Tecnologías Informáticas B

TUTORÍA Backend Refactoring 01

Vamos a explicar archivo por archivo los scripts php del backend refactorizado, puede usar el índice como enlaces a cada archivo:

Archivo: server.php

Este archivo es el **punto de entrada principal** del backend. Se encarga de:

- Habilitar el modo debug para desarrollo.
- Configurar los headers para permitir peticiones desde otro dominio (CORS).
- Atender preflight requests del navegador (método **OPTIONS**).
- Derivar el procesamiento de las peticiones a un archivo de rutas.

<?php

Inicia el script PHP. Obligatorio en todos los archivos .php.

```
/**
 * DEBUG MODE
 */
ini_set('display_errors', 1);
error_reporting(E_ALL);
```

- **ini_set('display_errors', 1);**: Le indica a PHP que muestre los errores directamente en pantalla. Muy útil para depurar durante el desarrollo.
- **error_reporting(E_ALL);**: Muestra todos los tipos de errores (notices, warnings, fatales, etc).

¡Importante! Esto debe desactivarse en producción (sitios reales) para evitar mostrar detalles internos al usuario final.

```
header("Access-Control-Allow-Origin: *");
```

- Permite que cualquier frontend (desde cualquier origen *) pueda acceder al backend.
- Es parte de **CORS** (*Cross-Origin Resource Sharing*), necesario cuando el frontend y backend no están en el mismo dominio o puerto.

```
header("Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS");
```

- Le indica al navegador qué métodos HTTP acepta el backend.

```
header("Access-Control-Allow-Headers: Content-Type");
```

Permite que el navegador envíe encabezados personalizados, como **Content-Type: application/json.**

Estas tres líneas son claves para que **fetch()** desde JavaScript funcione correctamente con métodos como **POST**, **PUT**, etc. Esto es obligatorio para evitar errores CORS.

```
require_once("./routes/studentsRoutes.php");
```

- Incluye el archivo que define las **rutas** o la lógica que responderá a la petición actual.
- Usa require_once para incluirlo **solo una vez** (evita

errores por múltiples inclusiones).

- Este archivo debería analizar la URL, método y decidir qué controlador invocar.

✅ Buenas prácticas aplicadas

- Separación de responsabilidades: este archivo solo configura y delega.
- Headers CORS correctamente definidos.
- Manejo de OPTIONS para preflight.
- Debug activado durante desarrollo.

Posibles mejoras para el futuro

- En producción: desactivar los errores (display_errors = 0) y registrarlos en un log.
- Validar que studentsRoutes.php maneje correctamente rutas y errores.
- Modularizar el enrutamiento si se suman más módulos (ya lo estamos haciendo).
- Agregar un manejador para respuestas 404 si la ruta no es reconocida.

Archivo: studentsRoutes.php

Este archivo se encarga de:

- Conectar a la base de datos.
- Incluir el controlador que maneja la lógica del CRUD.
- Delegar las peticiones al controlador según el método HTTP usado (GET, POST, PUT, DELETE).

```
require_once("./config/databaseConfig.php");
```

- Incluye el archivo de configuración de la base de datos.
- Crea y expone la variable $conn que se usará para interactuar con MySQLi.
- Se usa require_once para evitar duplicación si este archivo ya fue incluido antes.

```
require_once("./controllers/studentsController.php");
```

- Incluye el archivo que contiene las funciones handleGet(), handlePost(), etc.
- Estas funciones implementan la lógica de negocio (CRUD sobre estudiantes).

```
switch ($_SERVER['REQUEST_METHOD']) {
```

- Lee el método HTTP con el que se hizo la solicitud (por ejemplo GET, POST, PUT, etc).
- PHP lo obtiene desde la variable global $_SERVER['REQUEST_METHOD'].

Cada caso del switch

```
    case 'GET':
        handleGet($conn);
        break;
```

Si es una solicitud **GET**, llama a handleGet() pasando la conexión **$conn**. Del mismo modo para **PUT**, **UPDATE** y **DELETE.**

```
    default:
        http_response_code(405);
        echo json_encode(["error" => "Método no permitido"]);
        break;
```

Si el método no es ninguno de los anteriores, responde con un 405 (Method Not Allowed) y un mensaje en JSON. Ver códigos de respuesta HTTP

✅ Buenas prácticas aplicadas

- Separación clara: este archivo no contiene lógica de negocio, solo enruta la solicitud.
- Código simple y mantenible: solo gestiona un módulo (students) y puede escalar si se aplica el mismo patrón.
- Usa json_encode() para enviar mensajes en formato JSON, adecuado para consumir desde el frontend.

Posibles mejoras para el futuro

Para escalar a múltiples módulos en el futuro (por ejemplo: profesores, materias, usuarios, etc), podrías:

- Agregar una lógica para detectar el módulo por parámetro GET o por URL amigable:

```
$module = $_GET['module'] ?? 'students';

if ($module === 'students') {
    require_once("./controllers/studentsController.php");
    // switch acá
} elseif ($module === 'teachers') {
    require_once("./controllers/teachersController.php");
    // otro switch
}
```

- **O usar un sistema de rutas tipo "router" más dinámico**, por ejemplo:

```
$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

if ($path === '/api/students') {
    require_once("./controllers/studentsController.php");
    dispatchStudents($method, $conn);
}
```

Archivo: config/databaseConfig.php

Este archivo configura y establece una conexión con la base de datos MySQL usando la extensión MySQLi (MySQL Improved). Es fundamental en cualquier aplicación que use datos persistentes. Este archivo puede cambiar en producción (por ejemplo, al usar un servidor externo o variables de entorno).

```
$host = "localhost";
$user = "students_user";
$password = "12345";
$database = "students_db";
```

Define las **credenciales de conexión** a la base de datos:

- $host: servidor de base de datos. "localhost" indica que la base de datos está en el mismo servidor que el script PHP.
- $user: nombre de usuario de MySQL autorizado para acceder a la base de datos.
- $password: contraseña del usuario MySQL.

- $database: nombre de la base de datos que se va a usar.

```
$conn = new mysqli($host, $user, $password, $database);
```

- Crea un nuevo objeto mysqli y establece la conexión usando las credenciales anteriores.
- Este objeto ($conn) se usará luego en todos los models y controllers para ejecutar consultas.

```
if ($conn->connect_error) {
    http_response_code(500);
    die(json_encode(["error" => "Database connection failed"]));

}
```

Verifica si hubo error en la conexión ($conn->connect_error).

- Si falla:
    - Responde con un código HTTP **500 Internal Server Error**.
    - Envía un mensaje JSON con una clave error.
    - Usa **die()** para **detener la ejecución del script inmediatamente**.

Este chequeo evita que el backend siga funcionando sin acceso a la base de datos, lo cual sería peligroso y podría causar errores encadenados.

✅ Buenas prácticas aplicadas

- Código simple, directo, fácil de entender.
- Manejo de errores básico, pero efectivo.

- No expone detalles sensibles de la base de datos en el mensaje de error.
- Uso del objeto **$conn** compartido para centralizar la conexión.

Posibles mejoras para el futuro

1. **Variables de entorno** (con **.env** y **getenv()** o **$_ENV**) para ocultar credenciales:

```
$host = getenv("DB_HOST");
$user = getenv("DB_USER");
```

2. **Encapsular en una función o clase** si vas a reutilizar conexiones múltiples veces (por ejemplo, patrón Singleton). (***ALGO QUE NO VAMOS A HACER EN ESTA MATERIA)***
3. **Soporte para UTF-8** si manejás caracteres especiales:

```
$conn->set_charset("utf8mb4");
```

Archivo: controllers/studentsController.php

Este archivo se encarga de **recibir, procesar y responder** las solicitudes HTTP. Es parte del patrón **MVC** (Modelo - Vista - Controlador), donde:

- Los ***Modelos*** acceden a los datos (models/students.php).
- Los ***Controladores*** procesan las peticiones y devuelven respuestas.
- Las ***Vistas*** en este caso están en el frontend JavaScript.

Este archivo contiene funciones específicas para manejar las

solicitudes HTTP según el tipo de operación que se quiera hacer sobre los estudiantes: obtener, crear, actualizar o eliminar.

```
require_once("./models/students.php");
```

Este **require_once** importa las funciones del modelo students.php, donde están definidas:

- getAllStudents()
- getStudentById()
- createStudent()
- updateStudent()
- deleteStudent()

Principio de separación de responsabilidades: el controlador no accede directamente a la base de datos, sino que delega esa tarea al modelo.

Funciones del controlador

function handleGet($conn)

```
if (isset($_GET['id'])) {
    $result = getStudentById($conn, $_GET['id']);
    echo json_encode($result->fetch_assoc());
} else {
    $result = getAllStudents($conn);
    $data = [];
    while ($row = $result->fetch_assoc()) {
        $data[] = $row;
    }
```

```
    echo json_encode($data);
}
```

- Esta función se ejecuta cuando el método HTTP es GET.
- Si se pasa un id por la URL (ejemplo: ?id=3), se devuelve solo ese estudiante.
- Si no se pasa id, se devuelve la lista completa.
- fetch_assoc() convierte cada fila en un array asociativo (clave => valor).
- json_encode() transforma los datos a formato JSON para que el frontend pueda interpretarlos.

function handlePost($conn)

```
$input = json_decode(file_get_contents("php://input"), true);
if (createStudent($conn, $input['fullname'], $input['email'],
$input['age'])) {
    echo json_encode(["message" => "Estudiante agregado
correctamente"]);
} else {
    http_response_code(500);
    echo json_encode(["error" => "No se pudo agregar"]);
}
```

- Maneja la creación de un nuevo estudiante con datos enviados en formato JSON.
- **file_get_contents("php://input")**: obtiene el **cuerpo crudo** de la solicitud.
- **json_decode(..., true)**: convierte el JSON en un array PHP.
- Si la operación tiene éxito, responde con un mensaje; si falla, con error 500.

function handlePut($conn)

```
$input = json_decode(file_get_contents("php://input"), true);
if (updateStudent($conn, $input['id'], $input['fullname'],
$input['email'], $input['age'])) {
    echo json_encode(["message" => "Actualizado correctamente"]);
} else {
    http_response_code(500);
    echo json_encode(["error" => "No se pudo actualizar"]);
}
```

- Se usa cuando el cliente quiere **editar** un estudiante.
- Requiere el **id** del estudiante a modificar y los nuevos valores.
- Internamente, llama a **updateStudent()** del modelo.

function handleDelete($conn)

```
$input = json_decode(file_get_contents("php://input"), true);
if (deleteStudent($conn, $input['id'])) {
    echo json_encode(["message" => "Eliminado correctamente"]);
} else {
    http_response_code(500);
    echo json_encode(["error" => "No se pudo eliminar"]);
}
```

- Elimina el estudiante cuyo `id` se recibe por JSON.
- El modelo se encarga de ejecutar el **DELETE** en la base de datos.

✅ Buenas prácticas aplicadas

- Todo está encapsulado en funciones reutilizables.

- Separación clara entre la lógica de negocio (controladores) y acceso a datos (modelos).
- Evita que `server.php` se vuelva un archivo grande y desordenado.
- Facilita testeo y escalabilidad.

Posibles mejoras para el futuro

- Validación de los datos de entrada (`fullname`, `email`, `age`, etc.).
- Sanitización para prevenir errores y vulnerabilidades.
- Un sistema de logging de errores.
- Reutilizar lógica de `json_decode(file_get_contents(...))` en una función común.

Archivo: models/students.php

Este archivo contiene todas las funciones relacionadas con el **acceso a la base de datos** para el módulo de estudiantes. Es el modelo de datos del patrón MVC (Modelo-Vista-Controlador), y **su único rol** es interactuar con la base de datos usando SQL.

Ninguna función aquí debería preocuparse por cómo se muestra la información, ni por la lógica de negocio o control de flujo. Solo datos.

Función: getAllStudents($conn)

```php
$sql = "SELECT * FROM students";
return $conn->query($sql);
```

- Ejecuta una consulta SQL que selecciona **todos los**

registros de la tabla **students**.

- Usa **->query()** directamente porque **no hay entrada del usuario** (no hay riesgo de inyección SQL).
- Devuelve un objeto **mysqli_result** que se recorre en el controlador con **fetch_assoc()**.

Función: getStudentById($conn, $id)

```
$sql = "SELECT * FROM students WHERE id = ?";
$stmt = $conn->prepare($sql);
$stmt->bind_param("i", $id);
$stmt->execute();
return $stmt->get_result();
```

Obtiene un único estudiante usando su id.

¿Por qué usar prepare() y bind_param()?

Porque estamos usando un valor proporcionado por el usuario (**$id**).

- Previene ataques de **inyección SQL**.
- **"i"** indica que el parámetro es de tipo **integer**.

Función: createStudent($conn, $fullname, $email, $age)

```
$sql = "INSERT INTO students (fullname, email, age) VALUES (?, ?, ?)";
$stmt = $conn->prepare($sql);
$stmt->bind_param("ssi", $fullname, $email, $age);
return $stmt->execute();
```

Inserta un nuevo estudiante en la base de datos (**Validación y sanitización** de los datos deberían hacerse **antes** de llegar al modelo en el controlador o middleware).

- "**ssi**" indica los tipos de los parámetros: string, string, integer.
- **->execute()** devuelve **true** o **false** si la consulta tuvo éxito.

Función: updateStudent($conn, $id, $fullname, $email, $age)

```
$sql = "UPDATE students SET fullname = ?, email = ?, age = ? WHERE id = ?";
$stmt = $conn->prepare($sql);
$stmt->bind_param("ssii", $fullname, $email, $age, $id);
return $stmt->execute();
```

Modifica los datos de un estudiante existente.

- "**ssii**" → **fullname** y **email** son strings, **age** e **id** son enteros.
- Siempre usar **prepare()** y **bind_param()** para cualquier entrada externa.

Función: deleteStudent($conn, $id)

```
$sql = "DELETE FROM students WHERE id = ?";
$stmt = $conn->prepare($sql);
$stmt->bind_param("i", $id);
return $stmt->execute();
```

Elimina a un estudiante por su **id**.

- El uso de **prepare()** + **bind_param()** es obligatorio para evitar inyección SQL.

✅ **Buenas prácticas aplicadas**

- Separación clara del modelo respecto al controlador y la vista.
- Uso exclusivo de prepare() para todas las consultas con entrada de usuario.
- Uso explícito de tipos en bind_param() (buena documentación implícita).
- Devolución clara de resultados: query() o execute() permiten al controlador actuar según el resultado.

Apéndice sobre inyección SQL:

Una inyección SQL (SQL Injection) ocurre cuando un atacante "inyecta" código SQL malicioso dentro de una consulta que no está correctamente protegida, con el objetivo de:

- Leer datos sensibles.
- Modificar o eliminar datos.
- Obtener control del servidor.

Ocurre **cuando el código SQL se construye concatenando directamente datos del usuario**.

Ejemplo vulnerable (¡NO hacer!)

```
// Supongamos que $id viene de un formulario o de $_GET
$id = $_GET['id'];
$sql = "SELECT * FROM students WHERE id = $id";
$result = $conn->query($sql);
```

Si el atacante accede a:

```
http://localhost/proyecto/backend/server.php?id=1 OR 1=1
```

El valor de **$id** será:

```
1 OR 1=1
```

Y la consulta quedará:

```
SELECT * FROM students WHERE id = 1 OR 1=1
```

Resultado: el atacante ve a **todos los estudiantes** porque **1=1** siempre es verdadero.

O peor aún:

```
http://localhost/proyecto/backend/server.php?id=1; DROP TABLE
students;
```

Esto puede llegar a borrar toda la tabla si no hay protección adecuada (dependiendo del motor y configuración).

¿Cómo lo evita prepare()?

Cuando usamos **prepare()** y **bind_param()**, el SQL y los datos se envían separados al servidor.

```
$stmt = $conn->prepare("SELECT * FROM students WHERE id = ?");
$stmt->bind_param("i", $id);  // "i" = integer
$stmt->execute();
```

Lo importante:

- El **?** es un **marcador de posición**, no se sustituye por texto.
- **bind_param()** asegura que el valor sea del tipo esperado (entero en este caso).

- Si alguien pone **1 OR 1=1**, ese texto se interpretará como **una cadena de texto**, no como código SQL.

Resultado: el motor de MySQL sabe que el valor no forma parte del SQL. No puede ejecutarse como código.

Metáfora para explicar prepare()

Imaginá que tenés una fotocopiadora con un espacio en blanco y un cartel que dice:
"Escribí tu nombre aquí y te damos una copia personalizada."

- Si escribís tu nombre, todo bien.
- Pero si intentás dibujar una bomba, **la máquina no entiende dibujos**, solo texto de nombre.

prepare() hace lo mismo: le dice a MySQL *"esto es un espacio seguro para un dato, no lo interpretes como código."*

✅ Mejores prácticas relacionadas

- Siempre usar `prepare()` + `bind_param()` o **PDO con consultas preparadas**.
- Nunca concatenar variables directamente en una consulta SQL.
- Validar (verificar formato) y sanitizar (eliminar caracteres peligrosos) también ayuda, pero no reemplaza las consultas preparadas.
- No mostrar errores SQL al usuario final (pueden revelar estructuras de la base de datos).