CET218

Advanced Web Programming

05 - Accessing Database using PHP & PDO

Dr. Ahmed Said

Start →

Querying a MySQL Database with PHP

- The reason for using `PHP` as an interface to MySQL is to format the results of SQL queries in a form visible in a web page.
- As long as you can log in to your `MySQL` installation using your username and password, you can also do so from PHP.
- However, instead of using MySQL's command line to enter instructions and view output, you will create query strings that are passed to MySQL.
- When `MySQL` returns its response, **it will come as a data structure that PHP can recognize instead of the formatted output you see when you work on the command line**.
- Further `PHP` commands can retrieve the data and format it for the web page.

Why Should You Use MySQL/MariaDB?

1. Popular database management system that is used in many web applications.

2. Open-source* and free to use.

3. A powerful database management system that can handle large amounts of data.

4. Easy to use and has a simple syntax.

5. Fast and efficient.

6. Secure and reliable with features like encryption, user authentication, and role-based access control ensuring data protection.

7. Additionally, built-in tools for database observability allow developers to monitor and optimize performance effectively.

Connecting to MySQL

- `PHP` offers three primary methods for MySQL database connection:
    1. `MySQL` (obsolete due to security vulnerabilities)
    2. `MySQLi`
    3. `PDO`
- `MySQLi` and `PDO` are actively supported and widely used.

1. MySQLi

- `MySQLi`, a PHP extension tailored for MySQL databases, was launched with PHP 5.0.0.
- Supporting MySQL versions 4.1.13 and later, it delivers enhanced features and functionality.
- `MySQLi` also provides advancements in security, speed, and flexibility.
- It supports both `procedural` and `object-oriented programming` styles, easing the transition from the older `mysql()` extension.
- `MySQLi` also includes advanced features like prepared statements and transactions for enhanced security and performance.

2. PDO (PHP Data Objects)

- The `PHP Data Objects (PDO)` extension acts as a database abstraction layer, providing a consistent `Application Programming Interface (API)` for interacting with various database types, including `MySQL`, `PostgreSQL`, `SQLite`, and more.
- This allows developers to switch between different databases without significant code changes.
- `PDO` supports prepared statements, error handling, and fetching data in different formats.
- Its portability makes it a great choice for projects that may require future support for multiple database types.
- Should I Use `MySQLi` or `PDO`?

Should I Use MySQLi or PDO?

- Both `MySQLi` and `PDO` are excellent choices for PHP-MySQL interaction. The best choice depends on project requirements and coding style preferences.
- `MySQLi` is ideal for projects working exclusively with `MySQL` databases. It provides both procedural and object-oriented interfaces and directly supports MySQL-specific features.
- `PDO` is better suited for projects that may need to interact with multiple database types, thanks to its unified API.
- `Security Considerations`: Both MySQLi and PDO, when used with prepared statements, protect against **SQL injection attacks**.
- When choosing between `MySQLi` and `PDO`, consider the following:
  - Use `MySQLi` for exclusive `MySQL` work and preference for a MySQL-specific API.
  - Use `PDO` for potential multi-database support or preference for a consistent interface.

Querying a MySQL Database with PHP

- The reason for using PHP as an interface to MySQL is to format the results of SQL queries in a form visible in a web page.
- To query a MySQL database using PHP, you'll need to follow these steps:

1. **Connect to the MySQL database** using the appropriate method (MySQLi or PDO).
2. Prepare a SQL query string.
3. Execute the query using `mysqli_query()` or `PDO::query()`.
4. Retrieve the results using `mysqli_fetch_assoc()` or `PDO::fetch()`.
5. Repeat steps 2 to 4 until all desired data has been retrieved.
6. Disconnect from the database.
7. Display the results on a web page.

Creating a Login File

- Most websites developed with PHP contain multiple program files that will require access to MySQL and will thus need the login and password details.
- Therefore, it's sensible to create a single file to store these and then include that file wherever it's needed.

```php
<?php // login.php
  $host = 'localhost'; // Change as necessary
  $data = 'publications'; // Datanase name
  $user = 'root'; // username
  $pass = 'mysql'; // Password
  $chrs = 'utf8mb4';
  $attr = "mysql:host=$host;dbname=$data;charset=$chrs";
  $opts =
  [
    PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE  => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES    => false,
  ];
```

PDO Connection Options

```php
$opts = [
    PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES   => false,
];
```

- `PDO::ATTR_ERRMODE` : Determines error reporting behavior
    - `PDO::ERRMODE_EXCEPTION` : Throws PDOExceptions (recommended for development)
    - `PDO::ERRMODE_WARNING` : Issues PHP warnings
    - `PDO::ERRMODE_SILENT` : Only sets error codes (default)
- `PDO::ATTR_DEFAULT_FETCH_MODE` : Sets default fetch mode
    - `PDO::FETCH_ASSOC` : Returns results as associative arrays (column name => value)
    - `PDO::FETCH_OBJ` : Returns results as objects
    - `PDO::FETCH_BOTH` : Returns both numeric and associative arrays (default)
- `PDO::ATTR_EMULATE_PREPARES` : When false, uses real prepared statements
    - Setting to false improves security and handles data types properly

Connecting to MySQL Database

- Now that you have saved the `login.php` file, you can include it in any PHP files that will need to access the database by using the `require_once` statement.
- This is preferable to an include statement, as it will generate a fatal error if the file is not found.

```php
<?php
  require_once 'login.php';
  try {
    $pdo = new PDO($attr, $user, $pass, $opts);
  }
  catch (PDOException $e) {
    throw new PDOException($e->getMessage(), (int)$e->getCode());
  }
?>
```

warning

- You should also never be tempted to output the contents of any error message received from MySQL. Rather than helping your users, you could give away sensitive information to hackers, such as login details. Instead, just guide the user with information on how to overcome their difficulty based on what the error message reports to your code.

Querying a MySQL Database with PHP

- Sending a query to MySQL from PHP is as simple as including the relevant SQL in the query method of a connection object.

```php
<?php
  $query = "SELECT * FROM classics";
  $result = $pdo->query($query);
?>
```

- Here the variable `$query` is assigned a string containing the query to be made and then passed to the query method of the `$pdo` object, which returns a result that we place in the object `$result`.
- All the data returned by MySQL is now stored in an easily interrogable format in the `$result` object.

Fetching a result

- **Basic Query Execution**: Used for simple queries without parameters

```php
require_once 'login.php';
try
{
  $pdo = new PDO($attr, $user, $pass, $opts);
}
catch (PDOException $e)
{
  throw new PDOException($e->getMessage(), (int)$e->getCode());
}

// Simple query
$stmt = $pdo->query('SELECT * FROM users');

// Fetch all results as an associative array
$users = $stmt->fetchAll();

// Loop through results
foreach ($users as $user) {
    echo $user['name'] . '<br>';
}
```

Fetching a result, cont.

- Once you have an object returned in `$result`, you can use it to extract the data you want, **one item at a time**, using the `fetch()` method of the object.
- This method returns an array containing the next row of data from the result set.
- If there are no more rows to fetch, it returns `false`.

```php
require_once 'login.php';
...

$query = "SELECT * FROM classics";
$result = $pdo->query($query);
while ($row = $result->fetch())
{
  echo 'Author: ' . htmlspecialchars($row['author']) . "<br>";
  echo 'Title: ' . htmlspecialchars($row['title']) . "<br>";
  echo 'Category: ' . htmlspecialchars($row['category']) . "<br>";
  echo 'Year: ' . htmlspecialchars($row['year']) . "<br>";
  echo 'ISBN: ' . htmlspecialchars($row['isbn']) . "<br><br>";
}
```

Fetching a result, cont.

- When displaying data in a browser whose source was (or may have been) user input, there's always a risk of sneaky HTML characters being embedded within it—even if you believe it to have been previously sanitized—which could potentially be used for a **cross-site scripting (XSS) attack**.
- The simple way to prevent this possibility is to embed all such output within a call to the function `htmlspecialchars`, which replaces all such characters with harmless HTML entities.

Fetching a row while specifying the style

- The fetch method can return data in various styles, including the following:
  - `PDO::FETCH_ASSOC` : Returns the next row as an array indexed by column name
  - `PDO::FETCH_BOTH` (default) : Returns the next row as an array indexed by both column name and number
  - `PDO::FETCH_LAZY` : Returns the next row as an anonymous object that combines the benefits of the `PDO::FETCH_BOTH` mode and an object-oriented approach
  - `PDO::FETCH_OBJ` : Returns the next row as an anonymous object with column name as properties
  - `PDO::FETCH_NUM` : Returns an array indexed by column number

```php
...
// Fetching a row while specifying the style
while ($row = $result->fetch(PDO::FETCH_BOTH)) // Style of fetch
{
  ...
}
$result->setFetchMode(PDO::FETCH_ASSOC);
$user = $result->fetch();
...
```

Closing a connection

- PHP will eventually return the memory it has allocated for objects after you have finished with the script, so in small scripts, you don’t usually need to worry about releasing memory yourself.
- However, should you wish to close a PDO connection manually, you simply set it to null like this:

```php
$pdo = null;
```

Secure Queries with Prepared Statements

- **Using Prepared Statements**: Essential for secure queries with user input
- Prepared statements prevent SQL injection by separating SQL code from data

```php
// Prepare statement
$result = $pdo->prepare('SELECT * FROM users WHERE id = :id');

// Bind parameter
$result->bindParam(':id', $id, PDO::PARAM_INT);

// Execute query
$result->execute();
$user = $result->fetch();
```

- Parameter types can be explicitly set for better security:
  - `PDO::PARAM_INT` for integers
  - `PDO::PARAM_STR` for strings (default)
  - `PDO::PARAM_BOOL` for booleans
  - `PDO::PARAM_NULL` for NULL values

CRUD Operations

- **CRUD** Operations:
  - **C**reate (INSERT)
  - **R**ead (SELECT)
  - **U**pdate (UPDATE)
  - **D**elete (DELETE)
- CRUD operations are essential for managing data in a database

CRUD Operations: Adding Data

- Create operations add new records to your database:

```php
...
function addUser($pdo, $name, $email) {
    try {
        $query = "INSERT INTO users VALUES ($name, $email)";
        $result = $pdo->query($query);
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}

// Usage
$newUserId = addUser($pdo, "John Doe", "john@example.com");
echo "New user added with ID: $newUserId";
```

CRUD Operations: Adding Data (Secure Version)

- Create operations add new records to your database:

```php
...
function addUser($pdo, $name, $email) {
    try {
        // Prepare SQL statement
        $stmt = $pdo->prepare("INSERT INTO users (name, email) VALUES (:name, :email)");
        $stmt->bindParam(':name', $name);
        $stmt->bindParam(':email', $email);
        $stmt->execute();
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}

// Usage
$newUserId = addUser($pdo, "John Doe", "john@example.com");
echo "New user added with ID: $newUserId";
```

CRUD Operations: Retrieving Data

```php
function getAllUsers($pdo) {
    try {
        $stmt = $pdo->query("SELECT * FROM users ORDER BY name");
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}
function getUserById($pdo, $id) {
    try {
        $stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id");
        $stmt->bindParam(':id', $id, PDO::PARAM_INT);
        $stmt->execute();
        return $stmt->fetch(PDO::FETCH_ASSOC);
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}

// Usage
$users = getAllUsers($pdo);
foreach ($users as $user) {
    echo $user['name'] . " - " . $user['email'] . "<br>";
}
```

CRUD Operations: Retrieving Data

- Important retrieval concepts:
  - Use `WHERE` clauses to filter results
  - Use `ORDER BY` to sort results
  - For large datasets, consider pagination using `LIMIT` and `OFFSET`

CRUD Operations: Updating Data

- Update operations modify existing records:

```php
function updateUser($pdo, $id, $name, $email) {
    try {
        $stmt = $pdo->prepare("UPDATE users SET name = :name, email = :email WHERE id = :id");

        // Bind parameters
        $stmt->bindParam(':id', $id, PDO::PARAM_INT);
        $stmt->bindParam(':name', $name);
        $stmt->bindParam(':email', $email);

        // Execute statement
        $stmt->execute();

        return $stmt->rowCount(); // Returns number of rows affected
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}

// Usage
$rowsUpdated = updateUser($pdo, 1, "John Smith", "john.smith@example.com");
echo "Rows updated: $rowsUpdated";
```

CRUD Operations: Updating Data, Cont.

- Update best practices:
    - Always include a `WHERE` clause to avoid updating all records
    - Check `rowCount()` to verify changes were made
    - Only update fields that have actually changed
    - Consider using transactions for multi-table updates
    - Include validation to ensure data integrity
    - For critical records, consider logging changes or keeping change history

CRUD Operations: Deleting Data

- Delete operations remove records from your database:

```php
function deleteUser($pdo, $id) {
    try {
        $stmt = $pdo->prepare("DELETE FROM users WHERE id = :id");
        $stmt->bindParam(':id', $id, PDO::PARAM_INT);
        $stmt->execute();

        return $stmt->rowCount(); // Returns number of rows affected
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}

// Usage
$rowsDeleted = deleteUser($pdo, 1);
echo "Rows deleted: $rowsDeleted";
```

CRUD Operations: Deleting Data, cont.

- Deletion safety measures:
  - Always include a `WHERE` clause to avoid deleting all records
  - Verify the operation with `rowCount()`
  - Consider implementing "soft deletes" using a status flag
  - Be aware of foreign key constraints - related records may need special handling
  - In critical applications, implement a confirmation process
  - For sensitive data, consider keeping an audit trail of deletions

Using AUTO_INCREMENT

- **AUTO_INCREMENT** is a MySQL feature that automatically assigns a unique identifier to each new record inserted into a table.
- When using AUTO_INCREMENT, **you cannot know what value has been given to a column before a row is inserted**.
- Instead, if you need to know it, you must ask MySQL afterward using the `mysql_insert_id` function.
- This need is common: for instance, when you process a purchase, you might insert a new customer into a Customers table and then refer to the newly created CustId when inserting a purchase into the Purchases table.

Note

Using AUTO_INCREMENT is recommended instead of selecting the highest ID in the id column and incrementing it by one, because concurrent queries could change the values in that column after the highest value has been fetched and before the calculated value is stored.

Using AUTO_INCREMENT, cont.

```php
<?php
  require_once 'login.php';
  try
  {
    $pdo = new PDO($attr, $user, $pass, $opts);
  }
  catch (PDOException $e)
  {
    throw new PDOException($e->getMessage(), (int)$e->getCode());
  }
    $query = "INSERT INTO cats VALUES(NULL, 'Lynx', 'Stumpy', 5)";
    $result = $pdo->query($query);
    echo "The Insert ID was: " . $pdo->lastInsertId();
?>
```

Using insert IDs

- It's very common to insert data in multiple tables:
  - a book followed by its author, a customer followed by their purchase, and so on.
- When doing this with an autoincrement column, you will need to retain the insert ID returned for storing in the related table.
- For example, let's assume that these cats can be "adopted" by the public as a means of raising funds, and that when a new cat is stored in the cats table, we also want to create a key to tie it to the animal's adoptive owner.

```php
$query = "INSERT INTO cats VALUES(NULL, 'Lynx', 'Stumpy', 5)";
$result = $pdo->query($query);
$insertID = $pdo->lastInsertId();

$query = "INSERT INTO owners VALUES($insertID, 'Ann', 'Smith')";
$result = $pdo->query($query);
```

Practical Example: Processing Form Data

- A complete workflow for handling form submissions and database operations:

```php
require_once 'login.php';
// Check if form is submitted
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    try {
        $pdo = new PDO($attr, $user, $pass, $opts);
        // Get values from form
        $name = $_POST['name'];
        $email = $_POST['email'];

        // Insert data using prepared statement
        $stmt = $pdo->prepare("INSERT INTO users (name, email) VALUES (:name, :email)");
        $stmt->bindParam(':name', $name);
        $stmt->bindParam(':email', $email);
        $stmt->execute();

        echo "User added successfully";
    } catch(PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
```

Deleting a Record

- Deletion operations should always be handled carefully as they are irreversible:

```php
// Delete user by ID
function deleteUser($pdo, $userId) {
    try {
        $stmt = $pdo->prepare("DELETE FROM users WHERE id = :id");
        $stmt->bindParam(':id', $userId, PDO::PARAM_INT);
        $stmt->execute();

        // Check if any rows were affected
        if ($stmt->rowCount() > 0) {
            return "User deleted successfully";
        } else {
            return "No user found with that ID";
        }
    } catch(PDOException $e) {
        return "Error: " . $e->getMessage();
    }
}
// Usage with form submission
if (isset($_POST['delete'])) {
    $userId = $_POST['user_id'];
```

The $_POST array

- The `$_POST` array is an associative array that contains the values submitted via a POST request.
- It is populated by the browser when a form is submitted to a PHP script.
- The POST request is usually preferred (because it prevents placing unsightly data in the browser's address bar), and so we use it here.
- Depending on whether a form has been set to use the POST or the GET method, either the `$_POST` or the `$_GET` associative array will be populated with the form data. They can both be read in exactly the same way.
- The keys of the array correspond to the names of the form fields, and the values are the submitted data.
- Each field has an element in the array named after that field. So, if a form contains a field named isbn, the `$_POST` array contains an element keyed by the word isbn. The PHP program can read that field by referring to either `$_POST['isbn']`

Assignment

- Explorer using **code examples** the usage of the different fetching methods in PDO
- Create a page to retrieve a large dataset (such as 100 rows) and create pagination (such as Number of Rows = 20 per page) using `LIMIT` & `OFFSET`

THANK YOU