

# Server-side Web Development

## Unit 06. Databases operations. Guided example.



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

## Index

<b>1 Database creation</b>	<b>2</b>
<b>2 Application structure</b>	<b>3</b>
2.1 Login.php . . . . .	4
2.2 IDbAccess.php . . . . .	5
2.3 DBConnection.php . . . . .	6
2.4 LoginDao.php . . . . .	6
2.4.1 Select methods . . . . .	7
2.4.2 Insert method . . . . .	8
2.4.3 Delete method . . . . .	9
2.4.4 Update method . . . . .	9
2.5 login_list.php . . . . .	10
2.6 login_form.php . . . . .	11

In this example we're going to do the basic database operations. We will use a simple database to manage emails and passwords for the users of a web site.

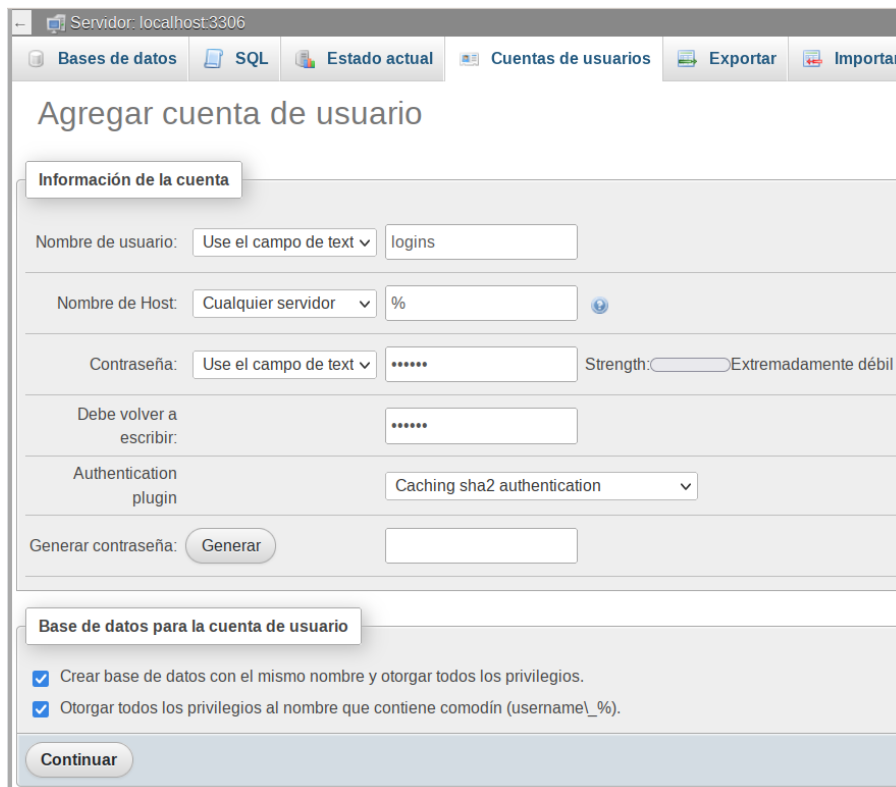
## 1 Database creation

The schema of our database is pretty simple: a single table with the next structure:

- **id**: INT, auto-increment, primary key
- **email**: VARCHAR(100), not null
- **password**: VARCHAR(100), not null

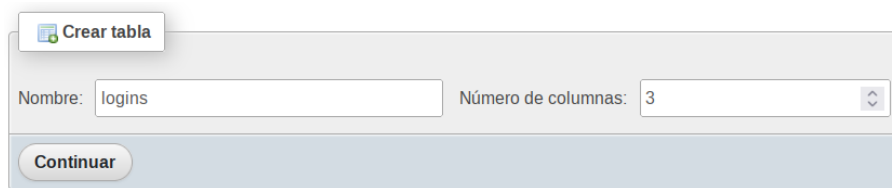
We can do all the process with **PhpMyAdmin**. If you don't have your MySQL database installed and configured, follow the process explained in Unit 1.

First of all, login to MySQL console or to PhpMyAdmin with an administrator user and create a new user named **logins** along with a database with the same name:

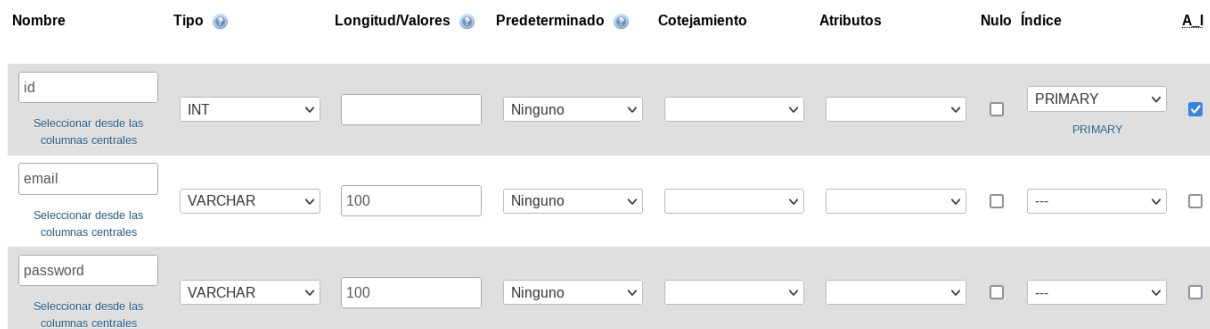


**Figure 1:** User and database creation

Next, go to the new database (**logins**) and create a new table named **logins**:

**Figure 2:** Table creation

Enter in the `logins` table and create the schema:

**Figure 3:** Columns creation

The same process can be done using the MySQL command line.

Additionally we can add some sample data:

```
INSERT INTO `logins` (`id`, `email`, `password`) VALUES (NULL,  
→ 'peter@mail.com', '12345678'), (NULL, 'mary@mail.com', '34567890');
```

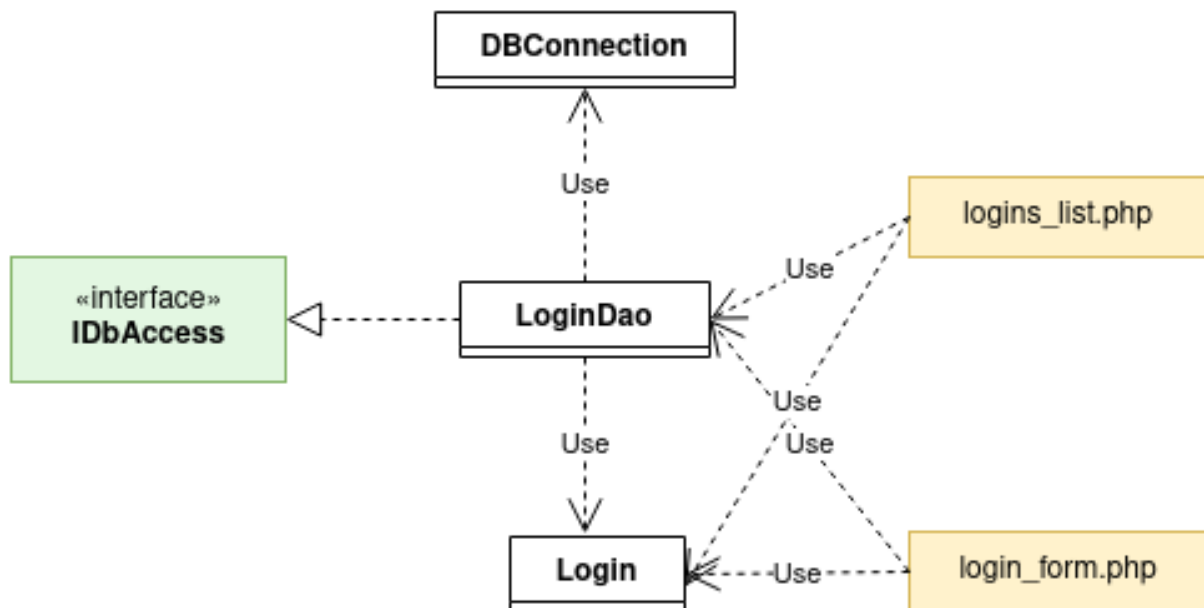
**Security warning:** In this sample we are storing the passwords in plain text to make it easier, but **don't do it in a real project!** Any attacker could see the stored passwords.

Also, you can run the import script in order to create the database with the sample data (this script doesn't create the `logins` user):

```
sudo mysql < logins.sql
```

## 2 Application structure

We want our application to have the next structure:



**Figure 4:** Class diagram

- **logins\_list.php**: script that will show (SELECT) the list of all the entries (id, and email), and a button to create a new entry.
- **login\_form.php**: script that will show (SELECT by id) the data of a login (id, login and password). In this script we can change the login data (UPDATE), delete the entry (DELETE), and create a new one (INSERT).
- **Login.php**: a class to store and retrieve the data of a single login.
- **LoginDao.php**: a Data Access Object class, used to do the operations with the database.
- **IDbAccess.php**: interface with the methods that must be implemented by LoginDao.php.
- **DBConnection**: a convenience class used to connect to the database.

## 2.1 Login.php

This is a simple data class, with the private fields and their setters and getters:

```

<?php declare(strict_types=1);

class Login
{
    private int $id;
    private string $email;
    private string $password;

```

```
public function __construct()
{
    $this->id = 0;
    $this->email = "";
    $this->password = "";
}

//Getters and setters...
```

Note: At this point it could be a good idea to create a directory for the classes, named `models` or `classes`, for example.

## 2.2 IDbAccess.php

This is an **interface** with the methods that must be implemented by `LoginDao`:

```
interface IDbAccess
{
    public static function getAll();
    public static function select($id);
    public static function insert($object);
    public static function delete($object);
    public static function update($object);
}
```

The reason for making this interface is because if we have several classes that need to access the database, they will all have the same operations (select, update, etc), making it easy to use. For the same reason we have made it as generic as possible.

## 2.3 DBConnection.php

This is a class with a static method which has the parameters for connecting to our database. The method `connectDB()` returns the connection to the DB or `null` if a problem occurs:

```
class DBConnection
{
    //Database connection data
    private static $servername = "localhost";
    private static $dbname = "logins";
    private static $username = "logins";
    private static $password = "your_password";

    public static function connectDB(): ?PDO
    {
        $servername = self::$servername;
        $dbname = self::$dbname;
        $username = self::$username;
        $password = self::$password;

        try {
            $conn = new PDO("mysql:host=$servername;dbname=$dbname",
                $username, $password,
                array(PDO::ATTR_PERSISTENT => true));
            // set the PDO error mode to exception
            $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            return $conn;
        } catch (PDOException $e) {
            echo "Connection failed: " . $e->getMessage();
            return null;
        }
    }
}
```

In the line 19 we are setting the default error mode to *Exception* (which is the default since PHP 8.0) that throws a **PDOException** and sets its properties to reflect the error code and error information.

## 2.4 LoginDao.php

This class must implement the interface `IDbAccess`, defining the methods used to access the database. All database work is done here.

First of all, we declare strict types, import the required files and declare that this class will implement the methods from the interface IDbAccess:

```
<?php declare(strict_types=1);
require_once __DIR__.'./Login.php';
require_once __DIR__.'../DBConnection.php';
require_once __DIR__.'../IDbAccess.php';

class LoginDao implements IDbAccess {
    ...
}
```

Let's see each method individually:

### 2.4.1 Select methods

Here we have 2 methods, one for selecting all the rows of the table and the other one for selecting a single row by its id. The selectAll method is:

```
public static function getAll(): ?array
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("SELECT * FROM logins");
        $stmt->setFetchMode(PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE,
            'Login');
        $stmt->execute();
        return $stmt->fetchAll();
    } else {
        return null;
    }
}
```

This method connects to the DB using the static method of the DBConnection class. Then, if the connection is not null, it uses a prepared statement with the query.

With the method `$stmt->setFetchMode(PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE, 'Login')` we are indicating that the statement must return an object of the class Login. Pay attention to the `PDO::FETCH_PROPS_LATE` flag: without it the result will consist in an empty object.



Finally, the script executes the statement, fetches all the rows in an array with the method `fetchAll()` and returns it. If an error occurs, we return `null`, so that the script that uses this method could check if it has been successful before reading the data.

The **select by id** method is as follows:

```
public static function select($id): ?Login
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        // The user input is automatically quoted, so there is no risk of a
        // → SQL injection attack.
        $stmt = $conn->prepare("SELECT * FROM logins WHERE id = :id");
        $stmt->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
            → 'Login');
        $stmt->execute(['id' => $id]);
        $login = $stmt->fetch();
        if ($login)
            return $login;
    }
    return null;
}
```

This method is similar to the previous one, but passing the `id` to the prepared statement using a **named placeholder (:id)**:

```
$stmt = $conn->prepare("SELECT * FROM logins WHERE id = :id");
...
$stmt->execute(['id' => $id]);
```

The `fetch()` method returns a `Login` object or `false` on failure, so we can check the returned value and return `null` if an error occurs.

### 2.4.2 Insert method

This method has a `Login` object as parameter and returns the number of rows inserted to the database (0 on failure, 1 on success). The steps are the same as before, but preparing an **insert statement**. Here, the use of a prepared statement is essential to avoid SQL injection attacks.

```
public static function insert($object): int
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("INSERT INTO logins (id, email, password)
        ↳ VALUES (:id, :email, :password)");
        $stmt->execute(['id'=>null, 'email'=>$object->getEmail(),
        ↳ 'password'=>$object->getPassword()]);
        return $stmt->rowCount(); //Return the number of rows affected
    }
    return 0;
}
```

Note that we are passing `null` as `id` because in the database schema the `id` is an auto-increment integer, so passing `null` lets the database assign an automatic `id` to the field.

The `rowCount()` method returns the number of rows affected by the query, so we can know if the insertion has been successful.

### 2.4.3 Delete method

This method is similar to the previous one, except for the query:

```
public static function delete($object): int
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("DELETE FROM logins WHERE id=:id");
        $stmt->execute(['id'=>$object->getId()]);
        return $stmt->rowCount(); //Return the number of rows affected
    }
    return 0;
}
```

### 2.4.4 Update method

This method is pretty similar to the previous ones: we create the connection and the statement, retrieve the data from the Login object passed as parameter, do the query and return the number of rows affected (0 on failure):

```
public static function update($object): int
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("UPDATE logins SET email=:email,
        ↳ password=:password WHERE id=:id");
        $stmt->execute(['id'=>$object->getId(),
        ↳ 'email'=>$object->getEmail(),
        ↳ 'password'=>$object->getPassword()]);
        return $stmt->rowCount(); //Return the number of rows affected
    }
    return 0;
}
```

## 2.5 login\_list.php

In this script we make an index page with a table showing the Id and email of each entry. Each row has a button which opens a form to edit the email and password fields. We also have a button to open an empty form for a new entry:

# Login list

Create new login

	ID	Email
Edit/View	1	peter@mail.com
Edit/View	2	mary@mail.com

**Figure 5:** login\_list.php

In the first part of the script, we import the Login and LoginDao classes and create an array of login objects using the LoginDao::getAll() method:

```
<?php
require_once __DIR__.' /Login.php';
require_once __DIR__.' /LoginDao.php';

$logins = LoginDao::getAll();
?>
```

In the HTML body, we create a form to submit the chosen id to the login\_form script:

```
<form action="login_form.php" method="get" id="form1" style="border: none">
```

Finally we create a table populated with the next PHP code:

```
<?php
foreach ($logins as $login){
    echo "    <tr>\n";
    echo "        <td style='text-align: center'> <button type='submit'
    ↪ form='form1' name='id' value='"
    . $login->getId() . "' > Edit/View </button> </td>";
    echo " <td> " . $login->getId() . "</td>";
    echo " <td> " . $login->getEmail() . "</td>";
    echo "    </tr>\n";
}
?>
```

Each button has the object id as value, which is the data passed with the method GET.

Note that we are using objects to get the data (\$login->getId(), etc...).

## 2.6 login\_form.php

In this script we can create/update or delete an entry. It is similar to the form of the Unit 4, but here we can view and edit the password field. We can see each entry id too, but in a read only field. The “Save” button creates a new login and inserts it in the database, if the login’s id is 0 (that means it’s a new entry), or updates it otherwise. The delete button deletes an existing entry.

First of all, we read the id sent with the GET method from the login\_list script and perform a query to the database with the LoginDao::select method and store the object returned in the \$login variable:

```
if($_SERVER['REQUEST_METHOD'] == "GET") {  
    if (isset($_REQUEST["id"]) && !empty($_REQUEST["id"])) {  
        $login = LoginDao::select($_REQUEST['id']);  
    }  
}
```

After that, we do the same for the POST method, used for data validation as in the Unit 5 guided example.

If no errors are found in the validation process (checking the `$err` boolean variable), the next step is to perform the requested operations.

If the button pressed is “Submit”, we check the entry `id`: if it is 0 the operation is an `insert`:

```
if (!$err) {  
    if(isset($_POST['submit'])) {  
        if($login->getId() == 0){  
            //New login  
            try {  
                LoginDao::insert($login);  
                $opMsg = "New login inserted";  
                $login = new Login(); //If no errors, make a new empty  
                                     login to clear the fields  
            } catch (Exception $e) {  
                echo "Error inserting login: " . $e->getMessage();  
            }  
        } ...  
    }  
}
```

We use the `$opMsg` variable to show the value of the operation performed.

If the operation has been successful, we assign a new object to the `$login` variable ( `$login = new Login()` ) in order to clear the inputs when the form is reloaded.

If the value of the `id` is different from 0, then the operation is an `update`:

```
else {  
    //Update Login  
    LoginDao::update($login);  
    $opMsg = "Login updated";  
    $login = new Login();  
}
```

If the pressed button is “Delete”, we perform the delete operation:

```
if(isset($_POST['delete']) && $login->getId() != 0) {
    //Delete login
    LoginDao::delete($login);
    $opMsg = "Login deleted";
    $login = new Login();
}
```

In the HTML body, we show the \$opMsg message and show the form:

```
<h1>Edit login</h1>
<p> <?= $opMsg ?> </p>

<form method="post" action="<?php echo
↳ htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    <label> ID: <br>
        <input type="text" name="id" readonly value="<?= $login->getId();?>">
    </label>
    <label> E-mail: <br>
        <input type="text" name="email" value="<?= $login->getEmail();?>">
        ↳ <span class="error"> <?= $emailErr; ?> </span>
    </label>
    <label>Password: <br>
        <input type="text" name="pass" value="<?=
↳ $login->getPassword();?>"> <span class="error"> <?= $passErr;
↳ ?> </span>
    </label>

    <input type="submit" name="submit" value="Save">
    <input type="submit" name="delete" value="Delete" <?= $login->getId()
↳ == 0 ? "disabled" : "" ?> >
</form>
```

The validation process is similar to the one shown in unit 4, but here we are working with objects.

You can get all the code from the [GitHub repository](#).