PHP Data Objects

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.

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

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 5.0.0 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 5.1.0 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*.

PDO Configuration Options

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

For further details and definitions of the PHP_INI_* constants, see the php.ini directives.

Here's a short explanation of the configuration directives.

*pdo.dsn.** string
Defines DSN alias. See PDO::__construct() for thorough explanation.

Resource Types

This extension has no resource types defined.

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_KEY_PAIR (integer)
Fetch into an array where the 1st column is a key and all subsequent columns are values

PDO::FETCH_CLASSTYPE (integer)

PDO::FETCH_SERIALIZE (integer)
As **PDO::FETCH_INTO** but object is provided as a serialized string. Available since PHP 5.1.0.

PDO::FETCH_PROPS_LATE (integer)
Available since PHP 5.2.0

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 #1 - 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)
Available since PHP 5.2.0

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::NULL_NATURAL (integer)

PDO::NULL_EMPTY_STRING (integer)

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.

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 #2 - 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 #3 - 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 #4 - 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 #5 - Persistent connections

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

Note

If you wish to use persistent connections, you must set **PDO::ATTR_PERSISTENT** in the array of driver options passed to the PDO constructor. If setting this attribute with PDO::setAttribute() after instantiation of the object, the driver will not use persistent connections.

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 #6 - 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 #7 - 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 #8 - 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 #9 - 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 #10 - 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 #11 - 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 #12 - 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 #13 - 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 #14 - 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);

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

Example #15 - 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();
?>
```

The PDO class

Introduction

Represents a connection between PHP and a database server.

Class synopsis

PDO

PDO {

PDO::__construct (string `$dsn` [, string `$username` [, string `$password` [, array `$driver_options`]]])

bool **PDO::beginTransaction** (void)

bool **PDO::commit** (void)

string **PDO::errorCode** (void)

array **PDO::errorInfo** (void)

int **PDO::exec** (string `$statement`)

mixed **PDO::getAttribute** (int `$attribute`)

array **PDO::getAvailableDrivers** (void)

string **PDO::lastInsertId** ([string `$name`])

PDOStatement **PDO::prepare** (string `$statement` [, array `$driver_options`])

PDOStatement **PDO::query** (string `$statement`)

string **PDO::quote** (string `$string` [, int `$parameter_type`])

bool **PDO::rollBack** (void)

bool **PDO::setAttribute** (int `$attribute`, mixed `$value`)
}

PDO::beginTransaction

PDO::beginTransaction -- Initiates a transaction

Description

bool **PDO::beginTransaction** (void)

Turns off autocommit mode. While autocommit mode is turned off, changes made to the database via the PDO object instance are not committed until you end the transaction by calling PDO::commit(). Calling PDO::rollBack() will roll back all changes to the database and return the connection to autocommit mode.

Some databases, including MySQL, automatically issue an implicit COMMIT when a database definition language (DDL) statement such as DROP TABLE or CREATE TABLE is issued within a transaction. The implicit COMMIT will prevent you from rolling back any other changes within the transaction boundary.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #16 - Roll back a transaction

The following example begins a transaction and issues two statements that modify the database before rolling back the changes. On MySQL, however, the DROP TABLE statement automatically commits the transaction so that none of the changes in the transaction are rolled back.

```
<?php
/* Begin a transaction, turning off autocommit */
$dbh->beginTransaction();

/* Change the database schema and data */
$sth = $dbh->exec("DROP TABLE fruit");
$sth = $dbh->exec("UPDATE dessert
   SET name = 'hamburger'");

/* Recognize mistake and roll back changes */
$dbh->rollBack();

/* Database connection is now back in autocommit mode */
?>
```

See Also

- PDO::commit()
- PDO::rollBack()

PDO::commit

PDO::commit -- Commits a transaction

Description

bool **PDO::commit** (void)

Commits a transaction, returning the database connection to autocommit mode until the next call to PDO::beginTransaction() starts a new transaction.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #17 - Commit a transaction

```
<?php
/* Begin a transaction, turning off autocommit */
$dbh->beginTransaction();

/* Change the database schema */
$sth = $dbh->exec("DROP TABLE fruit");

/* Commit the changes */
$dbh->commit();

/* Database connection is now back in autocommit mode */
?>
```

See Also

- PDO::beginTransaction()
- PDO::rollBack()

PDO::__construct

PDO::__construct -- Creates a PDO instance representing a connection to a database

Description

PDO::__construct (string `$dsn` [, string `$username` [, string `$password` [, array `$driver_options`]]])

Creates a PDO instance to represent a connection to the requested database.

Parameters

dsn
: The Data Source Name, or DSN, contains the information required to connect to the database. In general, a DSN consists of the PDO driver name, followed by a colon, followed by the PDO driver-specific connection syntax. Further information is available from the PDO driver-specific documentation. The *`dsn`* parameter supports three different methods of specifying the arguments required to create a database connection:

  Driver invocation
  : *`dsn`* contains the full DSN.

  URI invocation
  : *`dsn`* consists of **uri:** followed by a URI that defines the location of a file containing the DSN string. The URI can specify a local file or a remote URL.
    uri:file:///path/to/dsnfile

  Aliasing
  : *`dsn`* consists of a name *`name`* that maps to `pdo.dsn.` *`name`* in *php.ini* defining the DSN string.

Note
The alias must be defined in *php.ini*, and not*.htaccess* or *httpd.conf*

username
: The user name for the DSN string. This parameter is optional for some PDO drivers.

password
: The password for the DSN string. This parameter is optional for some PDO drivers.

driver_options
: A key=>value array of driver-specific connection options.

Return Values

Returns a PDO object on success.

Errors/Exceptions

PDO::__construct() throws a PDOException if the attempt to connect to the requested database fails.

Examples

Example #18 - Create a PDO instance via driver invocation

```
<?php
/* Connect to an ODBC database using driver invocation */
$dsn = 'mysql:dbname=testdb;host=127.0.0.1';
$user = 'dbuser';
$password = 'dbpass';

try {
   $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
   echo 'Connection failed: ' . $e->getMessage();
}

?>
```

Example #19 - Create a PDO instance via URI invocation

The following example assumes that the file */usr/local/dbconnect* exists with file permissions that enable PHP to read the file. The file contains the PDO DSN to connect to a DB2 database through the PDO_ODBC driver:

```
odbc:DSN=SAMPLE;UID=john;PWD=mypass
```

The PHP script can then create a database connection by simply passing the *uri:* parameter and pointing to the file URI:

```
<?php
/* Connect to an ODBC database using driver invocation */
$dsn = 'uri:file:///usr/local/dbconnect';
$user = '';
$password = '';

try {
   $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
   echo 'Connection failed: ' . $e->getMessage();
}

?>
```

Example #20 - Create a PDO instance using an alias

The following example assumes that *php.ini* contains the following entry to enable a connection to a MySQL database using only the alias *mydb*:

```
[PDO]
pdo.dsn.mydb="mysql:dbname=testdb;host=localhost"
```

```
<?php
/* Connect to an ODBC database using an alias */
$dsn = 'mydb';
$user = '';
$password = '';

try {
   $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
   echo 'Connection failed: ' . $e->getMessage();
}

?>
```

PDO::errorCode

PDO::errorCode -- Fetch the SQLSTATE associated with the last operation on the database handle

Description

string **PDO::errorCode** (void)

Return Values

Returns a SQLSTATE, a five-character alphanumeric identifier defined in the ANSI SQL-92 standard. Briefly, an SQLSTATE consists of a two-character class value followed by a three-character subclass value. A class value of 01 indicates a warning and is accompanied by a return code of SQL_SUCCESS_WITH_INFO. Class values other than '01', except for the class 'IM', indicate an error. The class 'IM' is specific to warnings and errors that derive from the implementation of PDO (or perhaps ODBC, if you're using the ODBC driver) itself. The subclass value '000' in any class indicates that there is no subclass for that SQLSTATE.

PDO::errorCode() only retrieves error codes for operations performed directly on the database handle. If you create a PDOStatement object through PDO::prepare() or PDO::query() and invoke an error on the statement handle, PDO::errorCode() will not reflect that error. You must call PDOStatement::errorCode() to return the error code for an operation performed on a particular statement handle.

Examples

Example #21 - Retrieving a SQLSTATE code

```
<?php
/* Provoke an error -- the BONES table does not exist */
$dbh->exec("INSERT INTO bones(skull) VALUES ('lucy')");

echo "\nPDO::errorCode(): ";
print $dbh->errorCode();
?>
```

The above example will output:

```
PDO::errorCode(): 42S02
```

See Also

- PDO::errorInfo()
- PDOStatement::errorCode()
- PDOStatement::errorInfo()

PDO::errorInfo

PDO::errorInfo -- Fetch extended error information associated with the last operation on the database handle

Description

array **PDO::errorInfo** (void)

Return Values

PDO::errorInfo() returns an array of error information about the last operation performed by this database handle. The array consists of the following fields:

Element	Information
0	SQLSTATE error code (a five-character alphanumeric identifier defined in the ANSI SQL standard).
1	Driver-specific error code.
2	Driver-specific error message.

PDO::errorInfo() only retrieves error information for operations performed directly on the database handle. If you create a PDOStatement object through PDO::prepare() or PDO::query() and invoke an error on the statement handle, PDO::errorInfo() will not reflect the error from the statement handle. You must call PDOStatement::errorInfo() to return the error information for an operation performed on a particular statement handle.

Examples

Example #22 - Displaying errorInfo() fields for a PDO_ODBC connection to a DB2 database

```
<?php
/* Provoke an error -- bogus SQL syntax */
$stmt = $dbh->prepare('bogus sql');
if (!$stmt) {
   echo "\nPDO::errorInfo():\n";
   print_r($dbh->errorInfo());
}
?>
```

The above example will output:

```
PDO::errorInfo():
Array
(
   [0] => HY000
   [1] => 1
   [2] => near "bogus": syntax error
)
```

See Also

- PDO::errorCode()
- PDOStatement::errorCode()
- PDOStatement::errorInfo()

PDO::exec

PDO::exec -- Execute an SQL statement and return the number of affected rows

Description

int **PDO::exec** (string `$statement`)

PDO::exec() executes an SQL statement in a single function call, returning the number of rows affected by the statement.

PDO::exec() does not return results from a SELECT statement. For a SELECT statement that you only need to issue once during your program, consider issuing PDO::query(). For a statement that you need to issue multiple times, prepare a PDOStatement object with PDO::prepare() and issue the statement with PDOStatement::execute().

Parameters

`statement`
The SQL statement to prepare and execute.

Return Values

PDO::exec() returns the number of rows that were modified or deleted by the SQL statement you issued. If no rows were affected, PDO::exec() returns *0*.

Warning
This function may return Boolean **FALSE**, but may also return a non-Boolean value which evaluates to **FALSE**, such as *0* or "". Please read the section on Booleans for more information. Use the === operator for testing the return value of this function.

The following example incorrectly relies on the return value of PDO::exec(), wherein a statement that affected 0 rows results in a call to die():

```
<?php
$db->exec() or die($db->errorInfo());
?>
```

Examples

Example #23 - Issuing a DELETE statement

Count the number of rows deleted by a DELETE statement with no WHERE clause.

```
<?php
$dbh = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');

/* Delete all rows from the FRUIT table */
$count = $dbh->exec("DELETE FROM fruit WHERE colour = 'red'");

/* Return number of rows that were deleted */
print("Deleted $count rows.\n");
?>
```

The above example will output:

```
Deleted 1 rows.
```

See Also

- PDO::prepare()
- PDO::query()
- PDOStatement::execute()

PDO::getAttribute

PDO::getAttribute -- Retrieve a database connection attribute

Description

mixed **PDO::getAttribute** (int `$attribute`)

This function returns the value of a database connection attribute. To retrieve PDOStatement attributes, refer to PDOStatement::getAttribute().

Note that some database/driver combinations may not support all of the database connection attributes.

Parameters

`attribute`
One of the *PDO::ATTR_** constants. The constants that apply to database connections are as follows:

- *PDO::ATTR_AUTOCOMMIT*
- *PDO::ATTR_CASE*
- *PDO::ATTR_CLIENT_VERSION*
- *PDO::ATTR_CONNECTION_STATUS*
- *PDO::ATTR_DRIVER_NAME*
- *PDO::ATTR_ERRMODE*
- *PDO::ATTR_ORACLE_NULLS*
- *PDO::ATTR_PERSISTENT*
- *PDO::ATTR_PREFETCH*
- *PDO::ATTR_SERVER_INFO*
- *PDO::ATTR_SERVER_VERSION*
- *PDO::ATTR_TIMEOUT*

Return Values

A successful call returns the value of the requested PDO attribute. An unsuccessful call returns *null*.

Examples

Example #24 - Retrieving database connection attributes

```
<?php
$conn = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');
```

```
$attributes = array(
   "AUTOCOMMIT", "ERRMODE", "CASE", "CLIENT_VERSION", "CONNECTION_STATUS",
   "ORACLE_NULLS", "PERSISTENT", "PREFETCH", "SERVER_INFO",
"SERVER_VERSION",
   "TIMEOUT"
);

foreach ($attributes as $val) {
   echo "PDO::ATTR_$val: ";
   echo $conn->getAttribute(constant("PDO::ATTR_$val")) . "\n";
}
?>
```

See Also

- PDO::setAttribute()
- PDOStatement::getAttribute()
- PDOStatement::setAttribute()

PDO::getAvailableDrivers

PDO::getAvailableDrivers -- Return an array of available PDO drivers

Description

array **PDO::getAvailableDrivers** (void)

This function returns all currently available PDO drivers which can be used in *DSN* parameter of PDO::__construct(). This is a static method.

Return Values

PDO::getAvailableDrivers() returns an array of PDO driver names. If no drivers are available, it returns an empty array.

Examples

Example #25 - A PDO::getAvailableDrivers() example

```
<?php
print_r(PDO::getAvailableDrivers());
?>
```

The above example will output something similar to:

```
Array
(
   [0] => mysql
   [1] => sqlite
)
```

PDO::lastInsertId

PDO::lastInsertId -- Returns the ID of the last inserted row or sequence value

Description

string **PDO::lastInsertId** ([string `$name`])

Returns the ID of the last inserted row, or the last value from a sequence object, depending on the underlying driver. For example, **PDO_PGSQL()** requires you to specify the name of a sequence object for the *`name`* parameter.

Note
This method may not return a meaningful or consistent result across different PDO drivers, because the underlying database may not even support the notion of auto-increment fields or sequences.

Parameters

`name`
: Name of the sequence object from which the ID should be returned.

Return Values

If a sequence name was not specified for the *`name`* parameter, PDO::lastInsertId() returns a string representing the row ID of the last row that was inserted into the database.

If a sequence name was specified for the *`name`* parameter, PDO::lastInsertId() returns a string representing the last value retrieved from the specified sequence object.

If the PDO driver does not support this capability, PDO::lastInsertId() triggers an *IM001* SQLSTATE.

PDO::prepare

PDO::prepare -- Prepares a statement for execution and returns a statement object

Description

PDOStatement **PDO::prepare** (string `$statement` [, array `$driver_options`])

Prepares an SQL statement to be executed by the PDOStatement::execute() method. The SQL statement can contain zero or more named (:name) or question mark (?) parameter markers for which real values will be substituted when the statement is executed. You cannot use both named and question mark parameter markers within the same SQL statement; pick one or the other parameter style.

You must include a unique parameter marker for each value you wish to pass in to the statement when you call PDOStatement::execute(). You cannot use a named parameter marker of the same name twice in a prepared statement. You cannot bind multiple values to a single named parameter in, for example, the IN() clause of an SQL statement.

Calling PDO::prepare() and PDOStatement::execute() for statements that will be issued multiple times with different parameter values optimizes the performance of your application by allowing the driver to negotiate client and/or server side caching of the query plan and meta information, and helps to prevent SQL injection attacks by eliminating the need to manually quote the parameters.

PDO will emulate prepared statements/bound parameters for drivers that do not natively support them, and can also rewrite named or question mark style parameter markers to something more appropriate, if the driver supports one style but not the other.

Parameters

`statement`
: This must be a valid SQL statement for the target database server.

`driver_options`
: This array holds one or more key=>value pairs to set attribute values for the PDOStatement object that this method returns. You would most commonly use this to set the *PDO::ATTR_CURSOR* value to *PDO::CURSOR_SCROLL* to request a scrollable cursor. Some drivers have driver specific options that may be set at prepare-time.

Return Values

If the database server successfully prepares the statement, PDO::prepare() returns a PDOStatement object. If the database server cannot successfully prepare the statement, PDO::prepare() returns **FALSE**.

Examples

Example #26 - Prepare an SQL statement with named parameters

```
<?php
/* Execute a prepared statement by passing an array of values */
$sql = 'SELECT name, colour, calories
   FROM fruit
   WHERE calories < :calories AND colour = :colour';
$sth = $dbh->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));
$sth->execute(array(':calories' => 150, ':colour' => 'red'));
$red = $sth->fetchAll();
$sth->execute(array('calories' => 175, 'colour' => 'yellow'));
$yellow = $sth->fetchAll();
?>
```

Example #27 - Prepare an SQL statement with question mark parameters

```
<?php
/* Execute a prepared statement by passing an array of values */
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < ? AND colour = ?');
$sth->execute(array(150, 'red'));
$red = $sth->fetchAll();
$sth->execute(array(175, 'yellow'));
$yellow = $sth->fetchAll();
?>
```

See Also

- PDO::exec()
- PDO::query()
- PDOStatement::execute()

PDO::query

PDO::query -- Executes an SQL statement, returning a result set as a PDOStatement object

Description

PDOStatement **PDO::query** (string `$statement`)

PDOStatement **PDO::query** (string `$statement`, int `$PDO::FETCH_COLUMN`, int `$colno`)

PDOStatement **PDO::query** (string `$statement`, int `$PDO::FETCH_CLASS`, string `$classname`, array `$ctorargs`)

PDOStatement **PDO::query** (string `$statement`, int `$PDO::FETCH_INTO`, object `$object`)

PDO::query() executes an SQL statement in a single function call, returning the result set (if any) returned by the statement as a PDOStatement object.

For a query that you need to issue multiple times, you will realize better performance if you prepare a PDOStatement object using PDO::prepare() and issue the statement with multiple calls to PDOStatement::execute().

If you do not fetch all of the data in a result set before issuing your next call to PDO::query(), your call may fail. Call PDOStatement::closeCursor() to release the database resources associated with the PDOStatement object before issuing your next call to PDO::query().

Note
Although this function is only documented as having a single parameter, you may pass additional arguments to this function. They will be treated as though you called PDOStatement::setFetchMode() on the resultant statement object.

Parameters

`statement`
The SQL statement to prepare and execute.

Return Values

PDO::query() returns a PDOStatement object.

Examples

Example #28 - Demonstrate PDO::query

A nice feature of PDO::query() is that it enables you to iterate over the rowset returned by a successfully executed SELECT statement.

```
<?php
function getFruit($conn) {
   $sql = 'SELECT name, colour, calories FROM fruit ORDER BY name';
   foreach ($conn->query($sql) as $row) {
       print $row['NAME'] . "\t";
       print $row['COLOUR'] . "\t";
       print $row['CALORIES'] . "\n";
   }
}
?>
```

The above example will output:

```
apple   red     150
banana  yellow  250
kiwi    brown   75
lemon   yellow  25
orange  orange  300
pear    green   150
watermelon      pink    90
```

See Also

- PDO::exec()
- PDO::prepare()
- PDOStatement::execute()

PDO::quote

PDO::quote -- Quotes a string for use in a query.

Description

string **PDO::quote** (string `$string` [, int `$parameter_type`])

PDO::quote() places quotes around the input string (if required) and escapes special characters within the input string, using a quoting style appropriate to the underlying driver.

If you are using this function to build SQL statements, you are *strongly* recommended to use PDO::prepare() to prepare SQL statements with bound parameters instead of using PDO::quote() to interpolate user input into a SQL statement. Prepared statements with bound parameters are not only more portable, more convenient, immune to SQL injection, but are often much faster to execute than interpolated queries, as both the server and client side can cache a compiled form of the query.

Not all PDO drivers implement this method (notably PDO_ODBC). Consider using prepared statements instead.

Parameters

`string`
: The string to be quoted.

`parameter_type`
: Provides a data type hint for drivers that have alternate quoting styles. The default value is PDO::PARAM_STR.

Return Values

Returns a quoted string that is theoretically safe to pass into an SQL statement. Returns **FALSE** if the driver does not support quoting in this way.

Examples

Example #29 - Quoting a normal string

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Simple string */
$string = 'Nice';
print "Unquoted string: $string\n";
```

```
print "Quoted string: " . $conn->quote($string) . "\n";
?>
```

The above example will output:

```
Unquoted string: Nice
Quoted string: 'Nice'
```

Example #30 - Quoting a dangerous string

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Dangerous string */
$string = 'Naughty \' string';
print "Unquoted string: $string\n";
print "Quoted string:" . $conn->quote($string) . "\n";
?>
```

The above example will output:

```
Unquoted string: Naughty ' string
Quoted string: 'Naughty '' string'
```

Example #31 - Quoting a complex string

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Complex string */
$string = "Co'mpl''ex \"st'\"ring";
print "Unquoted string: $string\n";
print "Quoted string: " . $conn->quote($string) . "\n";
?>
```

The above example will output:

```
Unquoted string: Co'mpl''ex "st'"ring
Quoted string: 'Co''mpl''''ex "st''"ring'
```

See Also

- PDO::prepare()
- PDOStatement::execute()

PDO::rollBack

PDO::rollBack -- Rolls back a transaction

Description

bool **PDO::rollBack** (void)

Rolls back the current transaction, as initiated by PDO::beginTransaction(). It is an error to call this method if no transaction is active.

If the database was set to autocommit mode, this function will restore autocommit mode after it has rolled back the transaction.

Some databases, including MySQL, automatically issue an implicit COMMIT when a database definition language (DDL) statement such as DROP TABLE or CREATE TABLE is issued within a transaction. The implicit COMMIT will prevent you from rolling back any other changes within the transaction boundary.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #32 - Roll back a transaction

The following example begins a transaction and issues two statements that modify the database before rolling back the changes. On MySQL, however, the DROP TABLE statement automatically commits the transaction so that none of the changes in the transaction are rolled back.

```
<?php
/* Begin a transaction, turning off autocommit */
$dbh->beginTransaction();

/* Change the database schema and data */
$sth = $dbh->exec("DROP TABLE fruit");
$sth = $dbh->exec("UPDATE dessert
   SET name = 'hamburger'");

/* Recognize mistake and roll back changes */
$dbh->rollBack();

/* Database connection is now back in autocommit mode */
?>
```

See Also

- PDO::beginTransaction()
- PDO::commit()

PDO::setAttribute

PDO::setAttribute -- Set an attribute

Description

bool **PDO::setAttribute** (int `$attribute`, mixed `$value`)

Sets an attribute on the database handle. Some of the available generic attributes are listed below; some drivers may make use of additional driver specific attributes.

- *PDO::ATTR_CASE*: Force column names to a specific case.
  - *PDO::CASE_LOWER*: Force column names to lower case.
  - *PDO::CASE_NATURAL*: Leave column names as returned by the database driver.
  - *PDO::CASE_UPPER*: Force column names to upper case.

- *PDO::ATTR_ERRMODE*: Error reporting.
  - *PDO::ERRMODE_SILENT*: Just set error codes.
  - *PDO::ERRMODE_WARNING*: Raise E_WARNING.
  - *PDO::ERRMODE_EXCEPTION*: Throw exceptions.

- *PDO::ATTR_ORACLE_NULLS* (available with all drivers, not just Oracle): Conversion of NULL and empty strings.
  - *PDO::NULL_NATURAL*: No conversion.
  - *PDO::NULL_EMPTY_STRING*: Empty string is converted to **NULL**.
  - *PDO::NULL_TO_STRING*: NULL is converted to an empty string.

- *PDO::ATTR_STRINGIFY_FETCHES*: Convert numeric values to strings when fetching. Requires bool.
- *PDO::ATTR_STATEMENT_CLASS*: Set user-supplied statement class derived from PDOStatement. Cannot be used with persistent PDO instances. Requires *array(string classname, array(mixed constructor_args))*.
- *PDO::ATTR_AUTOCOMMIT* (available in OCI, Firebird and MySQL): Whether to autocommit every single statement.
- *PDO::MYSQL_ATTR_USE_BUFFERED_QUERY* (available in MySQL): Use buffered queries.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

The PDOStatement class

Introduction

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

Class synopsis

PDOStatement

PDOStatement implements Traversable {

bool **PDOStatement::bindColumn** (mixed `$column`, mixed `&$param` [, int `$type` [, int `$maxlen` [, mixed `$driverdata`]]])

bool **PDOStatement::bindParam** (mixed `$parameter`, mixed `&$variable` [, int `$data_type` [, int `$length` [, mixed `$driver_options`]]])

bool **PDOStatement::bindValue** (mixed `$parameter`, mixed `$value` [, int `$data_type`])

bool **PDOStatement::closeCursor** (void)

int **PDOStatement::columnCount** (void)

string **PDOStatement::errorCode** (void)

array **PDOStatement::errorInfo** (void)

bool **PDOStatement::execute** ([array `$input_parameters`])

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

array **PDOStatement::fetchAll** ([int `$fetch_style` [, int `$column_index` [, array `$ctor_args`]]])

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

mixed **PDOStatement::fetchObject** ([string `$class_name` [, array `$ctor_args`]])

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

```
    array PDOStatement::getColumnMeta ( int $column )

    bool PDOStatement::nextRowset ( void )

    int PDOStatement::rowCount ( void )

    bool PDOStatement::setAttribute ( int $attribute, mixed $value )

    bool PDOStatement::setFetchMode ( int $mode )
}
```

PDOStatement->bindColumn

PDOStatement->bindColumn -- Bind a column to a PHP variable

Description

bool **PDOStatement::bindColumn** (mixed `$column`, mixed `&$param` [, int `$type` [, int `$maxlen` [, mixed `$driverdata`]]])

PDOStatement::bindColumn() arranges to have a particular variable bound to a given column in the result-set from a query. Each call to PDOStatement::fetch() or PDOStatement::fetchAll() will update all the variables that are bound to columns.

Note
Since information about the columns is not always available to PDO until the statement is executed, portable applications should call this function *after* PDOStatement::execute().

Parameters

`column`
: Number of the column (1-indexed) or name of the column in the result set. If using the column name, be aware that the name should match the case of the column, as returned by the driver.

`param`
: Name of the PHP variable to which the column will be bound.

`type`
: Data type of the parameter, specified by the PDO::PARAM_* constants.

`maxlen`
: A hint for pre-allocation.

`driverdata`
: Optional parameter(s) for the driver.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #33 - Binding result set output to PHP variables

Binding columns in the result set to PHP variables is an effective way to make the data contained in each row immediately available to your application. The following example demonstrates how PDO allows you to bind and retrieve columns with a variety of options and with intelligent defaults.

```
<?php
function readData($dbh) {
 $sql = 'SELECT name, colour, calories FROM fruit';
 try {
   $stmt = $dbh->prepare($sql);
   $stmt->execute();

   /* Bind by column number */
   $stmt->bindColumn(1, $name);
   $stmt->bindColumn(2, $colour);

   /* Bind by column name */
   $stmt->bindColumn('calories', $cals);

   while ($row = $stmt->fetch(PDO::FETCH_BOUND)) {
     $data = $name . "\t" . $colour . "\t" . $cals . "\n";
     print $data;
   }
 }
 catch (PDOException $e) {
   print $e->getMessage();
 }
}
readData($dbh);
?>
```

The above example will output:

```
apple   red     150
banana  yellow  175
kiwi    green   75
orange  orange  150
mango   red     200
strawberry      red     25
```

See Also

- PDOStatement::execute()
- PDOStatement::fetch()
- PDOStatement::fetchAll()
- PDOStatement::fetchColumn()

PDOStatement->bindParam

PDOStatement->bindParam -- Binds a parameter to the specified variable name

Description

bool **PDOStatement::bindParam** (mixed `$parameter`, mixed `&$variable` [, int `$data_type` [, int `$length` [, mixed `$driver_options`]]])

Binds a PHP variable to a corresponding named or question mark placeholder in the SQL statement that was use to prepare the statement. Unlike PDOStatement::bindValue(), the variable is bound as a reference and will only be evaluated at the time that PDOStatement::execute() is called.

Most parameters are input parameters, that is, parameters that are used in a read-only fashion to build up the query. Some drivers support the invocation of stored procedures that return data as output parameters, and some also as input/output parameters that both send in data and are updated to receive it.

Parameters

`parameter`
: Parameter identifier. For a prepared statement using named placeholders, this will be a parameter name of the form *:name*. For a prepared statement using question mark placeholders, this will be the 1-indexed position of the parameter.

`variable`
: Name of the PHP variable to bind to the SQL statement parameter.

`data_type`
: Explicit data type for the parameter using the PDO::PARAM_* constants. Defaults to PHP native type. To return an INOUT parameter from a stored procedure, use the bitwise OR operator to set the PDO::PARAM_INPUT_OUTPUT bits for the *`data_type`* parameter.

`length`
: Length of the data type. To indicate that a parameter is an OUT parameter from a stored procedure, you must explicitly set the length.

`driver_options`

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #34 - Execute a prepared statement with named placeholders

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < :calories AND colour = :colour');
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindParam(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

Example #35 - Execute a prepared statement with question mark placeholders

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindParam(2, $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

Example #36 - Call a stored procedure with an INOUT parameter

```
<?php
/* Call a stored procedure with an INOUT parameter */
$colour = 'red';
$sth = $dbh->prepare('CALL puree_fruit(?)');
$sth->bindParam(1, $colour, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 12);
$sth->execute();
print("After pureeing fruit, the colour is: $colour");
?>
```

See Also

- PDO::prepare()
- PDOStatement::execute()
- PDOStatement::bindValue()

PDOStatement->bindValue

PDOStatement->bindValue -- Binds a value to a parameter

Description

bool **PDOStatement::bindValue** (mixed `$parameter`, mixed `$value` [, int `$data_type`])

Binds a value to a corresponding named or question mark placeholder in the SQL statement that was use to prepare the statement.

Parameters

`parameter`
: Parameter identifier. For a prepared statement using named placeholders, this will be a parameter name of the form *:name*. For a prepared statement using question mark placeholders, this will be the 1-indexed position of the parameter.

`value`
: The value to bind to the parameter.

`data_type`
: Explicit data type for the parameter using the PDO::PARAM_* constants. Defaults to PHP native type.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #37 - Execute a prepared statement with named placeholders

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < :calories AND colour = :colour');
$sth->bindValue(':calories', $calories, PDO::PARAM_INT);
$sth->bindValue(':colour', $colour, PDO::PARAM_STR);
$sth->execute();
?>
```

Example #38 - Execute a prepared statement with question mark placeholders

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < ? AND colour = ?');
$sth->bindValue(1, $calories, PDO::PARAM_INT);
$sth->bindValue(2, $colour, PDO::PARAM_STR);
$sth->execute();
?>
```

See Also

- PDO::prepare()
- PDOStatement::execute()
- PDOStatement::bindParam()

PDOStatement->closeCursor

PDOStatement->closeCursor -- Closes the cursor, enabling the statement to be executed again.

Description

bool **PDOStatement::closeCursor** (void)

PDOStatement::closeCursor() frees up the connection to the server so that other SQL statements may be issued, but leaves the statement in a state that enables it to be executed again.

This method is useful for database drivers that do not support executing a PDOStatement object when a previously executed PDOStatement object still has unfetched rows. If your database driver suffers from this limitation, the problem may manifest itself in an out-of-sequence error.

PDOStatement::closeCursor() is implemented either as an optional driver specific method (allowing for maximum efficiency), or as the generic PDO fallback if no driver specific function is installed. The PDO generic fallback is semantically the same as writing the following code in your PHP script:

```
<?php
do {
   while ($stmt->fetch())
       ;
   if (!$stmt->nextRowset())
       break;
} while (true);
?>
```

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #39 - A PDOStatement::closeCursor() example

In the following example, the *$stmt* PDOStatement object returns multiple rows but the application fetches only the first row, leaving the PDOStatement object in a state of having unfetched rows. To ensure that the application will work with all database drivers, the author inserts a call to PDOStatement::closeCursor() on *$stmt* before executing the *$otherStmt* PDOStatement object.

```
<?php
/* Create a PDOStatement object */
```

```
$stmt = $dbh->prepare('SELECT foo FROM bar');

/* Create a second PDOStatement object */
$otherStmt = $dbh->prepare('SELECT foobaz FROM foobar');

/* Execute the first statement */
$stmt->execute();

/* Fetch only the first row from the results */
$stmt->fetch();

/* The following call to closeCursor() may be required by some drivers */
$stmt->closeCursor();

/* Now we can execute the second statement */
$otherStmt->execute();
?>
```

See Also

- PDOStatement::execute()

PDOStatement->columnCount

PDOStatement->columnCount -- Returns the number of columns in the result set

Description

int **PDOStatement::columnCount** (void)

Use PDOStatement::columnCount() to return the number of columns in the result set represented by the PDOStatement object.

If the PDOStatement object was returned from PDO::query(), the column count is immediately available.

If the PDOStatement object was returned from PDO::prepare(), an accurate column count will not be available until you invoke PDOStatement::execute().

Return Values

Returns the number of columns in the result set represented by the PDOStatement object. If there is no result set, PDOStatement::columnCount() returns *0*.

Examples

Example #40 - Counting columns

This example demonstrates how PDOStatement::columnCount() operates with and without a result set.

```
<?php
$dbh = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');

$sth = $dbh->prepare("SELECT name, colour FROM fruit");

/* Count the number of columns in the (non-existent) result set */
$colcount = $sth->columnCount();
print("Before execute(), result set has $colcount columns (should be 0)\n");

$sth->execute();

/* Count the number of columns in the result set */
$colcount = $sth->columnCount();
print("After execute(), result set has $colcount columns (should be 2)\n");

?>
```

The above example will output:

```
Before execute(), result set has 0 columns (should be 0)
```

```
After execute(), result set has 2 columns (should be 2)
```

See Also

- PDO::prepare()
- PDOStatement::execute()
- PDOStatement::rowCount()

PDOStatement->errorCode

PDOStatement->errorCode -- Fetch the SQLSTATE associated with the last operation on the statement handle

Description

string **PDOStatement::errorCode** (void)

Return Values

Identical to PDO::errorCode(), except that PDOStatement::errorCode() only retrieves error codes for operations performed with PDOStatement objects.

Examples

Example #41 - Retrieving a SQLSTATE code

```
<?php
/* Provoke an error -- the BONES table does not exist */
$err = $dbh->prepare('SELECT skull FROM bones');
$err->execute();

echo "\nPDOStatement::errorCode(): ";
print $err->errorCode();
?>
```

The above example will output:

```
PDOStatement::errorCode(): 42S02
```

See Also

- PDO::errorCode()
- PDO::errorInfo()
- PDOStatement::errorInfo()

PDOStatement->errorInfo

PDOStatement->errorInfo -- Fetch extended error information associated with the last operation on the statement handle

Description

array **PDOStatement::errorInfo** (void)

Return Values

PDOStatement::errorInfo() returns an array of error information about the last operation performed by this statement handle. The array consists of the following fields:

Element	Information
0	SQLSTATE error code (a five-character alphanumeric identifier defined in the ANSI SQL standard).
1	Driver-specific error code.
2	Driver-specific error message.

Examples

Example #42 - Displaying errorInfo() fields for a PDO_ODBC connection to a DB2 database

```
<?php
/* Provoke an error -- the BONES table does not exist */
$sth = $dbh->prepare('SELECT skull FROM bones');
$sth->execute();

echo "\nPDOStatement::errorInfo():\n";
$arr = $sth->errorInfo();
print_r($arr);
?>
```

The above example will output:

```
PDOStatement::errorInfo():
Array
(
   [0] => 42S02
   [1] => -204
   [2] => [IBM][CLI Driver][DB2/LINUX] SQL0204N  "DANIELS.BONES" is an
```

```
undefined name.  SQLSTATE=42704
)
```

See Also

- PDO::errorCode()
- PDO::errorInfo()
- PDOStatement::errorCode()

PDOStatement->execute

PDOStatement->execute -- Executes a prepared statement

Description

bool **PDOStatement::execute** ([array `$input_parameters`])

Execute the prepared statement. If the prepared statement included parameter markers, you must either:

- call PDOStatement::bindParam() to bind PHP variables to the parameter markers: bound variables pass their value as input and receive the output value, if any, of their associated parameter markers
- or pass an array of input-only parameter values

Parameters

`input_parameters`
An array of values with as many elements as there are bound parameters in the SQL statement being executed. You cannot bind multiple values to a single parameter; for example, you cannot bind two values to a single named parameter in an IN() clause.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #43 - Execute a prepared statement with bound variables

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < :calories AND colour = :colour');
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindParam(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

Example #44 - Execute a prepared statement with an array of insert values (named parameters)

```
<?php
/* Execute a prepared statement by passing an array of insert values */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < :calories AND colour = :colour');
$sth->execute(array(':calories' => $calories, ':colour' => $colour));
?>
```

Example #45 - Execute a prepared statement with an array of insert values (placeholders)

```
<?php
/* Execute a prepared statement by passing an array of insert values */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < ? AND colour = ?');
$sth->execute(array($calories, $colour));
?>
```

Example #46 - Execute a prepared statement with question mark placeholders

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
   FROM fruit
   WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindParam(2, $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

See Also

- PDO::prepare()
- PDOStatement::bindParam()
- PDOStatement::fetch()
- PDOStatement::fetchAll()
- PDOStatement::fetchColumn()

PDOStatement->fetch

PDOStatement->fetch -- Fetches the next row from a result set

Description

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

Fetches a row from a result set associated with a PDOStatement object. The *`fetch_style`* parameter determines how PDO returns the row.

Parameters

`fetch_style`
Controls how the next row will be returned to the caller. This value must be one of the *PDO::FETCH_** constants, defaulting to *PDO::FETCH_BOTH*.

- *PDO::FETCH_ASSOC*: returns an array indexed by column name as returned in your result set
- *PDO::FETCH_BOTH* (default): returns an array indexed by both column name and 0-indexed column number as returned in your result set
- *PDO::FETCH_BOUND*: returns **TRUE** and assigns the values of the columns in your result set to the PHP variables to which they were bound with the PDOStatement::bindColumn() method
- *PDO::FETCH_CLASS*: returns a new instance of the requested class, mapping the columns of the result set to named properties in the class. If *`fetch_style`* includes PDO::FETCH_CLASSTYPE (e.g. *PDO::FETCH_CLASS* | *PDO::FETCH_CLASSTYPE*) then the name of the class is determined from a value of the first column.
- *PDO::FETCH_INTO*: updates an existing instance of the requested class, mapping the columns of the result set to named properties in the class
- *PDO::FETCH_LAZY*: combines *PDO::FETCH_BOTH* and *PDO::FETCH_OBJ*, creating the object variable names as they are accessed
- *PDO::FETCH_NUM*: returns an array indexed by column number as returned in your result set, starting at column 0
- *PDO::FETCH_OBJ*: returns an anonymous object with property names that correspond to the column names returned in your result set

`cursor_orientation`
For a PDOStatement object representing a scrollable cursor, this value determines which row will be returned to the caller. This value must be one of the *PDO::FETCH_ORI_** constants, defaulting to *PDO::FETCH_ORI_NEXT*. To request a

scrollable cursor for your PDOStatement object, you must set the *PDO::ATTR_CURSOR* attribute to *PDO::CURSOR_SCROLL* when you prepare the SQL statement with PDO::prepare().

offset
: For a PDOStatement object representing a scrollable cursor for which the *cursor_orientation* parameter is set to *PDO::FETCH_ORI_ABS*, this value specifies the absolute number of the row in the result set that shall be fetched. For a PDOStatement object representing a scrollable cursor for which the *cursor_orientation* parameter is set to *PDO::FETCH_ORI_REL*, this value specifies the row to fetch relative to the cursor position before PDOStatement::fetch() was called.

Return Values

The return value of this function on success depends on the fetch type. In all cases, **FALSE** is returned on failure.

Examples

Example #47 - Fetching rows using different fetch styles

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Exercise PDOStatement::fetch styles */
print("PDO::FETCH_ASSOC: ");
print("Return next row as an array indexed by column name\n");
$result = $sth->fetch(PDO::FETCH_ASSOC);
print_r($result);
print("\n");

print("PDO::FETCH_BOTH: ");
print("Return next row as an array indexed by both column name and
number\n");
$result = $sth->fetch(PDO::FETCH_BOTH);
print_r($result);
print("\n");

print("PDO::FETCH_LAZY: ");
print("Return next row as an anonymous object with column names as
properties\n");
$result = $sth->fetch(PDO::FETCH_LAZY);
print_r($result);
print("\n");

print("PDO::FETCH_OBJ: ");
print("Return next row as an anonymous object with column names as
properties\n");
$result = $sth->fetch(PDO::FETCH_OBJ);
print $result->NAME;
print("\n");
?>
```

The above example will output:

```
PDO::FETCH_ASSOC: Return next row as an array indexed by column name
Array
(
   [NAME] => apple
   [COLOUR] => red
)

PDO::FETCH_BOTH: Return next row as an array indexed by both column name and
number
Array
(
   [NAME] => banana
   [0] => banana
   [COLOUR] => yellow
   [1] => yellow
)

PDO::FETCH_LAZY: Return next row as an anonymous object with column names as
properties
PDORow Object
(
   [NAME] => orange
   [COLOUR] => orange
)

PDO::FETCH_OBJ: Return next row as an anonymous object with column names as
properties
kiwi
```

Example #48 - Fetching rows with a scrollable cursor

```
<?php
function readDataForwards($dbh) {
 $sql = 'SELECT hand, won, bet FROM mynumbers ORDER BY BET';
 try {
   $stmt = $dbh->prepare($sql, array(PDO::ATTR_CURSOR =>
PDO::CURSOR_SCROLL));
   $stmt->execute();
   while ($row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_NEXT)) {
     $data = $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
     print $data;
   }
   $stmt = null;
 }
 catch (PDOException $e) {
   print $e->getMessage();
 }
}
function readDataBackwards($dbh) {
 $sql = 'SELECT hand, won, bet FROM mynumbers ORDER BY bet';
 try {
   $stmt = $dbh->prepare($sql, array(PDO::ATTR_CURSOR =>
PDO::CURSOR_SCROLL));
   $stmt->execute();
   $row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_LAST);
```

```
    do {
      $data = $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
      print $data;
    } while ($row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_PRIOR));
    $stmt = null;
  }
  catch (PDOException $e) {
    print $e->getMessage();
  }
}

print "Reading forwards:\n";
readDataForwards($conn);

print "Reading backwards:\n";
readDataBackwards($conn);
?>
```

The above example will output:

```
Reading forwards:
21    10    5
16    0     5
19    20    10

Reading backwards:
19    20    10
16    0     5
21    10    5
```

See Also

- PDO::prepare()
- PDOStatement::execute()
- PDOStatement::fetchAll()
- PDOStatement::fetchColumn()
- PDOStatement::fetchObject()
- PDOStatement::setFetchMode()

PDOStatement->fetchAll

PDOStatement->fetchAll -- Returns an array containing all of the result set rows

Description

array **PDOStatement::fetchAll** ([int `$fetch_style` [, int `$column_index` [, array `$ctor_args`]]])

Parameters

`fetch_style`
: Controls the contents of the returned array as documented in PDOStatement::fetch(). Defaults to *PDO::FETCH_BOTH*. To return an array consisting of all values of a single column from the result set, specify *PDO::FETCH_COLUMN*. You can specify which column you want with the *`column-index`* parameter. To fetch only the unique values of a single column from the result set, bitwise-OR *PDO::FETCH_COLUMN* with *PDO::FETCH_UNIQUE*. To return an associative array grouped by the values of a specified column, bitwise-OR *PDO::FETCH_COLUMN* with *PDO::FETCH_GROUP*.

`column_index`
: Returns the indicated 0-indexed column when the value of *`fetch_style`* is *PDO::FETCH_COLUMN*. Defaults to *0*.

`ctor_args`
: Arguments of custom class constructor.

Return Values

PDOStatement::fetchAll() returns an array containing all of the remaining rows in the result set. The array represents each row as either an array of column values or an object with properties corresponding to each column name.

Using this method to fetch large result sets will result in a heavy demand on system and possibly network resources. Rather than retrieving all of the data and manipulating it in PHP, consider using the database server to manipulate the result sets. For example, use the WHERE and SORT BY clauses in SQL to restrict results before retrieving and processing them with PHP.

Examples

Example #49 - Fetch all remaining rows in a result set

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
```

```
$sth->execute();

/* Fetch all of the remaining rows in the result set */
print("Fetch all of the remaining rows in the result set:\n");
$result = $sth->fetchAll();
print_r($result);
?>
```

The above example will output:

```
Fetch all of the remaining rows in the result set:
Array
(
   [0] => Array
       (
           [NAME] => pear
           [0] => pear
           [COLOUR] => green
           [1] => green
       )

   [1] => Array
       (
           [NAME] => watermelon
           [0] => watermelon
           [COLOUR] => pink
           [1] => pink
       )

)
```

Example #50 - Fetching all values of a single column from a result set

The following example demonstrates how to return all of the values of a single column from a result set, even though the SQL statement itself may return multiple columns per row.

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Fetch all of the values of the first column */
$result = $sth->fetchAll(PDO::FETCH_COLUMN, 0);
var_dump($result);
?>
```

The above example will output:

```
Array(3)
(
   [0] =>
   string(5) => apple
   [1] =>
   string(4) => pear
   [2] =>
   string(10) => watermelon
```

```
)
```

Example #51 - Grouping all values by a single column

The following example demonstrates how to return an associative array grouped by the values of the specified column in the result set. The array contains three keys: values *apple* and *pear* are returned as arrays that contain two different colours, while *watermelon* is returned as an array that contains only one colour.

```
<?php
$insert = $dbh->prepare("INSERT INTO fruit(name, colour) VALUES (?, ?)");
$insert->execute('apple', 'green');
$insert->execute('pear', 'yellow');

$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Group values by the first column */
var_dump($sth->fetchAll(PDO::FETCH_COLUMN|PDO::FETCH_GROUP));
?>
```

The above example will output:

```
array(3) {
 ["apple"]=>
 array(2) {
   [0]=>
   string(5) "green"
   [1]=>
   string(3) "red"
 }
 ["pear"]=>
 array(2) {
   [0]=>
   string(5) "green"
   [1]=>
   string(6) "yellow"
 }
 ["watermelon"]=>
 array(1) {
   [0]=>
   string(5) "green"
 }
}
```

See Also

- PDO::query()
- PDOStatement::fetch()
- PDOStatement::fetchColumn()
- PDO::prepare()

- PDOStatement::setFetchMode()

PDOStatement->fetchColumn

PDOStatement->fetchColumn -- Returns a single column from the next row of a result set

Description

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

Returns a single column from the next row of a result set or **FALSE** if there are no more rows.

Parameters

`column_number`
: 0-indexed number of the column you wish to retrieve from the row. If no value is supplied, PDOStatement::fetchColumn() fetches the first column.

Return Values

PDOStatement::fetchColumn() returns a single column in the next row of a result set.

Warning
There is no way to return another column from the same row if you use PDOStatement::fetchColumn() to retrieve data.

Examples

Example #52 - Return first column of the next row

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Fetch the first column from the next row in the result set */
print("Fetch the first column from the next row in the result set:\n");
$result = $sth->fetchColumn();
print("name = $result\n");

print("Fetch the second column from the next row in the result set:\n");
$result = $sth->fetchColumn(1);
print("colour = $result\n");
?>
```

The above example will output:

```
Fetch the first column from the next row in the result set:
name = lemon
Fetch the second column from the next row in the result set:
colour = red
```

See Also

- PDO::query()
- PDOStatement::fetch()
- PDOStatement::fetchAll()
- PDO::prepare()
- PDOStatement::setFetchMode()

PDOStatement->fetchObject

PDOStatement->fetchObject -- Fetches the next row and returns it as an object.

Description

mixed **PDOStatement::fetchObject** ([string `$class_name` [, array `$ctor_args`]])

Fetches the next row and returns it as an object. This function is an alternative to PDOStatement::fetch() with **PDO::FETCH_CLASS** or **PDO::FETCH_OBJ** style.

Parameters

`class_name`
: Name of the created class, defaults to *stdClass*.

`ctor_args`
: Elements of this array are passed to the constructor.

Return Values

Returns an instance of the required class with property names that correspond to the column names or **FALSE** in case of an error.

See Also

- PDOStatement::fetch()

PDOStatement->getAttribute

PDOStatement->getAttribute -- Retrieve a statement attribute

Description

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

Gets an attribute of the statement. Currently, no generic attributes exist but only driver specific:

- *PDO::ATTR_CURSOR_NAME* (Firebird and ODBC specific): Get the name of cursor for *UPDATE ... WHERE CURRENT OF*.

Return Values

Returns the attribute value.

See Also

- PDO::getAttribute()
- PDO::setAttribute()
- PDOStatement::setAttribute()

PDOStatement->getColumnMeta

PDOStatement->getColumnMeta -- Returns metadata for a column in a result set

Description

array **PDOStatement::getColumnMeta** (int `$column`)

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Retrieves the metadata for a 0-indexed column in a result set as an associative array.

Warning
Not all PDO drivers support PDOStatement::getColumnMeta().

Parameters

`column`
: The 0-indexed column in the result set.

Return Values

Returns an associative array containing the following values representing the metadata for a single column:

Column metadata

Name	Value
native_type	The PHP native type used to represent the column value.
driver:decl_type	The SQL type used to represent the column value in the database. If the column in the result set is the result of a function, this value is not returned by PDOStatement::getColumnMeta().

flags	Any flags set for this column.
name	The name of this column as returned by the database.
table	The name of this column's table as returned by the database.
len	The length of this column. Normally *-1* for types other than floating point decimals.
precision	The numeric precision of this column. Normally *0* for types other than floating point decimals.
pdo_type	The type of this column as represented by the *PDO::PARAM_** constants.

Returns **FALSE** if the requested column does not exist in the result set, or if no result set exists.

ChangeLog

Version	Description
5.2.3	*table* field

Examples

Example #53 - Retrieving column metadata

The following example shows the results of retrieving the metadata for a single column generated by a function (COUNT) in a PDO_SQLITE driver.

```
<?php
$select = $DB->query('SELECT COUNT(*) FROM fruit');
$meta = $select->getColumnMeta(0);
var_dump($meta);
?>
```

The above example will output:

```
array(6) {
 ["native_type"]=>
 string(7) "integer"
 ["flags"]=>
```

```
  array(0) {
  }
  ["name"]=>
  string(8) "COUNT(*)"
  ["len"]=>
  int(-1)
  ["precision"]=>
  int(0)
  ["pdo_type"]=>
  int(2)
}
```

See Also

- PDOStatement::columnCount()
- PDOStatement::rowCount()

PDOStatement->nextRowset

PDOStatement->nextRowset -- Advances to the next rowset in a multi-rowset statement handle

Description

bool **PDOStatement::nextRowset** (void)

Some database servers support stored procedures that return more than one rowset (also known as a result set). PDOStatement::nextRowset() enables you to access the second and subsequent rowsets associated with a PDOStatement object. Each rowset can have a different set of columns from the preceding rowset.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #54 - Fetching multiple rowsets returned from a stored procedure

The following example shows how to call a stored procedure, MULTIPLE_ROWSETS, that returns three rowsets. We use a do / while loop to loop over the PDOStatement::nextRowset() method, which returns false and terminates the loop when no more rowsets can be returned.

```
<?php
$sql = 'CALL multiple_rowsets()';
$stmt = $conn->query($sql);
$i = 1;
do {
   $rowset = $stmt->fetch(PDO::FETCH_NUM);
   if ($rowset) {
       printResultSet($rowset, $i);
   }
   $i++;
} while ($stmt->nextRowset());

function printResultSet(&$rowset, $i) {
   print "Result set $i:\n";
   foreach ($rowset as $row) {
       foreach ($row as $col) {
           print $col . "\t";
       }
       print "\n";
   }
   print "\n";
}
?>
```

The above example will output:

```
Result set 1:
apple    red
banana   yellow

Result set 2:
orange   orange    150
banana   yellow    175

Result set 3:
lime     green
apple    red
banana   yellow
```

See Also

- PDOStatement::columnCount()
- PDOStatement::execute()
- PDOStatement::getColumnMeta()
- PDO::query()

PDOStatement->rowCount

PDOStatement->rowCount -- Returns the number of rows affected by the last SQL statement

Description

int **PDOStatement::rowCount** (void)

PDOStatement::rowCount() returns the number of rows affected by the last DELETE, INSERT, or UPDATE statement executed by the corresponding *PDOStatement* object.

If the last SQL statement executed by the associated *PDOStatement* was a SELECT statement, some databases may return the number of rows returned by that statement. However, this behaviour is not guaranteed for all databases and should not be relied on for portable applications.

Return Values

Returns the number of rows.

Examples

Example #55 - Return the number of deleted rows

PDOStatement::rowCount() returns the number of rows affected by a DELETE, INSERT, or UPDATE statement.

```
<?php
/* Delete all rows from the FRUIT table */
$del = $dbh->prepare('DELETE FROM fruit');
$del->execute();

/* Return number of rows that were deleted */
print("Return number of rows that were deleted:\n");
$count = $del->rowCount();
print("Deleted $count rows.\n");
?>
```

The above example will output:

```
Deleted 9 rows.
```

Example #56 - Counting rows returned by a SELECT statement

For most databases, PDOStatement::rowCount() does not return the number of rows affected by a SELECT statement. Instead, use PDO::query() to issue a SELECT

COUNT(*) statement with the same predicates as your intended SELECT statement, then use PDOStatement::fetchColumn() to retrieve the number of rows that will be returned. Your application can then perform the correct action.

```
<?php
$sql = "SELECT COUNT(*) FROM fruit WHERE calories > 100";
if ($res = $conn->query($sql)) {

   /* Check the number of rows that match the SELECT statement */
 if ($res->fetchColumn() > 0) {

       /* Issue the real SELECT statement and work with the results */
        $sql = "SELECT name FROM fruit WHERE calories > 100";
      foreach ($conn->query($sql) as $row) {
          print "Name: " .  $row['NAME'] . "\n";
        }
   }
   /* No rows matched -- do something else */
 else {
     print "No rows matched the query.";
   }
}

$res = null;
$conn = null;
?>
```

The above example will output:

```
apple
banana
orange
pear
```

See Also

- PDOStatement::columnCount()
- PDOStatement::fetchColumn()
- PDO::query()

PDOStatement->setAttribute

PDOStatement->setAttribute -- Set a statement attribute

Description

bool **PDOStatement::setAttribute** (int `$attribute`, mixed `$value`)

Sets an attribute on the statement. Currently, no generic attributes are set but only driver specific:

- *PDO::ATTR_CURSOR_NAME* (Firebird and ODBC specific): Set the name of cursor for *UPDATE ... WHERE CURRENT OF*.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

See Also

- PDO::getAttribute()
- PDO::setAttribute()
- PDOStatement::getAttribute()

PDOStatement->setFetchMode

PDOStatement->setFetchMode -- Set the default fetch mode for this statement

Description

bool **PDOStatement::setFetchMode** (int `$mode`)

bool **PDOStatement::setFetchMode** (int `$PDO::FETCH_COLUMN`, int `$colno`)

bool **PDOStatement::setFetchMode** (int `$PDO::FETCH_CLASS`, string `$classname`, array `$ctorargs`)

bool **PDOStatement::setFetchMode** (int `$PDO::FETCH_INTO`, object `$object`)

Parameters

`mode`
The fetch mode must be one of the *PDO::FETCH_** constants.

Return Values

Returns *1* on success or **FALSE** on failure.

Examples

Example #57 - Setting the fetch mode

The following example demonstrates how PDOStatement::setFetchMode() changes the default fetch mode for a PDOStatement object.

```
<?php
$sql = 'SELECT name, colour, calories FROM fruit';
try {
 $stmt = $dbh->query($sql);
 $result = $stmt->setFetchMode(PDO::FETCH_NUM);
 while ($row = $stmt->fetch()) {
   print $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
 }
}
catch (PDOException $e) {
 print $e->getMessage();
}
?>
```

The above example will output:

```
apple   red     150
banana  yellow  250
orange  orange  300
kiwi    brown   75
lemon   yellow  25
pear    green   150
watermelon      pink    90
```

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_IBM	IBM DB2
PDO_INFORMIX	IBM Informix Dynamic Server
PDO_MYSQL	MySQL 3.x/4.x/5.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

Microsoft SQL Server and Sybase Functions (PDO_DBLIB)

Introduction

Warning
This extension is *EXPERIMENTAL*. The behaviour of this extension?including the names of its functions and any other documentation surrounding this extension?may change without notice in a future release of PHP. This extension should be used at your own risk.

PDO_DBLIB is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to Microsoft SQL Server and Sybase databases through the FreeTDS libary.

On Windows, you should use the PDO_ODBC driver to connect to Microsoft SQL Server and Sybase databases, as the native Windows DB-LIB is ancient, thread un-safe and no longer supported by Microsoft.

PDO_DBLIB DSN

PDO_DBLIB DSN -- Connecting to Microsoft SQL Server and Sybase databases

Description

The PDO_DBLIB Data Source Name (DSN) is composed of the following elements:
DSN prefix
: The DSN prefix is **sybase:** if PDO_DBLIB was linked against the FreeTDS libraries, **mssql:** if PDO_DBLIB was linked against the Microsoft SQL Server libraries, or **dblib:** if linked against any other variety of DB-lib.

host
: The hostname on which the database server resides.

dbname
: The name of the database.

Examples

Example #58 - PDO_DBLIB DSN examples

The following examples show a PDO_DBLIB DSN for connecting to Microsoft SQL Server and Sybase databases:

```
mssql:host=localhost;dbname=testdb
sybase:host=localhost;dbname=testdb
dblib:host=localhost;dbname=testdb
```

Firebird/Interbase Functions (PDO_FIREBIRD)

Introduction

Warning
This extension is *EXPERIMENTAL*. The behaviour of this extension?including the names of its functions and any other documentation surrounding this extension?may change without notice in a future release of PHP. This extension should be used at your own risk.

PDO_FIREBIRD is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to Firebird and Interbase databases.

PDO_FIREBIRD DSN

PDO_FIREBIRD DSN -- Connecting to Firebird and Interbase databases

Description

The PDO_FIREBIRD Data Source Name (DSN) is composed of the following elements:
DSN prefix
The DSN prefix is **firebird:**.

DataSource
The hostname on which the database server resides.

Port
The port number for the server on which the database is running.

Database
The name of the database.

User
The name of the user that will connect to the database.

Password
The password for the user.

Examples

Example #59 - PDO_FIREBIRD DSN examples

The following example shows a PDO_FIREBIRD DSN for connecting to Firebird and Interbase databases:

```
firebird:User=john;Password=mypass;Database=DATABASE.GDE;DataSource=localhost;Port=3050
```

IBM Functions (PDO_IBM)

Introduction

PDO_IBM is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to IBM databases.

Installation

To build the PDO_IBM extension, the DB2 Client v9.1 or later must be installed on the same system as PHP. The DB2 Client can be downloaded from the IBM » Application Development Site.

Note
Note The DB2 Client v9.1 or later supports direct access to DB2 for Linux, UNIX, and Windows v8 and v9.1 servers. The DB2 Client v9.1 also supports access to DB2 UDB for i5 and DB2 UDB for z/OS servers using the separately purchased » DB2 Connect product.

PDO_IBM is a » PECL extension, so follow the instructions in Installation of PECL extensions to install the PDO_IBM extension. Issue the *configure* command to point to the location of your DB2 Client header files and libraries as follows:

```
bash$ ./configure --with-pdo-ibm=/path/to/sqllib[,shared]
```

The *configure* command defaults to the value of the *DB2DIR* environment variable.

PDO_IBM DSN

PDO_IBM DSN -- Connecting to IBM databases

Description

The PDO_IBM Data Source Name (DSN) is based on the IBM CLI DSN. The major components of the PDO_IBM DSN are:

DSN prefix

The DSN prefix is **ibm:**.

DSN

The DSN can be any of the following:

- a) Data source setup using *db2cli.ini* or *odbc.ini*
- b) Catalogued database name i.e. database alias in the DB2 client catalog
- c) Complete connection string in the following format: `DRIVER={IBM DB2 ODBC DRIVER};DATABASE=` *`database`*`;HOSTNAME=` *`hostname`*`;PORT=` *`port`* `;PROTOCOL=TCPIP;UID=` *`username`*`;PWD=` *`password`*`;` where the parameters represent the following values:

  `database`

  The name of the database.

  `hostname`

  The hostname or IP address of the database server.

  `port`

  The TCP/IP port on which the database is listening for requests.

  `username`

  The username with which you are connecting to the database.

  `password`

  The password with which you are connecting to the database.

Examples

Example #60 - PDO_IBM DSN example using *db2cli.ini*

The following example shows a PDO_IBM DSN for connecting to an DB2 database cataloged as DB2_9 in *db2cli.ini*:

```
$db = new PDO("ibm:DSN=DB2_9", "", "");
```

```
[DB2_9]
Database=testdb
Protocol=tcpip
Hostname=11.22.33.444
Servicename=56789
```

Example #61 - PDO_IBM DSN example using a connection string

The following example shows a PDO_IBM DSN for connecting to an DB2 database named **testdb** using the DB2 CLI connection string syntax.

```
$db = new PDO("ibm:DRIVER={IBM DB2 ODBC DRIVER};DATABASE=testdb;" .
 "HOSTNAME=11.22.33.444;PORT=56789;PROTOCOL=TCPIP;", "testuser", "tespass");
```

Informix Functions (PDO_INFORMIX)

Introduction

PDO_INFORMIX is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to Informix databases.

Installation

To build the PDO_INFORMIX extension, the Informix Client SDK 2.81 UC1 or higher must be installed on the same system as PHP. The Informix Client SDK is available from the » IBM Informix Support Site.

PDO_INFORMIX is a » PECL extension, so follow the instructions in Installation of PECL extensions to install the PDO_INFORMIX extension. Issue the *configure* command to point to the location of your Informix Client SDK header files and libraries as follows:

```
bash$ ./configure --with-pdo-informix=/path/to/SDK[,shared]
```

The *configure* command defaults to the value of the *INFORMIXDIR* environment variable.

Scrollable cursors

PDO_INFORMIX supports scrollable cursors; however, they are not enabled by default. To enable scrollable cursor support, you must either set **ENABLESCROLLABLECURSORS=1** in the corresponding ODBC connection settings in *odbc.ini* or pass the **EnableScrollableCursors=1** clause in the DSN connection string.

PDO_INFORMIX DSN

PDO_INFORMIX DSN -- Connecting to Informix databases

Description

The PDO_INFORMIX Data Source Name (DSN) is based on the Informix ODBC DSN string. Details on configuring an Informix ODBC DSN are available from the » Informix Dynamic Server Information Center. The major components of the PDO_INFORMIX DSN are:

DSN prefix
: The DSN prefix is **informix:**.

DSN
: The DSN can be either a data source setup using *odbc.ini* or a complete » connection string.

Examples

Example #62 - PDO_INFORMIX DSN example using *odbc.ini*

The following example shows a PDO_INFORMIX DSN for connecting to an Informix database cataloged as Infdrv33 in *odbc.ini*:

```
$db = new PDO("informix:DSN=Infdrv33", "", "");

[ODBC Data Sources]
Infdrv33=INFORMIX 3.3 32-BIT

[Infdrv33]
Driver=/opt/informix/csdk_2.81.UC1G2/lib/cli/iclis09b.so
Description=INFORMIX 3.3 32-BIT
Database=common_db
LogonID=testuser
pwd=testpass
Servername=ids_server
DB_LOCALE=en_US.819
OPTIMIZEAUTOCOMMIT=1
ENABLESCROLLABLECURSORS=1
```

Example #63 - PDO_INFORMIX DSN example using a connection string

The following example shows a PDO_INFORMIX DSN for connecting to an Informix database named **common_db** using the Informix connection string syntax.

```
$db = new PDO("informix:host=host.domain.com; service=9800;
   database=common_db; server=ids_server; protocol=onsoctcp;
   EnableScrollableCursors=1", "testuser", "tespass");
```

MySQL Functions (PDO_MYSQL)

Introduction

PDO_MYSQL is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to MySQL 3.x, 4.x and 5.x databases.

PDO_MYSQL will take advantage of native prepared statement support present in MySQL 4.1 and higher. If you're using an older version of the mysql client libraries, PDO will emulate them for you.

Warning

Beware: Some MySQL table types (storage engines) do not support transactions. When writing transactional database code using a table type that does not support transactions, MySQL will pretend that a transaction was initiated successfully. In addition, any DDL queries issued will implicitly commit any pending transactions.

Predefined Constants

The constants below are defined by this driver, and will only be available when the extension has been either compiled into PHP or dynamically loaded at runtime. In addition, these driver-specific constants should only be used if you are using this driver. Using mysql-specific attributes with the postgres driver may result in unexpected behaviour. PDO::getAttribute() may be used to obtain the **PDO_ATTR_DRIVER_NAME** attribute to check the driver, if your code can run against multiple drivers.

PDO::MYSQL_ATTR_USE_BUFFERED_QUERY (integer)
If this attribute is set to **TRUE** on a PDOStatement, the MySQL driver will use the buffered versions of the MySQL API. If you're writing portable code, you should use PDOStatement::fetchAll() instead.

Example #64 - Forcing queries to be buffered in mysql

```
<?php
if ($db->getAttribute(PDO::ATTR_DRIVER_NAME) == 'mysql') {
   $stmt = $db->prepare('select * from foo',
       array(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true));
} else {
   die("my application only works with mysql; I should use
\$stmt->fetchAll() instead");
}
?>
```

PDO::MYSQL_ATTR_LOCAL_INFILE (integer)
Enable *LOAD LOCAL INFILE*.

PDO::MYSQL_ATTR_INIT_COMMAND (integer)
Command to execute when connecting to the MySQL server. Will automatically be re-executed when reconnecting.

PDO::MYSQL_ATTR_READ_DEFAULT_FILE (integer)
Read options from the named option file instead of from *my.cnf*.

PDO::MYSQL_ATTR_READ_DEFAULT_GROUP (integer)
Read options from the named group from *my.cnf* or the file specified with **MYSQL_READ_DEFAULT_FILE**.

PDO::MYSQL_ATTR_MAX_BUFFER_SIZE (integer)
Maximum buffer size. Defaults to 1 MiB.

PDO::MYSQL_ATTR_DIRECT_QUERY (integer)
Perform direct queries, don't use prepared statements.

PDO_MYSQL DSN

PDO_MYSQL DSN -- Connecting to MySQL databases

Description

The PDO_MYSQL Data Source Name (DSN) is composed of the following elements:
DSN prefix
The DSN prefix is **mysql:**.

host
The hostname on which the database server resides.

port
The port number where the database server is listening.

dbname
The name of the database.

unix_socket
The MySQL Unix socket (shouldn't be used with *host* or *port*).

Examples

Example #65 - PDO_MYSQL DSN examples

The following example shows a PDO_MYSQL DSN for connecting to MySQL databases:

```
mysql:host=localhost;dbname=testdb
```

More complete examples:

```
mysql:host=localhost;port=3307;dbname=testdb
mysql:unix_socket=/tmp/mysql.sock;dbname=testdb
```

Oracle Functions (PDO_OCI)

Introduction

Warning
This extension is *EXPERIMENTAL*. The behaviour of this extension?including the names of its functions and any other documentation surrounding this extension?may change without notice in a future release of PHP. This extension should be used at your own risk.

PDO_OCI is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to Oracle databases through the OCI library.

PDO_OCI DSN

PDO_OCI DSN -- Connecting to Oracle databases

Description

The PDO_OCI Data Source Name (DSN) is composed of the following elements:
DSN prefix
The DSN prefix is **oci:**.

dbname (Oracle Instant Client)
The URI for the Oracle Instant Client connection takes the form of `dbname=//` *hostname*`:` *port-number* `/` *database*. If you are connecting to a database defined in *tnsnames.ora*, use only the name of the database: `dbname=` *database*.

charset
The client-side character set for the current environment handle.

Examples

Example #66 - PDO_OCI DSN examples

The following examples show a PDO_OCI DSN for connecting to Oracle databases:

```
// Connect to a database defined in tnsnames.ora
oci:dbname=mydb

// Connect using the Oracle Instant Client
oci:dbname=//localhost:1521/mydb
```

ODBC and DB2 Functions (PDO_ODBC)

Introduction

PDO_ODBC is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to databases through ODBC drivers or through the IBM DB2 Call Level Interface (DB2 CLI) library. PDO_ODBC currently supports three different "flavours" of database drivers:

ibm-db2
: Supports access to IBM DB2 Universal Database, Cloudscape, and Apache Derby servers through the free DB2 client.

unixODBC
: Supports access to database servers through the unixODBC driver manager and the database's own ODBC drivers.

generic
: Offers a compile option for ODBC driver managers that are not explicitly supported by PDO_ODBC.

On Windows, PDO_ODBC is built into the PHP core by default. It is linked against the Windows ODBC Driver Manager so that PHP can connect to any database cataloged as a System DSN, and is the recommended driver for connecting to Microsoft SQL Server databases.

Installation

PDO_ODBC on UNIX systems

1. As of PHP 5.1, PDO_ODBC is included in the PHP source. You can compile the PDO_ODBC extension as either a static or shared module using the following *configure* commands.

   ibm_db2

   ```
   ./configure --with-pdo-odbc=ibm-db2,/opt/IBM/db2/V8.1/
   ```

   To build PDO_ODBC with the ibm-db2 flavour, you have to have previously installed the DB2 application development headers on the same machine on which you are compiling PDO_ODBC. The DB2 application development headers are an installable option in the DB2 servers, and are also available as part of the DB2 Application Development Client freely available for download from the IBM DB2 Universal Database » support site. If you do not supply a location for the DB2 libraries and headers to the *configure* command, PDO_ODBC defaults to */home/db2inst1/sqllib*.

   unixODBC

```
./configure --with-pdo-odbc=unixODBC,/usr/local
```

If you do not supply a location for the unixODBC libraries and headers to the *configure* command, PDO_ODBC defaults to */usr/local*.

generic

```
./configure --with-pdo-odbc=generic,/usr/local,libname,ldflags,cflags
```

Runtime Configuration

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

PDO_ODBC Configuration Options

Name	Default	Changeable	Changelog
pdo_odbc.connection_pooling	"strict"	PHP_INI_ALL	Available since PHP 5.1.0.
pdo_odbc.db2_instance_name	NULL	PHP_INI_SYSTEM	Available since PHP 5.1.1. Removed in PHP 6.0.0.

For further details and definitions of the PHP_INI_* constants, see the php.ini directives.

Here's a short explanation of the configuration directives.

`pdo_odbc.connection_pooling` string

Whether to pool ODBC connections. Can be one of "strict", "relaxed" or "off" (equals to ""). The parameter describes how strict the connection manager should be when matching connection parameters to existing pooled connections. **strict** is the recommend default, and will result in the use of cached connections only when all the connection parameters match exactly. **relaxed** will result in the use of cached connections when similar connection parameters are used. This can result in increased use of the cache, at the risk of bleeding connection information between (for example) virtual hosts. This setting can only be changed from the *php.ini* file, and affects the entire process; any other modules loaded into the process that use the same ODBC libraries will be affected too, including the Unified ODBC extension.

Warning
relaxed matching should not be used on a shared server, for security reasons.

Tip
Leave this setting at the default **strict** setting unless you have good reason to change it.

pdo_odbc.db2_instance_name string

If you compile PDO_ODBC using the *db2* flavour, this setting sets the value of the DB2INSTANCE environment variable on Linux and UNIX operating systems to the specified name of the DB2 instance. This enables PDO_ODBC to resolve the location of the DB2 libraries and make cataloged connections to DB2 databases. This setting can only be changed from the *php.ini* file, and affects the entire process; any other modules loaded into the process that use the same ODBC libraries will be affected too, including the Unified ODBC extension. This setting has no effect on Windows.

PDO_ODBC DSN

PDO_ODBC DSN -- Connecting to ODBC or DB2 databases

Description

The PDO_ODBC Data Source Name (DSN) is composed of the following elements:

DSN prefix
: The DSN prefix is **odbc:**. If you are connecting to a database cataloged in the ODBC driver manager or the DB2 catalog, you can append the cataloged name of the database to the DSN.

DSN
: The name of the database as cataloged in the ODBC driver manager or the DB2 catalog. Alternately, you can provide a complete ODBC connection string to connect to a database as described at » http://www.connectionstrings.com/.

UID
: The name of the user for the connection. If you specify the user name in the DSN, PDO ignores the value of the user name argument in the PDO constructor.

PWD
: The password of the user for the connection. If you specify the password in the DSN, PDO ignores the value of the password argument in the PDO constructor.

Examples

Example #67 - PDO_ODBC DSN example (ODBC driver manager)

The following example shows a PDO_ODBC DSN for connecting to an ODBC database cataloged as testdb in the ODBC driver manager:

```
odbc:testdb
```

Example #68 - PDO_ODBC DSN example (IBM DB2 uncataloged connection)

The following example shows a PDO_ODBC DSN for connecting to an IBM DB2 database named **SAMPLE** using the full ODBC DSN syntax:

```
odbc:DRIVER={IBM DB2 ODBC
DRIVER};HOSTNAME=localhost;PORT=50000;DATABASE=SAMPLE;PROTOCOL=TCPIP;UID=db2
inst1;PWD=ibmdb2;
```

Example #69 - PDO_ODBC DSN example (Microsoft Access uncataloged connection)

The following example shows a PDO_ODBC DSN for connecting to a Microsoft Access database stored at **C:\db.mdb** using the full ODBC DSN syntax:

```
odbc:Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\\db.mdb;Uid=Admin
```

PostgreSQL Functions (PDO_PGSQL)

Introduction

PDO_PGSQL is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to PostgreSQL databases.

Resource Types

This extension defines a stream resource returned by PDO::pgsqlLOBOpen().

PDO_PGSQL DSN

PDO_PGSQL DSN -- Connecting to PostgreSQL databases

Description

The PDO_PGSQL Data Source Name (DSN) is composed of the following elements, delimited by spaces:

DSN prefix

The DSN prefix is **pgsql:**.

host

The hostname on which the database server resides.

port

The port on which the database server is running.

dbname

The name of the database.

user

The name of the user for the connection. If you specify the user name in the DSN, PDO ignores the value of the user name argument in the PDO constructor.

password

The password of the user for the connection. If you specify the password in the DSN, PDO ignores the value of the password argument in the PDO constructor.

Note
The *bytea* fields are returned as streams.

Examples

Example #70 - PDO_PGSQL DSN examples

The following example shows a PDO_PGSQL DSN for connecting to a PostgreSQL database:

```
pgsql:host=localhost port=5432 dbname=testdb user=bruce password=mypass
```

PDO::pgsqlLOBCreate

PDO::pgsqlLOBCreate -- Creates a new large object

Description

string **PDO::pgsqlLOBCreate** (void)

PDO::pgsqlLOBCreate() creates a large object and returns the OID of that object. You may then open a stream on the object using PDO::pgsqlLOBOpen() to read or write data to it. The OID can be stored in columns of type OID and be used to reference the large object, without causing the row to grow arbitrarily large. The large object will continue to live in the database until it is removed by calling PDO::pgsqlLOBUnlink().

Large objects can be up to 2GB in size, but are cumbersome to use; you need to ensure that PDO::pgsqlLOBUnlink() is called prior to deleting the last row that references its OID from your database. In addition, large objects have no access controls. As an alternative, try the bytea column type; recent versions of PostgreSQL allow bytea columns of up to 1GB in size and transparently manage the storage for optimal row size.

Note
This function must be called within a transaction.

Parameters

PDO::pgsqlLOBCreate() takes no parameters.

Return Values

Returns the OID of the newly created large object on success, or **FALSE** on failure.

Examples

Example #71 - A PDO::pgsqlLOBCreate() example

This example creates a new large object and copies the contents of a file into it. The OID is then stored into a table.

```
<?php
$db = new PDO('pgsql:dbname=test host=localhost', $user, $pass);
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$db->beginTransaction();
$oid = $db->pgsqlLOBCreate();
$stream = $db->pgsqlLOBOpen($oid, 'w');
```

```
$local = fopen($filename, 'rb');
stream_copy_to_stream($local, $stream);
$local = null;
$stream = null;
$stmt = $db->prepare("INSERT INTO BLOBS (ident, oid) VALUES (?, ?)");
$stmt->execute(array($some_id, $oid));
$db->commit();
?>
```

See Also

- PDO::pgsqlLOBOpen()
- PDO::pgsqlLOBUnlink()
- pg_lo_create()

PDO::pgsqlLOBOpen

PDO::pgsqlLOBOpen -- Opens an existing large object stream

Description

resource **PDO::pgsqlLOBOpen** (string `$oid` [, string `$mode`])

PDO::pgsqlLOBOpen() opens a stream to access the data referenced by *`oid`*. If *`mode`* is *r*, the stream is opened for reading, if *`mode`* is *w*, then the stream will be opened for writing. You can use all the usual filesystem functions, such as fread(), fwrite() and fgets() to manipulate the contents of the stream.

Note
This function, and all manipulations of the large object, must be called and carried out within a transaction.

Parameters

`oid`
A large object identifier.

`mode`
If mode is *r*, open the stream for reading. If mode is *w*, open the stream for writing.

Return Values

Returns a stream resource on success, or **FALSE** on failure.

Examples

Example #72 - A PDO::pgsqlLOBOpen() example

Following on from the PDO::pgsqlLOBCreate() example, this code snippet retrieves the large object from the database and outputs it to the browser.

```
<?php
$db = new PDO('pgsql:dbname=test host=localhost', $user, $pass);
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$db->beginTransaction();
$stmt = $db->prepare("select oid from BLOBS where ident = ?");
$stmt->execute(array($some_id));
```

```
$stmt->bindColumn('oid', $lob, PDO::PARAM_LOB);
$stmt->fetch(PDO::FETCH_BOUND);
fpassthru($lob);
?>
```

See Also

- PDO::pgsqlLOBCreate()
- PDO::pgsqlLOBUnlink()
- pg_lo_open()

PDO::pgsqlLOBUnlink

PDO::pgsqlLOBUnlink -- Deletes the large object

Description

bool **PDO::pgsqlLOBUnlink** (string `$oid`)

Deletes a large object from the database identified by OID.

Note
This function must be called within a transaction.

Parameters

`oid`
A large object identifier

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #73 - A PDO::pgsqlLOBUnlink() example

This example unlinks a large object from the database prior to deleting the row that references it from the blobs table we've been using in our PDO::pgsqlLOBCreate() and PDO::pgsqlLOBOpen() examples.

```
<?php
$db = new PDO('pgsql:dbname=test host=localhost', $user, $pass);
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$db->beginTransaction();
$db->pgsqlLOBUnlink($oid);
$stmt = $db->prepare("DELETE FROM BLOBS where ident = ?");
$stmt->execute(array($some_id));
$db->commit();
?>
```

See Also

- PDO::pgsqlLOBOpen()
- PDO::pgsqlLOBCreate()

SQLite Functions (PDO_SQLITE)

Introduction

PDO_SQLITE is a driver that implements the PHP Data Objects (PDO) interface to enable access to SQLite 3 databases.

In PHP 5.1, the SQLite extension also provides a driver for SQLite 2 databases; while it is not technically a part of the PDO_SQLITE driver, it behaves similarly, so it is documented alongside it. The SQLite 2 driver for PDO is provided primarily to make it easier to import legacy sqlite 2 database files into an application that uses the faster, more efficient sqlite 3 driver. As a result, the SQLite 2 driver is not as feature-rich as the SQLite 3 driver.

PDO_SQLITE DSN

PDO_SQLITE DSN -- Connecting to SQLite databases

Description

The PDO_SQLITE Data Source Name (DSN) is composed of the following elements:
DSN prefix (SQLite 3)

The DSN prefix is **sqlite:**.

- To access a database on disk, append the absolute path to the DSN prefix.
- To create a database in memory, append*:memory:* to the DSN prefix.

DSN prefix (SQLite 2)

The SQLite extension in PHP 5.1 provides a PDO driver that supports accessing and creating SQLite 2 databases. This enables you to access databases you may have created with the SQLite extension in previous versions of PHP.

Note
The sqlite2 driver is only available in PHP 5.1.x if you have enabled both PDO and ext/sqlite. It is not currently available via PECL.

The DSN prefix for connecting to SQLite 2 databases is **sqlite2:**.

- To access a database on disk, append the absolute path to the DSN prefix.
- To create a database in memory, append*:memory:* to the DSN prefix.

Examples

Example #74 - PDO_SQLITE DSN examples

The following examples show PDO_SQLITE DSN for connecting to SQLite databases:

```
sqlite:/opt/databases/mydb.sq3
sqlite::memory:
sqlite2:/opt/databases/mydb.sq2
sqlite2::memory:
```

PDO->sqliteCreateAggregate()

PDO->sqliteCreateAggregate() -- Registers an aggregating User Defined Function for use in SQL statements

Description

PDO

bool **sqliteCreateAggregate** (string `$function_name`, callback `$step_func`, callback `$finalize_func` [, int `$num_args`])

Warning

This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

This method is similar to PDO->sqliteCreateFunction() except that it registers functions that can be used to calculate a result aggregated across all the rows of a query.

The key difference between this method and PDO->sqliteCreateFunction() is that two functions are required to manage the aggregate.

Parameters

`function_name`
The name of the function used in SQL statements.

`step_func`
Callback function called for each row of the result set. Your PHP function should accumulate the result and store it in the aggregation context. This function need to be defined as:
step (mixed `$context`, int `$rownumber`, mixed `$value1` [, mixed `$value2` [, mixed `$..`]])
context will be **NULL** for the first row; on subsequent rows it will have the value that was previously returned from the step function; you should use this to maintain the aggregate state. *rownumber* will hold the current row number.

`finalize_func`
Callback function to aggregate the "stepped" data from each row. Once all the rows have been processed, this function will be called and it should then take the data from

the aggregation context and return the result. Callback functions should return a type understood by SQLite (i.e. scalar type). This function need to be defined as:
fini (mixed `$context`, int `$rownumber`)
context will hold the return value from the very last call to the step function. *rownumber* will hold the number of rows over which the aggregate was performed. The return value of this function will be used as the return value for the aggregate.

num_args
Hint to the SQLite parser if the callback function accepts a predetermined number of arguments.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #75 - max_length aggregation function example

```
<?php
$data = array(
  'one',
  'two',
  'three',
  'four',
  'five',
  'six',
  'seven',
  'eight',
  'nine',
  'ten',
  );
$db = new PDO('sqlite::memory:');
$db->exec("CREATE TABLE strings(a)");
$insert = $db->prepare('INSERT INTO strings VALUES (?)');
foreach ($data as $str) {
   $insert->execute(array($str));
}
$insert = null;

function max_len_step(&$context, $rownumber, $string)
{
   if (strlen($string) > $context) {
       $context = strlen($string);
   }
}

function max_len_finalize(&$context, $rownumber)
{
   return $context;
}

$db->sqliteCreateAggregate('max_len', 'max_len_step', 'max_len_finalize');
```

```
var_dump($db->query('SELECT max_len(a) from strings')->fetchAll());

?>
```

In this example, we are creating an aggregating function that will calculate the length of the longest string in one of the columns of the table. For each row, the *max_len_step* function is called and passed a `context` parameter. The context parameter is just like any other PHP variable and be set to hold an array or even an object value. In this example, we are simply using it to hold the maximum length we have seen so far; if the `string` has a length longer than the current maximum, we update the context to hold this new maximum length.

After all of the rows have been processed, SQLite calls the *max_len_finalize* function to determine the aggregate result. Here, we could perform some kind of calculation based on the data found in the `context`. In our simple example though, we have been calculating the result as the query progressed, so we simply need to return the context value.

Tip

It is NOT recommended for you to store a copy of the values in the context and then process them at the end, as you would cause SQLite to use a lot of memory to process the query - just think of how much memory you would need if a million rows were stored in memory, each containing a string 32 bytes in length.

Tip

You can use PDO->sqliteCreateFunction() and PDO->sqliteCreateAggregate() to override SQLite native SQL functions.

Note

This method is not available with the SQLite2 driver. Use the old style sqlite API for that instead.

See Also

- PDO->sqliteCreateFunction()
- sqlite_create_function()
- sqlite_create_aggregate()

PDO->sqliteCreateFunction()

PDO->sqliteCreateFunction() -- Registers a User Defined Function for use in SQL statements

Description

PDO

bool **sqliteCreateFunction** (string `$function_name`, callback `$callback` [, int `$num_args`])

Warning

This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

This method allows you to register a PHP function with SQLite as an UDF (User Defined Function), so that it can be called from within your SQL statements.

The UDF can be used in any SQL statement that can call functions, such as SELECT and UPDATE statements and also in triggers.

Parameters

`function_name`
: The name of the function used in SQL statements.

`callback`
: Callback function to handle the defined SQL function.

Note

Callback functions should return a type understood by SQLite (i.e. scalar type).

`num_args`
: Hint to the SQLite parser if the callback function accepts a predetermined number of arguments.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #76 - PDO::sqliteCreateFunction() example

```
<?php
function md5_and_reverse($string)
{
   return strrev(md5($string));
}

$db = new PDO('sqlite:sqlitedb');
$db->sqliteCreateFunction('md5rev', 'md5_and_reverse', 1);
$rows = $db->query('SELECT md5rev(filename) FROM files')->fetchAll();
?>
```

In this example, we have a function that calculates the md5 sum of a string, and then reverses it. When the SQL statement executes, it returns the value of the filename transformed by our function. The data returned in *$rows* contains the processed result.

The beauty of this technique is that you do not need to process the result using a foreach() loop after you have queried for the data.

Tip

You can use PDO->sqliteCreateFunction() and PDO->sqliteCreateAggregate() to override SQLite native SQL functions.

Note

This method is not available with the SQLite2 driver. Use the old style sqlite API for that instead.

See Also

- PDO->sqliteCreateAggregate()
- sqlite_create_function()
- sqlite_create_aggregate()