Server-side
Web Development

Unit 06. PHP and databases.

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 PHP and databases

To store large amounts of data or very volatile information, the best option is always a database. In this section we are going to see how to work with relational databases from PHP. For the examples, we will use **MySQL / MariaDB** as the database server.

The PHP language has tools to launch data requests, or any other type of SQL instruction, to a database server. This returns the requested information and converts it into content that the client (the web browser) can display. This maximizes the dynamism of the application.

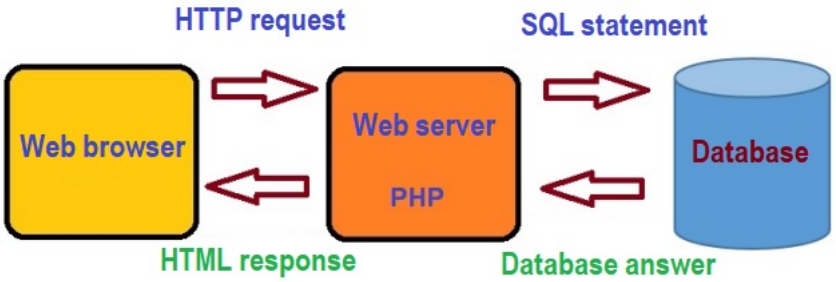

Figure 1: Access to databases

PHP has libraries to connect to a DBMS MySQL / MariaDB. Nowadays the most used is the **PDO** (***PHP Data Objects***) library, although there are also applications that work with the MySQLi library. The previous version, MySQL, is deprecated and is not included as of PHP 7. In this section we will work with PDO, but you can find additional examples for MySQLi.

An advantage of PDO is that it can also be used to connect to relational database management systems other than MySQL/MariaDB.

2 Security concerns when working with PHP and databases

For security reasons it is better not to use the **root** user. You should create a new user account with only the specific privileges it needs to work on the database that your website depends upon. You can even create a new user with all privileges only for certain user defined databases.

3 Connecting to a database

There are three methods of connecting to a MySQL Server from PHP:

- **MySQL library**: oldest method. It has been entirely removed from PHP since version 7.0.

- **MySQLi** library
- **PDO library**

There are a few differences between PDO and MySQLi, but the main one is that you can use the PDO library to connect to almost any database server (MySQL/MariaDB, PostgreSQL, Oracle server, Microsoft SQL Server…) For this reason, most recent PHP projects use the PDO library. However we will analyze both methods: MySQLi and PDO.

3.1 MySQLi library connection

Supports both: **procedural** and **object-oriented language**.

Object-oriented is better than procedural. Some MySQLi Functions are in the aliases and deprecated section of the PHP and MySQLi documentation because PHP is moving steadily in the direction of OO programming.

It's worth noting that the PDO library, which is considered the ideal for most DB code in PHP, is OOP oriented. It doesn't have a procedural interface.

There's also the point about the ability to create an extension class for your DB. For example:

```
class myDB extends mysqli {
// your own stuff here to extend and improve the base mysqli class
}
```

So, we will learn the MySQLi OOP Interface. We must create a connection object of the **mysqli class** this way:

```
$db=new mysqli($serverName,$userName,$password,$dbName);
```

This creates a connection with the selected database. We can check if the connection has been successful:

```
if($db->connect_error) {
    // connection error
}
```

To close the connection:

```
$db->close();
```

3.2 PDO library connection

Using PDO to establish a connection to a MySQL server:

```
new PDO('mysql:host=hostname;dbname=database', 'username','password')
```

In any case, it takes three arguments:

1. a string with the database type (mysql:), the hostname (host=localhost;), and the database name (dbname=ies)
2. the MySQL username
3. the MySQL password for that username

There may be a connection error. You should catch the exception using a `try...catch` statement:

```
try {
    $pdo = new PDO('mysql:host=localhost;dbname=xxxx','user', "1234");
    echo 'Database connection established.';
}
catch (PDOException $e) {
    echo 'Unable to connect to the database server.';
}
```

3.2.1 Configuring the connection

Our first task is to configure how our PDO object handles errors. By default, PDO switches to a *silent failure* mode after establishing a successful connection. We'd like our PDO object to throw a PDOException any time it fails doing our queries. We can configure it to do so by calling the PDO object's `setAttribute` method:

```
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
// With this line we set the PDO attribute that controls the error
// mode (PDO::ATTR_ERRMODE) to the mode that throws exceptions
// (PDO::ERRMODE_EXCEPTION)
```

By default, when PHP connects to MySQL/MariaDB, it uses the simpler ISO-8859-1 (or Latin-1) encoding instead of UTF-8.

In PDO the charset can be specified in the connection string:

```
$pdo = new PDO("mysql:host=$host;dbname=$db;charset=utf8", $user, $pass);
```

The charset option is only used since PHP 5.3.6, so take this into account when running an older version of PHP. In that case you should run the following statement after constructing the PDO object:

```
$pdo->exec('SET NAMES "utf8"');
```

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. It results in a faster web application:

```
$pdo = new PDO("mysql:host=$host;dbname=$db;charset=utf8", $user, $pass,
↪   array(PDO::ATTR_PERSISTENT => true));
```

> **Important**: 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.

Here's the complete code of the connection process:

```
try {
    $pdo = new PDO('mysql:host=localhost;dbname=xxxx;charset=utf8','user',
    ↪   '1234', array(PDO::ATTR_PERSISTENT => true));
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch (PDOException $e) {
    echo 'Unable to connect to the database server: ' . $e->getMessage();
    exit();
}
echo 'Database connection established.';
...
```

With `$pdo=null` we force the disconnection from the database server

4 Sending queries to the database

Once we have connected to the database, we need to send queries and any other SQL statement to be executed in the database server.

4.1 Queries with mysqli

Use the query() method of the mysqli connection instance to submit SQL statements:

```php
require_once 'login.php';
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
    die("Fatal Error: " . $conn->connect_error);
}
$query = 'SELECT * FROM comment'; // we create the SQL statement
$result = $conn->query($query); // we launch the query
if (!$result) die("Fatal Error");
$rows = $result->num_rows; // number of rows of the result
for ($j = 0 ; $j < $rows ; ++$j) {
    $row = $result->fetch_assoc();
    echo 'Value: '. htmlspecialchars($row['columnName']).'<br/>';
}
$result->close();
$conn->close();
```

Methods that we can use with the `result` object:

- `fetch_array()` method: creates an associative array, a numeric array, or both, based on the second parameter (MYSQLI_ASSOC, MYSQLI_NUM, or MYSQLI_BOTH)
- `fetch_assoc()` method: creates an associative array, using the data field names as the array keys.
- `fetch_row()` method: creates a numeric array, using numeric indexes for each data field (starting at 0, and using the data field order specified in the table or SELECT statement data field).
- `close()`: It's not necessary to free memory because php frees it at the end of the script. But in high traffic situations it can be useful.

4.1.1 Using prepared statements with mysqli

We should avoid SQL injection attacks in our application. It's very easy to launch this attack if we don't sanitize the data we include in the query. Anyway, the safest way of submitting data in a statement

is to use a **prepared statement**, which defines a template of the query you want to execute on the database server, and then sends the data separate from the template.

With a prepared statement, we create the query string as normal, but instead of including data values, we use a question mark as a placeholder for each value:

```
$sql = "INSERT INTO table VALUES (?, ?, ?...)";
```

Then we use the `prepare()` method to submit it:

```
$stmt = $conn->prepare($sql);
```

And then we will use `bind_param()` to pass the real values:

```
$stmt->bind_param("ssi",$value1,$value2,$value3...);
```

The first parameter (“ssi”) defines the type of each data values:

b: A blob data type value **i**: An integer data type value **d**: A double data type value **s**: A string data type value

Finally, we execute the prepared statement:

```
$stmt->execute();
```

It’s highly recommended to use prepared statements with operations that modify the database, but it’s a good practice to do so with queries when the query contains some PHP variable.

```
$sql = "SELECT * FROM users WHERE id=?"; // the SQL code
$stmt = $conn->prepare($sql); // we prepare the statement
$stmt->bind_param("i", $id); // bind the ? to real values
$stmt->execute(); // executes the statement
$result = $stmt->get_result(); // returns the result
while ($row = $result->fetch_assoc()) {
    echo $row['name'];
}
```

> It’s a good practice to separate the database interaction from the HTML output

Instead of the `while` loop we can fetch all the result with just one line:

```php
$data = $result->fetch_all(MYSQLI_ASSOC);
```

Now:

```php
foreach($data as $row) echo $row['column_name'];
```

4.1.2 Multiqueries with mysqli

We can execute one or multiple queries which are concatenated by a semicolon.

```php
$sql="INSERT INTO table VALUES(50,'RRHH1','VALENCIA');";
$sql .="INSERT INTO table VALUES(60,'RRHH2','VALENCIA');";
$sql .="INSERT INTO table VALUES(70,'RRHH3','MADRID')";
if ($conn->multi_query($sql)) { echo "Right"; }
else { echo "INSERT Failed"; }
```

Be careful with SQL injection: if the query contains any variable input then use parameterized prepared statements. Alternatively, the data must be properly formatted and all strings must be escaped using the `real_escape_string()` method or the methods we have seen in previous units like `strip_tags` or `htmlspecialchars`.

4.2 Queries with PDO

Unlike mysqli, PDO library has two different methods for SELECT queries and DML queries (INSERT, DELETE, UPDATE).

For DELETE, INSERT, and UPDATE queries (which modify the database content), we will use the `exec` method that returns the number of table rows (entries) that were affected by the query.

```php
$sql="...."; // the sql statement
$affectedRows=$pdo->exec($sql); // sql execution
if($affectedRows == 0) {
    echo "No rows inserted/updated/deleted";
}
```

For SELECT queries, we will use the `query` method that returns a list of all the rows (entries) returned from the query.

```php
$sql="..."; // the sql query
$result=$pdo->query($sql); // sql execution
```

Now we can go through the result with a loop and the `fetch` method of the PDOStatement class to which the result belongs. The `fetch` method returns the next row in the result set as an array. When the rows end `fetch` returns false.

A complete example:

```php
try {
    $pdo = new PDO('mysql:host=localhost;
                    dbname=databaseName;
                    charset=utf8','userName',
                    'password');
    $pdo->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
    $sql = 'SELECT field1,field2... FROM databaseTable';
    $result = $pdo->query($sql);
    while ($row = $result->fetch()) {
        echo $row['field1'];
        echo $row['field2'];
        ...
    }
} catch (PDOException $e) {
    echo 'Something is wrong with the database server: '
    .$e->getMessage().' in '.$e->getFile().':'.$e->getLine();
}
```

4.2.1 Prepared statements with PDO

```php
$sql = 'INSERT INTO SQLTable values (:v1,:v2...)';
$stmt = $pdo->prepare($sql);
$stmt->bindValue('realValue1','realValue2'...);
$stmt->execute();
```

An alternative way:

```php
$stmt->execute([':v1'=>'realValue1',':v2'=>'realValue2'...]);
```

We can work using the ? symbol as well.

```php
$sql = 'INSERT INTO SQLTable values (?,?)';
$stmt = $db->prepare($sql);
$value1 = "Good idea!";
$value2 = "CURDATE()";
$stmt->bindParam(1, $comment);
$stmt->bindParam(2, $date);
$stmt->execute();
```

An alternative way:

```php
$stmt->execute([$comment,$date]);
```

We can do queries with a prepared statement and store the result in an indexed array, associative array or in a new object of the requested class. We can tell which fetch mode to use with the methods `setFetchMode` or `fetch` of the `Statement` class:

```php
$stmt = $conn->prepare("SELECT * FROM persons");
$stmt->fetch(PDO::FETCH_ASSOC);
$stmt->execute();
$result = $stmt->fetchAll(); //$result is an associative array
```

If we want to store the result in an object, we need to add the `PDO::FETCH_CLASS` and `PDO::FETCH_PROPS_LATE` methods:

```php
$stmt = $conn->prepare("SELECT * FROM persons");
$stmt->fetch((PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE, 'Person');
$stmt->execute();
$result = $stmt->fetchAll(); //$result is an array of Person objects
```

For a single result, for instance a select by the primary key, we use the fetch method:

```php
$person = $stmt->fetch(); //$person is an object of the Person class
```

The inserts, updates and deletes are similar, but we don't need the `fetch` or `fetchAll` methods. For example:

```
$stmt = $conn->prepare("UPDATE cars SET brand=:brand, model=:model WHERE
↪  id=:id");
$stmt->execute(['id'=>$car->getId(), 'brand'=>$car->getBrand(),
↪  'model'=>$car->getModel()]);
$numrows = $stmt->rowCount();
```

We can use the method `rowCount` to know how many rows have been affected by the query.

5 Transactions

Database transactions ensure that a set of data changes will only be made permanent if every statement is successful.

PDO run "auto-commit" mode by default: It means that every query that you run has its own implicit transaction.

We can define a transaction with multiple queries by using the methods `beginTransaction` and `commit`. Between them, we define the statements that we want to execute, but the whole bunch of statements will be executed only when `commit` is reached.

If one of the statements causes an exception, we can use the `rollback` method to undo the changes and the database will return to its state before `beginTransaction`.

```
try {
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $pdo->beginTransaction();
    $pdo->exec("insert into staff(id, first, last) values (23,
    ↪  'Joe','Smith')");
    $pdo->exec("insert into salarychange(id, amount, changedate)
    values (23, 50000, NOW())");
    $pdo->commit();//save changes
} catch (Exception $e) {
    $pdo->rollBack(); //undo changes
    echo "Failed: " . $e->getMessage();
}
```

6 Dates with PHP and MySQL/MariaDB

In MySQL/MariaDB we have 5 types to represent date and time data:

- DATE: YYYY-MM-DD
- TIME: HH:MM:SS
- DATETIME: YYYY-MM-DD HH:MM:SS
- TIMESTAMP: same as DATETIME but with a different range of dates because of their internal implementation
- YEAR: YYYY

We need to know which type are we working with in order to prepare the date in its specific format. If we have dates in spanish format we must convert them to the database format.

An example with a DATE format field in the table:

```php
// this line converts a d-m-Y date into a Y-m-d one
$date = DateTime::createFromFormat('d-m-Y', $birthday)->format('Y-m-d');
// now we can insert the date into the table
$query = "INSERT INTO users(id, name, birthday)
 ↪  VALUES('$id','$name','$date')";
// we can use the SQL DATE_FORMAT function to get the date from
// the database in a certain format
// SELECT DATE_FORMAT(birthday, '%D %M %Y') from users;
```

Another interesting option is to use a `bigint` or `timestamp` column in the database table and save the date in PHP timestamp format (the amount of seconds from the beginning of Linux time). We can simply do this:

```php
$today=date_create();
$todayTimeStamp=$today->getTimeStamp();
echo date("d-m-Y",$todayTimeStamp)." - ".$todayTimeStamp;
```

We will get:

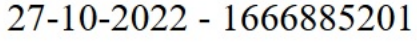

Figure 2: String Date and TimeStamp date

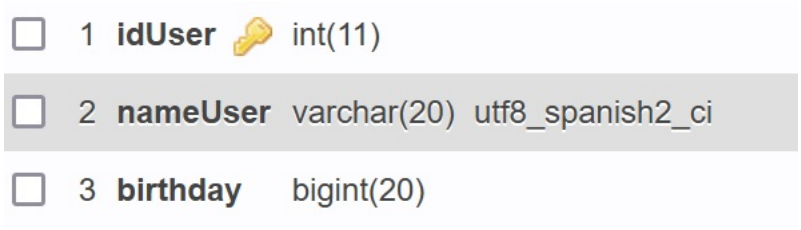

Figure 3: Table with a TIMESTAMP date

Now, if we have a table like this in our database:

We can insert a user doing:

```
$today=date_create();
$todayTimeStamp=$today->getTimeStamp();
echo date("d-m-Y",$todayTimeStamp)." - ".$todayTimeStamp;
try {
    $pdo = new PDO('mysql:host=localhost;dbname=daw2223','root', '1234');
}
catch (PDOException $e) { die("Connecting failure"); }
$sql="INSERT INTO user VALUES(NULL,'Fidel',$todayTimeStamp)";
$rows=$pdo->exec($sql);
if($rows==0) { echo "Insert failure";}
$pdo=null;
```

If we take a look to the database, this is what we see:

idUser	nameUser	birthday
2	Fidel	1666886035

Figure 4: PHP timestamp in the database

Now, to retrieve the date:

```
$sql="SELECT nameUser, birthday FROM user";
$result=$pdo->query($sql);
while($row=$result->fetch()) {
    echo "<strong>Name:</strong> ".
            $row['nameUser'].
            " <strong>Birthday:</strong> ".
```

```
            date("d-m-Y",$row['birthday']);
    echo "<br/>";
}
```

And this is the result:

Name: Fidel **Birthday:** 27-10-2022

Figure 5: Date retrieved from the database

7 Passwords in the database

It's a good practice not to save passwords to the database in plain text format.

The PHP function `password_hash()` creates a new password hash from a string password using a strong one-way hashing algorithm.

```
$strongPass = password_hash($textPassword, PASSWORD_DEFAULT);
$sql = "INSERT INTO login(usu, pass) values ('$name','$strongPass')";
```

We have used the PASSWORD_DEFAULT algorithm, but you can find other options in the PHP documentation.

We can't decode the password again into a string password, but when the user tries to validate we can compare the password (in plain text format) introduced by the user with the hashed password in the database using the function `password_verify()`.

```
$sql = "SELECT pass FROM login WHERE usu='John'";
$result = $conn->query($sql);
if (!$result) die("Fatal Error");
$row = $result->fetch_assoc();
if (password_verify($pass, $row['pass'])) echo 'Correct Password';
```