IPL
escola superior
de tecnologia e gestão
instituto politécnico
de leiria

DBMS and MySQL

MySQL databases in PHP using PDO

Fernando Silva & Marco Monteiro

Contributors

- Author(s):
  - Fernando Silva(Fernando.silva@ipleiria.pt)
  - Marco Monteiro (marco.monteiro@ipleiria.pt)

- Contributor(s):
  - Vitor Carreira (vitor.carreira@ipleiria.pt)

Summary

1. Database Integration
2. PDO
3. Database Security
4. References

1 – DATABASE ACCESS

Database Integration

- PHP has native connections available to many database systems: MySQL, PostgreSQL, Oracle, dbm, FilePro, DB2, Hyperwave, Informix, InterBase, Sybase, etc
- Other connectivity options:
  - SQLite
  - Open Database Connectivity (ODBC) - allow connectivity to any database that provides an ODBC driver
  - **PHP Database Objects** (PDO) - database access abstraction layer which allows consistent access and promotes secure coding practices

MySQL

- MySQL Database main strengths:
  - Portability - can be used on many different operating systems
  - Ease of configuration and use
  - Low cost (free open source license)
  - Low hardware requirements
- MySQL database access with PHP
  - mysql_* functions (deprecated)
  - MySQLi (procedural or object-oriented)
  - PDO MySQL

PDO – PHP Data Objects

- Defines a lightweight, consistent interface for accessing databases in PHP
- Provides a data-access abstraction layer
  - Regardless of which database you're using, you use the same functions to issue queries and fetch data
  - Note that it is impossible to switch database backends by changing a single line in PDO config - different SQL flavors
- The real PDO benefits are:
  - **Security**: prepared statements
  - **Usability**: many helper functions to automate routine operations
  - **Reusability**: unified API to access multitude of databases

PDO - Drivers

- The following drivers currently implement the PDO interface:

Driver name	Supported databases
PDO_CUBRID	Cubrid
PDO_DBLIB	FreeTDS / Microsoft SQL Server / Sybase
PDO_FIREBIRD	Firebird
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
PDO_SQLSRV	Microsoft SQL Server / SQL Azure
PDO_4D	4D

2 – PDO

PHP Data Objects

DB Access Typical Flow

- When Web Applications interact with Relational Databases with a *"plain"* data access technology (e.g. PDO) they usually use the following algorithm:

1. Establish a connection with the DB
2. Send a SQL command to the DB
   - select, insert, update, delete, create table, etc.
3. Retrieve the results
   - If the SQL command returns results (select)
4. Close the connection
   - This may be an automatic operation when the server terminates the web page processing

PDO – Establish a connection


```
$host= 'localhost';
$dbname= 'test';
$user= 'homestead';
$password= 'secret';
$charset= 'utf8';

$dsn= "mysql:host=$host;dbname=$dbname;charset=$charset";
$opt= [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];

$pdo = new PDO($dsn, $user, $password, $opt);
```

- Note: Error mode = PDO::ERRMODE_EXCEPTION (errors will throw exception) – it is the preferable error mode -
  Other supported modes: PDO::ERRMODE_SILENT; PDO::ERRMODE_WARNING

Queries

- There are 2 ways to run queries (select …) in PDO
  - **Query** method
    - Without parameters
  - With **Prepared Statements**
    - With or without parameters
    - Optimizes queries (better performance)

PDO::query


```
$pdo = new PDO( . . . );
$results = [];
$stmt = $pdo->query('SELECT fullname FROM users');

while ($row = $stmt->fetch()) {
    $results[] = $row;
}
```

- Don’t include variable on the query
  - If variables content is not sanitized, it might open up a security breach – SQL Inject might be possible

```
. . .
$stmt = $pdo->query("SELECT fullname FROM users
                     where username = $user");
. . .
```

PDO – Prepared Statements

- With prepared statements, the SQL command is parsed, compiled and optimized before executed
  - Better performance
- Also, prepared statement supports parameters
  - It can also be used with no parameters
- It is recommended to **always use Prepared Statements** (with or without parameters)
- If at least one parameter is going to be used:
  1. Substitute it with a placeholder
  2. Prepare your query
  3. Execute it, passing variables separately

PDO – Prepared Statements

1. Alter original query, adding placeholders in place of variables

- Code like this:

```
$sql= "SELECT * FROM users where email = '$email' "
```

- Will become:

```
$sql= "SELECT * FROM users where email = ?"
```

- Or:

```
$sql= "SELECT * FROM users where email = :email"
```

- PDO supports positional (**?**) and named (**:email**) placeholders

PDO – Prepared Statements

2. Prepare the query using **`PDO::prepare()`**
   (query is parsed, compiled and optimized)
3. To execute the query, run **execute()** method, passing variables (parameter values) in it, in the form of array

```
$sql = 'SELECT * FROM users WHERE email = ?';
$stmt= $pdo->prepare($sql);
$stmt->execute([$email]);
$user = $stmt->fetch();
```

```
$sql = 'SELECT * FROM users WHERE email = :email';
$stmt= $pdo->prepare($sql);
$stmt->execute(['email' => $email]);
$user = $stmt->fetch();
```

PDO – Retrieve the results - foreach

- **foreach** statement - The most basic and direct way to get multiple rows from a statement
  - `PDOStatement` can be iterated by using foreach statement

```
$sql = 'SELECT fullname FROM users';
$stmt= $pdo->prepare($sql);
$stmt->execute();

foreach($stmt as $row) {
    echo $row['fullname'] . "\n";
}
```

PDO – Retrieve the results - fetch

- **`fetch()`** – fetches a single row from the database and moves to the next row
  - Return false if no further rows are available
  - Default fetch mode (FETCH_BOTH) means that each row field can be accessed by index (position) or field name

```
$sql = 'SELECT fullname FROM users';
$stmt= $pdo->prepare($sql);
$stmt->execute();

while ($row = $stmt->fetch()) {
    echo $row[0];                 // Field position
    echo $row['fullname'];        // Field name
}
```

PDO – fetch mode

- Fetch modes - changes the data format of each row
  - `PDO::FETCH_NUM` - enumerated array
  - `PDO::FETCH_ASSOC` - associative array
  - `PDO::FETCH_BOTH` – **default** - both of the above
  - `PDO::FETCH_OBJ` - object
  - `PDO::FETCH_LAZY` allows all three (numeric, associative and object) methods without memory overhead
  - `PDO::FETCH_CLASS` creates an object of a particular class
- Change fetch mode:

```
$row = $stmt->fetch(PDO::FETCH_OBJ);
```

```
$stmt->setFetchMode(PDO::FETCH_OBJ);
$row = $stmt->fetch();
```

- Set default fetch mode in the options array of PDO constructor:

```
$opt = [ PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_CLASS ];
```

PDO – Retrieve the results - fetchAll

- **`fetchAll()`** – returns an array containing all rows in the result set
  - Return an empty array if sql command returns 0 rows, or false if failure

```
$sql = 'SELECT fullname FROM users';
$stmt= $pdo->prepare($sql);
$stmt->execute();

$result = $stmt->fetchAll();
foreach ($result as $row) {
    echo $row['fullname'];
}
```

PDO – Insert, update, delete

- To PDO, it doesn't matter which SQL command you're running – they're all the same:
  1. Add placeholders (parameters) to the SQL Command
  2. Prepare the SQL command
  3. Execute it, passing variables separately

```
$sql = "INSERT INTO users (email, fullname)
                    VALUES (:email, :fullname)";
$stmt = $pdo->prepare($sql);
$stmt->execute(['fullname' => $name, 'email' => $email]);
if ($stmt->rowCount() != 1) {
    echo "Something went wrong on last insert command";
}
```

rowCount() – returns the number of rows affected by last command executed

PDO – Closing the connection

- The connection remains active for the lifetime of the PDO object
- To close the connection, destroy the object by ensuring that all remaining references to it are deleted
  - Assign NULL to the variable that holds the object
  - If you don't do this explicitly, PHP will automatically close the connection when the script ends

```
$pdo = new PDO($dsn, $user, $password, $opt);
. . .
$pdo = null;      // Closes the connection
```

3 – DATABASE SECURITY

SQL Injection

- Use parameters instead of string concatenation
- Why? Parameters avoid SQL Injection.
- How? Analyze this example:

With Parameters

```
$u=$_POST["username"];
$s=$_POST["password"];

$q= "select userid from user".
    " where(username=?)".
    " and (password=?)";

$stmt = $pdo->prepare($q);
$stmt->execute([$u, $s]);
```

With String Concatenation

```
$u=$_POST["username"];
$s=$_POST["password"];

$q= "select userid from user".
    " where(username='$u')".
    " and (password='$s')";

$stmt = $pdo->prepare($q);
$stmt->execute();
```

SQL Injection

- SQL Injection examples. User fills ”password” field with an empty string and “username” with:

Username Field	SQL Command (previous code) after the string concatenation	Consequence
`john') #`	`select userid from user where(username='john') #') and (password='')`	Obtains UserID of “John” without knowing its password
`') or (1=1) --`	`select userid from user where(username='') or (1=1) -- ') and (password='')`	Obtains UserID of first user without knowing its password or username
`'); drop table user; #`	`select userid from user where(username=''); drop table user; #') and (password='')`	Removes “user” table from the Database !!!

SQL Syntax: “#” or “-- ” starts a comment

- With parameters SQL Injection is avoided. Examples would just return 0 rows

Handling special chars

- `htmlspecialchars($string)`
- `htmlentities($string)`
  - Some characters have a special meaning for HTML and if present can cause a validation error or even inject dangerous behavior
  - These methods encode characters present in $string that have a special meanings in HTML (&, <, >, ‘, “)
  - When displaying textual information retrieved from a database, always call htmlspecialchars or htmlentities

Handling special chars


```
$field_from_db = "<script type='text/javascript'>
     document.location='http://someevilpage.com';
     </script>";

printf("<p>%s</p>", $field_from_db);
```

FAIL! XSS (Cross Site Scripting)

OK. No XSS

```
$field_from_db = "<script type='text/javascript'>
     document.location='http://someevilpage.com';
     </script>";

printf("<p>%s</p>", htmlspecialchars($field_from_db));
// or
// printf("<p>%s</p>", htmlentities($field_from_db));
```

Database security

1. Put database access credentials in a file only accessible locally (as a failsafe the file should have extension .php)

Example: db.php

```
<?php
$global_pdo_connection = null;
function dbConn()
{
    global $global_pdo_connection;
    if (!empty($global_pdo_connection)) { // if already created
        return $global_pdo_connection;    // just reuse it
    }
    $host= 'localhost';       $dbname= 'dbname…';
    $user= 'dbuser…. ';       $pass=   'password…';
    $charset= 'utf8';
    $dsn= "mysql:host=$host;dbname=$dbname;charset=$charset";
    $opt= [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];
    $global_pdo_connection = new PDO($dsn, $user, $pass, $opt);
    return $global_pdo_connection;
}
```

Database security

2. Don’t grant DROP, ALTER, GRANT privileges to the database user used on the PDO connection
3. Use prepared statements whenever using variables in the query
4. Call htmlspecialchars or htmlentities for each string fetched from a database and before presenting it to the user
5. Hide (encrypt / hash) sensitive information such as passwords before storing it

```
<?php
    $password= password_hash($plaintext_pass,PASSWORD_DEFAULT);
?>
```

5 – REFERENCES

References

- Official (PHP)
  - http://www.php.net/
  - http://php.net/manual/en/book.pdo.php
- (The only proper) PDO tutorial
  - https://phpdelusions.net/pdo
- PDO Tutorial for MySQL Developers
  - http://wiki.hashphp.org/PDO_Tutorial_for_MySQL_Developers