NORTHERN KENTUCKY UNIVERSITY

11 - MySQL queries with PHP

ASE230 – Server-Side Programming

Nicholas Caporusso

Objectives

- Make secure queries to a database

Agenda

1. PHP Data Objects (PDO)
2. Transactions
3. Parameter binding
4. Stored procedures

PHP Data Objects (PDO)

PHP Data Objects (PDO)

- PDO is a Database Access Abstraction Layer
- PDO abstracts not only a database API, but also basic operations that otherwise have to be repeated hundreds of times in every application, making your code extremely WET (Write everything twice).

- The real PDO benefits are:
  - security (usable prepared statements)
  - usability (many helper functions to automate routine operations)
  - reusability (unified API to access multitude of databases, from SQLite to Oracle)

Connecting to a database

- PDO has a connection method called DSN.
- A DSN is basically a string of options that tell PDO which driver to use, and the connection details.
- PDO's constructor takes at most 4 parameters, DSN, username, password, and an array of driver options.

```
<?php

$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8mb4',
'username', 'password');
```

Configuring parameters

- There are several parameters which can be specified:
  - PDO::ERRMODE_EXCEPTION
    - PDO will throw a PDOException and set its properties to reflect the error code and error information. This setting is also useful during debugging.
  - PDO::FETCH_ASSOC returns results as an associative array (default is both indexed and associative)

```php
<?php
$host = '127.0.0.1';
$db = 'test';
$user = 'root';
$pass = '';
$charset = 'utf8';

$dsn = "mysql:host=$host;dbname=$db;charset=$charset";
$opt = [
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
PDO::ATTR_EMULATE_PREPARES => false,
];
$pdo = new PDO($dsn, $user, $pass, $opt);
```

Main differences with previous systems

- Important notes for the late mysql extension users:
  - Unlike old mysql_* functions, which can be used anywhere in the code, **PDO instance is stored in a regular variable**, which means it can be inaccessible inside functions - so, one has to make it accessible, by means of passing it via function parameters or using more advanced techniques.
  - **The connection has to be made only once**! No connects in every function. No connects in every class constructor. Otherwise, multiple connections will be created, which will eventually kill your database server. Thus, a sole PDO instance has to be created and then used through whole script execution.
  - It is very important to set charset through DSN - that's the only proper way because it tells PDO which charset is going to be used. Therefore forget about running SET NAMES query manually, either via query() or PDO::MYSQL_ATTR_INIT_COMMAND. Only if your PHP version is unacceptably outdated (namely below 5.3.6), you have to use SET NAMES query and always turn emulation mode off.

Executing queries

- There are two ways to run a query in PDO:
  1. if no variables are going to be used in the query, you can use the PDO::query() method
  2. if the query has variables, PDO has prepared statements which are the only proper way to run a query, as they prevent injection
     - SQL Injection is an exploit of improperly formatted query.

Executing queries without parameters

- They PDO::query() method will run your query and return special object of PDOStatement class which can be roughly compared to a resource, returned by mysql_query()

```php
<?php
$stmt = $pdo->query('SELECT name FROM users');
while ($row = $stmt->fetch())
{
echo $row['name'] . "\n";
}
```

Prepared statements (1/3)

- For every query you run, if at least one variable is going to be used, you have to substitute it with a placeholder, then prepare your query, and then execute it, passing variables separately.
- In most cases, you need only two functions
  - prepare()
  - execute()

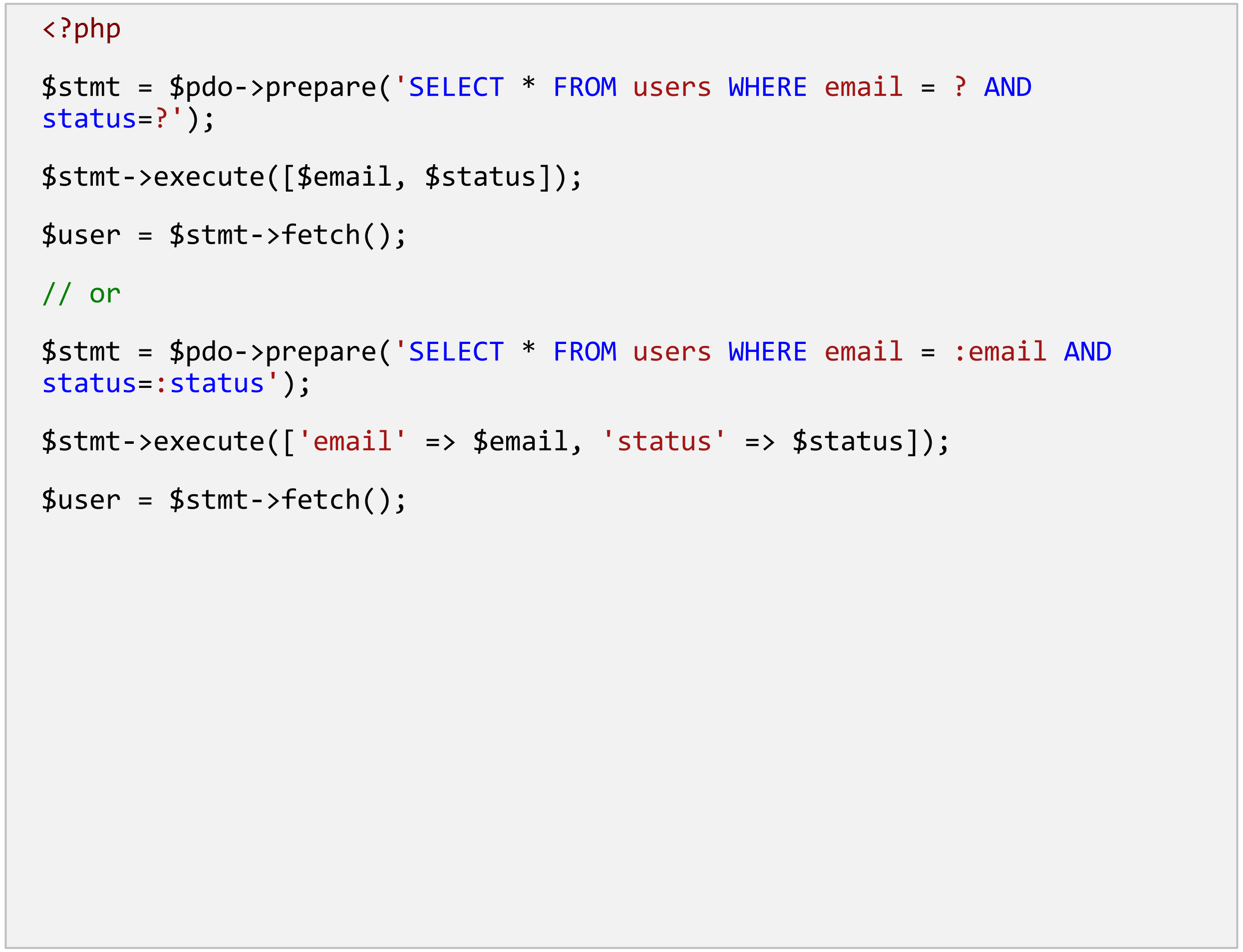
Prepared statements (2/3)

- First of all, you have to alter your query, adding placeholders in place of variables.
  - This:
    - `$sql = "SELECT * FROM users WHERE email = '$email' AND status= '$status'";`
  - will become
    - `$sql = 'SELECT * FROM users WHERE email = ? AND status=?';`
  - or
    - `$sql = 'SELECT * FROM users WHERE email = :email AND status=:status';`
- Note that PDO supports positional (?) and named (:email) placeholders, the latter always begins from a colon and can be written using letters, digits and underscores only. Also note that no quotes have to be ever used around placeholders.

Prepared statements (3/3)

- Having a query with placeholders, you have to prepare it, using the PDO::prepare() method. This function will return the same PDOStatement object we were talking about above, but without any data attached to it.

- Finally, to get the query executed, you must run execute() method of this object, passing variables in it, in the form of array.



```php
<?php

$stmt = $pdo->prepare('SELECT * FROM users WHERE email = ? AND
status=?');

$stmt->execute([$email, $status]);

$user = $stmt->fetch();

// or

$stmt = $pdo->prepare('SELECT * FROM users WHERE email = :email AND
status=:status');

$stmt->execute(['email' => $email, 'status' => $status]);

$user = $stmt->fetch();
```

Positioned VS named placeholders

Positional placeholders

- let you write shorter code, but are sensitive to the order of arguments
- you have to supply a regular array with values, You cannot mix positional and named placeholders in the same query.

Named placeholders

- make your code more verbose, they allow random binding order.
- You have to supply an associative array, where keys have to match the placeholder names in the query.

Prepared statements and multiple execution

- Sometimes you can use prepared statements for the multiple execution of a prepared query.
- It is slightly faster than performing the same query again and again, as it does query parsing only once.
- This feature would have been more useful if it was possible to execute a statement prepared in another PHP instance.

```php
<?php
$data = [
1 => 1000,
5 => 300,
9 => 200,
];
$stmt = $pdo->prepare('UPDATE users SET bonus = bonus + ? WHERE id = ?');
foreach ($data as $id => $bonus)
{
$stmt->execute([$bonus, $id]);
}
```

Running query statements

- You can use the method chaining and thus call execute() right along with prepare()

```php
<?php
$sql = "UPDATE users SET name = ? WHERE id = ?";
$pdo->prepare($sql)->execute([$name, $id]);
```

Getting row count

- The function “rowCount()” returns the number of affected rows by any query

```
<?php
$stmt = $pdo->prepare("DELETE FROM goods WHERE category = ?");
$stmt->execute([$cat]);
$deleted = $stmt->rowCount();
```

Getting data (1/2): foreach()

- The most basic and direct way to get multiple rows from a statement would be foreach() loop.

```php
<?php
$stmt = $pdo->query('SELECT name FROM users');
foreach ($stmt as $row)
{
echo $row['name'] . "\n";
}
```

Getting data (2/2): fetch()

- The “fetch” function fetches a single row from database, and moves the internal pointer in the result set, so consequent calls to this function will return all the resulting rows one by one

```php
<?php
$row = $stmt->fetch(PDO::FETCH_ASSOC);

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

Fetching all rows in a result

- PDOStatement::fetchAll() returns an array that consists of all the rows returned by the query.
- This function should not be used, if many rows has been selected. In such a case conventional while loop ave to be used, fetching rows one by one instead of getting them all into array at once. "Many" means more than it is suitable to be shown on the average web page.
- This function is mostly useful in a modern web application that never outputs data right away during fetching, but rather passes it to template.

```
<?php

$news = $pdo->query('SELECT * FROM news')->fetchAll(PDO::FETCH_CLASS,
'News');
```

Fetching a single column

- A neat helper function that returns value of the singe field of returned row.
- Very handy when we are selecting only one field.

```php
<?php
$stmt = $pdo->prepare("SELECT name FROM table WHERE id=?");
$stmt->execute([$id]);
$name = $stmt->fetchColumn();

$count = $pdo->query("SELECT count(*) FROM table")->fetchColumn();
```

Getting last inserted id

- The function lastInsertId returns the ID of the last inserted row or sequence value

- Returns the ID of the last inserted row, or the last value from a sequence object, depending on the underlying driver.

```php
<?php
$sql = "INSERT INTO names VALUES(:name)";
$stmt = $dbh->prepare(['name'=>'Test']);
$stmt->execute();
$lastId = $dbh->lastInsertId();
```

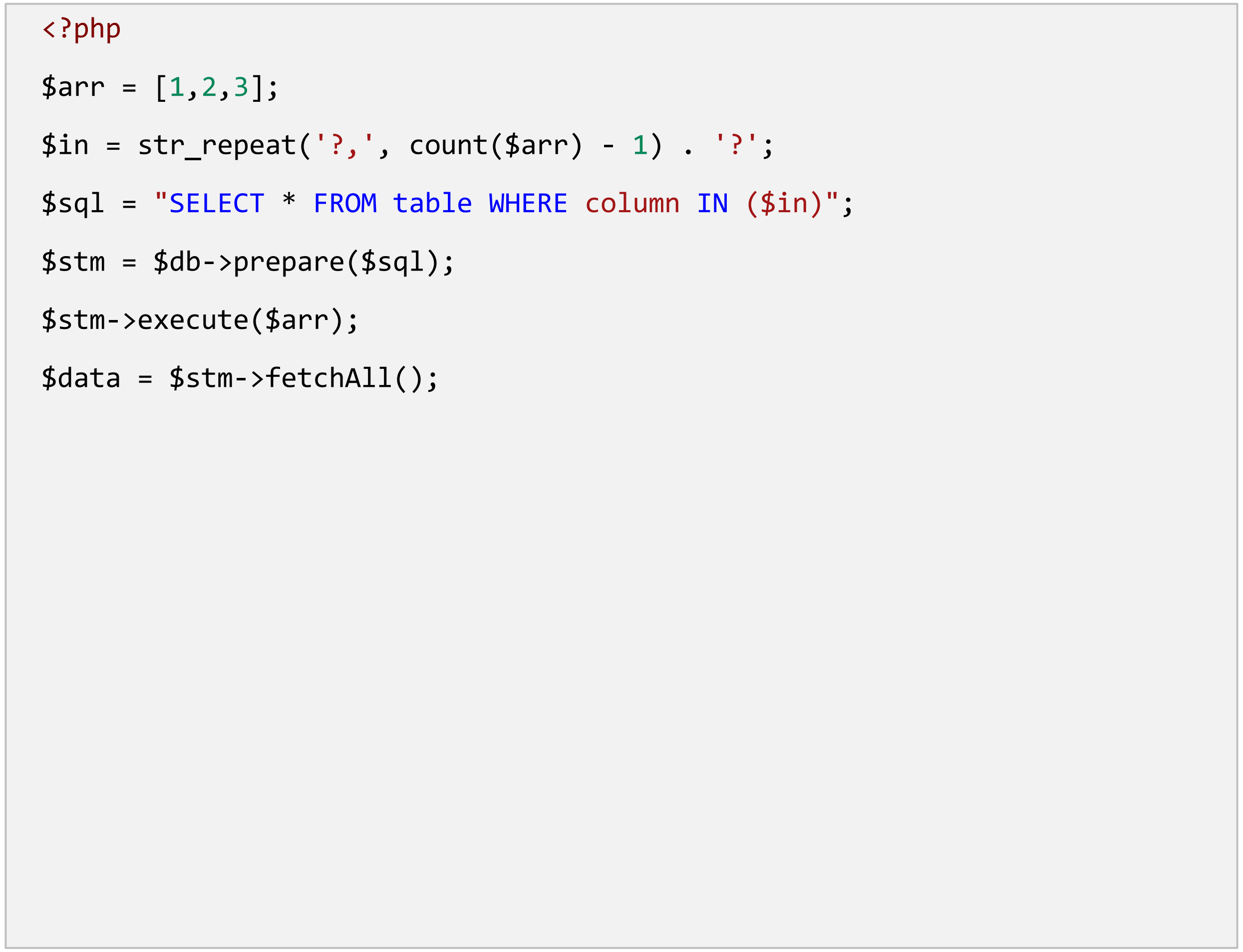
Prepared statements and LIKE clause

- Despite PDO's overall ease of use, there are some gotchas anyway, and I am going to explain some.
  - one of them is using placeholders with LIKE SQL clause.

```php
<?php
$search = "%$search%";
$stmt = $pdo->prepare("SELECT * FROM table WHERE name LIKE ?");
$stmt->execute([$search]);
$data = $stmt->fetchAll();
```

Prepared statements and IN clause

- Just like it was said above, it is impossible to substitute an arbitrary query part with a placeholder.
- Thus, for a comma-separated values, like for IN() SQL operator, one must create a set of ?s manually and put them into the query



```
<?php
$arr = [1,2,3];
$in = str_repeat('?,', count($arr) - 1) . '?';
$sql = "SELECT * FROM table WHERE column IN ($in)";
$stm = $db->prepare($sql);
$stm->execute($arr);
$data = $stm->fetchAll();
```

Transactions

What are transactions

- A transaction is a set of inter-dependent SQL statements that needs to execute in all-or-nothing mode.

- A transaction is successful if all SQL statements executed successfully.

- A failure of any statement will trigger the system to rollback to the original state to avoid data inconsistency.

Example of transaction

- A classic example of the transaction is a money transfer transaction from one bank account to another. It requires three steps:
  1. Check the balance of the transferred account to see if the amount is sufficient for the transfer.
  2. If the amount is sufficient, deduct the amount from the balance of the transferred account.
  3. Add the transfer amount to the balance of the receiving account.
- If an error occurs in the second step, the third step should not continue. In addition, if an error occurs in the third step, the second step must be reversed.
- The amounts of both bank accounts are intact in case of failure or adjusted correctly if the transaction is completed successfully.

Transactions: an example

- To successfully run a transaction, you have to make sure that error mode is set to exceptions, and learn three canonical methods:
  1. Start the transaction by calling the beginTransaction() method of the PDO object.
  2. Place the SQL statements and the commit() method call in a try block.
  3. Rollback the transaction in the catch block by calling the rollBack() method of the PDO object.

```
<?php
try {
$pdo->beginTransaction();
$stmt = $pdo->prepare("INSERT INTO users (name) VALUES (?)");
foreach (['Joe','Ben'] as $name)
{
$stmt->execute([$name]);
}
$pdo->commit();
}catch (Exception $e){
$pdo->rollback();
throw $e;
}
```

Transactions: some caveats

- Please note the following important things:

- you have catch an Exception, not PDOException, as it doesn't matter what particular exception aborted the execution.
- you should re-throw an exception after rollback, to be notified of the problem the usual way.
- also make sure that a table engine supports transactions (i.e. for Mysql it should be InnoDB, not MyISAM).

Parameter binding

How to bind parameters

- The function bindValue binds a value to a corresponding named or question mark placeholder in the SQL statement that was used to prepare the statement.

- It is similar to bindParam, though the parameter is passed by value, whereas in bindParam it’s by reference

```php
<?php
$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();
?>
```

Stored procedures

Stored procedures: an example

- Stored procedures are implemented with prepared statements:
  - They can be thought of as a kind of compiled template for the SQL that an application wants to run, that can be customized using variable parameters

```php
<?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();
?>
```

Running multiple queries

- When in emulation mode, PDO can run mutiple queries in the same statement, either via query() or prepare()/execute().
- To access the result of consequent queries one has to use PDOStatement::nextRowset()

```php
<?php
$stmt = $pdo->prepare("SELECT ?;SELECT ?");
$stmt->execute([1,2]);
do {
$data = $stmt->fetchAll();
var_dump($data);
} while ($stmt->nextRowset());
?>
```