

Tecnologías Informáticas B

TUTORÍA Full Stack Refactoring 03

Vamos a explicar archivo por archivo los archivos html, javascript y php que se utilizan en el proyecto:
<https://github.com/gabrielinuz/crud-php-prototipo-refactorizado-3.0.git>

Archivo: frontend/index.html.....	4
Archivo: frontend/students.html.....	8
Formulario de estudiante.....	9
Tabla de estudiantes.....	11
Pie de página con botón de volver.....	11
Consideraciones generales sobre Javascript:.....	12
async y await:.....	12
Archivo: frontend/js/controllers/studentsController.js.....	12
Importación de módulo.....	12
Esperar a que cargue el HTML completo.....	13
Función setupFormHandler(): configurar envío del formulario.....	13
Función setupCancelHandler(): configurar la cancelación del formulario.....	15
Función getFormData(): obtener datos del formulario.....	16
Función loadStudents(): cargar estudiantes desde el backend.....	16
Función renderStudentTable(): mostrar estudiantes en una tabla.....	17
Función createCell(text): crear una celda <td>.....	18
Función createActionsCell(student): botones Editar y Borrar.....	18
Función fillForm(student): cargar datos en el formulario.....	19
Función confirmDelete(id): confirmar y borrar.....	20
Archivo: frontend/js/api/studentsAPI.js.....	20
Archivo: frontend/js/api/apiFactory.js.....	21
Definición de función exportada:.....	21
Construcción de la URL de acceso al servidor:.....	22
Función interna sendJSON():.....	22
Retorno de createAPI(): objeto con funciones del módulo.....	23
¿Qué logramos con estos archivos en el frontend?.....	25
Archivo: backend/server.php.....	25
Líneas para mostrar errores:.....	26
Configuración de cabeceras HTTP.....	27
Función auxiliar para responder con código HTTP.....	27
Manejo de solicitudes HTTP tipo OPTIONS (preflight).....	28
El navegador a veces envía primero una solicitud de prueba llamada OPTIONS para	

verificar si puede hablar con el servidor. Si la solicitud es OPTIONS, respondemos con un código 200 y no hacemos más nada.

Obtención del módulo desde la URL.....	28
Validación del nombre del módulo.....	29
Carga del archivo de rutas del módulo.....	29
Inclusión condicional del archivo.....	30
¿Qué hace server.php?.....	30
Archivo: backend/routes/studentsRoutes.php.....	31
Comentario de prueba: invocación básica.....	31
Extensión personalizada de la ruta POST.....	32
Paso a paso:.....	32
Archivo: backend/config/databaseConfig.php.....	34
Variables de conexión.....	34
Conexión con MySQL.....	35
Verificación de error en la conexión.....	35
Resultado esperado.....	36
Resumen para estudiantes.....	36
Archivo: backend/routes/routesFactory.php.....	36
Función principal routeRequest(...).....	37
Detectar el método HTTP de la petición.....	37
Definir funciones por defecto.....	38
¿Qué hace exactamente?.....	38
Combinar handlers personalizados (si los hay).....	39
Validar si el método es soportado.....	39
Ejecutar el handler correspondiente.....	39
Verificar que sea una función válida.....	39
Resumen para estudiantes.....	40
Archivo: backend/controllers/studentsController.php.....	41
Incluir el modelo de estudiantes.....	41
Función handleGet(\$conn).....	41
Función handlePost(\$conn).....	42
Función handlePut(\$conn).....	43
Función handleDelete(\$conn).....	44
Resumen para estudiantes.....	45
Archivo: backend/models/students.php.....	45
Función: getAllStudents(\$conn).....	46
Función: getStudentById(\$conn, \$id).....	46
Función: createStudent(\$conn, \$fullname, \$email, \$age).....	47
Función: updateStudent(\$conn, \$id, \$fullname, \$email, \$age).....	48
Función: deleteStudent(\$conn, \$id).....	49
Resumen general para estudiantes.....	49
Archivo: frontend/html/studentsSubjects.html.....	50
Función general del archivo.....	50
¿Qué tipo de relación representa?.....	51

Secciones clave del archivo.....	51
1. <head>.....	51
2. Formulario de asignación (#relationForm).....	52
3. Tabla de relaciones.....	53
4. Footer.....	53
Estructura semántica completa.....	53
Archivo: frontend/js/controllers/studentsSubjectsController.js.....	55
Imports de APIs.....	55
Inicialización principal.....	55
Funciones.....	56
initSelects().....	56
setUpFormHandler().....	56
setUpCancelHandler().....	56
getFormData().....	56
clearForm().....	57
loadRelations().....	57
renderRelationsTable(relations).....	57
createActionsCell(relation).....	57
fillForm(relation).....	58
confirmDelete(id).....	58
Comentarios pedagógicos.....	58
Archivo: frontend/js/api/studentsSubjectsAPI.js.....	58
Archivo: backend/controllers/studentsSubjectsController.php.....	59
Archivo: backend/models/studentsSubjects.php.....	59
Función assignSubjectToStudent:.....	59
Función getAllSubjectsStudents(\$conn).....	60
Función getSubjectsByStudent(\$conn, \$student_id) (NO USADA).....	61
Función updateStudentSubject(\$conn, \$id, \$student_id, \$subject_id, \$approved)...	61
Función removeStudentSubject(\$conn, \$id).....	62
Resumen para estudiantes.....	62
Notas pedagógicas.....	63
Archivo: backend/config/script_inicial.sql.....	63

Archivo: frontend/index.html

Si bien html y css son lenguajes que han trabajado en Tecnologías A se explicarán algunas secciones que pueden resultar distintas o incluso a modo de repaso. Este archivo es el **punto de entrada del frontend**:

- Simplemente con tres enlaces/botones redirige a cada html que representa el frontend de cada módulo del ejemplo.

```
<!-- frontend/index.html -->
```

Comentario en HTML: Indica la ubicación del archivo dentro del proyecto. No afecta al funcionamiento del sitio web, sirve como guía para el programador.

```
<!DOCTYPE html>
```

Declaración del tipo de documento: Esta línea le indica al navegador que se trata de un documento HTML5. Es obligatoria y siempre debe estar al inicio del archivo HTML.

```
<html lang="es">
```

Etiqueta <html>: Marca el inicio del documento HTML.

Atributo lang="es": Informa que el contenido está en español, lo que ayuda a los navegadores, motores de búsqueda y lectores de pantalla a interpretar correctamente el contenido.

<head>

Etiqueta <head>: Aquí se colocan los metadatos del documento. Esta sección no se muestra directamente en la página, pero es esencial para su funcionamiento.

<meta charset="UTF-8" />

Codificación de caracteres: Establece la codificación del texto como UTF-8, que permite usar acentos, eñes y otros caracteres especiales propios del español y muchos otros idiomas.

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

Diseño adaptable (responsive): Hace que el sitio se vea bien en celulares, tablets y computadoras.

- **width=device-width** adapta el ancho al tamaño de la pantalla del dispositivo.
- **initial-scale=1.0** define el nivel de zoom inicial.

<title>Gestión de Educativa</title>

Título de la pestaña: Este texto aparece en la pestaña del navegador y ayuda al usuario a identificar la página abierta.

```
<link rel="icon" type="image/png" sizes="32x32"
href="img/favicon.png">
```

Ícono del sitio (favicon): Es la pequeña imagen que aparece en la pestaña del navegador, normalmente un logotipo o símbolo representativo del sitio.

```
<link rel="stylesheet" href="css/w3.css" />
<link rel="stylesheet" href="css/styles.css" />
```

Hojas de estilo (CSS):

- **w3.css** es una hoja de estilos externa proporcionada por la biblioteca W3.CSS, que ofrece clases predefinidas para diseño y colores.
- **styles.css** es una hoja de estilos propia del proyecto, donde se pueden definir reglas personalizadas.

```
</head>
```

Cierre de la sección `<head>`

```
<body class="w3-container">
```

Cuerpo del documento (`<body>`): Todo lo que aparece dentro del `<body>` se muestra al usuario en la pantalla.

Clase `w3-container`: Aplicada desde W3.CSS para dar margen interno (padding) y alineación apropiada al contenido.

```
<header>
  <h2 class="w3-center w3-margin-top">Panel Principal</h2>
</header>
```

Encabezado (<header>): Parte superior de la página, donde se suele colocar el título o logo.

- **<h2>** es un título de segundo nivel (hay 6 niveles de título, de **<h1>** a **<h6>**).
- **w3-center** centra el texto horizontalmente.
- **w3-margin-top** agrega un margen arriba del título para separación visual.

```
<main class="w3-bar w3-center">
```

Contenido principal (<main>): Aquí se ubica el contenido más importante de la página.

- **w3-bar**: crea una barra de botones horizontal.
- **w3-center**: centra los botones horizontalmente.

```
<a class="w3-btn w3-black" href="html/students.html">Gestión de Estudiantes</a>
```

Botón 1:

- Etiqueta **<a>** es un enlace.
- Clase **w3-btn** lo convierte en un botón visual.
- Clase **w3-black** lo pinta de negro.
- **href="html/students.html"** define la página a la que lleva ese botón.

- El texto "Gestión de Estudiantes" es lo que ve el usuario.

```
<a class="w3-btn w3-black" href="html/subjects.html">Gestión de  
Materias</a>
```

Botón 2: Igual que el anterior, pero lleva a la gestión de materias.

```
<a class="w3-btn w3-black"  
href="html/studentsSubjects.html">Asignación de Materias a  
Estudiantes</a>
```

Botón 3: Igual que los anteriores, pero lleva a la pantalla para asignar materias a los estudiantes.

```
</main>
```

Fin del contenido principal.

```
</body>  
</html>
```

Cierre del cuerpo (</body>) y del documento HTML (</html>).

Archivo: frontend/students.html

Hasta la etiqueta script se aplica la misma sintaxis que el archivo anterior. En todos los casos donde las etiquetas sean triviales o ya se haya mencionado su funcionalidad, no se

repetirá su explicación.

```
<script type="module"  
src="../../js/controllers/studentsController.js"></script>
```

Importa el archivo JavaScript que controla este módulo. Se usa `type="module"` porque estamos usando importaciones (`import { ... } from ...`) en el JS.

Formulario de estudiante

```
<form id="studentForm" class="w3-card w3-padding w3-margin-bottom  
w3-light-grey">
```

Se crea un formulario con ID `studentForm`. Tiene estilos que le dan bordes (`w3-card`), espacio interno (`w3-padding`), margen inferior y fondo gris claro. Es importante notar que no hay atributos `action=""`, ni `method=""` en este formulario, no son necesarios el envío del formulario y esto se manejará desde el controlador `studentsController.js`.

```
<input type="hidden" id="studentId" />
```

Este campo oculto se usa para saber si estamos editando un estudiante ya existente (tiene ID), o creando uno nuevo (ID vacío).

```
<div class="w3-row-padding">
```

Contenedor para agrupar los tres campos en una fila con

espaciado.

```
<div class="w3-third">  
  <label>Nombre completo</label>  
  <input class="w3-input" type="text" id="fullname" required />  
</div>
```

w3-third: divide el ancho en 3 columnas iguales. El campo **fullname** es obligatorio (**required**).

```
<div class="w3-third">  
  <label>Email</label>  
  <input class="w3-input" type="email" id="email" required />  
</div>
```

Campo tipo **email**: el navegador valida que el valor tenga formato de correo. También es obligatorio.

```
<div class="w3-third">  
  <label>Edad</label>  
  <input class="w3-input" type="number" id="age" required />  
</div>
```

Campo numérico para la **edad**. También obligatorio.

```
<div class="w3-margin-top">  
  <button class="w3-button w3-green" type="submit">Guardar</button>
```

Botón para **enviar el formulario**. Dispara el evento que captura **studentsController.js**. Y ese controlador después se encargará del envío de los datos a la API.

```
<button id="cancelBtn" class="w3-button w3-grey"  
  type="reset">Cancelar</button>
```

Botón para **limpiar el formulario**.

Tabla de estudiantes

```
<table class="w3-table-all w3-hoverable w3-card">
```

Tabla con todos los estudiantes cargados. Tiene estilos que hacen que:

- Cada fila se vea bien (**w3-table-all**)
- Se resalte al pasar el mouse (**w3-hoverable**)
- Tenga borde (**w3-card**)

```
<thead>
  <tr class="w3-light-blue">
    <th>Nombre</th>
    <th>Email</th>
    <th>Edad</th>
    <th>Acciones</th>
  </tr>
</thead>
```

Cabecera de la tabla, con fondo azul claro. Define las columnas.

```
<tbody id="studentTableBody"></tbody>
```

Cuerpo de la tabla. Empieza vacío. Este contenido lo genera **studentsController.js**, agregando una fila (**<tr>**) por cada estudiante.

Pie de página con botón de volver

```
<footer class="w3-center w3-margin-top">
  <a class="w3-btn w3-black" href="../index.html">Volver al
  Panel Principal</a>
</footer>
```

Botón negro centrado que permite volver al **index.html**, el

panel principal del sistema.

Consideraciones generales sobre Javascript:

async y await:

- **async** marca una función como asincrónica. Esto significa que puede usar **await** adentro. Hoy ya por defecto se puede usar **await** sin **async**.
- **await** pausa la ejecución hasta que la llamada de esa línea se resuelve. Hace que el código asincrónico se lea de forma sincrónica (ordenada).

Archivo:

frontend/js/controllers/studentsController.js

Este archivo se encarga de:

- De manejar la lógica de vista del módulo de estudiantes.
- Importa y utiliza `../api/studentsAPI.js` para acceder al backend y poder crear, obtener, actualizar y borrar datos del estudiante delegando esa funcionalidad a la API.
- Crea y actualiza el estado de todo elemento HTML dinámicamente.

```
import { studentsAPI } from '../api/studentsAPI.js';
```

Importación de módulo

- Esta línea importa las funciones del archivo `studentsAPI.js`, que contiene las funciones para

comunicarse con el backend (crear, actualizar, eliminar y obtener estudiantes).

- Esto permite usar `studentsAPI.create()`, `studentsAPI.update()`, etc., dentro de este archivo.

```
document.addEventListener('DOMContentLoaded', () =>
{
    loadStudents();
    setupFormHandler();
    setupCancelHandler();
});
```

Esperar a que cargue el HTML completo

Cuando el navegador termina de cargar toda la estructura de la página (`DOMContentLoaded`), se ejecuta esta función:

- `loadStudents()` carga y muestra todos los estudiantes actuales en la tabla.
- `setupFormHandler()` prepara el formulario para que cuando el usuario haga clic en "Guardar", se capture y procese la información.
- `setupCancelHandler()` configura el comportamiento del botón "Cancelar".

Función `setupFormHandler()`: configurar envío del formulario

```
function setupFormHandler()
{
    const form = document.getElementById('studentForm');
```

Se busca el formulario con el `id="studentForm"` desde el HTML.

```
form.addEventListener('submit', async e =>
{
    e.preventDefault();
```

Se configura el formulario para que, al enviarse, no recargue la página (`e.preventDefault()` cancela el comportamiento por defecto del navegador).

```
const student = getFormData();
```

Se extraen los datos del formulario usando una función llamada `getFormData()`.

```
try
{
    if (student.id)
    {
        await studentsAPI.update(student);
    }
    else
    {
        await studentsAPI.create(student);
    }
}
```

Dentro de un bloque Try (intenta el bloque y sino atrapa una excepción, osea un error inesperado para el cual no fue preparado el programa) hace lo siguiente: Si hay un ID (el campo oculto está lleno), significa que se está editando un estudiante → se llama `update()`. Si no hay ID, es un nuevo estudiante → se llama `create()`.

```
        clearForm();
        loadStudents();
    }
    catch (err)
    {
```

```

        console.error(err.message);
    }
    });
}

```

Después de guardar, se limpian los campos del formulario (`clearForm()`) y se recarga la tabla con `loadStudents()` para ver los cambios. Si algo dentro del bloque `try` hubiese “arrojado” una excepción, se atrapa en el bloque `catch` y se envía por salida de error en la consola de javascript el mensaje de error.

Función `setupCancelHandler()`: configurar la cancelación del formulario.

```

function setupCancelHandler()
{
    const cancelBtn = document.getElementById('cancelBtn');

```

Se obtiene el elemento button (botón de cancelar) que tiene `id="cancelBtn"`.

```

    cancelBtn.addEventListener('click', () =>
    {
        document.getElementById('studentId').value = '';
    });
}

```

Cuando el botón se hace clic, se borra el valor del campo oculto `studentId`. Esto asegura que, si el usuario había seleccionado un estudiante para editar, al cancelar ya no se intenta modificar nada por error.

Función `getFormData()`: obtener datos del formulario

```
function getFormData()
{
  return {
    id: document.getElementById('studentId').value.trim(),
    fullname: document.getElementById('fullname').value.trim(),
    email: document.getElementById('email').value.trim(),
    age: parseInt(document.getElementById('age').value.trim(), 10)
  };
}
```

Se crea un objeto (data object) JavaScript con los datos del formulario (sería como una estructura de datos del formulario para trabajar dentro del script):

- `trim()` elimina espacios en blanco innecesarios.
- `parseInt(..., 10)` convierte la edad a número entero (base 10).

Función `clearForm()`: limpiar el formulario

```
function clearForm()
{
  document.getElementById('studentForm').reset();
  document.getElementById('studentId').value = '';
}
```

`reset()` limpia todos los campos del formulario.

Además, se borra manualmente el campo oculto `studentId`, para que no se reutilice por error en un nuevo alta.

Función `loadStudents()`: cargar estudiantes desde el backend

```
async function loadStudents()
```



```

{
  try
  {
    const students = await studentsAPI.fetchAll();
    renderStudentTable(students);
  }
  catch (err)
  {
    console.error('Error cargando estudiantes:', err.message);
  }
}

```

Se llama a `studentsAPI.fetchAll()` para obtener todos los estudiantes desde el backend.

Luego se muestran en pantalla llamando a `renderStudentTable(students)` pasando por parámetro los datos de estudiantes recuperados de la API.

Función `renderStudentTable()`: mostrar estudiantes en una tabla

```

function renderStudentTable(students)
{
  const tbody = document.getElementById('studentTableBody');
  tbody.replaceChildren();

```

Se busca el cuerpo de la tabla donde se muestran los estudiantes, y se eliminan las filas anteriores con `replaceChildren()` para empezar desde cero.

```

  students.forEach(student =>
  {
    const tr = document.createElement('tr');

```

Para cada estudiante, se crea una nueva fila `<tr>`.

```

        tr.appendChild(createCell(student.fullname));
        tr.appendChild(createCell(student.email));
        tr.appendChild(createCell(student.age.toString()));
        tr.appendChild(createActionsCell(student));

```

Se crean celdas para nombre, email y edad, y una celda adicional con los botones de acción (Editar y Borrar).

```

        tbody.appendChild(tr);
    });
}

```

Finalmente, se agrega la fila a la tabla.

Función `createCell(text)`: crear una celda `<td>`

```

function createCell(text)
{
    const td = document.createElement('td');
    td.textContent = text;
    return td;
}

```

Crea una celda (`<td>`) con el texto recibido.

Usa `textContent` para seguridad (no interpreta HTML, solo texto plano así evita la ejecución de scripts maliciosos y el robo de información de los usuarios bloqueando las entradas HTML, más adelante en la carrera verán contenidos sobre ataques [XSS](#)).

Función `createActionsCell(student)`: botones Editar y Borrar

```

function createActionsCell(student)

```

```
{  
  const td = document.createElement('td');
```

Se crea una celda que contendrá los botones de acción.

```
  const editBtn = document.createElement('button');  
  editBtn.textContent = 'Editar';  
  editBtn.className = 'w3-button w3-blue w3-small';  
  editBtn.addEventListener('click', () => fillForm(student));
```

Botón "Editar" con clase azul.

Al hacer clic, se llama a `fillForm(student)` para llenar el formulario con los datos seleccionados.

```
  const deleteBtn = document.createElement('button');  
  deleteBtn.textContent = 'Borrar';  
  deleteBtn.className = 'w3-button w3-red w3-small  
w3-margin-left';  
  deleteBtn.addEventListener('click', () =>  
confirmDelete(student.id));
```

Botón "Borrar" con clase roja.

Al hacer clic, pide confirmación y, si el usuario acepta, llama a `confirmDelete()`.

```
  td.appendChild(editBtn);  
  td.appendChild(deleteBtn);  
  return td;  
}
```

Los botones se agregan a la celda y se retorna el elemento td.

Función `fillForm(student)`: cargar datos en el formulario

```
function fillForm(student)  
{  
  document.getElementById('studentId').value = student.id;
```

```

    document.getElementById('fullname').value = student.fullname;
    document.getElementById('email').value = student.email;
    document.getElementById('age').value = student.age;
}

```

Esta función copia los datos del estudiante seleccionado en el formulario, para permitir su edición.

Función `confirmDelete(id)`: confirmar y borrar

```

async function confirmDelete(id)
{
    if (!confirm('¿Estás seguro que deseas borrar este
estudiante?')) return;

```

Muestra un cuadro de confirmación. Si el usuario cancela, no se hace nada.

```

    try
    {
        await studentsAPI.remove(id);
        loadStudents();
    }
    catch (err)
    {
        console.error('Error al borrar:', err.message);
    }
}

```

Si el usuario confirma, se llama a `studentsAPI.remove()` para borrar en el backend, y se recarga la tabla.

Archivo: frontend/js/api/studentsAPI.js

```

import { createAPI } from './apiFactory.js';

```

- ♦ Importación de función: Aquí estamos trayendo (import) una

función llamada **createAPI**, que está definida en otro archivo: **apiFactory.js**. Equivalente a pedir prestada una herramienta ya construida para no tener que repetir código.

```
export const studentsAPI = createAPI('students');
```

Exportación de objeto API: Estamos creando un objeto llamado **studentsAPI** usando la función **createAPI**, y se lo pasamos el string 'students' como nombre del módulo.

Esto genera automáticamente funciones para obtener, crear, modificar y borrar estudiantes del backend, usando la URL:

```
../../backend/server.php?module=students
```

Exportamos **studentsAPI** para que pueda ser usado desde otros archivos como [studentsController.js](#).

Archivo: frontend/js/api/apiFactory.js

Este archivo define la función genérica **createAPI()** que permite crear objetos de acceso al backend para cualquier módulo (**students**, **subjects**, etc.).

```
export function createAPI(moduleName, config = {})
```

Definición de función exportada:

Se define y exporta una función llamada **createAPI** que recibe dos parámetros:


- **moduleName**: el nombre del módulo (por ejemplo **'students'**).

- **config**: un objeto opcional (vacío por defecto) que permite **configurar** opciones avanzadas si hace falta.

```
const API_URL = config.urlOverride ??
`../../backend/server.php?module=${moduleName}`;
```

Construcción de la URL de acceso al servidor:

- Si **config.urlOverride** está definido, se usa ese valor como URL personalizada.
- Si no, se construye una URL estándar apuntando a **server.php?module=nombre**.

 El operador **??** ([Operador de coalescencia nula](#)) significa: “si el valor de la izquierda es **null** o **undefined**, usar el de la derecha”.

Función interna **sendJSON()**:

```
async function sendJSON(method, data)
```

Esta **función auxiliar interna** se usa para enviar datos al servidor usando los métodos HTTP **POST**, **PUT** o **DELETE**.

```
const res = await fetch(API_URL,
{
  method,
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(data)
});
```

Se hace una petición **fetch()** al backend (**method: GET**):

- **method**: puede ser **POST**, **PUT** o **DELETE**.
- **headers**: indica que se está enviando JSON.

- **body**: convierte los datos a texto JSON antes de enviarlos con `JSON.stringify(data)`.

📌 El uso de `await` indica que se espera la respuesta antes de seguir.

```
if (!res.ok) throw new Error(`Error en ${method}`);
```

Si la respuesta del servidor no fue exitosa (`res.ok` es `false`), se lanza un error indicando el tipo de operación que falló.

```
return await res.json();
```

Si todo salió bien, se convierte la respuesta en un objeto JavaScript usando `res.json()` y se devuelve.

Retorno de `createAPI()`: objeto con funciones del módulo

```
return {
```

La función `createAPI` devuelve un objeto con 4 funciones que representan las operaciones básicas del CRUD:

1. `fetchAll`

```
async fetchAll()
{
  const res = await fetch(API_URL);
  if (!res.ok) throw new Error("No se pudieron obtener los datos");
  return await res.json();
},
```

Obtiene **todos los registros** del módulo desde el servidor.

- Usa `fetch` para hacer una petición GET simple.
- Si la respuesta no es válida, lanza un error.
- Devuelve los datos convertidos en objeto JSON.

2. create(data)

```
async create(data)
{
    return await sendJSON('POST', data);
},
```

Crea un nuevo registro en el backend (alta). Usa la función auxiliar **sendJSON** con método **POST**.

3. update(data)

```
async update(data)
{
    return await sendJSON('PUT', data);
},
```

♦ Actualiza un registro existente en el backend. Usa **sendJSON** con método **PUT**. El parámetro **data** debe incluir el **id** del registro que se desea actualizar.

4. remove(id)

```
async remove(id)
{
    return await sendJSON('DELETE', { id });
}
};
}
```

Elimina un registro del backend, usando **DELETE**. Se pasa un objeto **{ id }** como contenido de la petición.

¿Qué logramos con estos archivos en el frontend?

- **Reutilizar código:** No hace falta escribir de nuevo las funciones de **fetch**, **create**, **update**, **remove** para cada módulo.
- **Generalización:** Con solo cambiar el nombre del módulo (**students**, **subjects**, etc.), el resto del sistema funciona igual.
- **Modularidad:** Los controladores como **studentsController.js** trabajan con **studentsAPI** sin preocuparse por los detalles técnicos del servidor.

Archivo: backend/server.php

Este archivo es el **punto de entrada principal** al backend. Se encarga de **recibir las solicitudes** desde el frontend y redirigirlas al módulo correspondiente (por ejemplo, **students**, **subjects**, etc.).

```
<?php
/**
 *   File       : backend/server.php
 *   Project    : CRUD PHP
 *   Author     : Tecnologías Informáticas B - Facultad de
Ingeniería - UNMdP
 *   License    : http://www.gnu.org/licenses/gpl.txt  GNU GPL
3.0
 *   Date       : Mayo 2025
 *   Status     : Prototype
 *   Iteration  : 3.0 ( prototype )
 */
```

Este bloque es un **comentario de documentación**. No se ejecuta,

pero sirve para informar al lector:

- Qué archivo es y en qué ruta está.
- A qué proyecto pertenece.
- Quién lo hizo y cuándo.
- Qué licencia de uso tiene (GPL 3.0).
- En qué etapa del desarrollo está (prototipo, versión 3.0).

Líneas para mostrar errores:

```
/**FOR DEBUG: */  
// ini_set('display_errors', 1);  
// ini_set('display_startup_errors', 1);  
// error_reporting(E_ALL);
```

Las tres líneas juntas (**ini_set('display_errors', 1);**
ini_set('display_startup_errors', 1); error_reporting(E_ALL);)
son una combinación poderosa que se utiliza comúnmente en el
entorno de desarrollo para:

1. **Mostrar todos los errores y advertencias de PHP.**
2. **Asegurarse de que estos errores sean visibles directamente en la salida de la página.**
3. **Capturar incluso los errores que ocurren muy temprano en el proceso de carga de PHP.**

Importante: Nunca uses estas configuraciones en un **ambiente de producción** (un sitio web que está en línea y es usado por personas). Mostrar errores directamente a los usuarios puede revelar información sensible sobre tu aplicación o servidor (rutas de archivos, nombres de bases de datos, etc.), y además, no es una buena experiencia para el usuario. En


producción, lo ideal es que los errores se registren en un archivo de log y se muestre una página de error genérica al usuario.

Configuración de cabeceras HTTP

```
header("Access-Control-Allow-Origin: *");  
header("Access-Control-Allow-Methods: GET, POST, PUT, DELETE,  
OPTIONS");  
header("Access-Control-Allow-Headers: Content-Type");
```

Estas tres líneas son fundamentales cuando el **frontend** (JavaScript) está separado del **backend** (PHP).

1. **Access-Control-Allow-Origin: *** ➤ Permite que **cualquier sitio web** se comuniquen con este servidor. (*En producción, se suele reemplazar * por una dirección específica.*)
2. **Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS** ➤ Indica qué **métodos HTTP** se permiten. Son las acciones básicas del CRUD.
3. **Access-Control-Allow-Headers: Content-Type** ➤ Permite que las solicitudes incluyan el **tipo de contenido** (por ejemplo, JSON).

 Esto se conoce como configuración **CORS** ([Cross-Origin Resource Sharing](#)).

Función auxiliar para responder con código HTTP

```
function sendCodeMessage($code, $message = "")  
{  
    http_response_code($code);  
    echo json_encode(["message" => $message]);  
    exit();  
}
```

Esta función sirve para **enviar una respuesta HTTP personalizada** al frontend.

- `http_response_code($code)`: envía un código de estado HTTP (como `200`, `400`, `404`).
- `echo json_encode(...)`: convierte un mensaje PHP en texto JSON para que lo entienda el navegador.
- `exit()`: detiene el script inmediatamente.

Manejo de solicitudes HTTP tipo OPTIONS (preflight)

```
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS')
{
    sendCodeMessage(200); // 200 OK
}
```

El navegador a veces envía primero una solicitud de prueba llamada **OPTIONS** para verificar si puede hablar con el servidor. Si la solicitud es **OPTIONS**, respondemos con un código 200 y no hacemos más nada.

Obtención del módulo desde la URL

```
$uri = parse_url($_SERVER['REQUEST_URI']);
$query = $uri['query'] ?? '';
parse_str($query, $query_array);
$module = $query_array['module'] ?? null;
```

Estas líneas extraen el nombre del módulo desde la **URL del navegador**.

Por ejemplo para la URL: `server.php?module=students`

1. `$_SERVER['REQUEST_URI']`: contiene la ruta completa que pidió el navegador.
2. `parse_url(...)`: separa la parte `?module=students`.
3. `parse_str(...)`: convierte el string en un **array asociativo**.
4. Resultado: `$query_array['module']` tendrá el valor `'students'`.

Validación del nombre del módulo

```
if (!$module)
{
    sendCodeMessage(400, "Módulo no especificado");
}
```

Si no se especificó el parámetro `module`, respondemos con un error 400 usando la función auxiliar: `sendCodeMessage`.

```
if (!preg_match('/^\w+$/ ', $module))
{
    sendCodeMessage(400, "Nombre de módulo inválido");
}
```

Si el nombre del módulo contiene caracteres no permitidos, también damos error.

- La expresión regular `^\w+$` significa: solo letras, números y guiones bajos.
- Esto **protege contra inyecciones de código** o rutas maliciosas.

Carga del archivo de rutas del módulo

```
$routeFile = __DIR__ . "/routes/{$module}Routes.php";
```

Construimos el **path del archivo PHP** que maneja ese módulo. Por

ejemplo, si `$module` es `'students'`, entonces el archivo es: `backend/routes/studentsRoutes.php`.

`__DIR__` es una constante que contiene la ruta de esta carpeta (`backend/`).

Inclusión condicional del archivo

```
if (file_exists($routeFile))
{
    require_once($routeFile);
}
else
{
    sendCodeMessage(404, "Ruta para el módulo '{$module}' no encontrada");
}
```

Si el archivo existe, lo cargamos y ejecutamos con `require_once`. Ese archivo será responsable de manejar el `GET`, `POST`, `PUT`, `DELETE` del módulo. Si el archivo no existe, enviamos un error `404 Not Found`.

¿Qué hace server.php?

1. **Recibe la solicitud** del navegador o frontend.
2. **Valida** si el módulo es correcto.
3. **Redirige** la petición al archivo PHP adecuado según el módulo.
4. **Maneja solicitudes OPTIONS** automáticamente para CORS.
5. Si algo falla, **responde con un mensaje de error en JSON**.

Archivo: backend/routes/studentsRoutes.php

Este archivo se encarga de **manejar las rutas** específicas del módulo **students**. Actúa como un "puente" entre la solicitud que llega del frontend y el controlador que se encarga de procesarla.

```
require_once("../config/databaseConfig.php");  
require_once("../routes/routesFactory.php");  
require_once("../controllers/studentsController.php");
```

Usamos **require_once** para **cargar y ejecutar** otros archivos necesarios:

1. **databaseConfig.php**: contiene los datos para conectarse a la base de datos (usuario, contraseña, etc.).
2. **routesFactory.php**: contiene una función (**routeRequest**) que gestiona las rutas generales para **GET**, **POST**, **PUT**, **DELETE**.
3. **studentsController.php**: contiene funciones como **handleGet**, **handlePost**, etc., que hacen el trabajo real (consultar, insertar, modificar o borrar estudiantes).

Nota para estudiantes: **require_once** asegura que un archivo **solo se incluya una vez**, evitando errores por duplicación de funciones.

Comentario de prueba: invocación básica

```
// routeRequest($conn);
```

Esta línea está comentada, pero nos dice que si simplemente escribiéramos: `routeRequest($conn);` Estaríamos diciendo:

“Usá la función genérica `routeRequest` sin modificarla, con el controlador por defecto”.

Eso sería suficiente si no necesitamos hacer **validaciones especiales** para este módulo.

Extensión personalizada de la ruta POST

```
routeRequest($conn, [  
    'POST' => function($conn)  
    {  
        // Validación o lógica extendida  
        $input = json_decode(file_get_contents("php://input"),  
true);  
        if (empty($input['fullname']))  
        {  
            http_response_code(400);  
            echo json_encode(["error" => "Falta el nombre"]);  
            return;  
        }  
        handlePost($conn);  
    }  
]);
```

Esta es la **parte clave del archivo**.

Estamos usando la función `routeRequest` para definir una **ruta personalizada** para las solicitudes **POST** (crear estudiante). Esto **extiende** la lógica por defecto y nos permite agregar validaciones propias.

Paso a paso:

1. `routeRequest($conn, [...])`:

Llama a la función `routeRequest` pasándole:

- la conexión `$conn` a la base de datos
- un array que contiene qué hacer en caso de ciertas

acciones HTTP (**POST** en este caso)

2. **'POST' => function(\$conn) { ... }** Define lo que debe pasar si el método HTTP es POST (crear nuevo estudiante).
3. **json_decode(file_get_contents("php://input"), true);** Lee el **cuerpo del mensaje HTTP**, que viene en formato JSON, y lo convierte en un array de PHP. **php://input** es una forma de acceder al cuerpo bruto de la solicitud.

Esto se usa en solicitudes que **no vienen de un formulario clásico**, como las que hace **fetch()** desde JavaScript.

4. Validación:

```
if (empty($input['fullname']))
{
    http_response_code(400);
    echo json_encode(["error" => "Falta el nombre"]);
    return;
}
```

Si el campo **fullname** no está presente o está vacío, se responde con un error HTTP 400 (bad request) y un mensaje en JSON.

5. Si todo está bien:

```
handlePost($conn);
```

Se llama a la función **handlePost()** que está definida en el archivo **studentsController.php**. Esta función se encarga de **guardar el estudiante en la base de datos**.

Archivo: backend/config/databaseConfig.php

Este archivo tiene la función de **establecer la conexión con la base de datos MySQL**.

Variables de conexión

```
$host = "localhost";  
$user = "students_user_3";  
$password = "12345";  
$database = "students_db_3";
```

Estas cuatro líneas definen los **parámetros necesarios** para conectarse a la base de datos MySQL:

Variable	Significado
\$host	Dirección del servidor de base de datos (usualmente "localhost" en pruebas).
\$user	Usuario de MySQL con permisos sobre la base de datos.
\$password	Contraseña asociada a ese usuario.
\$database	Nombre de la base de datos a la que queremos acceder.


Nota para estudiantes:

- Estos datos deben coincidir con los que estén definidos en tu servidor MySQL.
- Nunca publiques datos reales de producción en GitHub o plataformas abiertas.

Conexión con MySQL

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

Esta línea **crea una conexión** a la base de datos usando la clase **mysqli** (MySQL Improved).

 ¿Qué hace?

- Crea un nuevo objeto **\$conn**.
- Este objeto se puede usar para hacer consultas (**SELECT**, **INSERT**, etc.).

Verificación de error en la conexión

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

Esta parte verifica si la conexión **falló**.

- **connect_error** es una propiedad que contiene el mensaje de error si hubo algún problema al conectar.
- Si hay un error:
 - Se responde con el código HTTP **500 (Error interno del servidor)**.
 - Se envía una respuesta en formato **JSON** que dice: **"Database connection failed"**.
 - **die()** detiene la ejecución del script inmediatamente.

Nota para estudiantes:

- Esta verificación es muy importante: sin conexión a la base de datos, el sistema **no puede funcionar**.

Resultado esperado

Si todo está correcto, al final de este archivo ya tendremos disponible la variable:

```
$conn
```

La cual usaremos en otras partes del backend para **consultar** o **modificar** la base de datos.

Resumen para estudiantes

- Este archivo **se ejecuta automáticamente** en todos los scripts del backend que necesitan conectarse a la base de datos.
- Define los datos de conexión (host, usuario, contraseña, nombre de la base).
- Intenta conectarse usando **new mysqli(...)**.
- Si la conexión falla, se responde con un error HTTP 500 y un mensaje JSON.

Archivo: backend/routes/routesFactory.php

Este archivo define una **función reutilizable** llamada **routeRequest(...)** que se encarga de **gestionar las rutas REST** (GET, POST, PUT, DELETE) y de **conectar cada método HTTP con la función que debe ejecutarse**.

Función principal `routeRequest(...)`

```
function routeRequest($conn, $customHandlers = [], $prefix =  
'handle')  
{
```

Esta línea declara una **función reutilizable** llamada **`routeRequest`**.

Parámetro	Qué representa
<code>\$conn</code>	Objeto de conexión a la base de datos (<code>mysqli</code>).
<code>\$customHandlers</code>	(opcional) Array de funciones personalizadas para ciertos métodos HTTP.
<code>\$prefix</code>	(opcional) Prefijo que se usa para construir los nombres por defecto de los handlers.

Este diseño permite que la función funcione **igual en todos los módulos** (**`students`**, **`subjects`**, etc.), pero también pueda **ser personalizada**.

Detectar el método HTTP de la petición

```
$method = $_SERVER['REQUEST_METHOD'];
```

Esta línea obtiene el método HTTP usado en la solicitud:

- Por ejemplo: **`"GET"`**, **`"POST"`**, **`"PUT"`** o **`"DELETE"`**.

- PHP guarda automáticamente los datos del servidor en el arreglo global `$_SERVER`.

Definir funciones por defecto

```
$defaultHandlers = [  
    'GET'    => $prefix . 'Get',  
    'POST'   => $prefix . 'Post',  
    'PUT'    => $prefix . 'Put',  
    'DELETE' => $prefix . 'Delete'  
];
```

Se crea un array de funciones por defecto (handlers) asociadas a cada método HTTP.

¿Qué hace exactamente?

- Si el prefijo es `'handle'`, se espera que las funciones se llamen:
 - `handleGet($conn)`
 - `handlePost($conn)`
 - `handlePut($conn)`
 - `handleDelete($conn)`

 Es decir: se usan nombres **convencionales** para que el código sea más fácil de mantener.

Combinar handlers personalizados (si los hay)

```
$handlers = array_merge($defaultHandlers, $customHandlers);
```

Esta línea **sobrescribe** los handlers por defecto **con versiones personalizadas**, si el usuario del módulo las definió. Por ejemplo, en **studentsRoutes.php**, se personaliza **POST**, pero los demás (**GET**, **PUT**, **DELETE**) usan los valores por defecto.

Validar si el método es soportado

```
if (!isset($handlers[$method]))
{
    http_response_code(405);
    echo json_encode(["error" => "Método $method no permitido"]);
    return;
}
```

Si el método HTTP no está soportado (por ejemplo, un **PATCH**), se responde con:

- Código HTTP **405** (Método no permitido).
- Un mensaje JSON explicando el error.

Ejecutar el handler correspondiente

```
$handler = $handlers[$method];
```

Esta línea guarda en **\$handler** el nombre de la función que debe ejecutarse para el método solicitado (**handleGet**, **handlePost**, etc.), o bien la función anónima personalizada.

Verificar que sea una función válida

```
if (is_callable($handler))
{
    $handler($conn);
}
```

Se verifica si el valor de `$handler` realmente es una función que se puede ejecutar.

- Si lo es, se llama a la función y se le pasa `$conn`.

Si no es una función válida...

```
else
{
    http_response_code(500);
    echo json_encode(["error" => "Handler para $method no es
válido"]);
}
```

Si el nombre del handler no corresponde a una función válida, se devuelve un:

- Código HTTP **500** (Error interno del servidor).
- Mensaje JSON explicando que el handler es inválido.

Resumen para estudiantes

- Esta función es como un “**despachador de rutas**”. Recibe las peticiones HTTP y llama a la función que las maneja.
- Tiene un comportamiento **por defecto** (basado en nombres convencionales como `handleGet`, `handlePost`, etc.).
- Puede ser **personalizado** para casos especiales (por ejemplo, validaciones en **POST**).
- Devuelve errores en formato **JSON** y con códigos HTTP estandarizados.

Archivo:

backend/controllers/studentsController.php

Este archivo es un **controlador**. Es decir, define funciones que se ejecutan cuando el usuario hace una operación sobre los estudiantes: **ver, crear, modificar o eliminar**.

Las funciones que contiene se llaman desde **routesFactory.php**, y usan funciones del modelo (ubicadas en **models/students.php**) para acceder a la base de datos.

Incluir el modelo de estudiantes

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

Esta línea **incluye** el archivo que contiene las funciones para trabajar con la tabla **students** en la base de datos.

- Esto permite usar funciones como **getAllStudents**, **createStudent**, **updateStudent**, etc.
- Se usa **require_once** para evitar incluir el mismo archivo más de una vez (lo que provocaría error).

Función handleGet(\$conn)

```
function handleGet($conn)
{
    $input = json_decode(file_get_contents("php://input"), true);
```

- Esta función se ejecuta cuando se hace una petición **GET**.
- Intenta leer el contenido del cuerpo de la petición (**php://input**) y convertirlo desde **JSON** a un **array de PHP** usando **json_decode**.

Aunque en **GET** no suele usarse el cuerpo del envío, sino que viaja por URL, este proyecto lo permite para poder filtrar por **id**.

```
if (isset($input['id']))
{
    $student = getStudentById($conn, $input['id']);
    echo json_encode($student);
}
```

Si el JSON recibido contiene un campo **"id"**, se busca un único estudiante con ese ID. Se usa la función del modelo **getStudentById(...)**. La respuesta se convierte nuevamente a JSON usando **json_encode**.

```
else
{
    $students = getAllStudents($conn);
    echo json_encode($students);
}
}
```

Si no se recibió ningún **id**, entonces se devuelven **todos los estudiantes**.

Función **handlePost(\$conn)**

```
function handlePost($conn)
{
    $input = json_decode(file_get_contents("php://input"), true);
```

Se ejecuta cuando el cliente hace una **petición POST**.

El **POST** se usa para **crear un nuevo estudiante**.

Se convierte el JSON recibido en un array.

```
$result = createStudent($conn, $input['fullname'],$input['email'],  
$input['age']);
```

Llama a la función `createStudent(...)`, pasando los datos que vienen del formulario: `fullname`, `email`, `age`.

```
if ($result['inserted'] > 0)  
{  
    echo json_encode(["message" => "Estudiante agregado  
correctamente"]);  
}
```

Si la creación fue exitosa (`inserted > 0`), se devuelve un mensaje JSON de éxito.

```
else  
{  
    http_response_code(500);  
    echo json_encode(["error" => "No se pudo agregar"]);  
}  
}
```

Si hubo un problema, se responde con:

- **Código HTTP 500** (error del servidor).
- Un mensaje JSON de error.

Función `handlePut($conn)`

```
function handlePut($conn)  
{  
    $input = json_decode(file_get_contents("php://input"), true);
```

- Se ejecuta cuando el cliente hace una **petición PUT** (actualización).
- Se leen los datos en formato JSON.

```

$result = updateStudent($conn, $input['id'], $input['fullname'],
$input['email'], $input['age']);
    if ($result['updated'] > 0)
    {
        echo json_encode(["message" => "Actualizado
correctamente"]);
    }

```

Si la actualización del estudiante fue exitosa (**updated > 0**), se devuelve un mensaje JSON de éxito.

```

    else
    {
        http_response_code(500);
        echo json_encode(["error" => "No se pudo actualizar"]);
    }
}

```

Si algo falla, se responde con error HTTP 500.

Función `handleDelete($conn)`

```

function handleDelete($conn)
{
    $input = json_decode(file_get_contents("php://input"), true);

```

- Se ejecuta cuando el cliente hace una **petición DELETE**.
- Lee el ID del estudiante a eliminar.

```

    $result = deleteStudent($conn, $input['id']);
    if ($result['deleted'] > 0)
    {
        echo json_encode(["message" => "Eliminado correctamente"]);
    }

```

Si se pudo eliminar el estudiante, se responde con un mensaje de éxito.

```
else
{
    http_response_code(500);
    echo json_encode(["error" => "No se pudo eliminar"]);
}
}
```

Si ocurre un problema, se devuelve un mensaje de error.

Resumen para estudiantes

- Este archivo **organiza el código que maneja cada tipo de acción**: ver, agregar, modificar, eliminar.
- Cada función:
 - **Lee los datos** del cuerpo de la petición (en JSON).
 - **Llama a una función del modelo** que interactúa con la base de datos.
 - **Devuelve un resultado** (mensaje de éxito o error) también en formato JSON.

IMPORTANTE: Los controladores siguen la convención del prefijo **handle**, lo que facilita su uso en **routesFactory**.

Archivo: backend/models/students.php

Este archivo define una serie de **funciones PHP** que ejecutan **consultas SQL** sobre la tabla **students**. Cada función realiza una acción típica de un CRUD:

- **getAllStudents**: Obtener todos los estudiantes.
- **getStudentById**: Obtener un estudiante por su ID.
- **createStudent**: Insertar un nuevo estudiante.

- **updateStudent**: Modificar un estudiante existente.
- **deleteStudent**: Eliminar un estudiante.

Función: getAllStudents(\$conn)

```
function getAllStudents($conn)
{
    $sql = "SELECT * FROM students";
    return $conn->query($sql)->fetch_all(MYSQLI_ASSOC);
}
```

Explicación:

- Define una consulta SQL que selecciona **todas las filas** de la tabla **students**.
- Usa **\$conn->query(\$sql)** para **ejecutar** esa consulta con MySQLi.
- Luego, **fetch_all(MYSQLI_ASSOC)** convierte el resultado en un **array asociativo** (clave: nombre de columna, valor: dato).
- Este array es fácil de convertir a JSON en el controlador.

IMPORTANTE: MYSQLI_ASSOC significa que los datos vienen como:

```
[
    ["id" => 1, "fullname" => "Juan", ...],
    ["id" => 2, "fullname" => "María", ...],
]
```

Función: getStudentById(\$conn, \$id)

```
function getStudentById($conn, $id)
{
    $stmt = $conn->prepare("SELECT * FROM students WHERE id = ?");
```

```

$stmt->bind_param("i", $id);
$stmt->execute();
$result = $stmt->get_result();
return $result->fetch_assoc();
}

```

Explicación:

1. **Prepara una consulta SQL** con **?** como marcador de posición.
2. **bind_param("i", \$id)** le dice a PHP: "Reemplazá el **?** con un entero (**i**) que viene en **\$id**".
3. **execute()** corre la consulta.
4. **get_result()** obtiene el resultado.
5. **fetch_assoc()** extrae una sola fila como array asociativo (ideal cuando se busca por **id**).

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

```

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

    return [
        'inserted' => $stmt->affected_rows,
        'id' => $conn->insert_id
    ];
}

```

Explicación:

1. Prepara una consulta **INSERT** para agregar un nuevo estudiante.
2. **bind_param("ssi", ...)**:

- `"s"` = string (fullname)
- `"s"` = string (email)
- `"i"` = integer (age)

3. Ejecuta la consulta.

4. Devuelve en un array:

- **inserted**: cuántas filas se insertaron (debería ser 1).
- **id**: el **id** autogenerado por la base de datos. Este array es útil para confirmar el éxito de la operación en el controlador.

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

```
function 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);
    $stmt->execute();

    return ['updated' => $stmt->affected_rows];
}
```

Explicación:

1. Prepara una consulta **UPDATE** para modificar un estudiante por su **id**.
2. **bind_param("ssii", ...)**:

- `"s"`: fullname
- `"s"`: email
- `"i"`: age

- `"i": id`
3. Ejecuta la consulta.
 4. Devuelve cuántas filas fueron modificadas (`updated`). Si `affected_rows = 0`, significa que el estudiante no existía o que los datos eran iguales.

Función: `deleteStudent($conn, $id)`


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

    return ['deleted' => $stmt->affected_rows];
}
```

Explicación:

1. Prepara una consulta `DELETE` para borrar un estudiante por su ID.
2. Se usa `bind_param("i", $id)` para evitar inyecciones SQL.
3. Ejecuta y devuelve cuántas filas fueron eliminadas.

Resumen general para estudiantes

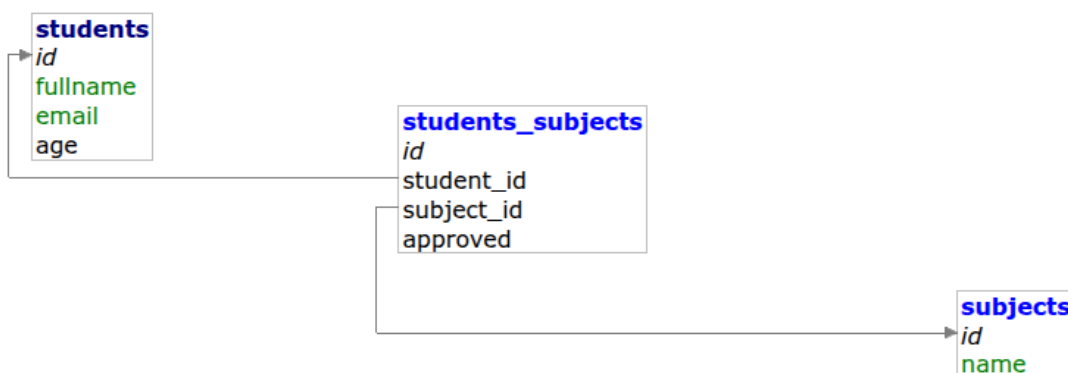
 Este archivo representa la **capa de acceso a datos (modelo)** de tu arquitectura.

- **No tiene lógica de interfaz** ni control de flujo.
- Cada función se conecta directamente con la **base de datos MySQL** usando MySQLi.
- Utiliza `prepare()` y `bind_param()` para evitar **inyección SQL** y manejar los tipos correctamente.

- El resultado de cada función se devuelve como un **array de PHP**, listo para ser convertido a **JSON** en los controladores.

A continuación se va a explicar el módulo que pertenece a la tabla intermedia `students_subjects` que relaciona estudiantes con materias y su aprobación. Se considera que el módulo `subjects` es trivial al tener un comportamiento idéntico al módulo `students`. Solo se abordarán los archivos que son distintos y no los que se reutilizan y fueron explicados anteriormente.

Es importante que tenga en mente al recorrer este ejemplo un diagrama de la relación de las tablas en la base de datos:



Archivo: frontend/html/studentsSubjects.html

Función general del archivo

Este archivo HTML forma parte del **frontend del módulo `students_subjects`**, y su propósito es permitir que un usuario:

- Asocie un estudiante con una materia.

- Marque si dicha materia está aprobada por el estudiante.
- Visualice todas las relaciones actuales.
- Edite o elimine relaciones existentes.

¿Qué tipo de relación representa?

Estamos trabajando con una **tabla intermedia** (habitualmente llamada **students_subjects**), que:

- Asocia **IDs de estudiantes** y **IDs de materias**.
- Incluye un campo adicional: **approved** (booleano).

Esto genera una interfaz ligeramente distinta a los módulos “simples” (**students**, **subjects**), ya que hay que seleccionar **dos entidades externas** y manipular un atributo adicional (**approved**).

Secciones clave del archivo

1. <head>


```
<head>
...
<script type="module"
src="../../js/controllers/studentsSubjectsController.js"></script>
</head>
```

Se importa el controlador JS del módulo (**studentsSubjectsController.js**), el cual será el encargado de:

- Cargar las listas de estudiantes y materias.
- Manejar eventos del formulario.
- Llenar la tabla con datos.
- Hacer peticiones **fetch** al backend usando una capa **studentsSubjectsAPI.js**.

2. Formulario de asignación (#relationForm)

```
<form id="relationForm" class="w3-card ...">
```

-  Este formulario permite crear o editar una **relación estudiante ↔ materia**.
- Tiene un `<input type="hidden" id="relationId" />` para guardar el ID de la relación (para editar).

Campo: Estudiante

```
<select id="studentIdSelect" class="w3-select" required>  
  <option value="" disabled selected>Seleccionar  
  estudiante</option>  
</select>
```

Será poblado dinámicamente con estudiantes desde JS (`studentsAPI.fetchAll()`).

Campo: Materia

```
<select id="subjectIdSelect" class="w3-select" required>  
  <option value="" disabled selected>Seleccionar materia</option>  
</select>
```

Se pobla dinámicamente con materias (`subjectsAPI.fetchAll()`).

Campo: ¿Aprobado?

```
<input type="checkbox" id="approved" class="w3-check" />
```

Permite marcar si el estudiante **aprobó** esa materia.

Botones

```
<button class="w3-button w3-green" type="submit">Guardar</button>  
<button id="cancelBtn" class="w3-button w3-grey"  
type="reset">Cancelar</button>
```

- **Guardar:** Crea o actualiza la relación.
- **Cancelar:** Limpia el formulario y desactiva el modo edición.

3. Tabla de relaciones

```
<table class="w3-table-all ...">
  <thead>
    <tr class="w3-light-blue">
      <th>Estudiante</th>
      <th>Materia</th>
      <th>Aprobado</th>
      <th>Acciones</th>
    </tr>
  </thead>
  <tbody id="relationTableBody"></tbody>
</table>
```

Esta tabla se completa dinámicamente. Cada fila representa una relación entre un estudiante y una materia, e incluye:

- Nombre del estudiante.
- Nombre de la materia.
- Un **Sí/No** visual si está aprobada.
- Acciones (botones de editar y eliminar).

4. Footer

```
<footer class="w3-center w3-margin-top">
  <a class="w3-btn w3-black" href="../index.html">Volver al Panel
  Principal</a>
</footer>
```

Un enlace de navegación estándar.

Estructura semántica completa

Elemento	Descripción
----------	-------------

<code><select id="studentIdSelect"></code>	Selector dinámico de estudiante.
<code><select id="subjectIdSelect"></code>	Selector dinámico de materia.
<code><input type="checkbox" id="approved"></code>	Checkbox para marcar si la materia está aprobada.
<code><input type="hidden" id="relationId"></code>	Oculto para almacenar el ID interno de la relación (modo edición).
<code><form id="relationForm"></code>	Usado para crear o editar relaciones.
<code><tbody id="relationTableBody"></code>	Contenedor donde se insertan las relaciones actuales vía DOM.

Archivo:

frontend/js/controllers/studentsSubjectsController.js

Imports de APIs

```
import { studentsAPI } from '../api/studentsAPI.js';
import { subjectsAPI } from '../api/subjectsAPI.js';
import { studentsSubjectsAPI } from
 '../api/studentsSubjectsAPI.js';
```

Se importan las funciones necesarias para acceder a:

- Estudiantes (**studentsAPI**)
- Materias (**subjectsAPI**)
- Relaciones entre ellos (**studentsSubjectsAPI**)

Inicialización principal

```
document.addEventListener('DOMContentLoaded', () => {
    initSelects();
    setupFormHandler();
    setupCancelHandler();
    loadRelations();
});
```

Al cargarse el DOM:

1. Se cargan los **<select>** con estudiantes y materias.
2. Se configura el evento **submit** del formulario.
3. Se configura el botón **Cancelar**.
4. Se carga la tabla de relaciones actuales.

Funciones

initSelects()

Carga estudiantes y materias en sus respectivos `<select>`.

```
const students = await studentsAPI.fetchAll();
const studentSelect = document.getElementById('studentIdSelect');
```

Cada estudiante se agrega como:

`<option value="ID">Nombre</option>`, igual para materias.

```
option.value = s.id;
option.textContent = s.fullname;
```

Si hay error, se captura con `try/catch` y se muestra en consola.

setupFormHandler()

Maneja el envío del formulario.

```
form.addEventListener('submit', async e => { e.preventDefault();
... });
```

- Llama a `getFormData()` para armar el objeto `{ id, student_id, subject_id, approved }`.
- Si hay `id`, hace `update`; si no, hace `create`.
- Luego limpia el formulario y recarga la tabla.

setupCancelHandler()

Resetea el formulario y limpia el campo oculto `relationId`.

getFormData()

Extrae y devuelve un objeto javascript con los valores

actuales del formulario.

```
approved: document.getElementById('approved').checked ? 1 : 0
```

Convierte el **checkbox** a entero (**1** o **0**), para facilitar la compatibilidad con el backend (MySQL no tiene **true/false**).

clearForm()

Resetea todos los campos del formulario y limpia el campo oculto **relationId**.

loadRelations()

Carga desde el backend la lista completa de relaciones.

```
const relations = await studentsSubjectsAPI.fetchAll();
```

Luego convierte el campo **approved** a número real, porque el backend lo envía como cadena **"0"** o **"1"**:

```
relations.forEach(rel => rel.approved = Number(rel.approved));
```

Esto evita el bug de que **"0"** (string) sea considerado [*truthy*](#).

renderRelationsTable(relations)

Limpia y reconstruye el **<tbody>** de la tabla usando DOM seguro (**createElement**).

```
tr.appendChild(createCell(rel.student_fullname));
tr.appendChild(createCell(rel.subject_name));
tr.appendChild(createCell(rel.approved ? 'Sí' : 'No'));
```

- Muestra "Sí"/"No" para el campo **approved**.
- Agrega celda con botones de acción (**Editar**, **Borrar**).

createActionsCell(relation)

Crea los botones de **Editar** y **Borrar**, y asigna sus eventos.

```
editBtn.addEventListener('click', () => fillForm(relation));
```

```
deleteBtn.addEventListener('click', () =>
confirmDelete(relation.id));
```

fillForm(relation)

Llena el formulario con los datos de una relación seleccionada para edición.

```
document.getElementById('approved').checked = !!relation.approved;
```

!!relation.approved convierte el número **0/1** a **false/true**.

confirmDelete(id)

Confirma el borrado con **window.confirm()**, y si el usuario acepta, llama a **studentsSubjectsAPI.remove(id)**.

Comentarios pedagógicos

Este controlador está especialmente diseñado para enseñar:

- Buen uso de **async/await** y **fetch** con APIs modulares.
- Manipulación segura del DOM sin **innerHTML**.
- Conversión entre tipos para evitar bugs comunes ("**0**" vs **0**).
- Separación de responsabilidades clara y mantenible.

Archivo:

frontend/js/api/studentsSubjectsAPI.js

Es similar a **studensAPI.js**, en los comentarios solo está explicitada la forma en que se puede cambiar la url de la API y cómo se puede extender otros comportamientos de API.

Archivo:

backend/controllers/studentsSubjectsController.php

De igual comportamiento que los otros controladores. Se puede deducir desde:

`backend/controllers/studentsController.php`

Archivo:

backend/models/studentsSubjects.php

Función `assignSubjectToStudent`:

```
function assignSubjectToStudent($conn, $student_id, $subject_id,
    $approved)
{
    $sql = "INSERT INTO students_subjects (student_id, subject_id,
    approved) VALUES (?, ?, ?)";
```

Se prepara un `INSERT` con placeholders `?` para evitar inyecciones SQL.

```
    $stmt = $conn->prepare($sql);
    $stmt->bind_param("iii", $student_id, $subject_id, $approved);
```

Se enlazan los valores a los placeholders. El `"iii"` indica que son tres enteros (`i` de *integer*).

```
    $stmt->execute();
```

Ejecuta la sentencia.

```

return
[
    'inserted' => $stmt->affected_rows,
    'id' => $conn->insert_id
];
}

```

Devuelve la cantidad de filas insertadas y el ID del nuevo registro.

Función getAllSubjectsStudents(\$conn)

```

function getAllSubjectsStudents($conn)
{
    $sql = "SELECT students_subjects.id,
                students_subjects.student_id,
                students_subjects.subject_id,
                students_subjects.approved,
                students.fullname AS student_fullname,
                subjects.name AS subject_name
            FROM students_subjects
            JOIN subjects ON students_subjects.subject_id =
subjects.id
            JOIN students ON students_subjects.student_id =
students.id";

```

Esta consulta recupera todas las relaciones **estudiante - materia**, junto con los **nombres completos** de estudiante y materia, gracias a los **JOIN**.

```

    return $conn->query($sql)->fetch_all(MYSQLI_ASSOC);
}

```

Devuelve el resultado como **array asociativo multidimensional**, listo para convertir a JSON.

Función **getSubjectsByStudent(\$conn, \$student_id)** **(NO USADA)**

```
function getSubjectsByStudent($conn, $student_id)
{
    $sql = "SELECT ss.subject_id, s.name, ss.approved
            FROM students_subjects ss
            JOIN subjects s ON ss.subject_id = s.id
            WHERE ss.student_id = ?";
```

Este es un filtro por estudiante. Se usa alias (**ss**, **s**) para acortar la sintaxis.

```
    $stmt = $conn->prepare($sql);
    $stmt->bind_param("i", $student_id);
    $stmt->execute();
```

Ejecuta la consulta preparada.

```
    $result= $stmt->get_result();
    return $result->fetch_all(MYSQLI_ASSOC);
}
```

return \$result->fetch_all(MYSQLI_ASSOC) trae más de una materia por estudiante en un array asociativo.

Función **updateStudentSubject(\$conn, \$id, \$student_id, \$subject_id, \$approved)**

```
function updateStudentSubject($conn, $id, $student_id,
$subject_id, $approved)
{
    $sql = "UPDATE students_subjects
            SET student_id = ?, subject_id = ?, approved = ?
            WHERE id = ?";
```

Actualiza una relación existente con nuevos valores.

```

$stmt = $conn->prepare($sql);
$stmt->bind_param("iiii", $student_id, $subject_id, $approved,
$id);
$stmt->execute();

return ['updated' => $stmt->affected_rows];
}

```

Devuelve cuántas filas fueron modificadas (idealmente 1).

Función `removeStudentSubject($conn, $id)`

```

function removeStudentSubject($conn, $id)
{
    $sql = "DELETE FROM students_subjects WHERE id = ?";
    $stmt = $conn->prepare($sql);
    $stmt->bind_param("i", $id);
    $stmt->execute();

    return ['deleted' => $stmt->affected_rows];
}

```

Elimina la relación estudiante-materia identificada por `id`.

Resumen para estudiantes

Función	Descripción breve
<code>assignSubjectToStudent()</code>	Crea nueva asignación (INSERT)
<code>getAllSubjectsStudents()</code>	Devuelve todas las relaciones con nombres completos
<code>getSubjectsByStudent()</code>	Devuelve materias de un solo estudiante (¡ver nota!)
<code>updateStudentSubject()</code>	Modifica una relación existente
<code>removeStudentSubject()</code>	Elimina una relación por su ID

Notas pedagógicas

- Las funciones usan **sentencias preparadas**: buena práctica de seguridad.
- Todas devuelven información útil para el controlador (**inserted**, **updated**, **deleted**).
- **fetch_assoc()** en **getSubjectsByStudent()** puede limitar resultados si hay más de una materia asignada.
- Las consultas JOIN permiten mostrar datos legibles en frontend sin consultas adicionales.

Archivo: backend/config/script inicial.sql

```
/******TODA ESTA PRIMER PARTE SE DEBE EJECUTAR COMO
ROOT******/
/*Crear la base de datos*/
CREATE DATABASE IF NOT EXISTS students_db_3
CHARACTER SET utf8
COLLATE utf8_unicode_ci;

/*Crear usuario de la base de datos*/
CREATE USER 'students_user_3'@'localhost' IDENTIFIED BY '12345';

/*Otorgar todos los permisos sobre la base de datos*/
GRANT ALL PRIVILEGES ON students_db_3.* TO
'students_user_3'@'localhost';

/*Aplicar los cambios en los permisos*/
FLUSH PRIVILEGES;
/******
******/

/******A PARTIR DE ACÁ SE PUEDE HACER COMO ROOT
O PARA MAYOR SEGURIDAD CON EL USUARIO
students_user_3******/
/*Usar la base de datos o ingresar en el Adminer o PHPMyAdmin a la
base de datos*/
```

```
USE students_db_3;
```

```
/*Crear la tabla students*/
```

```
CREATE TABLE IF NOT EXISTS students (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    fullname VARCHAR(100) NOT NULL,  
    email VARCHAR(100) NOT NULL UNIQUE,  
    age INT NOT NULL  
) ENGINE=INNODB;
```

```
/*Insertar algunos datos de prueba*/
```

```
INSERT INTO students (fullname, email, age) VALUES  
('Ana García', 'ana@example.com', 21),  
('Lucas Torres', 'lucas@example.com', 24),  
('Marina Díaz', 'marina@example.com', 22);
```

```
/*Crear la tabla subjects*/
```

```
CREATE TABLE subjects (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100) NOT NULL UNIQUE  
) ENGINE=INNODB;
```

```
/*Insertar materias de prueba*/
```

```
INSERT INTO subjects (name) VALUES  
('Tecnologías A'),  
('Tecnologías B'),  
('Algoritmos y Estructura de Datos I'),  
('Fundamentos de Informática');
```

```
/*Crear TABLA INTERMEDIA students_subjects
```

```
Constraints, o restricción UNIQUE(student_id, subject_id):
```

```
garantiza que un estudiante
```

```
no tenga dos veces la misma materia*/
```

```
/*approved: si está aprobada la materia o no (por defecto FALSE).
```

```
ON DELETE CASCADE: si eliminás un estudiante o materia, se borra  
su asignación automáticamente.*/
```

```
CREATE TABLE students_subjects (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    student_id INT NOT NULL,  
    subject_id INT NOT NULL,  
    approved BOOLEAN DEFAULT FALSE,  
    UNIQUE (student_id, subject_id),
```



```

        FOREIGN KEY (student_id) REFERENCES students(id) ON DELETE
CASCADE,
        FOREIGN KEY (subject_id) REFERENCES subjects(id) ON DELETE
CASCADE
    ) ENGINE=INNODB;

/*Insertar relaciones de prueba students_subjects*/
INSERT INTO students_subjects (student_id, subject_id, approved)
VALUES
(1, 1, 1),
(2, 2, 0);

/*VOLVER TODO A CERO, BORRAR BASE DE DATOS Y USUARIO (SE DEBERÍA
EJECUTAR COMO ROOT)*/
/*REVOKE ALL PRIVILEGES, GRANT OPTION FROM
'students_user_3'@'localhost';
DROP USER 'students_user_3'@'localhost';
DROP DATABASE students_db_3;*/

```