PDF_utf8_to_utf16 PDO::beginTransaction»

Last updated: Tue, 22 Sep 2015

CXVII. PDO Functions

Introduction

The PHP Data Objects (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP. Each database driver that implements the PDO interface can expose database-specific features as regular extension functions. Note that you cannot perform any database functions using the PDO extension by itself; you must use a database-specific PDO driver to access a database server.

PDO provides a *data-access* abstraction layer, which means that, regardless of which database you're using, you use the same functions to issue queries and fetch data. PDO does *not* provide a *database* abstraction; it doesn't rewrite SQL or emulate missing features. You should use a full-blown abstraction layer if you need that facility.

PDO ships with PHP 5.1, and is available as a PECL extension for PHP 5.0; PDO requires the new OO features in the core of PHP 5, and so will not run with earlier versions of PHP.

Installation

PHP 5.1 and up on Unix systems

1. If you're running a PHP 5.1 release, PDO and PDO_SQLITE is included in the distribution; it will be automatically enabled when you run configure. It is recommended that you build PDO as a shared extension, as this will allow you to take advantage of updates that are made available via PECL. The recommended configure line for building PHP with PDO support should enable zlib support (for the pecl installer) as well. You may also need to enable the PDO driver for your database of choice; consult the documentation for database-specific PDO drivers to find out more about that, but note that if you build PDO as a shared extension, you must build the PDO drivers as shared extensions. SQLite extension depends on PDO so if PDO is built as a shared extension, SQLite needs to be built the same way.

```
./configure --with-zlib --enable-pdo=shared --with-pdo-sqlite=shared
--with-sqlite=shared
```

2. After installing PDO as a shared module, you must edit your php.ini file so that the PDO extension will be loaded automatically when PHP runs. You will also need to enable any database specific drivers there too; make sure that they are listed after the pdo.so line, as PDO must be initialized before the database-specific extensions can be loaded. If you built PDO and the database-specific extensions statically, you can skip this step.

```
extension=pdo.so
```

3. Having PDO as a shared module will allow you to run **pecl upgrade pdo** as new versions of PDO are published, without forcing you to rebuild the whole of PHP. Note that if you do this, you also need to upgrade your database specific PDO drivers at the same time.

PHP 5.0 and up on Unix systems

1. PDO is available as a PECL extension from http://pecl.php.net/package/pdo. Installation can be performed via the **pecl** tool; this is enabled by default when you configure PHP. You should ensure that PHP was configured --with-zlib in order for **pecl** to be able to handle the compressed package files.

2. Run the following command to download, build, and install the latest stable version of PDO:

```
pecl install pdo
```

3. The **pecl** command automatically installs the PDO module into your PHP extensions directory. To enable the PDO extension on Linux or Unix operating systems, you must add the following line to `php.ini`:

```
extension=pdo.so
```

For more information about building PECL packages, consult the PECL installation section of the manual.

Windows users running PHP 5.1 and up

1. PDO and all the major drivers ship with PHP as shared extensions, and simply need to be activated by editing the `php.ini` file:

```
extension=php_pdo.dll
```

2. Next, choose the other database-specific DLL files and either use **dl()** to load them at runtime, or enable them in `php.ini` below `php_pdo.dll`. For example:

```
extension=php_pdo.dll
extension=php_pdo_firebird.dll
extension=php_pdo_informix.dll
extension=php_pdo_mssql.dll
extension=php_pdo_mysql.dll
extension=php_pdo_oci.dll
extension=php_pdo_oci8.dll
extension=php_pdo_odbc.dll
extension=php_pdo_pgsql.dll
extension=php_pdo_sqlite.dll
```

These DLLs should exist in the system's extension_dir. Note that PDO_INFORMIX is only available as a PECL extension.

Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 1. PDO Configuration Options

Name	Default	Changeable	Changelog
pdo.dsn.*		`php.ini` only	

For further details and definitions of the PHP_INI_* constants, see the Appendix G.

Here's a short explanation of the configuration directives.

*pdo.dsn.** **string**

Defines DSN alias. See PDO::__construct for thorough explanation.

PDO Drivers

The following drivers currently implement the PDO interface:

Driver name	Supported databases
PDO_DBLIB	FreeTDS / Microsoft SQL Server / Sybase
PDO_FIREBIRD	Firebird/Interbase 6
PDO_INFORMIX	IBM Informix Dynamic Server
PDO_MYSQL	MySQL 3.x/4.x
PDO_OCI	Oracle Call Interface
PDO_ODBC	ODBC v3 (IBM DB2, unixODBC and win32 ODBC)
PDO_PGSQL	PostgreSQL
PDO_SQLITE	SQLite 3 and SQLite 2

Connections and Connection management

Connections are established by creating instances of the PDO base class. It doesn't matter which driver you want to use; you always use the PDO class name. The constructor accepts parameters for specifying the database source (known as the DSN) and optionally for the username and password (if any).

Example 1. Connecting to MySQL

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
?>
```

If there are any connection errors, a *PDOException* object will be thrown. You may catch the exception if you want to handle the error condition, or you may opt to leave it for an application global exception handler that you set up via **set_exception_handler()**.

Example 2. Handling connection errors

```
<?php
try {
   $dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
   foreach ($dbh->query('SELECT * from FOO') as $row) {
      print_r($row);
   }
   $dbh = null;
} catch (PDOException $e) {
   print "Error!: " . $e->getMessage() . "<br/>";
   die();
}
?>
```

Warning

If your application does not catch the exception thrown from the PDO constructor, the default action taken by the zend engine is to terminate the script and display a back trace. This back trace will likely reveal the full database connection details, including the username and password. It is your responsibility to catch this exception, either explicitly (via a *catch* statement) or implicitly via **set_exception_handler()**.

Upon successful connection to the database, an instance of the PDO class is returned to your script. The connection remains active for the lifetime of that PDO object. To close the connection, you need to destroy the object by ensuring that all remaining references to it are deleted--you do this by assigning `NULL` to the variable that holds the object. If you don't do this explicitly, PHP will automatically close the connection when your script ends.

Example 3. Closing a connection

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
// use the connection here


// and now we're done; close it
$dbh = null;
?>
```

Many web applications will benefit from making persistent connections to database servers. Persistent connections are not closed at the end of the script, but are cached and re-used when another script requests a connection using the same credentials. The persistent connection cache allows you to avoid the overhead of establishing a new connection every time a script needs to talk to a database, resulting in a faster web application.

Example 4. Persistent connections

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass, array(
  PDO::ATTR_PERSISTENT => true
));
?>
```

Note: If you're using the PDO ODBC driver and your ODBC libraries support ODBC Connection Pooling (unixODBC and Windows are two that do; there may be more), then it's recommended that you don't use persistent PDO connections, and instead leave the connection caching to the ODBC Connection Pooling layer. The ODBC Connection Pool is shared with other modules in the process; if PDO is told to cache the connection, then that connection would never be returned to the ODBC connection pool, resulting in additional connections being created to service those other modules.

Transactions and auto-commit

Now that you're connected via PDO, you must understand how PDO manages transactions before you start issuing queries. If you've never encountered transactions before, they offer 4 major features: Atomicity, Consistency, Isolation and Durability (ACID). In layman's terms, any work carried out in a transaction, even if it is carried out in stages, is guaranteed to be applied to the database safely, and without interference from other connections, when it is committed. Transactional work can also be automatically undone at your request (provided you haven't already committed it), which makes error handling in your scripts easier.

Transactions are typically implemented by "saving-up" your batch of changes to be applied all at once; this has the nice side effect of drastically improving the efficiency of those updates. In other words, transactions can make your scripts faster and potentially more robust (you still need to use them correctly to reap that benefit).

Unfortunately, not every database supports transactions, so PDO needs to run in what is known as "auto-commit" mode when you first open the connection. Auto-commit mode means that every query that you run has its own implicit transaction, if the database supports it, or no transaction if the database doesn't support transactions. If you need a transaction, you must use the **PDO::beginTransaction()** method to initiate one. If the underlying driver does not support transactions, a PDOException will be thrown (regardless of your error handling settings: this is always a serious error condition). Once you are in a transaction, you may use **PDO::commit()** or **PDO::rollBack()** to finish it, depending on the success of the code you run during the transaction.

When the script ends or when a connection is about to be closed, if you have an outstanding transaction, PDO will automatically roll it back. This is a safety measure to help avoid inconsistency in the cases where the script terminates unexpectedly--if you didn't explicitly commit the transaction, then it is assumed that something went awry, so the rollback is performed for the safety of your data.

Warning
The automatic rollback only happens if you initiate the transaction via **PDO::beginTransaction()**. If you manually issue a query that begins a transaction PDO has no way of knowing about it and thus cannot roll it back if something bad happens.

Example 5. Executing a batch in a transaction

In the following sample, let's assume that we are creating a set of entries for a new employee, who has been assigned an ID number of 23. In addition to entering the basic data for that person, we also need to record their salary. It's pretty simple to make two separate updates, but by enclosing them within the **PDO::beginTransaction()** and **PDO::commit()** calls, we are guaranteeing that no one else will be able to see those changes until they are complete. If something goes wrong, the catch block rolls back all changes made since the transaction was started, and then prints out an error message.

```
<?php
try {
  $dbh = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2',
      array(PDO::ATTR_PERSISTENT => true));
  echo "Connected\n";
  $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

  $dbh->beginTransaction();
```

```
  $dbh->exec("insert into staff (id, first, last) values (23, 'Joe', 'Bloggs')");
  $dbh->exec("insert into salarychange (id, amount, changedate)
      values (23, 50000, NOW())");
  $dbh->commit();

} catch (Exception $e) {
  $dbh->rollBack();
  echo "Failed: " . $e->getMessage();
}
?>
```

You're not limited to making updates in a transaction; you can also issue complex queries to extract data, and possibly use that information to build up more updates and queries; while the transaction is active, you are guaranteed that no one else can make changes while you are in the middle of your work. In truth, this isn't 100% correct, but it is a good-enough introduction, if you've never heard of transactions before.

Prepared statements and stored procedures

Many of the more mature databases support the concept of prepared statements. What are they? You can think of them as a kind of compiled template for the SQL that you want to run, that can be customized using variable parameters. Prepared statements offer two major benefits:

- The query only needs to be parsed (or prepared) once, but can be executed multiple times with the same or different parameters. When the query is prepared, the database will analyze, compile and optimize it's plan for executing the query. For complex queries this process can take up enough time that it will noticeably slow down your application if you need to repeat the same query many times with different parameters. By using a prepared statement you avoid repeating the analyze/compile/optimize cycle. In short, prepared statements use fewer resources and thus run faster.

- The parameters to prepared statements don't need to be quoted; the driver handles it for you. If your application exclusively uses prepared statements, you can be sure that no SQL injection will occur. (However, if you're still building up other parts of the query based on untrusted input, you're still at risk).

Prepared statements are so useful that they are the only feature that PDO will emulate for drivers that don't support them. This ensures that you will be able to use the same data access paradigm regardless of the capabilities of the database.

Example 6. Repeated inserts using prepared statements

This example performs an INSERT query by substituting a *name* and a *value* for the named placeholders.

```
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// insert one row
$name = 'one';
$value = 1;
$stmt->execute();

// insert another row with different values
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

Example 7. Repeated inserts using prepared statements

This example performs an INSERT query by substituting a *name* and a *value* for the positional *?* placeholders.

```
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (?, ?)");
$stmt->bindParam(1, $name);
$stmt->bindParam(2, $value);

// insert one row
$name = 'one';
$value = 1;
$stmt->execute();

// insert another row with different values
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

Example 8. Fetching data using prepared statements

This example fetches data based on a key value supplied by a form. The user input is automatically quoted, so there is no risk of a SQL injection attack.

```
<?php
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where name = ?");
if ($stmt->execute(array($_GET['name']))) {
  while ($row = $stmt->fetch()) {
    print_r($row);
  }
}
?>
```

If the database driver supports it, you may also bind parameters for output as well as input. Output parameters are typically used to retrieve values from stored procedures. Output parameters are slightly more complex to use than input parameters, in that you must know how large a given parameter might be when you bind it. If the value turns out to be larger than the size you suggested, an error is raised.

Example 9. Calling a stored procedure with an output parameter

```
<?php
$stmt = $dbh->prepare("CALL sp_returns_string(?)");
$stmt->bindParam(1, $return_value, PDO::PARAM_STR, 4000);

// call the stored procedure
$stmt->execute();

print "procedure returned $return_value\n";
?>
```

You may also specify parameters that hold values both input and output; the syntax is similar to output parameters. In this next example, the string 'hello' is passed into the stored procedure, and when it returns, hello is replaced with the return value of the procedure.

Example 10. Calling a stored procedure with an input/output parameter

```
<?php
$stmt = $dbh->prepare("CALL sp_takes_string_returns_string(?)");
$value = 'hello';
$stmt->bindParam(1, $value, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 4000);
```

```
// call the stored procedure
$stmt->execute();

print "procedure returned $value\n";
?>
```

Example 11. Invalid use of placeholder

```
<?php
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where name LIKE '%?%'");
$stmt->execute(array($_GET['name']));

// placeholder must be used in the place of the whole value
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where name LIKE ?");
$stmt->execute(array("%$_GET[name]%"));
?>
```

Errors and error handling

PDO offers you a choice of 3 different error handling strategies, to fit your style of application development.

- **PDO::ERRMODE_SILENT**

  This is the default mode. PDO will simply set the error code for you to inspect using the **PDO::errorCode()** and **PDO::errorInfo()** methods on both the statement and database objects; if the error resulted from a call on a statement object, you would invoke the **PDOStatement::errorCode()** or **PDOStatement::errorInfo()** method on that object. If the error resulted from a call on the database object, you would invoke those methods on the database object instead.

- **PDO::ERRMODE_WARNING**

  In addition to setting the error code, PDO will emit a traditional E_WARNING message. This setting is useful during debugging/testing, if you just want to see what problems occurred without interrupting the flow of the application.

- **PDO::ERRMODE_EXCEPTION**

  In addition to setting the error code, PDO will throw a **PDOException** and set its properties to reflect the error code and error information. This setting is also useful during debugging, as it will effectively "blow up" the script at the point of the error, very quickly pointing a finger at potential problem areas in your code (remember: transactions are automatically rolled back if the exception causes the script to terminate).

  Exception mode is also useful because you can structure your error handling more clearly than with traditional PHP-style warnings, and with less code/nesting than by running in silent mode and explicitly checking the return value of each database call.

  See Exceptions for more information about Exceptions in PHP.

PDO standardizes on using SQL-92 SQLSTATE error code strings; individual PDO drivers are responsible for mapping their native codes to the appropriate SQLSTATE codes. The **PDO::errorCode()** method returns a single SQLSTATE code. If you need more specific information about an error, PDO also offers an **PDO::errorInfo()** method which returns an array containing the SQLSTATE code, the driver specific error code and driver specific error string.

Large Objects (LOBs)

At some point in your application, you might find that you need to store "large" data in your database. Large typically means "around 4kb or more", although some databases can happily handle up to 32kb before data becomes "large". Large objects can be either textual or binary in nature. PDO allows you to work with this large data type by using the **PDO::PARAM_LOB** type code in your **PDOStatement::bindParam()** or **PDOStatement::bindColumn()** calls. **PDO::PARAM_LOB** tells PDO to map the data as a stream, so that you can manipulate it using the PHP Streams API.

Example 12. Displaying an image from a database

This example binds the LOB into the variable named $lob and then sends it to the browser using **fpassthru()**. Since the LOB is represented as a stream, functions such as **fgets()**, **fread()** and **stream_get_contents()** can be used on it.

```
<?php
$db = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$stmt = $db->prepare("select contenttype, imagedata from images where id=?");
$stmt->execute(array($_GET['id']));
$stmt->bindColumn(1, $type, PDO::PARAM_STR, 256);
$stmt->bindColumn(2, $lob, PDO::PARAM_LOB);
$stmt->fetch(PDO::FETCH_BOUND);

header("Content-Type: $type");
fpassthru($lob);
?>
```

Example 13. Inserting an image into a database

This example opens up a file and passes the file handle to PDO to insert it as a LOB. PDO will do its best to get the contents of the file up to the database in the most efficient manner possible.

```
<?php
$db = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$stmt = $db->prepare("insert into images (id, contenttype, imagedata) values (?, ?, ?)");
$id = get_new_id(); // some function to allocate a new ID

// assume that we are running as part of a file upload form
// You can find more information in the PHP documentation

$fp = fopen($_FILES['file']['tmp_name'], 'rb');

$stmt->bindParam(1, $id);
$stmt->bindParam(2, $_FILES['file']['type']);
$stmt->bindParam(3, $fp, PDO::PARAM_LOB);

$stmt->beginTransaction();
$stmt->execute();
$stmt->commit();
?>
```

Example 14. Inserting an image into a database: Oracle

Oracle requires a slightly different syntax for inserting a lob from a file. It's also essential that you perform the insert under a transaction, otherwise your newly inserted LOB will be committed with a zero-length as part of the implicit commit that happens when the query is executed:

```
<?php
$db = new PDO('oci:', 'scott', 'tiger');
$stmt = $db->prepare("insert into images (id, contenttype, imagedata) " .
  "VALUES (?, ?, EMPTY_BLOB()) RETURNING imagedata INTO ?");
$id = get_new_id(); // some function to allocate a new ID

// assume that we are running as part of a file upload form
// You can find more information in the PHP documentation
```

```
$fp = fopen($_FILES['file']['tmp_name'], 'rb');

$stmt->bindParam(1, $id);
$stmt->bindParam(2, $_FILES['file']['type']);
$stmt->bindParam(3, $fp, PDO::PARAM_LOB);

$stmt->beginTransaction();
$stmt->execute();
$stmt->commit();
?>
```

Predefined Classes

PDO

Represents a connection between PHP and a database server.

Constructor

- PDO - constructs a new PDO object

Methods

- beginTransaction - begins a transaction
- commit - commits a transaction
- errorCode - retrieves an error code, if any, from the database
- errorInfo - retrieves an array of error information, if any, from the database
- exec - issues an SQL statement and returns the number of affected rows
- getAttribute - retrieves a database connection attribute
- lastInsertId - retrieves the value of the last row that was inserted into a table
- prepare - prepares an SQL statement for execution
- query - issues an SQL statement and returns a result set
- quote - returns a quoted version of a string for use in SQL statements
- rollBack - roll back a transaction
- setAttribute - sets a database connection attribute

PDOStatement

Represents a prepared statement and, after the statement is executed, an associated result set.

Methods

- bindColumn - binds a PHP variable to an output column in a result set
- bindParam - binds a PHP variable to a parameter in the prepared statement
- bindValue - binds a value to a parameter in the prepared statement
- closeCursor - closes the cursor, allowing the statement to be executed again
- columnCount - returns the number of columns in the result set

- errorCode - retrieves an error code, if any, from the statement
- errorInfo - retrieves an array of error information, if any, from the statement
- execute - executes a prepared statement
- fetch - fetches a row from a result set
- fetchAll - fetches an array containing all of the rows from a result set
- fetchColumn - returns the data from a single column in a result set
- getAttribute - retrieves a PDOStatement attribute
- getColumnMeta - retrieves metadata for a column in the result set
- nextRowset - retrieves the next rowset (result set)
- rowCount - returns the number of rows that were affected by the execution of an SQL statement
- setAttribute - sets a PDOStatement attribute
- setFetchMode - sets the fetch mode for a PDOStatement

PDOException

Represents an error raised by PDO. You should not throw a **PDOException** from your own code. See Exceptions for more information about Exceptions in PHP.

Example 15. The PDOException class

```
<?php
class PDOException extends Exception
{
    public $errorInfo = null;    // corresponds to PDO::errorInfo()
                                 // or PDOStatement::errorInfo()
    protected $message;          // textual error message
                                 // use Exception::getMessage() to access it
    protected $code;             // SQLSTATE error code
                                 // use Exception::getCode() to access it
}
?>
```

Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

Warning
PDO uses class constants since PHP 5.1. Prior releases use global constants in the form **PDO_PARAM_BOOL**.

PDO::PARAM_BOOL (**integer**)

Represents a boolean data type.

PDO::PARAM_NULL (**integer**)

Represents the SQL NULL data type.

PDO::PARAM_INT (**integer**)

Represents the SQL INTEGER data type.

PDO::PARAM_STR (**integer**)

Represents the SQL CHAR, VARCHAR, or other string data type.

PDO::PARAM_LOB (**integer**)

Represents the SQL large object data type.

PDO::PARAM_STMT (**integer**)

Represents a recordset type. Not currently supported by any drivers.

PDO::PARAM_INPUT_OUTPUT (**integer**)

Specifies that the parameter is an INOUT parameter for a stored procedure. You must bitwise-OR this value with an explicit PDO::PARAM_* data type.

PDO::FETCH_LAZY (**integer**)

Specifies that the fetch method shall return each row as an object with variable names that correspond to the column names returned in the result set. PDO::FETCH_LAZY creates the object variable names as they are accessed.

PDO::FETCH_ASSOC (**integer**)

Specifies that the fetch method shall return each row as an array indexed by column name as returned in the corresponding result set. If the result set contains multiple columns with the same name, PDO::FETCH_ASSOC returns only a single value per column name.

PDO::FETCH_NAMED (**integer**)

Specifies that the fetch method shall return each row as an array indexed by column name as returned in the corresponding result set. If the result set contains multiple columns with the same name, PDO::FETCH_NAMED returns an array of values per column name.

PDO::FETCH_NUM (**integer**)

Specifies that the fetch method shall return each row as an array indexed by column number as returned in the corresponding result set, starting at column 0.

PDO::FETCH_BOTH (**integer**)

Specifies that the fetch method shall return each row as an array indexed by both column name and number as returned in the corresponding result set, starting at column 0.

PDO::FETCH_OBJ (**integer**)

Specifies that the fetch method shall return each row as an object with property names that correspond to the column names returned in the result set.

PDO::FETCH_BOUND (**integer**)

Specifies that the fetch method shall return TRUE and assign the values of the columns in the result set to the PHP variables to which they were bound with the **PDOStatement::bindParam()** or **PDOStatement::bindColumn()** methods.

PDO::FETCH_COLUMN (**integer**)

Specifies that the fetch method shall return only a single requested column from the next row in the result set.

PDO::FETCH_CLASS (**integer**)

Specifies that the fetch method shall return a new instance of the requested class, mapping the columns to named properties in the class.

PDO::FETCH_INTO (**integer**)

Specifies that the fetch method shall update an existing instance of the requested class, mapping the columns to named properties in the class.

PDO::FETCH_FUNC (**integer**)

PDO::FETCH_GROUP (**integer**)

PDO::FETCH_UNIQUE (**integer**)

PDO::FETCH_CLASSTYPE (**integer**)

PDO::FETCH_SERIALIZE (**integer**)

Available since PHP 5.1.0.

PDO::FETCH_PROPS_LATE (**integer**)

PDO::ATTR_AUTOCOMMIT (**integer**)

If this value is **FALSE**, PDO attempts to disable autocommit so that the connection begins a transaction.

PDO::ATTR_PREFETCH (**integer**)

Setting the prefetch size allows you to balance speed against memory usage for your application. Not all database/driver combinations support setting of the prefetch size. A larger prefetch size results in increased performance at the cost of higher memory usage.

PDO::ATTR_TIMEOUT (**integer**)

Sets the timeout value in seconds for communications with the database.

PDO::ATTR_ERRMODE (**integer**)

See the Errors and error handling section for more information about this attribute.

PDO::ATTR_SERVER_VERSION (**integer**)

This is a read only attribute; it will return information about the version of the database server to which PDO is connected.

PDO::ATTR_CLIENT_VERSION (**integer**)

This is a read only attribute; it will return information about the version of the client libraries that the PDO driver is using.

PDO::ATTR_SERVER_INFO (**integer**)

This is a read only attribute; it will return some meta information about the database server to which PDO is connected.

PDO::ATTR_CONNECTION_STATUS (**integer**)

PDO::ATTR_CASE (**integer**)

Force column names to a specific case specified by the PDO::CASE_* constants.

PDO::ATTR_CURSOR_NAME (**integer**)

Get or set the name to use for a cursor. Most useful when using scrollable cursors and positioned updates.

PDO::ATTR_CURSOR (**integer**)

Selects the cursor type. PDO currently supports either **PDO::CURSOR_FWDONLY** and **PDO::CURSOR_SCROLL**. Stick with **PDO::CURSOR_FWDONLY** unless you know that you need a scrollable cursor.

PDO::ATTR_DRIVER_NAME (**string**)

Returns the name of the driver.

Example 16. using PDO::ATTR_DRIVER_NAME

```
<?php
if ($db->getAttribute(PDO::ATTR_DRIVER_NAME) == 'mysql') {
  echo "Running on mysql; doing something mysql specific here\n";
}
?>
```

PDO::ATTR_ORACLE_NULLS (**integer**)

Convert empty strings to SQL NULL values on data fetches.

PDO::ATTR_PERSISTENT (**integer**)

Request a persistent connection, rather than creating a new connection. See Connections and Connection management for more information on this attribute.

PDO::ATTR_STATEMENT_CLASS (**integer**)

PDO::ATTR_FETCH_CATALOG_NAMES (**integer**)

Prepend the containing catalog name to each column name returned in the result set. The catalog name and column name are separated by a decimal (.) character. Support of this attribute is at the driver level; it may not be supported by your driver.

PDO::ATTR_FETCH_TABLE_NAMES (**integer**)

Prepend the containing table name to each column name returned in the result set. The table name and column name are separated by a decimal (.) character. Support of this attribute is at the driver level; it may not be supported by your driver.

PDO::ATTR_STRINGIFY_FETCHES (**integer**)

PDO::ATTR_MAX_COLUMN_LEN (**integer**)

PDO::ATTR_DEFAULT_FETCH_MODE (**integer**)

PDO::ATTR_EMULATE_PREPARES (**integer**)

Available since PHP 5.1.3.

PDO::ERRMODE_SILENT (**integer**)

Do not raise an error or exception if an error occurs. The developer is expected to explicitly check for errors. This is the default mode. See Errors and error handling for more information about this attribute.

PDO::ERRMODE_WARNING (**integer**)

Issue a PHP E_WARNING message if an error occurs. See Errors and error handling for more information about this attribute.

PDO::ERRMODE_EXCEPTION (**integer**)

Throw a **PDOException** if an error occurs. See Errors and error handling for more information about this attribute.

PDO::CASE_NATURAL (**integer**)

Leave column names as returned by the database driver.

PDO::CASE_LOWER (**integer**)

Force column names to lower case.

PDO::CASE_UPPER (**integer**)

Force column names to upper case.

PDO::PDO_NULL_NATURAL (**integer**)

PDO::PDO_NULL_EMPTY_STRING (**integer**)

PDO::PDO_NULL_TO_STRING (**integer**)

PDO::FETCH_ORI_NEXT (**integer**)

Fetch the next row in the result set. Valid only for scrollable cursors.

PDO::FETCH_ORI_PRIOR (**integer**)

Fetch the previous row in the result set. Valid only for scrollable cursors.

PDO::FETCH_ORI_FIRST (**integer**)

Fetch the first row in the result set. Valid only for scrollable cursors.

PDO::FETCH_ORI_LAST (**integer**)

Fetch the last row in the result set. Valid only for scrollable cursors.

PDO::FETCH_ORI_ABS (**integer**)

Fetch the requested row by row number from the result set. Valid only for scrollable cursors.

PDO::FETCH_ORI_REL (**integer**)

Fetch the requested row by relative position from the current position of the cursor in the result set. Valid only for scrollable cursors.

PDO::CURSOR_FWDONLY (**integer**)

Create a PDOStatement object with a forward-only cursor. This is the default cursor choice, as it is the fastest and most common data access pattern in PHP.

PDO::CURSOR_SCROLL (**integer**)

Create a PDOStatement object with a scrollable cursor. Pass the PDO::FETCH_ORI_* constants to control the rows fetched from the result set.

PDO::ERR_NONE (**string**)

Corresponds to SQLSTATE '00000', meaning that the SQL statement was successfully issued with no errors or warnings. This constant is for your convenience when checking **PDO::errorCode()** or **PDOStatement::errorCode()** to determine if an error occurred. You will usually know if this is the case by examining the return code from the method that raised the error condition anyway.

PDO::PARAM_EVT_ALLOC (**integer**)

Allocation event

PDO::PARAM_EVT_FREE (**integer**)

Deallocation event

PDO::PARAM_EVT_EXEC_PRE (**integer**)

Event triggered prior to execution of a prepared statement.

PDO::PARAM_EVT_EXEC_POST (**integer**)

Event triggered subsequent to execution of a prepared statement.

PDO::PARAM_EVT_FETCH_PRE (**integer**)

Event triggered prior to fetching a result from a resultset.

PDO::PARAM_EVT_FETCH_POST (**integer**)

Event triggered subsequent to fetching a result from a resultset.

PDO::PARAM_EVT_NORMALIZE (**integer**)

Event triggered during bound parameter registration allowing the driver to normalize the parameter name.

Table of Contents

Last updated: Tue, 22 Sep 2015