APUNTES MAPEO OBJETO–RELACIONAL (MOR)

A continuación abordaremos el problema de la desadaptación entre el paradigma orientado a objetos, utilizado por lenguajes de programación como PHP etc, y el paradigma relacional utilizado por los sistemas de gestión de bases de datos (SGBD) como MySQL.

Este desajuste conceptual se conoce como:

Impedancia objeto-relacional (*object-relational impedance mismatch*).

¿De qué se trata este desajuste?

Hasta ahora, la forma que hemos tenido de programar el back-end, que se denomina procedural, describe un estilo de programación donde el código se organiza **en procedimientos, funciones y bloques secuenciales**, no en objetos, clases ni capas.

Un código típico al que estábamos acostumbrados hasta ahora tenía este aspecto:

```
$conexion = mysqli (...);

$sql = "SELECT * FROM productos WHERE precio < $precio";

$resultado = mysqli_query($conexion, $sql);

…operaciones con $resultado
```

Problemas graves del enfoque procedural:

Acoplamiento extremo

En el enfoque procedural, el SQL está escrito directamente dentro del mismo archivo que procesa la petición y genera la respuesta. Esto significa que la lógica de negocio, el acceso a datos y la presentación están mezclados en el mismo sitio. Si quieres cambiar la forma en que se consulta la base de datos, tienes que modificar cada archivo PHP que hace una operación similar. Cualquier cambio en la estructura de la tabla afecta directamente al código de los scripts que la usan.

Repetición de código

Cada archivo PHP procedural repite los mismos pasos una y otra vez: abrir conexión, preparar SQL, ejecutar la query, validar la entrada, procesar resultados… Como no hay clases ni métodos reutilizables, cada script es una repetición del anterior. En un proyecto pequeño parece soportable, pero cuando tienes 10, 20 o 50 operaciones distintas, terminas con el mismo bloque de conexión copiado decenas de veces. Un cambio puede obligar a editar todos los archivos manualmente.

Inseguridad

El procedural suele fomentar malas prácticas, como insertar directamente valores recibidos del usuario en la consulta SQL sin filtrarlos ni parametrizarlos. Esto abre la puerta a **inyecciones SQL**, uno de los ataques más comunes y peligrosos en aplicaciones web. Aprenderemos a utilizar consultas preparadas para solucionarlo.

Dificultad para escalar

Cuando la aplicación crece, modificar algo pequeño se convierte en un trabajo enorme. Si cambias un campo en la base de datos, por ejemplo renombrar "precio" a "precioUnidad", tienes que buscar y actualizar manualmente todas las consultas SQL repartidas por decenas de archivos. Como no existe una capa central de acceso a datos, los cambios son manuales y muy fáciles de olvidar en alguna parte, causando fallos silenciosos o errores críticos en producción.

Imposibilidad de integrar MVC, frameworks o pruebas unitarias

El enfoque procedural no separa correctamente las capas ni las responsabilidades: en un mismo script se reciben peticiones, se validan datos, se abre la conexión, se construyen consultas SQL, se ejecutan y después se generan las respuestas. Esta forma de trabajar puede funcionar en pequeños ejercicios, pero no encaja con las arquitecturas modernas del desarrollo web.

Frameworks y entornos actuales —como Laravel, Symfony, Node.js con Express, Spring Boot, Django, etc.— están diseñados para trabajar siempre con una estructura clara de capas: controladores, servicios, modelos, repositorios/DAOs, etc. Cada capa tiene su responsabilidad y se comunica con las demás mediante objetos, no mezclando lógica con el SQL.

Conclusión:

El estilo procedural solo es válido para scripts pequeños. Para aplicaciones mantenibles y seguras, es muy recomendable estructurar el backend.

Para reducir este choque conceptual, las aplicaciones modernas incorporan técnicas estructuradas para traducir correctamente **objetos ↔ registros relacionales**.

La clave de esta unidad no es "programar más", sino **ordenar profesionalmente el backend**, evitando el caos característico del PHP procedimental.

¿Qué es el Mapeo Objeto–Relacional?

El **mapeo objeto-relacional (MOR)** es el proceso mediante el cual se establece una relación coherente entre:

- Las **clases y objetos** del lenguaje de programación
- Las **tablas y registros** de la base de datos relacional

MOR implica:

1. **Representar una tabla como una clase (Modelo / DTO)**
2. **Representar cada fila como un objeto**
3. **Convertir automáticamente columnas ↔ atributos**
4. **Encapsular todas las operaciones SQL en capas de acceso a datos (DAO)**

Arquitectura de backend resultante (la iremos desmenuzando en la parte backend)

La UD5 introduce una arquitectura de **tres capas**, estándar en ingeniería del software.

1. Capa de Presentación (frontend)

- HTML/CSS/JS
- Uso de fetch()
- Comunicación via JSON

2. Capa de Controladores

- Archivos PHP situados en /public/
- Procesan la petición HTTP
- Validan entradas mínimas
- Llaman a los DAO
- Devuelven respuesta en JSON

Nunca contienen SQL.

3. Capa de Acceso a Datos

- DAO (Centraliza todo el acceso a datos)
- DTO
- Conexión PDO (Nos conecta a la BBDD)

Todo el acceso a base de datos queda unificado y centralizado aquí.

CONTROLADORES

Un controlador es un archivo PHP cuya única misión es:

1. Recibir una petición del frontend (por ejemplo, una llamada fetch()).
2. Validar los datos de entrada.
3. Llamar al DAO (clase que se encarga de toda la comunicación con la base de datos) para obtener o modificar datos.
4. Devolver una respuesta en JSON al frontend.

El controlador no contiene SQL, no abre conexiones y no procesa datos por sí mismo.
Su función es coordinar.

En esta unidad, todos los controladores viven en la carpeta:

/public/

porque son la “puerta de entrada” del backend: los endpoints a los que accede el navegador.

Esta capa separa el frontend del acceso a datos, permite validar datos antes de usar la base de datos, y en definitiva facilita el mantenimiento y la escalabilidad

El controlador actúa como un “guardia de seguridad” que decide qué hacer con cada petición.

Usando este código de ejemplo, de una aplicación en la que vamos a buscar productos:

```
<?php
header("Content-Type: application/json");
require_once "../dao/ProductoDAO.php";
$texto = $_POST["busqueda"];
$dao = new ProductoDAO();
$resultado = $dao->buscar($texto);
echo json_encode($resultado);
```

header("Content-Type: application/json");

El controlador declara explícitamente que toda respuesta será JSON, igual que en un API profesional.

require_once "../dao/ProductoDAO.php";

Carga la clase DAO que hará el trabajo.
El controlador no debe contener ninguna lógica de base de datos.

$_POST["busqueda"]

Recoge el dato enviado desde el fetch() del frontend.

$dao = new ProductoDAO();

Crea un objeto DAO, que será el que se encargue de todo el acceso a datos.

$dao->buscar($texto);

Invoca la lógica necesaria, que en este caso es un método que se llama buscar. El controlador NO sabe cómo funciona la búsqueda; solo la usa.

echo json_encode($resultado);

Devuelve los datos al frontend en JSON.

DATA ACCESS OBJECT (DAO)

La capa DAO (Data Access Object) tiene como misión aislar y encapsular completamente el acceso a base de datos, de manera que ninguna otra capa necesite saber SQL ni abrir conexiones.

Gracias a la capa DAO conseguimos:

- Centralizar todo el acceso a datos en un único lugar.
- Eliminar SQL de los controladores.

- Facilitar la reutilización de consultas.
- Reducir el acoplamiento entre el backend y la base de datos.
- Evitar código repetido (abrir conexiones, preparar consultas…).
- Permitir que los modelos (DTO) representen nuestras entidades de negocio.

Un DAO actúa como un *"traductor"* entre el mundo relacional (tablas, columnas, registros) y el mundo orientado a objetos (clases, atributos, objetos).

Llevado a la práctica con las tecnologías que estamos usando, un DAO es una clase PHP cuya única responsabilidad es realizar operaciones sobre una tabla o entidad concreta:

- Consultar (SELECT)
- Insertar (INSERT)
- Actualizar (UPDATE)
- Borrar (DELETE)

Cada tabla suele tener su DAO correspondiente, como por ejemplo:

- ProductoDAO → tabla *productos*
- UsuarioDAO → tabla *usuarios*
- PedidoDAO → tabla *pedidos*

Su sitio es la carpeta:

/dao/

Funcionamiento interno de un DAO

Un DAO siempre trabaja en colaboración con:

- **La clase Database, que profundizaremos más adelante, y centralizará la creación de la conexión a BBDD.**
- **La clase Modelo / DTO, que transforma cada fila obtenida de la BD en un objeto de la clase modelo.**

Las consultas que haremos a la BBDD estarán en un formato llamado Consultas Preparadas (Prepared Statements):

Son una forma segura de ejecutar sentencias SQL evitando que el usuario pueda "romper" la consulta o inyectar código malicioso. Son proporcionadas por la extensión PDO (PHP Data Objects) de PHP.

La inyección SQL es un tipo de ataque donde un usuario malintencionado envía texto especialmente diseñado para alterar la consulta SQL que el servidor esperaba ejecutar.

Si el programador concatena directamente valores recibidos del usuario dentro del SQL, el atacante puede *inyectar* partes adicionales de SQL con el fin de obtener datos privados,borrar tablas, modificar registros, acceder a cuentas sin contraseña, provocar fallos etc

Es uno de los ataques más frecuentes y peligrosos en aplicaciones web.

Supongamos este código:

```
$texto = $_POST["busqueda"];

$sql = "SELECT nombre FROM productos WHERE nombre LIKE '%$texto%'";

$result = $conn->query($sql);
```

Ataque 1. Imagina que un atacante escribe:

```
%' OR 1=1 --
```

La consulta generada sería:

```
SELECT nombre FROM productos WHERE nombre LIKE '%%' OR 1=1 -- %'
```

- **OR 1=1 → siempre es verdadero**
- **-- → convierte el resto en un comentario**

Resultado: se devuelven todos los productos de la base de datos.

Ataque 2. Imagina que el atacante escribe:

```
'; DROP TABLE usuarios; --
```

Esto haría que después del select se ejecute:

```
DROP TABLE usuarios;
```

Resultado: ¡SE BORRA LA TABLA!

La forma de evitar esto es usar consultas preparadas, que separan estrictamente:

- El código SQL
- Los datos que vienen del usuario

De esta forma el usuario *NO PUEDE ALTERAR* la estructura del SQL, porque sus valores jamás se concatenan dentro de la consulta.

La sintaxis básica es:

```
$stmt = $conn->prepare("SELECT nombre FROM productos WHERE nombre LIKE :txt");

$stmt->execute([":txt" => "%$texto%"]);
```

Y la explicación paso a paso:

Preparación del SQL (prepare)

```
$stmt = $conn->prepare("SELECT nombre FROM productos WHERE nombre LIKE :txt");
```

- El SQL se envía al servidor pero *sin* los datos del usuario.
- El servidor precompila la consulta y detecta la estructura.
- Se reserva un “hueco” llamado parámetro (:txt).

Ese hueco espera un valor, pero ya no se puede cambiar la consulta.

Envío de los valores (execute)

$stmt->execute([":txt" => "%$texto%"]);

- Los datos del usuario viajan *separados* del SQL.
- El servidor los trata como valores, NO como código.
- Aunque el usuario envíe "%' OR 1=1 --", PDO lo envía como texto literal.

La clave es que PDO se encarga de:

- Escapar caracteres peligrosos
- Quitar significado SQL
- Tratar los parámetros como cadenas, números, etc., pero nunca como código

Por tanto, en ningún momento los datos se pegan dentro de la consulta como texto plano.

Por eso, aunque el usuario escriba: *%' OR 1=1 --*

PDO lo convierte en un simple valor de texto: *"%' OR 1=1 --"*

No tiene efecto como SQL.

Llegados a este punto, ya podemos entender un código completo de ejemplo del DAO, en este caso se controlarán aquí las operaciones que hagamos con la tabla productos:

```
<?php
require_once "../db/Database.php";
require_once "../model/Producto.php";

class ProductoDAO {

  public function buscar($texto) {
    $conn = Database::getConnection();

    $stmt = $conn->prepare(
      "SELECT nombre, precio
       FROM productos
       WHERE nombre LIKE :txt"
```

```
        );
        $stmt->execute([":txt" => "%$texto%"]);

        $lista = [];

        while ($fila = $stmt->fetch(PDO::FETCH_ASSOC)) {
            $lista[] = new Producto($fila["nombre"], $fila["precio"]);
        }

        return $lista;
    }
}
```

require_once "../db/Database.php";

Incluye la clase que gestiona la conexión. El DAO **no crea conexiones**, solo las pide.

require_once "../model/Producto.php";

Incluye el modelo/DTO que usaremos para crear objetos a partir de los resultados SQL.

class ProductoDAO { ... }

La clase que manejará **todas las operaciones relativas a productos**.

public function buscar($texto)

Método que recibe un dato desde el controlador (pero NO desde HTTP), en este caso el texto que se quiere buscar, y devolverá la lista de productos que cumplan ese filtro. El DAO desconoce completamente el frontend.

$conn = Database::getConnection();

Solicita la conexión PDO centralizada.

$stmt = $conn->prepare("SELECT nombre, precio FROM productos

WHERE nombre LIKE :txt");

Produce SQL limpio y parametrizado, sin valores directos del usuario

$stmt->execute([":txt" => "%$texto%"]);

Se envía el parámetro de forma segura.

$lista[] = new Producto($fila["nombre"], $fila["precio"]);

Cada fila de la tabla se convierte en un objeto **Producto**, siguiendo el mapeo objeto-relacional.

return $lista;

La función devuelve un array de objetos (en este ejemplo en concreto un array de productos).

Ventajas del DAO:

- Si mañana la tabla *productos* cambia de nombre o estructura, solo modificamos ProductoDAO, no todo el backend.
- El método buscar() se puede usar desde varios controladores
- Siempre usamos sentencias preparadas → reducción del 99% de riesgo de SQL injection.
- Podemos añadir métodos como insertar(Producto $p), actualizar(Producto $p), eliminar($id), etc, sin modificar nada de capas superiores.