

# Server-side Web Development

## Unit 06. Databases operations. Guided example.



IES Jaume II El Just  
Tavernes de la Valldigna  
Departament d'Informàtica  
Curs 2024-25

## Index

<b>1 Database creation</b>	<b>2</b>
<b>2 Application structure</b>	<b>4</b>
2.1 Provider.php . . . . .	4
2.2 IDbAccess.php . . . . .	5
2.3 DBConnection.php . . . . .	6
2.4 ProviderRepository.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 . . . . .	10
2.5 provider_list.php . . . . .	11
2.6 provider_form.php . . . . .	12

In this example we're going to do the basic database operations. We will use a simple database to manage providers (see Guided example for Unit 4).

## 1 Database creation

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

- **id**: INT, auto-increment, primary key
- **name**: VARCHAR(100), not null
- **email**: VARCHAR(100), not null
- **cif**: VARCHAR(9), 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, provider to MySQL console or to PhpMyAdmin with an administrator user and create a new user named **providers** along with a database with the same name:

Agregar cuenta de usuario

Información de la cuenta

Nombre de usuario:

Use el campo de te ▾ providers

Nombre de Host:

Cualquier servidor ▾ % ⓘ

Contraseña:

Use el campo de te ▾ ..... Strength: (.....) Extremadamente débil

Debe volver a escribir:

.....

Authentication plugin

Caching sha2 authentication ▾

Generar contraseña:

Generar

Base de datos para la cuenta de usuario

☒ Crear base de datos con el mismo nombre y otorgar todos los privilegios.

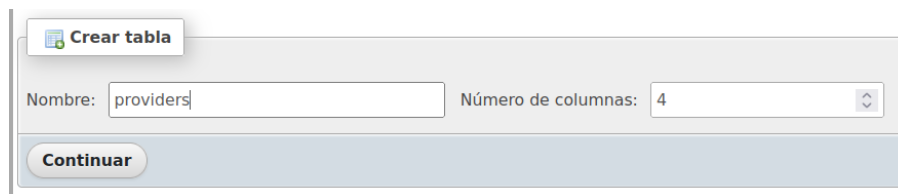
☐ Otorgar todos los privilegios al nombre que contiene comodín (username\_%).

Privilegios globales

☐ Seleccionar todo

**Figure 1:** User and database creation

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



**Figure 2:** Table creation

Enter in the providers table and create the schema:

Nombre	Tipo	Longitud/Valores	Predeterminado	Cotejamiento	Atributos	Nulo índice	A_I
id <small>Seleccionar desde las columnas centrales</small>	INT		Ninguno			<input type="checkbox"/> PRIMARY	<input checked="" type="checkbox"/>
name <small>Seleccionar desde las columnas centrales</small>	VARCHAR	100	Ninguno			<input type="checkbox"/> ---	<input type="checkbox"/>
email <small>Seleccionar desde las columnas centrales</small>	VARCHAR	100	Ninguno			<input type="checkbox"/> ---	<input type="checkbox"/>
cif <small>Seleccionar desde las columnas centrales</small>	VARCHAR	9	Ninguno			<input type="checkbox"/> ---	<input type="checkbox"/>

**Figure 3:** Columns creation

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

```
CREATE TABLE `providers`.`providers` ( `id` INT NOT NULL AUTO_INCREMENT ,
↪ `name` VARCHAR(100) NOT NULL , `email` VARCHAR(100) NOT NULL , `cif`
↪ VARCHAR(9) NOT NULL , PRIMARY KEY (`id`)) ENGINE = InnoDB;
```

Additionally we can add some sample data:

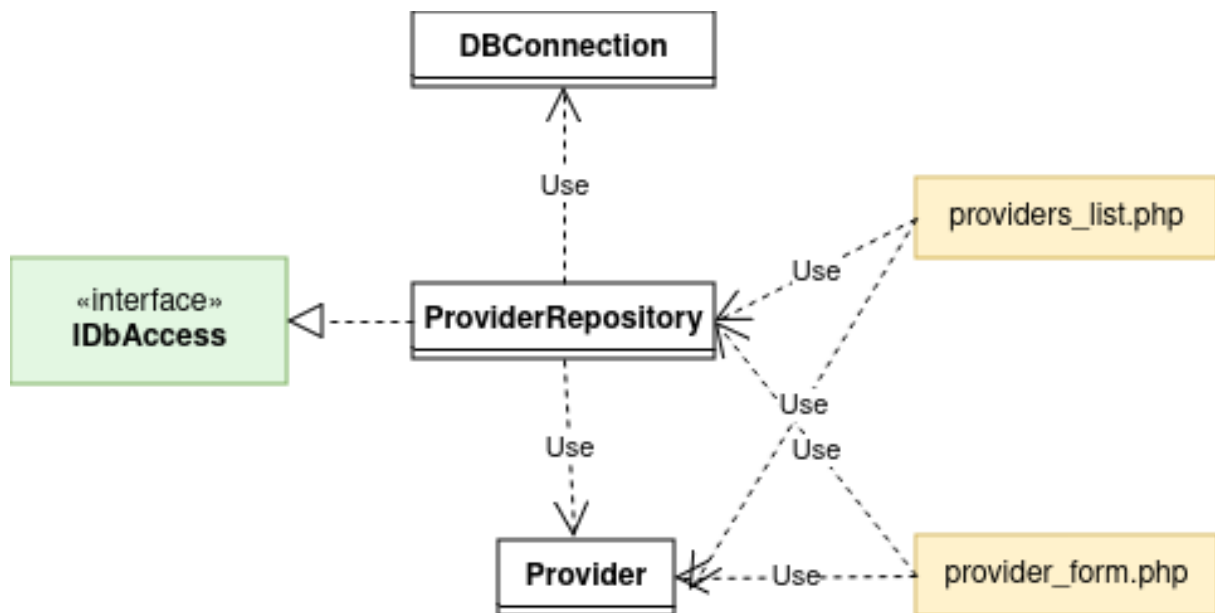
```
INSERT INTO `providers` (`id`, `name`, `email`, `cif`) VALUES (NULL, 'ACME',
↪ 'acme@mail.com', 'A12345678'), (NULL, 'Oscorp', 'oscorp@mail.com',
↪ 'B34567890');
```

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

```
sudo mysql < providers.sql
```

## 2 Application structure

We want our application to have the next structure:



**Figure 4:** Class diagram

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

### 2.1 Provider.php

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

```
<?php
declare(strict_types=1);

class Provider
{
    private int $id;
    private string $name;
    private string $email;
    private string $cif;

    public function __construct()
    {
        $this->id = 0;
        $this->name = '';
        $this->email = '';
        $this->cif = '';
    }

    //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 `ProviderRepository`:

```
<?php
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 that 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 = "providers";
    private static $username = "providers";
    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 22 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 ProviderRepository.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__.'./Provider.php';
require_once __DIR__.'../DBConnection.php';
require_once __DIR__.'../IDbAccess.php';

class ProviderDao 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 providers");
        $stmt->setFetchMode(PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE,
            'Provider');
        $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, 'Provider')` we are indicating that the statement must return an object of the class `Provider`. 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): ?Provider
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        // The user input is automatically quoted, so there is no risk of a
        // → SQL injection attack.
        $stmt = $conn->prepare("SELECT * FROM providers WHERE id = :id");
        $stmt->setFetchMode(PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE,
            → 'Provider');
        $stmt->execute(['id' => $id]);
        $provider = $stmt->fetch();
        if($provider) {
            return $provider;
        }
    }
    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 providers WHERE id = :id");
...
$stmt->execute(['id' => $id]);
```

The `fetch()` method returns a `Provider` 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 `Provider` object as parameter and returns the `id` of the last row inserted to the database, or `false` on failure. 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): false | string
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("INSERT INTO providers (id, name, email,
        ↪ cif) VALUES (:id, :name, :email, :cif)");
        $stmt->execute([
            'id' => null,
            'name' => $object->getName(),
            'email' => $object->getEmail(),
            'cif' => $object->getCif()
        ]);
        return $conn->lastInsertId();
    }
    return false;
}
```

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 `lastInsertId()` method returns the ID of the last inserted row or `false` if failed, 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 providers WHERE id=:id");
        $stmt->execute(['id'=>$object->getId()]);
        return $stmt->rowCount(); //Return the number of rows affected
    }
    return 0;
}
```

The `rowCount()` method returns the number of rows affected by the last SQL statement.

### 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 Provider 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 providers SET name = :name, email =
        ↳ :email, cif = :cif WHERE id = :id");
        $stmt->execute([
            'id' => $object->getId(),
            'name' => $object->getName(),
            'email' => $object->getEmail(),
            'cif' => $object->getCif()
        ]);
        return $stmt->rowCount(); //Return the number of rows affected
    }
    return 0;
}
```

## 2.5 provider\_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:

	Id	Name	Email	CIF
<input type="button" value="Edit/View"/>	1	ACME	acme@mail.com	A12345670
<input type="button" value="Edit/View"/>	2	Oscorp	oscorp@mail.com	B34567890

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

In the first part of the script, we import the Provider and ProviderDao classes and create an array of provider objects using the ProviderDao::getAll() method:

```
<?php
require_once __DIR__ . '/classes/ProviderRepository.php';
require_once __DIR__ . '/classes/Provider.php';

$providers = ProviderRepository::getAll();
?>
```

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

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

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

```
<?php
foreach ($providers as $provider){
    echo "    <tr>\n";
    echo "        <td style='text-align: center'> <button type='submit'
  form='form1' name='id' value='"
```

```
        . $provider->getId() . "' > Edit/View </button> </td>";  
    echo " <td> " . $provider->getId() . "</td>";  
    echo " <td> " . $provider->getName() . "</td>";  
    echo " <td> " . $provider->getEmail() . "</td>";  
    echo " <td> " . $provider->getCif() . "</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 (`$provider->getId()`, etc...).

## 2.6 provider\_form.php

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

# Edit Provider

Id:	<input type="text" value="1"/>
Name:	<input type="text" value="ACME"/>
Email:	<input type="text" value="acme@mail.com"/>
CIF:	<input type="text" value="A12345670"/>
<input type="button" value="Save"/> <input type="button" value="Delete"/>	

**Figure 6:** Provider form

First of all, we read the `id` sent with the GET method from the `provider_list` script and perform a query to the database with the `ProviderDao::select` method and store the object returned in the `$provider` variable:

```
if ($_SERVER["REQUEST_METHOD"] == "GET") {  
    if (!empty($_REQUEST["id"])) {  
        $provider = ProviderRepository::select($_REQUEST['id']);  
    }  
}
```

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

If no errors are found in the validation process (checking if the `$error` array is empty), 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 (empty($errors)) {  
    if(isset($_POST['save'])) {  
        if ($provider->getId() == 0) {  
            //New provider  
            try {  
                $id = ProviderRepository::insert($provider);  
                if ($id) {  
                    $opMsg = "New provider inserted with id $id";  
                    $provider = new Provider(); //If no errors, make a new  
                    // empty provider to clear the fields  
                }  
            } catch (Exception $e) {  
                echo "Error inserting provider: " . $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 `$provider` variable (`$provider = new Provider()`) 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 {
    try {
        ProviderRepository::update($provider);
        $opMsg = "Provider updated";
        $provider = new Provider();
    } catch (Exception $e) {
        echo "Error updating provider: " . $e->getMessage();
    }
}

```

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

```

if (isset($_POST['delete']) && $provider->getId() != 0) {
    try {
        ProviderRepository::delete($provider);
        $opMsg = "Provider deleted";
        $provider = new Provider();
    } catch (Exception $e) {
        echo "Error deleting provider: " . $e->getMessage();
    }
}

```

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

```

<h1>Edit Provider</h1>
<p> <?= $opMsg ?> </p>

<form method="post" action="<?php echo
    ↪ htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    <label for="id">Id:</label>
    <input type="text" id="id" name="id" value="<?= $provider->getId() ??
    ↪ ' ' ?>" readonly>

    <label for="name">Name:</label>
    <input type="text" id="name" name="name" value="<?=
    ↪ $provider->getName() ?? ' ' ?>">
    <span class="error"> <?= $errors['name'] ?? ' ' ?> </span> <br><br>

    <label for="email">Email:</label>
    <input type="text" id="email" name="email" value="<?=
    ↪ $provider->getEmail() ?? ' ' ?>">

```

```
<span class="error"> <?= $errors['email'] ?? ' ' ?> </span> <br><br>

<label for="cif">CIF:</label>
<input type="text" id="cif" name="cif" value="<?= $provider->getCif()
→ ?? ' ' ?>">
<span class="error"> <?= $errors['cif'] ?? ' ' ?> </span><br><br>

<input type="submit" value="Save" name="save">
<input type="submit" value="Delete" name="delete" <?=
→ $provider->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](#).