extensions in the past, you'll be quite comfortable with PDO's five relevant methods. These methods are introduced in this section and are accompanied by examples where practical.

All of the methods introduced in this section are part of the `PDOStatement` class, which is returned by several of the methods introduced in the previous section.

Returning the Number of Retrieved Columns

The `columnCount()` method returns the total number of columns returned in the result set. Its prototype follows:

```
integer PDOStatement::columnCount()
```

An example follows:

```
// Execute the query
$query = "SELECT sku, title FROM products ORDER BY title";
$result = $dbh->query($query);

// Report how many columns were returned
printf("There were %d product fields returned.", $result->columnCount());
```

Sample output follows:

```
There were 2 product fields returned.
```

Retrieving the Next Row in the Result Set

The `fetch()` method returns the next row from the result set, or `FALSE` if the end of the result set has been reached. Its prototype looks like this:

```
mixed PDOStatement::fetch([int fetch_style [, int cursor_orientation
                          [, int cursor_offset]]])
```

The way in which each column in the row is referenced depends upon how the `fetch_style` parameter is set. Seven settings are available:

- PDO_FETCH_ASSOC: Prompts fetch() to retrieve an array of values indexed by the column name.
- PDO_FETCH_BOTH: Prompts fetch() to retrieve an array of values indexed by both the column name and the numerical offset of the column in the row (beginning with 0). This is the default.
- PDO_FETCH_BOUND: Prompts fetch() to return TRUE and instead assign the retrieved column values to the corresponding variables as specified in the bindParam() method. See the later section "Setting Bound Columns" for more information about bound columns.
- PDO_FETCH_INTO: Retrieves the column values into an existing instance of a class. The respective class attributes must match the column values, and must be assigned as public scope. Alternatively, the __get() and __set() methods must be overloaded to facilitate assignment as described in Chapter 7.
- PDO_FETCH_LAZY: Creates associative and indexed arrays, in addition to an object containing the column properties, allowing you to use whichever of the three interfaces you choose.
- PDO_FETCH_NUM: Prompts fetch() to retrieve an array of values indexed by the numerical offset of the column in the row (beginning with 0).
- PDO_FETCH_OBJ: Prompts fetch() to create an object consisting of properties matching each of the retrieved column names.

The cursor_orientation parameter determines which row is retrieved if the object is a scrollable cursor. The cursor_offset parameter is an integer value representing the offset of the row to be retrieved relative to the present cursor position.

The following example retrieves all of the products from the database, ordering the results by title:

```
<?php

    // Connect to the database server
    $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");
```

```
    // Execute the query
    $stmt = $dbh->query("SELECT sku, title FROM products ORDER BY title");

    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        $sku = $row['sku'];
        $title = $row['title'];
        printf("Product: %s (%s) <br />", $title, $sku);
    }

?>
```

Sample output follows:

```
Product: AquaSmooth Toothpaste (TY232278)
Product: HeadsFree Shampoo (PO988932)
Product: Painless Aftershave (ZP457321)
Product: WhiskerWrecker Razors (KL334899)
```

Simultaneously Returning All Result Set Rows

The `fetchAll()` method works in a fashion quite similar to `fetch()`, except that a single call to it results in all rows in the result set being retrieved and assigned to the returned array. Its prototype follows:

```
array PDOStatement::fetchAll([int fetch_style])
```

The way in which the retrieved columns are referenced depends upon how the optional `fetch_style` parameter is set, which by default is set to `PDO_FETCH_BOTH`. See the preceding section regarding the `fetch()` method for a complete listing of all available `fetch_style` values.

The following example produces the same result as the example provided in the `fetch()` introduction, but this time depends on `fetchAll()` to ready the data for output:

```
    // Execute the query
    $stmt = $dbh->query("SELECT sku, title FROM products ORDER BY title");

    // Retrieve all of the rows
    $rows = $stmt->fetchAll();
```

```
// Output the rows
foreach ($rows as $row) {
    $sku = $row[0];
    $title = $row[1];
    printf("Product: %s (%s) <br />", $title, $sku);
}
```

Sample output follows:

```
Product: AquaSmooth Toothpaste (TY232278)
Product: HeadsFree Shampoo (PO988932)
Product: Painless Aftershave (ZP457321)
Product: WhiskerWrecker Razors (KL334899)
```

As to whether you choose to use `fetchAll()` over `fetch()`, it seems largely a matter of convenience. However, keep in mind that using `fetchAll()` in conjunction with particularly large result sets could place a large burden on the system, in terms of both database server resources and network bandwidth.

Fetching a Single Column

The `fetchColumn()` method returns a single column value located in the next row of the result set. Its prototype follows:

```
string PDOStatement::fetchColumn([int column_number])
```

The column reference, assigned to `column_number`, must be specified according to its numerical offset in the row, which begins at 0. If no value is set, `fetchColumn()` returns the value found in the first column. Oddly enough, it's impossible to retrieve more than one column in the same row using this method, as each call moves the row pointer to the next position; therefore, consider using `fetch()` should you need to do so.

The following example both demonstrates `fetchColumn()` and shows how subsequent calls to the method move the row pointer:

```
// Execute the query
$result = $dbh->query("SELECT sku, title FROM products ORDER BY title");

// Fetch the first row first column
$sku = $result->fetchColumn(0);
```

```
// Fetch the second row second column
$title =  $result->fetchColumn(1);

// Output the data.
echo "Product: $title ($sku)";
```

The resulting output follows. Note that the product title and SKU don't correspond to the correct values as provided in the sample table because, as mentioned, the row pointer advances with each call to `fetchColumn()`; therefore, be wary when using this method.

```
Product: AquaSmooth Toothpaste (PO988932)
```

Setting Bound Columns

In the previous section you learned how to set the `fetch_style` parameter in the `fetch()` and `fetchAll()` methods to control how the result set columns will be made available to your script. You were probably intrigued by the `PDO_FETCH_BOUND` setting, because it seems to enable you to avoid an additional step altogether when retrieving column values and instead just assign them automatically to predefined variables. Indeed this is the case, and it's accomplished using the `bindColumn()` method, introduced next.

The `bindColumn()` method is used to match a column name to a desired variable name, which, upon each row retrieval, will result in the corresponding column value being automatically assigned to the variable. Its prototype follows:

```
boolean PDOStatement::bindColumn(mixed column, mixed &param [, int type
                                 [, int maxlen [, mixed driver_options]]])
```

The `column` parameter specifies the column offset in the row, whereas the `&param` parameter defines the name of the corresponding variable. You can set constraints on the variable value by defining its type using the `type` parameter, and limiting its `length` using the `maxlen` parameter. Seven `type` parameter values are supported. See the earlier introduction to `bindParam()` for a complete listing.

The following example selects the `sku` and `title` columns from the `products` table where `id` equals 1, and binds the results according to a numerical offset and associative mapping, respectively:

```
<?php
    // Connect to the database server
    $dbh = new PDO("mysql:host=localhost;dbname=chp31", "webuser", "secret");

    // Create and prepare the query
    $query = "SELECT sku, title FROM products WHERE id=1";
    $stmt = $dbh->prepare($query);
    $stmt->execute();

    // Bind according to column offset
    $stmt->bindColumn(1, $sku);

    // Bind according to column title
    $stmt->bindColumn('title', $title);

    // Fetch the row
    $row = $stmt->fetch(PDO::FETCH_BOUND);

    // Output the data
    printf("Product: %s (%s)", $title, $sku);
?>
```

It returns the following:

```
Painless Aftershave (ZP457321)
```

Working with Transactions

PDO offers transaction support for those databases capable of executing transactions. Three PDO methods facilitate transactional tasks, `beginTransaction()`, `commit()`, and `rollback()`. Because Chapter 37 is devoted to a complete introduction to transactions, no examples are offered here; instead, brief introductions to these three methods are offered.

Beginning a Transaction

The `beginTransaction()` method disables autocommit mode, meaning that any database changes will not take effect until the `commit()` method is executed. Its prototype follows:

```
boolean PDO::beginTransaction()
```

Once either `commit()` or `rollback()` is executed, autocommit mode will automatically be enabled again.

Committing a Transaction

The `commit()` method commits the transaction. Its prototype follows:

```
boolean PDO::commit()
```

Rolling Back a Transaction

The `rollback()` method negates any database changes made since `beginTransaction()` was executed. Its prototype follows:

```
boolean PDO::rollback()
```

Summary

PDO offers users a powerful means for consolidating otherwise incongruous database commands, allowing for an almost trivial means for migrating an application from one database solution to another. Furthermore, it encourages greater productivity among the PHP language developers due to the separation of language-specific and database-specific features. If your clients expect an application that allows them to use a preferred database, you're encouraged to keep an eye on this new extension as it matures.