Moving from mysql_query to PDO

August 8th, 2014

The mysql extension has been marked as deprecated as of php 5.5.0 and will be removed completely in a future version.

It is old and not very user friendly.

This article will introduce PDO as a replacement.

Connecting to a Database

Connecting with mysql_connect

```
$link = mysql_connect('localhost', 'user', 'password');
mysql_select_db('database', $link);
```

The mysql_connect function returns a MySQL link identifier if the database connection was successful or FALSE if the connection was not successful. After connecting to the database server the database must be selected before any queries can be run on it. This is done with the mysql*select*db function.

Connecting with PDO

```
$dbh = new PDO('mysql:host=localhost;dbname=database', 'user', 'password');
```

The PDO constructor accepts a as the first parameter and the user and password as the second and third parameter.

This will create a new PDO instance if successful or throw a PDOException if the connection fails. I will go into error handling later on.

Running a Query

```
<?php
$result = mysql_query('SELECT * WHERE 1=1');
if (!$result) {
    die('Invalid query: ' . mysql_error());
}
```

Running a query with mysql_query

```
$result = mysql_query("SELECT * FROM tablename", $link);
```

The mysql_query function accepts the sql query as a string for the first parameter and the mysql link identifier returned from the mysql_connect function as the second parameter. For a successfully query it returns a resource for a query that would return a result-set such as a SELECT query and TRUE for a INSERT, UPDATE, or DELETE query. It returns FALSE if the query fails.

Running a query with PDO

```
$sth = $dbh->query("SELECT * FROM tablename");
```

The query method on the PDO object accepts the sql query as its only parameter. By default it returns an instance of PDOStatement if successful or FALSE if not. This can be changed my modifying the error mode for the connection. This will be explained further in the Handling Errors section.

Fetching Data

Fetching a single row with mysql*fetch*assoc

```
$users = array();
$result = mysql_query("SELECT * FROM users WHERE id = 1", $link);
$user = mysql_fetch_assoc($result)
```

To get the data from a query you can use the mysql*fetch** functions. To return the data as an associative array use the mysql*fetch*assoc function. To return the data as a numeric indexed array use the mysql*fetch*row function. To return the data as an object use the mysql*fetch*object function. It will return the next row in the result-set or FALSE of there are no more rows.

Fetching a single row with PDO

```
$sth = $dbh->query("SELECT * FROM users WHERE id = 1");
$user = $sth->fetch(PDO::FETCH_ASSOC);
```

You can use the fetch method on the PDOStatement object returned from the PDO::query method. It accepts the fetch style as the first parameter. To return data as an associative array pass the PDO::FETCH*ASSOC constant. To return data as a numeric indexed array pass the PDO::FETCH*NUM constant. To return data as an object pass the PDO::FETCH_OBJ constant.

Fetching all rows with mysql*fetch*assoc

```
$users = array();
$result = mysql_query("SELECT * FROM users", $link);
while ($row = mysql_fetch_assoc($result)) {
    $users[] = $row;
}
```

To fetch all of the rows from a result-set you will need to loop through calling mysql*fetch** until it returns false. This will result in an multi-dimensional associative array with all the records in the users table.

Fetching all rows with PDO::fetchAll

```
$sth = $dbh->query("SELECT * FROM users");
$users = $sth->fetchAll(PDO::FETCH_ASSOC);
```

There is no need to loop to pull an entire result-set with PDO. Just use the PDOStatement::fetchAll method passing the result style you want.

Getting the row id of the inserted record with mysql*insert*id

```
mysql_query("INSERT INTO users (first_name, last_name, email_address) VALUES
('John', 'Doe', 'email@example.com')", $link);
$id = mysql_insert_id($link);
```

After performing an insert query call mysql*insert*id to get the row id of the last inserted record.

Getting the row id of the inserted record with PDO::lastInsertId

```
$dbh->query("INSERT INTO users (first_name, last_name, email_address) VALUES
('John', 'Doe', 'email@example.com')", $link);
$id = $dbh->lastInsertId();
```

After performing an insert query call the PDO::lastInsertId method to get the row id of the last inserted record.

Other Fetching Methods

One feature in PDO that isn't available in the mysql extension is a couple more advanced fetching methods.

I've already shown you the PDOStatement::fetch method to fetch a single row and the PDOStatement ::fetchAll method to fetch all rows.

Fetching a single column from a single record

Say you want to pull the user id of a user with a given email address. You do not need any of the other user data, just the id.

With the mysql extension you would end up with an array with one key an value.

```
$result = mysql_query("SELECT id FROM users WHERE email = 'bob@example.com'",
$link);
$row = mysql_fetch_assoc($result);
if ($row) {
    $id = $row['id'];
} else {
    $id = false;
}
```

With PDO you can fetch the id as a scalar value

```
$sth = $dbh->query("SELECT id FROM users WHERE email = 'bob@example.com'");
$id = $sth->fetchColumn();
```

Fetching an array of scalar values

Now say you wanted an array of user ids for all active users.

```
$users = array();
$result = mysql_query("SELECT id FROM users WHERE active = 1", $link);
while ($row = mysql_fetch_assoc($result)) {
    $users[] = $row['id'];
}

$sth = $dbh->query("SELECT id FROM users WHERE active = 1");
$users = $sth->fetchAll(PDO::FETCH_COLUMN);
```

Fetching key value pairs

Now say you wanted an array of users with the user id as the keys and the user's name as the values.

```
$users = array();
$result = mysql_query("SELECT id, name FROM users", $link);
while ($row = mysql_fetch_assoc($result)) {
    $users[$row['id']] = $row['name'];
}

$sth = $dbh->query("SELECT id, name FROM users");
$users = $sth->fetchAll(PDO::FETCH_KEY_PAIR);
```

Fetching into a specific class

Many times you want to populate a class with result from a query.

This is easy with PDO.

```
class User
{
    public $id;
    public $name;
    public $email;
}

$sth = $dbh->query("SELECT id, name, email FROM users WHERE id = 1");
$users = $sth->fetch(PDO::FETCH_CLASS, 'user');
```

This isn't that exciting with an example as simple as this but the User class can have other methods in it or validation with the __set() magic method. You can also pass it to methods with type hinting to ensure you are getting the correct data.

Handling Errors

How you handle errors is an important part of developing an application. It cannot be assumed that the database will always be up and every query will succeed.

Without checking if a given query was successful the code will continue running assuming that was query was successful. This can lead to further problems with the application. For example if you have a user registration form that inserts the user into the database then sends out an activation email it makes no sense to send the email unless the account was created successfully.

A lot of example code in the wild will use the die function to exit the script when a mysql* *function fails printing the database error using the [mysql*error] (http://us2.php.net/manual/en/function.mysql-error.php) function. This is generally not the best way to handle errors. Printing the errors to the screen can expose information to the user than can compromise security.

A better idea would be to log any errors that occur and show the user a human readable error message saying something went wrong either by redirecting them to an error page or showing the message on the current page.

For the following examples I'm going to assume that you have defined a function names handle_error that accepts an error message and error code.

```
/**
 *
 * Logs error
 *
 * @param string $message Human friendly error message
 * @param string $error Error message returned from function or method that
failed
 * @param int $code Error code from failed function or method
 * @param mixed $extra Any extra data you want to log.
 *
 * @return void
 */
function log_error($message, $error = null, $code = null, $extra = null){
    // This would log the error to an error log
    // You could use an existing logging library or use a simple fwrite.
    // For a good logging library I recommend monolog
https://github.com/Seldaek/monolog
}
```

Handling errors with mysql*connect and mysql*query

In general the mysql_* functions return FALSE when they fail. After calling one of them you need to check the result to see if it was successful.

The function mysql_error will return a text error message of the last error that occurred and mysql_errno will return a numeric error code.

```
$link = mysql_connect('localhost', 'user', 'password');
if ($link === false) {
    log_error("Failed to connect to database", mysql_error(), mysql_errno());
    // This would store the human readable message for the next request
    flash_message("There was a problem connecting to the database. Please try
again later.");
    // This could redirect to a separate error page or just to itself
    header('location: error.php');
    exit;
}
if (mysql_select_db('database', $link) === false) {
    log_error("Failed to select database", mysql_error(), mysql_errno());
    // This would store the human readable message for the next request
    flash_message("There was a problem connecting to the database. Please try
again later.");
    // This could redirect to a separate error page or just to itself
    header('location: error.php');
    exit;
}
```

You can handle errors with queries much the same way.

```
$result = mysql_query("INSERT INTO users (first_name, last_name, email_address)
VALUES ('John', 'Doe', 'email@example.com')", $link);
if ($result === false) {
    log_error("Failed to insert user", mysql_error(), mysql_errno());
    // This would store the human readable message for the next request
    flash_message("There was a problem creating your account. Please try again
later.");
    // This could redirect to a separate error page or just to itself
    header('location: error.php');
    exit;
}
```

Handling errors with PDO

PDO has multiple ways of handling errors.

There are three error modes for PDO.

The first is PDO::ERRMODE*SILENT. This acts much like the mysql** functions in that after calling a PDO method you need to check PDO::errorCode or PDO::errorInfo to see if it was successful.

The second error mode is PDO::ERRMODE*WARNING. This is much the same except an E*WARNING message is also thrown.

The final error mode is PDO::ERRMODE_EXCEPTION. This one throws a PDOException when an error occurs. This is the method I recommend and will be using it for further examples.

```
// You can set the error mode using the fourth options parameter on the
constructor
$dbh = new PDO($dsn, $user, $password, array(PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION));

// or you can use the setAttribute method to set the error mode on an existing
connection
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

try {
    $dbh = new PDO($dsn, $user, $password, array(PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION));
} catch (PDOException $e) {
    log_error("Failed to connect to database", $e->getMessage(), $e->getCode(),
array('exception' => $e));
}

try {
    $dbh->query("INVALID SQL");
} catch (PDOException $e) {
    log_error("Failed to run query", $e->getMessage(), $e->getCode(),
array('exception' => $e));
}
```

Escaping Data

Whenever you execute a query that contains any data other than string literals you need to guard yourself against SQL Injection.

All data should be escaped whether it comes from a posted form, another database query, read from a file, or returned from a web service. Anything that is not a hard-coded string that you typed yourself cannot be trusted and needs to be escaped.

Escaping data with the mysql extension

Escaping data with the mysql extension is done with the mysql*real*escape_string function.

```
$query = sprintf("SELECT * FROM users WHERE username='%s' AND password='%s'",
    mysql_real_escape_string($name, $link),
    mysql_real_escape_string($password, $link)
);
$result = mysql_query($query, $link);
if ($result === false) {
    log_error("Failed to select user", mysql_error(), mysql_errno());
```

```
    $user = false;
} else {
    $user = mysql_fetch_assoc($result)
}
```

Escaping data with PDO

PDO has a similar method of escaping data PDO::quote. However this isn't the best way of handling passing data to queries with PDO.

Prepared Statements

Up until now I haven't covered anything that would provide a real advantage to switching to PDO. Sure some of the syntax is a little shorter and easier to work with but are just differences in syntax for the same features.

Now I'll get to features that you just cannot do with the plain mysql extension.

Prepared statements work a little differently than just replacing placeholders with escaped strings like you would do with the mysql*real*escape_string function or the PDO::quote method.

With a prepared statement the query is actually sent to the database with the placeholders. This query is compiled and a statement is returned. You then send the data over separately and the database does handles all the escaping and replacing for you. This makes them much safer and less error prone than escaping the data yourself.

With PDO there are two ways of providing parameters for a prepared statement.

```
try {
    $sth = $dbh->prepare("SELECT * FROM users WHERE username = ? AND password
= ?");
    $sth->execute(array($username, $password));
    $user = $sth->fetch(PDO::FETCH_ASSOC);
} catch (PDOException $e) {
    log_error("Failed to select user", $e->getMessage(), $e->getCode(),
array('exception' => $e));
}

try {
    $sth = $dbh->prepare("SELECT * FROM users WHERE username = :username AND
password = :password");
    $sth->execute(array(
        'username' => $username,
        'password' => $password
    ));
    $user = $sth->fetch(PDO::FETCH_ASSOC);
} catch (PDOException $e) {
    log_error("Failed to select user", $e->getMessage(), $e->getCode(),
array('exception' => $e));
}
```

Notice that in both of these he values are not surrounded in quotes like with the mysql extension. Adding quotes will result in a failed query.

I recommend always using a prepared statement if your query involves any data that you would otherwise need to escape.

Executing a prepared statement multiple times

Another big advantage to using a prepared statement is when you need to execute a query multiple times with different data, such as inserting multiple records.

You can prepare the query and execute it multiple times.

```
$users = array(
  array(
    'name' => 'Bob',
    'email' => 'bob@example.com'
  ),array(
    'name' => 'Steve',
    'email' => 'steve@example.com'
  ),array(
    'name' => 'Carl',
    'email' => 'carl@example.com'
  ),array(
    'name' => 'John',
    'email' => 'john@example.com'
  ),array(
    'name' => 'Ken',
    'email' => 'ken@example.com'
  )
);
$sth = $dbh->prepare("INSERT INTO users (name, email) VALUES (:name, :email)");
foreach ($users as $user) {
    try {
        $sth->execute($user);
    } catch (PDOException $e) {
        log_error("Failed to insert user", $e->getMessage(), $e->getCode(),
array('exception' => $e));
    }
}
```

Because the query is only parsed and compiled by the database once executing a prepared statement multiple times will be much faster than running the full query multiple times.

Escaping identifiers

One limitation to parameters in prepared statements is that they can only be used for values. You use parameters for things like table or column names.

For example the following will not work.

```
$column = $_POST['column'];
$value = $_POST['search'];
$sth = $dbh->prepare("SELECT * FROM users where :column = :value");
$sth->bindValue(':column', $column);
$sth->bindValue(':value', $value);
$sth->execute();
```

Since you should know what columns a table has the best way to do something like this is to check the column against a whitelist of accaptible data before passing to your query.

```
$column = $_POST['column'];
$value = $_POST['search'];

$allowed = array('name', 'email');
if(!in_array($column, $allowed)){
    // throw an error and do not execute the query.
```

```
} else {
    $sth = $dbh->prepare("SELECT * FROM users where $column = :value");
    $sth->bindValue(':value', $value);
    $sth->execute();
}
```

Transactions

Transactions are another useful feature that isn't available with the mysql extension.

Lets say you have a user table that stores the user's login credentials and you have a user_profile that stores extra profile data for that user with a foreign key to the user table.

user	user_profile
user_id	profile_id
name	user_id
email	bio
password	interests

When you add a new user you also want to add the user's profile and then send a welcome email to the user.

What happens if a query fails?

If the insert into the user table fails you can throw an error and not try to insert the profile.

But what happens if the insert into the user_profile table fails? It could break your application to have a user without a profile. Do you try to delete the newly inserted user if the second query fails?

What happens if the welcome email fails? What if this email is vital to the application and that if it fails you may as well have never created the account? Do you delete both records if it fails?

This is a already messy and could get worse with more tables or other steps.

Fortunately transactions can help with this.

In this example assume the MailSender class throws an exception if the email fails to be sent.

```
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$dbh->beginTransaction();
try {
    $sth = $dbh->prepare("INSERT INTO user (name, email, password) VALUES (?, ?,
?)");
    $sth->execute(array($name, $email, $password));
    $user_id = $dbh->lastInsertId();
    $sth->prepare("INSERT INTO user_profile (user_id, bio, interests) VALUES (?,
?, ?)");
    $sth->execute(array($user_id, $bio, $interests));
    $mailer = new MailSender();
    $mailer->sendWelcomeEmail($name, $email);
    $dbh->commit();
} catch (Exception $e) {
    $dbh->rollBack();
    // handle the error
}
```

If any of the queries fail or the mail sender fails to send the mail the executed queries will be rolled back.

Conclusion

I've only scratched the surface of what PDO can do beyond the basic mysql extension.

Once you've used it a while and get used to the syntax you'll wonder how you ever lived without it.

https://www.thedevfiles.com/2014/08/moving-from-mysql-query-to-pdo