Grado en Ingeniería Informática
Programación Web

Práctica Entregable II

PHP+Javascript

Cómo se Hizo

Autor/a:
Mabilia Stella Rinelli Padrón

8 de Junio de 2025
Prof. Juan Manuel Fernández Luna

UNIVERSIDAD DE GRANADA
E.T.S. de Ingenierías Informática y de Telecomunicación

Índice

Conexión a la Base de Datos

Para conectar mi aplicación con la base de datos, he optado por una solución basada en dos archivos: *configuracion.inc.php* y *datosObject.class.inc.php*. La idea principal ha sido separar la configuración de los datos de acceso de la lógica de conexión propiamente dicha, lo que me ha permitido mantener el código más limpio y organizado.

En el archivo *configuracion.inc.php* he definido las constantes necesarias para la conexión. Ahí he incluido el DSN (DB_DSN), que indica el tipo de base de datos (MySQL), el host, el nombre de la base de datos y el conjunto de caracteres. También he especificado el usuario (DB_USUARIO) y la contraseña (DB_CONTRASENIA) para acceder a la base de datos.

Además, he añadido algunas constantes adicionales como el tamaño de página para la paginación (TAMANIO_PAGINA) y los nombres de las dos tablas que uso en el proyecto: *actividades* y *usuarios*.

Por otro lado, en el archivo *datosObject.class.inc.php* he creado una clase abstracta llamada *DatosObject*. Esta clase se encarga de manejar la conexión con la base de datos utilizando PDO (PHP Data Objects). Lo que he hecho ha sido implementar una conexión tipo singleton, usando una propiedad estática para asegurarme de que solo se establezca una única conexión durante toda la ejecución de la aplicación.

Dentro de esa clase, he definido un método llamado *conectar()*, que crea la instancia de PDO si aún no existe. En ese proceso, configuro algunos parámetros importantes: por ejemplo, hago que se lancen excepciones si ocurre algún error haciendo uso de *PDO::ERRMODE_EXCEPTION*, y también he activado el uso de conexiones persistentes para reducir el coste de abrir y cerrar conexiones constantemente. Si algo falla durante la conexión, el sistema muestra un mensaje de error indicando el problema, gracias al manejo de excepciones con *try/catch*.

También he añadido un método *desconectar()*, que simplemente pone la conexión a *null* si en algún momento quiero cerrarla manualmente. Aunque PHP suele cerrar la conexión al terminar el script, me ha parecido útil tener ese control extra.

Con este enfoque, cualquier clase que necesite interactuar con la base de datos puede heredar de *DatosObject* y usar el método *conectar()* directamente, tal como vimos en las clases de prácticas.

Clases que heredan de DatosObject: Usuarios y Actividades

Para gestionar el login y el registro de usuarios en mi aplicación, he creado la clase *Usuarios* en el archivo *usuarios.class.inc.php*. Esta clase hereda de *DatosObject*, lo que me permite reutilizar el sistema de conexión a la base de datos explicado en el apartado anterior.

He definido una estructura de datos con los campos necesarios para representar a un usuario (que son los mismos que tiene el formulario de registro) y he preparado el constructor para inicializar automáticamente los valores recibidos.

Para la autenticación, he implementado el método estático *loginUsuario()*. Este método recibe el nombre de usuario y la contraseña introducidos por el usuario. Primero, establezco la conexión con la base de datos mediante el método *conectar()* heredado de *DatosObject*. Luego, realizo una consulta para buscar al usuario en la tabla correspondiente, utilizando sentencias preparadas con *PDO* para evitar posibles inyecciones SQL. Si se encuentra una coincidencia y la contraseña proporcionada coincide con la que está almacenada (tras verificarla con *password_verify()*), inicio una nueva sesión y guardo en las variables de sesión los datos básicos del usuario (como el *username* y el *tipo*). En caso de error en las credenciales, muestro un mensaje al usuario con la opción de volver al inicio.

También he creado el método *registrarUsuario()*, que se encarga de insertar un nuevo usuario en la base de datos. Este método también utiliza una sentencia preparada y recoge todos los datos del usuario desde un array asociativo. Antes de almacenar la contraseña, la codifico con *password_hash()* para asegurar que se guarde de forma segura. Si el registro se realiza correctamente, redirijo automáticamente al usuario a una página de confirmación. Si se produce algún error durante el proceso, muestro un mensaje explicativo que ayuda a identificar el problema.

Para gestionar las actividades de la aplicación, he creado la clase *Actividades* en el archivo *actividades.class.inc.php*. Al igual que *Usuarios*, esta clase hereda de *DatosObject*.

Dentro de la clase he definido un array protegido llamado *$datos*, que representa la estructura de cada actividad, con los campos *nombre*, *modalidad*, *pistas*, *descripcion*, *imagen* e *id*. En el constructor, inicializo el objeto con los valores proporcionados, siempre que coincidan con las claves definidas en ese array.

Entre los métodos principales, destaco los siguientes:

- → **obtenerActividad($id)**: Este método permite recuperar una única actividad a partir de su identificador. Hace una consulta a la base de datos con una sentencia preparada, y devuelve una instancia de la clase *Actividades* si se encuentra la actividad. Es útil, por ejemplo, para mostrar los detalles de una actividad concreta o para precargar datos en un formulario de edición.

- → **obtenerActividades($filaInicio, $numeroFilas, $orden)**: Este método devuelve un conjunto de actividades paginadas, comenzando en la fila indicada y con el número de registros especificado. Además, utiliza SQL_CALC_FOUND_ROWS para obtener el número total de actividades disponibles, lo que permite implementar correctamente la paginación en la interfaz. El resultado se devuelve como un array con dos

elementos: una lista de objetos *Actividades* y el número total de registros.

→ **obtenerTodas()**: Se trata de un método más sencillo que simplemente devuelve todas las actividades de la base de datos, ordenadas por categoría y nombre.
Está pensado para situaciones en las que no es necesaria la paginación, como en el menú lateral.

→ **registrarActividad($datos)**: Con este método doy de alta una nueva actividad en la base de datos. Utiliza una sentencia INSERT con parámetros preparados y recibe los datos en un array asociativo. Esto permite añadir nuevas actividades a través de un formulario.

→ **editarActividad($datos)**: Este método permite modificar los datos de una actividad existente. Ejecuta una sentencia UPDATE, donde todos los campos se actualizan en base al identificador de la actividad (id). El proceso es muy similar al del registro, pero orientado a la edición.

→ **eliminarActividad($id)**: Finalmente, este método permite eliminar una actividad por su ID. Ejecuta una sentencia DELETE protegida con *bindValue()* y, como en los métodos anteriores, maneja los errores mediante excepciones con *try-catch*.

También he incluido un método *get($campo)* que permite acceder de forma controlada a los valores internos de una actividad.

Con estas dos clases he intentado centralizar y organizar toda la lógica relacionada con los usuarios y las actividades.

1. Darse de alta con el Formulario de Registro

Para permitir que el público general se dé de alta en la aplicación de forma autónoma, he implementado una solución que integra el formulario de registro con la lógica de la clase *Usuarios*. La pieza clave de esta funcionalidad es el archivo *registrar_usuario.php*, que actúa como intermediario entre el formulario HTML y la base de datos.

En primer lugar, el archivo *registrar_usuario.php* importa la clase *Usuarios* mediante *require_once 'usuarios.class.inc.php';*

Esto permite acceder directamente a los métodos definidos dentro de dicha clase, que, como hemos visto, ya está preparada para manejar la inserción de nuevos usuarios en la

base de datos. Gracias a esto, no es necesario repetir lógica de conexión o la validación SQL en el script principal.

A continuación, el script comprueba si la petición se ha realizado mediante el método POST, lo que garantiza que los datos provienen del envío del formulario:

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
   Usuarios::registrarUsuario($_POST);
}
```

Si se cumple esta condición, se llama al método estático *registrarUsuario()* de la clase *Usuarios*, pasándole como argumento el array *$_POST*, que contiene todos los datos introducidos por el usuario en el formulario.

2. Modificación del index para permitir identificación del usuario e inicio y cierre de sesión

Para permitir la identificación de usuarios, he creado un archivo llamado *login.php*, que se encarga de procesar los datos enviados desde el formulario de acceso. En él, recojo el nombre de usuario y la contraseña, y llamo al método *loginUsuario()* de la clase *Usuarios*, que ya había definido anteriormente. Si todo es válido, inicio una sesión utilizando *session_start()* y guardo en variables de sesión el nombre de usuario y su tipo (por ejemplo, si es usuario simple o administrador). De esta forma, el sistema puede reconocer al usuario mientras se desplaza por las distintas secciones del sitio.

Para mostrar contenido diferente según si el usuario ha iniciado sesión o no, he modificado el archivo *index.php* utilizando una estructura condicional *if (!$usuario)* que comprueba si existe una sesión activa (a través de $_SESSION['username']). Si no hay sesión, se muestra el formulario de acceso y un enlace para registrarse; en caso contrario, se imprime un mensaje de bienvenida con el nombre de usuario (<?php echo htmlspecialchars($usuario); ?>) y su tipo (<?php echo htmlspecialchars($tipo); ?>), además de un enlace a *logout.php* para cerrar sesión.

Para cerrar la sesión, creé el archivo *logout.php*, que destruye los datos de sesión utilizando *session_unset()* y *session_destroy()*, y redirige de nuevo al usuario al *index.php*.

3. Modificación del documento actividades para que el administrador pueda gestionar las actividades

Para permitir que el administrador del club pueda gestionar las actividades, he creado una interfaz dentro del archivo *actividades.php* que distingue entre usuarios normales y administradores mediante la variable *$es_admin*, que se evalúa a *true* si el tipo de usuario

en sesión (*$_SESSION['tipo']*) es *'admin'*. A partir de ahí, si el usuario es administrador, se muestra un botón para "Crear nueva actividad" que enlaza con *actividad_nueva.php*, y se añaden enlaces para editar y eliminar cada actividad desde la vista general.

Todas las actividades se recuperan mediante el método *obtenerActividades(...)* de la clase *Actividades,* lo que también permite controlar la paginación en bloques de nueve elementos. Para esto, he calculado el número de página actual con la variable *$_GET['pagina']* y he determinado el total de páginas dividiendo el número total de actividades por el tamaño por página (*TAMANIO_PAGINA*). Luego, he añadido controles de navegación para moverse entre páginas con enlaces "Anterior" y "Siguiente".

Para completar la funcionalidad de gestión de actividades por parte del administrador, he implementado tres archivos adicionales: *actividad_nueva.php* para la creación de nuevas actividades, *actividad_editar.php* para su edición y *actividad_eliminar.php* para su eliminación. En todos ellos, si el usuario que accede no es administrador (*$_SESSION['tipo'] !== 'admin*), es redirigido automáticamente al *index.php* para evitar accesos no autorizados.

En el archivo *actividad_nueva.php*, he creado un formulario accesible solo para administradores, desde el cual pueden introducir todos los datos necesarios para registrar una nueva actividad: nombre, categoría, modalidad, número de pistas, descripción y una imagen asociada. Para facilitar la selección de imágenes, he utilizado la función *glob()* que recorre automáticamente la carpeta *imagenes/* y carga todos los archivos compatibles (.jpg, .png, .gif, etc.) como opciones de un *<select>*. Cuando el formulario se envía mediante *POST*, recojo los datos y los paso al método *Actividades::registrarActividad($datos)*, que se encarga de guardarlos en la base de datos. Si todo va bien, se redirige al usuario de vuelta a *actividades.php*; si ocurre algún error, se muestra un mensaje al usuario.

Por otro lado, en *actividad_editar.php* he seguido una lógica muy similar, pero en este caso primero recupero la actividad específica mediante *Actividades::obtenerActividad($id)* y precargo sus datos en el formulario para que el administrador pueda modificarlos. Al enviar el formulario, los datos actualizados se envían al método *editarActividad($datos)* de la clase *Actividades*, y de nuevo, si no hay errores, se redirige al listado general de actividades. He añadido también un pequeño bloque para mostrar errores en pantalla si algo falla durante la actualización. Ambos formularios están complementados con validación en el lado del cliente utilizando JavaScript *(validar_nuevaAct.js* y *validar_editarAct.js*), en otro apartado hablaremos de estos documentos.

Para completar el ciclo, el archivo *actividad_eliminar.php*, permite al administrador eliminar una actividad existente. Primero, compruebo si el usuario tiene permisos de administrador mediante la sesión y, si no es así, redirijo a *index.php*. Después obtengo el *id* de la actividad a eliminar desde la URL. Si el formulario de confirmación se envía por *POST*, llamo al método *Actividades::eliminarActividad($id)* y redirijo al usuario al listado general. En caso contrario, recupero los datos de la actividad con *Actividades::obtenerActividad($id)* para

mostrar un mensaje de confirmación, donde se solicita al usuario confirmar si desea eliminarla. De esta manera, evito eliminaciones accidentales y capturo y muestro cualquier error que pueda surgir durante la eliminación.

4. Generación automática del menú de actividades

Para generar el menú de actividades de forma dinámica según las categorías presentes en la base de datos, he implementado una solución basada en la recuperación completa de todas las actividades mediante el método *Actividades::obtenerTodas()*.

A continuación, en el archivo *actividades.php*, recorro ese conjunto de datos y organizo las actividades en un array *$menu* agrupado por categoría. Esto me permite construir una estructura de menú lateral en la que cada categoría aparece como título (usando **) y bajo ella se listan las actividades correspondientes como enlaces individuales (*<a href="actividad.php?id=...">*). De esta forma, el menú se adapta automáticamente a los contenidos existentes en la base de datos sin necesidad de modificar manualmente el HTML si se añaden o eliminan actividades.

Además, siguiendo la consigna del ejercicio, no se permite la subida de archivos desde equipos locales por razones de seguridad. Por ello, en los formularios de creación y edición de actividades (*actividad_nueva.php* y *actividad_editar.php*), la selección de imágenes se hace a partir de un listado de archivos que ya están en un directorio del servidor. Como hemos mencionado antes, para lograrlo he utilizado la función *glob()* en PHP. Así, cada actividad puede tener una imagen asignada, pero sin que el admin suba nuevos ficheros.

5. Validación de formularios con javascript

En este apartado explicaré el archivo de validación del formulario de registro de cliente, todos los demás formularios tienen validaciones similares.

Para la validación he creado un script JavaScript que se ejecuta cuando la página se carga y que intercepta el envío del formulario para validarlo. Este script está asociado al evento *DOMContentLoaded*, y dentro de él he vinculado una función al evento *onsubmit* del formulario (que tiene el id *form-altausuario*). Esta función se encarga de recoger los valores de todos los campos, aplicarles las validaciones correspondientes y, si detecta errores, cancelar el envío e informar al usuario.

Recojo el valor de todos los campos del formulario (nombre, apellido, correo, teléfono, fecha de nacimiento,..) usando *document.getElementById(...).value*, y aplico una serie de comprobaciones. Por ejemplo, para validar el campo de correo electrónico, primero recojo el valor y luego uso una expresión regular para comprobar que tiene un formato válido:

```
let email = document.getElementById('email').value.trim();
```

```
if (!email) {
   errores.push("El correo electrónico es obligatorio.");
} else if (!/^[^@\s]+@[^@\s]+\.[^@\s]+$/.test(email)) {
        errores.push("El correo electrónico no es válido. Debe tener formato usuario@dominio.com");
}
```

En este caso, si el campo está vacío, muestro un mensaje indicando que es obligatorio. Si está relleno pero no tiene un formato adecuado, muestro otro mensaje específico sobre el formato correcto.

Otro ejemplo es la validación del teléfono, donde me aseguro de que el campo contenga exactamente 9 dígitos numéricos:

```
let telefono = document.getElementById('telefono').value.trim();
if (!telefono) {
   errores.push("El teléfono es obligatorio.");
} else if (!/^\d{9}$/.test(telefono)) {
   errores.push("El teléfono debe tener 9 dígitos numéricos.");
}
```

También he implementado una validación para que la fecha de nacimiento tenga el formato *YYYY-MM-DD*:

```
let fecha = document.getElementById('fecha_nacimiento').value.trim();
if (fecha && !/^\d{4}-\d{2}-\d{2}$/.test(fecha)) {
   errores.push("La fecha de nacimiento debe tener formato YYYY-MM-DD.");
}
```

Una vez hechas todas las validaciones, si se detecta algún error, cancelo el envío del formulario con *e.preventDefault()* y muestro los errores en pantalla dentro del contenedor con id errores:

```
if (errores.length > 0) {
   e.preventDefault();
   document.getElementById('errores').innerHTML =
      '<ul>' + errores.map(e => `<li>${e}</li>`).join('') + '</ul>';
}
```

6. Carrusel de actividades

Para resolver el ejercicio del carrusel de actividades en la página *index.php*, he creado una sección específica en el HTML donde se muestra inicialmente una sola actividad destacada

con su imagen de fondo, título, modalidad y número de pistas. Toda esta información está encapsulada dentro de un enlace que dirige a la página *actividad.php* con el ID correspondiente a la actividad seleccionada, de forma que el usuario puede acceder directamente al detalle de esa actividad con solo hacer clic en el carrusel. Justo debajo de esta zona, he añadido dos botones con flechas a izquierda y derecha para que el usuario pueda navegar entre las distintas actividades disponibles.

Para que el carrusel funcione dinámicamente, he desarrollado un script en JavaScript (*carrusel.js*) que gestiona el comportamiento del carrusel. En este script, he declarado una variable *indice* que controla qué actividad se está mostrando. La función *mostrarActividad(i)* es la encargada de actualizar la imagen de fondo del carrusel (*style.backgroundImage*), el contenido textual (*textContent* de los elementos de título, modalidad y pistas) y el enlace correspondiente (*href* del contenedor clicable).

Para navegar entre las actividades, he añadido dos manejadores de evento para los botones de las flechas: al hacer clic en la flecha izquierda, se muestra la actividad anterior, y al hacer clic en la derecha, la siguiente. En ambos casos, he usado el operador módulo *(%)* para que, si se llega al principio o al final de la lista, se vuelva al último o al primero respectivamente.

Finalmente, al cargar la página, llamo a *mostrarActividad(indice)* para que se muestre la primera actividad desde el inicio, y toda la lógica se apoya en una variable global *window.actividades* que contiene los datos de todas las actividades cargadas previamente.

Innovaciones

Principalmente he implementado 4 innovaciones. La primera consiste en que, en vez de utilizar cookies para guardar las sesiones, he utilizado las **sesiones** de PHP. El uso de estas lo detallo en la sección 2 del documento. La otra innovación también se puede ver en los apartados anteriores, consiste en el uso de **hashing** para cifrar la contraseña del usuario.

Las otras dos innovaciones están relacionadas con Javascript: le he añadido **animaciones** al ratón y a los títulos de las páginas.

Animación del Ratón

He creado la animación para el cursor del ratón añadiendo un *div* dinámicamente al cargar la página. Este *div*, al que le puse el ID *custom-cursor*, lo añadí al *body* y lo uso como si fuera un cursor alternativo. Para darle estilo, también creé un bloque *<style>* desde JavaScript, donde le asigné una imagen de fondo (una pelota de béisbol, ya que es el tema principal de la página), tamaño fijo, forma redonda y una transición suave para que el movimiento sea fluido. Lo coloqué con *position: fixed* para que siempre siga al ratón, y

desactivé los eventos del puntero sobre él con *pointer-events: none*, así no interfiere con otros elementos de la página.

Por último, con un *addEventListener* que *escucha* el movimiento del ratón, actualizo su posición usando *transform: translate* con las coordenadas del ratón en cada momento. Así consigo que parezca que el nuevo cursor "persigue" al puntero.

Animación de Títulos

Para el título de las páginas a las que redirige el menú, he creado un efecto de "máquina de escribir". Al cargar el contenido del DOM, selecciono el elemento con el ID *titulo* y guardo su texto original en una variable. Luego, borro ese texto del DOM para empezar desde cero.

A continuación, defino una función llamada *escribir* que va mostrando cada letra del texto original una a una, con una pequeña pausa entre cada carácter (en este caso, 75 milisegundos). Para lograr esto, uso *setTimeout* de forma recursiva: cada vez que se añade una letra al título, se espera un poco antes de llamar a la misma función para añadir la siguiente. De esta manera, el texto aparece como si alguien lo estuviera escribiendo en tiempo real.

CREDENCIALES ADMINISTRADOR

usuario: administrador
contraseña: admin123456