

LUCAS EGA ACADEMY

PHP

```
<?php
namespace App\Server;
class Handler {
    public function run() {
        echo "Server running.";
        if ($request) {
            return response()->json($data);
        }
    }
}
?>
```



SERVER-SIDE SCRIPTING *and*
MODERN WEB DEVELOPMENT



Curso Completo de PHP

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México

lucasegaacademy.com

ÍNDICE

PRÓLOGO.....	4
UNIDAD 1: PRIMEROS PASOS CON PHP.....	10
UNIDAD 2: VARIABLES Y TIPOS DE DATOS.....	38
UNIDAD 3: CONTROL DEL FLUJO DEL PROGRAMA.....	70
UNIDAD 4: FUNCIONES Y ARREGLOS.....	102
UNIDAD 5: FORMULARIOS WEB.....	139
UNIDAD 6: COOKIES Y SESIONES.....	178
UNIDAD 7: BASES DE DATOS CON PDO.....	216
UNIDAD 8: CRUD COMPLETO.....	254
UNIDAD 9: SUBIDA DE ARCHIVOS Y SEGURIDAD.....	295
UNIDAD 10: PROYECTO FINAL Y DESPLIEGUE.....	334
EXAMEN FINAL.....	377

PRÓLOGO

Bienvenido a "Curso Completo de PHP", el libro de Lucas EGA Academy para aprender desde cero el lenguaje que mueve buena parte del internet que usas todos los días. Este libro nació del curso en video de 30 clases de la academia, pero aquí tienes todo explicado con mucho más detalle: conceptos a fondo, analogías de la vida real, errores comunes de principiantes, buenas prácticas de la industria, ejemplos paso a paso para probar en tu computadora, prácticas con soluciones explicadas y un examen final para medir lo que aprendiste.

Cada vez que entras a tu red social favorita, que compras en línea, que lees un blog o que llenas un formulario de registro, hay un servidor ejecutando código para atenderte. En muchísimos de esos casos, ese código está escrito en PHP. Aprender PHP es aprender a construir la parte del internet que trabaja detrás de la pantalla: la que guarda tus datos, verifica tu contraseña y te muestra justo lo que pediste.

PHP lleva más de 25 años en activo y no es casualidad. Es el lenguaje detrás de WordPress, que mueve más del 40 por ciento de todos los sitios web del planeta. Facebook nació en PHP. Wikipedia funciona con PHP. Cuando aprendes PHP no estás aprendiendo una moda pasajera: estás aprendiendo una herramienta probada, con millones de empleos y proyectos esperándote.

A quién va dirigido este libro

Este libro es para ti si empiezas desde cero en tecnología y quieres una habilidad concreta, útil y con salida laboral real. PHP es uno de los caminos más rápidos para conseguir tu primer trabajo como programador: las empresas necesitan gente que mantenga y construya sitios y sistemas, y PHP está en todas partes.

Es para ti si ya sabes HTML y CSS y sientes que tus páginas están "muertas": bonitas pero estáticas, incapaces de guardar un dato o responder a un usuario. PHP es la pieza que les da vida. Es para ti si vienes de JavaScript y quieres dominar también el lado del servidor. Es para ti si administras un negocio y quieres entender -o construir tú mismo- el sistema que registra tus ventas, tus clientes y tu inventario.

No necesitas matemáticas avanzadas ni experiencia programando. Solo necesitas una computadora, curiosidad y ganas de practicar. Todo lo demás lo ponemos aquí, paso a paso.

Si vienes de los cursos de HTML, CSS o JavaScript de Lucas EGA Academy, este libro encaja perfecto: ya sabes construir lo que el usuario ve, y ahora aprenderás lo que pasa del otro lado, en el servidor.

Por qué PHP sigue siendo el rey del servidor

PHP significa "PHP: Hypertext Preprocessor", un nombre recursivo que a los programadores nos causa gracia. Nació en 1994 cuando Rasmus Lerdorf escribió unas herramientas para su página personal, y creció hasta convertirse en el lenguaje de servidor más usado del mundo.

La razón de su éxito es práctica: PHP fue diseñado para la web desde el primer día. Mientras otros lenguajes necesitan configuraciones complicadas para mostrar una página, en PHP escribes HTML, le metes una etiqueta `<?php ?>` en medio, y ya tienes una página dinámica. Esa simplicidad hizo que millones de personas aprendieran a programar con PHP, y esa comunidad enorme significa algo muy valioso para ti: cualquier duda que tengas ya la tuvo alguien antes, y la respuesta está a una búsqueda de distancia.

Además PHP es gratis, funciona en Windows, Mac y Linux, y casi cualquier hosting barato del mundo lo soporta. Para publicar tu primer proyecto real no necesitas pagar servidores caros: con un hosting compartido de unos cuantos dólares al mes tienes suficiente. Esa barrera de entrada tan baja es otra razón por la que PHP sigue formando programadores todos los días.

Y no te dejes engañar por quien dice que PHP "ya murió": las versiones modernas de PHP (8 en adelante) son rápidas, tienen tipado estricto opcional, un ecosistema maduro con el framework Laravel, y siguen moviendo una porción gigantesca de la web. Aprender PHP hoy es una apuesta segura.

Cómo está organizado este libro

El libro está organizado en 10 unidades, cada una agrupa 3 de las 30 clases originales del curso en video. Cada unidad sigue la misma estructura probada:

- Introducción: qué verás en la unidad y por qué importa, con ejemplos de la vida real.
- Explicación detallada: los conceptos a fondo, con analogías, errores comunes que cometen los principiantes y buenas prácticas de la industria.
- Ejemplos paso a paso: código PHP completo para copiar y probar en tu computadora, explicado línea por línea.

- Práctica: ejercicios con soluciones explicadas paso a paso. La programación se aprende programando, no solo leyendo.
- Resumen: los puntos clave de la unidad para repasar rápido.

Al final hay un EXAMEN FINAL con 50 preguntas: opción múltiple, verdadero o falso y preguntas abiertas donde escribirás tu propio código. Cada respuesta viene explicada en las soluciones. Si lo resuelves con confianza, estás listo para construir aplicaciones reales con PHP.

Este es tu mapa:

Unidad 1: Primeros pasos con PHP (Clases 1 a 3)

Desde cero absoluto. Clase 1: qué es PHP, cómo funciona la comunicación entre navegador y servidor. Clase 2: instalación paso a paso de XAMPP para tener tu servidor local. Clase 3: tu primer programa y cómo mezclar PHP con HTML. Terminas con un servidor funcionando en tu máquina.

Unidad 2: Variables y tipos de datos (Clases 4 a 6)

La memoria del programa. Clase 4: variables y el signo de pesos. Clase 5: tipos de datos - textos, números, booleanos - y la diferencia entre comillas simples y dobles. Clase 6: operadores para calcular, comparar y concatenar. Aprenderás a guardar y transformar información.

Unidad 3: Control del flujo del programa (Clases 7 a 9)

Tu programa empieza a tomar decisiones. Clase 7: condicionales if, else y elseif. Clase 8: switch para menús y opciones. Clase 9: bucles while, do-while y for para repetir tareas. Ya podrás escribir programas que piensan y trabajan solos.

Unidad 4: Funciones y arreglos (Clases 10 a 12)

Organización profesional. Clase 10: funciones para no repetir código. Clase 11: arreglos para guardar listas de datos. Clase 12: arreglos asociativos y foreach, la combinación más usada de PHP. Tu código se volverá limpio y reutilizable.

Unidad 5: Formularios web (Clases 13 a 15)

El puente con el usuario. Clase 13: formularios HTML que envían datos. Clase 14: recibir datos con \$_GET. Clase 15: recibir datos con \$_POST y validarlos. Aprenderás a capturar lo que el usuario escribe y a verificarlo antes de usarlo.

Unidad 6: Cookies y sesiones (Clases 16 a 18)

Memoria entre visitas. Clase 16: cookies para recordar datos en el navegador. Clase 17: sesiones para guardar datos en el servidor. Clase 18: tu primer sistema de login. Tus aplicaciones empezarán a reconocer a cada usuario.

Unidad 7: Bases de datos con PDO (Clases 19 a 21)

Donde viven los datos. Clase 19: conectar PHP con MySQL usando PDO. Clase 20: consultas para leer y guardar información. Clase 21: sentencias preparadas, la defensa contra la inyección SQL. Conectarás tu código con una base de datos real.

Unidad 8: CRUD completo (Clases 22 a 24)

La habilidad que piden los empleos. Clases 22 a 24: construir un sistema completo para Crear, Leer, Actualizar y Eliminar registros - el famoso CRUD - con formularios, tablas y mensajes de confirmación. El 90 por ciento de las aplicaciones son un CRUD con adornos.

Unidad 9: Subida de archivos y seguridad (Clases 25 a 27)

Nivel profesional. Clase 25: subir imágenes y archivos al servidor. Clase 26: seguridad básica - XSS, validaciones y las trampas más comunes. Clase 27: contraseñas con hash y login seguro. Aprenderás a proteger tu aplicación y a tus usuarios.

Unidad 10: Proyecto final y despliegue (Clases 28 a 30)

Graduación. Clases 28 y 29: construir un blog completo con registro, login, publicaciones con imagen y comentarios, usando todo lo aprendido. Clase 30: subir tu proyecto a un hosting real para que todo el mundo lo vea, y tu mapa de qué aprender después.

Cómo estudiar este libro para sacarle el máximo provecho

Un consejo de alguien que ha enseñado a miles: no leas este libro como una novela. La programación entra por los dedos, no solo por los ojos. Por cada ejemplo que veas, escríbelo tú mismo en tu editor, ejecútalo, y luego cámbiale algo para ver qué pasa. Romper el código a propósito es una de las mejores formas de aprender.

Sigue este método en cada unidad:

1. Lee la introducción y la explicación detallada con calma. Si una analogía te hace clic, ya entendiste el concepto.
2. Copia cada ejemplo paso a paso y ejecútalo en tu servidor local. Lee la explicación línea por línea hasta que cada línea tenga sentido.
3. Haz los ejercicios de la práctica SIN ver la solución primero. Atorarte es parte del proceso: la solución está para cuando de verdad la necesites.
4. Repasa el resumen de la unidad al día siguiente. Si algo no te suena familiar, vuelve a esa sección.

No te saltes unidades pensando que "eso ya lo sabes". Cada unidad construye sobre la anterior, y los detalles que parecen obvios son los que más puntos cuestan en entrevistas de trabajo.

Dedícale sesiones cortas pero constantes: una hora diaria vale más que siete horas un solo día. En unas semanas habrás construido tu primer CRUD, y en un par de meses tendrás un proyecto completo publicado en internet con tu nombre. Ese proyecto, más que cualquier certificado, es lo que te abre puertas.

Lo que necesitas para empezar

Solo tres cosas, todas gratis:

- Una computadora con Windows, Mac o Linux. No necesitas una máquina potente: cualquier computadora de los últimos diez años sirve.
- XAMPP, que instalaremos juntos en la Unidad 1. Te da el servidor Apache, PHP y MySQL en un solo paquete.
- Un editor de código. Te recomiendo Visual Studio Code, gratis y el más usado del mundo, pero cualquiera sirve para empezar.

No necesitas pagar hosting, ni dominios, ni nada más hasta la Unidad 10, donde veremos cómo publicar tu proyecto. Todo el aprendizaje ocurre gratis en tu propia máquina.

Un último mensaje antes de empezar

Programar es un superpoder que se construye con práctica, no con talento. Todos los programadores profesionales que conoces empezaron exactamente donde estás tú: con una pantalla en blanco y la duda de si esto es para ellos. La diferencia entre ellos y los que se rindieron no fue la inteligencia: fue seguir escribiendo código un día más.

PHP es un excelente primer lenguaje de servidor porque te recompensa rápido: en la primera semana ya verás páginas dinámicas funcionando en tu navegador, hechas por ti. Esa sensación de "yo hice esto" es el combustible que te llevará hasta el final.

Así que instala XAMPP, abre tu editor, y empecemos. Nos vemos en la Unidad 1.

Esteban Lucas EGA Academy - lucasegaacademy.com

UNIDAD 1: PRIMEROS PASOS CON PHP

Bienvenido a la primera unidad del Curso Completo de PHP de Lucas EGA Academy. Si este es tu primer contacto con un lenguaje de programación del lado del servidor, estás en el lugar correcto: aquí no asumimos que ya sabes programar. Empezamos desde cero absoluto, con palabras sencillas, ejemplos que puedes copiar y probar en tu propia computadora, y explicaciones que te dicen no solo el "qué", sino también el "por qué" y el "cómo".

En esta unidad vas a aprender exactamente qué es PHP, por qué es uno de los lenguajes más usados en toda la historia de la web, y por qué sigue siendo una habilidad que vale oro en el mercado laboral del 2026. Vas a instalar tu propio servidor local con XAMPP para que puedas programar y ver resultados sin pagar hospedaje ni subir archivos a ningún lado. Vas a escribir tus primeros programas, desde el clásico "Hola mundo" hasta una página que muestra la hora real del servidor y una calculadora que suma dos números. Al terminar, tendrás las bases para construir páginas web dinámicas: páginas que cambian según quién las visita, la hora que sea o los datos que el usuario escriba en un formulario.

Pero antes de tocar código, déjame contarte por qué PHP merece tu tiempo.

Piensa en la web que usas todos los días. Cuando entras a un blog, cuando compras algo en una tienda en línea, cuando lees las noticias o cuando un amigo te dice "mira mi página web", hay una probabilidad muy alta de que haya PHP trabajando detrás de las cortinas. Los estudios de uso de tecnologías web (como los de W3Techs, que analizan millones de sitios) muestran año tras año que PHP está presente en alrededor de tres de cada cuatro sitios web que usan un lenguaje de programación del lado del servidor. Es decir, aproximadamente el 75 por ciento de la web dinámica corre con PHP. Es una cifra enorme si lo piensas: tres de cada cuatro sitios.

¿Y qué sitios son esos? El más famoso es WordPress, el sistema con el que están contruidos más del 40 por ciento de TODOS los sitios web del planeta: blogs, páginas de empresas, tiendas, portafolios, periódicos digitales. WordPress está escrito en PHP. Cuando alguien instala WordPress, está corriendo PHP. Además, gigantes de la tecnología usaron PHP en sus inicios y en partes importantes de su historia: Facebook nació escrito en PHP cuando Mark Zuckerberg lo programó en su dormitorio de Harvard; Wikipedia, la enciclopedia más grande del mundo, corre sobre MediaWiki, que está escrito en PHP; Slack, Mailchimp, Etsy y miles de tiendas hechas con WooCommerce, PrestaShop o Magento también respiran PHP.

¿Por qué un lenguaje con más de 30 años de historia (nació en 1994, creado por Rasmus Lerdorf) sigue tan vivo? Porque hace una cosa muy bien y la hace de forma sencilla: tomar una petición de un navegador y devolver una página web armada al momento, con datos

reales. Y porque su comunidad es gigantesca: si tienes un problema con PHP, alguien ya lo tuvo antes y lo documentó en español, en foros, en videos y en cursos como este.

Otra razón poderosa para aprender PHP en 2026: es la puerta de entrada más amable a la programación del lado del servidor. No necesitas compilar nada, no necesitas configuraciones complicadas de frameworks, no necesitas saber matemáticas avanzadas. Con un editor de texto y un servidor local (que vas a instalar en esta misma unidad, gratis), puedes empezar a crear cosas reales en tu primera hora. Y lo que aprendas aquí (variables, condiciones, ciclos, funciones, formularios, bases de datos) te servirá después para cualquier otro lenguaje, porque son los conceptos universales de la programación.

Esta unidad está dividida en tres clases. La clase 1 responde la pregunta "¿qué es PHP y para qué sirve?", con la analogía del restaurante que te va a acompañar durante todo el curso. La clase 2 es totalmente práctica: instalamos XAMPP paso a paso en Windows, encendemos el servidor Apache y entendemos qué es eso de "localhost" que vas a ver mil veces. La clase 3 es tu primer contacto con el código: las etiquetas de PHP, el punto y coma, el comando echo, los comentarios, y tu primer programa "Hola mundo". Después vienen los ejemplos paso a paso, la práctica con ejercicios resueltos y el resumen.

Un consejo antes de empezar: no te saltes la instalación ni los ejemplos "fáciles". En programación, la confianza se construye haciendo, no leyendo. Cada vez que veas un bloque de código en este libro, cópialo, pégalo en tu editor, guárdalo y ábrelo en tu navegador por localhost. Romper cosas y arreglarlas es la forma más rápida de aprender. Los errores que vas a cometer (y créeme, los vas a cometer) son parte del plan: en esta unidad hay una sección completa dedicada a los errores más comunes de principiantes, para que cuando te pasen, los reconozcas al instante y sepas qué hacer.

¿Listo? Vamos a construir tu primera página dinámica. Tu viaje con PHP empieza ahora mismo.

Explicación detallada

Qué es PHP y cómo funciona: la analogía del restaurante

PHP son las siglas de "PHP: Hypertext Preprocessor", que en español sería algo así como "Preprocesador de Hipertexto". No te preocupes por memorizar el nombre; lo importante es entender qué hace. PHP es un lenguaje de programación que se ejecuta en el servidor, no en tu computadora. Esa es la diferencia fundamental con HTML, CSS y JavaScript, que se ejecutan en el navegador del visitante (del lado del cliente).

Para entenderlo, imagina un restaurante.

Tú eres el cliente sentado en la mesa. El menú que te dan es la página web que ves en tu navegador: bonita, con fotos, con precios, con descripciones. Pero tú no ves la cocina. En la cocina hay cocineros, ingredientes, recetas y un almacén lleno de cosas. Esa cocina es el servidor. El mesero es el que va y viene entre tu mesa y la cocina: toma tu pedido, lo lleva a la cocina, espera a que lo preparen y te trae el platillo terminado.

En la web funciona igual:

- El cliente en la mesa eres tú con tu navegador (Chrome, Firefox, Edge), pidiendo una página.
- El mesero es el protocolo HTTP: lleva tu petición ("quiero ver la página de inicio") hasta el servidor y trae de vuelta la respuesta.
- La cocina es el servidor web (por ejemplo, Apache), donde vive el sitio.
- El cocinero es PHP: recibe el pedido, revisa los ingredientes disponibles (la base de datos, la hora actual, los datos que el usuario escribió en un formulario), prepara el platillo siguiendo la receta (el código PHP que el programador escribió) y le entrega al mesero un plato terminado.
- El plato terminado es una página HTML normal y corriente, que el navegador muestra sin saber (ni necesitar saber) que PHP la cocinó.

Este ciclo se llama petición-respuesta y ocurre cada vez que visitas una página dinámica:

1. Escribes una dirección en el navegador o das clic en un enlace. Eso es la petición.
2. La petición viaja por internet hasta el servidor donde está alojado el sitio.
3. El servidor ve que el archivo pedido termina en .php y se lo pasa al intérprete de PHP.
4. PHP ejecuta el código: puede consultar una base de datos ("dame los últimos 10 artículos"), revisar la hora, leer lo que el usuario escribió en un formulario, hacer cálculos, decidir qué mostrar y qué ocultar.
5. PHP genera como resultado una página HTML pura, sin rastro visible del código que la creó.
6. El servidor envía ese HTML de vuelta al navegador.
7. El navegador muestra la página. Si das clic en "ver código fuente", solo verás HTML: el código PHP nunca llega al visitante. Es un secreto de cocina.

Esta última parte es muy importante por dos razones. Primera, seguridad: como el código PHP nunca sale del servidor, puedes poner ahí contraseñas de bases de datos, lógica de negocio y

validaciones sin que nadie las vea. Segunda, dinamismo: la misma receta puede producir platillos distintos según el pedido. Una sola página PHP puede mostrar "Buenos días" en la mañana y "Buenas noches" en la noche, o mostrarle a cada usuario su propio nombre, usando el mismo archivo.

HTML solo es como un platillo ya preparado y empaquetado: siempre se ve igual. PHP es la cocina completa: prepara cada platillo al momento, según lo que pida cada cliente.

Un detalle técnico que vale la pena conocer desde ahora: PHP es un lenguaje interpretado. Eso significa que no necesitas "compilar" tu programa antes de ejecutarlo, como pasa con lenguajes como C o Java. Escribes el código, lo guardas, lo pides por el navegador y el intérprete de PHP lo lee y lo ejecuta línea por línea en ese momento. Si hay un error, te lo dice en ese momento. Esto hace que el ciclo de prueba sea rapidísimo: cambias una línea, recargas la página y ves el resultado. Para aprender, no hay nada mejor.

Instalación paso a paso de XAMPP en Windows

Para practicar PHP necesitas un servidor en tu propia computadora. No necesitas pagar hospedaje ni tener internet: vas a crear un "servidor local", es decir, tu computadora va a hacer el papel de cliente y de servidor al mismo tiempo. Es como montar el restaurante completo dentro de tu casa para practicar las recetas antes de abrir al público.

La forma más sencilla de lograrlo en Windows es con XAMPP, un paquete gratuito que instala de un solo golpe todo lo que necesitas: Apache (el servidor web, nuestro "restaurante"), PHP (el "cocinero"), MySQL/MariaDB (la "despensa" donde se guardan los datos) y phpMyAdmin (una herramienta visual para administrar la base de datos). XAMPP es gratis, es de código abierto y lo usan millones de estudiantes en todo el mundo. Vamos paso a paso.

Paso 1: Descarga XAMPP. Abre tu navegador y entra al sitio oficial: <https://www.apachefriends.org>. Es el sitio de los creadores de XAMPP, así que es la descarga segura. Verás botones para Windows, Linux y macOS. Elige la versión para Windows (busca el instalador .exe de la versión más reciente de PHP; cualquiera de las versiones 8.x está perfecta para este curso). El archivo pesa alrededor de 150 MB, así que la descarga puede tardar unos minutos según tu conexión.

Advertencia de seguridad: descarga XAMPP únicamente del sitio oficial [apachefriends.org](https://www.apachefriends.org). Hay sitios que ofrecen "XAMPP gratis" con programas extra incluidos que no quieres. Si el sitio te pide pagar o instalar "aceleradores de descarga", estás en el lugar equivocado.

Paso 2: Ejecuta el instalador. Haz doble clic en el archivo descargado. Windows te preguntará si permites que la aplicación haga cambios en el dispositivo; responde que sí. Es posible que

tu antivirus muestre una advertencia; XAMPP es software legítimo y muy conocido, puedes continuar con confianza.

Paso 3: Elige los componentes. El instalador te mostrará una lista de casillas: Apache, MySQL, FileZilla, Mercury, Tomcat, PHP, phpMyAdmin, etc. Para este curso necesitas como mínimo Apache, MySQL, PHP y phpMyAdmin (estos últimos dos suelen venir marcados junto con los principales). Puedes dejar las demás casillas cómo están o desmarcar FileZilla, Mercury y Tomcat si quieres una instalación más ligera; no los usaremos. Haz clic en Next.

Paso 4: Elige la carpeta de instalación. Por defecto será C:\xampp. Te recomiendo dejarla así, porque todos los tutoriales, incluido este libro, asumen esa ruta. Evita instalarlo en "Archivos de programa", porque los permisos de Windows en esa carpeta pueden causar problemas al guardar tus archivos. Haz clic en Next.

Paso 5: Completa la instalación. El instalador copiará los archivos; tardará unos minutos. Al final te preguntará si quieres abrir el Panel de Control de XAMPP; di que sí. Si el firewall de Windows te pregunta si permites que Apache se comuniquen en redes, acepta (marca las casillas de redes privadas); es necesario para que el servidor funcione.

Paso 6: Enciende Apache. Se abrirá el Panel de Control de XAMPP, una ventana con una lista de módulos. En la fila de Apache haz clic en el botón "Start". Si todo sale bien, el fondo de la palabra Apache se pondrá verde y verás los números de puerto (normalmente 80 y 443) junto al nombre. Eso significa que tu servidor web ya está encendido: el restaurante abrió sus puertas.

Si el botón Start no funciona y el fondo se pone rojo, el problema más común es que otro programa ya está usando el puerto 80. El culpable habitual es Skype, o a veces IIS (el servidor de Windows). Ciérralos o cambia el puerto de Apache en la configuración (puertos 8080 es la alternativa clásica). No te asustes: es un problema común con solución conocida, y una búsqueda de "XAMPP Apache port 80 blocked" te dará instrucciones con imágenes.

Paso 7: Comprueba que funciona. Abre tu navegador y escribe en la barra de direcciones: <http://localhost> (sin www, sin .com). Si ves la página de bienvenida de XAMPP con el logo y un menú, felicidades: tu servidor local está funcionando. Acabas de hacer tu primera petición a tu propio servidor y recibiste tu primera respuesta.

Guarda muy bien esta ruta: C:\xampp\htdocs. Esa carpeta es la "cocina" pública de tu servidor: todo archivo que pongas ahí podrá ser visto a través del navegador usando localhost. Por ejemplo, si guardas un archivo llamado saludo.php dentro de htdocs, lo verás escribiendo `http://localhost/saludo.php`. Si lo guardas dentro de una subcarpeta, por ejemplo C:\xampp\htdocs\curso\saludo.php, la dirección será `http://localhost/curso/saludo.php`. Memoriza esta regla: la carpeta htdocs es la raíz de tu sitio local.

Qué es el servidor Apache y qué significa localhost

Ya encendiste Apache, pero ¿qué es exactamente? Apache es el programa que hace el papel del restaurante: está todo el tiempo escuchando, esperando a que alguien pida una página. Cuando llega una petición, Apache busca el archivo correspondiente dentro de htdocs. Si el archivo es un .html, lo sirve tal cual, como entregar un platillo ya empaquetado. Si es un .php, primero se lo pasa al cocinero (el intérprete de PHP) para que lo prepare, y luego entrega el resultado.

Apache es el servidor web más usado del mundo junto con Nginx, y lleva décadas funcionando. No necesitas aprender a configurarlo a fondo para este curso; con saber encenderlo y apagarlo desde el Panel de Control de XAMPP tienes suficiente por ahora. Un buen hábito: enciende Apache (y MySQL cuando trabajemos con bases de datos) cada vez que vayas a practicar, y apágalos cuando termines, desde el botón Stop del panel.

Y localhost, ¿qué es? Es un nombre especial que significa "esta misma computadora". Cuando escribes `http://localhost` en el navegador, le estás diciendo: "oye, navegador, no salgas a internet; pregúntale al servidor que está corriendo aquí mismo, en esta máquina". Tu computadora tiene asignada para sí misma la dirección IP 127.0.0.1, y localhost es simplemente el apodo amable de esa dirección. Por eso `http://localhost` y `http://127.0.0.1` son lo mismo.

Esto tiene una consecuencia práctica muy importante que debes grabarte desde el día uno: para que PHP funcione, SIEMPRE debes abrir tus archivos a través de localhost (o 127.0.0.1), nunca con doble clic desde el explorador de archivos. Si haces doble clic en un archivo .php, el navegador lo abre como archivo local (la dirección empezará con `file:///C:/...`) y verás el código PHP crudo o una página en blanco, porque nadie "cocinó" el archivo: no pasó por Apache ni por PHP. En cambio, si lo abres como `http://localhost/mi-archivo.php`, la petición pasa por Apache, PHP ejecuta el código y recibes la página terminada. Esta es la causa número uno de confusión en principiantes, así que repítelo como mantra: PHP se ve por localhost, no por doble clic.

Para que te quede clarísimo, haz esta prueba cuando termines de instalar: crea con el Bloc de notas un archivo llamado prueba.php dentro de C:\xampp\htdocs con este contenido: `<?php echo "Funciona"; ?>`. Guárdalo. Ahora ábrelo de las dos formas: primero con doble clic (verás el código tal cual o algo raro) y luego escribiendo `http://localhost/prueba.php` en el navegador (verás la palabra Funciona). Esa diferencia es exactamente la diferencia entre pedirle al cocinero y mirar la receta cruda.

La sintaxis básica: etiquetas, punto y coma, echo y comentarios

Ya tienes el restaurante montado. Ahora vamos a escribir la primera receta. La sintaxis de un lenguaje son sus reglas de escritura: dónde empieza el código, cómo termina cada instrucción, cómo se muestran mensajes. PHP tiene una sintaxis sencilla, pero es estricta con los detalles: una coma o un punto y coma fuera de lugar, y el cocinero se niega a trabajar. Vamos parte por parte.

Las etiquetas de apertura y cierre: `<?php` y `?>`. Todo código PHP debe vivir dentro de estas etiquetas. La etiqueta de apertura `<?php` le dice al servidor: "atención, lo que sigue es código PHP, ejecútalo". La etiqueta de cierre `?>` dice: "se acabó el código, lo que sigue es HTML normal". Todo lo que esté fuera de las etiquetas se envía al navegador tal cual, sin procesar.

Esto es lo genial de PHP: puedes mezclarlo con HTML libremente. Puedes tener una página HTML normal y, justo donde necesites algo dinámico (la fecha de hoy, el nombre del usuario, el resultado de una suma), abres una etiqueta `<?php`, escribes el código, la cierras con `?>` y sigues con tu HTML. Es como si en la receta pudieras intercalar notas para el cocinero entre las instrucciones que el cliente sí puede leer.

Ejemplo mínimo para que lo veas:

```
<html>
<body>
<h1>Mi primera página con PHP</h1>
<p>Hoy es un gran día.</p>
<?php
echo "Este texto lo generó PHP.";
?>
</body>
</html>
```

El navegador mostrará el título, el párrafo "Hoy es un gran día." y el texto "Este texto lo generó PHP.". El visitante jamás sabrá cuál parte escribió PHP y cuál era HTML fijo... bueno, en este ejemplo tan simple sí se notaría poco, pero en un sitio real la mezcla es invisible.

Un detalle: si tu archivo contiene SOLO código PHP (sin nada de HTML), puedes omitir la etiqueta de cierre `?>` al final. De hecho, es una buena práctica omitirla en archivos 100 por ciento PHP, porque evita un error clásico: espacios o saltos de línea invisibles después del `?>` que luego causan problemas con encabezados y sesiones (ya verás eso en unidades avanzadas). Por ahora quédate con la regla simple: si el archivo mezcla HTML y PHP, cierra tus etiquetas; si es puro PHP, puedes dejar el `?>` final fuera.

El punto y coma: `;`. En PHP, cada instrucción termina con punto y coma. Es como el punto al final de una oración: le dice al intérprete "aquí termina esta orden, sigue con la siguiente". Sin el punto y coma, PHP no sabe dónde termina una instrucción y dónde empieza la otra, y te mostrará un error de sintaxis.

```
<?php
echo "Hola";
echo "Mundo";
?>
```

Las dos líneas anteriores son dos instrucciones separadas gracias al punto y coma. Si olvidas el punto y coma en la primera, PHP intentará leer las dos líneas como una sola instrucción, se confundirá y te dirá algo como "Parse error: syntax error, unexpected...". Cuando veas ese mensaje, lo primero que debes revisar es si te falta un punto y coma en la línea anterior a la que marca el error. El 90 por ciento de los errores de sintaxis de un principiante son un punto y coma olvidado. Grábatelo.

`echo`: el comando para mostrar cosas. `echo` es la instrucción que le ordena a PHP "muestra este texto en la página". Es el altavoz del cocinero: todo lo que pongas después de `echo` terminará en el HTML que recibe el navegador. Puedes mostrar texto entre comillas, números, el contenido de variables (las veremos en la siguiente unidad a fondo, pero aquí las usaremos un poco) y hasta etiquetas HTML completas.

```
<?php
echo "Hola mundo";
echo 2026;
echo "<h1>Título generado por PHP</h1>";
?>
```

La primera línea muestra el texto `Hola mundo`. La segunda muestra el número `2026` (los números no necesitan comillas). La tercera muestra un título HTML: fíjate qué poderoso es esto, porque significa que PHP puede generar cualquier parte de la página, incluyendo su estructura.

También existe print, que hace casi lo mismo que echo, pero echo es más usado y un poco más rápido. En este curso usaremos echo siempre.

Los comentarios: notas para humanos. Un comentario es una línea (o un bloque) que PHP ignora por completo: no la ejecuta, no la muestra. Sirve para dejar notas dentro del código: explicar qué hace una parte complicada, recordar por qué tomaste una decisión, o desactivar temporalmente una línea para probar algo. Hay tres formas de escribir comentarios en PHP:

```
<?php
// Este es un comentario de una línea (el más usado)
# Este también es un comentario de una línea (estilo alternativo)
echo "Hola"; // puedes comentar al final de una instrucción
/*
Este es un comentario
de varias líneas.
Todo lo que esté aquí dentro
PHP lo ignora.
*/
echo "Mundo";
?>
```

El visitante de tu página verá "HolaMundo" (bueno, "Hola" pegado a "Mundo", porque no pusimos espacios ni saltos de línea HTML) y jamás verá tus comentarios. Ojo: esto es distinto de los comentarios HTML (<!-- como este -->), que SÍ viajan hasta el navegador y cualquiera puede verlos con "ver código fuente". Los comentarios PHP se quedan en la cocina; los comentarios HTML llegan a la mesa. No escribas contraseñas ni secretos ni siquiera en comentarios PHP si el archivo pudiera llegar a manos equivocadas, pero al menos sabes que el visitante normal no los verá.

Acostúmbrate a comentar tu código desde el primer día. Tu "yo" del futuro (el que vuelva a leer ese código dentro de tres meses) te lo va a agradecer. Un buen comentario no repite lo obvio ("// muestra Hola" junto a un echo "Hola" no aporta nada); explica el porqué: "// mostramos el saludo según la hora del servidor".

Errores comunes de principiantes (y cómo reconocerlos)

Cometer errores es parte del aprendizaje, pero hay errores tan comunes que vale la pena conocerlos por adelantado, como las señales de tránsito antes de manejar. Aquí están los cinco errores que comete todo principiante de PHP, con sus síntomas y su solución.

Error 1: olvidar el punto y coma. Síntoma: página en blanco o un mensaje de "Parse error: syntax error" que menciona una línea. Casi siempre la línea culpable es la ANTERIOR a la que

indica el mensaje. Solución: revisa el final de cada instrucción y agrega el ; que falta. Con el tiempo, tus dedos lo escribirán solos.

Error 2: abrir el archivo .php con doble clic en vez de por localhost. Síntoma: ves el código PHP crudo en el navegador, o la página sale en blanco, o la dirección empieza con file:///. Solución: abre siempre tus archivos escribiendo `http://localhost/nombre-del-archivo.php` en la barra del navegador. Recuerda: sin mesero (Apache) no hay cocina (PHP).

Error 3: guardar el archivo fuera de htdocs. Síntoma: el navegador dice "No se puede acceder a este sitio" o "404 Not Found" aunque el archivo existe en tu computadora. Solución: verifica que el archivo esté dentro de `C:\xampp\htdocs` (o en una subcarpeta de htdocs) y que la dirección del navegador coincida con esa ubicación. Un archivo en el Escritorio no lo ve Apache, por más que exista.

Error 4: mezclar comillas. En PHP puedes usar comillas dobles " o simples ' para encerrar texto, pero no puedes mezclarlas en el mismo texto: `echo "Hola' es un error.` Síntoma: parse error. Solución: abre y cierra con el mismo tipo de comilla. Si tu texto necesita llevar comillas dentro, usa el otro tipo afuera: `echo "El dijo 'hola'";` funciona perfecto. En unidades siguientes veremos la diferencia fina entre comillas dobles y simples (las dobles interpretan variables dentro del texto, las simples no), pero por ahora quédate con no mezclarlas.

Error 5: olvidar encender Apache. Síntoma: el navegador dice que no puede conectarse a localhost, aunque ayer todo funcionaba. Solución: abre el Panel de Control de XAMPP y dale Start a Apache. Parece obvio, pero a todos nos ha pasado: es como llegar al restaurante y encontrarlo cerrado porque nadie abrió. Hazlo un hábito: practicar PHP empieza por encender el servidor.

Un consejo general para depurar: cuando algo falle, lee el mensaje de error con calma, de arriba hacia abajo. PHP suele decirte el archivo y la línea aproximada del problema. Los mensajes están en inglés, pero con un traductor a la mano o con la práctica los entenderás rápido. Y nunca, nunca borres todo y empieces de cero por frustración sin antes leer el error: el mensaje es tu mejor pista.

Buenas prácticas desde el primer día

Las buenas prácticas son hábitos que separan al aficionado del profesional. No son reglas del lenguaje (tu código funcionará aunque no las sigas), pero hacen tu vida más fácil y tu código más respetable. Adopta estas cinco desde hoy.

1. Nombres de variables claros. En PHP las variables empiezan con el signo \$ (las estudiaremos a fondo en la unidad 2, pero empieza a acostumbrarte). Usa nombres que

describan lo que guardan: \$nombre_usuario es mejor que \$n, \$precio_total es mejor que \$p. Si el nombre tiene varias palabras, usa el estilo "camelCase" (\$nombreUsuario) o guiones bajos (\$nombre_usuario); elige uno y sé constante. Evita la ñ y los acentos en nombres de variables: técnicamente PHP los permite en versiones recientes, pero te pueden dar dolores de cabeza al mover archivos entre sistemas. Usa \$ano en vez de \$año, por ejemplo.

2. Indentación consistente. La indentación es el sangrado (los espacios al inicio de la línea) que muestra la estructura del código. Cuando una instrucción está "dentro" de otra (lo verás con condiciones y ciclos), indéntala con 4 espacios o una tabulación. Un código bien indentado se lee como un texto bien puntuado; uno sin indentar es cómo leer todo de corrido sin respirar.
3. Guarda tus archivos con codificación UTF-8. Los acentos y la ñ del español necesitan una codificación que los represente bien. UTF-8 es el estándar universal. El Bloc de notas de Windows a veces guarda en ANSI por defecto, lo que convierte tus acentos en símbolos raros. Usa un editor como Visual Studio Code (gratis) o Notepad++, que guardan en UTF-8 por defecto, y agrega en tus páginas HTML la etiqueta `<meta charset="UTF-8">` dentro del `<head>`. Así tus "á é í ó ú ñ ¿ ¡" se verán perfectos en el navegador.
4. Un archivo, una responsabilidad. No mezcles en el mismo archivo el código que calcula cosas con toneladas de HTML decorativo sin orden. En esta unidad está bien mezclar un poco para aprender, pero conforme avances, separa: un archivo para la lógica, y que el HTML solo muestre resultados. Tu futuro yo profesional te lo agradecerá.
5. Prueba en el navegador después de cada cambió pequeño. No escribas 100 líneas y luego pruebes todo junto: si algo falla, no sabrás cuál de las 100 líneas es la culpable. Escribe un poco, guarda, recarga localhost, verifica, y sigue. Este hábito por sí solo te ahorrará horas de frustración.

Ejemplos paso a paso

La teoría sin práctica es cómo aprender a nadar leyendo un manual en la orilla: útil, pero no te mojas. Aquí vienen seis ejemplos completos. Para cada uno: crea el archivo en C:\xampp\htdocs (puedes crear una carpeta llamada curso-php para tener todo ordenado), pega el código, guárdalo con codificación UTF-8 y ábrelo en el navegador con `http://localhost/...` Cada ejemplo viene explicado línea por línea. No avances al siguiente hasta haber visto funcionar el anterior en tu pantalla.

Ejemplo 1: Hola mundo, el ritual de todo programador

Todo lenguaje de programación tiene su "Hola mundo": el primer programa que escribe quien lo aprende. Es un ritual con décadas de historia. El nuestro es así de simple, y aun así ya contiene las tres piezas fundamentales: las etiquetas, el echo y el punto y coma.

Guarda este archivo como `C:\xampp\htdocs\curso-php\hola.php` y ábrelo como `http://localhost/curso-php/hola.php`.

```
<?php
// Mi primer programa en PHP
echo "Hola mundo";
?>
```

Explicación línea por línea:

- Línea 1: `<?php`. Es la etiqueta de apertura. Le dice al servidor: "lo que sigue es código PHP, ejecútalo, no lo muestres tal cual". Sin esta etiqueta, el servidor trataría todo como texto normal.
- Línea 2: `// Mi primer programa en PHP`. Es un comentario de una línea. PHP lo ignora por completo; está aquí para que tú (o quien lea tu código) sepa qué hace este archivo. El navegador no mostrará este texto.
- Línea 3: `echo "Hola mundo";`. Esta es la instrucción real. `echo` ordena mostrar el texto que está entre comillas. El punto y coma al final marca el fin de la instrucción; sin él, PHP marcaría error.
- Línea 4: `?>`. Es la etiqueta de cierre. Indica que terminó el código PHP.

Lo que verás en el navegador será simplemente: Hola mundo. Parece poco, pero piensa en lo que pasó detrás: tu petición viajó a Apache, Apache se lo pasó a PHP, PHP ejecutó el `echo` y devolvió una página con ese texto. Acabas de operar el ciclo petición-respuesta completo. Si das clic derecho y eliges "ver código fuente", verás solo "Hola mundo", sin rastro de las etiquetas `<?php`. El secreto de cocina funcionó.

Prueba a cambiar el texto entre las comillas por tu nombre, guarda y recarga la página (F5). Esa sensación de cambiar algo y ver el resultado al instante es el corazón de programar.

Ejemplo 2: PHP dentro de HTML, la mezcla que hace magia

PHP brilla cuando se mezcla con HTML: el HTML pone la estructura bonita y PHP rellena las partes que cambian. Guarda este archivo como `saludo-html.php`.

```
<html>
<head>
<meta charset="UTF-8">
<title>Mi página dinámica</title>
</head>
<body>
<h1>Bienvenido a mi sitio</h1>
<p>Esta parte es HTML fijo: siempre se ve igual.</p>
<?php
echo "<p>Esta parte la generó PHP al momento.</p>";
?>
<p>Y aquí volvemos al HTML normal.</p>
</body>
</html>
```

Explicación línea por línea:

- Líneas 1 a 9: es HTML puro y normal: la estructura de la página, la codificación UTF-8 (para que los acentos se vean bien), el título de la pestaña, un encabezado h1 y un párrafo fijo.
- Línea 10: <?php. Abrimos la "ventana a la cocina": a partir de aquí el servidor ejecutará código.
- Línea 11: echo "<p>Esta parte la generó PHP al momento.</p>";. Fíjate en algo hermoso: dentro del echo hay etiquetas HTML (<p>...</p>). PHP no distingue: para él es solo texto que debe enviar al navegador, y el navegador lo interpretará como un párrafo. Así es como PHP construye páginas: generando HTML al vuelo.
- Línea 12: ?>. Cerramos la ventana; volvemos a HTML normal.
- Líneas 13 a 15: más HTML fijo y el cierre de la página.

El resultado en pantalla: un título, tres párrafos. El visitante no puede saber cuál párrafo generó PHP. Y aquí está la idea que te acompañará todo el curso: todo lo dinámico de un sitio (el nombre del usuario, la lista de productos, los comentarios de un blog) se genera así, con PHP imprimiendo HTML al momento.

Experimento: cambia el texto dentro del echo por una etiqueta <h2> con tu nombre. Recarga. ¿Viste cómo PHP puede generar encabezados, listas, tablas, lo que sea? Esa es la puerta a todo lo demás.

Ejemplo 3: variables en un saludo personalizado

Una página que siempre dice lo mismo es apenas el calentamiento. El verdadero poder empieza con las variables: cajitas con nombre donde guardas datos que pueden cambiar. Guarda este archivo como saludo-variable.php.

```
<?php
// Guardamos datos en variables (las cajitas con nombre)
$nombre = "Esteban";
$curso = "PHP";
$día = "sábado";

// Usamos las variables dentro del saludo
echo "Hola " . $nombre . ", bienvenido al curso de " . $curso . ". Hoy es " . $día . ".";
?>
```

Explicación línea por línea:

- Líneas 2 a 4: creamos tres variables. \$nombre guarda el texto "Esteban", \$curso guarda "PHP", \$día guarda "sábado". El signo = no significa "igual que en matemáticas", significa "guarda en la cajita de la izquierda el valor de la derecha". Las variables en PHP siempre empiezan con \$. El texto va entre comillas porque es una cadena de caracteres (string).
- Línea 7: el echo arma el saludo uniendo (concatenando) pedazos de texto con las variables. El punto . es el operador de concatenación en PHP: pega cadenas. Entonces "Hola " . \$nombre produce "Hola Esteban", y así sucesivamente hasta formar la oración completa.

El navegador mostrará: Hola Esteban, bienvenido al curso de PHP. Hoy es sábado.

Ahora lo interesante: cambia el valor de \$nombre por el tuyo, guarda y recarga. La página ahora te saluda a ti, sin que hayas tocado la línea del echo. Esa es la magia de las variables: el código es el mismo, los datos cambian. Imagina esto a gran escala: Facebook no tiene una página distinta programada para cada usuario; tiene UNA página PHP que lee la variable con tu nombre y te saluda a ti. Acabas de entender, en miniatura, cómo funcionan los sitios con millones de usuarios.

Nota para curiosos: si usas comillas dobles, puedes meter la variable directamente dentro del texto: echo "Hola \$nombre"; también funciona. Con comillas simples no: echo 'Hola \$nombre'; mostraría literalmente "Hola \$nombre". Esta diferencia la veremos a fondo en la unidad 2; por ahora, el punto . para unir es tu herramienta segura.

Ejemplo 4: la fecha del servidor con date()

¿Y si la página pudiera saber en qué día estamos sin que se lo digas? La función date() le pregunta al servidor qué fecha y hora tiene y te la devuelve formateada como quieras. Es tu primer encuentro con una función de PHP: una herramienta lista para usar. Guarda este archivo como fecha.php.

```
<?php
// date() devuelve la fecha actual del servidor con el formato que le pidamos
$hoy = date("d/m/Y");
echo "Hoy es " . $hoy . ".";
?>
```

Explicación línea por línea:

- Línea 2: `$hoy = date("d/m/Y");`. Llamamos a la función date() pasándole entre paréntesis y comillas el formato deseado: d es el día con dos dígitos, m es el mes con dos dígitos, Y (mayúscula) es el año con cuatro dígitos. El resultado (por ejemplo "19/09/2026") se guarda en la variable \$hoy.
- Línea 3: mostramos la frase armada con la variable.

El navegador mostrará algo como: Hoy es 19/09/2026. (La fecha exacta dependerá del día en que lo pruebes, y esa es precisamente la gracia: mañana dirá otra fecha sin que cambies nada del código.)

La función date() acepta muchas letras de formato: l (L minúscula) da el nombre del día en inglés, H da la hora en formato 24 horas, i los minutos. Prueba cambiar el formato a "d-m-Y H:i" y recarga: verás fecha y hora. Un detalle honesto: date() devuelve la fecha del SERVIDOR, no la de tu reloj de pulsera. En tu computadora local coinciden, pero cuando tu página esté en un hospedaje de otro país, la hora será la de ese servidor; en unidades avanzadas veremos cómo manejar zonas horarias. Por ahora, disfruta tu primera página que "sabe" qué día es.

Ejemplo 5: calculadora simple de suma

Vamos a hacer que PHP calcule. Una calculadora que suma dos números parece poca cosa, pero contiene una idea enorme: PHP puede hacer operaciones matemáticas con datos. Guarda este archivo como suma.php.

```
<?php
// Guardamos dos números en variables
$numero1 = 25;
$numero2 = 17;

// PHP hace la suma y guarda el resultado
$resultado = $numero1 + $numero2;

// Mostramos la operación completa
echo "La suma de " . $numero1 . " + " . $numero2 . " es igual a " . $resultado .
".";
?>
```

Explicación línea por línea:

- Líneas 2 y 3: guardamos los números 25 y 17 en \$numero1 y \$numero2. Fíjate que los números van SIN comillas: con comillas serían texto, sin comillas son números de verdad con los que se puede operar.
- Línea 6: \$resultado = \$numero1 + \$numero2;. PHP suma los valores (25 + 17) y guarda 42 en \$resultado. El + aquí sí es el de matemáticas.
- Línea 9: armamos el mensaje final concatenando texto, variables y más texto.

El navegador mostrará: La suma de 25 + 17 es igual a 42.

Cambia los valores de \$numero1 y \$numero2, recarga, y la página calculará la nueva suma sola. Parece trivial, pero detente a pensar: acabas de programar una máquina que calcula. Todo sistema que haga cuentas (un carrito de compras que suma precios, un banco que calcula intereses, una escuela que promedia calificaciones) está hecho de operaciones como esta, solo que con los números llegando desde formularios y bases de datos en vez de estar escritos a mano. En la unidad de formularios harás esta misma calculadora pero con números que el usuario escriba en la página. Por hoy, celebra: tu programa ya piensa con números.

Ejemplo 6: página que muestra la hora (y saluda según el momento del día)

Cerremos con un ejemplo que junta todo: variables, la función date() y algo nuevo y emocionante, la estructura if, que le permite a PHP tomar decisiones. Guarda este archivo como hora.php.

```

<?php
// Obtenemos la hora actual del servidor (formato 24 horas, sin ceros a la
izquierda)
$hora = date("G");

// Decidimos el saludo según la hora: esto es una estructura de control "if"
if ($hora < 12) {
    $saludo = "Buenos días";
} elseif ($hora < 19) {
    $saludo = "Buenas tardes";
} else {
    $saludo = "Buenas noches";
}

// Mostramos la hora y el saludo correspondiente
echo "Son las " . $hora . " horas. " . $saludo . ", bienvenido a Lucas EGA
Academy.";
?>

```

Explicación línea por línea:

- Línea 2: `$hora = date("G");`. La letra G en `date()` devuelve la hora actual en formato de 24 horas sin cero inicial: a las 9 de la mañana devuelve 9, a las 3 de la tarde devuelve 15, a las 10 de la noche devuelve 22.
- Líneas 5 a 11: la estructura `if/elseif/else`. Se lee así: "si la hora es menor que 12, el saludo es Buenos días; si no, pero es menor que 19, es Buenas tardes; en cualquier otro caso, es Buenas noches". PHP evalúa las condiciones en orden y ejecuta solo el bloque que corresponda. Las llaves `{ }` delimitan cada bloque. Esta es la primera vez que tu programa TOMA UNA DECISIÓN, y es un momento histórico en tu aprendizaje: hasta ahora el programa hacía siempre lo mismo; ahora se comporta distinto según las circunstancias.
- Línea 14: mostramos la hora y el saludo que resultó elegido.

Abre la página en distintos momentos del día (o cambia temporalmente la hora de tu computadora para probar) y verás cómo el saludo cambia solo: en la mañana "Buenos días", en la tarde "Buenas tardes", en la noche "Buenas noches". Un solo archivo, tres comportamientos. Esto es una página dinámica de verdad.

Si algo no te sale a la primera, no te frustres: revisa la sección de errores comunes, verifica los puntos y comas, confirma que Apache esté encendido y que estés abriendo el archivo por localhost. Cada error que corriges hoy es un error que no te detendrá mañana.

Práctica

La mejor forma de fijar lo aprendido es hacerlo con tus propias manos. Aquí tienes cinco ejercicios ordenados de menor a mayor dificultad. Intenta resolver cada uno SIN mirar la solución: crea el archivo, escribe el código, Pruébalo en localhost y solo cuando funcione (o cuando de plano te atores después de intentarlo en serio) revisa la solución. Equivocarte aquí, en privado, es gratis y es donde más se aprende.

Ejercicio 1

Crea un archivo llamado presentación.php que muestre en el navegador tu nombre completo, tu edad y tu ciudad, cada dato en un párrafo HTML distinto (<p>...</p>). Usa tres instrucciones echo, una por párrafo. No olvides las etiquetas PHP ni los puntos y coma.

Solución 1

```
<?php
echo "<p>Me llamo Esteban Ega.</p>";
echo "<p>Tengo 30 años.</p>";
echo "<p>Vivo en Ciudad de México.</p>";
?>
```

Explicación paso a paso:

- Cada línea es una instrucción echo independiente que termina con punto y coma.
- Dentro de cada echo hay un párrafo HTML completo: PHP envía ese texto al navegador y el navegador lo muestra como párrafo, con su salto de línea visual.
- Las etiquetas <?php y ?> envuelven todo el código; sin ellas, el servidor no ejecutaría nada.
- Adapta los datos a los tuyos: cambia el nombre, la edad y la ciudad, guarda y recarga. Verás tus propios datos en pantalla, generados por PHP.
- Error típico aquí: olvidar un punto y coma al final de alguna línea. Si ves "Parse error", revisa el final de cada echo.

Ejercicio 2

Crea un archivo llamado mi-edad-futura.php. Guarda tu edad actual en una variable llamada \$edad. Luego calcula cuántos años tendrás dentro de 10 años, guarda ese resultado en una variable llamada \$edad_futura y muestra el mensaje: "Hoy tengo X años. En 10 años tendré Y

años." (con tus números reales en X e Y). Usa concatenación con el punto para armar el mensaje.

Solución 2

```
<?php
// Mi edad actual
$edad = 30;

// Calculo mi edad dentro de 10 años
$edad_futura = $edad + 10;

// Muestro el mensaje armando el texto con las variables
echo "Hoy tengo " . $edad . " años. En 10 años tendré " . $edad_futura . " años.";
?>
```

Explicación paso a paso:

- Línea 2: guardamos la edad actual (30, cámbiala por la tuya) en la variable \$edad. Va sin comillas porque es un número.
- Línea 5: sumamos 10 a la edad actual y guardamos el resultado en \$edad_futura. PHP calcula 30 + 10 y guarda 40.
- Línea 8: con el operador punto . unimos cinco pedazos: texto, variable, texto, variable, texto. El resultado es una oración completa con tus números.
- Fíjate en los espacios: escribimos "Hoy tengo " con un espacio al final y " años." con un espacio al inicio, para que al unir no queden las palabras pegadas a los números. Este detalle de los espacios es uno de los más olvidados por principiantes: si tu mensaje sale como "Hoy tengo30años", ya sabes que te faltaron espacios en las cadenas.
- Prueba cambiar \$edad y recargar: el mensaje se recalcula solo. Esa es la variable trabajando para ti.

Ejercicio 3

Crea un archivo llamado restaurante.php que use comentarios para documentar cada parte y que muestre un "menú del día" usando PHP dentro de HTML. La página debe tener un título h1 con el nombre de un restaurante inventado por ti, y tres platillos en una lista HTML (con), donde cada platillo lo imprima una instrucción echo distinta. Agrega al inicio del código un comentario de varias líneas que describa qué hace el programa.

Solución 3

```
<html>
<head>
<meta charset="UTF-8">
<title>Menú del día</title>
</head>
<body>
<?php
/*
Programa: Menú del día del restaurante
Descripción: muestra el nombre del restaurante y sus tres
platillos del día. Cada platillo lo imprime un echo distinto
para practicar la mezcla de PHP con HTML.
*/
echo "<h1>Restaurante El Búho Feliz</h1>";
echo "<p>Menú del día:</p>";
echo "<ul>";
echo "<li>Tacos al pastor</li>";
echo "<li>Enchiladas verdes</li>";
echo "<li>Flan napolitano de postre</li>";
echo "</ul>";
?>
</body>
</html>
```

Explicación paso a paso:

- La estructura HTML (html, head, body, meta charset) es el "plato" y el código PHP es lo que lo rellena.
- El comentario de varias líneas entre /* y */ documenta el programa: qué es y qué hace. PHP lo ignora; el navegador nunca lo verá. Acostúmbrate a este hábito.
- Cada echo imprime una pieza del HTML: el título, el subtítulo, la apertura de la lista, cada platillo como elemento y el cierre de la lista.
- Pudimos haberlo hecho con un solo echo largo, pero usar varios echos hace el código más legible: cada línea hace una sola cosa. Legibilidad primero, siempre.
- Cambia el nombre del restaurante y los platillos por los tuyos. Observa que puedes generar listas completas con PHP: imagina una tienda que imprime sus 500 productos así, uno por uno, leyéndolos de una base de datos. El mecanismo es exactamente el mismo que acabas de usar.

Ejercicio 4

Crea un archivo llamado operaciones.php que funcione como una mini calculadora de cuatro operaciones. Guarda dos números en \$a y \$b (elige los valores que quieras), calcula la suma, la resta, la multiplicación y la división, guarda cada resultado en su variable y muestra las cuatro operaciones con sus resultados, cada una en su propia línea. Para los saltos de línea usa la etiqueta
 de HTML dentro de los echos.

Solución 4

```
<?php
// Los dos números con los que vamos a operar
$a = 20;
$b = 4;

// Calculamos las cuatro operaciones básicas
$suma = $a + $b;
$resta = $a - $b;
$multiplicacion = $a * $b;
$division = $a / $b;

// Mostramos cada operación con su resultado, una por línea
echo "Suma: " . $a . " + " . $b . " = " . $suma . "<br>";
echo "Resta: " . $a . " - " . $b . " = " . $resta . "<br>";
echo "Multiplicación: " . $a . " * " . $b . " = " . $multiplicacion . "<br>";
echo "División: " . $a . " / " . $b . " = " . $division . "<br>";
?>
```

Explicación paso a paso:

- Guardamos 20 y 4 en \$a y \$b. Sin comillas, porque son números para operar.
- Calculamos las cuatro operaciones: + suma, - resta, * multiplicación (en programación se usa el asterisco, no la equis) y / división. Cada resultado va a su propia variable con nombre claro.
- En cada echo armamos la expresión completa: "Suma: 20 + 4 = 24" y al final agregamos
, que es la etiqueta HTML de salto de línea. Sin el
, todo saldría pegado en un solo renglón, porque el navegador ignora los saltos de línea del código fuente.
- Cambia los valores de \$a y \$b y recarga: las cuatro operaciones se recalculan. Cuidado con poner \$b = 0: dividir entre cero es un error matemático y PHP te lo hará saber. En la unidad de condiciones aprenderás a protegerte de eso.

- Este ejercicio resume la unidad entera: variables, operaciones, concatenación, echo y HTML generado por PHP, todo junto.

Ejercicio 5

Crea un archivo llamado `saludo-inteligente.php` que combine fecha y decisión, como el ejemplo 6 pero más completo. Debe mostrar: la fecha completa de hoy en formato "día/mes/año", la hora actual en formato de 24 horas con minutos, y un mensaje que cambie según la hora: "Buenos días" (antes de las 12), "Buenas tardes" (de 12 a 18:59) y "Buenas noches" (de 19 en adelante). Además, si el día de la semana es sábado o domingo (usa `date("N")`, que devuelve 1 para lunes hasta 7 para domingo), debe agregar al final el texto "¡Feliz fin de semana!". Todo dentro de un HTML básico con título.

Solución 5

```

<html>
<head>
<meta charset="UTF-8">
<title>Saludo inteligente</title>
</head>
<body>
<?php
// Obtenemos fecha, hora y día de la semana del servidor
$fecha = date("d/m/Y");
$hora = date("G");
$minutos = date("i");
$dia_semana = date("N"); // 1 = lunes, ..., 6 = sábado, 7 = domingo

// Elegimos el saludo según la hora
if ($hora < 12) {
    $saludo = "Buenos días";
} elseif ($hora < 19) {
    $saludo = "Buenas tardes";
} else {
    $saludo = "Buenas noches";
}

// Armamos el mensaje principal
echo "<h1>" . $saludo . "</h1>";
echo "<p>Hoy es " . $fecha . " y son las " . $hora . ":" . $minutos . "
horas.</p>";

// Si es fin de semana (sábado = 6 o domingo = 7), agregamos el mensaje extra
if ($dia_semana == 6 || $dia_semana == 7) {
    echo "<p>¡Feliz fin de semana!</p>";
}
?>
</body>
</html>

```

Explicación paso a paso:

- Líneas de datos: con date() obtenemos cuatro piezas: la fecha en formato d/m/Y, la hora en formato G (0 a 23), los minutos con i, y el número del día de la semana con N (1 lunes, 7 domingo). Cada dato va a su variable.
- El primer if/elseif/else elige el saludo comparando la hora: menos de 12 es mañana, menos de 19 es tarde, lo demás es noche. Solo un bloque se ejecuta.

- Mostramos el saludo como título h1 y la fecha con hora en un párrafo, concatenando variables y texto.
- El segundo if es independiente: pregunta si el día es 6 (sábado) O 7 (domingo). El operador || significa "o": basta que una de las dos condiciones sea verdadera. Si es fin de semana, se imprime el párrafo extra; entre semana, no aparece nada.
- Fíjate en el == (doble igual) para comparar: un solo = GUARDA un valor en una variable, el doble == PREGUNTA si dos valores son iguales. Confundirlos es un error clasicísimo: if (\$dia_semana = 6) guardaría 6 en la variable en vez de comparar, y el resultado sería un desastre silencioso. Cuando un if se comporte raro, revisa que hayas usado ==.
- Este programa ya es una pequeña aplicación de verdad: lee el reloj del servidor, toma dos decisiones y genera una página distinta según el momento. Si lograste hacerlo (aunque sea con ayuda de la solución), ya piensas como programador: descomponer un problema en datos, decisiones y salida.

Preguntas frecuentes de principiantes

Antes del resumen, respondamos las dudas que casi todos los estudiantes tienen al terminar su primera semana con PHP. Si alguna de estas preguntas ya te pasó por la cabeza, vas por buen camino: significa que estás experimentando de verdad.

¿Por qué mi página sale en blanco y no muestra ningún error? La página en blanco es el síntoma más desconcertante porque no te dice nada. En la mayoría de los casos hay un error de PHP pero el servidor está configurado para no mostrarlo. Revisa tres cosas en orden: primero, que Apache esté encendido en el Panel de Control de XAMPP; segundo, que estés abriendo el archivo por `http://localhost` y no con doble clic; tercero, busca un punto y coma olvidado o una llave sin cerrar. Si nada de eso funciona, agrega temporalmente estas dos líneas al inicio de tu archivo, justo después de `<?php: error_reporting(E_ALL); ini_set('display_errors', 1);`. Esto le ordena a PHP que muestre todos los errores en pantalla. Cuando termines de depurar, quítalas o coméntalas, porque en un sitio real los errores no deben verse.

¿Puedo usar el Bloc de notas de Windows para programar? Puedes, pero no deberías. El Bloc de notas no colorea el código, no te avisa de errores obvios y a veces guarda en codificación ANSI, lo que rompe tus acentos. Descarga Visual Studio Code (gratis, de Microsoft) o Notepad++: colorean la sintaxis (verás las etiquetas PHP de un color, los textos de otro, los comentarios de otro), numeran las líneas (útil cuando PHP te dice "error en la línea 14") y guardan en UTF-8 por defecto. Un buen editor es como un buen cuchillo para el cocinero: no hace la receta por ti, pero todo es más fácil.

¿PHP y JavaScript son lo mismo? No, aunque los nombres se parezcan y a veces trabajen juntos. PHP se ejecuta en el servidor antes de que la página llegue al navegador; JavaScript (en su uso clásico) se ejecuta en el navegador del visitante después de que la página llegó. PHP no puede reaccionar a un clic sin recargar la página; JavaScript sí. JavaScript no puede consultar directamente una base de datos del servidor; PHP sí. En un sitio moderno ambos colaboran: PHP prepara los datos en el servidor y JavaScript les da interactividad en el navegador. Aprender PHP primero es una excelente decisión porque te enseña la lógica de programación que después aplicarás en JavaScript y en cualquier otro lenguaje.

¿Necesito saber HTML antes de aprender PHP? No es estrictamente obligatorio para esta unidad, porque PHP puede generar páginas él solo, pero te recomiendo fuertemente aprender lo básico de HTML en paralelo (etiquetas como h1, p, ul, li, a, img y la estructura de un documento). Piensa en HTML como los platos y la vajilla: PHP cocina la comida, pero necesitas dónde servirla. Este curso asume un HTML muy básico y te lo irá recordando cuando sea necesario, así que no te detengas por eso.

¿Y si quiero que mis páginas las vea otra persona? Mientras prácticas en localhost, solo tú (en tu computadora) puedes verlas. Cuando quieras publicar en internet necesitarás un hospedaje (hosting) con PHP, que hoy cuesta desde unos pocos dólares al mes, y un dominio. Casi todos los hospedajes compartidos traen PHP y MySQL preinstalados: subirías tus archivos .php por FTP y listo, sin instalar nada. Pero no corras: domina primero el servidor local. Todo lo que aprendas en localhost funcionará igual en un hospedaje real, porque PHP es PHP en cualquier servidor.

¿PHP está muerto? Es una pregunta que verás en foros y redes, normalmente de gente que no lo usa. Los números responden solos: tres de cada cuatro sitios dinámicos, WordPress moviendo más del 40 por ciento de la web, millones de empleos activos pidiendo PHP, y versiones modernas (PHP 8.x) rápidas y con sintaxis actual. "PHP está muerto" se dice desde hace 15 años y el lenguaje sigue aquí, más vivo y más rápido que nunca. Lo que sí está muerto es el PHP mal escrito de 2005; el PHP moderno, bien practicado, es una habilidad que paga las cuentas.

Ideas para seguir practicando por tu cuenta

Los ejercicios resueltos son el arranque, pero la fluidez llega cuando juegas con el código sin red de seguridad. Aquí tienes siete retos cortos para esta semana, ordenados de fácil a travieso. Ninguno requiere nada que no hayas visto en esta unidad.

- Modifica el ejemplo del saludo con variables para que use tu nombre real, tu ciudad y tu comida favorita, y muéstralo con tres echos distintos.
- Cambia el ejemplo de la fecha para que muestre la fecha en formato "19 de septiembre de 2026", armando el texto con concatenación (pista: `date("d")`, `date("m")` y `date("Y")` por separado).
- Haz que la calculadora del ejemplo 5 reste, multiplique y divida además de sumar, mostrando las cuatro operaciones.
- En el ejemplo de la hora, agrega un cuarto saludo: si la hora es mayor o igual a 22 o menor que 5, que diga "Buenas noches... ¡a dormir!".
- Crea una página que muestre tu edad en días aproximados (edad por 365) usando una multiplicación.
- Crea una página con tu top 5 de películas o canciones usando una lista `<ul>` generada con cinco echos.
- Rompe algo a propósito: quita un punto y coma, mezcla comillas, apaga Apache y abre por localhost. Observa cada mensaje de error y anota qué lo causó. Este es el entrenamiento más valioso: la próxima vez que veas esos mensajes en un proyecto real, no entrarás en pánico.

Un último consejo de esta unidad: guarda todos tus archivos de práctica en una carpeta ordenada, por ejemplo `C:\xampp\htdocs\curso-php`, con nombres claros. Dentro de unas semanas, cuando sepas mucho más, volverás a estos primeros archivos y te reirás con cariño de lo simples que eran. Esa risa será la prueba de cuánto avanzaste.

Resumen de la unidad

Llegaste al final de la primera unidad. Si instalaste XAMPP, encendiste Apache, viste tus archivos por localhost y lograste que PHP te saludara por tu nombre y te dijera la hora, ya hiciste más que la mayoría de la gente que "quiere aprender a programar" algún día. Repasemos lo esencial:

- PHP es un lenguaje del lado del servidor: se ejecuta en el servidor y genera páginas HTML al momento, según los datos y las circunstancias de cada petición.
- El ciclo petición-respuesta funciona como un restaurante: el navegador (cliente) pide, HTTP (mesero) lleva el pedido, Apache (restaurante) lo recibe, PHP (cocinero) prepara la página con los ingredientes disponibles y devuelve HTML terminado. El código PHP nunca llega al visitante.

- PHP está detrás de aproximadamente tres de cada cuatro sitios dinámicos de la web: WordPress (más del 40 por ciento de todos los sitios), Wikipedia, y los inicios de Facebook, entre miles de ejemplos.
- XAMPP instala gratis y de un golpe Apache, PHP, MySQL y phpMyAdmin en Windows; la carpeta C:\xampp\htdocs es la raíz de tu sitio local.
- Apache es el servidor web que escucha peticiones; localhost (127.0.0.1) significa "esta misma computadora" y es la única forma correcta de abrir archivos PHP: nunca con doble clic desde el explorador.
- Todo código PHP vive entre las etiquetas `<?php` y `?>`; fuera de ellas, el contenido se envía tal cual como HTML.
- Cada instrucción PHP termina con punto y coma (;); olvidarlo es el error número uno de los principiantes.
- `echo` muestra texto, números, variables y HTML en la página; es el altavoz de PHP.
- Los comentarios (`//`, `#` y `/* */`) son notas que PHP ignora y que el visitante nunca ve; úsalos para documentar tu código.
- Las variables (que empiezan con `$`) guardan datos que pueden cambiar; el punto `.` une (concatena) textos y variables.
- La función `date()` devuelve la fecha y hora del servidor con el formato que le pidas (`"d/m/Y"`, `"G"`, `"i"`, `"N"`, entre otros).
- PHP hace operaciones matemáticas (+, -, *, /); los números para operar van sin comillas.
- La estructura `if/elseif/else` permite que el programa tome decisiones y se comporte distinto según las condiciones.
- Errores clásicos que ya sabes reconocer: falta de punto y coma, abrir el `.php` por `file://` en vez de `localhost`, archivo fuera de `htdocs`, comillas mezcladas, Apache apagado y confundir `=` (guardar) con `==` (comparar).
- Buenas prácticas adoptadas desde hoy: nombres de variables claros, indentación consistente, archivos en UTF-8 con `<meta charset="UTF-8">`, probar en el navegador después de cada cambio pequeño y leer los mensajes de error con calma.
- Para mostrar errores en pantalla durante el desarrollo puedes usar `error_reporting(E_ALL)` junto con `ini_set('display_errors', 1)` al inicio del archivo; recuerda quitarlas o comentarlas antes de publicar.

- Un editor con coloreado de sintaxis como Visual Studio Code o Notepad++ hace el trabajo más cómodo que el Bloc de notas: numera líneas, guarda en UTF-8 y te ayuda a ver la estructura del código de un vistazo.

En la Unidad 2 profundizaremos en las variables y los tipos de datos: textos, números, booleanos, arreglos y todo lo que puedes guardar en esas cajitas con nombre. Aprenderás a pedir datos al usuario con formularios, a validar lo que escriben y a tomar decisiones más elaboradas con condiciones anidadas. Cada unidad construye sobre la anterior, así que lo que hoy parece un truco simple (mostrar la hora, sumar dos números) mañana será parte de un sistema completo: un blog, una tienda, un panel de administración.

Guarda este archivo de la unidad y tus programas de práctica en un lugar seguro. Dentro de unas semanas vas a querer volver a ellos, no por nostalgia, sino para medir tu progreso con tus propios ojos. Cada curso de esta academia sigue la misma filosofía: primero entender, luego hacer, y al final construir algo real con tus manos. Y recuerda la regla de oro de este curso: ante la duda, prueba. Cambia el código, recarga localhost, observa qué pasa. Nos vemos en la Unidad 2: tu restaurante ya está abierto y el cocinero ya sabe sus primeras recetas.

El programador que experimenta aprende diez veces más rápido que el que solo lee, así que abre tu editor y empieza hoy mismo.

UNIDAD 2: VARIABLES Y TIPOS DE DATOS

Bienvenido a la Unidad 2 del Curso Completo de PHP de Lucas EGA Academy. Si en la Unidad 1 aprendiste a instalar tu ambiente, a abrir tu primer archivo .php y a lograr que el servidor te mostrara tu primer "Hola mundo" en pantalla, en esta unidad vamos a dar el salto más importante de todos: vas a enseñarle a PHP a recordar cosas.

Piensa en esto por un momento. Un programa que no recuerda nada es como una persona con amnesia total: puede decirte su nombre una vez, pero al segundo siguiente ya lo olvidó y tiene que empezar de cero. ¿De qué te serviría un programa que saluda a un usuario si a los dos segundos ya no sabe cómo se llama? ¿De qué te serviría una tienda en línea que suma el precio de los productos pero que no puede guardar el total para mostrártelo al final? ¿De qué te serviría un juego donde tu puntuación se borra cada vez que pasa algo? De nada. Absolutamente de nada. Un programa sin memoria no puede hacer nada útil, y las variables son, ni más ni menos, la memoria de tu programa.

En esta unidad vas a aprender qué es una variable, cómo se crea en PHP, qué tipos de datos puede guardar y cómo usar operadores para hacer cálculos, comparar valores y tomar decisiones básicas. Estas son las clases 4, 5 y 6 del curso, unidas en un solo recorrido largo y detallado porque todo este tema forma un bloque: no se entiende un operador si no se entiende el tipo de dato con el que trabaja, y no se entiende el tipo de dato si no se entiende la variable que lo guarda.

La analogía que va a acompañarte durante toda la unidad es la de las cajas etiquetadas. Imagina que PHP te da un almacén vacío, enorme, con miles de cajas. Cada caja tiene una etiqueta donde escribes un nombre, y adentro puedes guardar una cosa: un número, un texto, una respuesta de sí o no. Cuando necesitas ese valor, no tienes que buscar caja por caja: solo dices el nombre de la etiqueta y PHP te entrega lo que hay dentro. Eso es exactamente una variable: un espacio de memoria con un nombre que tú eliges, donde guardas un valor para usarlo después. Y lo bonito es que puedes abrir la caja y cambiar lo que hay dentro cuando quieras. Si guardaste el número 5 y después necesitas que sea 10, abres la caja, sacas el 5, metes el 10, y listo. Por eso se llaman variables: porque su valor varía, cambia.

Pero ojo, porque este almacén tiene sus reglas. La etiqueta de cada caja tiene que empezar con el signo de pesos \$, como si fuera el precio de la caja. No puedes ponerle a una caja un nombre que empiece con número, ni usar espacios en el nombre. Y lo que guardes adentro puede ser de varios tipos: un texto (que en programación se llama string), un número entero (integer), un número con decimales (float), un valor de verdadero o falso (boolean) o la ausencia total de valor (null). PHP, a diferencia de otros lenguajes, no te obliga a decirle de

antemano qué tipo de cosa vas a guardar en cada caja: tú metes lo que sea y PHP lo detecta solo. Eso es cómodo, pero también es una trampa si no entiendes bien cómo funciona, porque PHP a veces convierte los tipos de datos por su cuenta y te puede dar resultados que no esperabas. En esta unidad vas a aprender exactamente cuándo pasa eso y cómo evitarlo.

También vas a conocer a los operadores, que son las herramientas con las que manipulas lo que hay en las cajas. Los operadores aritméticos te dejan sumar, restar, multiplicar y dividir. Los operadores de asignación te dejan guardar y actualizar valores de forma rápida. Los operadores de comparación te dejan preguntar si dos cosas son iguales o diferentes, y aquí vas a descubrir la diferencia entre `==` y `===`, que es uno de los temas que más dolores de cabeza les da a los principiantes y que vamos a desmenuzar con calma y con analogía incluida. Los operadores lógicos te dejan combinar condiciones con "y", "o" y "no". Y el operador de concatenación, que es un simple punto, te deja pegar textos para armar mensajes, recibos y páginas completas.

Al final de la unidad vas a tener las herramientas para escribir programas que guardan el nombre de un usuario, calculan el total de una compra, deciden si alguien puede entrar a un lugar según su edad, y arman textos combinando datos. Eso ya no es un "Hola mundo": eso es un programa de verdad, aunque sea pequeño.

Un consejo antes de empezar: esta unidad es larga a propósito. No la leas como si fuera una novela, de corrido y sin tocar la computadora. La forma correcta de estudiarla es con tu editor abierto y tu servidor corriendo: lees una sección, copias los ejemplos, los corres, los rompes a propósito para ver qué error sale, y luego los arreglas. El que aprende a programar es el que se equivoca muchas veces, no el que lee muchas páginas. Ten a la mano esta unidad como referencia y vuelve a ella cada vez que algo no te cuadre. Vamos a empezar.

Explicación detallada

En esta sección vamos a construir tu conocimiento paso a paso. Cada subsección es un ladrillo: no te brinques ninguna, porque la siguiente se apoya en la anterior. Al final de cada tema encontrarás advertencias de errores típicos y consejos prácticos que te van a ahorrar horas de frustración.

Variables en PHP: el signo \$ y los nombres válidos

Una variable en PHP siempre empieza con el signo de pesos \$. Siempre. Sin excepción. Es la marca que le dice a PHP: "oye, esto que sigue es el nombre de una caja, no una palabra cualquiera". Mira este ejemplo:

```
$nombre = "Carlos";
```

Aquí le estás diciendo a PHP: crea una caja con la etiqueta \$nombre y guarda adentro el texto "Carlos". A partir de ese momento, cada vez que escribas \$nombre en tu código, PHP va a buscar esa caja y te va a dar lo que tiene dentro.

¿Por qué el signo de pesos? La razón histórica es que PHP nació como un lenguaje para páginas web y sus creadores querían una forma rápida de distinguir las variables del texto normal dentro del HTML. Hoy en día es simplemente parte de la identidad del lenguaje. Piensa en el \$ como el asa de la caja: sin asa no la puedes agarrar, y sin \$ PHP no reconoce la variable.

Ahora, las reglas para los nombres de las variables. No puedes ponerle a tu caja cualquier etiqueta que se te ocurra; hay reglas, y si las rompes PHP te va a marcar error y tu página se va a quedar en blanco o te va a mostrar un mensaje de error. Las reglas son estas:

- El nombre siempre empieza con \$ seguido de una letra o un guion bajo _. Nunca puede empezar con un número. \$nombre es válido, \$_nombre es válido, \$1nombre no es válido.
- Después del primer carácter, puedes usar letras, números y guiones bajos. \$edad1 es válido, \$mi_variable es válido.
- No puedes usar espacios. Si quieres un nombre de dos palabras, usa el guion bajo o la notación camelCase (la primera palabra en minúsculas y las siguientes con mayúscula inicial). \$nombre_usuario es válido, \$nombreUsuario también es válido, \$nombre usuario no es válido.
- No puedes usar caracteres especiales como acentos, eñes en el nombre, guiones medios, puntos o signos de exclamación. \$año no es una buena idea (aunque en algunas versiones puede funcionar, te va a dar problemas), \$año-claro tampoco. Mantén los nombres en letras del alfabeto inglés sin acentos.
- PHP distingue entre mayúsculas y minúsculas en los nombres de variables. \$Nombre, \$nombre y \$NOMBRE son tres cajas completamente diferentes. Esto es una fuente clásica de errores: creas \$nombre con minúscula y luego intentas usar \$Nombre con mayúscula, y PHP te dice que la variable no existe o te muestra un aviso. Acostúmbrate a ser consistente.

Para crear una variable no necesitas declararla de antemano ni decirle qué tipo de dato va a guardar. En otros lenguajes como Java o C tienes que anunciar: "voy a crear una variable que guarda números enteros y se llama edad". En PHP no. Simplemente escribes \$edad = 25; y listo: PHP crea la caja, mete el 25 y además se da cuenta solito de que es un número entero.

Esto se llama tipado dinámico, y es una de las razones por las que PHP es tan amigable para empezar. Pero ojo: esta libertad tiene un precio, y lo vas a ver cuando hablemos de comparaciones.

También puedes cambiar el valor de una variable en cualquier momento. Mira:

```
$edad = 25;  
$edad = 26;
```

Primero la caja \$edad guarda el 25. Después abres la caja, sacas el 25 y metes el 26. El valor viejo se pierde para siempre, a menos que lo hayas guardado en otra variable antes. Esto es utilísimo: imagina una variable \$total que empiece en 0 y a la que le vas sumando el precio de cada producto que el cliente mete al carrito. Cada suma reemplaza el valor anterior con el nuevo total.

Otro detalle importante: puedes crear una variable y usarla en la misma línea o en líneas separadas, y puedes usar el valor de una variable para crear otra:

```
$precio = 100;  
$iva = 16;  
$total = $precio + $iva;
```

Aquí \$total guarda 116, que es el resultado de sumar lo que hay en las cajas \$precio y \$iva. Fíjate en que las cajas originales no se vacían: \$precio sigue guardando 100 y \$iva sigue guardando 16. Usar una variable no la gasta, es cómo leer la etiqueta de la caja sin abrirla.

Errores típicos con variables que vas a cometer (todos los cometemos) y cómo reconocerlos:

- Olvidar el \$ al usar la variable. Escribes \$nombre = "Carlos"; y luego escribes echo nombre; sin el \$. PHP va a pensar que nombre es una palabra suelta, no tu caja, y te va a mostrar un error o un aviso. La regla de oro: el \$ se usa tanto al crear la variable como cada vez que la usas después.
- Usar un nombre con espacio o con acento. Te va a dar un error de sintaxis que a veces es difícil de ubicar porque el mensaje de PHP no siempre apunta a la línea exacta con claridad.
- Confundir mayúsculas y minúsculas. \$total no es lo mismo que \$Total. Elige un estilo y mantenlo en todo tu código.

Consejo práctico: desde el primer día usa nombres descriptivos. \$x = 25; funciona, pero dentro de dos semanas no vas a saber qué era \$x. \$edad_del_cliente = 25; se explica solo. Vamos a profundizar en esto en la subsección de buenas prácticas.

Los tipos de datos: string, integer, float, boolean y null

Todo lo que guardas en una caja tiene un tipo. El tipo le dice a PHP qué clase de cosa es el valor y, por lo tanto, qué operaciones tienen sentido con él. No tiene sentido dividir un nombre entre dos, ni tiene sentido preguntar si "hola" es mayor que "adiós" como si fueran números. Los tipos son el contrato entre tú y PHP sobre lo que se puede hacer con cada valor.

En esta unidad nos vamos a concentrar en los cinco tipos básicos. PHP tiene más (arreglos, objetos, recursos), pero esos los verás en unidades posteriores. Por ahora, domina estos cinco y tendrás la base de todo.

El tipo string es texto. Cualquier secuencia de caracteres: letras, números, espacios, símbolos. En PHP un string siempre va entre comillas, simples o dobles. "Carlos", 'hola mundo', "12345" (ojo: entre comillas, esto es texto, no número), "me llamó Carlos y tengo 25 años". Todo eso son strings. Un string puede estar vacío: "" es un string válido que no contiene nada, como una caja con una nota en blanco adentro.

El tipo integer es un número entero, sin decimales: 25, -10, 0, 1000000. Sirve para contar cosas: edades, cantidades de productos, número de intentos. Los enteros se escriben sin comillas: \$edad = 25; Si le pones comillas, \$edad = "25";, entonces ya no es un número, es un texto que casualmente tiene dígitos. Esta diferencia parece tonta pero es importantísima, y vas a ver por qué en la parte de comparaciones.

El tipo float es un número con decimales: 19.99, -3.5, 0.75. En español estamos acostumbrados a escribir los decimales con coma (19,99), pero en programación el separador decimal es el punto. Si escribes \$precio = 19,99; PHP no lo va a entender como un número. Siempre punto: \$precio = 19.99; Los floats sirven para precios, promedios, medidas, cualquier cosa que necesite fracciones.

El tipo boolean solo tiene dos valores posibles: true (verdadero) o false (falso). Se escribe sin comillas y en minúsculas: \$es_mayor_de_edad = true; Los booleanos son la base de todas las decisiones en programación. ¿El usuario está registrado? true o false. ¿La contraseña es correcta? true o false. ¿El producto tiene existencias? true o false. Todo programa real está lleno de preguntas que se responden con sí o no, y los booleanos son cómo se guardan esas respuestas.

El tipo null representa la ausencia de valor. Una variable que vale null es una caja que existe pero está vacía, y además PHP sabe que está vacía a propósito. No es lo mismo que 0 (cero es un número), no es lo mismo que "" (string vacío es un texto vacío), no es lo mismo que false (false es una respuesta negativa). null es "aquí no hay nada todavía" o "esto no aplica". Ejemplo: \$segundo_nombre = null; significa que la persona no tiene segundo nombre

registrado. Es diferente de `$segundo_nombre = ""`; que podría significar que sí lo preguntaste y la persona dejó el campo en blanco. La diferencia es sutil pero en programas reales importa.

¿Cómo sabes qué tipo tiene una variable? PHP te da una función maravillosa para eso: `var_dump()`. Le pasas una variable y te muestra su tipo y su valor con todo detalle. La veremos a fondo en su propia subsección, pero adelanto un ejemplo para que veas cómo PHP detecta los tipos solo:

```
$nombre = "Carlos";  
$edad = 25;  
$precio = 19.99;  
$activo = true;  
$apellido = null;  
var_dump($nombre);  
var_dump($edad);  
var_dump($precio);  
var_dump($activo);  
var_dump($apellido);
```

Esto te mostraría algo como: `string(6) "Carlos", int(25), float(19.99), bool(true), NULL`. Fíjate en que tú nunca le dijiste a PHP los tipos; él los dedujo del valor que guardaste. Eso es el tipado dinámico en acción.

Una advertencia importante sobre el tipado dinámico: como PHP no te obliga a declarar el tipo, también te deja cambiar el tipo de una variable cuando quieras. Puedes hacer `$dato = 25`; y después `$dato = "hola"`; La misma caja primero guardó un número y luego un texto. PHP no se queja. Esto es flexible, pero también peligroso si no llevas control: imagina que una variable que todo tu programa supone que es número de repente guarda un texto porque en algún punto del código la reutilizaste para otra cosa. El programa no va a tronar de inmediato, pero los cálculos van a salir mal de formas raras. Consejo práctico: no reutilices una variable para guardar cosas de distinto tipo. Si necesitas otro dato, crea otra variable con otro nombre.

Comillas simples vs dobles e interpolación

En PHP puedes escribir strings con comillas simples (`'hola'`) o con comillas dobles (`"hola"`). Para un texto fijo, sin variables adentro, las dos funcionan igual. La diferencia aparece cuando metes una variable dentro del string. A esto se le llama interpolación: PHP "interpola", es decir, mete el valor de la variable dentro del texto.

Con comillas dobles, PHP sí reemplaza las variables por sus valores:

```
$nombre = "Carlos";  
echo "Hola, $nombre";
```

Esto muestra en pantalla: Hola, Carlos. PHP vio \$nombre dentro de las comillas dobles, fue a buscar la caja, sacó el valor "Carlos" y lo puso en el texto.

Con comillas simples, PHP no hace eso. Lo que escribes es literal:

```
$nombre = "Carlos";  
echo 'Hola, $nombre';
```

Esto muestra en pantalla: Hola, \$nombre. Tal cual, con el signo de pesos y todo. Las comillas simples le dicen a PHP: "no toques nada de lo que está aquí adentro, muéstralo tal cómo está".

La analogía: las comillas dobles son como un formato de carta con espacios en blanco que se rellenan automáticamente ("Estimado _____, le informamos..."), mientras que las comillas simples son como una fotocopia: sale exactamente lo que pusiste, sin rellenos.

¿Cuándo usar cada una? La regla práctica es sencilla:

- Usa comillas dobles cuando quieras meter variables dentro del texto. Es muy cómodo para armar mensajes: "Bienvenido, \$nombre, tienes \$puntos puntos".
- Usa comillas simples para textos fijos, sin variables. Además, las comillas simples son un poquito más rápidas porque PHP no tiene que revisar si hay variables adentro (en programas pequeños la diferencia es insignificante, pero es bueno saberlo).

Hay un detalle con las comillas dobles: si quieres mostrar el signo de pesos literal dentro de un string con comillas dobles, tienes que escaparlos con una diagonal invertida: "El precio es \\$100". Si no lo escapas, PHP va a pensar que \$100 es una variable. Con comillas simples no tienes ese problema: 'El precio es \$100' se muestra tal cual.

Otro detalle: ¿qué pasa si el texto mismo necesita llevar comillas? Por ejemplo, quieres mostrar: Él dijo "hola". Si tu string usa comillas dobles por fuera, las de adentro chocan: "Él dijo \"hola\" está mal. Tienes dos soluciones: escapar las comillas internas con diagonal invertida ("Él dijo \"hola\"") o usar el otro tipo de comilla por fuera (Él dijo 'hola'). La segunda opción es más limpia y legible.

Y un caso especial de la interpolación: cuando la variable va pegada a otro texto sin espacio, PHP puede confundirse sobre dónde termina el nombre de la variable. Por ejemplo: "Me gustan los \${fruta}s" no... mejor dicho, mira este caso:

```
$fruta = "manzana";  
echo "Me gustan las $frutas";
```

PHP va a buscar una variable llamada \$frutas (con s), que no existe, en lugar de entender \$fruta seguida de la letra s. Para estos casos existen las llaves: "Me gustan las {\$fruta}s". Las llaves le dicen a PHP exactamente dónde empieza y dónde termina el nombre de la variable. No es algo que uses todos los días, pero cuando lo necesites te salva.

Error típico: usar comillas simples esperando que se reemplacen las variables, y luego pasar media hora preguntándote por qué en pantalla sale el nombre de la variable en lugar de su valor. Si ves \$nombre impreso literalmente en tu página, lo primero que debes revisar es si usaste comillas simples.

Operadores aritméticos

Los operadores aritméticos son los que ya conoces de la escuela, solo que con algunos símbolos diferentes. Sirven para hacer cálculos con números (enteros o floats):

- Suma: + . \$total = 5 + 3; guarda 8.
- Resta: - . \$cambió = 100 - 35; guarda 65.
- Multiplicación: * (asterisco, no la x ni el punto). \$area = 5 * 4; guarda 20.
- División: / (diagonal, no el símbolo de división). \$promedio = 10 / 4; guarda 2.5.
- Módulo: % . Este es el menos conocido y uno de los más útiles. Te da el residuo de una división: 10 % 3 es 1, porque 10 entre 3 cabe 3 veces (9) y sobra 1. 10 % 2 es 0, porque 10 es divisible entre 2.
- Exponenciación: . 2 ³ es 2 elevado a la 3, o sea 8.

El módulo merece una explicación extra porque los principiantes no le ven el uso y después resulta que lo necesitan a cada rato. ¿Para qué sirve saber el residuo? Para muchas cosas: saber si un número es par o impar (\$número % 2 == 0 significa que es par), repartir turnos en ciclos, saber si ya pasaron bloques completos de algo. Ejemplo de la vida real: tienes 53 alumnos y quieres formar equipos de 5. 53 % 5 te dice cuántos alumnos sobran sin equipo (3). O quieres mostrar una lista y pintar de otro color una de cada 3 filas: usas el número de fila % 3.

PHP respeta el orden de las operaciones matemáticas que aprendiste en la escuela: primero multiplicaciones y divisiones, luego sumas y restas, y puedes usar paréntesis para forzar el orden. \$total = 100 + 50 * 2; da 200, no 300, porque primero se multiplica 50 * 2 y luego se

suma 100. Si quieres que se sume primero, usa paréntesis: `$total = (100 + 50) * 2`; da 300. Cuando tengas duda, pon paréntesis: no cuestan nada y hacen tu intención clarísima para quien lea tu código después (incluyéndote a ti mismo dentro de un mes).

Un detalle con la división: si divides entre cero, PHP no truena de forma elegante; en versiones recientes lanza un error. En la vida real, antes de dividir conviene asegurarse de que el divisor no sea cero, sobre todo si el divisor viene de un dato que escribió el usuario. Lo verás con más detalle cuando lleguemos a las condicionales en la Unidad 3, pero quédate con la idea.

También puedes combinar variables y números en las operaciones, y guardar el resultado en una variable nueva o en una existente:

```
$precio = 100;
$cantidad = 3;
$subtotal = $precio * $cantidad;
$iva = $subtotal * 0.16;
$total = $subtotal + $iva;
```

Este es el esqueleto de cualquier cálculo de tienda en línea. Fíjate en cómo cada variable tiene un nombre que dice exactamente qué guarda: eso no es casualidad, es buena práctica desde el día uno.

Error típico: usar la `x` para multiplicar o la coma para los decimales por costumbre de la escuela o del español. En PHP la multiplicación es `*` y los decimales llevan punto. `$precio = 19,99`; no hace lo que crees (de hecho, te va a dar un error de sintaxis o un comportamiento raro). Acostumbra tus dedos desde ahora.

Operadores de asignación

Ya conoces el operador de asignación básico: el signo `=` que guarda un valor en una variable. `$nombre = "Carlos"`; Pero PHP te da versiones abreviadas muy cómodas para los casos en los que quieres actualizar una variable usando su propio valor anterior. Son los operadores de asignación combinada:

- `$total += 50`; es lo mismo que `$total = $total + 50`; Suma 50 al valor que ya tenía `$total`.
- `$total -= 20`; es lo mismo que `$total = $total - 20`;
- `$total *= 2`; es lo mismo que `$total = $total * 2`;
- `$total /= 4`; es lo mismo que `$total = $total / 4`;
- `$texto .= " mundo"`; es lo mismo que `$texto = $texto . " mundo"`; Este es especial: funciona con strings y sirve para ir pegando texto. Lo verás en la subsección de concatenación.

La analogía: imagina que tienes una alcancía con \$100. \$total += 50; es como meter un billete de 50 en la alcancía sin sacarla, sin contarla de nuevo: simplemente agregas. Es más corto y más claro que escribir toda la operación completa.

Además existen los operadores de incremento y decremento, que son atajos para sumar o restar 1:

- `$contador++`; suma 1 a \$contador. Es lo mismo que `$contador += 1`; o `$contador = $contador + 1`;
- `$contador--`; resta 1 a \$contador.

Estos dos los vas a usar muchísimo en ciclos (loops), que verás en la Unidad 4, para contar vueltas: cada vez que el ciclo se repite, `$contador++` aumenta la cuenta en uno.

Un detalle curioso y tramposo: `++` puede ir antes o después de la variable (`++$contador` o `$contador++`), y hay una diferencia sutil en el valor que devuelve la expresión en ese momento. Para lo que vas a hacer en este curso, úsalo siempre después (`$contador++`;) como instrucción sola en su propia línea, y no tendrás ningún problema. La diferencia entre pre-incremento y post-incremento es un tema de examen en otros cursos, pero en la práctica casi nadie lo necesita.

Error típico gravísimo: confundir `=` (asignación) con `==` (comparación). `$edad = 18`; guarda 18 en la variable. `$edad == 18`; pregunta si lo que hay en \$edad es igual a 18. Si escribes `if ($edad = 18)` cuando querías preguntar, no estás preguntando nada: estás guardando 18 en \$edad, y además esa asignación siempre se considera verdadera, así que tu condición siempre se cumple. Es el error número uno de los principiantes en todos los lenguajes, y lo vamos a machacar en la subsección de comparaciones hasta que te salga natural distinguirlos.

Operadores de comparación: `==` vs `===` y el tipado débil

Los operadores de comparación sirven para hacer preguntas sobre los valores, y siempre responden con un boolean: true o false. Son la base de las decisiones en tu programa (que verás a fondo en la Unidad 3 con las condicionales), así que vale la pena entenderlos muy bien desde ahora.

Estos son los operadores de comparación:

- `==` : igual que. Pregunta si dos valores son iguales.
- `===` : idéntico que. Pregunta si dos valores son iguales Y del mismo tipo.
- `!=` : diferente que. Lo contrario de `==`.

- `!==` : no idéntico que. Lo contrario de `===`.
- `>` : mayor que.
- `<` : menor que.
- `>=` : mayor o igual que.
- `<=` : menor o igual que.

Los de mayor y menor son intuitivos: `5 > 3` es `true`, `5 < 3` es `false`. El tema interesante, el que separa a los principiantes de los programadores que ya se quemaron una vez, es la diferencia entre `==` y `===`.

PHP es un lenguaje de tipado débil. Eso significa que cuando comparas con `==`, PHP intenta ser "amable" y convierte los tipos automáticamente para que la comparación tenga sentido. Si comparas el número 5 con el texto "5", PHP piensa: "bueno, el texto '5' se puede leer como el número 5, así que son iguales". Y te responde `true`. Mira:

```
var_dump(5 == "5");
```

Esto muestra `bool(true)`. El número 5 y el string "5" son considerados iguales por `==`, aunque uno es número y el otro es texto.

Con `===`, PHP es estricto: no convierte nada. Para que `===` diga `true`, los valores tienen que ser iguales y del mismo tipo:

```
var_dump(5 === "5");
```

Esto muestra `bool(false)`. Son valores "iguales" a simple vista, pero de tipos diferentes, así que no son idénticos.

La analogía: imagina que `==` es como un portero buena onda que deja pasar a alguien si "se parece" al de la foto, aunque la credencial sea de otro color. `===` es como un portero estricto que revisa foto, nombre, tipo de credencial y holograma: si algo no coincide exactamente, no pasa. En programación, casi siempre quieres al portero estricto, porque las sorpresas del portero buena onda son las que generan los errores más difíciles de encontrar.

¿Por qué es peligroso el tipado débil? Porque PHP convierte tipos en situaciones que no esperas. Algunos ejemplos clásicos que han causado dolores de cabeza reales:

- `0 == "hola"` puede darte `true` en versiones viejas de PHP, porque PHP convertía "hola" a número y como no empezaba con dígitos lo convertía a 0. (En PHP 8 esto se corrigió y ahora da `false`, pero el susto histórico queda como lección.)

- "10" == 10 da true. Si estás comparando un dato que vino de un formulario (que siempre llega como string) con un número, == te puede dar un true engañoso.
- false == 0 da true, y null == false también da true con ==. Con ===, cada uno solo es idéntico a sí mismo.

La regla práctica que te va a salvar la vida: usa === y !== por defecto, siempre que puedas. Solo usa == cuando deliberadamente quieras permitir la conversión de tipos y sepas exactamente qué estás haciendo. Los programadores con experiencia en PHP escriben === casi siempre; no es casualidad, es cicatriz de batallas pasadas.

Otro punto: cuando compares strings con == o ===, la comparación es exacta, carácter por carácter, y distingue mayúsculas de minúsculas: "Carlos" == "carlos" es false. Si necesitas comparar sin importar mayúsculas, hay funciones para eso que verás en la unidad de strings, pero por ahora quédate con que la comparación de texto es literal.

Error típico número uno (lo repetimos porque es el más común de todos): escribir = en lugar de == dentro de una condición. if (\$edad = 18) no pregunta nada: asigna 18 a \$edad y la condición siempre resulta verdadera. Tu programa va a comportarse como si todos tuvieran 18 años, y te vas a volver loco buscando el error porque "el código se ve bien". Acostúmbrate a leer tus condiciones en voz alta: "si edad ES IGUAL A 18" lleva doble signo. Algunos programadores escriben las comparaciones al revés, if (18 == \$edad), para que si se les olvida un signo el código truene de inmediato en lugar de fallar en silencio. Es un truco válido, aunque no obligatorio.

Error típico número dos: comparar un string con un número usando == y sorprenderse del resultado. Si un dato viene de un formulario HTML, siempre llega como string, aunque el usuario haya escrito dígitos. "25" == 25 es true, pero "25" === 25 es false. Cuando trabajes con formularios (Unidad 6), ten esto siempre presente.

Operadores lógicos

Los operadores lógicos te dejan combinar varias preguntas en una sola condición más grande. Son tres:

- && o and: Y lógico. El resultado es true solo si las dos condiciones son verdaderas. Si una de las dos es falsa, todo es falso.
- || o or: O lógico. El resultado es true si al menos una de las dos condiciones es verdadera. Solo es falso si las dos son falsas.
- ! : NO lógico. Invierte el valor: !true es false, !false es true.

En la práctica se usan && y || (los símbolos), no las palabras and y or, porque las palabras tienen una prioridad diferente que puede causar sorpresas. Quédate con && y ||.

La analogía de la vida real: piensa en los requisitos para entrar a un antro. El cadenero te dice: "puedes pasar si tienes identificación oficial Y tienes 18 años o más". Son dos condiciones unidas por un Y: si te falta una, no entras. En código:

```
$tiene_identificacion = true;
$edad = 20;
$puede_entrar = $tiene_identificacion && $edad >= 18;
```

\$puede_entrar va a ser true solo si las dos cosas se cumplen. Si \$edad fuera 16, la segunda condición es falsa y el && hace que todo sea falso: no entra.

Con el O es como una promoción: "tienes descuento si eres estudiante O si es tu cumpleaños". Basta con que una de las dos sea verdadera:

```
$es_estudiante = false;
$es_cumpleanos = true;
$tiene_descuento = $es_estudiante || $es_cumpleanos;
```

Aquí \$tiene_descuento es true porque al menos una condición se cumple.

El ! es como decir "no": "puedes pasar si NO estás en la lista de vetados":

```
$esta_vetado = false;
$puede_pasar = !$esta_vetado;
```

\$puede_pasar es true porque \$esta_vetado es false y el ! lo invierte.

Puedes combinar varios operadores en condiciones más largas, y usar paréntesis para dejar claro el orden. Ejemplo completo: "entra si (es mayor de edad y trae identificación) o si viene con un adulto responsable":

```
$puede_entrar = ($edad >= 18 && $tiene_identificacion) || $viene_con_adulto;
```

PHP evalúa primero lo que está entre paréntesis y luego el ||. Sin paréntesis también funcionaría porque && tiene prioridad sobre ||, pero los paréntesis hacen tu intención obvia para cualquiera que lea el código. Recuerda la regla de oro de esta unidad: cuando dudes, pon paréntesis.

Un detalle útil: PHP usa evaluación de cortocircuito. Esto significa que en un `&&`, si la primera condición es falsa, PHP ni siquiera revisa la segunda, porque ya sabe que el resultado total va a ser falso. Y en un `||`, si la primera es verdadera, se salta la segunda. En tus programas esto casi siempre es una ventaja (ahorra trabajo), y a veces se usa a propósito para evitar errores, como revisar primero que una variable no sea null antes de usarla.

Error típico: confundir `&&` con `||`. "Quiero que entre si tiene identificación Y es mayor de edad" se escribe con `&&`. Si escribes `||` por accidente, entraría cualquiera que cumpla una sola de las dos, incluyendo menores con identificación falsa... digo, con identificación pero menores de edad. Lee tu condición en voz alta en español y verifica que el conector lógico sea el correcto.

Concatenación con el punto

Concatenar significa pegar strings uno tras otro para formar un texto más largo. En PHP el operador de concatenación es el punto `.` (sí, un simple punto). Es uno de los operadores que más vas a usar, porque armar mensajes, páginas HTML y recibos es pegar textos todo el día.

```
$nombre = "Carlos";  
$saludo = "Hola, " . $nombre . ". Bienvenido.";  
echo $saludo;
```

Esto muestra: Hola, Carlos. Bienvenido. El punto pega "Hola, " con el valor de `$nombre` y con ". Bienvenido.", formando un solo string que se guarda en `$saludo`.

Fíjate en los espacios: el punto no agrega espacios automáticamente. Si escribes "Hola," . `$nombre` sin espacio después de la coma, sale "Hola,Carlos". Tú eres responsable de los espacios, así que fíjate bien dónde los pones: normalmente van dentro de los strings, al final o al principio.

También puedes concatenar el resultado de operaciones o de funciones, y puedes ir acumulando texto en una variable con el operador `.=` que ya viste en asignaciones:

```
$recibo = "=== RECIBO ===";  
$recibo .= "Cliente: Carlos";  
$recibo .= "Total: $116";  
echo $recibo;
```

Cada `.=` pega el texto nuevo al final de lo que ya había en `$recibo`. Esta técnica de ir acumulando es exactamente como vas a armar páginas HTML dinámicas más adelante: empiezas con una variable vacía y le vas pegando pedazos.

Nota: también puedes lograr algo parecido con la interpolación en comillas dobles ("Hola, \$nombre"), y para mensajes simples la interpolación es más legible. La concatenación con punto brilla cuando armas textos largos por partes o cuando mezclas comillas simples con variables. Con el tiempo desarrollarás tu propio estilo; lo importante es que domines las dos formas.

Error típico: usar + para pegar strings por costumbre de otros lenguajes (en JavaScript, por ejemplo, + sí concatena). En PHP, "hola" + "mundo" no pega los textos: PHP intenta convertir los strings a números y sumarlos, y te da 0 con un aviso. Para pegar textos en PHP siempre usa el punto.

var_dump(): tu lupa para depurar

var_dump() es una función que recibe una variable (o un valor directo) y te muestra información completa sobre ella: el tipo de dato y el valor. Es la herramienta de depuración más simple y más usada en PHP, y en esta unidad va a ser tu mejor amiga.

```
$edad = 25;  
var_dump($edad);
```

En pantalla verás: int(25). Te dice: es un integer y vale 25.

Con un string te dice además la longitud: var_dump("Carlos"); muestra string(6) "Carlos". El 6 es el número de caracteres. Esto es utilísimo para detectar espacios invisibles: si un string que debería decir "Carlos" te muestra string(7), sabes que hay un espacio de más escondido por ahí.

Puedes pasarle varias variables separadas por comas y te las muestra todas en orden:

```
var_dump($nombre, $edad, $activo);
```

¿Para qué sirve en la vida real? Para responder la pregunta que te vas a hacer mil veces: "¿qué demonios tiene adentro esta variable en este punto del programa?". Cuando un cálculo sale mal, cuando una comparación no se comporta como esperabas, cuando un formulario te manda datos raros, pones un var_dump() antes y después del punto sospechoso y comparas. Es como abrir la caja y mirar adentro directamente, sin suposiciones.

Consejo práctico: cuando estés aprendiendo, abusa de var_dump(). Ponlo por todos lados. No hay nada de malo en un código de práctica lleno de var_dump(); ya los quitarás cuando el programa funcione. Los programadores profesionales también lo usan a diario para depurar, así que no es una herramienta "de novatos": es una herramienta de trabajo real.

Combinado con la etiqueta HTML pre, la salida se ve más ordenada en el navegador, pero eso es un detalle de presentación que verás cuando mezclemos PHP con HTML. Por ahora, con ver el tipo y el valor te basta.

Errores comunes: la galería de los horrores

Vamos a reunir en un solo lugar los errores que más cometen los principiantes con variables, tipos y operadores, con su síntoma y su cura. Guárdate está lista; vas a volver a ella.

1. Usar `=` en vez de `==` en una comparación. Síntoma: una condición que siempre se cumple (o un comportamiento ilógico) y un código que "se ve bien". Cura: relee tus condiciones buscando el doble signo; lee en voz alta "es igual a".
2. Olvidar el `$` al usar una variable. Síntoma: PHP muestra un aviso de variable no definida o un error, o imprime el nombre literal. Cura: el `$` va siempre, al crear y al usar.
3. Comparar string con número usando `==` y esperar `===`. Síntoma: `"25" == 25` da true y tu lógica toma un camino inesperado. Cura: usa `===` por defecto.
4. Nombres de variables inconsistentes en mayúsculas. Síntoma: aviso de variable no definida aunque "sí la definiste". Cura: `$total` y `$Total` son diferentes; elige minúsculas y sé consistente.
5. Usar `+` para concatenar strings. Síntoma: el resultado es 0 o un número raro en lugar del texto pegado. Cura: para pegar textos usa el punto.
6. Coma en lugar de punto para decimales. Síntoma: error de sintaxis o valor incorrecto. Cura: en PHP los decimales siempre llevan punto: 19.99.
7. Comillas simples esperando interpolación. Síntoma: en pantalla aparece `$nombre` en lugar del valor. Cura: para interpolar usa comillas dobles.
8. Dividir entre cero. Síntoma: error en tiempo de ejecución. Cura: verifica que el divisor no sea cero antes de dividir, sobre todo si viene del usuario.
9. Reutilizar una variable para otro tipo de dato. Síntoma: cálculos que de repente salen mal en partes lejanas del código. Cura: una variable, un propósito, un tipo; si necesitas otro dato, crea otra variable.
10. Olvidar los espacios al concatenar. Síntoma: textos pegados como "Hola,Carlos". Cura: incluye los espacios dentro de los strings.

Buenas prácticas: nombres descriptivos y constantes

El código se escribe una vez y se lee muchas veces. Esa es la frase que debes tatuarte en la mente. Tu código lo vas a leer tú dentro de tres meses (y no te vas a acordar de nada), lo van a leer tus compañeros, tu jefe, o la persona que te contrate. Escribir código claro no es amabilidad: es supervivencia.

La buena práctica número uno de esta unidad: usa nombres descriptivos para tus variables. Compara:

```
$a = 100;  
$b = 3;  
$c = $a * $b;
```

contra:

```
$precio_unitario = 100;  
$cantidad = 3;  
$subtotal = $precio_unitario * $cantidad;
```

El primer código funciona igual, pero es un acertijo. El segundo se explica solo. La regla es: si tienes que poner un comentario para explicar qué guarda una variable, el nombre está mal; cámbialo por uno que no necesite explicación.

Sobre el idioma de los nombres: puedes usar español o inglés, pero sé consistente en todo tu proyecto. Si empiezas con \$nombre, \$edad, \$precio, no cambies a \$name, \$age, \$price a la mitad. En la industria se usa mucho el inglés porque es el idioma de la documentación y de los equipos internacionales, pero para tus proyectos de aprendizaje el español está perfecto y es más claro para ti. Lo que no se vale es mezclar: \$nombre_del_user es un crimen contra la legibilidad.

Otras convenciones de nombres: usa minúsculas con guiones bajos (\$nombre_completo) o camelCase (\$nombreCompleto), pero no mezcles los dos estilos en el mismo proyecto. Evita nombres de una sola letra excepto en contadores de ciclos (\$i es aceptable por tradición en loops). Y nunca uses nombres que se parezcan demasiado entre sí (\$total1, \$total2, \$total_final, \$totalFinal): es una invitación a confundirlos.

La buena práctica número dos: usa constantes para los valores que no deben cambiar. Si el IVA es 16% y lo escribes como 0.16 regado en veinte lugares de tu código, el día que cambie a 17% tendrás que cazar esos veinte lugares uno por uno, y seguro se te olvida alguno. En cambio, si defines una constante una sola vez y la usas en todas partes, el cambio se hace en un solo lugar.

En PHP hay dos formas de definir constantes:

```
define("IVA", 0.16);  
const IVA = 0.16;
```

Las dos crean una constante llamada IVA con valor 0.16. Las constantes, por convención, se escriben en MAYÚSCULAS, y ojo: no llevan \$ al usarlas. Se usan así:

```
$subtotal = 100;  
$total = $subtotal + ($subtotal * IVA);
```

define() es una función y puede usarse en cualquier parte del código; const se usa normalmente al inicio de un archivo o dentro de clases (las clases las verás mucho después). Para tu nivel, con define() tienes de sobra. La diferencia técnica entre ambas no te debe preocupar por ahora.

¿Por qué las constantes no llevan \$? Porque no son cajas variables: son etiquetas fijas, valores grabados en piedra. PHP las trata de forma distinta y por eso no necesitan el signo. Si intentas cambiar el valor de una constante, PHP te va a marcar error, y eso es justamente lo que quieres: una protección contra cambios accidentales.

¿Qué cosas merecen ser constantes? Los impuestos y tasas, los límites (intentos máximos, edad mínima), los mensajes que se repiten, las rutas de archivos, las claves de configuración. La regla: si es un valor fijo que usas en más de un lugar, hazlo constante.

Y una última buena práctica que engloba todo: inicializa tus variables. No uses una variable sin haberle dado un valor antes. PHP te dejará hacerlo con un aviso, pero el valor será null y tu lógica puede comportarse de formas extrañas. Acostúmbrate a darles un valor inicial sensato en cuanto las creas: \$total = 0; \$nombre = ""; \$encontrado = false;. Es como etiquetar la caja y meterle algo adentro desde el principio, aunque sea un valor "vacío" intencional.

Ejemplos paso a paso

Aquí vamos a ver ejemplos completos, listos para copiar, pegar y correr en tu servidor. Cada ejemplo viene explicado línea por línea. Te recomiendo que no solo los leas: escríbelos tú mismo en tu editor, córrelos, y luego modifícalos para ver qué pasa. La programación se aprende con los dedos, no solo con los ojos.

Ejemplo 1: declarar y mostrar variables

Este es el ejemplo más básico de todos: crear cajas, guardar valores y mostrarlos en pantalla.

```
$nombre = "Mariana";  
$edad = 28;  
$ciudad = "Guadalajara";  
echo "Me llamo $nombre, tengo $edad años y vivo en $ciudad.";
```

Línea por línea:

- `$nombre = "Mariana";` Crea la variable `$nombre` y guarda el string "Mariana". Fíjate en el `$` al inicio, el `=` para asignar, las comillas dobles porque es texto, y el punto y coma al final de la instrucción (en PHP cada instrucción termina con `;`).
- `$edad = 28;` Crea `$edad` y guarda el integer 28. Sin comillas, porque es un número.
- `$ciudad = "Guadalajara";` Crea `$ciudad` con el string "Guadalajara".
- `echo "Me llamó $nombre, tengo $edad años y vivo en $ciudad.";` La instrucción `echo` muestra texto en pantalla. Como el string usa comillas dobles, PHP interpola las tres variables y muestra: Me llamó Mariana, tengo 28 años y vivo en Guadalajara.

Prueba a cambiar los valores y vuelve a correr el script. Cambia las comillas dobles por simples y observa cómo ahora se muestran los nombres de las variables en lugar de sus valores. Romper las cosas a propósito es una de las mejores formas de aprender.

Ejemplo 2: conocer los tipos con `var_dump()`

Aquí usamos `var_dump()` para ver con nuestros propios ojos qué tipo le asigna PHP a cada valor.

```
$texto = "Hola mundo";  
$entero = 42;  
$decimal = 3.14;  
$verdadero = true;  
$vacio = null;  
var_dump($texto);  
var_dump($entero);  
var_dump($decimal);  
var_dump($verdadero);  
var_dump($vacio);
```

Línea por línea:

- Las primeras cinco líneas crean una variable de cada tipo básico: string, integer, float, boolean y null. Nota que true se escribe sin comillas y en minúsculas, y null también.
- `var_dump($texto);` muestra `string(10) "Hola mundo"`. El 10 es la longitud del texto.
- `var_dump($entero);` muestra `int(42)`.
- `var_dump($decimal);` muestra `float(3.14)`.
- `var_dump($verdadero);` muestra `bool(true)`.
- `var_dump($vacío);` muestra `NULL`.

Este ejemplo es tu plantilla de laboratorio: cada vez que tengas duda sobre qué tipo tiene una variable en tu código, copia estas líneas, adapta los nombres y corre el script. Ver el tipo con tus propios ojos elimina las suposiciones, y las suposiciones son la madre de los errores.

Ejemplo 3: interpolación con comillas dobles

Un ejemplo dedicado a la diferencia entre comillas simples y dobles, porque es un error que vas a cometer al menos una vez.

```
$producto = "tacos al pastor";  
$precio = 15;  
echo "Hoy hay $producto a $$precio la orden."  
echo 'Hoy hay $producto a $$precio la orden.'
```

Línea por línea:

- `$producto = "tacos al pastor";` guarda un string.
- `$precio = 15;` guarda un integer.

- `echo "Hoy hay $producto a $$precio la orden.";` Con comillas dobles, PHP reemplaza `$producto` por "tacos al pastor" y `$precio` por 15. Pero ojo con `$$precio`: el primer `$` está escapado con la diagonal invertida (`\$`), así que se muestra como un signo de pesos literal, y el segundo `$precio` se interpola. El resultado es: Hoy hay tacos al pastor a \$15 la orden.
- `echo 'Hoy hay $producto a $$precio la orden.';` Con comillas simples no hay interpolación: se muestra el texto tal cual, incluyendo los nombres de las variables: Hoy hay \$producto a \$\$precio la orden.

Moraleja del ejemplo: si en tu pantalla aparecen los nombres de tus variables en lugar de sus valores, revisa las comillas. El 99% de las veces ese es el problema.

Ejemplo 4: operaciones aritméticas en una compra

Un ejemplo realista: calcular el total de una compra con subtotal, IVA y descuento.

```
$precio_unitario = 120;
$cantidad = 4;
$subtotal = $precio_unitario * $cantidad;
$iva = $subtotal * 0.16;
$descuento = 50;
$total = $subtotal + $iva - $descuento;
echo "Subtotal: $$subtotal";
echo "IVA: $$iva";
echo "Descuento: $$descuento";
echo "Total a pagar: $$total";
```

Línea por línea:

- `$precio_unitario = 120;` y `$cantidad = 4;` son los datos de entrada. En un programa real vendrían de un formulario o de una base de datos; aquí los escribimos directo para practicar.
- `$subtotal = $precio_unitario * $cantidad;` multiplica 120 por 4 y guarda 480. Usar variables en lugar de números sueltos hace que el cálculo se entienda solo.
- `$iva = $subtotal * 0.16;` calcula el 16% de 480, que es 76.8. Nota el punto decimal, no coma.
- `$descuento = 50;` un descuento fijo de 50.

- $\$total = \$subtotal + \$iva - \$descuento$; suma y resta en una sola línea: $480 + 76.8 - 50 = 506.8$. PHP respeta el orden: aquí solo hay sumas y restas, que se evalúan de izquierda a derecha.
- Las cuatro líneas de echo muestran cada concepto. Cada una usa interpolación con el $\$$ escapado para mostrar el signo de pesos.

El resultado en pantalla sería: Subtotal: \$480, IVA: \$76.8, Descuento: \$50, Total a pagar: \$506.8. Este esqueleto (datos de entrada, cálculos intermedios con nombres claros, resultado final) es el patrón que vas a repetir en cientos de programas.

Ejemplo 5: comparaciones estrictas vs débiles

El ejemplo que te va a grabar a fuego la diferencia entre `==` y `===`.

```
$edad_texto = "18";
$edad_numero = 18;
var_dump($edad_texto == $edad_numero);
var_dump($edad_texto === $edad_numero);
var_dump($edad_numero === 18);
var_dump(0 == false);
var_dump(0 === false);
```

Línea por línea:

- `$edad_texto = "18";` guarda el string "18" (con comillas: es texto). Esto simula un dato que viene de un formulario HTML, que siempre llega como string.
- `$edad_numero = 18;` guarda el integer 18.
- `var_dump($edad_texto == $edad_numero);` muestra `bool(true)`. Con `==`, PHP convierte el string "18" a número y los considera iguales. El portero buena onda deja pasar.
- `var_dump($edad_texto === $edad_numero);` muestra `bool(false)`. Con `===`, los tipos no coinciden (string vs integer), así que no son idénticos. El portero estricto no deja pasar.
- `var_dump($edad_numero === 18);` muestra `bool(true)`. Mismo valor y mismo tipo: idénticos.
- `var_dump(0 == false);` muestra `bool(true)`. Con `==`, el cero y el falso se consideran iguales. Esta es una de esas conversiones automáticas que causan errores sutiles.
- `var_dump(0 === false);` muestra `bool(false)`. Con `===`, integer y boolean son tipos distintos: no idénticos.

La lección práctica: cuando compares datos que vienen del usuario (formularios) con valores de tu programa, usa `===` para no llevarte sorpresas, o convierte explícitamente el tipo antes de comparar. Este ejemplo vale oro cuando llegues a la unidad de formularios.

Ejemplo 6: operadores lógicos, ¿puede entrar al antro?

Un ejemplo divertido y realista que combina operadores lógicos y de comparación: el cadenero digital.

```
$edad = 22;
$tiene_identificacion = true;
$esta_en_lista_vetados = false;
$es_mayor = $edad >= 18;
$puede_entrar = $es_mayor && $tiene_identificacion && !$esta_en_lista_vetados;
var_dump($puede_entrar);
$edad = 17;
$tiene_identificacion = true;
$esta_en_lista_vetados = false;
$es_mayor = $edad >= 18;
$puede_entrar = $es_mayor && $tiene_identificacion && !$esta_en_lista_vetados;
var_dump($puede_entrar);
```

Línea por línea (primera mitad):

- `$edad = 22; $tiene_identificacion = true; $esta_en_lista_vetados = false;` son los datos de la persona: 22 años, trae identificación, no está vetada.
- `$es_mayor = $edad >= 18;` guarda en `$es_mayor` el resultado de la comparación `22 >= 18`, que es `true`. Fíjate: una comparación produce un boolean, y ese boolean se puede guardar en una variable como cualquier otro valor. Esta técnica de guardar condiciones en variables con nombres claros hace el código mucho más legible.
- `$puede_entrar = $es_mayor && $tiene_identificacion && !$esta_en_lista_vetados;` combina las tres condiciones: tiene que ser mayor (`true`) Y tener identificación (`true`) Y NO estar vetada (`!$esta_en_lista_vetados` invierte `false` a `true`). `true && true && true = true`.
- `var_dump($puede_entrar);` muestra `bool(true)`: sí puede entrar.

Segunda mitad: cambiamos la edad a 17 y repetimos. Ahora `$es_mayor` es `false`, así que `false && true && true = false`. `var_dump` muestra `bool(false)`: no puede entrar, aunque traiga identificación y no esté vetada. Con `&&` basta que una condición falle para que todo falle.

Prueba a cambiar los valores: pon `$esta_en_lista_vetados = true`; con edad 22 y observa cómo el `!` invierte el `true` a `false` y bloquea la entrada. Jugar con estos valores es la mejor forma de interiorizar cómo funcionan `&&`, `||` y `!`.

Ejemplo 7: concatenación para armar un recibo

Un ejemplo que junta concatenación, el operador `.` y la interpolación para armar un texto largo por partes, como harías con un recibo o con un correo.

```
$cliente = "Mariana Lopez";
$producto = "Curso de PHP";
$precio = 299;
$recibo = "==== RECIBO =====";
$recibo .= "Cliente: " . $cliente;
$recibo .= "Producto: " . $producto;
$recibo .= "Precio: $" . $precio;
$recibo .= "Gracias por tu compra, $cliente.";
$recibo .= "===== ";
echo $recibo;
```

Línea por línea:

- Las primeras tres líneas guardan los datos: nombre del cliente, producto y precio.
- `$recibo = "==== RECIBO ====="`; inicia la variable `$recibo` con el encabezado. A partir de aquí vamos a ir acumulando.
- `$recibo .= "Cliente: " . $cliente`; pega al final de `$recibo` el texto "Cliente: Mariana Lopez". El `.` toma lo que ya había y le agrega lo nuevo.
- Las siguientes líneas agregan el producto, el precio y el mensaje de agradecimiento. Nota que en "Gracias por tu compra, \$cliente." usamos interpolación dentro de un string con comillas dobles, mezclada con concatenación: las dos técnicas conviven sin problema.
- `echo $recibo`; muestra todo el recibo armado.

El resultado es un bloque de texto con toda la información. Esta técnica de acumular en una variable es exactamente la que usarás en unidades posteriores para armar páginas HTML completas: empiezas con `$pagina = ""`; y le vas pegando el encabezado, el contenido y el pie de página. Si dominas este ejemplo, ya tienes la mitad del camino andado para generar páginas dinámicas.

Como ejercicio extra de este ejemplo: modifica el código para que el recibo incluya el IVA del 16% y el total, usando lo que aprendiste en el Ejemplo 4. Si puedes combinar ambos ejemplos sin mirar, ya dominas variables y operadores aritméticos.

Práctica

Llegó la hora de ensuciarte las manos. Aquí tienes cinco ejercicios ordenados de más fácil a más retador. La regla de oro: intenta cada ejercicio por tu cuenta antes de mirar la solución. Si te atorras, vuelve a leer la sección de explicación correspondiente, no saltes directo a la solución. Equivocarte aquí, en privado, es gratis; equivocarte en un trabajo ya es otra historia.

Ejercicio 1

Declara tres variables: \$nombre con tu nombre, \$edad con tu edad y \$ciudad con tu ciudad. Luego muestra en pantalla una frase como "Me llamó [nombre], tengo [edad] años y vivo en [ciudad]." usando interpolación con comillas dobles. Después haz una segunda versión de la frase usando concatenación con el punto y comillas simples donde sea posible.

Solución 1

```
$nombre = "Carlos";
$edad = 30;
$ciudad = "Monterrey";
echo "Me llamo $nombre, tengo $edad años y vivo en $ciudad.";
echo 'Me llamo ' . $nombre . ', tengo ' . $edad . ' años y vivo en ' . $ciudad .
'.';
```

Explicación paso a paso:

- Las tres primeras líneas crean las variables con tus datos. El nombre y la ciudad van entre comillas dobles porque son strings; la edad va sin comillas porque es un integer. Puedes usar los valores que quieras; lo importante es respetar los tipos.
- La primera versión del echo usa comillas dobles e interpolación: PHP reemplaza \$nombre, \$edad y \$ciudad por sus valores automáticamente. Es la forma más corta y legible para este caso.
- La segunda versión usa comillas simples y el operador punto para pegar cada pedazo. Fíjate en los espacios: 'Me llamó ' termina con un espacio, y ', tengo ' empieza y termina con espacio. Si olvidas esos espacios, las palabras salen pegadas a los valores. Esta

versión es más larga, pero te obliga a practicar la concatenación, que necesitarás para textos más complejos.

- Las dos líneas muestran exactamente la misma frase en pantalla. Compruébalo corriendo el script: si las dos salidas son idénticas, lo hiciste bien.

Ejercicio 2

Crea un script que calcule el salario semanal de un trabajador. Declara `$horas_trabajadas` con valor 40, `$pago_por_hora` con valor 85.50, calcula `$salario_bruto` multiplicando ambas, calcula el impuesto del 12% sobre el salario bruto en `$impuestos`, y el `$salario_netto` restando los impuestos al bruto. Muestra los tres valores con `var_dump()` para verificar sus tipos, y luego muestra una frase legible con el salario neto usando interpolación.

Solución 2

```
$horas_trabajadas = 40;
$pago_por_hora = 85.50;
$salario_bruto = $horas_trabajadas * $pago_por_hora;
$impuestos = $salario_bruto * 0.12;
$salario_netto = $salario_bruto - $impuestos;
var_dump($salario_bruto);
var_dump($impuestos);
var_dump($salario_netto);
echo "El salario neto semanal es de $$salario_netto.";
```

Explicación paso a paso:

- `$horas_trabajadas = 40;` guarda un integer y `$pago_por_hora = 85.50;` guarda un float. Nota el punto decimal en 85.50, no coma.
- `$salario_bruto = $horas_trabajadas * $pago_por_hora;` multiplica 40 por 85.50 y guarda 3420. Cuando multiplicas un integer por un float, PHP te devuelve un float. Por eso `var_dump($salario_bruto)` muestra `float(3420)`.
- `$impuestos = $salario_bruto * 0.12;` calcula el 12%: $3420 * 0.12 = 410.4$.
- `$salario_netto = $salario_bruto - $impuestos;` resta: $3420 - 410.4 = 3009.6$.
- Los tres `var_dump()` te confirman los tipos: los tres son float, porque en cuanto un float entra a la operación, el resultado es float. Esta es una buena costumbre: verificar con `var_dump()` que tus cálculos producen el tipo que esperas.

- El echo final usa interpolación con el signo de pesos escapado (\\$) para mostrar: El salario neto semanal es de \$3009.6.
- Como reto extra: cambia las horas a 48 y el pagó a 100, corre de nuevo y verifica que todo se recalcule solo. Esa es la magia de usar variables en lugar de números escritos a mano: cambias los datos de entrada y todo lo demás se ajusta.

Ejercicio 3

Escribe un script que reciba la edad de una persona en \$edad y determine si puede votar (18 años o más) usando operadores de comparación. Guarda el resultado en \$puede_votar. Luego agrega una segunda condición: también necesita tener credencial vigente (\$tiene_credencial). Calcula \$puede_votar_final combinando ambas con un operador lógico. Prueba tu script con al menos tres combinaciones de valores diferentes usando var_dump() para ver los resultados, y explica con tus palabras por qué cada combinación da el resultado que da.

Solución 3

```
$edad = 20;
$tiene_credencial = true;
$puede_votar = $edad >= 18;
$puede_votar_final = $puede_votar && $tiene_credencial;
var_dump($puede_votar_final);
$edad = 16;
$tiene_credencial = true;
$puede_votar = $edad >= 18;
$puede_votar_final = $puede_votar && $tiene_credencial;
var_dump($puede_votar_final);
$edad = 25;
$tiene_credencial = false;
$puede_votar = $edad >= 18;
$puede_votar_final = $puede_votar && $tiene_credencial;
var_dump($puede_votar_final);
```

Explicación paso a paso:

- Primera combinación: edad 20, con credencial. \$puede_votar = 20 >= 18, que es true. \$puede_votar_final = true && true, que es true. var_dump muestra bool(true): sí puede votar.

- Segunda combinación: edad 16, con credencial. `$puede_votar = 16 >= 18`, que es `false`. `$puede_votar_final = false && true`, que es `false`. Con `&&` basta que una falle: no puede votar por ser menor, aunque tenga credencial.
- Tercera combinación: edad 25, sin credencial. `$puede_votar = true`, pero `$puede_votar_final = true && false = false`. No puede votar porque le falta la credencial.
- Fíjate en la técnica de guardar la comparación en `$puede_votar` antes de combinarla: el código se lee casi como español ("puede votar final es puede votar y tiene credencial"). Cuando en la Unidad 3 veas las condicionales, esta forma de escribir te va a hacer la vida mucho más fácil.
- Como reto extra: agrega una tercera condición con `||`, por ejemplo `$es_extranjero_con_permiso`, de modo que pueda votar si (es mayor y tiene credencial) o si tiene permiso especial. Usa paréntesis para agrupar.

Ejercicio 4

Arma el recibo de una tiendita usando concatenación. Declara `$cliente`, `$producto1`, `$precio1`, `$producto2`, `$precio2`. Calcula `$subtotal` sumando los dos precios, calcula el IVA del 16% y el `$total`. Luego arma una variable `$recibo` empezando con un encabezado y usando `.=` para agregar una línea por cada dato: cliente, cada producto con su precio, subtotal, IVA y total. Muestra el recibo con `echo`. Usa constantes con `define()` para el IVA y para el nombre de la tienda.

Solución 4

```

define("NOMBRE_TIENDA", "Abarrotes El Colibri");
define("IVA", 0.16);
$cliente = "Don Pancho";
$producto1 = "Kilo de tortillas";
$precio1 = 28;
$producto2 = "Litro de leche";
$precio2 = 32;
$subtotal = $precio1 + $precio2;
$iva_calculado = $subtotal * IVA;
$total = $subtotal + $iva_calculado;
$recibo = "==== " . NOMBRE_TIENDA . " =====";
$recibo .= "Cliente: " . $cliente;
$recibo .= "Producto: " . $producto1 . " - $" . $precio1;
$recibo .= "Producto: " . $producto2 . " - $" . $precio2;
$recibo .= "Subtotal: $" . $subtotal;
$recibo .= "IVA (16%): $" . $iva_calculado;
$recibo .= "TOTAL: $" . $total;
$recibo .= "Gracias por su compra.";
echo $recibo;

```

Explicación paso a paso:

- Las dos líneas de `define()` crean las constantes `NOMBRE_TIENDA` e `IVA`. Van en mayúsculas por convención y no llevan \$ ni al definir las ni al usarlas. Si mañana el IVA cambia, solo tocas esa línea.
- Se declaran los datos de la compra: cliente, dos productos con sus precios. Los precios son integers porque no tienen centavos; si los tuvieran, serían floats.
- `$subtotal` suma los precios: $28 + 32 = 60$. `$iva_calculado` multiplica por la constante IVA: $60 * 0.16 = 9.6$. `$total` suma: 69.6.
- `$recibo` empieza con el encabezado, donde se concatena la constante `NOMBRE_TIENDA` (sin \$, recuerda). Luego cada línea se agrega con `.=`, pegando etiquetas fijas con los valores de las variables.
- Observa cómo en `"Producto: " . $producto1 . " - $" . $precio1` se encadenan tres concatenaciones en una sola línea: etiqueta, valor, separador, valor. El punto se puede encadenar cuantas veces quieras.
- El `echo` final muestra todo el recibo de un jalón. Este patrón (acumular en una variable e imprimir al final) es el que usarás para generar HTML dinámico en unidades posteriores.
- Como reto extra: agrega un tercer producto y un descuento fijo usando una constante `DESCUENTO`, y réstalo antes de calcular el IVA o después (decide tú el orden y observa

cómo cambia el total; en la vida real el orden sí importa y depende de las reglas del negocio).

Ejercicio 5

El ejercicio retador: escribe un script que simule la validación de un inicio de sesión. Declara \$usuario_correcto = "admin", \$clave_correcta = "secreto123", y luego \$usuario_ingresado y \$clave_ingresada con valores de prueba. Usa comparaciones ESTRUCTURAS (===) para verificar si el usuario y la clave coinciden, combinando con &&. Guarda el resultado en \$acceso_permitido y muéstralo con var_dump(). Luego prueba estos casos: (a) usuario y clave correctos, (b) usuario correcto pero clave incorrecta, (c) usuario "0" intentando entrar cuando el usuario correcto es "admin" (prueba también qué pasaría si usaras == en lugar de === con el usuario "0" y explica la diferencia). Escribe tus conclusiones en comentarios dentro del código.

Solución 5

```
$usuario_correcto = "admin";
$clave_correcta = "secreto123";
$usuario_ingresado = "admin";
$clave_ingresada = "secreto123";
$usuario_ok = $usuario_ingresado === $usuario_correcto;
$clave_ok = $clave_ingresada === $clave_correcta;
$acceso_permitido = $usuario_ok && $clave_ok;
var_dump($acceso_permitido);
$usuario_ingresado = "admin";
$clave_ingresada = "otra_clave";
$usuario_ok = $usuario_ingresado === $usuario_correcto;
$clave_ok = $clave_ingresada === $clave_correcta;
$acceso_permitido = $usuario_ok && $clave_ok;
var_dump($acceso_permitido);
$usuario_ingresado = "0";
$clave_ingresada = "secreto123";
$usuario_ok_estrieto = $usuario_ingresado === $usuario_correcto;
$usuario_ok_debil = $usuario_ingresado == $usuario_correcto;
var_dump($usuario_ok_estrieto);
var_dump($usuario_ok_debil);
```

Explicación paso a paso:

- Primero se guardan las credenciales correctas en \$usuario_correcto y \$clave_correcta. En un programa real vendrían de una base de datos; aquí las escribimos directo para practicar la lógica.

- Caso (a): usuario "admin" y clave "secreto123", ambos correctos. `$usuario_ok = "admin" === "admin"` es true; `$clave_ok = true`; `$acceso_permitido = true && true = true`. `var_dump` muestra `bool(true)`: acceso permitido.
- Caso (b): usuario correcto, clave "otra_clave". `$usuario_ok = true`, `$clave_ok = "otra_clave" === "secreto123" = false`. `true && false = false`. Acceso denegado, como debe ser.
- Caso (c): el interesante. `$usuario_ingresado = "0"`. Con comparación estricta, `"0" === "admin"` es false: tipos iguales (ambos string) pero valores distintos. Con comparación débil, `"0" == "admin"` también es false en PHP 8 (en versiones viejas, comparaciones raras entre strings y números daban sorpresas). Los dos `var_dump` te muestran el resultado de cada operador para que compares.
- La conclusión que debes escribir en tus comentarios: en validaciones de seguridad SIEMPRE usa `===`. La comparación débil hace conversiones automáticas que un atacante podría aprovechar, y en un login no hay lugar para "porteros buena onda". Además, nota el patrón de guardar cada comparación en su propia variable (`$usuario_ok`, `$clave_ok`) antes de combinarlas: si el acceso falla, puedes hacer `var_dump` de cada una por separado y saber exactamente cuál de las dos falló, en lugar de adivinar.
- Como reto extra: agrega una variable `$intentos_restantes` que empiece en 3 y reste 1 con el operador `--` cada vez que el acceso sea denegado, y muestra cuántos intentos quedan. Cuando llegues a la Unidad 4 (ciclos), podrás convertir esto en un sistema de intentos real.

Resumen de la unidad

Llegaste al final de la Unidad 2. Aquí tienes los puntos clave, condensados para repaso rápido. Si alguno no te queda claro, regresa a su subsección antes de seguir a la Unidad 3.

- Una variable es un espacio de memoria con un nombre donde guardas un valor para usarlo después: la memoria de tu programa. Piensa en cajas etiquetadas.
- En PHP todas las variables empiezan con `$`. El nombre debe empezar con letra o guion bajo, no puede empezar con número ni llevar espacios o caracteres especiales, y PHP distingue mayúsculas de minúsculas (`$total` no es `$Total`).
- No necesitas declarar el tipo de una variable: PHP lo deduce del valor (tipado dinámico). Puedes cambiar el valor cuando quieras, pero no reutilices una variable para datos de distinto tipo.

- Los cinco tipos básicos son: string (texto entre comillas), integer (número entero sin comillas), float (número con punto decimal), boolean (true o false) y null (ausencia intencional de valor).
- Con comillas dobles hay interpolación (las variables se reemplazan por sus valores); con comillas simples el texto sale literal. Si ves el nombre de tu variable impreso en pantalla, revisa las comillas.
- Los operadores aritméticos son + - * / % . **El módulo % da el residuo y sirve para saber si un número es par, repartir en ciclos y más. Respeta el orden de operaciones y usa paréntesis cuando dudes. Los decimales siempre llevan punto, nunca coma.**
- El operador = asigna (guarda un valor). Los operadores == y === comparan: == convierte tipos automáticamente (tipado débil, el portero buena onda) y === exige mismo valor y mismo tipo (el portero estricto). Usa === por defecto.
- El error más común de los principiantes es escribir = cuando querían ==. Si una condición siempre se cumple y "el código se ve bien", busca el signo sencillo.
- Los operadores lógicos son && (Y), || (O) y ! (NO). Con && todo debe ser verdadero; con || basta uno; ! invierte el valor. Agrupa con paréntesis para que tu intención sea obvia.
- Para pegar strings usa el punto ., nunca el +. El operador .= acumula texto en una variable, que es como armarás recibos, mensajes y páginas HTML.
- var_dump() muestra el tipo y el valor de una variable: tu lupa para depurar. Úsala sin miedo cada vez que algo no cuadre.
- Usa nombres descriptivos y consistentes (elige español o inglés, no mezcles), y convierte en constantes con define() los valores fijos que se repiten (impuestos, límites, nombres). Las constantes van en MAYÚSCULAS y no llevan \$.
- Inicializa tus variables con un valor sensato desde que las creas (\$total = 0;) en lugar de usarlas vacías.

En la Unidad 3 vas a usar todo esto para tomar decisiones: las estructuras condicionales (if, else, switch) te dejarán ejecutar código diferente según lo que haya en tus variables. Las comparaciones y los operadores lógicos que dominaste aquí son el corazón de esas decisiones, así que llegas bien armado. Nos vemos en la siguiente unidad.

UNIDAD 3: CONTROL DEL FLUJO DEL PROGRAMA

Introducción

Hasta este momento, todo lo que tus programas de PHP han hecho es avanzar en línea recta. Instrucción tras instrucción, de arriba hacia abajo, como si caminaran por un pasillo sin puertas ni vueltas. Declaran una variable, la muestran en pantalla, hacen una operación y se acabó. Eso está bien para empezar, pero ningún programa de verdad funciona así. Imagina una aplicación bancaria que te dejara sacar dinero sin preguntar si tienes saldo, o una tienda en línea que te cobrara sin revisar si tu tarjeta pasó. Un programa que no puede tomar decisiones ni repetir tareas es como un robot que solo sabe caminar hacia adelante: tarde o temprano se estrella contra la pared.

En esta unidad vas a darle a tus programas las dos habilidades que los vuelven realmente útiles: decidir y repetir. Decidir significa que el programa evalúe una situación y elija un camino u otro. Repetir significa que pueda hacer la misma tarea muchas veces sin que tengas que escribirla una y otra vez. Con estas dos habilidades, tus programas dejan de ser recetas rígidas y se convierten en asistentes que reaccionan a lo que pasa.

Piensa en un semáforo. Un semáforo no es una luz que se prende y se apaga al azar: es una máquina que toma decisiones todo el tiempo. Si la luz está en rojo, los carros se detienen. Si está en verde, avanzan. Si está en amarillo, se preparan para frenar. Nadie le escribe al semáforo una lista infinita de instrucciones del tipo "a las 8:00:01 pon verde, a las 8:00:02 sigue en verde, a las 8:00:03 sigue en verde...". Eso sería imposible. En lugar de eso, el semáforo tiene reglas: mientras la luz sea verde, deja pasar; cuando se cumpla el tiempo, cambia a amarillo; cuando el amarillo termine, pon rojo. Fíjate bien: ahí hay una decisión (¿qué color está prendido?) y una repetición (vuelve a revisar una y otra vez). Eso es exactamente lo que aprenderás a programar en esta unidad: el semáforo de tu código.

La otra analogía que te va a acompañar es la de lavar los trastes. Cuando lavas trastes no piensas cada movimiento por separado: tomas un plato, lo enjabonas, lo enjuagas, lo pones a secar... y repites. Tomas otro plato, lo enjabonas, lo enjuagas, lo pones a secar... y repites. La instrucción es la misma, lo que cambia es el plato. Si tuvieras que escribir un programa sin bucles para lavar diez platos, tendrías que copiar y pegar las mismas cuatro líneas diez veces. Y si fueran cien platos, cien veces. Los bucles existen para que escribas la instrucción una sola vez y el programa la repita las veces que hagan falta. Mientras haya platos sucios en el

fregadero, sigue lavando. Cuando ya no haya, detente. Esa frase - "mientras haya platos sucios, sigue lavando" - es literalmente la lógica del bucle while que vas a aprender.

En esta unidad verás primero las estructuras de decisión: if, else y elseif, que son como las bifurcaciones del camino. Después conocerás a switch, que es un atajo elegante cuando tienes muchas opciones que dependen de un solo valor. Luego entrarás al mundo de los bucles: while, do-while y for, cada uno con su personalidad y sus casos ideales. Aprenderás también dos comandos de control fino, break y continue, que te permiten salir de un bucle o saltarte una vuelta cuando lo necesites. Y cerrarás con lo más importante de todo: los errores que cometen todos los principiantes (incluido el punto y coma traicionero después del if y el signo de igual confundido con el de comparación) y las buenas prácticas que separan el código de aficionado del código de profesional.

No tengas prisa. Esta unidad es larga porque el tema lo merece: el control del flujo es el corazón de la programación. Una vez que lo domines, podrás escribir programas que validen formularios, que muestren contenido distinto según el usuario, que procesen listas de datos, que generen tablas automáticamente y que respondan de forma inteligente a lo que pase. Todo lo que sigue en este libro - funciones, arreglos, bases de datos - se construye sobre lo que vas a aprender aquí. Así que respira, prepárate un café y empecemos. Tu semáforo está a punto de encenderse.

Explicación detallada

El if: la primera decisión del programa

La estructura if es la forma más básica y más importante de tomar una decisión en un programa. Su lógica es idéntica a la que usas todos los días: si llueve, llevo paraguas. Si tengo hambre, como algo. Si el semáforo está en rojo, me detengo. Hay una condición y hay una acción que solo ocurre cuando la condición se cumple. En PHP se escribe así: la palabra if, entre paréntesis la condición que se evalúa, y entre llaves el bloque de código que se ejecuta únicamente cuando la condición es verdadera.

Vamos con un ejemplo de la vida real: verificar la mayoría de edad. Imagina que estás programando el registro de una página que vende boletos para un concierto donde solo entran mayores de edad. El programa recibe la edad del usuario y tiene que decidir: si la edad es 18 o más, muestra el mensaje de bienvenida y permite continuar; si no, no hace nada especial (por ahora). El código sería algo así: defines la variable edad con el valor que dio el usuario, y luego escribes if con la condición edad mayor o igual que 18. Dentro de las llaves pones el mensaje de bienvenida. Cuando PHP llega a esa línea, evalúa la condición: si es verdadera,

entra a las llaves y ejecuta lo que hay dentro; si es falsa, salta todo el bloque y sigue con lo que venga después, como si ese código no existiera.

Lo crucial que debes entender es que la condición de un `if` siempre se reduce a un valor booleano: verdadero o falso. No hay medias tintas. La expresión `edad >= 18` no es una pregunta filosófica: PHP la evalúa y obtiene `true` o `false`, y con base en eso decide. Por eso funcionan también condiciones más elaboradas, como las que combinan varios operadores lógicos: si la edad es mayor o igual a 18 y el usuario aceptó los términos, entonces puede continuar. Los operadores lógicos que viste en la unidad anterior (`and`, `or`, `not`) son los mejores amigos del `if`, porque te permiten construir condiciones que describen situaciones reales con precisión.

Un detalle que parece menor pero causa más dolores de cabeza que cualquier otro en los principiantes: después de la condición del `if` NO va punto y coma. El punto y coma en PHP significa "aquí termina la instrucción", así que si escribes `if (edad >= 18);` con ese punto y coma al final, le estás diciendo a PHP que la instrucción del `if` está vacía y que ya terminó. El bloque de llaves que sigue se ejecutará siempre, haya o no haya condición, porque ya no pertenece al `if`. Es como ponerle un candado a una puerta y dejar la ventana abierta: la condición queda inútil. Grábate esto: `if (condición)` y directo las llaves, sin punto y coma entre ellos.

Otro punto importante: el bloque del `if` puede contener una sola línea o cien líneas. Todo lo que esté dentro de las llaves se ejecuta cuando la condición es verdadera. Y las llaves, aunque PHP permite omitirlas cuando el bloque tiene una sola línea, es mejor costumbre ponerlas siempre. Más adelante, en la sección de buenas prácticas, verás por qué esta costumbre te va a salvar de errores muy tontos pero muy caros.

El else: el camino alternativo

El `if` solo cubre la mitad de la historia: qué pasa cuando la condición se cumple. Pero en la vida real casi siempre hay un plan B. Si llueve, llevo paraguas; si no llueve, no lo llevo. Si el usuario es mayor de edad, entra al concierto; si no lo es, le mostramos un mensaje de que no puede entrar. Para ese "si no" existe el `else`, que se coloca justo después del bloque del `if` y contiene el código que se ejecuta cuando la condición es falsa.

Volvamos al ejemplo de la mayoría de edad, ahora completo. El programa recibe la edad del usuario. Si la edad es 18 o mayor, muestra "Bienvenido, puedes entrar al concierto". Si no (else), muestra "Lo sentimos, este evento es solo para mayores de 18 años". Fíjate en la elegancia: con una sola condición cubres los dos escenarios posibles, y uno de los dos bloques se va a ejecutar sí o sí. Nunca se ejecutan los dos, y nunca se queda sin ejecutar ninguno. Es una bifurcación perfecta del camino.

El else no lleva condición entre paréntesis, porque no la necesita: es simplemente "todo lo demás". Si la condición del if fue falsa, entra al else automáticamente. Esto lo hace muy útil, pero también exige que pienses bien tu condición, porque el else va a atrapar absolutamente todo lo que no cumpla el if. Si tu condición del if está mal planteada, el else se ejecutará en casos donde no debería. Por ejemplo, si por error escribes la condición al revés (edad menor que 18 en el if), el mensaje de bienvenida caería en el else y se mostraría a los menores. El programa haría exactamente lo que le pediste, que no es lo mismo que lo que querías. Por eso conviene leer tus condiciones en voz alta: "si la edad es mayor o igual a 18, bienvenido; si no, lo sentimos". Si suena lógico al decirlo, probablemente está bien escrito.

Una trampa común con el else es querer ponerle una condición, como si fuera un segundo if. No funciona así: else no acepta paréntesis con condición. Si necesitas evaluar otra condición distinta, lo que buscas es elseif, que es el tema de la siguiente sección. El else es el final del camino, el cajón donde cae todo lo que no entró en las condiciones anteriores.

El elseif: cuando hay más de dos caminos

La vida rara vez se divide en solo dos opciones. Piensa en las calificaciones de la escuela: no es solo aprobado o reprobado. Un 10 es excelente, un 8 o 9 es bueno, un 6 o 7 es suficiente, y menos de 6 es reprobado. Son cuatro caminos distintos según el valor de una sola variable. Para estos casos existe elseif, que se coloca entre el if y el else y permite encadenar tantas condiciones como necesites.

La lógica es la siguiente: PHP evalúa primero la condición del if. Si es verdadera, ejecuta su bloque y se salta todo lo demás, incluyendo todos los elseif y el else. Si es falsa, pasa al primer elseif y evalúa su condición. Si esa es verdadera, ejecuta su bloque y se salta el resto. Y así sucesivamente hasta llegar al else final, que atrapa todo lo que no cumplió ninguna condición anterior. Es como una serie de filtros en cascada: el valor cae por el primero que lo atrape.

Vamos con el ejemplo completo del sistema de calificaciones. Supón que la variable calificación tiene un valor del 0 al 10. Escribes: if (calificación == 10), muestra "Excelente, calificación perfecta". elseif (calificación >= 8), muestra "Muy bien, buen desempeño". elseif (calificación >= 6), muestra "Aprobado, pero puedes mejorar". else, muestra "Reprobado, hay que estudiar más". Observa el orden: se evalúa de arriba hacia abajo, de la condición más exigente a la menos exigente. Si un alumno sacó 9, la primera condición (igual a 10) es falsa, pero la segunda (mayor o igual a 8) es verdadera, así que muestra "Muy bien" y se salta todo lo demás. El orden importa muchísimo: si pusieras primero la condición calificación >= 6, un 10 caería ahí y nunca llegaría a la de "Excelente". La regla de oro es ordenar tus elseif de lo más específico a lo más general, como colar arena: primero la malla fina, luego la gruesa.

Puedes encadenar todos los `elseif` que quieras, pero ojo con exagerar: una cadena de diez `elseif` seguidos suele ser señal de que deberías usar `switch` (que veremos en la siguiente sección) o replantear la lógica. Además, el `else` final es opcional pero muy recomendable: funciona como red de seguridad para valores inesperados. En el ejemplo de calificaciones, ¿qué pasa si por error la variable trae un 15 o un valor negativo? Con el `else` final al menos muestras un mensaje controlado en lugar de un silencio confuso. En programación, los silencios confusos son peores que los mensajes de error: al menos el error te dice que algo anda mal.

Un detalle de sintaxis: en PHP puedes escribir `elseif` todo junto o `else if` separado, y ambas formas funcionan. Sin embargo, la forma recomendada y más usada en la comunidad es `elseif` en una sola palabra. Usa esa y tu código se verá profesional y consistente con lo que verás en proyectos reales.

Los operadores de comparación, repasados en contexto

Ya conociste los operadores de comparación en la unidad anterior, pero ahora que los vas a usar dentro de condiciones reales vale la pena repasarlos con calma, porque aquí es donde se juega el partido. Un `if` con la comparación equivocada no marca error: simplemente toma la decisión equivocada en silencio, y esos son los errores más difíciles de encontrar.

La lista completa es corta. El operador `==` compara si dos valores son iguales, sin importar tanto el tipo de dato. El operador `===` compara si son iguales y además del mismo tipo: es más estricto. El operador `!=` significa diferente, y `!==` es su versión estricta. Luego están los de orden: `>` mayor que, `<` menor que, `>=` mayor o igual que, `<=` menor o igual que. Con estos seis (más sus versiones estrictas) construyes todas las condiciones del mundo.

La diferencia entre `==` y `===` merece un ejemplo concreto porque es una de las trampas clásicas de PHP. Imagina que un formulario te manda la edad como texto, "18", entre comillas, porque todo lo que llega de un formulario llega como cadena de texto. Si comparas `"18" == 18`, PHP dice que son iguales, porque el `==` es flexible y convierte los tipos para comparar. Pero si comparas `"18" === 18`, PHP dice que son diferentes, porque uno es texto y el otro es número. ¿Cuál debes usar? Depende de lo que necesites, pero la recomendación general es usar `===` cuando quieras estar seguro de que el tipo también coincide, y `==` cuando solo te importe el valor. En condiciones de seguridad (contraseñas, permisos) prefiere siempre la versión estricta.

El error más famoso de todos los tiempos en programación es confundir `=` con `==`. El signo `=` solo asigna: `$edad = 18` mete el valor 18 dentro de la variable `edad`. El doble `==` compara: `$edad == 18` pregunta si la variable vale 18. Si dentro de un `if` escribes `if ($edad = 18)` en lugar

de `if ($edad == 18)`, no estás preguntando nada: estás metiendo 18 en la variable, y como asignar un valor distinto de cero se evalúa como verdadero, el `if` siempre se ejecuta. Tu programa dirá "bienvenido" hasta a un bebé de dos años. PHP no te va a marcar error porque sintácticamente es válido; es lógicamente un desastre. Cada vez que escribas una condición, haz una pausa de un segundo y cuenta los signos de igual: para comparar, siempre dos.

Para comparar rangos, como en el ejemplo de las calificaciones, combina los operadores con `and`. Por ejemplo, para saber si una calificación está entre 6 y 7 escribirías: `si_calificación >= 6 and calificación <= 7`. Fíjate que necesitas mencionar la variable dos veces; no puedes escribir `6 <= calificación <= 7` como en matemáticas, porque PHP no lo interpreta así. Cada comparación es independiente y se une con operadores lógicos. Y recuerda los paréntesis cuando mezcles `and` con `or`: la expresión `edad >= 18 and (tiene_boleto or es_invitado)` deja clarísimo que el boleto o la invitación son alternativas entre sí, pero la mayoría de edad es obligatoria.

El switch: el conmutador de luz de las decisiones

Imagina que llegas a tu casa de noche y hay un tablero con varios interruptores: uno prende la luz de la sala, otro la de la cocina, otro la del pasillo. No enciendes todos para ver cuál era: miras la etiqueta, mueves el interruptor correcto y listo. El `switch` funciona igual: cuando tienes una sola variable y quieres hacer algo distinto según su valor exacto, el `switch` te deja etiquetar cada caso y saltar directo al que corresponda, sin evaluar una cascada de condiciones.

La sintaxis es así: escribes `switch` seguido de la variable entre paréntesis, y dentro de las llaves pones varios bloques `case`, cada uno con un valor posible y dos puntos. Después de cada `case` va el código para ese caso y, muy importante, la palabra `break`. Al final puedes poner un bloque `default`, que es el equivalente al `else`: se ejecuta cuando ningún `case` coincidió.

El ejemplo perfecto es el menú del día de una fonda. La variable `dia_semana` trae el día, y según el día se sirve un guisado distinto: `case "lunes":` pozole; `case "martes":` tacos; `case "miercoles":` enchiladas; y así. Con `if` y `elseif` tendrías que escribir `dia_semana == "lunes"`, `dia_semana == "martes"`... repitiendo la variable en cada condición. Con `switch` la escribes una sola vez y cada `case` solo pone el valor. Cuando los casos son muchos, el `switch` es más corto, más legible y más fácil de mantener: agregar un nuevo día es agregar un `case`, sin tocar la lógica de los demás.

Ahora, el `break` olvidado: es el error clásico del `switch` y tienes que conocerlo. El `break` le dice a PHP "ya terminaste este caso, salte del `switch`". Si lo omites, PHP no se detiene al terminar el código del `case`: sigue ejecutando el siguiente `case`, y el siguiente, hasta encontrar un `break`.

o hasta que se acabe el switch. Esto se llama "caída" entre casos. Si el día es lunes y olvidas el break, el programa servirá pozole, tacos, enchiladas y todo lo demás, uno tras otro. A veces la caída se usa a propósito (por ejemplo, cuando varios días comparten el mismo menú pones varios case seguidos sin break entre ellos hasta llegar al código común), pero el 99 por ciento de las veces que olvidas el break es un error. Regla simple: cada case lleva su break, salvo que tengas una razón muy consciente para omitirlo, y en ese caso déjalo comentado para que tu yo del futuro no lo "arregle".

¿Cuándo conviene el switch sobre el if? Cuando comparas una sola variable contra varios valores exactos. El switch solo hace comparaciones de igualdad, no sirve para rangos (no puedes poner case ≥ 8). Así que para las calificaciones con rangos sigue siendo mejor el if con elseif, pero para el menú del día, para elegir idioma, para procesar la opción de un menú numérico, el switch es el rey. Piensa así: si tu cadena de elseif repite la misma variable con == una y otra vez, es momento de cambiarla por un switch. Tu código te lo va a agradecer y quien lo lea también.

El bloque default merece una mención especial: es tu red de seguridad. En el menú del día, ¿qué pasa si la variable trae "domingo" y la fonda cierra los domingos? El default muestra "Hoy no hay servicio". Sin default, el switch simplemente no haría nada y el usuario se quedaría viendo una pantalla vacía, sin saber si el programa falló o si así era. Acostúmbrate a poner default siempre, aunque sea con un mensaje genérico: un programa que responde "no entendí esa opción" es mil veces mejor que uno que se queda callado.

El while: repetir mientras se cumpla la condición

Volvamos a los trastes. La regla era: mientras haya platos sucios en el fregadero, sigue lavando. El bucle while es exactamente esa frase traducida a código. Su sintaxis es parecida a la del if: la palabra while, entre paréntesis la condición, y entre llaves el bloque que se repite. La diferencia fundamental es que el if evalúa la condición una sola vez, mientras que el while la evalúa una y otra vez: ejecuta el bloque, regresa, vuelve a evaluar la condición; si sigue siendo verdadera, ejecuta el bloque otra vez; y así hasta que la condición se vuelva falsa. En ese momento sale del bucle y continúa con lo que sigue.

El ejemplo clásico es contar del 1 al 10. Creas una variable contador con valor 1. Escribes while (contador <= 10): dentro del bloque muestras el valor del contador y luego le sumas 1. Vamos a seguirle la pista paso a paso, porque entender este recorrido es entender los bucles. Primera vuelta: contador vale 1, la condición 1 <= 10 es verdadera, muestra 1, suma 1, contador ahora vale 2. Segunda vuelta: 2 <= 10 verdadera, muestra 2, contador vale 3. Y así sigue... Décima vuelta: contador vale 10, 10 <= 10 verdadera, muestra 10, contador vale 11. Vuelta once: la condición 11 <= 10 es falsa, el bucle termina. Se mostraron los números del 1 al 10 y el programa siguió adelante. Magia simple y poderosa.

Fíjate en el ingrediente secreto: la línea que incrementa el contador. Sin ella, el contador siempre valdría 1, la condición 1 <= 10 siempre sería verdadera y el bucle no terminaría nunca: un bucle infinito que mostraría el número 1 hasta que se acabara la memoria o tu paciencia. Todo bucle while necesita que algo dentro del bloque acerque la condición a volverse falsa. En el ejemplo de los trastes, cada plato que lavas reduce los platos sucios; en el contador, cada incremento acerca el valor al límite. Si nada cambia la condición, nada detiene el bucle. Este es el error número uno de los principiantes con while, así que cada vez que escribas uno pregúntate: ¿qué línea de mi bloque hace que la condición eventualmente sea falsa? Si no encuentras la respuesta, no ejecutes ese código todavía.

El while es ideal cuando no sabes de antemano cuántas veces se va a repetir algo. Por ejemplo: leer líneas de un archivo hasta que se acabe, pedirle al usuario su contraseña hasta que la escriba bien, procesar los resultados de una base de datos hasta que ya no haya más. En todos esos casos no puedes decir "repite 10 veces" porque no sabes cuántos datos vienen; lo que sí sabes es la condición de paro: mientras haya datos, sigue. Esa es la pregunta para elegir while: ¿sé cuántas vueltas dará, o solo sé cuándo debe detenerse? Si solo sabes cuándo detenerse, while es tu bucle.

Un consejo práctico: inicializa siempre tu variable de control antes del bucle. Si escribes while (contador <= 10) sin haberle dado un valor inicial a contador, PHP trabajará con una variable vacía o inexistente y el comportamiento será impredecible. La receta del while tiene tres pasos: 1) inicializar la variable antes del bucle, 2) poner la condición entre paréntesis, 3) modificar la variable dentro del bloque para que el bucle pueda terminar. Memorízala y los bucles infinitos dejarán de visitarte.

El do-while: primero actúa, después pregunta

El do-while es el hermano rebelde del while: hace exactamente lo mismo, repetir un bloque mientras una condición sea verdadera, pero con una diferencia crucial de orden. El while primero evalúa la condición y después ejecuta el bloque; el do-while primero ejecuta el bloque y después evalúa la condición. Parece un detalle menor, pero cambia todo: el do-while

garantiza que el bloque se ejecute al menos una vez, aunque la condición sea falsa desde el principio.

La analogía perfecta es probar la sopa. Cuando cocinas, no preguntas "¿le falta sal?" antes de probarla: primero la pruebas (actúas) y después decides si le falta sal (evalúas). Si le falta, le pones sal y vuelves a probar. La primera probada ocurre sí o sí, sin condiciones previas. El `do-while` es eso: `do { prueba la sopa; } while (le falta sal)`. Incluso si la sopa ya estaba perfecta, la probaste una vez. Con un `while` normal, si la condición inicial fuera falsa, el bloque no se ejecutaría ni una sola vez: sería como decidir si la sopa necesita sal sin haberla probado nunca.

La sintaxis tiene su peculiaridad: se escribe `do`, luego el bloque entre llaves, y después `while` (condición) con punto y coma al final. Ese punto y coma final es obligatorio y es una de las pocas veces en PHP donde un punto y coma cierra una estructura de control. No lo olvides o tendrás un error de sintaxis.

El caso de uso estrella del `do-while` es el juego de "adivina el número". El programa piensa un número y el jugador tiene que adivinarlo. Necesitas pedirle un número al jugador al menos una vez, y seguir pidiendo mientras no adivine. Con `do-while` queda natural: `do { pide el número; dile si es mayor o menor; } while (número != secreto)`. La primera petición ocurre garantizada, y el bucle sigue mientras no adivine. Si intentarás esto con `while`, tendrías que pedir el número una vez antes del bucle y luego repetir la petición dentro del bucle: código duplicado, feo y propenso a errores. Cada vez que necesites "haz esto al menos una vez y repite si hace falta", piensa en `do-while`: menús interactivos que se muestran mínimo una vez, validaciones donde pides un dato y lo vuelves a pedir si es inválido, juegos por turnos.

Ojo con el mismo peligro del `while`: el `do-while` también puede volverse infinito si nada dentro del bloque acerca la condición a su fin. En el juego de adivinar, si olvidas pedir el nuevo número dentro del bucle, el jugador nunca podrá cambiar su respuesta y el programa preguntará eternamente. La receta de seguridad es la misma: asegúrate de que algo dentro del bloque modifique las variables de la condición.

El for: la receta de tres pasos

Si el `while` es "repite mientras se cumpla esto", el `for` es "repite esto un número exacto de veces". Es el bucle más usado en programación y tiene una sintaxis compacta que al principio intimida pero que en realidad es una receta de cocina con tres ingredientes, separados por punto y coma dentro de los paréntesis: `for (inicio; condición; incremento)`.

Vamos a desmenuzar la receta con el ejemplo de la tabla de multiplicar del 5. Quieres mostrar $5 \times 1 = 5$, $5 \times 2 = 10$, hasta $5 \times 10 = 50$. El `for` queda así: `for ($i = 1; $i <= 10; $i++)`. Primer

ingrediente, el inicio: $i = 1$. Se ejecuta una sola vez, antes de empezar las vueltas, y sirve para preparar la variable contadora. Es como poner los ingredientes sobre la mesa antes de cocinar. Segundo ingrediente, la condición: $i \leq 10$. Se evalúa antes de cada vuelta; mientras sea verdadera, el bloque se ejecuta. Es el equivalente a "hornea mientras el pan no esté dorado". Tercer ingrediente, el incremento: $i++$. Se ejecuta al final de cada vuelta, después del bloque, y normalmente acerca la variable al final. Es como "voltea la tortilla" después de cada lado.

Sigamos el recorrido vuelta por vuelta para que quede cristalino. Inicio: i vale 1. ¿ $i \leq 10$? Sí. Ejecuta el bloque: muestra " $5 \times 1 = 5$ ". Incremento: i pasa a 2. ¿ $i \leq 10$? Sí. Muestra " $5 \times 2 = 10$ ". Incremento: i pasa a 3... Y así hasta que i vale 10: muestra " $5 \times 10 = 50$ ", i pasa a 11, la condición $11 \leq 10$ es falsa y el bucle termina. Diez vueltas exactas, ni una más ni una menos. Esa precisión es la gran virtud del for.

El for brilla cuando sabes de antemano cuántas repeticiones necesitas: mostrar los 12 meses del año, generar las 10 filas de una tabla, procesar los 30 alumnos de una lista, contar regresivamente del 10 al 1 para un despegue (for ($i = 10$; $i \geq 1$; $i--$)... fíjate que el incremento también puede ser un decremento). La variable contadora por convención se llama i (de "índice" o "iterador"), y si necesitas bucles anidados usarás j , k para los internos. No es obligatorio, pero todo programador del mundo entiende qué significa i en un for, así que úsala y tu código hablará el idioma universal.

Un error típico con el for es poner mal la condición de paro por uno: el famoso "error de uno más, uno menos". Si quieres los números del 1 al 10 y escribes $i < 10$, obtendrás del 1 al 9: te faltó el 10. Si escribes $i \leq 10$ empezando en 0, obtendrás 11 vueltas (del 0 al 10). La forma de no equivocarte es contar con los dedos la primera y la última vuelta antes de ejecutar: ¿con $i = 1$ y $i \leq 10$ la última vuelta muestra el 10? Sí. ¿La primera muestra el 1? Sí. Listo, diez vueltas correctas. Este hábito de simular mentalmente los extremos te va a ahorrar horas de depuración.

Otro detalle: los tres ingredientes del for son opcionales por separado, aunque casi siempre usarás los tres. Podrías escribir for (;), que es un bucle infinito a propósito (se usa con un break dentro para salir bajo alguna condición), pero no te lo recomiendo mientras aprendes: es como manejar sin frenos. Mantén tus for completos y predecibles: inicio claro, condición clara, incremento claro.

Break y continue: el control fino del bucle

A veces necesitas alterar el ritmo normal del bucle. Imagina que estás revisando una lista de 100 alumnos buscando a uno en particular: cuando lo encuentras en la posición 23, ¿para qué seguir revisando los otros 77? Ahí entra break, que significa "salte del bucle inmediatamente". El programa abandona el bucle en ese instante y continúa con la primera línea después de él. Es el botón de emergencia, la salida rápida.

El ejemplo clásico: buscar un número en una lista. Recorres un arreglo de calificaciones con un for, y cuando encuentras la primera calificación reprobatoria muestras un mensaje y ejecutas break. Sin el break, el bucle seguiría revisando todas las demás calificaciones aunque ya encontraste lo que buscabas: trabajo desperdiciado. El break también es la forma elegante de salir de un bucle infinito intencional: while (true) con un if adentro que ejecuta break cuando se cumple la condición de salida. Es un patrón válido y muy usado, siempre que la condición del break sea alcanzable.

Su compañero es continue, que significa "sáltate el resto de esta vuelta y pasa a la siguiente". Si break es salir del edificio, continue es saltarse un escalón. Ejemplo: estás procesando las calificaciones de un grupo y quieres calcular el promedio solo de los aprobados. Dentro del bucle pones: si la calificación es menor a 6, continue. Esa vuelta se interrumpe, la calificación reprobada no se suma, y el bucle pasa directo al siguiente alumno. Sin continue tendrías que envolver todo el procesamiento en un if, indentando más y más el código; con continue el código queda plano y legible: primero descartas lo que no te sirve, luego procesas lo que sí.

Una advertencia importante: break y continue afectan solo al bucle donde están escritos directamente. Si tienes bucles anidados (uno dentro de otro) y pones un break en el interno, solo sales del interno; el externo sigue dando vueltas. Esto confunde a muchos principiantes que esperan salir de todo. PHP permite especificar cuántos niveles romper con break 2 (sale de dos niveles), pero úsalo con moderación porque puede volver el código difícil de seguir. En general, si necesitas salir de varios niveles a la vez, quizá tu lógica necesita replantearse con una función o una bandera.

Bucles infinitos: el monstruo que tú mismo creas (y cómo domarlo)

Un bucle infinito es un bucle cuya condición nunca se vuelve falsa, así que repite su bloque eternamente. En la práctica no es eternamente: el programa se queda "colgado", consume memoria o tiempo hasta que el servidor lo mata o tú lo detienes con frustración. Todos los programadores, sin excepción, han creado bucles infinitos por accidente. No es señal de torpeza: es un rito de paso. Lo importante es saber reconocer las tres causas y cómo prevenirlas.

Causa uno: olvidar modificar la variable de control. El ejemplo del contador que nunca se incrementa: `while ($contador <= 10) { muestra $contador; }` sin la línea `$contador++`. La condición siempre es verdadera porque nada cambia. Prevención: la receta de tres pasos del `while` (inicializar, condicionar, modificar) y la revisión mental antes de ejecutar.

Causa dos: modificar la variable en la dirección equivocada. Quieres contar del 1 al 10 pero escribes `$contador--` en lugar de `$contador++`. El contador baja a 0, a -1, a -100... y la condición `$contador <= 10` sigue siendo verdadera para siempre, porque cada vez el número es más chico. Es como querer llegar a la esquina caminando hacia atrás: cada paso te aleja. Prevención: verifica que el cambio acerque la variable al valor que hace falsa la condición. Si cuentas hacia arriba, incrementa; si cuentas hacia abajo, decrementa; y la condición debe apuntar al límite correcto.

Causa tres: una condición que lógicamente nunca puede ser falsa. Por ejemplo `while ($numero != 0)` donde dentro del bloque solo muestras el número pero nunca lo modificas, o una condición con `or` donde una parte siempre es verdadera. Prevención: lee la condición en voz alta y pregúntate "¿qué tendría que pasar para que esto sea falso?" Si no hay una respuesta clara ligada a algo que ocurre dentro del bloque, hay un problema.

Cuando un bucle infinito ya está corriendo en tu servidor local, no entres en pánico: detén la ejecución (en la terminal con `Ctrl+C`, en el navegador deteniendo la carga) y revisa esas tres causas en orden. Y un truco de profesional mientras depuras: pon un contador de seguridad. Si tu bucle debería dar máximo 100 vueltas, agrega una variable `$seguridad` que se incremente en cada vuelta y un `if` que haga `break` cuando pase de 1000. Así, si algo sale mal, el programa se detiene solo en lugar de colgarse. Cuando el código ya funciona, puedes quitar el contador o dejarlo como protección.

Bucles anidados: un bucle dentro de otro

Un bucle anidado es simplemente un bucle cuyo bloque contiene a otro bucle completo. El bucle de afuera se llama externo y el de adentro interno. La lógica es como las manecillas del reloj: el minutero (interno) da vueltas completas una y otra vez, y cada vez que termina una vuelta, el horario (externo) avanza un pasito. Por cada vuelta del bucle externo, el bucle interno ejecuta todas sus vueltas completas.

El ejemplo más claro es generar una tabla de multiplicar completa, del 1 al 5. El bucle externo recorre los números del 1 al 5 (las tablas), y por cada uno, el bucle interno recorre del 1 al 10 (los multiplicadores). Primera vuelta del externo: tabla del 1; el interno muestra `1x1`, `1x2`... `1x10`. Segunda vuelta del externo: tabla del 2; el interno muestra `2x1`, `2x2`... `2x10`. Total de operaciones: 5 por 10 = 50. Fíjate en la multiplicación: con bucles anidados el total de vueltas

internas es el producto de las vueltas de cada nivel. Dos bucles de 10 dan 100 vueltas; tres bucles de 10 dan 1000. Esto es importante porque los bucles anidados pueden volverse lentos muy rápido si los niveles son grandes: es el primer encuentro con la idea de que no todos los algoritmos cuestan lo mismo.

Otro uso natural de los anidados es generar tablas HTML. Una tabla tiene filas y columnas: el bucle externo genera las filas (`<tr>`) y el interno genera las celdas de cada fila (`<td>`). Con dos `for` pequeños produces una tabla de 5 filas por 4 columnas, es decir, 20 celdas, sin escribir 20 líneas de HTML. Esta es una de las aplicaciones más comunes en PHP real: dibujar calendarios, cuadrículas de productos, tableros de juego. Cuando veas una estructura de "filas que contienen columnas", piensa en bucles anidados casi por reflejo.

Dos consejos para no perderte en los anidados. Primero, usa variables distintas para cada nivel: `$i` para el externo, `$j` para el interno (y `$k` si hubiera un tercero). Si usas la misma variable en los dos niveles, el interno la modificará y el externo perderá la cuenta: un desastre silencioso. Segundo, mantén el cuerpo de cada bucle corto; si el bloque interno crece mucho, es señal de que conviene mover esa lógica a otro lado (más adelante aprenderás a usar funciones para eso). Los anidados son poderosos pero se vuelven ilegibles rápido si los dejas crecer sin control.

Errores comunes: la galería de los horrores (para que no te espanten)

Todos los errores de esta sección los comete todo el mundo al aprender. Conócelos ahora y cuando te topes con uno no perderás horas: lo reconocerás por su cara.

El punto y coma después del `if` es el más traicionero. Escribes `if ($edad >= 18);` y luego el bloque entre llaves. Ese punto y coma le dice a PHP que el `if` terminó ahí, con un bloque vacío. Las llaves que siguen ya no pertenecen al `if`: se ejecutan siempre. El resultado es que tu "verificación de edad" deja pasar a todos. Lo cruel es que el código se ve casi bien a simple vista. Regla: entre la condición y la llave de apertura no va nada, ni punto y coma ni comentarios.

El signo de igual simple en vez del doble es el más famoso. `if ($password = "secreto")` asigna en lugar de comparar, y como asignar un texto no vacío es verdadero, el `if` siempre se ejecuta: tu validación de contraseña acepta cualquier cosa. Cuenta los iguales cada vez: comparación son dos, asignación es uno. Algunos programadores escriben la comparación al revés, `if ("secreto" == $password)`, para que si se les olvida un igual el código marque error en lugar de fallar en silencio; es un truco válido si te acomoda, aunque lo normal es simplemente estar atento.

Olvidar incrementar el contador en el while ya lo vimos: bucle infinito garantizado. El síntoma es una página que nunca termina de cargar. La cura es la receta de tres pasos: inicializar antes, condicionar en el paréntesis, modificar dentro.

El break faltante en el switch produce la "caída" entre casos: se ejecutan varios case seguidos cuando solo querías uno. El síntoma es que tu programa hace de más: el menú del día sirve todos los guisados, la opción elegida ejecuta también la siguiente. La cura es poner break al final de cada case, sin excepciones mientras aprendes.

El error de uno más o uno menos en el for (< en vez de <= o viceversa) hace que tu bucle dé una vuelta de más o una de menos. El síntoma típico: una tabla con una fila faltante o una de más, un conteo que llega a 9 en vez de 10. La cura es simular la primera y la última vuelta con lápiz y papel (o mentalmente) antes de dar el código por bueno.

Comparar textos con mayúsculas y minúsculas mezcladas también muerde: "Lunes" no es igual a "lunes" para PHP. Si tu switch espera "lunes" y la variable trae "Lunes", caerá en el default. La cura es normalizar los datos antes de comparar (más adelante verás funciones para pasar todo a minúsculas) o asegurarte de que los valores que comparas vengan en el formato esperado.

Y el clásico de los anidados: reutilizar la misma variable contadora en el bucle interno y el externo. El interno la deja en su valor final y el externo ya no sabe en qué vuelta iba. Usa \$i, \$j, \$k, una por nivel, siempre.

Buenas prácticas: escribe código que tu yo del futuro te agradezca

Las buenas prácticas no son caprichos estéticos: son hábitos que evitan errores y hacen que tu código se pueda leer, entender y modificar meses después, por ti o por otra persona.

Pon llaves siempre, aunque el bloque tenga una sola línea. PHP permite escribir `if ($edad >= 18) echo "Hola";` sin llaves, y funciona. El problema llega cuando meses después agregas una segunda línea debajo creyendo que pertenece al if: sin llaves, esa segunda línea se ejecuta siempre, y tendrás un error fantasma difícilísimo de detectar. Las llaves cuestan dos caracteres y te compran tranquilidad. Úsalas siempre, sin excepción.

Escribe condiciones legibles, que se lean como español. `if ($edad >= 18)` se entiende al instante; `if ($e >= 18)` obliga a adivinar qué es \$e. Usa nombres de variables descriptivos: \$edad_usuario, \$calificación, \$dia_semana. Y si una condición se vuelve muy larga, divídela en variables booleanas con nombre claro: `$es_mayor = $edad >= 18; $tiene_boleto = ...; if ($es_mayor and $tiene_boleto)`. Cada variable cuenta una parte de la historia y el if final se lee solo.

Evita la anidación profunda. Un if dentro de un if dentro de un if dentro de un while es como una caja dentro de otra caja dentro de otra: para llegar al centro tienes que recordar todas las condiciones que te trajeron ahí. Si te descubres con más de tres niveles de indentación, detente y replantea: quizá puedas validar primero los casos inválidos y salir temprano (con un else que termine, o más adelante con return), dejando el camino principal plano y legible. El código plano se entiende; el código en forma de pirámide se sufre.

Comenta el porqué, no el qué. Un comentario que dice "// incrementa el contador" sobre la línea \$i++ no aporta nada: el código ya dice eso. Un comentario útil explica la intención: "// se permite un intentó extra porque es la primera vez que el usuario entra". Los comentarios buenos responden "por qué", no "qué".

Y por último, prueba tus estructuras con valores en los extremos: el valor mínimo, el máximo, uno fuera de rango, un texto vacío. En el ejemplo de calificaciones prueba con 10, con 0, con 6 (el límite del aprobado) y con un valor raro como -3. Los errores viven en las fronteras: el 6 que debería aprobar pero reprueba por un < mal puesto, el 10 que cae en el mensaje equivocado por el orden de los elseif. Probar los bordes es el hábito que separa al programador que "cree que funciona" del que "sabe que funciona".

Ejemplos paso a paso

En esta sección cada ejemplo trae su código completo y después una explicación línea por línea. Te recomiendo copiar cada ejemplo, ejecutarlo y luego modificarlo: cambia los valores, rompe algo a propósito y observa qué pasa. Así se aprende de verdad.

Ejemplo 1: verificar la mayoría de edad

Este es el ejemplo fundacional del if con else. El programa recibe una edad y decide si la persona puede entrar a un evento para mayores de edad.

```
<?php
$edad = 20;

if ($edad >= 18) {
    echo "Bienvenido, puedes entrar al evento.";
} else {
    echo "Lo sentimos, el evento es solo para mayores de 18.";
}
?>
```

Explicación línea por línea:

- La línea `$edad = 20;` crea la variable `edad` y le guarda el valor 20. En un programa real este valor vendría de un formulario, pero para el ejemplo lo ponemos directo.
- La línea `if ($edad >= 18) {` abre la estructura de decisión. PHP evalúa la condición `$edad >= 18`, es decir, pregunta si 20 es mayor o igual que 18. La respuesta es verdadera.
- Como la condición fue verdadera, PHP entra al bloque y ejecuta `echo "Bienvenido, puedes entrar al evento.";`, que muestra ese mensaje en pantalla.
- La llave de cierre `}` termina el bloque del `if`, y la palabra `else` abre el bloque alternativo. Como ya se ejecutó el `if`, PHP salta todo el bloque del `else` sin siquiera mirarlo.
- Si cambiaras `$edad` a 15, la condición sería falsa, PHP saltaría el primer bloque y ejecutaría el del `else`, mostrando el mensaje de rechazo. Uno de los dos mensajes aparece siempre, nunca los dos y nunca ninguno.

Ejemplo 2: sistema de calificaciones con `elseif`

Aquí encadenamos varias condiciones para clasificar una calificación del 0 al 10 en cuatro niveles. Fíjate en el orden: de la condición más exigente a la más general.

```
<?php
$calificacion = 8;

if ($calificacion == 10) {
    echo "Excelente: calificacion perfecta.";
} elseif ($calificacion >= 8) {
    echo "Muy bien: buen desempeno.";
} elseif ($calificacion >= 6) {
    echo "Aprobado: puedes mejorar.";
} else {
    echo "Reprobado: hay que estudiar mas.";
}
?>
```

Explicación línea por línea:

- `$calificación = 8;` guarda el valor a evaluar. Prueba después con 10, con 6, con 4 y con 0 para ver cada camino.
- `if ($calificación == 10)` pregunta si la calificación es exactamente 10. Con valor 8 la respuesta es falsa, así que PHP no entra a este bloque y pasa al siguiente.

- elseif (\$calificación >= 8) pregunta si 8 es mayor o igual que 8. Es verdadera, así que ejecuta echo "Muy bien: buen desempeño.";
- En cuanto un bloque se ejecuta, PHP se salta todos los elseif restantes y el else final. Por eso un 8 nunca llega a la rama de "Aprobado" aunque 8 también sea mayor o igual que 6: el orden de evaluación lo atrapó antes.
- elseif (\$calificación >= 6) solo se evalúa si las dos condiciones anteriores fueron falsas, es decir, para calificaciones de 6 y 7.
- El else final atrapa todo lo menor a 6 y muestra el mensaje de reprobado. También atraparía valores raros como -2 o 99, así que en un sistema real convendría validar el rango antes, pero como red de seguridad funciona.

Ejemplo 3: menú del día con switch

La fonda "El Buen Sazón" sirve un guisado distinto cada día. Como comparamos una sola variable contra valores exactos, el switch es la herramienta ideal.

```
<?php
$dia = "miercoles";

switch ($dia) {
    case "lunes":
        echo "Hoy toca pozole.";
        break;
    case "martes":
        echo "Hoy tocan tacos.";
        break;
    case "miercoles":
        echo "Hoy tocan enchiladas.";
        break;
    case "jueves":
        echo "Hoy toca mole.";
        break;
    case "viernes":
        echo "Hoy toca pescado.";
        break;
    default:
        echo "Hoy no hay servicio.";
}
?>
```

Explicación línea por línea:

- \$día = "miercoles"; guarda el día a consultar. Observa que usamos minúsculas sin acento para evitar el problema de "Miércoles" contra "miercoles".
- switch (\$día) { le dice a PHP: toma el valor de \$día y compáralo con cada case hasta encontrar el que coincida.
- PHP revisa case "lunes": "miercoles" no es igual a "lunes", así que lo salta. Lo mismo con case "martes".
- Al llegar a case "miercoles": hay coincidencia, así que ejecuta echo "Hoy tocan enchiladas.";
- La palabra break; detiene el switch inmediatamente. Sin ese break, PHP seguiría ejecutando el case del jueves y el del viernes aunque el día sea miércoles: la famosa caída entre casos.
- Los case restantes se saltan gracias al break. Si \$día fuera "domingo", ningún case coincidiría y se ejecutaría el bloque default, mostrando "Hoy no hay servicio". El default es tu red de seguridad: sin él, un día desconocido produciría un silencio total.

Ejemplo 4: contar del 1 al 10 con while

El bucle while en su forma más pura: repetir mientras una condición sea verdadera, con la variable de control modificada dentro del bloque.

```
<?php
$contador = 1;

while ($contador <= 10) {
    echo "Numero: " . $contador . "<br>";
    $contador++;
}
echo "Termine de contar.";
?>
```

Explicación línea por línea:

- \$contador = 1; inicializa la variable de control antes del bucle. Este es el paso uno de la receta del while: sin inicialización, la condición trabajaría con un valor impredecible.
- while (\$contador <= 10) { es la condición que se evalúa antes de cada vuelta. Mientras el contador sea menor o igual a 10, el bloque se repite.

- echo "Número: " . \$contador . "
"; muestra el valor actual. El punto concatena el texto con el número, y
 es un salto de línea en HTML para que cada número salga en su renglón.
- \$contador++; incrementa el contador en 1. Este es el paso tres de la receta y el más importante: sin esta línea el contador siempre valdría 1 y el bucle sería infinito.
- Recorrido mental: vuelta 1 muestra el 1 y el contador pasa a 2; vuelta 2 muestra el 2 y pasa a 3... vuelta 10 muestra el 10 y el contador pasa a 11; la condición 11 <= 10 es falsa y el bucle termina.
- echo "Termine de contar."; está fuera del bucle, así que se ejecuta una sola vez al final. Fíjate cómo la indentación y las llaves dejan clarísimo qué está dentro y qué está fuera del bucle.

Ejemplo 5: tabla de multiplicar con for

El for compacta inicio, condición e incremento en una sola línea. Ideal cuando sabes exactamente cuántas vueltas necesitas: aquí, 10.

```
<?php
$numero = 5;

for ($i = 1; $i <= 10; $i++) {
    $resultado = $numero * $i;
    echo $numero . " x " . $i . " = " . $resultado . "<br>";
}
?>
```

Explicación línea por línea:

- \$numero = 5; guarda de qué número queremos la tabla. Cambia este valor a 7 o a 9 y obtendrás otra tabla sin tocar el bucle: esa es la gracia de parametrizar.
- for (\$i = 1; \$i <= 10; \$i++) { contiene la receta de tres pasos: inicio (\$i = 1, se ejecuta una sola vez), condición (\$i <= 10, se evalúa antes de cada vuelta) e incremento (\$i++, se ejecuta al final de cada vuelta).
- \$resultado = \$numero * \$i; calcula la multiplicación de esta vuelta y la guarda. En la primera vuelta es 5 * 1 = 5; en la segunda, 5 * 2 = 10.
- echo \$numero . " x " . \$i . " = " . \$resultado . "
"; arma la línea "5 x 1 = 5" concatenando piezas de texto con los valores. Los puntos unen todo en una sola cadena.

- Cuando \$i llega a 11, la condición \$i <= 10 es falsa y el bucle termina después de exactamente 10 vueltas. Ni una más ni una menos: esa precisión es la firma del for.
- Si por error escribieras \$i < 10 en la condición, la tabla llegaría solo hasta "5 x 9": el clásico error de uno menos. Simula siempre la última vuelta antes de dar el código por terminado.

Ejemplo 6: adivina el número con do-while

El juego donde el programa piensa un número y el jugador lo adivina. Necesitamos pedir al menos una vez sí o sí, así que el do-while es perfecto. Como no podemos pedir datos de verdad al jugador en este ejemplo de libro, simulamos sus intentos con un arreglo.

```
<?php
$secreto = 7;
$intentos = array(3, 9, 7);
$indice = 0;

do {
    $intento = $intentos[$indice];
    $indice++;
    echo "Intentas con el " . $intento . ": ";
    if ($intento < $secreto) {
        echo "muy bajo.<br>";
    } elseif ($intento > $secreto) {
        echo "muy alto.<br>";
    } else {
        echo "correcto.<br>";
    }
} while ($intento != $secreto);

echo "Adivinaste en " . $indice . " intentos.";
?>
```

Explicación línea por línea:

- \$secreto = 7; es el número que hay que adivinar. \$intentos = array(3, 9, 7); simula lo que el jugador iría escribiendo: primero 3, luego 9, luego 7. \$índice = 0; marca por cuál intentó vamos.
- do { abre el bloque que se ejecutará al menos una vez, sin preguntar nada antes. Esta es la diferencia esencial con el while.

- `$intentó = $intentos[$índice];` toma el intentó actual del arreglo y `$índice++`; avanza al siguiente para la próxima vuelta. Estas dos líneas son las que hacen que la condición pueda cambiar: sin ellas el juego se ciclaría con el primer intentó para siempre.
- El `if` con `elseif` de adentro da la pista: si el intentó es menor que el secreto dice "muy bajo", si es mayor dice "muy alto", y si es igual dice "correcto". Fíjate cómo un `if` puede vivir tranquilamente dentro de un bucle.
- `} while ($intentó != $secreto);` cierra el bloque y evalúa la condición al final. Si el intentó fue distinto del secreto, se repite la vuelta; si fue igual, el bucle termina. El punto y coma después del paréntesis es obligatorio en el `do-while`.
- Recorrido: vuelta 1 intenta con 3 (muy bajo), vuelta 2 intenta con 9 (muy alto), vuelta 3 intenta con 7 (correcto) y la condición `7 != 7` es falsa, así que sale. El mensaje final dice que adivinó en 3 intentos.
- Aunque el primer intentó hubiera sido el correcto, el bloque se habría ejecutado una vez de todos modos: esa garantía es la razón de ser del `do-while`.

Ejemplo 7: tabla HTML generada con un bucle anidado

Dos bucles, uno dentro del otro, para dibujar una cuadrícula: el externo crea las filas y el interno crea las celdas de cada fila. De aquí salen calendarios, tableros y catálogos en el PHP real.

```
<?php
echo "<table border='1'>";

for ($fila = 1; $fila <= 4; $fila++) {
    echo "<tr>";
    for ($columna = 1; $columna <= 3; $columna++) {
        echo "<td>Fila " . $fila . ", Col " . $columna . "</td>";
    }
    echo "</tr>";
}

echo "</table>";
?>
```

Explicación línea por línea:

- `echo "<table border='1'>";` abre la tabla HTML antes de los bucles. Está fuera de ambos, así que se ejecuta una sola vez.

- `for ($fila = 1; $fila <= 4; $fila++)` { es el bucle externo: dará 4 vueltas, una por cada fila de la tabla. La variable se llama \$fila para que el código se lea solo.
- `echo "<tr>";` abre una fila de la tabla. Ocurre una vez por cada vuelta del bucle externo, es decir, 4 veces en total.
- `for ($columna = 1; $columna <= 3; $columna++)` { es el bucle interno: por cada vuelta del externo, este da sus 3 vueltas completas. Usa una variable distinta (\$columna) para no interferir con la del externo.
- `echo "<td>Fila " . $fila . ", Col " . $columna . "</td>";` crea una celda que dice en qué fila y columna está, por ejemplo "Fila 2, Col 3". Esta línea se ejecuta $4 \times 3 = 12$ veces: el producto de las vueltas de cada nivel.
- `echo "</tr>";` cierra la fila actual. Fíjate que está dentro del bucle externo pero fuera del interno: se ejecuta 4 veces, una por fila.
- `echo "</table>";` cierra la tabla, una sola vez al final. El resultado es una tabla de 4 filas por 3 columnas generada con apenas dos bucles: imagina escribir esas 12 celdas a mano y entenderás el poder de los anidados.

Ejemplo 8: break y continue en acción

Recorremos las calificaciones de un grupo: nos saltamos las reprobadas con continue para no meterlas al promedio, y nos detenemos con break en cuanto encontramos la primera calificación perfecta.

```

<?php
$calificaciones = array(5, 8, 4, 10, 9, 7);
$suma = 0;
$cuenta = 0;

for ($i = 0; $i < 6; $i++) {
    $cal = $calificaciones[$i];

    if ($cal < 6) {
        echo "Reprobado (" . $cal . "): se salta.<br>";
        continue;
    }

    if ($cal == 10) {
        echo "Perfecto (" . $cal . "): nos detenemos aqui.<br>";
        break;
    }

    $suma = $suma + $cal;
    $cuenta++;
}

echo "Aprobados procesados: " . $cuenta . ", suma: " . $suma . ".";
?>

```

Explicación línea por línea:

- `$calificaciones = array(5, 8, 4, 10, 9, 7);` es la lista a procesar. `$suma` y `$cuenta` empiezan en 0 y acumularán solo las calificaciones aprobadas que no sean perfectas.
- `for ($i = 0; $i < 6; $i++) {` recorre las 6 posiciones del arreglo. Los arreglos empiezan en la posición 0, por eso el contador va de 0 a 5.
- `$cal = $calificaciones[$i];` copia la calificación actual a una variable con nombre claro para trabajar cómodamente.
- El primer `if` pregunta si `$cal < 6`. Para el 5 y el 4 la respuesta es sí: muestra el mensaje y ejecuta `continue`, que salta directo a la siguiente vuelta. Las líneas de suma nunca se ejecutan para los reprobados.
- El segundo `if` pregunta si `$cal == 10`. Cuando llega al 10 muestra el mensaje y ejecuta `break`, que abandona el bucle por completo: el 9 y el 7 finales ni siquiera se revisan.
- `$suma = $suma + $cal;` y `$cuenta++;` solo se alcanzan para calificaciones aprobadas y no perfectas, es decir, para el 8. Al final `$cuenta` vale 1 y `$suma` vale 8.

- El mensaje final lo confirma. Observa el poder de combinar ambas herramientas: continue filtra lo que no sirve vuelta por vuelta, y break detiene todo cuando ya encontraste lo que buscabas.

Práctica

Es tu turno. Intenta resolver cada ejercicio por tu cuenta antes de mirar la solución: la programación se aprende con los dedos, no solo con los ojos. Si te atorras, relee la sección correspondiente y vuelve a intentarlo.

Ejercicio 1

Escribe un programa que reciba la temperatura en grados Celsius en una variable \$temperatura y muestre un mensaje según el clima: si la temperatura es mayor o igual a 30, "Hace mucho calor"; si es mayor o igual a 20, "Clima agradable"; si es mayor o igual a 10, "Hace fresco"; en cualquier otro caso, "Hace frío". Prueba tu programa con los valores 32, 25, 15 y 5.

Solución 1

```
<?php
$temperatura = 25;

if ($temperatura >= 30) {
    echo "Hace mucho calor.";
} elseif ($temperatura >= 20) {
    echo "Clima agradable.";
} elseif ($temperatura >= 10) {
    echo "Hace fresco.";
} else {
    echo "Hace frio.";
}
?>
```

Explicación paso a paso:

- Guardamos el valor a evaluar en \$temperatura. Para probar los cuatro casos del enunciado, cambia esta línea a 32, 25, 15 y 5, y ejecuta el programa cada vez.
- El if pregunta primero por la condición más exigente: \$temperatura >= 30. Con 32 entra aquí y muestra "Hace mucho calor", saltándose todo lo demás.

- Con 25, la primera condición es falsa y PHP pasa al primer elseif: $25 \geq 20$ es verdadera, así que muestra "Clima agradable".
- Con 15, las dos primeras condiciones son falsas; el segundo elseif ($15 \geq 10$) es verdadera y muestra "Hace fresco".
- Con 5, ninguna condición se cumple y el else final muestra "Hace frío".
- El orden descendente (30, 20, 10) es la clave: si pusieras primero ≥ 10 , el valor 32 caería ahí y nunca llegaría a "mucho calor". Siempre ordena de lo más específico a lo más general.

Ejercicio 2

Escribe un programa con switch que reciba en la variable \$mes un número del 1 al 12 y muestre a qué estación del año pertenece en México: primavera (3, 4, 5), verano (6, 7, 8), otoño (9, 10, 11) e invierno (12, 1, 2). Si el número no está entre 1 y 12, debe mostrar "Mes invalido". Aprovecha que varios case pueden compartir el mismo código.

Solución 2


```

<?php
$mes = 7;

switch ($mes) {
    case 3:
    case 4:
    case 5:
        echo "Primavera.";
        break;
    case 6:
    case 7:
    case 8:
        echo "Verano.";
        break;
    case 9:
    case 10:
    case 11:
        echo "Otono.";
        break;
    case 12:
    case 1:
    case 2:
        echo "Invierno.";
        break;
    default:
        echo "Mes invalido.";
}
?>

```

Explicación paso a paso:

- Guardamos el mes a consultar en \$mes. Prueba con 7, con 12, con 4 y con 15 para ver cada rama, incluida la de error.
- Este ejercicio usa a propósito la "caída" entre casos: varios case seguidos sin break entre ellos comparten el mismo bloque de código. Cuando \$mes vale 7, PHP entra en case 7 (después de saltar 3, 4, 5 y 6), no encuentra break, y sigue ejecutando hasta llegar al echo "Verano." y su break.
- Fíjate que aquí la omisión del break es intencional y está clara: los case agrupados caen juntos al mismo mensaje. Esto es muy distinto a olvidar el break por accidente.
- Cada grupo termina con su break para no caer al grupo siguiente: sin el break después de "Verano.", un mes 7 mostraría también "Otono.", "Invierno." y todo lo demás.

- El default atrapa cualquier valor fuera del 1 al 12, como 0, 15 o -3, y muestra "Mes invalido". Es la red de seguridad que todo switch debería tener.
- Compara esta solución con cómo se vería con if y elseif: tendrías que repetir \$mes == 3 or \$mes == 4 or \$mes == 5... en cada condición. El switch queda mucho más limpio cuando varios valores llevan al mismo resultado.

Ejercicio 3

Escribe un programa que use un bucle while para mostrar los números pares del 2 al 20, uno por línea. Después escribe la versión equivalente con un bucle for. Ambas versiones deben producir exactamente la misma salida.

Solución 3

Versión con while:

```
<?php
$numero = 2;

while ($numero <= 20) {
    echo $numero . "<br>";
    $numero = $numero + 2;
}
?>
```

Versión con for:

```
<?php
for ($numero = 2; $numero <= 20; $numero = $numero + 2) {
    echo $numero . "<br>";
}
?>
```

Explicación paso a paso:

- En la versión while seguimos la receta de tres pasos: inicializar (\$numero = 2 antes del bucle), condicionar (while (\$numero <= 20)) y modificar dentro del bloque (\$numero = \$numero + 2).
- El incremento es de 2 en 2 en lugar de 1 en 1: así solo tocamos los pares. Recorrido: muestra 2, pasa a 4, muestra 4, pasa a 6... muestra 20, pasa a 22, la condición 22 <= 20 es falsa y termina. Diez números en total.

- Si por error incrementaras de 1 en 1, obtendrías todos los números del 2 al 20, pares e impares. El tamaño del paso define qué valores visitas: es una decisión de diseño, no un accidente.
- En la versión for, los tres pasos de la receta quedan compactados en los paréntesis: inicio (\$número = 2), condición (\$número <= 20) e incremento (\$número = \$número + 2). El cuerpo del bucle queda con una sola línea.
- Ambas versiones son correctas y producen la misma salida. ¿Cuál elegir? Cuando el número de vueltas se conoce de antemano, el for suele leerse mejor porque toda la lógica de control está junta en una línea. El while brilla cuando no sabes cuántas vueltas habrá.
- Como ejercicio extra, modifica ambas versiones para mostrar los impares del 1 al 19: solo cambia el valor inicial a 1 y el límite a 19. Si te sale a la primera en ambas versiones, ya dominas la receta.

Ejercicio 4

Escribe un programa que genere con bucles anidados una pirámide de asteriscos de 5 pisos, así:

```
* *** *****
```

Cada fila *i* debe tener exactamente *i* asteriscos. Pista: el bucle externo controla las filas (del 1 al 5) y el bucle interno repite el asterisco tantas veces como indique la fila actual.

Solución 4

```
<?php
for ($fila = 1; $fila <= 5; $fila++) {
    for ($columna = 1; $columna <= $fila; $columna++) {
        echo "*";
    }
    echo "<br>";
}
?>
```

Explicación paso a paso:

- El bucle externo for (\$fila = 1; \$fila <= 5; \$fila++) da 5 vueltas, una por cada piso de la pirámide. La variable \$fila siempre sabe en qué piso estamos.

- El bucle interno for ($\$columna = 1; \$columna \leq \$fila; \$columna++$) es la parte ingeniosa: su condición de paro no es un número fijo, sino la variable $\$fila$ del bucle externo. En la vuelta 1 del externo, el interno da 1 vuelta; en la vuelta 2, da 2 vueltas; y así.
- Dentro del bucle interno, echo `"*"`; imprime un asterisco sin salto de línea, así los asteriscos de la misma fila quedan pegados uno junto al otro.
- Después de que el bucle interno termina sus vueltas, echo `"<br>"`; (qué está dentro del externo pero fuera del interno) baja a la siguiente línea para empezar el siguiente piso.
- Recorrido completo: fila 1 imprime `"*"` y salto; fila 2 imprime `"*"` **y salto; fila 3 imprime `"*"` y salto; fila 4 imprime `"*"` y salto; fila 5 imprime `"*"` y salto.** **Total de asteriscos: $1 + 2 + 3 + 4 + 5 = 15$.**
- Este patrón (el límite del bucle interno depende del externo) aparece en muchos problemas reales: tablas triangulares, calendarios donde cada semana tiene distinto número de días ocupados, y dibujos por consola. Domínalo y tendrás una herramienta muy versátil.
- Como reto extra, intenta la pirámide invertida (5, 4, 3, 2, 1 asteriscos): solo cambia la condición del bucle externo para que $\$fila$ vaya del 5 al 1.

Ejercicio 5

Escribe un programa que simule una alcancía: empieza con $\$ahorro = 0$ y cada "día" deposita $\$día * 10$ pesos, donde $\$día$ es el número del día (el día 1 deposita 10, el día 2 deposita 20, etc.). Usa un bucle para acumular hasta que el ahorro llegue o supere los 500 pesos, y al final muestra en qué día se alcanzó la meta y cuánto se ahorró en total. Si en algún día el depósito supera los 100 pesos, el programa debe detenerse ese mismo día con `break` porque "ya es mucho". Usa `continue` para saltar los días pares (esos días no se deposita nada).

Solución 5

```

<?php
$ahorro = 0;
$dia = 0;

while ($ahorro < 500) {
    $dia++;

    if ($dia % 2 == 0) {
        continue;
    }

    $deposito = $dia * 10;

    if ($deposito > 100) {
        echo "Dia " . $dia . ": el deposito de " . $deposito . " es demasiado, nos
detenemos.<br>";
        break;
    }

    $ahorro = $ahorro + $deposito;
    echo "Dia " . $dia . ": deposito " . $deposito . ", ahorro total " . $ahorro .
".<br>";
}

echo "Meta alcanzada o detenida el dia " . $dia . " con " . $ahorro . " pesos.";
?>

```

Explicación paso a paso:

- \$ahorro = 0; y \$día = 0; inicializan las dos variables de control antes del bucle, siguiendo la receta del while.
- while (\$ahorro < 500) es la condición de paro natural: el bucle sigue mientras no se haya llegado a la meta. No sabemos de antemano cuántos días tomará, por eso el while es la elección correcta y no el for.
- \$día++; va al inicio del bloque para que el primer día sea el 1 y no el 0. El orden importa: si incrementaras al final, el día 0 se procesaría primero.
- El primer if usa el operador módulo: \$día % 2 == 0 detecta los días pares (los divisibles entre 2). Cuando toca un día par, continue salta el resto de la vuelta: ese día no se deposita nada y se pasa directo al siguiente día. Fíjate que el \$día++ ya ocurrió, así que el bucle no se cicla: esto es clave, porque si el continue estuviera antes del incremento, los días pares se repetirían para siempre.

- $\$deposito = \$día * 10$; calcula cuánto toca depositar ese día: día 1 = 10, día 3 = 30, día 5 = 50, etc.
- El segundo if es la válvula de seguridad: si $\$deposito > 100$, muestra el aviso y ejecuta break, abandonando el bucle aunque no se haya llegado a los 500. Es el ejemplo del "botón de emergencia".
- Si pasamos ambos filtros, $\$ahorro = \$ahorro + \$deposito$; acumula el depósito y se muestra el avance del día.
- Recorrido real: día 1 deposita 10 (ahorro 10), día 2 se salta, día 3 deposita 30 (ahorro 40), día 4 se salta, día 5 deposita 50 (ahorro 90), día 6 se salta, día 7 deposita 70 (ahorro 160), día 8 se salta, día 9 deposita 90 (ahorro 250), día 10 se salta, día 11: el depósito sería 110, mayor a 100, así que se detiene con break. El mensaje final dice que se detuvo el día 11 con 250 pesos.
- Este ejercicio combina todo lo de la unidad: while con condición de paro, continue para filtrar, break para detener, if con módulo y acumuladores. Si lo entendiste completo, estás listo para la siguiente unidad.

Resumen de la unidad

- El flujo de un programa es el orden en que se ejecutan sus instrucciones. Con las estructuras de esta unidad el programa puede tomar decisiones y repetir tareas en lugar de avanzar siempre en línea recta.
- El if evalúa una condición una sola vez: si es verdadera ejecuta su bloque, si es falsa lo salta. Después de la condición no lleva punto y coma.
- El else es el camino alternativo: se ejecuta cuando la condición del if fue falsa. No lleva condición propia.
- El elseif encadena varias condiciones en cascada: se evalúan de arriba hacia abajo y solo se ejecuta el bloque de la primera que sea verdadera. Ordénalas de la más específica a la más general.
- Para comparar usa == (igualdad de valor) o === (igualdad de valor y tipo). Nunca uses = (asignación) dentro de una condición: no marca error pero siempre se evalúa como verdadera.
- El switch compara una sola variable contra varios valores exactos con bloques case. Es más limpio que una cadena larga de elseif cuando todos comparan lo mismo con ==.

- Cada case del switch debe terminar con break, o el programa "caerá" al siguiente case ejecutando código que no correspondía. El bloque default es la red de seguridad para valores inesperados.
- El while repite su bloque mientras la condición sea verdadera, evaluándola antes de cada vuelta. Su receta: inicializar la variable antes del bucle, poner la condición entre paréntesis y modificar la variable dentro del bloque.
- El do-while hace lo mismo que el while pero ejecuta el bloque primero y evalúa después, así que garantiza al menos una vuelta. Lleva punto y coma después del while final. Ideal para menús y validaciones que deben mostrarse mínimo una vez.
- El for compacta inicio, condición e incremento en una línea: for (\$i = 1; \$i <= 10; \$i++). Es perfecto cuando sabes de antemano cuántas vueltas necesitas.
- El break abandona el bucle por completo y el continue salta el resto de la vuelta actual para pasar a la siguiente. Solo afectan al bucle donde están escritos.
- Un bucle infinito ocurre cuando la condición nunca se vuelve falsa: por olvidar modificar la variable de control, por modificarla en la dirección equivocada o por una condición imposible de falsar. Antes de ejecutar un bucle, pregúntate qué línea acerca la condición a su fin.
- Los bucles anidados ponen un bucle dentro de otro: por cada vuelta del externo, el interno completa todas las suyas. Usa una variable distinta por nivel (\$i, \$j, \$k) y recuerda que el total de vueltas se multiplica.
- Errores clásicos que ya sabes reconocer: punto y coma después del if, = en vez de ==, contador sin incrementar, break faltante en el switch, error de uno más o uno menos en el for, y comparar textos con mayúsculas distintas.
- Buenas prácticas: pon llaves siempre aunque el bloque sea de una línea, escribe condiciones que se lean como español con variables de nombre claro, evita más de tres niveles de anidación, comenta el porqué y no el qué, y prueba tus estructuras con los valores de los bordes.
- Con decisiones y repeticiones ya puedes validar formularios, clasificar datos, generar tablas HTML automáticamente, procesar listas y construir menús interactivos: la base de todo programa real está en tus manos.

UNIDAD 4: FUNCIONES Y ARREGLOS

En las unidades anteriores aprendiste lo básico de PHP: variables, tipos de datos, condicionales y ciclos. Ya sabes guardar un dato en una variable, tomar decisiones con if y repetir acciones con while y for. Pero hasta ahora todo tu código ha sido como una lista de instrucciones corrida de principio a fin, como una cinta transportadora que solo va en una dirección.

En esta unidad vas a aprender las dos herramientas que convierten a un programador principiante en un programador de verdad: las funciones y los arreglos. Con ellas vas a poder guardar datos en colecciones ordenadas y reutilizar bloques de código sin tener que escribirlos cien veces. Te aviso desde ahorita: este es el punto donde muchos estudiantes sienten que la programación "hace clic" en su cabeza. Todo lo que viene después en el libro (formularios, bases de datos, sesiones, proyectos reales) se construye sobre lo que vas a ver aquí. Así que respira hondo, porque viene lo bueno.

Piensa en tu cocina. Imagina que todos los días preparas un licuado de platano. Si no tuvieras una licuadora, tendrías que machacar el platano a mano, buscar leche, mezclar todo en un vaso, batir con un tenedor... y al día siguiente repetir todo el proceso desde cero, otra vez, y otra vez. Sería un infierno. La licuadora es un aparato que hace UNA tarea compleja: le echas platano, leche y hielo, aprietas un botón, y sale el licuado. No necesitas saber cómo funcionan las aspas por dentro. Solo necesitas saber que entra y que sale.

Una función en PHP es exactamente eso: una licuadora de código. Tomas un bloque de instrucciones que haces una y otra vez, le pones un nombre, y cada vez que lo necesites lo "enciendes" escribiendo su nombre. Ya no repites el proceso completo. Le das los ingredientes (los parámetros) y te regresa el resultado (el return). Por ejemplo: imagina que en tu página web necesitas calcular el precio con IVA en diez lugares distintos. Sin funciones, escribes la multiplicación diez veces. Si un día cambia el IVA del 16 al 15 por ciento, tienes que buscar las diez multiplicaciones y cambiarlas una por una, con el riesgo de que se te olvide alguna. Con una función, el calculo vive en un solo lugar: la cambias una vez y los diez lugares quedan actualizados automáticamente. Eso es poder de verdad.

Ahora piensa en los arreglos. Vuelve a la cocina y piensa en tu alacena. No guardas las latas tiradas por toda la casa: las pones en un anaquel donde cada espacio tiene su lugar. Los arreglos son eso: un mueble para guardar datos. En lugar de tener treinta variables sueltas (\$alumno1, \$alumno2, \$alumno3...) que es imposible de manejar, guardas los treinta nombres en un solo arreglo y PHP te da herramientas para recorrerlos, contarlos, buscarlos y ordenarlos.

Hay dos tipos de arreglos que vas a dominar en esta unidad. Los arreglos indexados son como los casilleros de un gimnasio: cada casillero tiene un número (0, 1, 2, 3...) y guardas algo en cada uno. Los arreglos asociativos son como un diccionario o como la agenda de tu teléfono: en vez de números usas palabras como llave. En tu agenda no buscas "el contacto número 47", buscas "Maria" y te sale su teléfono. Esa diferencia (número contra nombre) es la clave de todo.

Y para recorrer estos muebles de datos existe `foreach`, que es como pasar tu dedo por cada casillero del gimnasio abriéndolo uno por uno: te entrega lo que hay dentro sin que tengas que contar cuantos casilleros hay. Es uno de los ciclos más usados en PHP y lo vas a usar en todos los proyectos del libro.

En esta unidad también vas a conocer el concepto de alcance de variables, que responde a una pregunta muy práctica: por qué una variable que creaste afuera de una función no se puede usar adentro como si nada. La analogía es simple: lo que pasa en tu casa es asunto tuyo, pero en la calle hay reglas distintas. Las variables dentro de una función viven "en su casa" y no salen a la calle, y las de afuera no entran a la casa sin permiso. Entender esto te ahorra horas de frustración.

Lo que vas a ver en esta unidad, en orden:

- Qué es una función y como crearla: parámetros, argumentos y el famoso `return`.
- La diferencia entre `return` y `echo`, con la analogía de la máquina que te regresa algo contra la que solo lo anuncia por bocina.
- Valores por defecto en los parámetros y el alcance de las variables (local vs global).
- Arreglos indexados: crearlos, leerlos, contarlos con `count()`, agregar elementos con `[]`.
- Arreglos asociativos: el diccionario de PHP, llaves con nombre.
- `foreach`: como recorrer cualquier arreglo, con la sintaxis de la flecha doble `=>` explicada sin miedo.
- Arreglos multidimensionales: tablas como las de Excel, un arreglo dentro de otro.
- Funciones útiles de PHP para arreglos: `isset`, `empty`, `print_r`, `in_array`, `array_push`.
- Los errores más comunes (y como evitarlos): olvidar el `return`, usar variables globales sin declararlas, índices que no existen, confundir `=` con `==`.
- Buenas prácticas: una función, una tarea; nombres claros tipo `calcularTotal`.

Cuando termines esta unidad vas a poder escribir programas que se ven y se sienten profesionales: un carrito de compras que suma precios, una agenda de contactos, una tabla de

alumnos con calificaciones, un menu de navegación generado automáticamente. Todo eso sale de aquí. Manos a la obra.

Explicación detallada

Que es una función

Una función es un bloque de código al que le pones un nombre para poder usarlo muchas veces sin escribirlo de nuevo. Es como grabar una rutina: la defines una vez y la ejecutas cuando quieras.

La sintaxis es sencilla. Se escribe la palabra function, luego el nombre que tu elijas, luego paréntesis y luego llaves con las instrucciones adentro:

```
function saludar() {  
    echo "Hola, bienvenido a la academia";  
}  
  
saludar();  
saludar();
```

Fíjate en las dos partes. Primero la definición: function saludar() { ... }. Ahí solo estas describiendo que hace la función, pero todavia no pasa nada. Es cómo escribir la receta en una libreta: la receta existe, pero nadie ha cocinado. La segunda parte es la llamada: saludar();. Ahí si se ejecuta el código de adentro. Si llamas la función dos veces, el mensaje aparece dos veces. Si nunca la llamas, el código nunca se ejecuta.

Reglas para nombrar funciones:

- El nombre no puede empezar con un número.
- No puede tener espacios. Usa guion bajo si necesitas separar palabras: calcular_total.
- No distingue mayusculas de minusculas al llamarlas (Saludar()) funciona igual que saludar()), pero por costumbre se escriben en minusculas.
- El nombre debe describir lo que hace. saludar, calcularIVA, mostrarMenu son buenos nombres. cosa, hacer, funcion1 son malos nombres porque no dicen nada.

La primera vez que defines una función puede parecer trabajo extra: para que definir algo si puedo escribir el echo directamente. La ventaja aparece cuando ese mismo bloque se necesita en cinco, diez o cincuenta lugares de tu programa. Ahí es donde la función se vuelve tu mejor aliada.

Parámetros y argumentos

Volvamos a la licuadora. Una licuadora que siempre hiciera el mismo licuado de platano seria útil, pero una licuadora donde tu eliges la fruta es mucho mejor. Los parámetros son los ingredientes que la función acepta: se escriben dentro de los paréntesis al definirla. Los argumentos son los ingredientes reales que le pasas cuando la llamas.

```
function saludarPersona($nombre) {  
    echo "Hola " . $nombre . ", bienvenido";  
}  
  
saludarPersona("Maria");  
saludarPersona("Pedro");
```

Aquí \$nombre es el parámetro: es como el hueco vacío donde ira el ingrediente. "Maria" y "Pedro" son los argumentos: los ingredientes de verdad que le pasas en cada llamada. La primera llamada imprime "Hola Maria, bienvenido" y la segunda "Hola Pedro, bienvenido". La misma función, resultados distintos según lo que le des.

Una función puede recibir varios parámetros, separados por comas. El orden importa: el primer argumento llena el primer parámetro, el segundo llena el segundo, y así.

```
function presentar($nombre, $edad) {  
    echo $nombre . " tiene " . $edad . " anos";  
}  
  
presentar("Luis", 25);
```

Si llamas presentar("Luis", 25), \$nombre vale "Luis" y \$edad vale 25. Pero cuidado: si inviertes el orden y llamas presentar(25, "Luis"), PHP no se queja; simplemente pone 25 en \$nombre y "Luis" en \$edad, y el mensaje sale mal: "25 tiene Luis años". PHP confía en que tu pones los argumentos en el orden correcto. Consejo práctico: pon los parámetros en un orden que tenga lógica, como de lo general a lo específico, y manten ese orden siempre.

Tip importante: dentro de la función, los parámetros se comportan como variables normales. Puedes sumarlos, concatenarlos, meterlos en condicionales, lo que quieras.

return vs echo: la máquina que regresa contra la que anuncia

Este es uno de los temas que más confunde a los principiantes, así que vamos despacio con una analogía.

Imagina dos máquinas expendedoras. La máquina A tiene bocina: metes tu moneda, y la máquina ANUNCIA por la bocina "aquí esta tu refresco", pero no te da nada en las manos. La máquina B no tiene bocina: metes tu moneda y te ENTREGA el refresco en la mano, y tú decides que hacer con el: tomarlo, guardarlo en tu mochila o dárselo a alguien más.

En PHP, echo es la máquina A: muestra el resultado en pantalla y ahí se acaba la historia. Ese valor mostrado no lo puedes guardar en una variable ni usarlo para otra operación. return es la máquina B: te regresa el valor a las manos y tú decides que hacer con el.

Mira la diferencia:

```
function sumarConEcho($a, $b) {  
    echo $a + $b;  
}  
  
function sumarConReturn($a, $b) {  
    return $a + $b;  
}  
  
sumarConEcho(5, 3);  
$resultado = sumarConReturn(5, 3);  
echo "El doble es: " . ($resultado * 2);
```

sumarConEcho(5, 3) imprime 8 en pantalla y punto. No puedes hacer nada más con ese 8. En cambio, sumarConReturn(5, 3) te entrega el 8 "en la mano" y lo guardas en \$resultado. Después puedes multiplicarlo, sumarlo con otra cosa, meterlo en un if, guardarlo en una base de datos. Mucho más útil.

La regla de oro: si el resultado de la función lo vas a necesitar para seguir calculando o para tomar decisiones, usa return. Si lo único que quieres es mostrar algo en pantalla en ese momento, usa echo. Y ojo con algo: cuando PHP encuentra un return, la función termina ahí mismo. Cualquier línea después del return dentro de la función nunca se ejecuta. Es como la puerta de salida: una vez que sales, ya no regresas.

```
function ejemplo($n) {
    return $n * 2;
    echo "Esto nunca se muestra";
}
```

Ese echo de la última línea jamás se ejecuta. PHP sale de la función en cuanto ve el return.

Otro detalle: una función puede tener varios return en distintos caminos, por ejemplo dentro de un if. Eso está bien y es muy comun:

```
function esMayorDeEdad($edad) {
    if ($edad >= 18) {
        return true;
    }
    return false;
}
```

Valores por defecto en los parámetros

A veces quieres que un parámetro tenga un valor "de fabrica" por si quien llama a la función no le pasa nada. Eso se hace poniendo = valor después del parámetro en la definición.

```
function saludarPersona($nombre = "visitante") {
    echo "Hola " . $nombre;
}

saludarPersona("Ana");
saludarPersona();
```

La primera llamada imprime "Hola Ana". La segunda, como no le pasaste argumento, usa el valor por defecto y imprime "Hola visitante". Es como el restaurante donde si no dices que tomar, te traen agua natural: hay una opción predeterminada.

Regla: los parámetros con valor por defecto deben ir al final de la lista. Esto falla:

```
function mal($a = 10, $b) { }
```

PHP no sabría que hacer si llamas mal(5): el 5 iría a \$a o a \$b? Para evitar esa confusión, primero los parámetros obligatorios y al final los opcionales:

```
function bien($b, $a = 10) { }
```

Usa valores por defecto para cosas que casi siempre son iguales pero que a veces quieres cambiar: el número de decimales al mostrar un precio, el mensaje de bienvenida, el límite de una lista.

Alcance de variables: la casa y la calle

Aquí viene un concepto que al principio parece una molestia pero que en realidad es una protección. Se llama alcance (scope en inglés) y responde a la pregunta: donde puede usarse cada variable.

La analogía: tu casa y la calle. Dentro de tu casa puedes hacer lo que quieras: poner la música que te guste, caminar en pijama. En la calle hay reglas distintas y no puedes llevarte los muebles de tu casa a media avenida. Las variables dentro de una función viven en su casa: existen solo ahí adentro. Las variables de afuera viven en la calle: no entran a la casa sin permiso.

```
$nombre = "Carlos";

function intentar() {
    echo $nombre;
}

intentar();
```

Esto NO funciona como esperas. La función intentar() no ve la variable \$nombre de afuera. Para PHP, la \$nombre de adentro y la de afuera son dos cosas distintas que viven en mundos separados. Si ejecutas ese código, verás un aviso de variable no definida y no se imprime nada (o se imprime vacío).

Hay dos formas correctas de trabajar con esto. La primera y recomendada: pasa el dato como parámetro. Es como invitar a alguien a tu casa: entra por la puerta, con permiso.

```
$nombre = "Carlos";

function saludarBien($nombre) {
    echo "Hola " . $nombre;
}

saludarBien($nombre);
```

La segunda forma es la palabra `global`, que le dice a la función "usa la variable de afuera". Funciona, pero usala poco, porque hace tu código difícil de seguir: la función depende de algo que no se ve en su definición.

```
$contador = 0;

function aumentar() {
    global $contador;
    $contador = $contador + 1;
}

aumentar();
aumentar();
echo $contador;
```

Esto imprime 2. Sin la palabra `global`, la función crearia su propia `$contador` local que moriria al terminar la función, y la de afuera seguiria en 0.

También pasa al revés: una variable creada DENTRO de la función no existe afuera.

```
function crear() {
    $secreto = "1234";
}

crear();
echo $secreto;
```

Esto falla: `$secreto` murió cuando la función terminó. Es como una conversación dentro de tu casa: cuando sales a la calle, nadie afuera la escucho. Si necesitas ese valor afuera, la función debe regresarlo con `return`.

Por qué existe esta separación? Imagina un programa grande con veinte funciones. Si todas compartieran las mismas variables, cualquier función podría cambiar por accidente el valor que otra necesita. Seria un caos. El alcance local protege tu código: cada función trabaja con sus propias variables sin pisar las de las demás. Acostumbrate a pensar así: la función recibe datos por parámetros y entrega resultados con `return`. Esa es la forma limpia y profesional.

Arreglos indexados: los casilleros numerados

Un arreglo es una variable que guarda varios valores juntos en lugar de uno solo. El arreglo indexado es el más simple: es como una fila de casilleros de gimnasio numerados. Cada casillero tiene un número (el índice) y dentro hay algo guardado (el valor). En PHP los

casilleros se numeran empezando desde el 0, no desde el 1. Esto al principio se siente raro, pero es la convención en casi todos los lenguajes de programación.

Hay varias formas de crear un arreglo. La moderna usa corchetes:

```
$frutas = ["manzana", "platano", "naranja"];
```

La clasica usa la palabra array (hace exactamente lo mismo):

```
$frutas = array("manzana", "platano", "naranja");
```

Las dos formas son validas. En este libro usaremos los corchetes porque es la forma moderna y más corta, pero si ves array() en código de otras personas, ya sabes qué es lo mismo.

En ese ejemplo PHP asigno los números automáticamente: "manzana" quedó en el casillero 0, "platano" en el 1 y "naranja" en el 2. Para leer un valor escribes el nombre del arreglo y el número entre corchetes:

```
echo $frutas[0];  
echo $frutas[2];
```

El primero imprime "manzana" y el segundo "naranja". El número entre corchetes se llama índice. Recuerda: el primer elemento es el 0. Si intentas leer \$frutas[3] y no existe ese casillero, PHP te muestra un aviso de índice no definido. Más adelante veremos cómo evitar ese error con isset.

También puedes crear el arreglo casillero por casillero:

```
$colores = [];  
$colores[0] = "rojo";  
$colores[1] = "azul";  
$colores[2] = "verde";
```

Primero creas un arreglo vacío con [] y luego llenas cada posición. Y puedes cambiar el contenido de un casillero en cualquier momento:

```
$frutas[1] = "mango";
```

Ahora el casillero 1 ya no dice "platano", dice "mango". Los arreglos no son de piedra: puedes modificarlos cuando quieras.

count(): cuantos casilleros hay

Para saber cuantos elementos tiene un arreglo se usa la función count(). Es como contar los casilleros ocupados del gimnasio.

```
$frutas = ["manzana", "platano", "naranja"];  
echo count($frutas);
```

Esto imprime 3. Ojo: count() te dice cuantos elementos hay, pero el último índice es uno menos, porque se empieza en 0. Si hay 3 elementos, los índices son 0, 1 y 2. Esta diferencia es la causa del error más común al recorrer arreglos con for, como verás en un momento.

Agregar elementos con []

Para agregar un elemento al final del arreglo, usas corchetes vacíos. PHP solo pone el valor en el siguiente casillero libre.

```
$frutas = ["manzana", "platano"];  
$frutas[] = "naranja";  
$frutas[] = "uva";
```

Después de esas dos líneas, el arreglo tiene cuatro elementos: manzana (0), platano (1), naranja (2), uva (3). No necesitas calcular tu el número: PHP lo hace por ti. Esta es la forma más común de ir construyendo una lista poco a poco, por ejemplo cuando lees datos de un formulario o de una base de datos.

Un truco útil: puedes mezclar tipos de datos en el mismo arreglo. PHP no se queja si guardas textos, números y booleanos juntos:

```
$mezcla = ["Juan", 25, true, 1.75];
```

Esto es válido. En la práctica, los arreglos suelen guardar datos del mismo tipo (una lista de nombres, una lista de precios), pero el lenguaje te da libertad total.

Recorrer un arreglo con for

Recorrer significa pasar por cada elemento, uno por uno, para hacer algo con él. Con un arreglo indexado puedes usar el ciclo for que ya conoces, porque los índices son números seguidos.

```
$frutas = ["manzana", "platano", "naranja"];

for ($i = 0; $i < count($frutas); $i++) {
    echo "Fruta numero " . $i . ": " . $frutas[$i] . "<br>";
}
```

Vamos línea por línea. \$i empieza en 0 porque el primer casillero es el 0. La condición es \$i < count(\$frutas), o sea \$i < 3. Fíjate que es menor que (<), no menor o igual (<=). Si pusieras <=, en la última vuelta \$i valdría 3 e intentarías leer \$frutas[3], que no existe. Ese es el famoso error de "uno de más" (off by one) que a todos nos pasa al principio. Memoriza la fórmula: for (\$i = 0; \$i < count(\$arreglo); \$i++). Dentro del ciclo, \$frutas[\$i] lee el casillero número \$i, que va cambiando en cada vuelta: 0, 1, 2.

El resultado sería:

Fruta número 0: manzana Fruta número 1: platano Fruta número 2: naranja

(El "
" es una etiqueta HTML para salto de línea; si pruebas el código en el navegador verás cada fruta en su renglón.)

Este patrón de for con count() lo vas a ver miles de veces en código PHP. Aprendetelo de memoria.

Arreglos asociativos: el diccionario

El arreglo indexado usa números como llave de cada casillero. Pero en la vida real casi nunca buscamos las cosas por número: en tu agenda del teléfono no buscas "el contacto 47", buscas "Maria". En un diccionario no buscas "la palabra número 1200", buscas la palabra "casa". El arreglo asociativo funciona así: en vez de números, cada valor tiene una llave con nombre (un texto).

```
$persona = [
    "nombre" => "Maria",
    "edad" => 28,
    "ciudad" => "Guadalajara"
];
```

Fíjate en la flecha doble =>. Se lee como "apunta a" o "vale". "nombre" => "Maria" significa: la llave "nombre" apunta al valor "Maria". La llave es como la etiqueta del casillero: en vez de un número frío, tiene un nombre que describe lo que guarda.

Para leer un valor usas la llave entre corchetes, igual que con los índices numéricos:

```
echo $persona["nombre"];  
echo $persona["edad"];
```

Esto imprime "Maria" y 28. Para cambiar un valor o agregar una llave nueva es igual de fácil:

```
$persona["edad"] = 29;  
$persona["telefono"] = "5551234567";
```

La primera línea cambia la edad de 28 a 29. La segunda agrega una llave nueva "teléfono" que antes no existía. PHP no distingue entre "cambiar" y "agregar": si la llave existe la actualiza, si no existe la crea.

Cuando conviene cada tipo? Usa arreglo indexado cuando el orden importa y los datos son del mismo tipo: una lista de productos, los días de la semana, las calificaciones de un alumno. Usa arreglo asociativo cuando cada dato tiene un significado distinto y quieres encontrarlo por su nombre: los datos de una persona, la configuración de tu sitio, un producto con su precio y su stock.

Puedes mezclar llaves con nombre y números en el mismo arreglo, aunque en la práctica casi no se hace porque confunde. Lo importante es que entiendas que la llave puede ser texto o número.

foreach: el recorrido automatico

Recorrer un arreglo asociativo con for es un dolor de cabeza, porque las llaves son textos y no números seguidos. Para eso existe foreach, un ciclo diseñado especialmente para arreglos. Es como pasar tu dedo por cada casillero: PHP abre cada uno y te entrega su contenido, sin que tengas que saber cuantos hay ni cómo se llaman.

La forma más simple:

```
$frutas = ["manzana", "platano", "naranja"];  
  
foreach ($frutas as $fruta) {  
    echo "Me gusta la " . $fruta . "<br>";  
}
```

Se lee así: "para cada elemento de \$frutas, guardalo en \$fruta y ejecuta este bloque". En cada vuelta, \$fruta vale algo distinto: primero "manzana", luego "platano", luego "naranja". Tu no manejas ningun contador; PHP se encarga de todo. Es más corto, más claro y sin riesgo del error de "uno de más".

La forma completa te entrega también la llave de cada elemento, usando la flecha doble ==>:

```
$persona = [  
    "nombre" => "Maria",  
    "edad" => 28,  
    "ciudad" => "Guadalajara"  
];  
  
foreach ($persona as $llave => $valor) {  
    echo $llave . ": " . $valor . "<br>";  
}
```

Se lee: "para cada elemento de \$persona, guarda su llave en \$llave y su valor en \$valor". El resultado:

nombre: Maria edad: 28 ciudad: Guadalajara

La sintaxis \$llave ==> \$valor dentro del foreach es la misma flecha que usaste al crear el arreglo, y significa lo mismo: la llave apunta al valor. Piensalo como una balanza: a la izquierda lo que identifica, a la derecha lo que contiene.

Con arreglos indexados también puedes pedir la llave, y te dará los números 0, 1, 2:

```
$frutas = ["manzana", "platano", "naranja"];  
  
foreach ($frutas as $indice => $fruta) {  
    echo "Casillero " . $indice . " tiene " . $fruta . "<br>";  
}
```

Consejo práctico: cuando solo necesitas los valores, usa la forma simple (foreach (\$arreglo as \$valor)). Cuando necesitas saber la llave o el índice, usa la forma completa (foreach (\$arreglo as \$llave => \$valor)). El nombre de las variables \$llave y \$valor lo eliges tu; lo importante es el orden: primero la llave, después el valor.

Un error común es intentar modificar el arreglo dentro del foreach esperando que el cambio se guarde. Por defecto, foreach trabaja con una COPIA de cada valor. Si haces esto, no pasa nada en el arreglo original:

```
$precios = [100, 200, 300];

foreach ($precios as $precio) {
    $precio = $precio * 2;
}
```

Después del ciclo, \$precios sigue valiendo [100, 200, 300]. La variable \$precio era una copia desechable. Si de verdad necesitas modificar los valores dentro del ciclo, usa el índice:

```
for ($i = 0; $i < count($precios); $i++) {
    $precios[$i] = $precios[$i] * 2;
}
```

Así si se guardan los cambios, porque estas escribiendo directamente en el casillero.

Arreglos multidimensionales: la tabla de Excel

Un arreglo multidimensional es un arreglo que guarda otros arreglos adentro. La analogía perfecta es una tabla de Excel: tienes filas y columnas. Cada fila es un arreglo, y la tabla completa es un arreglo de filas.

El ejemplo clásico: una lista de alumnos donde cada alumno tiene nombre y calificación.

```
$alumnos = [
    ["nombre" => "Ana", "calificacion" => 9],
    ["nombre" => "Luis", "calificacion" => 7],
    ["nombre" => "Sofia", "calificacion" => 10]
];
```

\$alumnos es un arreglo indexado con 3 elementos (índices 0, 1, 2). Cada elemento es a su vez un arreglo asociativo con las llaves "nombre" y "calificación". Para leer un dato específico usas dos pares de corchetes: el primero elige la fila, el segundo la columna.

```
echo $alumnos[0]["nombre"];
echo $alumnos[2]["calificacion"];
```

El primero imprime "Ana" (fila 0, columna "nombre"). El segundo imprime 10 (fila 2, columna "calificación"). Leelo de izquierda a derecha: "del arreglo \$alumnos, dame la fila 0, y de ahí la llave nombre".

Para recorrer una tabla así se usan dos foreach anidados, o más comunmente un foreach para las filas y acceso directo a las columnas:

```
foreach ($alumnos as $alumno) {  
    echo $alumno["nombre"] . " saco " . $alumno["calificacion"] . "<br>";  
}
```

En cada vuelta, \$alumno es una fila completa (un arreglo asociativo). Dentro del ciclo lo tratas como cualquier arreglo asociativo: \$alumno["nombre"], \$alumno["calificación"].

Las tablas también pueden ser de puros números, como un tablero:

```
$tablero = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
];  
  
echo $tablero[1][2];
```

Esto imprime 6: fila 1 (la segunda), columna 2 (la tercera). Es el mismo principio.

Donde vas a usar esto en la vida real? En todos lados. Cuando consultes una base de datos, cada fila de resultados llega como un arreglo y el conjunto de filas como un arreglo de arreglos. Un carrito de compras es una lista donde cada producto tiene nombre, precio y cantidad. Un menu de navegación es una lista donde cada opción tiene texto y enlace. Dominar los arreglos multidimensionales es dominar la forma en que PHP maneja datos reales.

Funciones utiles para trabajar con arreglos

PHP trae muchas funciones listas para usar con arreglos. Aquí están las cinco que más vas a necesitar. Son como las herramientas basicas de una caja de herramientas: con estas cinco resuelves el 90 por ciento de los casos.

isset: pregunta si una variable existe y no es nula. Es tu detector de casilleros vacios. Antes de leer \$persona["teléfono"], pregunta si esa llave existe para evitar el aviso de índice no definido.

```
$persona = ["nombre" => "Maria"];

if (isset($persona["telefono"])) {
    echo $persona["telefono"];
} else {
    echo "No tiene telefono registrado";
}
```

Como "teléfono" no existe en el arreglo, `isset` regresa `false` y se imprime "No tiene teléfono registrado". Sin `isset`, PHP mostraría un aviso feo. Acostumbrate: siempre que leas una llave que podría no existir (datos de un formulario, por ejemplo), envuélvela en `isset`. Es una de las funciones que más vas a escribir en tu vida como programador PHP.

`empty`: pregunta si una variable está vacía. Considera vacíos estos valores: "" (texto vacío), 0, "0", null, `false` y el arreglo vacío []. Es útil para validar formularios: si el usuario dejó un campo en blanco, `empty` lo detecta.

```
$nombre = "";

if (empty($nombre)) {
    echo "Tienes que escribir tu nombre";
}
```

Cuidado con la diferencia: `isset` pregunta "existe?", `empty` pregunta "tiene algo útil adentro?". Una variable puede existir y estar vacía al mismo tiempo. Ejemplo: `$x = ""` existe (`isset` dice `true`) pero está vacía (`empty` dice `true`). Y una variable que no existe: `isset` dice `false` y `empty` también dice `true` (no da error, simplemente la considera vacía).

`print_r`: muestra el contenido completo de un arreglo de forma legible. Es tu rayos X para depurar. Cuando un arreglo no se comporta como esperas, lo primero que haces es mirarlo por dentro con `print_r`.

```
$frutas = ["manzana", "platano", "naranja"];
print_r($frutas);
```

En el navegador verás algo así:

```
Array ( [0] => manzana [1] => platano [2] => naranja )
```

No es bonito para el usuario final, es una herramienta para ti mientras programas. Cuando algo falle, antes de adivinar, imprime el arreglo con `print_r` y mira que tiene adentro de verdad. Te ahorra horas. Un truco: si lo usas dentro de etiquetas `<pre>` se ve mucho más ordenado:

```
echo "<pre>";
print_r($frutas);
echo "</pre>";
```

`in_array`: pregunta si un valor existe dentro de un arreglo. Regresa true o false. Es como preguntar "hay mango en la frutería?".

```
$frutas = ["manzana", "platano", "naranja"];

if (in_array("platano", $frutas)) {
    echo "Si hay platano";
}

if (in_array("mango", $frutas)) {
    echo "Si hay mango";
} else {
    echo "No hay mango";
}
```

El primer if se cumple, el segundo no. El orden de los parámetros es: primero lo que buscas, después el arreglo donde buscas. Memorízalo así: `in_array`(aguja, pajar) - la aguja primero, el pajar después.

`array_push`: agrega uno o más elementos al final de un arreglo. Hace lo mismo que los corchetes vacíos [], pero puede agregar varios valores de un solo golpe.

```
$frutas = ["manzana"];
array_push($frutas, "platano", "naranja");
```

Ahora `$frutas` tiene los tres elementos. Con [] necesitarías dos líneas; con `array_push` lo haces en una. Para agregar un solo elemento, la mayoría de los programadores prefiere `$frutas[] = "platano"` por ser más corto. Usa `array_push` cuando agregues varios a la vez.

Errores comunes (y como evitarlos)

Estos son los cuatro errores que todo principiante comete con funciones y arreglos. Los pongo aquí para que cuando te pasen (porque te van a pasar), los reconozcas en segundos en vez de sufrir una hora.

Error 1: olvidar el return. Defines una función que calcula algo, pero se te olvida regresarlo. La función hace el cálculo por dentro y luego... no entrega nada. Quien la llama recibe null (nada).


```
function calcularDoble($n) {  
    $resultado = $n * 2;  
}  
  
$doble = calcularDoble(5);  
echo $doble;
```

Esto no imprime 10. Imprime nada, porque la función nunca hizo return. El calculo se hizo y se tiro a la basura. La versión correcta:

```
function calcularDoble($n) {  
    return $n * 2;  
}
```

Cada vez que una función deba entregar un valor, verifica que tenga su return. Una buena costumbre: escribe el return primero y luego el calculo, así nunca se te olvida.

Error 2: usar una variable global sin declararla. Ya lo viste en la sección de alcance: las variables de afuera no entran solas a la función. Si necesitas un valor de afuera, pasalo como parámetro. Si de plano necesitas la global, escribe global \$variable al inicio de la función. Pero repito el consejo: los parámetros son el camino limpio.

```
$iva = 0.16;  
  
function precioConIvaMal($precio) {  
    return $precio * (1 + $iva);  
}
```

Esto falla porque \$iva no existe dentro de la función. La versión correcta pasa el iva como parámetro:

```
function precioConIva($precio, $iva = 0.16) {  
    return $precio * (1 + $iva);  
}
```

Así la función es independiente y hasta puedes calcular con otro iva si algun día cambia.

Error 3: leer un índice o llave que no existe. Pasa mucho con arreglos asociativos: escribes mal la llave o supones que existe un dato que no llegó.

```
$persona = ["nombre" => "Maria"];  
echo $persona["nomber"];
```

"nomber" está mal escrito (le falta la r). PHP muestra un aviso de índice no definido. La defensa es isset antes de leer llaves dudosas, y revisar con print_r que el arreglo tenga lo que crees que tiene. Cuando veas el aviso "Undefined array key", ya sabes: estas leyendo una llave que no existe. Revisa la ortografía de la llave.

Error 4: confundir = con == dentro de funciones (y en general). El signo = asigna un valor, el doble == compara. Dentro de un if, usar = por accidente siempre da true y es un error silencioso y traicionero.

```
function esAdmin($rol) {  
    if ($rol = "admin") {  
        return true;  
    }  
    return false;  
}
```

Ese if no esta preguntando si \$rol es "admin": le esta ASIGNANDO "admin" a \$rol, y una asignación siempre se considera verdadera. La función regresa true para cualquier rol. La versión correcta usa ==:

```
function esAdmin($rol) {  
    if ($rol == "admin") {  
        return true;  
    }  
    return false;  
}
```

Si tu programa "siempre entra al if" aunque no debería, revisa primero si escribiste = en vez de ==. Es el error más comun de todos los lenguajes, no solo de PHP.

Buenas prácticas

Una función, una tarea. Cada función debe hacer una sola cosa y hacerla bien. Si tu función se llama calcularTotalYGuardarEnBaseDeDatosYEnviarCorreo, tienes un problema: esta haciendo tres cosas. Dividela en tres funciones: calcularTotal(), guardarEnBaseDeDatos(), enviarCorreo(). Así cada una se puede probar, entender y reutilizar por separado. Las funciones largas que hacen de todo son dificiles de leer y un infierno de mantener.

Nombres verbo + sustantivo. El nombre de la función debe decir que hace, empezando con un verbo: `calcularTotal`, `obtenerNombre`, `validarCorreo`, `mostrarMenu`, `guardarUsuario`. Cuando leas `calcularTotal($precios)` en tu código, entiendes al instante que calcula el total de esos precios, sin necesidad de ver que hay adentro. Evita nombres vagos como `proceso()`, `datos()`, `hacer()`. Tu yo del futuro (y tus compañeros) te lo van a agradecer.

No repitas código: si lo escribes dos veces, hazlo función. Esta es la regla de oro de esta unidad. La primera vez que copies y pegues un bloque de código, detente y conviértelo en función. El código duplicado es una bomba de tiempo: cuando haya que cambiarlo, tendrás que acordarte de todos los lugares donde lo pegaste, y siempre se te olvida uno.

Las funciones no deben depender de variables globales. Una buena función es como una máquina autocontenida: todo lo que necesita entra por los parámetros, todo lo que produce sale por el `return`. Si tu función solo funciona cuando existe cierta variable global afuera, es frágil: no la puedes reutilizar en otro proyecto ni probarla por separado.

Comenta lo necesario, no lo obvio. Un comentario útil explica el *porqué*, no el *QUE*:

```
function descuento($precio) {  
    return $precio * 0.9;  
}
```

No necesitas un comentario que diga "calcula el descuento". En cambio, si hay una razón de negocio, explícala:

```
function descuento($precio) {  
    return $precio * 0.9;  
}
```

Esa función está clara por sí sola. Reserva los comentarios para decisiones extrañas, formulas raras o advertencias.

Empieza simple y crece. No intentes escribir la función perfecta desde el primer intento. Primero haz que funcione con un caso sencillo, pruebala, y luego agrega casos. La programación es iterativa: versión 1 que funciona, versión 2 que funciona mejor.

Ejemplos paso a paso

Ejemplo 1: función saludar

El ejemplo más simple posible: una función sin parámetros que muestra un mensaje. Sirve para entender la estructura básica de definir y llamar.

```
function saludar() {  
    echo "Hola, bienvenido a Lucas EGA Academy<br>";  
}  
  
saludar();  
saludar();
```

Línea por línea:

- `function saludar() {`: la palabra `function` le dice a PHP que vamos a definir una función. `saludar` es el nombre que elegimos. Los paréntesis vacíos indican que no recibe parámetros. La llave `{` abre el cuerpo de la función.
- `echo "Hola, bienvenido a Lucas EGA Academy<br>";`: la única instrucción de la función. Muestra el mensaje en pantalla. El `<br>` es un salto de línea en HTML.
- `}`: cierra el cuerpo de la función. Hasta aquí solo la definimos; todavía no se ejecutó nada.
- Línea en blanco: separación visual, no afecta al programa.
- `saludar();`: primera llamada. Ahora sí se ejecuta el código de adentro y se imprime el mensaje una vez.
- `saludar();`: segunda llamada. Se imprime el mensaje otra vez. Con dos llamadas reutilizamos la misma definición.

Si mañana quieres cambiar el mensaje, lo cambias en un solo lugar (dentro de la función) y las dos llamadas muestran el texto nuevo. Esa es la magia.

Ejemplo 2: función con parámetros y return para calcular IVA

Ahora algo útil de verdad: una función que recibe el precio de un producto y regresa el precio con IVA incluido, usando `return` para que el resultado se pueda seguir usando.

```
function calcularPrecioConIva($precio, $iva = 0.16) {
    $total = $precio * (1 + $iva);
    return $total;
}

$precioPlayera = 299;
$precioFinal = calcularPrecioConIva($precioPlayera);

echo "Precio sin IVA: $" . $precioPlayera . "<br>";
echo "Precio con IVA: $" . $precioFinal . "<br>";
echo "Dos playeras cuestan: $" . ($precioFinal * 2) . "<br>";
```

Línea por línea:

- `function calcularPrecioConIva($precio, $iva = 0.16) {`: definimos la función con dos parámetros. `$precio` es obligatorio: cada llamada debe dar el precio. `$iva` es opcional con valor por defecto 0.16 (el 16 por ciento), por si no se especifica otro.
- `$total = $precio * (1 + $iva);`: el calculo. Si el precio es 299 y el iva 0.16, esto es $299 * 1.16 = 346.84$. Se guarda en la variable local `$total`, que solo existe dentro de la función.
- `return $total;`: entregamos el resultado a quien llamó la función. La función termina aquí.
- `}`: cierra la definición.
- `$precioPlayera = 299;`: variable normal fuera de la función, con el precio del producto.
- `$precioFinal = calcularPrecioConIva($precioPlayera);`: llamamos la función pasándole 299. Como no dimos el segundo argumento, usa el iva por defecto 0.16. El valor regresado (346.84) se guarda en `$precioFinal`.
- Los tres `echo` muestran: el precio original, el precio con IVA, y el precio de dos playeras usando el resultado guardado. Fíjate que esto último solo es posible porque usamos `return`: el valor quedó en nuestras manos y pudimos multiplicarlo por 2.

Prueba mental: si el IVA cambiara a 0.15, solo tocarías el valor por defecto en la definición y todo el programa quedaría actualizado. Sin función, buscarías cada multiplicación por 1.16 en tu código.

Ejemplo 3: lista de compras con arreglo indexado

Un arreglo indexado guardando los productos de una lista de compras, mostrando como crearlo, agregar elementos y contarlos.

```

$compras = ["tortillas", "huevos", "leche"];

$compras[] = "pan";
$compras[] = "queso";

echo "Tienes " . count($compras) . " productos en la lista:<br>";

for ($i = 0; $i < count($compras); $i++) {
    echo ($i + 1) . ". " . $compras[$i] . "<br>";
}

```

Línea por línea:

- `$compras = ["tortillas", "huevos", "leche"];`: creamos el arreglo con tres productos. Quedan en los índices 0, 1 y 2.
- `$compras[] = "pan";`: agregamos "pan" al final. PHP lo pone en el índice 3 automáticamente.
- `$compras[] = "queso";`: agregamos "queso". Queda en el índice 4.
- `echo "Tienes " . count($compras) . " productos en la lista:<br>";`: `count()` regresa 5, así que se imprime "Tienes 5 productos en la lista:".
- `for ($i = 0; $i < count($compras); $i++) {`: el ciclo clásico. `$i` empieza en 0, se repite mientras `$i` sea menor que 5, y aumenta de uno en uno.
- `echo ($i + 1) . ". " . $compras[$i] . "<br>;`: mostramos el número de producto empezando en 1 (por eso `$i + 1`, más amable para el usuario) y el valor del casillero `$i`.
- `}`: cierra el `for`.

El resultado muestra una lista numerada del 1 al 5. Observa el detalle de `($i + 1)`: por dentro PHP cuenta desde 0, pero al usuario le mostramos desde 1 porque así lo esperan las personas normales.

Ejemplo 4: recorrer con `for` y con `foreach`

El mismo arreglo recorrido de las dos formas, para que compares y sientas la diferencia. Spoiler: `foreach` es más comodo.

```

$alumnos = ["Ana", "Luis", "Sofia", "Pedro"];

echo "Con for:<br>";
for ($i = 0; $i < count($alumnos); $i++) {
    echo "- " . $alumnos[$i] . "<br>";
}

echo "Con foreach:<br>";
foreach ($alumnos as $alumno) {
    echo "- " . $alumno . "<br>";
}

```

Línea por línea:

- \$alumnos = ["Ana", "Luis", "Sofia", "Pedro"];: arreglo indexado con cuatro nombres.
- echo "Con for:
";: título de la primera versión.
- for (\$i = 0; \$i < count(\$alumnos); \$i++) {: ciclo clásico con contador. Necesitas acordarte de empezar en 0, usar < (no <=) y aumentar \$i.
- echo "- " . \$alumnos[\$i] . "
";: lees el casillero número \$i en cada vuelta.
- } : cierra el for.
- Línea en blanco: separación.
- echo "Con foreach:
";: título de la segunda versión.
- foreach (\$alumnos as \$alumno) {: por cada elemento del arreglo, guardalo en \$alumno. Sin contadores, sin índices, sin count().
- echo "- " . \$alumno . "
";: \$alumno ya trae el valor listo para usar.
- } : cierra el foreach.

Las dos versiones imprimen exactamente lo mismo. La versión con for tiene más piezas que pueden fallar (el límite, el incremento). La versión con foreach es más corta y más difícil de romper. Por eso, cuando solo necesitas los valores, foreach es la opción recomendada.

Ejemplo 5: agenda con arreglo asociativo

Una agenda de contactos donde cada persona se busca por su nombre (la llave) en vez de por un número. El ejemplo perfecto del poder de los arreglos asociativos.

```

$agenda = [
    "Maria" => "555-1234",
    "Pedro" => "555-5678",
    "Ana" => "555-9012"
];

$agenda["Luis"] = "555-3456";

$buscar = "Pedro";

if (isset($agenda[$buscar])) {
    echo "El telefono de " . $buscar . " es " . $agenda[$buscar];
} else {
    echo "No encuentre a " . $buscar . " en la agenda";
}

```

Línea por línea:

- `$agenda = [ ... ];`: creamos el arreglo asociativo con tres contactos. Cada nombre (llave) apunta a un teléfono (valor) usando la flecha `=>`.
- `"Maria" => "555-1234";`: la llave "Maria" guarda el teléfono "555-1234". Y así con los demás. La última línea no lleva coma (aunque PHP la permite, es más limpio sin ella).
- `$agenda["Luis"] = "555-3456";`: agregamos un contacto nuevo. Como la llave "Luis" no existía, PHP la crea.
- `$buscar = "Pedro";`: el nombre que queremos buscar. En un programa real vendría de un formulario o de la URL.
- `if (isset($agenda[$buscar])) {`: preguntamos si la llave "Pedro" existe en la agenda. Esta es la protección contra el error de índice no definido.
- `echo "El teléfono de " . $buscar . " es " . $agenda[$buscar];`: si existe, mostramos el teléfono guardado en esa llave.
- `} else {`: si no existe...
- `echo "No encuentre a " . $buscar . " en la agenda";`: mensaje amable en vez de un error feo.
- `}`: cierra el if.

Cambia `$buscar` a "Rosa" y verás el mensaje de "no encuentre". Este patrón (arreglo asociativo + `isset`) es exactamente cómo funcionan las sesiones de usuario y los datos de formularios en PHP. Lo usaras toda tu vida.

Ejemplo 6: tabla de alumnos con arreglo multidimensional

Una tabla donde cada fila es un alumno con nombre y tres calificaciones. Combinamos arreglo multidimensional, foreach y una función que calcula el promedio.

```
function calcularPromedio($calificaciones) {
    $suma = 0;
    foreach ($calificaciones as $cal) {
        $suma = $suma + $cal;
    }
    return $suma / count($calificaciones);
}

$alumnos = [
    ["nombre" => "Ana", "califs" => [9, 8, 10]],
    ["nombre" => "Luis", "califs" => [7, 6, 8]],
    ["nombre" => "Sofia", "califs" => [10, 10, 9]]
];

foreach ($alumnos as $alumno) {
    $prom = calcularPromedio($alumno["califs"]);
    echo $alumno["nombre"] . " - Promedio: " . $prom . "<br>";
}
```

Línea por línea:

- function calcularPromedio(\$calificaciones) {: la función recibe un arreglo de calificaciones. Fíjate: los parámetros también pueden ser arreglos, no solo números o textos.
- \$suma = 0;; acumulador en cero. Aquí iremos sumando cada calificación.
- foreach (\$calificaciones as \$cal) {: recorreremos cada calificación del arreglo recibido.
- \$suma = \$suma + \$cal;; sumamos la calificación actual al acumulador.
- }; cierra el foreach.
- return \$suma / count(\$calificaciones);: dividimos la suma entre la cantidad de calificaciones. count() nos dice cuantas hay, así la función sirve para 3, 5 o 20 calificaciones.
- }; cierra la función.
- Línea en blanco.

- `$alumnos = [ ... ]`; la tabla. Es un arreglo indexado de 3 filas; cada fila es un arreglo asociativo con "nombre" y "califs". Y "califs" es a su vez un arreglo indexado de números. Tres niveles: tabla > fila > lista de calificaciones.
- `foreach ($alumnos as $alumno) {`: recorremos fila por fila. En cada vuelta `$alumno` es un arreglo asociativo.
- `$prom = calcularPromedio($alumno["califs"]);`: tomamos la lista de calificaciones de esa fila (`$alumno["califs"]`) y se la pasamos a la función. El promedio regresado se guarda en `$prom`.
- `echo $alumno["nombre"] . " - Promedio: " . $prom . "<br>;`: mostramos el nombre y su promedio.
- `}`: cierra el `foreach`.

El resultado: Ana - Promedio: 9, Luis - Promedio: 7, Sofia - Promedio: 9.67 (aprox). Observa como la función `calcularPromedio` no sabe nada de alumnos ni de nombres: solo sabe promediar listas de números. Esa independencia es lo que la hace reutilizable. Mañana la puedes usar para promediar precios, edades o temperaturas sin cambiarle nada.

Ejemplo 7: carrito de compras que suma con función

El ejemplo más cercano a una tienda real: un carrito donde cada producto tiene nombre, precio y cantidad, y una función que calcula el total. Junta todo lo visto: arreglos multidimensionales, `foreach`, funciones con `return` y parámetros.

```

function calcularTotalCarrito($carrito) {
    $total = 0;
    foreach ($carrito as $producto) {
        $subtotal = $producto["precio"] * $producto["cantidad"];
        $total = $total + $subtotal;
    }
    return $total;
}

$carrito = [
    ["nombre" => "Playera", "precio" => 299, "cantidad" => 2],
    ["nombre" => "Gorra", "precio" => 149, "cantidad" => 1],
    ["nombre" => "Tenis", "precio" => 899, "cantidad" => 1]
];

foreach ($carrito as $producto) {
    $sub = $producto["precio"] * $producto["cantidad"];
    echo $producto["nombre"] . " x" . $producto["cantidad"] . " = $" . $sub .
    "<br>";
}

$total = calcularTotalCarrito($carrito);
echo "TOTAL A PAGAR: $" . $total;

```

Línea por línea:

- function calcularTotalCarrito(\$carrito) {: la función recibe todo el carrito (un arreglo de productos).
- \$total = 0;: acumulador del total general.
- foreach (\$carrito as \$producto) {: recorreremos producto por producto.
- \$subtotal = \$producto["precio"] * \$producto["cantidad"];: el subtotal de esa línea: precio por cantidad. Para la playera: $299 * 2 = 598$.
- \$total = \$total + \$subtotal;: sumamos ese subtotal al total general.
- } : cierra el foreach.
- return \$total;: entregamos el total. La función termina.
- } : cierra la función.
- \$carrito = [...] : el carrito con tres productos, cada uno con nombre, precio y cantidad.
- foreach (\$carrito as \$producto) {: primer recorrido: mostramos el detalle línea por línea.

- `$sub = $producto["precio"] * $producto["cantidad"];` calculamos el subtotal para mostrarlo.
- `echo ...;` imprimimos algo como "Playera x2 = \$598".
- `};` cierra el foreach.
- `$total = calcularTotalCarrito($carrito);` llamamos la función con todo el carrito y guardamos el resultado.
- `echo "TOTAL A PAGAR: $" . $total;` mostramos el total: $598 + 149 + 899 = 1646$.

Este es el esqueleto de cualquier tienda en línea. En la vida real los productos vendrían de una base de datos y el carrito de la sesión del usuario, pero la lógica es exactamente esta: recorrer, multiplicar precio por cantidad, acumular. Si entiendes este ejemplo al cien, entiendes el corazón del comercio electrónico.

Ejemplo 8: menú de navegación generado con foreach

En vez de escribir a mano cada enlace del menú de tu sitio, guardas las opciones en un arreglo asociativo y PHP genera el HTML solo. Si mañana agregas una sección, solo agregas una línea al arreglo.

```
$menu = [
    "Inicio" => "index.php",
    "Cursos" => "cursos.php",
    "Blog" => "blog.php",
    "Contacto" => "contacto.php"
];

echo "<ul>";
foreach ($menu as $texto => $enlace) {
    echo "<li><a href=" . $enlace . ">" . $texto . "</a></li>";
}
echo "</ul>";
```

Línea por línea:

- `$menu = [ ... ];` el arreglo asociativo. La llave es el texto que verá el usuario ("Inicio"); el valor es el archivo al que apunta ("index.php").
- `echo "<ul>";` abrimos una lista HTML desordenada.

- `foreach ($menu as $texto => $enlace) {`: recorremos el arreglo pidiendo llave y valor. En cada vuelta `$texto` trae "Inicio", "Cursos", etc., y `$enlace` trae "index.php", "cursos.php", etc. Aquí la flecha `=>` dentro del `foreach` separa la llave del valor, igual que al crear el arreglo.
- `echo "<li><a href=" . $enlace . ">" . $texto . "</a></li>";`: armamos cada elemento de la lista. Fíjate como mezclamos HTML con variables PHP usando el punto de concatenación. Para "Inicio" genera: `<li><a href=index.php>Inicio</a></li>`. (En HTML real las comillas del href irían con comillas dobles escapadas o simples; aquí lo simplificamos para enfocarnos en el `foreach`.)
- `}`: cierra el `foreach`.
- `echo "</ul>";`: cerramos la lista.

El navegador muestra una lista con cuatro enlaces. La belleza está en el mantenimiento: para agregar la sección "Tienda" solo escribes una línea más en el arreglo: "Tienda" => "tienda.php", y el menu se actualiza solo. Sin arreglo tendrías que copiar y pegar HTML y rezar por no equivocarte en un enlace. Este patrón (arreglo de datos + `foreach` que genera HTML) es la base de cómo se construyen páginas dinámicas en PHP.

Práctica

Ejercicio 1

Escribe una función llamada `esPar` que reciba un número y regrese `true` si el número es par y `false` si es impar. Después usa esa función para revisar los números del 1 al 10 e imprimir cuáles son pares. Pista: un número es par si el resto de dividirlo entre 2 es cero; en PHP el resto se obtiene con el operador `%`.

Solución 1

```
function esPar($numero) {
    if ($numero % 2 == 0) {
        return true;
    }
    return false;
}

for ($i = 1; $i <= 10; $i++) {
    if (esPar($i)) {
        echo $i . " es par<br>";
    }
}
```

Explicación paso a paso:

- `function esPar($numero) {`: definimos la función con un parámetro: el número a revisar.
- `if ($numero % 2 == 0) {`: el operador `%` da el resto de la división. Si `$numero` es 4, `4 % 2` da 0 porque 4 entre 2 es exacto. Si es 5, `5 % 2` da 1. Entonces esta condición pregunta: el resto es cero? Ojo: aquí usamos `==` para comparar, no `=` para asignar.
- `return true;`: si el resto fue cero, el número es par y regresamos true. La función termina aquí.
- `return false;`: si llegamos a esta línea es porque el if no se cumplió, así que el número es impar y regresamos false.
- `}`: cierra la función.
- `for ($i = 1; $i <= 10; $i++) {`: recorremos los números del 1 al 10. Aquí si usamos `<=` porque queremos incluir el 10.
- `if (esPar($i)) {`: llamamos la función con cada número. Como regresa true o false, la podemos usar directamente como condición del if, sin comparar contra nada más.
- `echo $i . " es par<br>";`: solo se ejecuta cuando la función regreso true.
- `}`: cierran el if y el for.

El resultado imprime: 2 es par, 4 es par, 6 es par, 8 es par, 10 es par. Fíjate en el patrón: la función `esPar` encapsula la lógica de "ser par" y el ciclo se dedica solo a recorrer números. Cada parte hace su tarea, como mandan las buenas prácticas.

Ejercicio 2

Crea un arreglo indexado con los nombres de 5 de tus películas favoritas. Después: a) agrega una sexta película al final usando [], b) imprime cuántas películas hay con count(), c) recorre el arreglo con foreach e imprime cada película numerada desde el 1.

Solución 2

```
$peliculas = ["El Padrino", "Volver al Futuro", "Coco", "Toy Story", "Titanic"];

$peliculas[] = "Avengers";

echo "Tienes " . count($peliculas) . " películas favoritas:<br>";

$numero = 1;
foreach ($peliculas as $peli) {
    echo $numero . ". " . $peli . "<br>";
    $numero++;
}
```

Explicación paso a paso:

- `$peliculas = [...];`: creamos el arreglo con 5 películas en los índices 0 a 4.
- `$peliculas[] = "Avengers";`: agregamos la sexta al final. PHP la pone en el índice 5 sin que tengamos que calcularlo.
- `echo "Tienes " . count($peliculas) . ....`: count() regresa 6 y lo mostramos.
- `$numero = 1;`: creamos un contador manual que empieza en 1, porque queremos numerar para el usuario desde el 1, no desde el 0.
- `foreach ($peliculas as $peli) {`: recorreremos cada película. No necesitamos los índices, solo los valores.
- `echo $numero . ". " . $peli . "<br>;`: imprimimos "1. El Padrino", etc.
- `$numero++;`: aumentamos el contador para la siguiente vuelta.
- `}`: cierra el foreach.

Nota: también podríamos haber usado la forma `foreach ($peliculas as $índice => $peli)` y mostrar `($índice + 1)`. Las dos formas son válidas; el contador manual es más fácil de leer cuando apenas empiezas.

Ejercicio 3

Crea un arreglo asociativo llamado `$producto` con las llaves "nombre", "precio" y "stock", con los valores que tu quieras. Después escribe una función llamada `mostrarProducto` que reciba ese arreglo como parámetro e imprima los tres datos en líneas separadas. Llama la función. Pista: dentro de la función accedes a los datos con `$producto["nombre"]`, etc.

Solución 3

```
$producto = [
    "nombre" => "Audifonos Bluetooth",
    "precio" => 549,
    "stock" => 20
];

function mostrarProducto($prod) {
    echo "Nombre: " . $prod["nombre"] . "<br>";
    echo "Precio: $" . $prod["precio"] . "<br>";
    echo "Stock: " . $prod["stock"] . " piezas<br>";
}

mostrarProducto($producto);
```

Explicación paso a paso:

- `$producto = [...]`: el arreglo asociativo con los datos del producto. Cada llave describe que es el dato.
- `function mostrarProducto($prod) {`: la función recibe el arreglo completo como parámetro. Le pusimos `$prod` al parámetro para no confundirlo con la variable de afuera `$producto`; podría llamarse igual, pero así se ve más claro que son dos variables distintas (una vive afuera, otra adentro de la función).
- Las tres líneas de `echo` leen cada llave del arreglo recibido: `$prod["nombre"]`, `$prod["precio"]`, `$prod["stock"]`. Como el arreglo llegó por parámetro, no hay problema de alcance: la función trabaja con su copia local.
- `}`: cierra la función.
- `mostrarProducto($producto);`: llamamos la función pasándole el arreglo. Se imprimen los tres datos.

Este ejercicio demuestra algo poderoso: puedes pasar arreglos enteros a las funciones. La función no necesita saber de donde vino el arreglo ni cómo se llama afuera; solo necesita que

tenga las llaves correctas. Así se construyen piezas reutilizables: mañana le pasas otro producto y funciona igual.

Ejercicio 4

Tienes el arreglo `$precios = [120, 85, 200, 45, 300]`. Escribe una función llamada `precioMayor` que reciba un arreglo de precios y regrese el precio más alto usando `foreach` y `return` (no uses funciones de PHP como `max`, hazlo tu). Después imprime el resultado. Pista: guarda el primer elemento como "el mayor hasta ahora" y comparalo con cada uno de los demás.

Solución 4

```
function precioMayor($precios) {
    $mayor = $precios[0];
    foreach ($precios as $precio) {
        if ($precio > $mayor) {
            $mayor = $precio;
        }
    }
    return $mayor;
}

$precios = [120, 85, 200, 45, 300];
echo "El precio mayor es: $" . precioMayor($precios);
```

Explicación paso a paso:

- `function precioMayor($precios) {`: la función recibe el arreglo de precios.
- `$mayor = $precios[0];`: truco clave. Empezamos suponiendo que el mayor es el primer elemento (120). Así tenemos un punto de comparación desde el inicio. Si empezáramos con 0 funcionaría para precios positivos, pero con el primer elemento el código también sirve si hubiera números negativos.
- `foreach ($precios as $precio) {`: recorreremos cada precio.
- `if ($precio > $mayor) {`: preguntamos si el precio actual le gana al campeón actual.
- `$mayor = $precio;`: si le gana, hay nuevo campeón: actualizamos `$mayor`.
- `}`: cierran el `if` y el `foreach`. Al terminar el recorrido, `$mayor` tiene el valor más grande de todos: primero fue 120, luego 200 le ganó, luego 300 le ganó a 200. Nadie le ganó a 300.
- `return $mayor;`: entregamos el resultado.

- `};` cierra la función.
- Las últimas dos líneas crean el arreglo y muestran el resultado: "El precio mayor es: \$300".

Este algoritmo se llama "buscar el máximo" y es uno de los patrones fundamentales de la programación. La idea de "supon un ganador y comparalo con cada candidato" la vas a usar en cientos de problemas distintos.

Ejercicio 5

Crea un arreglo multidimensional `$tienda` con 3 productos; cada producto es un arreglo asociativo con "nombre" y "precio". Después usa `foreach` para imprimir cada producto con este formato: "Producto: [nombre] - Precio: \$[precio]". Al final, calcula e imprime el precio promedio de los tres productos usando una función. Pista: puedes reutilizar la lógica de suma y `count()` de los ejemplos de la unidad.

Solución 5

```
function calcularPromedioPrecios($productos) {
    $suma = 0;
    foreach ($productos as $p) {
        $suma = $suma + $p["precio"];
    }
    return $suma / count($productos);
}

$tienda = [
    ["nombre" => "Mouse", "precio" => 250],
    ["nombre" => "Teclado", "precio" => 600],
    ["nombre" => "Monitor", "precio" => 3500]
];

foreach ($tienda as $producto) {
    echo "Producto: " . $producto["nombre"] . " - Precio: $" . $producto["precio"]
    . "<br>";
}

$promedio = calcularPromedioPrecios($tienda);
echo "Precio promedio: $" . $promedio;
```

Explicación paso a paso:

- `function calcularPromedioPrecios($productos) {`: recibe la tabla completa de productos.
- `$suma = 0;`: acumulador.
- `foreach ($productos as $p) {`: recorreremos cada fila. `$p` es un arreglo asociativo con "nombre" y "precio".
- `$suma = $suma + $p["precio"];`: sumamos solo el precio de cada fila. Ignoramos el nombre porque no lo necesitamos para el promedio.
- `}`: cierra el `foreach`.
- `return $suma / count($productos);`: dividimos entre la cantidad de productos. `count()` regresa 3, así la función sirve para cualquier cantidad de productos, no solo 3.
- `}`: cierra la función.
- `$tienda = [ ... ];`: la tabla de 3 productos.
- `foreach ($tienda as $producto) {`: recorreremos para mostrar el catálogo.
- `echo "Producto: " . ...;`: imprimimos cada línea con el formato pedido.
- `}`: cierra el `foreach`.
- `$promedio = calcularPromedioPrecios($tienda);`: llamamos la función con toda la tabla y guardamos el resultado: $(250 + 600 + 3500) / 3 = 1450$.
- `echo "Precio promedio: $" . $promedio;`: mostramos el promedio.

Este ejercicio junta todo lo de la unidad en un mini programa real: tabla de datos, recorrido para mostrar, función que calcula. Si lo entendiste sin atorarte, ya dominas lo esencial de funciones y arreglos.

Resumen de la unidad

- Una función es un bloque de código con nombre que defines una vez y llamas muchas veces. Se define con `function nombre() { }` y se ejecuta escribiendo `nombre();`.
- Los parámetros son los datos que la función acepta (se declaran al definirla) y los argumentos son los valores reales que le pasas (al llamarla). El orden de los argumentos debe coincidir con el de los parámetros.
- `return` entrega un valor a quien llamó la función para seguir usandolo; `echo` solo lo muestra en pantalla. Si el resultado se va a reutilizar, usa `return`. Todo lo que este después de un `return` dentro de la función nunca se ejecuta.

- Los parámetros pueden tener valores por defecto (\$iva = 0.16), que se usan cuando la llamada no pasa ese argumento. Los parámetros con valor por defecto van al final de la lista.
- Alcance de variables: las variables creadas dentro de una función solo existen ahí (locales); las de afuera no entran solas (globales). Pasa los datos como parámetros y regresa resultados con return; esa es la forma limpia.
- Un arreglo guarda varios valores en una sola variable. El arreglo indexado usa números como llave (empezando en 0); el arreglo asociativo usa textos como llave con la sintaxis "llave" => valor.
- count(\$arreglo) dice cuantos elementos hay. \$arreglo[] = valor agrega un elemento al final sin calcular el índice.
- El ciclo for con la fórmula for (\$i = 0; \$i < count(\$a); \$i++) recorre arreglos indexados usando \$a[\$i]. Cuidado con usar <= en vez de <: leerías un índice que no existe.
- foreach recorre cualquier arreglo sin contadores: foreach (\$a as \$valor) entrega cada valor, y foreach (\$a as \$llave => \$valor) entrega también la llave o índice.
- Un arreglo multidimensional es un arreglo dentro de otro, como una tabla: \$tabla[filas][columnas]. Es la forma en que PHP maneja listas de registros (alumnos, productos, usuarios).
- Funciones útiles: isset(\$a["llave"]) pregunta si la llave existe; empty(\$x) pregunta si está vacía; print_r(\$a) muestra el contenido del arreglo para depurar; in_array(\$valor, \$a) busca un valor; array_push(\$a, \$v1, \$v2) agrega varios elementos al final.
- Errores típicos: olvidar el return (la función calcula pero no entrega nada), usar una variable global dentro de la función sin pasarla como parámetro, leer una llave que no existe (revisalo con isset y print_r), y escribir = (asignar) en vez de == (comparar) dentro de un if.
- Buenas prácticas: una función hace una sola tarea; nombralas con verbo + sustantivo (calcularTotal); si repites un bloque de código dos veces, conviértelo en función; evita depender de variables globales; comenta el porqué, no el que.

UNIDAD 5: FORMULARIOS WEB

Bienvenido a la unidad más importante de este curso. Hasta ahora has aprendido a escribir PHP que vive solo en el servidor: declara variables, toma decisiones, repite acciones y organiza el código en funciones. Todo eso es muy útil, pero si lo piensas bien, esos programas todavía están encerrados en una cajita de cristal. El usuario puede ver el resultado en su pantalla, pero no puede hablarle al programa, no puede pedirle nada, no puede darle información. Es como tener una ventanilla de tramites con el vidrio cerrado y sin ranura para pasar los papeles: del otro lado esta la oficina completa, con archiveros, computadoras y empleados, pero nadie puede entregar ni pedir nada.

Esta unidad abre esa ranura. Los formularios web son el puente entre el usuario y PHP. Son el mecanismo que permite que una persona escriba su nombre, su correo o su búsqueda en una página y que tu código PHP reciba esos datos, los revise y responda. Sin formularios no hay registros de usuarios, no hay inicios de sesión, no hay buscadores, no hay comentarios, no hay carritos de compra, no hay nada de lo que hace útil a una página web real. Con formularios, tu PHP deja de ser un ejercicio de escritorio y se convierte en una aplicación que conversa con personas de verdad.

Piensa en la analogía del buzón de sugerencias de una tienda. El cliente escribe un papelito, lo dobla, lo mete por la ranura del buzón y se va. Días después, alguien abre el buzón por dentro, lee los papelitos y actúa según lo que dicen. En la web funciona igual: el usuario llena un formulario y presiona el botón de enviar. Esos datos viajan por internet hasta tu servidor, PHP los recoge, los revisa y responde con una página nueva. El usuario nunca ve tu código, solo ve el buzón (el formulario) y la respuesta (la página de resultado). Tu trabajo en esta unidad es aprender a construir el buzón y, sobre todo, a leer bien los papelitos que llegan por dentro.

En esta unidad vas a ver las clases 13, 14 y 15 del curso, y cada una abre una puerta nueva:

En la clase 13 aprenderás a construir un formulario HTML de verdad: la etiqueta form, los campos input, el atributo name (qué es la etiqueta del papelito, sin la cual PHP no sabe que dato está recibiendo), y los atributos action y method, que dicen a donde viajan los datos y por cual camino. También entenderás que es exactamente lo que viaja en una petición cuando el usuario presiona enviar, para que dejes de verlo como magia y empieces a verlo como un sobre con una dirección.

En la clase 14 conocerás los dos caminos por los que viajan los datos: GET y POST. Aprenderás que `$_GET` es como una postal: el mensaje va escrito a la vista, en la misma dirección de la página, y por eso sirve para búsquedas y filtros pero jamás para contraseñas. Y

que `$_POST` es como un sobre cerrado: el mensaje viaja escondido dentro del cuerpo de la petición, y por eso sirve para registros, inicios de sesión y todo lo que no debe andar a la vista. También verás que existe `$_REQUEST`, que mezcla los dos, y por qué los programadores con experiencia lo evitan.

En la clase 15 entrarás al tema que separa a los aficionados de los profesionales: la validación de datos. Aprenderás que nunca, por ningún motivo, debes confiar en lo que el usuario escribe en un formulario. Verás las herramientas de PHP para revisar datos: `isset` para saber si un dato llegó, `empty` para saber si viene vacío, `trim` para quitar espacios de más, `strlen` para medir la longitud, `is_numeric` para saber si es número y `filter_var` para validar correos como un profesional. También aprenderás a sanitizar con `htmlspecialchars`, que es como pasar el agua por un filtro antes de tomarla: convierte los caracteres peligrosos en texto inofensivo. Y verás como repoblar el formulario cuando hay errores, para que el usuario no tenga que escribir todo de nuevo, y como mostrar mensajes de error amigables que ayuden en lugar de regañar.

Al terminar esta unidad serás capaz de construir formularios completos y funcionales: un saludo personalizado, un buscador, un registro de usuarios con validación de correo y contraseña, un formulario de contacto que marca campo por campo que salió mal, una calculadora que recibe números por formulario, una encuesta con casillas de verificación y formularios que recuerdan lo que el usuario ya escribió. En pocas palabras: vas a dominar la ranura del buzón por los dos lados, el que construye el formulario y el que lee los datos.

Un consejo antes de empezar, de esos que te ahorran horas de frustración: en esta unidad vas a cometer errores, y está bien, porque los errores de esta unidad son los más instructivos de todo el curso. Olvidar el atributo `name` en un campo, confundir GET con POST, olvidar validar un dato vacío... todos los programadores pasaron por ahí. La diferencia entre quien aprende y quien se atora no es no equivocarse, es leer con calma el mensaje de error, revisar campo por campo y probar de nuevo. Ten tu servidor local encendido, tu editor abierto y las ganas de romper cosas: así se aprende PHP de verdad.

Explicación detallada

El formulario HTML: el buzón que ve el usuario

Todo empieza en el navegador, con HTML. Un formulario web se construye con la etiqueta `form`, y dentro de ella van los campos donde el usuario escribe: cuadros de texto, listas, casillas, botones. Cuando el usuario presiona el botón de enviar, el navegador recoge todo lo que se escribió, lo empaqueta y lo manda al servidor. PHP, del otro lado, lo desempaca.

La etiqueta form necesita dos atributos que son la dirección del sobre: action y method. El atributo action dice a que archivo PHP se van a mandar los datos. Si escribes action="saludo.php", los datos viajan a saludo.php. Si lo dejas vacío con action="", los datos se mandan a la misma página donde esta el formulario, lo cual es muy comun y muy práctico, como verás en los ejemplos. El atributo method dice por cual camino viajan los datos: method="get" o method="post". Si no pones method, el navegador usa GET por defecto, así que mejor ponlo siempre explícito para que tu código sea claro.

Dentro del form van los campos, y aquí esta el detalle más importante de toda la unidad, el que causa el ochenta por ciento de los dolores de cabeza de los principiantes: cada campo que quieras recibir en PHP debe tener el atributo name. El name es la etiqueta del papelito. Cuando el navegador empaqueta los datos, usa el name como nombre del dato y lo que el usuario escribió como valor. Si un campo no tiene name, el navegador simplemente lo ignora al enviar, como si ese papelito nunca hubiera entrado al buzón. PHP recibe los datos en arreglos especiales usando esos names como llaves, así que sin name no hay dato, por más bonito que se vea el campo en pantalla.

Mira la anatomía de un campo típico: input type="text" name="nombre". El type dice que clase de campo es (texto, contraseña, casilla, botón), el name dice cómo se va a llamar el dato cuando llegue a PHP. Hay otros atributos utiles: value pone un valor inicial en el campo, placeholder muestra un texto gris de ayuda que se borra al escribir, required le pide al navegador que no deje enviar el formulario si el campo está vacío (ojo: esto es solo una ayuda visual, nunca una validación de verdad, como verás en la clase 15).

Los tipos de input más usados son: text para texto corto como nombres, password para contraseñas (muestra puntitos en lugar de letras), email para correos (el navegador revisa por encima que parezca correo), number para números (con flechitas para subir y bajar), checkbox para casillas que se pueden marcar varias a la vez, radio para opciones donde solo se puede elegir una, submit para el botón de enviar, y textarea (que no es input sino su propia etiqueta) para textos largos como mensajes o comentarios. También existe select con sus option para listas desplegables, muy útil para países, categorías o cualquier lista cerrada de opciones.

Un formulario completo y bien armado se ve así en su estructura: la etiqueta form con action y method, una etiqueta label por cada campo (el texto que le dice al usuario que escribir), el input con su name, y al final el botón submit. El label no es decoración: ayuda a que el usuario entienda que va en cada campo y mejora la accesibilidad. Acostumbrate a ponerlo siempre.

Que viaja en la petición: el sobre por dentro

Cuando el usuario presiona enviar, el navegador arma lo que se llama una petición HTTP y la manda al servidor. Piensa en la petición como un sobre de correo con dos partes: el encabezado, que es como la parte de afuera del sobre (a donde va, por qué camino, que tipo de contenido trae), y el cuerpo, que es la carta que va adentro.

Si el método es GET, los datos van escritos en la propia dirección, después de un signo de interrogación. Por ejemplo, si el formulario tiene un campo `name="nombre"` y el usuario escribió "Ana", la dirección quedaría algo como `saludo.php?nombre=Ana`. A eso se le llama cadena de consulta o query string. Cada dato va como `nombre=valor`, y si hay varios se separan con el signo `&`: `saludo.php?nombre=Ana&edad=25`. Todo va a la vista: en la barra del navegador, en el historial, en los registros del servidor.

Si el método es POST, la dirección queda limpia (solo `saludo.php`) y los datos viajan escondidos dentro del cuerpo de la petición, como la carta dentro del sobre cerrado. El usuario no los ve en la barra del navegador, no quedan en el historial de la misma forma y no se guardan al compartir un enlace.

Esta diferencia lo es todo, y es la base para decidir que método usar en cada caso. Pero antes de decidir, necesitas conocer las dos manos con las que PHP recoge el correo: `$_GET` y `$_POST`.

`$_GET`: la postal

`$_GET` es un arreglo especial de PHP (una variable superglobal, de esas que existen en cualquier parte de tu código sin necesidad de declararlas) que contiene todos los datos que llegaron por el método GET, es decir, los que venían escritos en la dirección. Cada `name` del formulario se convierte en una llave del arreglo, y lo que el usuario escribió se convierte en el valor.

Si la dirección es `saludo.php?nombre=Ana`, entonces en PHP `$_GET["nombre"]` vale "Ana". Si la dirección es `buscar.php?q=zapatos&categoria=ropa`, entonces `$_GET["q"]` vale "zapatos" y `$_GET["categoria"]` vale "ropa". Así de directo: el `name` del campo es la llave, el texto escrito es el valor.

La analogía perfecta para GET es la postal. Cuando mandas una postal, el mensaje va escrito a la vista, sin sobre: el cartero lo lee, quien la reciba en la casa lo lee, cualquiera que la vea pasar lo lee. No pondrías en una postal tu contraseña del banco ni el número de tu tarjeta, verdad? Pues lo mismo con GET: como los datos van visibles en la URL, jamás debes usar GET para contraseñas, datos bancarios o información privada.

Entonces, para que sirve GET? Para todo lo que no es secreto y que además conviene poder compartir o guardar: búsquedas, filtros y paginación. Piensa en un buscador: cuando buscas "zapatos rojos" en una tienda en línea, la dirección queda algo como `tienda.php?buscar=zapatos+rojos`. Eso es buenísimo, porque puedes copiar esa dirección, mandarsela a un amigo por mensaje y al abrirla vera exactamente la misma búsqueda. Lo mismo con los filtros ("mostrar solo talla M") y con las páginas de resultados ("página 3"). Todo eso son datos que no importa que se vean, y que además ganan mucho al poder compartirse como enlace. Esa es la regla de oro: si el resultado se puede compartir con un enlace, GET es el camino correcto.

Otra ventaja de GET es que es fácil de probar: puedes escribir la dirección a mano en el navegador con los datos que quieras y ver que responde tu PHP. Durante el desarrollo vas a usar mucho este truco para probar rápido sin llenar el formulario una y otra vez.

Pero GET tiene limites que debes conocer. Las direcciones tienen un tamaño máximo (depende del navegador y del servidor, pero anda por los dos mil caracteres más o menos), así que no sirve para mandar textos largos como el mensaje de un formulario de contacto extenso. Y repito lo más importante: todo queda visible y guardado en el historial, así que es el camino equivocado para cualquier dato sensible.

Un detalle práctico: como los datos de GET van en la dirección, los espacios y caracteres especiales se codifican solos. Un espacio se convierte en `%20` o en `+`, una enie y las tildes también se codifican. PHP los decodifica automáticamente al llenar `$_GET`, así que tu recibes el texto normal y no tienes que preocuparte por eso. Solo tenlo presente cuando veas direcciones raras en la barra del navegador: no están rotas, están codificadas.

`$_POST`: el sobre cerrado

`$_POST` es el otro arreglo superglobal: contiene todos los datos que llegaron por el método POST, es decir, los que viajaron escondidos en el cuerpo de la petición. Se usa exactamente igual que `$_GET`: si el formulario tiene un campo `name="correo"`, en PHP `$_POST["correo"]` tiene lo que el usuario escribió. La única diferencia es el camino por el que viajaron los datos.

La analogía para POST es el sobre cerrado. Metes la carta en el sobre, lo cierras, y por fuera solo se ve la dirección del destinatario. El cartero sabe a donde va, pero no puede leer el contenido sin abrirlo. Así viajan los datos POST: la dirección queda limpia y el contenido va protegido dentro del cuerpo de la petición, fuera de la vista del curioso que mire por encima del hombro, fuera del historial del navegador en la forma en que GET si queda, y fuera de los enlaces que se comparten.

Ojo con un malentendido muy comun entre principiantes: que los datos vayan "ocultos" en POST no significa que vayan cifrados ni que sean imposibles de interceptar. Un sobre cerrado lo puede abrir alguien en el camino si quiere. La verdadera protección de los datos en internet la da el HTTPS (el candadito del navegador), no el método POST. Lo que POST si te da es discreción frente al usuario casual: no quedan a la vista en la pantalla ni en el historial. Para datos sensibles eso es lo mínimo indispensable, pero no lo confundas con seguridad total.

Cuando usar POST? Para todo lo que modifica algo o lo que debe mantenerse privado: registros de usuarios, inicios de sesión, contraseñas, formularios de contacto con datos personales, comentarios, compras, cualquier acción que cambie información en tu sistema. La regla práctica que usan los profesionales es esta: si la acción cambia algo en el servidor (guardar, borrar, actualizar) o si los datos son privados, usa POST. Si solo estas pidiendo información para ver (buscar, filtrar, navegar), usa GET.

Hay una razón técnica detras de esa regla, además de la privacidad: los navegadores tratan distinto las dos peticiones. Si recargas una página que llegó por POST, el navegador te pregunta "quieres volver a enviar los datos?", porque sabe que reenviar podría duplicar una acción (registrar dos veces al mismo usuario, cobrar dos veces una compra). Con GET no pregunta, porque pedir información dos veces no hace dano. Esa es otra pista: si duplicar la acción seria un problema, el método correcto es POST.

POST tampoco tiene el limite de tamaño tan estricto de las URLs, así que sirve para textos largos, y es el único de los dos que puede enviar archivos (con un ajuste extra en el formulario que verás en unidades avanzadas). Para los formularios de esta unidad, quédate con la idea central: POST es el camino de los datos que importan y que no deben andar a la vista.

`$_REQUEST`: el cajón mezclado (y por qué evitarlo)

Existe un tercer arreglo superglobal llamado `$_REQUEST` que contiene los datos de GET, de POST y de las cookies, todos mezclados en un solo cajón. A primera vista parece comodo: en lugar de pensar si el dato llegó por GET o por POST, simplemente lo pides a `$_REQUEST` y listo.

Los programadores con experiencia lo evitan, y la razón es sencilla: `$_REQUEST` te quita información valiosa. Cuando usas `$_GET` o `$_POST`, tu código deja claro por qué camino esperas el dato, y eso es una forma de documentar tu intención. Si un formulario de registro lee la contraseña con `$_POST["clave"]`, cualquiera que lea el código entiende qué ese dato debe viajar por el camino privado. Si lo lees con `$_REQUEST["clave"]`, ya no se sabe si llegó por la postal o por el sobre cerrado, y un atacante podría mandar ese mismo dato por GET (visible en la URL) y tu código lo aceptaria igual, sin quejarse.

Además, si el mismo nombre de dato llega por GET y por POST a la vez, `$_REQUEST` se queda con uno solo según un orden de prioridad configurado en el servidor, y ese comportamiento puede variar de un servidor a otro. Eso es una fuente de errores difíciles de rastrear. La buena práctica es simple y fácil de recordar: usa siempre `$_GET` para lo que esperas por GET y `$_POST` para lo que esperas por POST. Se explícito. Tu yo del futuro, depurando código a las dos de la mañana, te lo va a agradecer.

Validación de datos: no confíes en nadie (ni en el formulario)

Llegamos al tema más importante de la unidad, y probablemente uno de los más importantes de todo tu camino como programador: la validación. La regla de oro es corta y hay que gravarsela como mantra: nunca confíes en lo que manda el usuario.

Esto suena dramático, pero tiene una explicación muy práctica. Tu formulario HTML vive en el navegador del usuario, y el usuario (o un programa malicioso haciéndose pasar por usuario) puede mandar a tu PHP lo que quiera, se haya llenado el formulario o no. Puede quitar el atributo `required` con las herramientas del navegador, puede escribir un correo que no es correo, puede mandar campos vacíos, puede incluso construir peticiones a mano sin abrir tu página. Tu PHP no tiene forma de saber si los datos vinieron de tu bonito formulario o de un ataque: solo ve datos llegando. Por eso la validación debe hacerse siempre en el servidor, en tu código PHP, sin excepciones.

La validación en el navegador (el atributo `required`, el `type="email"`, el JavaScript que revisa antes de enviar) es una cortesía para el usuario honesto: le avisa rápido si olvidó algo, sin esperar al viaje de ida y vuelta al servidor. Es útil y debes usarla, pero es como el candado de una bicicleta: detiene al distraído, no al ladrón. La validación de verdad, la que protege tu sistema, vive en PHP.

Validar significa hacer preguntas sobre cada dato antes de usarlo: llegó este dato? viene vacío? tiene la longitud correcta? es un número? tiene formato de correo? PHP te da una caja de herramientas muy completa para responder esas preguntas. Vamos una por una.

`isset`: la pregunta "llegó el dato?". La función `isset` revisa si una variable existe y no es nula. Su uso más común en formularios es preguntar si el formulario ya fue enviado: `if (isset($_POST["enviar"]))`. Si el botón de enviar tiene `name="enviar"`, ese dato solo existe en `$_POST` cuando el usuario ya presionó el botón. Así tu página puede mostrar el formulario vacío la primera vez y procesar los datos solo cuando ya llegaron. También sirve para revisar campos individuales: `isset($_POST["nombre"])` te dice si ese campo llegó, aunque venga vacío. Recuerda: `isset` pregunta por existencia, no por contenido.

`empty`: la pregunta "viene vacío?". La función `empty` revisa si un valor se considera vacío: una cadena vacía "", el número 0, la cadena "0", un arreglo vacío, o algo que no existe. Es perfecta para detectar campos que el usuario dejó en blanco: `if (empty($_POST["nombre"]))`. Pero cuidado con su trampa más famosa: `empty` considera vacío al número 0 y a la cadena "0". Si tu formulario pide la edad y alguien escribe 0, `empty` dirá que está vacío aunque el usuario si escribió algo. Para textos como nombres y correos no hay problema, pero para números usa comparaciones más precisas, como verás en los ejemplos.

`trim`: quitar los espacios de los extremos. Los usuarios dejan espacios al inicio o al final sin querer (" Ana "), o a proposito para burlar validaciones (un campo con un solo espacio " " pasa la prueba de "no está vacío" aunque en realidad no dice nada). La función `trim` quita los espacios (y otros caracteres invisibles) del inicio y del final de una cadena: `trim(" Ana ")` devuelve "Ana". La práctica profesional es pasar cada texto por `trim` antes de validarlo: `$nombre = trim($_POST["nombre"]);` y después validar `$nombre`. Así un campo con puros espacios se convierte en cadena vacía y tu validación de `empty` lo atrapa correctamente.

`strlen`: medir la longitud. La función `strlen` cuenta cuantos caracteres tiene una cadena. Sirve para reglas como "la contraseña debe tener al menos 8 caracteres": `if (strlen($clave) < 8)`. También para poner topes maximos, porque un usuario podría pegar un texto de un millon de caracteres y saturar tu sistema: `if (strlen($mensaje) > 500)`. Combinar un mínimo y un máximo es una validación clasica y muy efectiva.

`is_numeric`: es número? La función `is_numeric` revisa si un valor es numerico o una cadena que representa un número: `is_numeric("25")` es verdadero, `is_numeric("veinticinco")` es falso. Utilisima para edades, cantidades y precios. Pero ojo: todo lo que llega de un formulario llega como cadena de texto, incluso los números. Si necesitas hacer cuentas, convierte después de validar: `$edad = (int)$_POST["edad"];`. Primero validas que sea numerico, luego lo conviertes al tipo que necesitas.

`filter_var` con `FILTER_VALIDATE_EMAIL`: el validador profesional de correos. Validar un correo con código manual es un dolor de cabeza: tiene que tener arroba, punto, texto antes y después... PHP trae la solución hecha: `filter_var($correo, FILTER_VALIDATE_EMAIL)`. Si el correo es válido, devuelve el correo; si no, devuelve false. Así de simple: `if (filter_var($correo, FILTER_VALIDATE_EMAIL) === false) { /* correo invalido */ }`. Usala siempre para correos en lugar de inventar tu propia revisión con `strpos` o expresiones regulares complicadas. La función `filter_var` tiene más filtros (para URLs, números enteros, etc.), pero el de correo es el que vas a usar el noventa por ciento de las veces.

El patrón profesional para validar un formulario completo es este: primero revisas si el formulario fue enviado con `isset`, luego tomas cada dato, le pasas `trim`, y lo sometes a sus

reglas una por una, guardando los mensajes de error en un arreglo. Si al final el arreglo de errores está vacío, los datos están bien y puedes procesarlos (guardarlos, mandarlos por correo, mostrar el resultado). Si hay errores, vuelves a mostrar el formulario con los mensajes. Este patrón lo verás funcionando en los ejemplos paso a paso, hasta que te salga de memoria.

Sanitización con `htmlspecialchars`: el filtro de agua

Validar es revisar que el dato cumpla las reglas. Sanitizar es otra cosa: es limpiar el dato para que sea seguro mostrarlo en una página. La herramienta principal se llama `htmlspecialchars`, y su trabajo es convertir los caracteres especiales de HTML en texto inofensivo.

El problema que resuelve es este: imagina que tu página saluda al usuario mostrando lo que escribió: `echo "Hola, " . $_GET["nombre"];`. Si un usuario escribe "Ana", la página muestra "Hola, Ana". Perfecto. Pero si un usuario malicioso escribe código HTML o JavaScript en el campo, por ejemplo una etiqueta `script`, tu página lo mostraría como código real y el navegador lo ejecutaría. Eso se llama inyección de código, y es una de las vulnerabilidades más comunes y peligrosas de la web. El atacante podría robar datos de otros visitantes o desfigurar tu página.

La analogía es el filtro de agua: el agua del río puede traer tierra y bichos; antes de tomarla la pasas por un filtro que retiene lo peligroso y deja pasar lo bueno. `htmlspecialchars` es ese filtro para el texto que vas a mostrar: convierte `<` en `<`, `>` en `>`, `&` en `&` y las comillas en sus versiones seguras. Después de pasarlo por el filtro, el navegador muestra esos caracteres como texto normal en pantalla, en lugar de interpretarlos como código. El mensaje malicioso se ve como texto raro, pero no se ejecuta. Peligro neutralizado.

La regla es clara: cada vez que vayas a mostrar en pantalla un dato que vino del usuario, pasalo por `htmlspecialchars`. Sin excepciones, sin "este campo es seguro porque yo lo controlo". El costo es una función, el beneficio es no regalarle tu página a un atacante. En los ejemplos verás que se vuelve un hábito automático: `echo htmlspecialchars($nombre);`

Un detalle importante: sanitizar es para mostrar, no para guardar. Si vas a guardar datos en una base de datos (tema de unidades futuras), ahí se usan otras técnicas. Por ahora, quédate con la regla de oro de esta unidad: valida al recibir, sanitiza al mostrar.

Repoplar el formulario: no hagas escribir dos veces

Imagina que llenas un formulario largo de registro: nombre, correo, contraseña, confirmación, país... presionas enviar y la página te dice "el correo no es válido". Miras el formulario y esta completamente vacío. Tienes que escribir todo de nuevo. Dan ganas de cerrar la página y no

volver, verdad? Pues eso es exactamente lo que le haces a tus usuarios si no repoblas el formulario tras un error.

Repoblar significa volver a poner en cada campo lo que el usuario ya había escrito, para que solo corrija lo que salió mal. La técnica es sencilla: en el atributo value de cada input, escribes el valor que llegó por POST (ya sanitizado). Algo así: `input type="text" name="nombre" value="<?php echo htmlspecialchars($nombre); ?>"`. La primera vez que se carga la página, \$nombre está vacío y el campo se muestra vacío. Si hubo un error y se vuelve a mostrar el formulario, \$nombre tiene lo que el usuario escribió y aparece en el campo. Para los textarea el valor va entre las etiquetas de apertura y cierre, y para los select y checkbox se marca la opción elegida con selected o checked según corresponda.

Este detalle separa las páginas amateurs de las profesionales. Cuesta unas líneas más de código por campo, pero el usuario lo nota y lo agradece. En el ejemplo del formulario de contacto completo lo verás aplicado campo por campo.

Mensajes de error amigables: ayuda, no reganos

Cuando la validación encuentra problemas, hay que decirselo al usuario, pero cómo se lo dices importa tanto como detectarlo. Un mensaje como "Error: entrada invalida" no ayuda a nadie: el usuario no sabe que hizo mal ni cómo arreglarlo. Un buen mensaje de error es específico, está cerca del campo problemático y sugiere la solución.

Compara: "Error en el formulario" contra "El correo debe tener un formato válido, por ejemplo usuario@ejemplo.com". El primero frustra, el segundo guía. La técnica profesional es guardar un mensaje por cada campo que falle, en un arreglo de errores con el nombre del campo como llave: `$errores["correo"] = "Escribe un correo válido, por ejemplo usuario@ejemplo.com";`. Luego, al mostrar el formulario, debajo de cada campo revisas si hay un error para ese campo y lo muestras. Así el usuario ve exactamente qué corregir y dónde.

También conviene mostrar un resumen arriba del formulario ("Revisa los campos marcados, hay 2 errores por corregir") para que de un vistazo sepa qué paso. Y un consejo de tono: los mensajes deben sonar como ayuda, no como regano. "El nombre es obligatorio" suena mejor que "No escribiste el nombre, error". Parece un detalle menor, pero la experiencia del usuario se construye con estos detalles.

Errores comunes: la galería de los horrores (para no visitarla)

Cada generación de estudiantes de PHP comete los mismos errores en esta unidad. Conocerlos de antemano es como tener el mapa de los baches antes de manejar por la calle:

Olvidar el atributo name en un campo. El campo se ve perfecto en pantalla, el usuario escribe, presiona enviar... y PHP dice que el dato no existe. Revisas el código veinte minutos hasta que descubres que el input no tenía name. Solución: cada vez que un dato "no llega", lo primero que revisas es el name del campo en el HTML.

Usar GET para contraseñas o datos sensibles. El formulario funciona, el registro se completa, todo parece bien... hasta que notas que la contraseña quedó escrita en la barra del navegador, en el historial y en los registros del servidor. Solución: repasa la regla (datos privados o acciones que cambian algo = POST) antes de escribir el method.

No validar nunca porque "el formulario ya tiene required". El required del navegador se quita en dos clics con las herramientas de desarrollo, y las peticiones se pueden fabricar a mano. Tu PHP recibe basura y la procesa como si fuera valida. Solución: validación en el servidor siempre, sin excepciones; lo del navegador es solo cortesía.

Confundir isset con empty. Usar isset para saber si un campo viene vacío no funciona: isset dice "existe", y un campo enviado vacío si existe. Usar empty para saber si el formulario fue enviado tampoco funciona bien: empty(\$_POST) con el formulario vacío... mejor usa isset sobre el botón de enviar o sobre un campo específico. Solución: isset pregunta existencia, empty pregunta contenido. Memoriza la diferencia.

Mostrar datos del usuario sin htmlspecialchars. Funciona perfecto con usuarios normales y se convierte en un agujero de seguridad con el primer usuario malicioso. Solución: convierte en habito el echo con htmlspecialchars para todo dato que venga de fuera.

Olvidar que todo llega como texto. Comparar \$_POST["edad"] > 18 funciona por conversión automática, pero hacer cuentas raras o validaciones estrictas con === contra números falla porque "25" no es idéntico a 25. Solución: valida con is_numeric y convierte con (int) o (float) cuando necesites el número de verdad.

Buenas prácticas: el código de honor del formulariologo

Para cerrar la explicación, las reglas que distinguen al profesional. No son opcionales si quieres que tus páginas sean confiables:

Valida siempre en el servidor aunque haya validación en el navegador. Ya lo dijimos, pero vale repetirlo porque es la regla número uno: lo del navegador es ayuda visual, lo de PHP es la ley.

Nunca confíes en lo que manda el usuario. Ni en el contenido, ni en el formato, ni en que el dato venga del campo que tú crees. Revisa todo: existencia con isset, contenido con empty y trim, formato con las funciones de validación.

Sanitiza al mostrar. Todo dato que venga de fuera y se muestre en pantalla pasa por `htmlspecialchars`. Hazlo un reflejo, como ponerte el cinturón al subir al coche.

Usa el método correcto para cada caso. GET para buscar, filtrar y navegar (datos visibles y compartibles). POST para registrar, identificar y cambiar cosas (datos privados y acciones). Si dudas, pregunta: "¿me molestaría que este dato quedará en el historial del navegador?" Si la respuesta es sí, es POST.

Dale al usuario una experiencia digna. Repobla los campos tras un error, muestra mensajes específicos junto a cada campo, y confirma con un mensaje claro cuando todo salió bien ("Registro completado, bienvenido"). Un formulario que funciona pero maltrata al usuario es un formulario a medias.

Con estas bases claras, es hora de ver todo en acción. En la siguiente sección cada ejemplo está explicado línea por línea: léelos con calma, escríbelos tú mismo en tu editor y rompelos a propósito para ver qué pasa. Así se aprende.

Ejemplos paso a paso

Ejemplo 1: formulario de saludo con GET

Este es el "hola mundo" de los formularios: una página que pide tu nombre y te saluda. Usa GET porque el nombre no es un dato sensible y porque así puedes ver cómo viaja en la dirección. Guárdalo como `saludo.php` y ábrelo en tu navegador.


```

<?php
$nombre = "";
if (isset($_GET["nombre"])) {
    $nombre = trim($_GET["nombre"]);
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Saludo con GET</title>
</head>
<body>
    <h1>Dime tu nombre</h1>
    <form action="" method="get">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" placeholder="Escribe tu
nombre">
        <input type="submit" value="Saludar">
    </form>
    <?php if ($nombre !== "") { ?>
        <p>Hola, <?php echo htmlspecialchars($nombre); ?>. Bienvenido a PHP.</p>
    <?php } ?>
</body>
</html>

```

Explicación línea por línea. Las primeras cuatro líneas son PHP puro antes del HTML: se crea la variable \$nombre vacía, y luego se pregunta con isset si llegó el dato "nombre" por GET. Si el usuario todavía no envió el formulario, \$_GET["nombre"] no existe y \$nombre se queda vacía. Si ya lo envió, se guarda el valor pasándole trim para quitar espacios de los extremos. Fíjate que este PHP va al inicio para preparar los datos antes de pintar la página.

Luego viene el HTML normal: doctype, head con el charset UTF-8 (importante para que las tildes y enies se vean bien) y el título. En el body hay un encabezado h1 y el formulario. El form tiene action="" (los datos se mandan a esta misma página) y method="get" (viajan en la dirección). Dentro van el label asociado al campo, el input de tipo texto con name="nombre" (sin este name, PHP jamás vería el dato) y el botón submit con value="Saludar", que es el texto que se ve en el botón.

Después del formulario hay un bloque PHP que solo muestra el saludo si \$nombre no está vacío: if (\$nombre !== ""). Dentro se imprime el párrafo con el nombre, pasando SIEMPRE por htmlspecialchars para mostrarlo de forma segura. Prueba esto: escribe tu nombre, presiona

Saludar y mira la barra del navegador. Verás algo como `saludo.php?nombre=Ana`. Esos son tus datos viajando como postal, a la vista de todos. Ahora escribe código HTML en el campo, por ejemplo `<b>hola</b>`, y observa que se muestra como texto y no como negritas: el filtro de agua (`htmlspecialchars`) esta funcionando.

Ejemplo 2: buscador con GET

Un buscador es el caso de uso perfecto para GET: la búsqueda se puede compartir como enlace y no hay nada privado. Esta página simula un catálogo pequeño de productos y filtra los que coincidan con lo que el usuario buscó.

```

<?php
$catalogo = array("manzana", "mango", "platano", "pera", "mandarina", "melon");
$busqueda = "";
$resultados = array();
if (isset($_GET["q"])) {
    $busqueda = trim($_GET["q"]);
    if ($busqueda !== "") {
        foreach ($catalogo as $fruta) {
            if (strpos($fruta, $busqueda) !== false) {
                $resultados[] = $fruta;
            }
        }
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Buscador de frutas</title>
</head>
<body>
    <h1>Buscador</h1>
    <form action="" method="get">
        <label for="q">Buscar fruta:</label>
        <input type="text" id="q" name="q" value="<?php echo
htmlspecialchars($busqueda); ?>">
        <input type="submit" value="Buscar">
    </form>
    <?php if (isset($_GET["q"])) { ?>
        <h2>Resultados para "<?php echo htmlspecialchars($busqueda); ?>":</h2>
        <?php if (count($resultados) > 0) { ?>
            <ul>
                <?php foreach ($resultados as $r) { ?>
                    <li><?php echo htmlspecialchars($r); ?></li>
                <?php } ?>
            </ul>
        <?php } else { ?>
            <p>No se encontraron frutas con ese texto.</p>
        <?php } ?>
    <?php } ?>
</body>
</html>

```

Explicación línea por línea. Primero se define un arreglo \$catálogo con seis frutas: es nuestra "base de datos" de juguete. Luego \$búsqueda empieza vacía y \$resultados como arreglo vacío. Si llegó el dato "q" por GET, se guarda con trim en \$búsqueda, y si no está vacía se recorre el catálogo con foreach: la función strpos busca el texto dentro de cada fruta sin importar mayúsculas o minúsculas, y si lo encuentra (devuelve algo distinto de false), la fruta se agrega a \$resultados. Nota el detalle: strpos puede devolver 0 cuando el texto esta al inicio, y 0 se considera falso en PHP, por eso la comparación debe ser estricta con !== false. Este es un error clásico que ya estas aprendiendo a evitar.

El formulario usa method="get" y el campo se llama "q" (de query, como en los buscadores reales). Fíjate en el atributo value del input: lleva el valor de \$búsqueda ya sanitizado. Esto es el repoblado en su forma más simple: si buscas "man", presionas Buscar y la página muestra los resultados, el campo conserva la palabra "man" en lugar de quedar vacío. Pequeño detalle, gran diferencia de experiencia.

Después del formulario, si se envió una búsqueda (isset(\$_GET["q"])), se muestra el encabezado con lo buscado (sanitizado, como siempre) y luego dos caminos: si hay resultados se listan en una lista ul con un foreach, y si no hay se muestra el mensaje amigable de "no se encontraron". Prueba buscar "man": verás manzana, mango y mandarina. Copia la dirección completa del navegador, pegala en otra pestana y verás los mismos resultados: esa es la magia de GET, el enlace guarda la búsqueda.

Ejemplo 3: formulario de registro con POST

Ahora cambiamos de camino: un registro de usuario con nombre, correo y contraseña. Como hay una contraseña de por medio y la acción "crea" algo (un usuario), el método correcto es POST. Los datos viajarán en el cuerpo de la petición, fuera de la vista.

```

<?php
$mensaje = "";
if (isset($_POST["registrar"])) {
    $nombre = trim($_POST["nombre"]);
    $correo = trim($_POST["correo"]);
    $clave = $_POST["clave"];
    if ($nombre === "" || $correo === "" || $clave === "") {
        $mensaje = "Todos los campos son obligatorios.";
    } elseif (filter_var($correo, FILTER_VALIDATE_EMAIL) === false) {
        $mensaje = "El correo no tiene un formato valido.";
    } elseif (strlen($clave) < 8) {
        $mensaje = "La contraseña debe tener al menos 8 caracteres.";
    } else {
        $mensaje = "Registro exitoso. Bienvenido, " . htmlspecialchars($nombre) .
        ".";
    }
}
?>

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Registro con POST</title>
</head>
<body>
    <h1>Crear cuenta</h1>
    <?php if ($mensaje !== "") { ?>
        <p><?php echo $mensaje; ?></p>
    <?php } ?>
    <form action="" method="post">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre"><br><br>
        <label for="correo">Correo:</label>
        <input type="email" id="correo" name="correo"><br><br>
        <label for="clave">Contraseña:</label>
        <input type="password" id="clave" name="clave"><br><br>
        <input type="submit" name="registrar" value="Registrarme">
    </form>
</body>
</html>

```

Explicación línea por línea. La variable \$mensaje guardara lo que se le va a decir al usuario. La condición isset(\$_POST["registrar"]) detecta si el formulario ya fue enviado: el botón submit tiene name="registrar", así que ese dato solo existe en \$_POST después de presionarlo. Es el patrón estandar para páginas que muestran el formulario y lo procesan en el mismo archivo.

Dentro del if se recogen los tres datos. Nombre y correo pasan por trim; la contraseña no, porque en las contraseñas los espacios podrían ser intencionales (aunque en la práctica muchos sistemas si los recortan; aquí la dejamos intacta para no complicar). Luego viene la validación en cadena con if / elseif: primero se revisa que ningun campo este vacío, después que el correo tenga formato valido con filter_var, después que la clave mida al menos 8 caracteres con strlen. Solo si pasa las tres pruebas se arma el mensaje de éxito, incluyendo el nombre sanitizado con htmlspecialchars. El orden importa: se revisa lo más básico primero y lo más específico después.

En el HTML, el formulario usa method="post": fíjate que la dirección del navegador quedará limpia después de enviar, sin rastro de los datos. Los campos usan type="text", type="email" (el navegador revisa por encima el formato) y type="password" (muestra puntitos). Cada label esta asociado a su campo. El mensaje se muestra arriba del formulario solo si no está vacío.

Prueba enviar el formulario con datos correctos y mira la barra del navegador: la dirección no muestra nada de lo que escribiste. Esa es la diferencia con el ejemplo 1. Ahora prueba dejar campos vacios, poner un correo sin arroba y una clave de 3 caracteres: verás cada mensaje de error en acción. Así se siente la validación del lado del servidor.

Ejemplo 4: validación de email y contraseña con reglas múltiples

Este ejemplo se enfoca solo en la validación fina de dos campos problematicos: el correo y la contraseña. La contraseña debe cumplir varias reglas a la vez (longitud minima, al menos un número, al menos una mayúscula), y cada regla que falle genera su propio mensaje. Es el patrón que usaras en registros reales.

```
<?php
$errores = array();
$correo = "";
if (isset($_POST["validar"])) {
    $correo = trim($_POST["correo"]);
    $clave = $_POST["clave"];
    if ($correo === "") {
        $errores[] = "El correo es obligatorio.";
    } elseif (filter_var($correo, FILTER_VALIDATE_EMAIL) === false) {
        $errores[] = "El correo no tiene un formato valido.";
    }
}
```

```

    if (strlen($clave) < 8) {
        $errores[] = "La contraseña debe tener al menos 8 caracteres.";
    }
    if (!preg_match("/[0-9]/", $clave)) {
        $errores[] = "La contraseña debe incluir al menos un numero.";
    }
    if (!preg_match("/[A-Z]/", $clave)) {
        $errores[] = "La contraseña debe incluir al menos una mayuscula.";
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Validacion de correo y clave</title>
</head>
<body>
    <h1>Valida tus datos</h1>
    <?php if (isset($_POST["validar"])) { ?>
        <?php if (count($errores) === 0) { ?>
            <p>Datos validos. El correo <?php echo htmlspecialchars($correo); ?> esta
correcto.</p>
            <?php } else { ?>
                <ul>
                    <?php foreach ($errores as $e) { ?>
                        <li><?php echo htmlspecialchars($e); ?></li>
                    <?php } ?>
                </ul>
            <?php } ?>
        <?php } ?>
        <form action="" method="post">
            <label for="correo">Correo:</label>
            <input type="text" id="correo" name="correo" value="<?php echo
htmlspecialchars($correo); ?>"><br><br>
            <label for="clave">Contraseña:</label>
            <input type="password" id="clave" name="clave"><br><br>
            <input type="submit" name="validar" value="Validar">
        </form>
    </body>
</html>

```

Explicación línea por línea. Aquí el patrón cambia: en lugar de una sola variable \$mensaje, se usa un arreglo \$errores donde se acumula un mensaje por cada regla que falle. Esta es la técnica profesional porque permite decirle al usuario todo lo que está mal de una sola vez, en lugar de corregirlo de a un error por intentó (qué es desesperante).

La validación del correo tiene dos pasos: primero que no este vacío, y solo si no está vacío se revisa el formato con filter_var. El elseif evita el doble mensaje: si el campo está vacío no tiene sentido decir además que el formato es invalido. La contraseña pasa por tres pruebas independientes con if separados (no elseif), porque aquí si queremos acumular todos los fallos: strlen para la longitud minima, y preg_match con dos patrones simples para exigir al menos un dígito ([0-9]) y al menos una mayúscula ([A-Z]). La función preg_match devuelve 1 si encuentra el patrón y 0 si no, así que el signo ! invierte la lógica: "si NO se encontró un número, agrega el error".

Al mostrar los resultados: si el formulario fue enviado y el arreglo de errores está vacío (count(\$errores) === 0), se felicita al usuario mostrando su correo sanitizado. Si hay errores, se listan en una ul recorriendo el arreglo con foreach, sanitizando cada mensaje con htmlspecialchars (aunque los mensajes los escribiste tu, el habito de sanitizar todo lo que se imprime no se negocia).

El campo de correo usa repoblado con value y htmlspecialchars, para que el usuario no tenga que reescribirlo si la clave falló. El campo de contraseña NO se repobla a proposito: por seguridad, las contraseñas nunca se devuelven a la pantalla, ni siquiera en el value. Es una buena práctica que debes adoptar desde ahora: lo sensible no se reimprime.

Prueba con la clave "abc": verás los tres mensajes de la contraseña juntos. Prueba con "Abcdef12": pasara todas las pruebas. Este ejemplo te entrena el ojo para el patrón "acumular errores en un arreglo", que es el corazon del siguiente ejemplo, el formulario de contacto completo.

Ejemplo 5: formulario de contacto completo con errores por campo

Este es el ejemplo más completo de la unidad: un formulario de contacto con nombre, correo y mensaje, donde cada campo muestra su propio error justo debajo, se repoblan los valores tras el falló y al final se confirma el envió. Es el patrón que usaras en casi todos tus proyectos reales.

```
<?php
$nombre = ""; $correo = ""; $mensaje = "";
$errores = array();
$enviado = false;
```



```

if (isset($_POST["contactar"])) {
    $nombre = trim($_POST["nombre"]);
    $correo = trim($_POST["correo"]);
    $mensaje = trim($_POST["mensaje"]);
    if ($nombre === "") {
        $errores["nombre"] = "Escribe tu nombre, por favor.";
    } elseif (strlen($nombre) < 3) {
        $errores["nombre"] = "El nombre debe tener al menos 3 letras.";
    }
    if ($correo === "") {
        $errores["correo"] = "Escribe tu correo, por favor.";
    } elseif (filter_var($correo, FILTER_VALIDATE_EMAIL) === false) {
        $errores["correo"] = "Ese correo no parece valido. Revisa que tenga @ y
dominio.";
    }
    if ($mensaje === "") {
        $errores["mensaje"] = "Escribe tu mensaje, por favor.";
    } elseif (strlen($mensaje) < 10) {
        $errores["mensaje"] = "El mensaje debe tener al menos 10 caracteres.";
    } elseif (strlen($mensaje) > 500) {
        $errores["mensaje"] = "El mensaje no puede pasar de 500 caracteres.";
    }
    if (count($errores) === 0) {
        $enviado = true;
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Contacto</title>
</head>
<body>
    <h1>Contactanos</h1>
    <?php if ($enviado) { ?>
        <p>Gracias, <?php echo htmlspecialchars($nombre); ?>. Recibimos tu mensaje y
te responderemos a <?php echo htmlspecialchars($correo); ?>.</p>
    <?php } else { ?>
        <form action="" method="post">
            <label for="nombre">Nombre:</label><br>
                <input type="text" id="nombre" name="nombre" value="<?php echo
htmlspecialchars($nombre); ?>"><br>
                    <?php if (isset($errores["nombre"])) { ?><span><?php echo
htmlspecialchars($errores["nombre"]); ?></span><br><?php } ?><br>

```

```

        <label for="correo">Correo:</label><br>
        <input type="text" id="correo" name="correo" value="<?php echo
htmlspecialchars($correo); ?>"><br>
        <?php if (isset($errores["correo"])) { ?><span><?php echo
htmlspecialchars($errores["correo"]); ?></span><br><?php } ?><br>
        <label for="mensaje">Mensaje:</label><br>
        <textarea id="mensaje" name="mensaje" rows="5" cols="40"><?php echo
htmlspecialchars($mensaje); ?></textarea><br>
        <?php if (isset($errores["mensaje"])) { ?><span><?php echo
htmlspecialchars($errores["mensaje"]); ?></span><br><?php } ?><br>
        <input type="submit" name="contactar" value="Enviar mensaje">
    </form>
    <?php } ?>
</body>
</html>

```

Explicación línea por línea. La preparación inicial declara las tres variables de datos vacías, el arreglo \$errores y la bandera \$enviado en false. La bandera es un truco limpio: en lugar de preguntar varias veces si hubo errores, se marca true solo cuando todo salió bien, y el HTML decide que mostrar según su valor.

La validación sigue el patrón de errores por campo: \$errores usa el nombre del campo como llave (\$errores["nombre"], \$errores["correo"], \$errores["mensaje"]). Cada campo tiene sus reglas: el nombre no vacío y con mínimo 3 letras, el correo no vacío y con formato válido, el mensaje no vacío, con mínimo 10 y máximo 500 caracteres. Fíjate que el máximo de 500 protege tu sistema de textos gigantescos. Si al final no hay errores, \$enviado se pone en true. (En un proyecto real, aquí guardarías el mensaje en una base de datos o lo enviarías por correo; en esta unidad el "procesamiento" es mostrar la confirmación.)

El HTML tiene dos grandes ramas: si \$enviado es true, se muestra el mensaje de agradecimiento con el nombre y el correo sanitizados, y el formulario desaparece (ya no tiene sentido mostrarlo). Si no, se muestra el formulario. Cada campo sigue el mismo micro-patrón de tres líneas: el label, el input con su value repoblado y sanitizado, y justo debajo un bloque que muestra el error de ese campo solo si existe en el arreglo. El textarea es especial: no usa value, el contenido va entre la etiqueta de apertura y la de cierre, también sanitizado.

Prueba enviarlo vacío: verás los tres errores, cada uno bajo su campo, con mensajes que dicen que hacer. Corrige solo el nombre y reenvía: el nombre se conserva (re poblado) y solo quedan los otros dos errores. Esa es la experiencia profesional completa: errores específicos, campos que recuerdan, y confirmación clara al final. Estudia este ejemplo hasta poder escribirlo de memoria, porque su estructura se repite en registro de usuarios, publicación de comentarios y casi cualquier formulario con validación.

Ejemplo 6: calculadora con formulario

Una calculadora que recibe dos números y la operación desde un formulario. Sirve para practicar la validación numerica con `is_numeric`, la conversión de tipos y el manejo de un caso especial famoso: la división entre cero.

```
<?php
$num1 = ""; $num2 = ""; $operacion = "sumar";
$resultado = null;
$error = "";
if (isset($_POST["calcular"])) {
    $num1 = trim($_POST["num1"]);
    $num2 = trim($_POST["num2"]);
    $operacion = $_POST["operacion"];
    if ($num1 === "" || $num2 === "") {
        $error = "Escribe los dos numeros.";
    } elseif (!is_numeric($num1) || !is_numeric($num2)) {
        $error = "Ambos valores deben ser numeros.";
    } elseif ($operacion === "dividir" && (float)$num2 == 0) {
        $error = "No se puede dividir entre cero.";
    } else {
        $a = (float)$num1;
        $b = (float)$num2;
        if ($operacion === "sumar") { $resultado = $a + $b; }
        elseif ($operacion === "restar") { $resultado = $a - $b; }
        elseif ($operacion === "multiplicar") { $resultado = $a * $b; }
        elseif ($operacion === "dividir") { $resultado = $a / $b; }
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Calculadora</title>
</head>
<body>
    <h1>Calculadora</h1>
    <?php if ($error !== "") { ?><p><?php echo htmlspecialchars($error); ?></p><?php }
    ?>
    <form action="" method="post">
        <label for="num1">Numero 1:</label>
        <input type="text" id="num1" name="num1" value="<?php echo
htmlspecialchars($num1); ?>"><br><br>
        <label for="operacion">Operacion:</label>
```

```

        <select id="operacion" name="operacion">
            <option value="sumar" <?php if ($operacion === "sumar") echo "selected";
?>>Sumar</option>
            <option value="restar" <?php if ($operacion === "restar") echo "selected";
?>>Restar</option>
            <option value="multiplicar" <?php if ($operacion === "multiplicar") echo
"selected"; ?>>Multiplicar</option>
            <option value="dividir" <?php if ($operacion === "dividir") echo
"selected"; ?>>Dividir</option>
        </select><br><br>
        <label for="num2">Numero 2:</label>
        <input type="text" id="num2" name="num2" value="<?php echo
htmlspecialchars($num2); ?>"><br><br>
        <input type="submit" name="calcular" value="Calcular">
    </form>
    <?php if ($resultado !== null) { ?>
        <p>El resultado es: <?php echo htmlspecialchars($resultado); ?></p>
    <?php } ?>
</body>
</html>

```

Explicación línea por línea. Las variables iniciales: \$num1 y \$num2 como cadenas vacías, \$operación con valor por defecto "sumar" (así el select muestra Sumar la primera vez), \$resultado en null (null significa "todavía no hay resultado", distinto de 0 que sí sería un resultado válido) y \$error vacío.

La validación tiene cuatro niveles en orden lógico: primero que no vengan vacíos, después que ambos sean numéricos con `is_numeric` (atrapa letras y símbolos), después el caso especial de la división entre cero. Fíjate en ese tercer nivel: se convierte \$num2 a float antes de comparar, porque el dato llega como cadena y "0" como texto necesita conversión para compararse bien como número. Solo si pasa las tres pruebas se convierten ambos a float y se hace la operación según el valor del select, con una cadena de `if / elseif`.

En el HTML, el select merece atención: cada option tiene un value ("sumar", "restar...") que es lo que viaja a PHP, y un texto visible ("Sumar"...) que es lo que ve el usuario. El repoblado de un select no usa value sino el atributo selected: cada option pregunta si \$operación coincide con su valor, y si sí, imprime selected para quedar marcada. Es el equivalente del repoblado para listas desplegables, y es uno de los detalles que más se les olvida a los principiantes.

El resultado se muestra solo si \$resultado no es null, es decir, solo cuando el cálculo se hizo de verdad. Prueba dividir 10 entre 0 y verás el mensaje del caso especial. Prueba escribir "hola" en un número y verás el error de `is_numeric`. Y nota algo importante: aunque los campos son `type="text"`, la validación numérica la hace PHP, no el navegador. Así debe ser.

Ejemplo 7: encuesta de gustos con checkboxes

Los checkboxes permiten elegir varias opciones a la vez, y PHP los recibe como un arreglo. El truco está en el name: se escribe con corchetes al final, name="gustos[]", para indicarle a PHP que agrupe todos los valores marcados en un solo arreglo.

```
<?php
$gustos = array();
$nombre = "";
if (isset($_POST["votar"])) {
    $nombre = trim($_POST["nombre"]);
    if (isset($_POST["gustos"])) {
        $gustos = $_POST["gustos"];
    }
}
$opciones = array("musica", "cine", "deportes", "lectura", "videojuegos", "cocina");
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Encuesta de gustos</title>
</head>
<body>
    <h1>Encuesta: que te gusta?</h1>
    <form action="" method="post">
        <label for="nombre">Tu nombre:</label>
        <input type="text" id="nombre" name="nombre" value="<?php echo
htmlspecialchars($nombre); ?>"><br><br>
        <p>Marca tus gustos (puedes elegir varios):</p>
        <?php foreach ($opciones as $op) { ?>
            <label>
                <input type="checkbox" name="gustos[]" value="<?php echo $op; ?>"
                <?php if (in_array($op, $gustos)) echo "checked"; ?>>
                <?php echo ucfirst($op); ?>
            </label><br>
        <?php } ?>
        <br>
        <input type="submit" name="votar" value="Votar">
    </form>
    <?php if (isset($_POST["votar"])) { ?>
        <h2>Resultados</h2>
        <?php if ($nombre !== "") { ?>
            <p><?php echo htmlspecialchars($nombre); ?> marco <?php echo
count($gustos); ?> gusto(s):</p>
```

```

        <?php } ?>
        <?php if (count($gustos) > 0) { ?>
            <ul>
                <?php foreach ($gustos as $g) { ?>
                    <li><?php echo htmlspecialchars($g); ?></li>
                <?php } ?>
            </ul>
        <?php } else { ?>
            <p>No marcaste ningun gusto.</p>
        <?php } ?>
    <?php } ?>
</body>
</html>

```

Explicación línea por línea. La preparación define \$gustos como arreglo vacío y las seis opciones de la encuesta en \$opciones. Cuando se envía el formulario, se recoge el nombre con trim y luego viene el detalle clave: if (isset(\$_POST["gustos"])). Si el usuario no marco ninguna casilla, el navegador no manda el dato "gustos" en absoluto, así que \$_POST["gustos"] no existe y hay que preguntar con isset antes de usarlo. Si si marco al menos una, \$_POST["gustos"] ya es un arreglo con los valores marcados, gracias a los corchetes del name. Este es el error típico: asumir que el arreglo siempre llega y obtener un aviso de "índice no definido" cuando no se marco nada.

El formulario genera las casillas con un foreach sobre \$opciones: cada checkbox tiene name="gustos[]" (los corchetes son la magia) y value con la opción. El repoblado de checkboxes usa el atributo checked: in_array(\$op, \$gustos) pregunta si esa opción esta entre las marcadas, y si si, imprime checked para que siga marcada tras enviar. El texto visible usa ucfirst para poner la primera letra en mayúscula ("música" se muestra como "Música").

Los resultados se muestran solo si se voto: el nombre sanitizado, cuantos gustos marco con count, y la lista de gustos recorriendo el arreglo con foreach y sanitizando cada uno. Si no marco ninguno, el mensaje lo dice amablemente en lugar de mostrar una lista vacía. Prueba marcar varias opciones y observa como llegan todas en el arreglo. Y prueba no marcar ninguna: gracias al isset, la página no se rompe.

Nota de seguridad para cuando avances: en un sistema real también validarías que los valores recibidos esten dentro de \$opciones, porque un atacante podría mandar valores inventados. En esta unidad basta con que entiendas el mecanismo de los corchetes y el isset.

Ejemplo 8: repoblado de campos tras error (técnica completa)

El último ejemplo aísla y practica la técnica de repoblado en todos los tipos de campo a la vez: texto, número, select, radio y checkbox. Es tu plantilla de referencia: cuando en el futuro dudes como repoblar un campo, vuelve aquí.

```
<?php
$nombre = ""; $edad = ""; $pais = ""; $genero = ""; $boletin = false;
$errores = array();
if (isset($_POST["guardar"])) {
    $nombre = trim($_POST["nombre"]);
    $edad = trim($_POST["edad"]);
    $pais = isset($_POST["pais"]) ? $_POST["pais"] : "";
    $genero = isset($_POST["genero"]) ? $_POST["genero"] : "";
    $boletin = isset($_POST["boletin"]);
    if ($nombre === "") { $errores["nombre"] = "El nombre es obligatorio."; }
    if ($edad === "" || !is_numeric($edad) || (int)$edad < 1) {
        $errores["edad"] = "Escribe una edad valida.";
    }
    if ($pais === "") { $errores["pais"] = "Elige un pais."; }
    if ($genero === "") { $errores["genero"] = "Elige una opcion."; }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Repoblado completo</title>
</head>
<body>
    <h1>Registro con repoblado</h1>
    <form action="" method="post">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" value="<?php echo
htmlspecialchars($nombre); ?>">
        <?php if (isset($errores["nombre"])) echo "<p>" .
htmlspecialchars($errores["nombre"]) . "</p>"; ?><br>
        <label for="edad">Edad:</label>
        <input type="number" id="edad" name="edad" value="<?php echo
htmlspecialchars($edad); ?>">
        <?php if (isset($errores["edad"])) echo "<p>" .
htmlspecialchars($errores["edad"]) . "</p>"; ?><br>
        <label for="pais">Pais:</label>
        <select id="pais" name="pais">
            <option value="">-- Elige --</option>
```

```

        <option value="mx" <?php if ($pais === "mx") echo "selected";
?>>Mexico</option>
        <option value="ar" <?php if ($pais === "ar") echo "selected";
?>>Argentina</option>
        <option value="co" <?php if ($pais === "co") echo "selected";
?>>Colombia</option>
    </select>

        <?php if (isset($errores["pais"])) echo "<p>" .
htmlspecialchars($errores["pais"]) . "</p>"; ?><br>
        <p>Genero:</p>
        <label><input type="radio" name="genero" value="f" <?php if ($genero === "f")
echo "checked"; ?>> Femenino</label>
        <label><input type="radio" name="genero" value="m" <?php if ($genero === "m")
echo "checked"; ?>> Masculino</label>
        <?php if (isset($errores["genero"])) echo "<p>" .
htmlspecialchars($errores["genero"]) . "</p>"; ?><br>
        <label><input type="checkbox" name="boletin" <?php if ($boletin) echo
"checked"; ?>> Quiero recibir el boletin</label><br><br>
        <input type="submit" name="guardar" value="Guardar">
    </form>
    <?php if (isset($_POST["guardar"]) && count($errores) === 0) { ?>
        <p>Datos guardados correctamente.</p>
    <?php } ?>
</body>
</html>

```

Explicación línea por línea. En la preparación, cada variable nace con su valor "vacío" según su tipo: cadenas vacías para texto, número y select, cadena vacía para el radio, y false para el checkbox del boletin. Los datos que pueden no llegar (país, generó, boletin) se recogen con `isset` y un valor por defecto, usando el operador ternario: `$país = isset($_POST["país"]) ? $_POST["país"] : ""`; El radio y el select siempre llegan si el usuario eligió algo, pero si no eligió nada el dato no existe, así que el `isset` es obligatorio. El checkbox es booleano puro: `isset($_POST["boletin"])` ya te dice si se marcó o no.

La validación es directa: nombre obligatorio, edad obligatoria además de numérica y mayor que cero (fíjate en la combinación de tres condiciones con `||`: cualquiera que falle marca el error), país y generó con selección obligatoria. El boletin no se valida porque es opcional.

Y ahora el repoblado, tipo por tipo, que es la estrella del ejemplo: los input de texto y número usan value con htmlspecialchars; el select usa selected en la option que coincida con \$país (incluyendo la option vacía "-- Elige --" que queda seleccionada por defecto la primera vez); los radio usan checked en el que coincida con \$generó (los radio comparten el mismo name="generó" pero distinto value: así el navegador sabe que solo se puede elegir uno); el checkbox usa checked si \$boletin es true.

Cada campo muestra su error justo debajo con el mismo micro-patrón: si existe en el arreglo, se imprime sanitizado. Al final, si se envió y no hay errores, se confirma que los datos se guardaron.

Haz este ejercicio mental con el ejemplo: envía el formulario eligiendo todo menos el país. Verás que el nombre, la edad, el género y el boletin se conservan, el país vuelve a "-- Elige --" y solo aparece el error del país. Eso es repoblado bien hecho: el usuario corrige una cosa y listo. Si dominas esta plantilla, dominas cualquier formulario.

Práctica

Ejercicio 1

Crea una página llamada edad.php con un formulario GET que pida la edad del usuario y le diga en que etapa de la vida esta: si es menor de 13 es "nino", si es menor de 18 es "adolescente", si es menor de 60 es "adulto" y si es mayor o igual a 60 es "adulto mayor". Debes validar que el dato llegue, que no este vacío y que sea numerico. Si la validación falla, muestra un mensaje de error amigable en lugar de la etapa.

Solución 1

```

<?php
$mensaje = "";
if (isset($_GET["edad"])) {
    $edad = trim($_GET["edad"]);
    if ($edad === "") {
        $mensaje = "Escribe tu edad, por favor.";
    } elseif (!is_numeric($edad)) {
        $mensaje = "La edad debe ser un numero.";
    } elseif ((int)$edad < 0) {
        $mensaje = "La edad no puede ser negativa.";
    } else {
        $e = (int)$edad;
        if ($e < 13) { $etapa = "nino"; }
        elseif ($e < 18) { $etapa = "adolescente"; }
        elseif ($e < 60) { $etapa = "adulto"; }
        else { $etapa = "adulto mayor"; }
        $mensaje = "Tienes " . $e . " anos: eres " . $etapa . ".";
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Etapas de la vida</title>
</head>
<body>
    <h1>En que etapa estas?</h1>
    <form action="" method="get">
        <label for="edad">Tu edad:</label>
        <input type="text" id="edad" name="edad">
        <input type="submit" value="Descubrir">
    </form>
    <?php if ($mensaje !== "") { ?>
        <p><?php echo htmlspecialchars($mensaje); ?></p>
    <?php } ?>
</body>
</html>

```

Explicación paso a paso. Primero se detecta si el formulario fue enviado con `isset($_GET["edad"])` y se guarda el valor con `trim`. La validación va en cascada: vacío, no numerico, negativo. Cada falla produce su mensaje amigable. Solo si pasa las tres pruebas se convierte a entero con `(int)` y se clasifica con la cadena de `if / elseif`, de menor a mayor: como cada condición atrapa a los menores que su limite, el orden ascendente funciona perfecto (un

nino de 10 cumple \$e < 13 y ya no se revisa lo demás). El mensaje final se arma concatenando, y al mostrarlo se pasa por htmlspecialchars por el habito de seguridad, aunque el contenido lo generamos nosotros. Se usa GET porque la edad no es sensible y el resultado se puede compartir como enlace. Prueba con 8, 15, 30, 70, con "hola" y con el campo vacío para ver cada rama.

Ejercicio 2

Crea un formulario POST de inicio de sesión (login.php) con campos usuario y contraseña. Valida que ninguno este vacío y que el usuario tenga al menos 4 caracteres y la contraseña al menos 6. Si todo está bien, muestra "Bienvenido, [usuario]". Si hay errores, muéstralos todos juntos en una lista. El campo usuario debe repoblarse tras un error, pero la contraseña no.

Solución 2

```
<?php
$usuario = "";
$errores = array();
$ok = false;
if (isset($_POST["entrar"])) {
    $usuario = trim($_POST["usuario"]);
    $clave = $_POST["clave"];
    if ($usuario === "") {
        $errores[] = "El usuario es obligatorio.";
    } elseif (strlen($usuario) < 4) {
        $errores[] = "El usuario debe tener al menos 4 caracteres.";
    }
    if ($clave === "") {
        $errores[] = "La contraseña es obligatoria.";
    } elseif (strlen($clave) < 6) {
        $errores[] = "La contraseña debe tener al menos 6 caracteres.";
    }
    if (count($errores) === 0) {
        $ok = true;
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Iniciar sesion</title>
</head>
```

```

<body>
  <h1>Iniciar sesion</h1>
  <?php if ($ok) { ?>
    <p>Bienvenido, <?php echo htmlspecialchars($usuario); ?>.</p>
  <?php } else { ?>
    <?php if (count($errores) > 0) { ?>
      <ul>
        <?php foreach ($errores as $e) { ?>
          <li><?php echo htmlspecialchars($e); ?></li>
        <?php } ?>
      </ul>
    <?php } ?>
    <form action="" method="post">
      <label for="usuario">Usuario:</label>
      <input type="text" id="usuario" name="usuario" value="<?php echo
htmlspecialchars($usuario); ?>"><br><br>
      <label for="clave">Contraseña:</label>
      <input type="password" id="clave" name="clave"><br><br>
      <input type="submit" name="entrar" value="Entrar">
    </form>
  <?php } ?>
</body>
</html>

```

Explicación paso a paso. Se usa el patrón de arreglo de errores para acumular todos los fallos y mostrarlos juntos en una lista, como en el ejemplo 4. Cada campo tiene dos reglas: obligatorio y longitud mínima, unidas con if / elseif para no duplicar mensajes del mismo campo. La bandera \$ok marca el éxito. En el HTML hay dos ramas: si \$ok, mensaje de bienvenida con el usuario sanitizado y el formulario desaparece; si no, se muestra la lista de errores (solo si hay) y el formulario. El campo usuario lleva value con repoblado sanitizado; el campo contraseña no lleva value, siguiendo la buena práctica de no reimprimir datos sensibles. Se usa POST porque hay una contraseña de por medio: verifica que la dirección quede limpia tras enviar. Prueba con usuario "ana" (3 letras) y clave "123" para ver los dos errores de longitud juntos.

Ejercicio 3

Crea un convertidor de temperatura con GET: el usuario escribe grados y elige en un select si quiere convertir de Celsius a Fahrenheit o de Fahrenheit a Celsius. Valida que el valor sea numérico. Las formulas son: $F = C * 9/5 + 32$ y $C = (F - 32) * 5/9$. Muestra el resultado con 1 decimal usando round. El select debe conservar la opción elegida tras convertir (re poblado con selected).

Solución 3

```
<?php
$grados = "";
$direccion = "c_a_f";
$resultado = null;
$error = "";
if (isset($_GET["convertir"])) {
    $grados = trim($_GET["grados"]);
    $direccion = $_GET["direccion"];
    if ($grados === "") {
        $error = "Escribe los grados a convertir.";
    } elseif (!is_numeric($grados)) {
        $error = "Los grados deben ser un numero.";
    } else {
        $g = (float)$grados;
        if ($direccion === "c_a_f") {
            $resultado = $g * 9 / 5 + 32;
        } else {
            $resultado = ($g - 32) * 5 / 9;
        }
        $resultado = round($resultado, 1);
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Convertidor de temperatura</title>
</head>
<body>
    <h1>Convertidor de temperatura</h1>
    <?php if ($error !== "") { ?><p><?php echo htmlspecialchars($error); ?></p><?php }
    ?>
    <form action="" method="get">
        <label for="grados">Grados:</label>
        <input type="text" id="grados" name="grados" value="<?php echo
htmlspecialchars($grados); ?>">
        <label for="direccion">Convertir:</label>
        <select id="direccion" name="direccion">
            <option value="c_a_f" <?php if ($direccion === "c_a_f") echo "selected";
?>>Celsius a Fahrenheit</option>
            <option value="f_a_c" <?php if ($direccion === "f_a_c") echo "selected";
?>>Fahrenheit a Celsius</option>
        </select>
```

```

        <input type="submit" name="convertir" value="Convertir">
    </form>
    <?php if ($resultado !== null) { ?>
        <p>Resultado: <?php echo htmlspecialchars($resultado); ?> grados.</p>
    <?php } ?>
</body>
</html>

```

Explicación paso a paso. La variable \$dirección nace con "c_a_f" como valor por defecto, así el select muestra la primera opción al cargar. El botón submit tiene name="convertir" para detectar el envío con isset. La validación revisa vacío y numerico con is_numeric; si pasa, se convierte a float y se aplica la fórmula según la dirección elegida. La función round(\$resultado, 1) redondea a un decimal para que el resultado se vea limpio (por ejemplo 98.6 en lugar de 98.6000000001). El repoblado cubre los dos tipos de campo: el input con value sanitizado y el select con selected en la option que coincida. El resultado se muestra solo si no es null. Se usa GET porque es una conversión informativa y compatible: convierte 100 grados de Celsius a Fahrenheit, copia la dirección y mandasela a alguien para comprobarlo. Prueba también escribir "mucho calor" para ver el error de validación.

Ejercicio 4

Crea una página de registro a un taller (taller.php) con POST que pida: nombre completo, correo, teléfono (solo dígitos, entre 8 y 15 caracteres) y turno (radio: matutino o vespertino). Valida cada campo con su propio mensaje de error guardado en un arreglo asociativo, repobla todos los campos tras un error (incluyendo el radio con checked) y muestra los errores junto a su campo. Si todo está bien, muestra un resumen de los datos registrados.

Solución 4

```

<?php
$nombre = ""; $correo = ""; $telefono = ""; $turno = "";
$errores = array();
$ok = false;
if (isset($_POST["inscribir"])) {
    $nombre = trim($_POST["nombre"]);
    $correo = trim($_POST["correo"]);
    $telefono = trim($_POST["telefono"]);
    $turno = isset($_POST["turno"]) ? $_POST["turno"] : "";
    if ($nombre === "") {
        $errores["nombre"] = "El nombre es obligatorio.";
    } elseif (strlen($nombre) < 5) {
        $errores["nombre"] = "Escribe tu nombre completo (al menos 5 letras).";
    }
}

```

```

    }
    if ($correo === "") {
        $errores["correo"] = "El correo es obligatorio.";
    } elseif (filter_var($correo, FILTER_VALIDATE_EMAIL) === false) {
        $errores["correo"] = "Escribe un correo valido, con @ y dominio.";
    }
    if ($telefono === "") {
        $errores["telefono"] = "El telefono es obligatorio.";
    } elseif (!ctype_digit($telefono)) {
        $errores["telefono"] = "El telefono solo puede tener digitos, sin espacios ni
guiones.";
    } elseif (strlen($telefono) < 8 || strlen($telefono) > 15) {
        $errores["telefono"] = "El telefono debe tener entre 8 y 15 digitos.";
    }
    if ($turno === "") {
        $errores["turno"] = "Elige un turno.";
    }
    if (count($errores) === 0) { $ok = true; }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Inscripcion al taller</title>
</head>
<body>
    <h1>Inscripcion al taller de PHP</h1>
    <?php if ($ok) { ?>
        <h2>Inscripcion exitosa</h2>
        <p>Nombre: <?php echo htmlspecialchars($nombre); ?></p>
        <p>Correo: <?php echo htmlspecialchars($correo); ?></p>
        <p>Telefono: <?php echo htmlspecialchars($telefono); ?></p>
        <p>Turno: <?php echo htmlspecialchars($turno); ?></p>
    <?php } else { ?>
        <form action="" method="post">
            <label for="nombre">Nombre completo:</label>
            <input type="text" id="nombre" name="nombre" value="<?php echo
htmlspecialchars($nombre); ?>">
            <?php if (isset($errores["nombre"])) echo "<p>" .
htmlspecialchars($errores["nombre"]) . "</p>"; ?><br>
            <label for="correo">Correo:</label>
            <input type="text" id="correo" name="correo" value="<?php echo
htmlspecialchars($correo); ?>">
            <?php if (isset($errores["correo"])) echo "<p>" .

```

```

htmlspecialchars($errores["correo"]) . "</p>"; ?><br>
    <label for="telefono">Telefono:</label>
        <input type="text" id="telefono" name="telefono" value="<?php echo
htmlspecialchars($telefono); ?>">
            <?php if (isset($errores["telefono"])) echo "<p>" .
htmlspecialchars($errores["telefono"]) . "</p>"; ?><br>
        <p>Turno:</p>
        <label><input type="radio" name="turno" value="matutino" <?php if ($turno ==
"matutino") echo "checked"; ?>> Matutino</label>
        <label><input type="radio" name="turno" value="vespertino" <?php if ($turno
=== "vespertino") echo "checked"; ?>> Vespertino</label>
            <?php if (isset($errores["turno"])) echo "<p>" .
htmlspecialchars($errores["turno"]) . "</p>"; ?><br>
        <input type="submit" name="inscribir" value="Inscribirme">
    </form>
    <?php } ?>
</body>
</html>

```

Explicación paso a paso. Este ejercicio combina todo lo aprendido: validación por campo con arreglo asociativo de errores, repoblado completo y confirmación con resumen. La novedad es la validación del teléfono con `ctype_digit`, una función que revisa si todos los caracteres de la cadena son dígitos (más estricta que `is_numeric`, que aceptaría puntos y signos). Las tres reglas del teléfono van en cascada: obligatorio, solo dígitos, longitud entre 8 y 15. El turno se recoge con `isset` y ternario porque los radio no mandan nada si no se elige ninguno.

El repoblado cubre texto (value sanitizado) y radio (checked cuando coincide el valor). Los errores aparecen junto a su campo con el micro-patrón ya conocido. Si todo sale bien, la bandera `$ok` muestra el resumen con los cuatro datos sanitizados y el formulario desaparece. Prueba enviar con teléfono "55-1234" (tiene guion) para ver el error de `ctype_digit`, y con solo 5 dígitos para ver el error de longitud. Este ejercicio es, en miniatura, un registro de evento real.

Ejercicio 5

Crea un formulario GET de filtro de productos: un campo de texto para buscar por nombre y un select de categoría (todas, electrónica, ropa, libros). Tienes un catálogo en un arreglo de arreglos con nombre, categoría y precio. Muestra solo los productos que coincidan con el texto (búsqueda parcial, sin importar mayúsculas) y con la categoría elegida. Si no se ha buscado nada, muestra todos los productos. La URL debe reflejar el filtro para poder compartirla.

Solución 5

```
<?php
$productos = array(
    array("nombre" => "Laptop", "categoria" => "electronica", "precio" => 15000),
    array("nombre" => "Telefono", "categoria" => "electronica", "precio" => 8000),
    array("nombre" => "Playera", "categoria" => "ropa", "precio" => 300),
    array("nombre" => "Pantalon", "categoria" => "ropa", "precio" => 600),
    array("nombre" => "Libro de PHP", "categoria" => "libros", "precio" => 450),
    array("nombre" => "Libro de cocina", "categoria" => "libros", "precio" => 380)
);
$texto = "";
$categoria = "todas";
if (isset($_GET["filtrar"])) {
    $texto = trim($_GET["texto"]);
    $categoria = $_GET["categoria"];
}
$mostrar = array();
foreach ($productos as $p) {
    $coincideTexto = ($texto === "") || (stripos($p["nombre"], $texto) !== false);
    $coincideCat = ($categoria === "todas") || ($p["categoria"] === $categoria);
    if ($coincideTexto && $coincideCat) {
        $mostrar[] = $p;
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Catalogo con filtros</title>
</head>
<body>
    <h1>Catalogo de productos</h1>
    <form action="" method="get">
        <label for="texto">Buscar:</label>
        <input type="text" id="texto" name="texto" value="<?php echo
htmlspecialchars($texto); ?>">
        <label for="categoria">Categoria:</label>
        <select id="categoria" name="categoria">
            <option value="todas" <?php if ($categoria === "todas") echo "selected";
?>>Todas</option>
            <option value="electronica" <?php if ($categoria === "electronica") echo
"selected"; ?>>Electronica</option>
            <option value="ropa" <?php if ($categoria === "ropa") echo "selected";
?>>Ropa</option>
```

```

        <option value="libros" <?php if ($categoria === "libros") echo "selected";
?>>Libros</option>
    </select>
    <input type="submit" name="filtrar" value="Filtrar">
</form>
<h2>Productos (<?php echo count($mostrar); ?>)</h2>
<?php if (count($mostrar) > 0) { ?>
    <ul>
        <?php foreach ($mostrar as $p) { ?>
            <li><?php echo htmlspecialchars($p["nombre"]); ?> - <?php echo
htmlspecialchars($p["categoria"]); ?> - $<?php echo $p["precio"]; ?></li>
        <?php } ?>
    </ul>
    <?php } else { ?>
        <p>Ningun producto coincide con el filtro.</p>
    <?php } ?>
</body>
</html>

```

Explicación paso a paso. El catálogo es un arreglo de arreglos asociativos, como los registros de una base de datos simplificada. Los valores por defecto (\$texto vacío, \$categoría "todas") hacen que, antes de filtrar, se muestren todos los productos: la condición de coincidencia de texto es verdadera cuando el texto está vacío, y la de categoría cuando es "todas". Solo después de presionar Filtrar (detectado con `isset($_GET["filtrar"])`) se toman los valores del formulario.

El filtrado recorre cada producto y calcula dos booleanos: \$coincideTexto usa `stripos` con comparación estricta `!== false` para la búsqueda parcial sin importar mayúsculas (el truco ya visto en el ejemplo 2), y \$coincideCat compara la categoría o acepta todo si es "todas". Solo si ambas son verdaderas el producto entra a \$mostrar. Es un filtro AND de dos criterios, el patrón base de cualquier buscador con filtros múltiples.

El formulario repobla el texto con `value` y el `select` con `selected`, así el filtro se mantiene visible tras aplicarlo. Los resultados muestran el conteo y la lista, con cada dato de texto sanitizado (el precio es número generado por nosotros, no necesita filtro). Cómo es GET, la URL refleja el filtro: busca "libro" en categoría "libros", copia la dirección y compartela; quien la abra vera el mismo resultado. Este ejercicio es el puente directo hacia las tiendas en línea reales.

Resumen de la unidad

- Los formularios web son el puente entre el usuario y PHP: el navegador empaqueta los datos y PHP los recibe, los valida y responde.

- La etiqueta form necesita action (a donde van los datos) y method (por qué camino: get o post); cada campo que quieras recibir debe tener el atributo name, sin excepción.
- En una petición GET los datos viajan visibles en la URL (la postal): sirve para búsquedas, filtros y paginación, y los resultados se pueden compartir como enlace. Jamás la uses para datos sensibles.
- En una petición POST los datos viajan ocultos en el cuerpo de la petición (el sobre cerrado): sirve para registros, inicios de sesión y acciones que cambian algo en el servidor.
- `$_GET` y `$_POST` son arreglos superglobales donde PHP pone los datos recibidos, usando el name de cada campo como llave. `$_REQUEST` los mezcla y es mejor evitarlo para no perder claridad ni control.
- Nunca confíes en lo que manda el usuario: la validación del navegador es solo una cortesía, la validación de verdad se hace siempre en el servidor con PHP.
- Las herramientas de validación: `isset` (llegó el dato?), `empty` (viene vacío?), `trim` (quita espacios de los extremos), `strlen` (mide la longitud), `is_numeric` (es número?), `ctype_digit` (son solo dígitos?) y `filter_var` con `FILTER_VALIDATE_EMAIL` (valida correos como profesional).
- Sanitiza al mostrar: pasa por `htmlspecialchars` todo dato que venga del usuario antes de imprimirlo, para evitar inyección de código.
- Repobla el formulario tras un error (`value` para inputs, `selected` para select, `checked` para radio y checkbox) para que el usuario solo corrija lo que falló; las contraseñas nunca se repoblan.
- Muestra mensajes de error específicos junto a cada campo, con tono de ayuda y no de regano, usando un arreglo de errores con el nombre del campo como llave.
- Errores típicos a evitar: olvidar el name, usar GET para contraseñas, no validar en el servidor, confundir `isset` con `empty` y mostrar datos sin sanitizar.
- Con lo aprendido ya puedes construir saludos personalizados, buscadores, registros con validación, formularios de contacto profesionales, calculadoras, encuestas y filtros de catálogo: la base de casi cualquier aplicación web real.

UNIDAD 6: COOKIES Y SESIONES

Bienvenido a la Unidad 6 del Curso Completo de PHP de Lucas EGA Academy. Esta unidad es una de las más importantes de todo el curso, porque aquí vas a aprender algo que cambia por completo la forma en que ves las páginas web: cómo hacer que PHP "recuerde" a las personas que visitan tu sitio.

Piensa por un momento en tu página favorita de compras. Entrás un lunes, agregas un producto al carrito, cierras la pestaña y te vas a cenar. El martes vuelves a entrar... y el producto sigue ahí, esperandote en el carrito. O piensa en tu correo electrónico: entras una vez con tu usuario y contraseña, y durante días el sitio te reconoce sin pedirte que inicies sesión otra vez. O cuando una página te dice "Bienvenido de nuevo, Esteban" aunque hayan pasado dos semanas desde tu última visita. Nada de eso es magia. Detrás de cada una de esas experiencias hay cookies y sesiones trabajando, y al terminar esta unidad tú vas a saber exactamente cómo funcionan y cómo usarlas en tus propios proyectos.

En esta unidad vamos a cubrir tres clases de tu ruta de aprendizaje: la clase 16, donde aprenderás todo sobre las cookies; la clase 17, donde dominarás las sesiones; y la clase 18, donde unirás todo para construir un sistema de login básico completo y funcional, con formulario de acceso, página protegida y botón para cerrar sesión. Es un sistema real, del tipo que se usa en miles de sitios web pequeños y medianos, y cuando lo tengas funcionando vas a sentir que por fin eres un programador web de verdad.

Pero antes de entrar en código, necesitas entender el problema. Y el problema es este: HTTP, el protocolo con el que tu navegador habla con los servidores, no tiene memoria. Cada vez que tu navegador pide una página, el servidor la entrega y se olvida por completo de ti. Es como si cada petición fuera la primera vez que te ve.

Imagínate que entras a un restaurante y el mesero te atiende muy bien. Le pides una limonada, te la trae, todo perfecto. Te paras al baño y regresas a tu mesa... y el mesero te mira como si nunca te hubiera visto. "Hola, bienvenido, ¿qué le ofrezco?" Tu le dices: "Oye, soy yo, el de la limonada". Y él te responde: "Perdón, no lo recuerdo". Vas al baño otra vez, regresas, y otra vez lo mismo. Sería exasperante, ¿verdad? Pues así se comporta HTTP por defecto: cada petición es un cliente nuevo para el servidor, un completo desconocido.

Para que el servidor pueda reconocerte, necesita un mecanismo para dejarte una marca, algo que le diga "este cliente ya estuvo aquí". Y aquí es donde entran nuestras dos herramientas.

La primera herramienta son las cookies. Siguiendo con la analogía del restaurante, una cookie es como si el mesero te diera un gafete de visitante al entrar: "Hola, soy Esteban, mesa 5, me

gusta la limonada". Tu te cuelgas el gafete y cada vez que el mesero te ve, lee tu gafete y ya sabe quien eres. La clave es que el gafete lo guardas TU, en tu navegador, y se lo muestras al servidor en cada visita. Así el servidor no necesita recordarte: tu le recuerdas quien eres.

La segunda herramienta son las sesiones. En la analogía del restaurante, una sesión es como un locker con llave. El mesero te da una llavecita con un número (ese número es el famoso "id de sesión") y guarda tus cosas en un locker del restaurante con ese mismo número. Tu solo cargas la llavecita, que es pequeña y ligera; las cosas pesadas (tu información) las guarda el restaurante en su locker. Cada vez que llegas, muestras tu llave, el mesero abre el locker correspondiente y ahí esta todo lo tuyo.

Por qué necesitamos las dos cosas y no solo una? Porque cada una tiene ventajas y desventajas. Las cookies son fáciles y duran mucho tiempo (pueden vivir meses en tu navegador), pero cómo se guardan en la computadora del visitante, no son un lugar seguro para información delicada: cualquiera podría leerlas o modificarlas. Las sesiones son mucho más seguras porque la información vive en el servidor, pero duran menos (normalmente se borran cuando cierras el navegador) y consumen recursos del servidor.

En la vida real los sitios combinan las dos: usan una cookie pequenita para guardar el número de llave (el id de sesión) y guardan todo lo importante en la sesión del servidor. Y para cosas no delicadas, como recordar tu tema favorito (claro u oscuro) o tu nombre para saludarte, usan cookies directamente. Al final de esta unidad vas a construir un "recuerdame" que combina ambas tecnicas, igual que lo hacen los sitios profesionales.

Un aviso importante antes de seguir: en esta unidad vas a ver por primera vez el concepto de "cabeceras HTTP" (headers). No te asustes, no necesitas ser experto. Solo entiende esto: cuando el servidor le responde a tu navegador, primero le manda unas instrucciones invisibles (las cabeceras) y después el contenido visible (el HTML). Las cookies y las sesiones se configuran en esas instrucciones invisibles, así que hay reglas estrictas sobre el orden: primero las instrucciones, después el contenido. Si intentas mandar instrucciones después de haber empezado a mandar contenido, PHP te va a gritar con el famoso error "headers already sent". En esta unidad te voy a explicar exactamente por qué pasa y como evitarlo para siempre.

También quiero que sepas hacia donde vamos después: en unidades futuras hablaremos de seguridad avanzada (como guardar contraseñas de forma segura con hash, algo que aquí solo mencionaremos como adelanto), de bases de datos (donde guardaras los usuarios de verdad en lugar de tenerlos escritos en el código) y de buenas prácticas profesionales. Esta unidad te da la base: el mecanismo para recordar usuarios. Lo demás son mejoras sobre esta base.

Así que respira, prepárate un café, y vamos a darle. Al terminar esta unidad vas a poder: crear y leer cookies, darles fecha de caducidad, borrarlas; iniciar sesiones, guardar y leer datos en ellas, destruirlas; explicar la diferencia entre cookies y sesiones como un profesional; y construir un sistema de login completo con página protegida y cierre de sesión. Nada mal para una sola unidad, verdad? Vamos paso a paso y verás que todo tiene sentido.

Explicación detallada

Que son las cookies

Una cookie es un pedacito de texto muy pequeño que un sitio web le pide al navegador que guarde. Eso es todo: texto pequeño, guardado en el navegador del visitante. Nada de programas, nada de virus, nada misterioso. Solo texto, como una notita.

La mejor analogía es la notita en el refrigerador. Imagínate que vas a casa de tu abuela y ella pega una notita en el refri que dice: "A Esteban le gusta el café sin azúcar". La próxima vez que llegas, tu abuela lee la notita y ya te sirve el café como te gusta, sin preguntarte. La cookie funciona igual: el sitio pega una notita en el "refri" de tu navegador que dice algo como "a este visitante le gusta el tema oscuro", y la próxima vez que entras, el sitio lee la notita y te muestra el tema oscuro directamente.

Técnicamente funciona así: cuando tu servidor PHP quiere guardar una cookie, le manda al navegador una instrucción en las cabeceras HTTP que dice algo como "guarda esto: nombre=tema, valor=oscura, y guardalo durante 30 días". El navegador guarda esa notita. Y aquí viene lo bonito: en TODAS las siguientes peticiones que el navegador le haga a ese mismo sitio, el navegador adjunta automáticamente las cookies guardadas. Es como si en cada visita le enseñaras tu gafete al mesero sin que nadie te lo pida. Tu código PHP no tiene que hacer nada especial para recibirlas: llegan solitas en el arreglo superglobal `$_COOKIE`.

Ojo con algo importante: las cookies pertenecen a un dominio. Una cookie que guardó `lucasegaacademy.com` solo se la muestra el navegador a `lucasegaacademy.com`, nunca a otro sitio. Así que no, las cookies de un sitio no las puede leer otro sitio. Esa es una regla de seguridad de los navegadores y es bueno que exista.

Que se puede guardar en una cookie? Texto corto: un nombre, una preferencia ("tema=oscuro", "idioma=es"), un contador de visitas, un identificador. Lo que NO debes guardar nunca en una cookie: contraseñas, números de tarjeta, datos personales delicados, ni nada que te duela si alguien lo lee o lo modifica. Recuerda: la cookie vive en la computadora del visitante, y el visitante (o alguien con acceso a su computadora) puede verla y cambiarla. Más adelante, en las buenas prácticas, profundizaremos en esto.

Cuanto pueden durar? Tú decides. Puedes crear cookies que duran unos minutos, unas horas, días, meses, o incluso un año. También puedes crear cookies de sesión del navegador: esas se borran solas cuando el visitante cierra el navegador. Todo depende del parámetro de expiración que le pases a la función `setcookie()`, que veremos ahora mismo.

La función `setcookie()` y sus parámetros

En PHP, para crear o modificar una cookie se usa la función `setcookie()`. Su forma básica es así:

```
setcookie(nombre, valor, expiración, ruta);
```

Vamos a desmenuzar cada parámetro:

- **nombre:** es el nombre de la cookie, como la etiqueta de la notita. Por ejemplo "usuario", "tema", "visitas". Tiene que ser un texto sin espacios (usa guion bajo si necesitas separar palabras, como "nombre_usuario").
- **valor:** es el contenido de la notita. Por ejemplo "Esteban", "oscuro", "5". Siempre se guarda como texto. Si necesitas guardar un número, PHP lo convierte a texto sin problema, y cuando lo lees puedes convertirlo de vuelta a número.
- **expiración:** es la fecha en la que la cookie caduca, expresada como una marca de tiempo Unix (los segundos transcurridos desde el 1 de enero de 1970). Suena complicado, pero en la práctica siempre lo calculas con la función `time()`, que te da la hora actual en ese formato. Entonces, si quieres que la cookie dure 1 hora, escribes `time() + 3600` (porque una hora tiene 3600 segundos). Si quieres que dure 30 días: `time() + (86400 * 30)`, porque un día tiene 86400 segundos. Memorízate estos números mágicos: 3600 = una hora, 86400 = un día, 604800 = una semana. Con esos tres puedes calcular casi cualquier duración.
- **ruta (path):** indica en que partes del sitio será válida la cookie. Si pones "/", la cookie estará disponible en todo el sitio. Es lo más común y lo recomendado cuando empiezas. Si pusieras `/tienda`, la cookie solo se enviaría cuando el visitante esté en páginas dentro de `/tienda`. Para no complicarte la vida, usa "/" casi siempre.

Un ejemplo completo:

```
setcookie("tema", "oscuro", time() + (86400 * 30), "/");
```

Esta línea le dice al navegador: "guarda una cookie llamada 'tema' con el valor 'oscuro', que caduque dentro de 30 días, valida para todo el sitio".

Ahora, la regla de oro de `setcookie()`: tiene que llamarse ANTES de cualquier salida. Ni un solo echo, ni un solo espacio o salto de línea en el HTML antes de la etiqueta de apertura de PHP, nada. ¿Por qué? Porque `setcookie()` manda su instrucción en las cabeceras HTTP, y las cabeceras tienen que ir antes del contenido. Si ya empezaste a mandar contenido (aunque sea un espacio en blanco accidental antes de `<?php`), PHP ya mandó las cabeceras y es demasiado tarde: te aparecera el error "headers already sent". Este es el error más comun de los principiantes con cookies y sesiones, y más adelante hay una subsección completa dedicada a el.

Otra cosa crucial que confunde a todos al principio: la cookie que creas con `setcookie()` NO está disponible inmediatamente en `$_COOKIE` durante esa misma carga de página. Piensalo: tu le acabas de pedir al navegador que la guarde, pero el navegador la guardara y te la devolvera en la SIGUIENTE petición. Es como mandar una carta: la escribes hoy, pero la respuesta llega mañana. Así que si haces `setcookie("tema", "oscuro", ...)` y en la siguiente línea lees `$_COOKIE["tema"]`, no encontraras nada. La verás a partir de la próxima vez que se cargue la página. Para trabajar con el valor en la misma página, guarda el valor en una variable normal de PHP además de en la cookie. En los ejemplos paso a paso verás este patrón una y otra vez.

Leer cookies con `$_COOKIE`

Leer una cookie es facilísimo: es un arreglo superglobal, así que está disponible en cualquier parte de tu código sin necesidad de declararlo global. Si guardaste una cookie llamada "tema", la lees así:

```
$tema = $_COOKIE["tema"];
```

Pero cuidado: si la cookie no existe (porque es la primera visita, porque caduco, o porque el visitante la borró), PHP te lanzara un aviso de "índice no definido". La forma profesional de leer cookies es verificar primero si existen con `isset()`:

```
if (isset($_COOKIE["tema"])) { $tema = $_COOKIE["tema"]; } else { $tema = "claro"; }
```

Este patrón (`isset` primero, valor por defecto después) lo vas a usar cientos de veces en tu vida como programador. Grabatelo.

También puedes usar el operador de fusión de null (??) que aprendiste en unidades anteriores para hacerlo en una sola línea:

```
$tema = $_COOKIE["tema"] ?? "claro";
```

Ambas formas son correctas. La segunda es más corta y elegante; la primera es más explícita para quien esta aprendiendo. Usa la que te haga sentir más comodo.

Un detalle práctico: los valores de las cookies viajan como texto, así que si guardaste un número (por ejemplo un contador de visitas), al leerlo conviene convertirlo a entero con (int) antes de hacer operaciones matemáticas con el. Verás esto en el ejemplo del contador de visitas.

Borrar cookies

Para borrar una cookie no existe una función "deletcookie()". El truco es volver a llamar a setcookie() con el mismo nombre, pero con una fecha de expiración en el pasado. Así le dices al navegador: "esta cookie ya caduco", y el navegador la elimina.

```
setcookie("tema", "", time() - 3600, "/");
```

Fíjate en los detalles: mismo nombre ("tema"), valor vacío (""), expiración en el pasado (time() - 3600, o sea, hace una hora), y la misma ruta ("/") que usaste al crearla. La ruta tiene que coincidir: si creaste la cookie con path "/" y la intentas borrar con otro path, el navegador la tratara como una cookie distinta y no la borrara. Este es un error típico que hace que la gente diga "no se borra mi cookie" cuando en realidad si se borró... otra que no era.

Igual que al crear, la cookie borrada seguira apareciendo en \$_COOKIE durante la carga actual de la página (porque el navegador la mandó al inicio de la petición, antes de que tu la borraras). Desaparecera a partir de la siguiente carga. Si necesitas que tu página actue como si ya no existiera, ajusta tu lógica con variables, igual que al crear.

Que son las sesiones

Si las cookies son notitas en el refri del visitante, las sesiones son lockers en el edificio del servidor. Vamos a desarrollar la analogía del locker con llave, porque entenderla bien te ahorra muchos dolores de cabeza.

Imagínate un edificio de oficinas con un sistema de visitantes. Cuando llegas por primera vez, en recepción te dan una llavecita con un número grabado, digamos el número 4821. El recepcionista anota en su libreta: "llave 4821 = Esteban, llegó a las 9:00, visita al piso 3". Tu te guardas la llavecita en el bolsillo (es pequena, no pesa nada). Cada vez que entras y sales del

edificio, muestras tu llavecita, el recepcionista busca el número 4821 en su libreta y sabe exactamente quien eres y que andas haciendo. Cuando te vas para siempre, devuelves la llave y el recepcionista tacha tu renglon de la libreta.

En este cuento: la llavecita es el id de sesión (un código largo y aleatorio que PHP genera), la libreta del recepcionista es el servidor (donde PHP guarda los datos de la sesión, normalmente en archivos temporales), y tu bolsillo es... una cookie! Si, leiste bien: PHP usa una cookie pequenita (normalmente llamada PHPSESSID) para guardar el id de sesión en el navegador. Así que las sesiones y las cookies trabajan en equipo: la cookie lleva la llave, el servidor guarda el locker.

La gran ventaja: como los datos viven en el servidor, el visitante no puede verlos ni modificarlos. Puedes guardar ahí el nombre de usuario, su nivel de permiso ("administrador" o "cliente"), los productos de su carrito de compras, y todo estara a salvo de miradas curiosas. La desventaja: cada sesión activa consume un poquito de memoria y disco en tu servidor, y las sesiones normalmente mueren cuando el visitante cierra el navegador (porque la cookie que lleva la llave es una cookie de sesión, sin fecha de expiración larga).

Cuando te conviene cada una? Usa sesiones para todo lo relacionado con "quien es este visitante y que puede hacer": login, permisos, carrito de compras, datos del proceso de compra. Usa cookies para preferencias no delicadas que quieres que duren mucho: tema claro/oscuro, idioma preferido, el nombre para saludar, el "recuerdame" del login (que veremos al final como combinación de ambas).

session_start() y el orden sagrado

Para usar sesiones en PHP, lo primero que tienes que hacer en tu script es llamar a session_start(). Esta función hace dos cosas: si el visitante ya trae una llave (el id de sesión en su cookie PHPSESSID), abre el locker correspondiente y carga los datos en el arreglo \$_SESSION; si no trae llave, le genera una nueva, le manda la llave al navegador y prepara un locker vacío.

La regla sagrada: session_start() tiene que ir ANTES de cualquier salida, igual que setcookie(). Tiene que ser de lo primerito en tu archivo, idealmente la primera línea después de <?php. ¿Por qué? Porque session_start() necesita mandar la cookie con el id de sesión en las cabeceras HTTP (cuando crea una sesión nueva), y además necesita leer las cabeceras que mandó el navegador. Todo lo de cabeceras va antes del contenido. Sin excepciones.

El patrón correcto al inicio de tus archivos:

```
<?php session_start(); // ... aquí va el resto de tu código ?>
```

El patrón INCORRECTO que causa el 90% de los dolores de cabeza:

```
<html> <head>...</head> <?php session_start(); // DEMASIADO TARDE, ya salio HTML ?>
```

Si haces eso, PHP te mostrara el temido error: "Warning: session_start(): Cannot start session when headers already sent". Traducido: "no puedo mandarte la llave porque ya empecé a mandarte la página". Y no es solo el HTML visible: un simple espacio o salto de línea antes de `<?php` también cuenta como salida. Incluso la marca invisible que algunos editores agregan al inicio de los archivos (el BOM de UTF-8) puede provocar este error. Por eso muchos programadores ponen `session_start()` como la primera línea de todos sus archivos PHP, sin nada antes, ni siquiera una línea en blanco.

Truco profesional: si tu proyecto tiene un archivo de configuración o cabecera común que incluyes en todas las páginas (con `include` o `require`), pon `session_start()` ahí una sola vez, al mero principio. Así no tienes que acordarte en cada archivo.

El arreglo `$_SESSION`

Una vez que llamaste a `session_start()`, tienes disponible el arreglo superglobal `$_SESSION`. Guardar datos es tan fácil como asignar valores:

```
$_SESSION["usuario"] = "esteban"; $_SESSION["rol"] = "administrador"; $_SESSION["carrito"] = ["manzanas", "pan", "leche"];
```

Y leerlos es igual de fácil:

```
echo "Hola, " . $_SESSION["usuario"];
```

Puedes guardar textos, números, arreglos e incluso arreglos dentro de arreglos (como un carrito de compras donde cada producto tiene nombre, precio y cantidad). PHP se encarga de guardar todo en el servidor y recuperarlo en la siguiente página donde llames a `session_start()`. Es casi mágico la primera vez que lo ves funcionar: guardas algo en `pagina1.php`, vas a `pagina2.php`, y ahí está, esperandote.

Para verificar si algo existe en la sesión, igual que con las cookies, usa `isset()`:

```
if (isset($_SESSION["usuario"])) { echo "Sesión activa para: " . $_SESSION["usuario"]; } else { echo "No has iniciado sesión."; }
```

Este `if` es la base de las páginas protegidas: "si no hay usuario en la sesión, no te dejo entrar". Lo verás en el sistema de login.

Para borrar un dato específico de la sesión sin destruir todo lo demás, usa `unset()`:

```
unset($_SESSION["carrito"]);
```

Esto vacía el carrito pero mantiene al usuario logueado, por ejemplo. Útil cuando quieres limpiar una parte sin cerrar la sesión completa.

session_destroy() y cerrar sesión correctamente

Cuando el usuario hace clic en "cerrar sesión", quieres borrar todo rastro de su sesión. La función `session_destroy()` elimina los datos de la sesión en el servidor (rompe el locker, siguiendo la analogía). Pero ojo: hay un detalle que muchos tutoriales omiten y que causa sesiones "fantasma" que no se cierran bien.

El procedimiento completo y correcto para cerrar sesión tiene tres pasos:

1. `session_start()` para abrir la sesión actual (si no la abres, no puedes destruirla).
2. Vaciar el arreglo `$_SESSION` con `$_SESSION = array();` (limpias las variables de la sesión en memoria).
3. `session_destroy()` para borrar los datos en el servidor.

Y como paso extra profesional, también borra la cookie que lleva el id de sesión (la llavecita), para que el navegador no siga presentando una llave de un locker que ya no existe:

```
if (isset($_COOKIE[session_name()])) { setcookie(session_name(), "", time() - 3600, "/"); }
```

La función `session_name()` te devuelve el nombre de la cookie de sesión (normalmente "PHPSESSID"), así no tienes que escribirlo a mano. En la sección de ejemplos verás el script de logout completo con estos cuatro pasos, listo para copiar y adaptar.

Un error común es llamar a `session_destroy()` y pensar que `$_SESSION` ya quedó vacío en esa misma página. No es así: `session_destroy()` borra los datos guardados en el servidor, pero el arreglo `$_SESSION` en memoria sigue teniendo los valores durante la ejecución actual. Por eso el paso 2 (`$_SESSION = array();`) es importante: limpia la memoria además del disco.

Cookies vs sesiones: comparativa frente a frente

Ya conoces las dos herramientas. Ahora pongamoslas frente a frente para que sepas exactamente cuando usar cada una. Imagina una tabla con dos columnas, "Cookies" y "Sesiones", y estas filas:

Donde se guardan los datos: las cookies se guardan en el navegador del visitante, en su computadora o teléfono. Las sesiones se guardan en el servidor, en archivos temporales que PHP administra por ti.

Quien puede ver los datos: las cookies las puede ver (y modificar) el visitante con las herramientas de su navegador; no son secretas. Los datos de sesión solo los ve el servidor; el visitante solo tiene la llavecita (el id), nunca el contenido del locker.

Cuanto duran: las cookies duran lo que tu decidas con el parámetro de expiración: minutos, días, meses. Las sesiones, por defecto, duran hasta que el visitante cierra el navegador, aunque el servidor también las limpia después de un tiempo de inactividad (normalmente unos 24 minutos por defecto en PHP, configurable).

Capacidad: las cookies son pequeñas, unos 4 KB por cookie y un límite de unas pocas docenas por dominio; no intentes guardar ahí un carrito de compras grande. Las sesiones pueden guardar mucho más (arreglos completos, carritos con decenas de productos) porque viven en el disco del servidor.

Seguridad: las cookies son inseguras para datos delicados, porque el visitante las controla. Las sesiones son seguras para datos delicados, porque el visitante no puede tocar el contenido.

Rendimiento: las cookies viajan en cada petición HTTP, así que si guardas mucho en cookies, cada página tarda un poquito más en pedirse (el navegador manda las notitas de ida y de vuelta). Las sesiones no viajan: solo viaja la llavecita, así que no afectan el tamaño de las peticiones, pero consumen memoria y disco del servidor por cada visitante activo.

Casos de uso típicos: cookies para recordar el tema visual, el idioma, el nombre para saludar, el "recuérdame" del login, el aviso de "ya acepte las cookies" (irónico pero cierto). Sesiones para saber si el usuario inició sesión, sus permisos, su carrito de compras, los datos de un formulario de varios pasos.

La regla de oro para decidir: pregúntate "me dolería si el visitante leyera o cambiara este dato?" Si la respuesta es sí, va en sesión. Si la respuesta es no y quieres que dure mucho tiempo, va en cookie. Y cuando necesites lo mejor de los dos mundos (duración larga + seguridad), combinalas: guarda en una cookie un código aleatorio (token) y en el servidor asocia ese código con el usuario. Eso es exactamente lo que hace el "recuérdame" que construiremos en los ejemplos.

Sistema de login básico paso a paso

Es hora de unir todo. Un sistema de login es el proyecto clásico donde cookies y sesiones brillan juntas, y es lo que vas a construir en la clase 18. Vamos a diseñar la arquitectura antes de ver el código, para que cada archivo tenga sentido.

Nuestro sistema tendrá cuatro piezas:

1. login.php: el formulario donde el visitante escribe su usuario y contraseña.
2. La verificación: el mismo login.php (o un archivo procesar.php) recibe los datos, verifica que sean correctos y, si lo son, guarda el nombre de usuario en `$_SESSION` y redirige a la zona privada.
3. panel.php: la página protegida. Lo primero que hace es revisar si hay usuario en la sesión; si no lo hay, redirige al login. Si lo hay, muestra el contenido privado.
4. logout.php: destruye la sesión y redirige al login.

Vamos paso por paso con la lógica:

Paso 1: el formulario. Es un formulario HTML normal con `method="post"` y dos campos: usuario y contraseña. Usa POST y no GET, porque los datos del login no deben aparecer en la URL (imagínate tu contraseña visible en la barra del navegador y guardada en el historial... un horror). Este formulario lo veremos completo en los ejemplos.

Paso 2: verificar las credenciales. Cuando llega el POST, comparas lo que escribió el visitante con los datos correctos. En un sistema real, esos datos correctos vivirían en una base de datos; en nuestro sistema básico los tendremos en variables o en un arreglo (es un ejemplo didáctico, y en la unidad de bases de datos lo conectaras con MySQL de verdad). Si coinciden: guardas el usuario en la sesión con `$_SESSION["usuario"] = $usuario;` y rediriges al panel con `header("Location: panel.php");` seguido de `exit;`. El `exit` es importante: después de mandar una redirección, debes detener el script para que no siga ejecutando código que ya no debe ejecutarse.

Paso 3: la página protegida. Al inicio de panel.php, después de `session_start()`, pones el guardián:

```
if (!isset($_SESSION["usuario"])) { header("Location: login.php"); exit; }
```

Traducido: "si no hay usuario en la sesión, no entras; te mandó al login". Así de simple es proteger una página. Cualquier página que quieras proteger lleva ese bloque al inicio. Cuando tengas 20 páginas privadas, pondrás ese bloque (o un include con ese bloque) en las 20.

Paso 4: el logout. Es el script de cierre de sesión que ya describimos: `session_start()`, vaciar `$_SESSION`, `session_destroy()`, borrar la cookie de sesión, y redirigir al login. El usuario verá el formulario de login otra vez, y si intenta entrar directo a panel.php escribiendo la URL, el guardián lo devolverá al login. Sistema completo y funcional.

Nota honesta de seguridad: en este sistema básico compararemos la contraseña en texto plano (la escribimos directamente en el código). Esto es SOLO para aprender el mecanismo del login. En la vida real jamás se guardan ni se comparan contraseñas en texto plano: se

guardan sus "hashes" (huellas digitales irreversibles) con funciones como `password_hash()` y se verifican con `password_verify()`. Lo veremos a fondo en la unidad de seguridad. Por ahora, aprende el flujo; el blindaje viene después. Si alguien te dice "tu login no es seguro", respondeles: "lo se, es la versión didáctica; el hash viene en la siguiente unidad". Y tendrás razón.

Errores comunes (y como evitarlos)

Esta sección te va a ahorrar horas de frustración. Son los errores que comete todo el mundo con cookies y sesiones, descritos con su sintoma y su cura.

Error 1: llamar a `session_start()` o `setcookie()` después de haber mandado HTML. Sintoma: el famoso "Warning: Cannot start session when headers already sent" o "Cannot modify header information - headers already sent". Cura: pon `session_start()` como la primera línea de tu archivo PHP, antes de cualquier HTML, echo, espacio o salto de línea. Revisa también que no haya espacios antes de `<?php`. Si usas includes, asegurate de que los archivos incluidos tampoco manden salida antes.

Error 2: esperar que `$_COOKIE` tenga la cookie recién creada. Sintoma: creas la cookie y en la siguiente línea la lees, pero está vacía. Cura: recuerda que la cookie llega en la siguiente petición. Usa una variable normal de PHP para trabajar con el valor en la página actual, y lee `$_COOKIE` a partir de la siguiente carga.

Error 3: comparar contraseñas en texto plano en un sistema real. Sintoma: ninguno visible... hasta que alguien roba tu código o tu base de datos y tiene las contraseñas de todos tus usuarios. Cura: en proyectos reales usa `password_hash()` para guardar y `password_verify()` para comparar. En esta unidad lo hacemos en texto plano solo como ejemplo didáctico; en la unidad de seguridad lo harás como profesional.

Error 4: no destruir la sesión al salir, o destruirla a medias. Sintoma: el usuario hace clic en "salir" pero si regresa a la página protegida, sigue dentro. Cura: usa el procedimiento completo de logout: `session_start()`, `$_SESSION = array()`, `session_destroy()`, borrar la cookie de sesión con `setcookie(session_name(), "", time() - 3600, "/")`, y redirigir. Los cuatro pasos, siempre.

Error 5: olvidar `session_start()` en alguna página. Sintoma: guardas algo en `$_SESSION` en una página y en la otra no aparece, o te sale un aviso de variable no definida. Cura: cada archivo PHP que lea o escriba la sesión necesita su propio `session_start()` al inicio. No basta con haberlo puesto en otra página: cada petición es independiente (recuerda al mesero sin memoria). Un archivo común incluido en todas las páginas es la solución elegante.

Error 6: no redirigir (o no poner exit) después del login/logout. Sintoma: después de iniciar sesión, si el usuario recarga la página, el navegador le pregunta si quiere reenviar el formulario, y a veces se duplica una acción. Cura: después de procesar el login, redirige con `header("Location: panel.php")` y `exit`. Es el patrón POST-Redirect-GET: procesas el POST y rediriges a una página GET, así recargar es seguro.

Error 7: guardar datos sensibles en cookies. Sintoma: ninguno hasta que es demasiado tarde. Cura: en cookies solo preferencias y datos no delicados. Todo lo que identifique al usuario o le de permisos va en sesión. Y si necesitas un "recuérdame" persistente, usa un token aleatorio en la cookie, nunca el nombre de usuario ni la contraseña.

Error 8: no validar que la cookie existe antes de leerla. Sintoma: avisos de "Undefined index" cuando es la primera visita. Cura: siempre `isset()` o el operador `??` con un valor por defecto antes de usar `$_COOKIE["algo"]`.

Buenas prácticas

Para cerrar la explicación, las costumbres que distinguen a un principiante de alguien que programa en serio:

- Regenera el id de sesión al iniciar sesión. PHP tiene la función `session_regenerate_id(true)`, que le da al usuario una llave nueva justo cuando entra. Esto previene un ataque llamado "fijación de sesión", donde un atacante intenta plantarle a la víctima un id de sesión conocido. Llamala justo después de verificar las credenciales, antes de guardar el usuario en la sesión. Es una sola línea y te protege de un ataque real.
- No guardes datos sensibles en cookies. Ya lo dijimos tres veces en esta unidad; que sea la cuarta y la definitiva. Las cookies son para preferencias. Punto.
- Cierra la sesión siempre con el procedimiento completo. No basta con `session_destroy()` a secas: vacía el arreglo, destruye los datos y borra la cookie de la llave. Hazte un archivo `logout.php` reutilizable y usalo en todos tus proyectos.
- Ponle expiración razonable a tus cookies. Una cookie de "tema" puede durar un año sin problema. Una cookie de "recuérdame" suele durar 30 días. No pongas expiraciones de 10 años "por si acaso": si algún día necesitas invalidarlas, las cookies eternas son un dolor de cabeza.
- Usa nombres de cookie claros y con prefijo de tu proyecto. En lugar de "tema", usa algo como `"ega_tema"`. Así evitas choques con cookies de otros scripts o plugins que corran en el mismo dominio.

- Protege tus páginas privadas con el guardián al inicio, siempre después de `session_start()`. Y si un día agregas niveles de permiso (administrador vs cliente), el guardián crece naturalmente: primero revisa si hay sesión, después revisa el rol guardado en `$_SESSION["rol"]`.
- En el login, no digas cual dato falló. Si el usuario se equivoca, muestra un mensaje generico como "Usuario o contraseña incorrectos", no "el usuario no existe" ni "la contraseña está mal". Decir cuál de los dos falló le regala información a un atacante que esta adivinando cuentas. Detalle pequeño, mentalidad profesional.

Con la teoría clara, es hora de ensuciarnos las manos. En la siguiente sección verás ocho ejemplos completos, explicados línea por línea, que cubren todo lo que aprendiste aquí.

Ejemplos paso a paso

Ejemplo 1: recordar el nombre del visitante con una cookie

Este es el "hola de nuevo" de los sitios web: la primera vez el visitante escribe su nombre, y en las siguientes visitas la página lo saluda sin preguntar. Es el uso más clásico y amable de las cookies.

```

<?php
$nombre = $_COOKIE["visitante"] ?? "";

if ($_SERVER["REQUEST_METHOD"] === "POST" && !empty($_POST["nombre"])) {
    $nombre = trim($_POST["nombre"]);
    setcookie("visitante", $nombre, time() + (86400 * 30), "/");
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Bienvenido</title>
</head>
<body>
    <?php if ($nombre !== ""): ?>
        <h1>Hola de nuevo, <?php echo htmlspecialchars($nombre); ?>!</h1>
        <p>Que bueno verte por aqui otra vez.</p>
    <?php else: ?>
        <h1>Hola, visitante nuevo</h1>
        <form method="post">
            <label>Como te llamas?
                <input type="text" name="nombre" required>
            </label>
            <button type="submit">Guardar mi nombre</button>
        </form>
    <?php endif; ?>
</body>
</html>

```

Explicación línea por línea:

- `$nombre = $_COOKIE["visitante"] ?? "";` : leemos la cookie "visitante". Si no existe (primera visita), \$nombre queda como texto vacío. El operador ?? nos ahorra el isset.
- `if ($_SERVER["REQUEST_METHOD"] === "POST" && !empty($_POST["nombre"])) :` detectamos si el visitante envió el formulario y si el campo nombre no viene vacío. REQUEST_METHOD nos dice como llegó la petición.
- `$nombre = trim($_POST["nombre"]);` : limpiamos espacios al inicio y al final del nombre, y lo guardamos en la variable \$nombre para usarlo DE INMEDIATO en esta misma página (recuerda: la cookie aun no está en \$_COOKIE).

- `setcookie("visitante", $nombre, time() + (86400 * 30), "/");` : creamos la cookie que durara 30 días, valida en todo el sitio. Fíjate que esta línea esta antes de cualquier HTML: cumple la regla de oro.
- El resto es HTML con dos caminos: si `$nombre` no está vacío, saludamos con `htmlspecialchars()` (para evitar que alguien inyecte código HTML malicioso en el nombre: buena costumbre desde el día uno); si está vacío, mostramos el formulario.

Pruebalo: escribe tu nombre, envia, y recarga la página. Ahí esta tu nombre, saludandote. Cierra el navegador, vuelve mañana: seguira ahí, porque la cookie dura 30 días.

Ejemplo 2: contador de visitas con cookie

Un clásico didactico que te ensena a leer, modificar y re-guardar una cookie, además de convertir texto a número.

```
<?php
$visitas = isset($_COOKIE["contador"]) ? (int)$_COOKIE["contador"] : 0;
$visitas++;
setcookie("contador", (string)$visitas, time() + (86400 * 365), "/");
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Contador de visitas</title>
</head>
<body>
    <h1>Has visitado esta pagina <?php echo $visitas; ?> veces</h1>
    <?php if ($visitas === 1): ?>
        <p>Bienvenido por primera vez!</p>
    <?php elseif ($visitas < 5): ?>
        <p>Ya nos estamos conociendo.</p>
    <?php else: ?>
        <p>Eres un visitante frecuente, gracias por volver.</p>
    <?php endif; ?>
</body>
</html>
```

Explicación línea por línea:

- `$visitas = isset($_COOKIE["contador"]) ? (int)$_COOKIE["contador"] : 0;` : si la cookie existe, la convertimos a entero con `(int)`; si no, empezamos en 0. La conversión es importante porque las cookies siempre son texto, y `"5" + 1` en modo texto nos daría problemas.
- `$visitas++;` : sumamos la visita actual. Esta es la primera vez que el número crece.
- `setcookie("contador", (string)$visitas, time() + (86400 * 365), "/");` : guardamos el nuevo total por un año entero. Convertimos a texto con `(string)` por claridad, aunque PHP lo haría solo.
- El HTML muestra mensajes distintos según el número de visitas, usando la variable `$visitas` (no `$_COOKIE["contador"]`, que todavía tiene el valor viejo en esta carga).

Nota para pensar: este contador lo puede manipular el visitante borrando o editando su cookie. Para estadísticas serias se usan otros métodos; para aprender el ciclo leer-modificar-guardar, es perfecto.

Ejemplo 3: preferencia de tema claro/oscuro con cookie

Este ejemplo es oro puro porque es exactamente lo que hacen los sitios reales con su botón de modo oscuro. Además te enseña a cambiar la cookie desde enlaces (GET) en lugar de un formulario.

```

<?php
$tema = $_COOKIE["ega_tema"] ?? "claro";

if (isset($_GET["tema"]) && ($_GET["tema"] === "claro" || $_GET["tema"] ===
"oscuro")) {
    $tema = $_GET["tema"];
    setcookie("ega_tema", $tema, time() + (86400 * 365), "/");
}

$fondo = $tema === "oscuro" ? "#1a1a1a" : "#ffffff";
$texto = $tema === "oscuro" ? "#f0f0f0" : "#1a1a1a";
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mi sitio con tema <?php echo $tema; ?></title>
</head>
<body style="background-color: <?php echo $fondo; ?>; color: <?php echo $texto;
?>;">
    <h1>Tema actual: <?php echo $tema; ?></h1>
    <p><a href="?tema=claro">Tema claro</a> | <a href="?tema=oscurο">Tema
oscurο</a></p>
    <p>Tu eleccion se guarda por un ano en una cookie.</p>
</body>
</html>

```

Explicación línea por línea:

- `$tema = $_COOKIE["ega_tema"] ?? "claro";` : leemos la preferencia guardada, o "claro" por defecto. Fíjate en el prefijo "ega_" en el nombre de la cookie: buena práctica para evitar choques.
- `if (isset($_GET["tema"]) && ($_GET["tema"] === "claro" || $_GET["tema"] === "oscurο"))` : si la URL trae ?tema=oscurο o ?tema=claro, lo aceptamos. La validación es clave: solo aceptamos esos dos valores exactos, nunca cualquier texto que mande el visitante. Esto se llama "lista blanca" y es un habito de seguridad fundamental.
- `$tema = $_GET["tema"];` y `setcookie(...)` : actualizamos la variable para esta página y guardamos la cookie por un ano.
- `$fondo` y `$texto` se eligen con operadores ternarios según el tema, y se inyectan en el estilo del body. Cambia el tema, recarga, cierra el navegador, vuelve en una semana: tu elección sigue ahí.

Este patrón (leer preferencia, permitir cambiarla, guardar en cookie, aplicar de inmediato con variable) lo vas a reutilizar para idioma, tamaño de letra y cualquier otra preferencia.

Ejemplo 4: iniciar sesión y guardar el usuario en \$_SESSION

Ahora entramos a las sesiones. Este ejemplo muestra el corazón del login: verificar credenciales y guardar al usuario en la sesión. Es la mitad del sistema completo.

```
<?php
session_start();

$usuario_valido = "admin";
$clave_valida = "1234";
$error = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $usuario = trim($_POST["usuario"] ?? "");
    $clave = $_POST["clave"] ?? "";

    if ($usuario === $usuario_valido && $clave === $clave_valida) {
        session_regenerate_id(true);
        $_SESSION["usuario"] = $usuario;
        $_SESSION["entrada"] = date("Y-m-d H:i:s");
        header("Location: panel.php");
        exit;
    } else {
        $error = "Usuario o contraseña incorrectos.";
    }
}

?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Iniciar sesión</title>
</head>
<body>
    <h1>Iniciar sesión</h1>
    <?php if ($error !== ""): ?>
        <p style="color: red;"><?php echo $error; ?></p>
    <?php endif; ?>
    <form method="post">
        <label>Usuario:
            <input type="text" name="usuario" required>
        </label><br><br>
```

```

        <label>Contraseña:
            <input type="password" name="clave" required>
        </label><br><br>
        <button type="submit">Entrar</button>
    </form>
    <p>Prueba con usuario "admin" y contraseña "1234".</p>
</body>
</html>

```

Explicación línea por línea:

- session_start(); : la primera línea, antes de todo. Abre (o crea) la sesión.
- \$usuario_valido y \$clave_valida: las credenciales correctas escritas en el código. En un sistema real vendrían de una base de datos; aquí es didactico. Y recuerda el aviso: en producción nunca compares contraseñas en texto plano, usarias password_verify().
- if (\$_SERVER["REQUEST_METHOD"] === "POST") : solo procesamos cuando se envió el formulario.
- \$usuario = trim(\$_POST["usuario"] ?? ""); : leemos y limpiamos el usuario. El ?? "" evita avisos si el campo no llegó.
- if (\$usuario === \$usuario_valido && \$clave === \$clave_valida) : comparamos con === (estricto: mismo valor y mismo tipo).
- session_regenerate_id(true); : le damos una llave nueva al usuario justo al entrar. Buena práctica anti-fijación de sesión, en una sola línea.
- \$_SESSION["usuario"] = \$usuario; : guardamos el nombre en el locker del servidor. Este es EL momento donde el visitante pasa de anonimo a reconocido.
- \$_SESSION["entrada"] = date("Y-m-d H:i:s"); : guardamos también la fecha y hora de entrada, para mostrarla luego en el panel. Las sesiones pueden guardar varios datos, no solo uno.
- header("Location: panel.php"); exit; : redirigimos a la zona privada y detenemos el script. Patrón POST-Redirect-GET.
- else { \$error = "Usuario o contraseña incorrectos."; } : mensaje generico que no revela si falló el usuario o la clave. Mentalidad profesional desde el ejemplo.
- El HTML muestra el error en rojo si existe, y el formulario usa method="post" (los datos no aparecen en la URL) e input type="password" (oculta lo que se escribe).

Si las credenciales son correctas, el visitante termina en panel.php con su sesión activa. Si son incorrectas, ve el error y puede intentarlo de nuevo.

Ejemplo 5: página protegida que redirige si no hay sesión

Este es el guardián. Cualquier página que pongas detras de este bloque solo la veran los usuarios que iniciaron sesión. Es el companero inseparable del ejemplo 4.

```
<?php
session_start();

if (!isset($_SESSION["usuario"])) {
    header("Location: login.php");
    exit;
}

$usuario = $_SESSION["usuario"];
$entrada = $_SESSION["entrada"] ?? "hora no registrada";
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Panel privado</title>
</head>
<body>
    <h1>Bienvenido a tu panel, <?php echo htmlspecialchars($usuario); ?></h1>
    <p>Iniciaste sesion el: <?php echo htmlspecialchars($entrada); ?></p>
    <p>Esta es la zona privada. Solo los usuarios registrados pueden verla.</p>
    <p><a href="logout.php">Cerrar sesion</a></p>
</body>
</html>
```

Explicación línea por línea:

- session_start(); : sin esta línea, \$_SESSION estaria vacío y el guardián dejaria pasar a nadie... o peor, marcaria error. Cada página que use la sesión necesita su propio session_start().
- if (!isset(\$_SESSION["usuario"])) : el guardián. El signo ! significa "si NO existe usuario en sesión". Si alguien intenta entrar directo escribiendo panel.php en la barra del navegador sin haber iniciado sesión, cae aquí.
- header("Location: login.php"); exit; : lo mandamos al login y detenemos todo. Nada del contenido privado se genera ni se envia.
- \$usuario = \$_SESSION["usuario"]; : si paso el guardián, leemos los datos con confianza.

- `$entrada = $_SESSION["entrada"] ?? "hora no registrada";` : leemos la hora de entrada con valor por defecto, por si alguna sesión vieja no la tuviera.
- El HTML saluda con `htmlspecialchars()` (nunca confíes ciegamente en datos que vienen del usuario, ni siquiera los que tú mismo guardaste) y ofrece el enlace para cerrar sesión.

Prueba de fuego: abre `panel.php` en una ventana de incognito (sin sesión) y verás como te devuelve al login. Inicia sesión y entra: contenido privado. Esa es la magia del guardián, y funciona igual para 1 página que para 100.

Ejemplo 6: cerrar sesión correctamente

El logout completo, con los cuatro pasos que evitan sesiones fantasma. Guardalo como `logout.php` y enlazalo desde tus páginas privadas.

```
<?php
session_start();

$_SESSION = array();

if (isset($_COOKIE[session_name()])) {
    setcookie(session_name(), "", time() - 3600, "/");
}

session_destroy();

header("Location: login.php");
exit;
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Saliendo...</title>
</head>
<body>
    <p>Cerrando sesion...</p>
</body>
</html>
```

Explicación línea por línea:

- `session_start();` : abrimos la sesión actual. No puedes destruir lo que no has abierto.

- `$_SESSION = array();` : vaciamos todas las variables de sesión en memoria. Después de esta línea, `$_SESSION["usuario"]` ya no existe.
- `if (isset($_COOKIE[session_name()]))` : verificamos si el navegador trae la cookie con la llave de sesión. `session_name()` nos da su nombre real (normalmente "PHPSESSID") sin tener que escribirlo a mano.
- `setcookie(session_name(), "", time() - 3600, "/");` : borramos esa cookie poniendole fecha en el pasado. El navegador tira la llavecita a la basura.
- `session_destroy();` : borramos los datos de la sesión en el servidor. El locker se vacía y se marca para eliminación.
- `header("Location: login.php"); exit;` : mandamos al visitante de vuelta al login. El HTML de abajo casi nunca se ve (la redirección es inmediata), pero está ahí por si algún navegador no sigue la redirección.

Después de pasar por `logout.php`, si el visitante intenta volver a `panel.php`, el guardián lo rechazará: la sesión ya no existe. Cierre limpio y completo.

Ejemplo 7: carrito de compras con sesiones

Aquí las sesiones muestran su verdadero poder: guardar estructuras de datos completas (un arreglo de productos) que persisten mientras el visitante navega por la tienda. Es el mismo principio que usan las tiendas reales.

```
<?php
session_start();

if (!isset($_SESSION["carrito"])) {
    $_SESSION["carrito"] = [];
}

$catalogo = [
    "manzana" => 10,
    "pan" => 25,
    "leche" => 30,
    "queso" => 60,
];

if (isset($_GET["agregar"]) && isset($catalogo[$_GET["agregar"]])) {
    $producto = $_GET["agregar"];
    $_SESSION["carrito"][$producto] = ($_SESSION["carrito"][$producto] ?? 0) + 1;
    header("Location: carrito.php");
    exit;
```

```

}

if (isset($_GET["vaciar"])) {
    unset($_SESSION["carrito"]);
    $_SESSION["carrito"] = [];
    header("Location: carrito.php");
    exit;
}

$total = 0;
foreach ($_SESSION["carrito"] as $producto => $cantidad) {
    $total += $catalogo[$producto] * $cantidad;
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mi carrito</title>
</head>
<body>
    <h1>Tiendita</h1>
    <h2>Catalogo</h2>
    <ul>
        <?php foreach ($catalogo as $producto => $precio): ?>
            <li><?php echo $producto; ?> - $<?php echo $precio; ?>
                <a href="?agregar=<?php echo $producto; ?>">[Agregar]</a>
            </li>
        <?php endforeach; ?>
    </ul>
    <h2>Tu carrito</h2>
    <?php if (empty($_SESSION["carrito"])): ?>
        <p>Tu carrito esta vacio.</p>
    <?php else: ?>
        <ul>
            <?php foreach ($_SESSION["carrito"] as $producto => $cantidad): ?>
                <li><?php echo $producto; ?> x <?php echo $cantidad; ?>
                    = $<?php echo $catalogo[$producto] * $cantidad; ?></li>
            <?php endforeach; ?>
        </ul>
        <p><strong>Total: $<?php echo $total; ?></strong></p>
        <p><a href="?vaciar=1">Vaciar carrito</a></p>
    <?php endif; ?>
</body>
</html>

```

Explicación línea por línea:

- `session_start();` y el `if` que inicializa `$_SESSION["carrito"]` como arreglo vacío si no existe. Así el resto del código puede asumir que el carrito siempre es un arreglo.
- `$catálogo`: arreglo asociativo `producto => precio`. En una tienda real saldría de una base de datos; aquí está fijo para concentrarnos en la sesión.
- `if (isset($_GET["agregar"]) && isset($catálogo[$_GET["agregar"]]))` : si la URL trae `?agregar=manzana`, verificamos que "manzana" exista en el catálogo (lista blanca otra vez: nunca aceptes un producto que no vendes).
- `$_SESSION["carrito"][$producto] = ($_SESSION["carrito"][$producto] ?? 0) + 1;` : la línea estrella. Si el producto ya estaba en el carrito, suma 1 a su cantidad; si no estaba, empieza en 1. El carrito es un arreglo dentro de la sesión: `["manzana" => 2, "pan" => 1]`.
- `header("Location: carrito.php"); exit;` : redirigimos para que recargue la página no agregue el producto otra vez (POST-Redirect-GET aplicado a GET).
- El bloque "vaciar": `unset($_SESSION["carrito"])` borra el carrito y lo reiniciamos como arreglo vacío.
- El `foreach` calcula el total multiplicando precio por cantidad de cada producto.
- El HTML lista el catálogo con enlaces [Agregar] y muestra el carrito con cantidades, subtotales y total.

Fíjate en algo hermoso: el visitante puede navegar a otras páginas de tu sitio y al volver al carrito, todo sigue ahí. Eso es la sesión trabajando: el locker del servidor guarda sus cosas mientras el pasea con su llavecita. Y cuando cierra el navegador, el carrito se olvida, que es el comportamiento normal de una sesión.

Ejemplo 8: "recuérdame" combinando cookie y sesión

El gran final: lo mejor de los dos mundos. Una cookie de larga duración guarda un código secreto (token); el servidor asocia ese código con el usuario. Así el login sobrevive aunque se cierre el navegador, sin guardar nada delicado en la cookie.

```
<?php
session_start();

// Base de tokens: en un sistema real viviría en la base de datos
$tokens_validos = [
    "abc123token" => "admin",
];
```

```

$usuario = $_SESSION["usuario"] ?? null;

// Si no hay sesion pero hay cookie de "recuerdame", intentamos revivir la sesion
if ($usuario === null && isset($_COOKIE["ega_recuerdame"])) {
    $token = $_COOKIE["ega_recuerdame"];
    if (isset($tokens_validos[$token])) {
        session_regenerate_id(true);
        $usuario = $tokens_validos[$token];
        $_SESSION["usuario"] = $usuario;
    } else {
        // Token desconocido: lo borramos por seguridad
        setcookie("ega_recuerdame", "", time() - 3600, "/");
    }
}

// Procesar el formulario de login
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $u = trim($_POST["usuario"] ?? "");
    $c = $_POST["clave"] ?? "";
    if ($u === "admin" && $c === "1234") {
        session_regenerate_id(true);
        $_SESSION["usuario"] = $u;
        $usuario = $u;
        if (isset($_POST["recuerdame"])) {
            $nuevo_token = bin2hex(random_bytes(16));
            // En un sistema real: guardar $nuevo_token asociado a $u en la base de
datos
            setcookie("ega_recuerdame", $nuevo_token, time() + (86400 * 30), "/");
        }
    }
}

?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Login con recuerdame</title>
</head>
<body>
    <?php if ($usuario !== null): ?>
        <h1>Hola, <?php echo htmlspecialchars($usuario); ?> (sesion activa)</h1>
        <p><a href="logout.php">Cerrar sesion</a></p>
    <?php else: ?>
        <h1>Iniciar sesion</h1>
        <form method="post">

```

```

                                <label>Usuario:  <input   type="text"   name="usuario"
required></label><br><br>
                                <label>Contraseña:  <input   type="password"   name="clave"
required></label><br><br>
                                <label><input type="checkbox" name="recuerdame" value="1"> Recuerdame en
este equipo</label><br><br>
                                <button type="submit">Entrar</button>
                                </form>
                                <?php endif; ?>
</body>
</html>

```

Explicación línea por línea:

- `$tokens_validos`: simula la tabla de la base de datos donde guardarías "token => usuario". En producción, cada token se genera aleatorio y se guarda junto al usuario; aquí hay uno fijo para que puedas probar.
- `$usuario = $_SESSION["usuario"] ?? null;` : primero intentamos la vía normal: hay sesión activa?
- El bloque del "recuerdame": si no hay sesión pero el navegador trae la cookie `ega_recuerdame`, buscamos ese token en nuestra base. Si existe, revivimos la sesión: regeneramos el id y guardamos el usuario en `$_SESSION`. El visitante entra sin escribir nada, días después de su última visita.
- Si el token NO existe en la base (alguien lo invento o fue revocado), borramos la cookie de inmediato. Nunca confíes en un token desconocido.
- En el procesamiento del login, si las credenciales son correctas y el visitante marco la casilla "recuerdame", generamos un token con `bin2hex(random_bytes(16))`: 32 caracteres hexadecimales aleatorios, prácticamente imposibles de adivinar. En un sistema real lo guardaríamos en la base de datos asociado al usuario; aquí lo mostramos como concepto.
- `setcookie("ega_recuerdame", $nuevo_token, time() + (86400 * 30), "/");` : la cookie vive 30 días. Cuando caduque, el "recuerdame" dejara de funcionar y el usuario tendra que loguearse de nuevo: comportamiento correcto y esperado.
- El formulario agrega la casilla de verificación "Recuerdame en este equipo". Si no la marcas, el login funciona igual pero solo dura la sesión del navegador.

Observa la elegancia del diseño: la cookie jamás contiene el nombre de usuario ni la contraseña, solo un código aleatorio sin significado. Si alguien roba la cookie, solo obtiene un código que tú puedes revocar en el servidor en cualquier momento (borrando el renglón de tu base de datos). Así combinan cookies y sesiones los sitios profesionales, y ahora tú sabes como.

Práctica

Ejercicio 1

Crea una página llamada `saludo.php` que pregunte el nombre del visitante una sola vez y lo recuerde durante 7 días usando una cookie. Si la cookie ya existe, la página debe saludar directamente sin mostrar el formulario. Además, agrega un enlace "Olvidar mi nombre" que borre la cookie y vuelva a mostrar el formulario.

Ejercicio 2

Crea un contador de visitas mejorado (`visitas.php`) que use una cookie con duración de 30 días y muestre mensajes personalizados: primera visita ("Bienvenido por primera vez"), de 2 a 4 visitas ("Que bueno verte de nuevo"), 5 o más ("Eres un visitante VIP"). El contador debe empezar en 1 en la primera visita, no en 0.

Ejercicio 3

Construye un mini sistema de login de dos páginas: `acceso.php` con el formulario (usuario "ega" y contraseña "academia2026") y `secreta.php` protegida con el guardián de sesión. Si las credenciales son correctas, guarda el usuario en la sesión, regenera el id de sesión y redirige a `secreta.php`. Agrega también `salir.php` que cierre la sesión con el procedimiento completo y redirija a `acceso.php`.

Ejercicio 4

Crea una página `idioma.php` que permita elegir entre español e inglés con dos enlaces, guarde la elección en una cookie de 1 año llamada "ega_idioma" y muestre el saludo correspondiente ("Hola, bienvenido a la academia" o "Hello, welcome to the academy"). La elección debe aplicarse de inmediato, sin necesidad de recargar la página manualmente, y debe recordar la elección aunque se cierre el navegador.

Ejercicio 5

Crea una lista de tareas pendientes (tareas.php) que use sesiones: un formulario para agregar tareas, la lista de tareas guardada en `$_SESSION["tareas"]` como un arreglo, un enlace para eliminar cada tarea individual y un enlace para borrar todas las tareas. Las tareas deben persistir mientras el navegador este abierto, aunque el usuario navegue a otras páginas y regrese.

Solución 1


```

<?php
if (isset($_GET["olvidar"])) {
    setcookie("nombre_visitante", "", time() - 3600, "/");
    header("Location: saludo.php");
    exit;
}

$nombre = $_COOKIE["nombre_visitante"] ?? "";

if ($_SERVER["REQUEST_METHOD"] === "POST" && !empty(trim($_POST["nombre"] ?? "")))
{
    $nombre = trim($_POST["nombre"]);
    setcookie("nombre_visitante", $nombre, time() + (86400 * 7), "/");
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Saludo personalizado</title>
</head>
<body>
    <?php if ($nombre !== ""): ?>
        <h1>Hola, <?php echo htmlspecialchars($nombre); ?>!</h1>
        <p>Te recuerdo durante 7 dias gracias a una cookie.</p>
        <p><a href="?olvidar=1">Olvidar mi nombre</a></p>
    <?php else: ?>
        <h1>Como te llamas?</h1>
        <form method="post">
            <input type="text" name="nombre" required>
            <button type="submit">Guardar</button>
        </form>
    <?php endif; ?>
</body>
</html>

```

Explicación paso a paso:

- Paso 1: detectamos ?olvidar=1 en la URL. Si viene, borramos la cookie con fecha en el pasado y redirigimos a saludo.php limpio (sin el parámetro en la URL). La redirección es importante para que el enlace no quede "armado" al recargar.
- Paso 2: leemos la cookie con ?? para evitar avisos en la primera visita.

- Paso 3: si se envió el formulario con un nombre no vacío (usamos trim para rechazar nombres de puros espacios), lo guardamos en la variable y creamos la cookie por 7 días: $86400 * 7$.
- Paso 4: el HTML decide que mostrar. Con nombre: saludo + enlace para olvidar. Sin nombre: formulario. Usamos htmlspecialchars() al imprimir el nombre por seguridad.
- Todo el código PHP que toca cookies esta antes del HTML: la regla de oro se cumple.

Solución 2

```
<?php
$visitas = isset($_COOKIE["mis_visitas"]) ? (int)$_COOKIE["mis_visitas"] : 0;
$visitas++;
setcookie("mis_visitas", (string)$visitas, time() + (86400 * 30), "/");

if ($visitas === 1) {
    $mensaje = "Bienvenido por primera vez";
} elseif ($visitas <= 4) {
    $mensaje = "Que bueno verte de nuevo";
} else {
    $mensaje = "Eres un visitante VIP";
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Contador mejorado</title>
</head>
<body>
    <h1><?php echo $mensaje; ?></h1>
    <p>Numero de visita: <?php echo $visitas; ?></p>
</body>
</html>
```

Explicación paso a paso:

- Paso 1: leemos la cookie y la convertimos a entero; si no existe, partimos de 0.
- Paso 2: incrementamos ANTES de guardar, así la primera visita muestra 1 y no 0. Este orden (leer, incrementar, guardar) es el patrón correcto del contador.
- Paso 3: guardamos el nuevo valor por 30 días.

- Paso 4: elegimos el mensaje con if/elseif/else según los rangos pedidos: exactamente 1, de 2 a 4 (≤ 4), 5 o más (el else final).
- Paso 5: mostramos el mensaje y el número usando la variable \$visitas, que ya tiene el valor actualizado para esta carga.

Solución 3

Archivo acceso.php:

```

<?php
session_start();

if (isset($_SESSION["usuario"])) {
    header("Location: secreta.php");
    exit;
}

$error = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $usuario = trim($_POST["usuario"] ?? "");
    $clave = $_POST["clave"] ?? "";
    if ($usuario === "ega" && $clave === "academia2026") {
        session_regenerate_id(true);
        $_SESSION["usuario"] = $usuario;
        header("Location: secreta.php");
        exit;
    } else {
        $error = "Usuario o contraseña incorrectos.";
    }
}

?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Acceso</title>
</head>
<body>
    <h1>Acceso restringido</h1>
    <?php if ($error !== ""): ?>
        <p style="color: red;"><?php echo $error; ?></p>
    <?php endif; ?>
    <form method="post">
        <label>Usuario: <input type="text" name="usuario" required></label><br><br>
        <label>Contraseña: <input type="password" name="clave"
required></label><br><br>
        <button type="submit">Entrar</button>
    </form>
</body>
</html>

```

Archivo secreta.php:

```

<?php
session_start();

if (!isset($_SESSION["usuario"])) {
    header("Location: acceso.php");
    exit;
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Zona secreta</title>
</head>
<body>
    <h1>Zona secreta</h1>
    <p>Hola, <?php echo htmlspecialchars($_SESSION["usuario"]); ?>. Solo los
registrados ven esto.</p>
    <p><a href="salir.php">Cerrar sesion</a></p>
</body>
</html>

```

Archivo salir.php:

```

<?php
session_start();
$_SESSION = array();
if (isset($_COOKIE[session_name()])) {
    setcookie(session_name(), "", time() - 3600, "/");
}
session_destroy();
header("Location: acceso.php");
exit;
?>

```

Explicación paso a paso:

- Paso 1 (acceso.php): session_start() al inicio. Si ya hay sesión activa, redirigimos directo a secreta.php para no mostrar el login a quien ya entro.
- Paso 2: al recibir el POST, comparamos con las credenciales pedidas ("ega" / "academia2026"). Si coinciden: regeneramos el id de sesión, guardamos el usuario en \$_SESSION y redirigimos a secreta.php con exit.

- Paso 3: si no coinciden, mostramos el mensaje generico de error sin decir que dato falló.
- Paso 4 (secreta.php): el guardián. Sin session_start() y sin el if que redirige, la página no estaría protegida. Con ellos, solo entra quien inicio sesión.
- Paso 5 (salir.php): el procedimiento completo de logout en orden: abrir sesión, vaciar el arreglo, borrar la cookie de la llave, destruir los datos en el servidor y redirigir a acceso.php.
- Recuerda el aviso de seguridad: las credenciales en texto plano son solo para este ejercicio didactico; en un proyecto real usarias password_hash() y password_verify().

Solución 4

```
<?php
$id idioma = $_COOKIE["ega_idioma"] ?? "es";

if (isset($_GET["idioma"]) && ($_GET["idioma"] === "es" || $_GET["idioma"] ===
"en")) {
    $idioma = $_GET["idioma"];
    setcookie("ega_idioma", $idioma, time() + (86400 * 365), "/");
}

$saludo = $idioma === "en" ? "Hello, welcome to the academy" : "Hola, bienvenido a
la academia";
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Selector de idioma</title>
</head>
<body>
    <h1><?php echo $saludo; ?></h1>
    <p><a href="?idioma=es">Espanol</a> | <a href="?idioma=en">English</a></p>
    <p>Tu idioma preferido se recuerda por un ano.</p>
</body>
</html>
```

Explicación paso a paso:

- Paso 1: leemos la cookie "ega_idioma"; por defecto, espanol ("es").
- Paso 2: si la URL trae ?idioma=es o ?idioma=en, validamos con lista blanca (solo esos dos valores exactos) y actualizamos tanto la variable como la cookie por un ano.

- Paso 3: elegimos el saludo con un ternario según el idioma actual.
- Paso 4: como usamos la variable \$idioma (actualizada en esta misma carga) y no \$_COOKIE directamente, el cambio se aplica de inmediato sin recargar manualmente. La cookie, a su vez, garantiza que la próxima visita (aunque cierres el navegador) conserve tu elección.

Solución 5

```
<?php
session_start();

if (!isset($_SESSION["tareas"])) {
    $_SESSION["tareas"] = [];
}

if ($_SERVER["REQUEST_METHOD"] === "POST" && !empty(trim($_POST["tarea"] ?? ""))) {
    $_SESSION["tareas"][] = trim($_POST["tarea"]);
    header("Location: tareas.php");
    exit;
}

if (isset($_GET["borrar"]) && isset($_SESSION["tareas"][$(int)$_GET["borrar"]])) {
    unset($_SESSION["tareas"][$(int)$_GET["borrar"]]);
    $_SESSION["tareas"] = array_values($_SESSION["tareas"]);
    header("Location: tareas.php");
    exit;
}

if (isset($_GET["borrar_todas"])) {
    $_SESSION["tareas"] = [];
    header("Location: tareas.php");
    exit;
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mis tareas</title>
</head>
<body>
    <h1>Mis tareas pendientes</h1>
    <form method="post">
        <input type="text" name="tarea" placeholder="Nueva tarea" required>
```

```

        <button type="submit">Agregar</button>
    </form>
    <?php if (empty($_SESSION["tareass"])): ?>
        <p>No tienes tareas pendientes. Disfruta tu día!</p>
    <?php else: ?>
        <ul>
            <?php foreach ($_SESSION["tareass"] as $i => $tarea): ?>
                <li><?php echo htmlspecialchars($tarea); ?>
                    <a href="?borrar=<?php echo $i; ?>">[eliminar]</a>
                </li>
            <?php endforeach; ?>
        </ul>
        <p><a href="?borrar_todas=1">Borrar todas las tareas</a></p>
    <?php endif; ?>
</body>
</html>

```

Explicación paso a paso:

- Paso 1: session_start() e inicialización del arreglo \$_SESSION["tareass"] si no existe. El operador [] agrega al final del arreglo.
- Paso 2: al enviar el formulario, validamos que la tarea no este vacía (trim rechaza puros espacios), la agregamos a la sesión y redirigimos para evitar duplicados al recargar (POST-Redirect-GET).
- Paso 3: para eliminar una tarea, la URL trae su índice (?borrar=2). Convertimos a entero con (int) y verificamos que ese índice exista antes de hacer unset, para no borrar nada indebido. Después reindexamos con array_values() para que los índices queden consecutivos (0, 1, 2...) y los enlaces de eliminar sigan apuntando a la tarea correcta.
- Paso 4: "borrar todas" simplemente reinicia el arreglo a vacío.
- Paso 5: el HTML muestra el formulario siempre, y debajo la lista con un enlace [eliminar] por tarea (usando su índice \$i) o el mensaje de lista vacía. Todo con htmlspecialchars() al imprimir.
- Las tareas sobreviven a la navegación porque viven en la sesión del servidor; al cerrar el navegador, la sesión muere y la lista se olvida, que es el comportamiento esperado de este ejercicio.

Resumen de la unidad

- HTTP no tiene memoria: cada petición es como un cliente nuevo para el servidor, por eso PHP necesita cookies y sesiones para reconocer a los visitantes.

- Las cookies son pequeños textos que se guardan en el navegador del visitante; sirven para preferencias y datos no delicados que deben durar mucho tiempo.
- Se crean con `setcookie(nombre, valor, expiración, ruta)`, donde la expiración se calcula con `time()` más segundos (3600 = 1 hora, 86400 = 1 día) y la ruta "/" las hace validas en todo el sitio.
- Se leen con `$_COOKIE`, siempre verificando con `isset()` o el operador `??` para evitar avisos cuando no existen.
- Se borran llamando a `setcookie()` con el mismo nombre y una fecha de expiración en el pasado; la ruta debe coincidir con la usada al crearlas.
- Una cookie recién creada o borrada solo se refleja en `$_COOKIE` a partir de la siguiente carga de página; en la página actual trabaja con una variable normal de PHP.
- Las sesiones guardan los datos en el servidor (como un locker con llave) y solo viaja al navegador el id de sesión en una pequeña cookie; son el lugar correcto para datos delicados como el login.
- `session_start()` debe ser lo primero en cada archivo que use sesiones, antes de cualquier salida HTML, o aparecerán los errores de "headers already sent".
- Los datos de sesión viven en el arreglo superglobal `$_SESSION`, donde puedes guardar textos, números y arreglos completos como un carrito de compras.
- Cerrar sesión correctamente requiere cuatro pasos: `session_start()`, vaciar `$_SESSION`, borrar la cookie del id de sesión y llamar a `session_destroy()`, terminando con una redirección.
- Una página protegida usa el guardián: si no existe `$_SESSION["usuario"]`, se redirige al login con `header()` y `exit`.
- Un sistema de login básico tiene cuatro piezas: formulario (POST), verificación de credenciales con `session_regenerate_id()`, página protegida con el guardián y script de logout completo.
- Nunca guardes datos sensibles en cookies; para el "recuérdame" usa un token aleatorio en la cookie asociado al usuario en el servidor, nunca el usuario ni la contraseña.
- Las credenciales en texto plano son solo para aprender el mecanismo; en proyectos reales las contraseñas se guardan con `password_hash()` y se verifican con `password_verify()`, tema de la unidad de seguridad.

UNIDAD 7: BASES DE DATOS CON PDO

Bienvenido a una de las unidades más importantes de todo este curso. Hasta ahora, tus programas de PHP han trabajado con datos que viven dentro del propio código: variables, arreglos y archivos. Pero hay un problema serio con todo eso: en cuanto termina de ejecutarse el script, la memoria del servidor se limpia y todo lo que tu programa sabía se olvida. Es como si tu aplicación tuviera amnesia total cada vez que alguien presiona F5.

En esta unidad vas a aprender a darle memoria permanente a tus programas usando una base de datos real: MySQL (y su hermano gemelo MariaDB), conectandote desde PHP con la extensión moderna y segura llamada PDO. Al terminar serás capaz de conectar tu código a una base de datos, leer registros, insertar datos de formularios, actualizar y borrar información, paginar resultados y -lo más importante de todo- protegerte contra el ataque más famoso del desarrollo web: la inyección SQL.

Por qué una aplicación sin base de datos olvida todo al recargar

Imagina que tienes un cuaderno de verdad, de papel, donde anotas los precios de los productos de tu tiendita. Cada noche guardas el cuaderno en un cajón. A la mañana siguiente lo sacas y ahí sigue todo: nada se borró. Eso es una base de datos: un cuaderno permanente y confiable.

Ahora imagina que en lugar del cuaderno intentas recordar todos los precios en tu cabeza, sin anotar nada. Te vas a dormir y al despertar... ya olvidaste la mitad. Eso es lo que le pasa a PHP: las variables solo existen mientras el script se está ejecutando. Cuando el script termina, el servidor "se duerme" y olvida todo. A esto se le llama memoria de corto plazo o memoria volátil.

Tu navegador también tiene una memoria de corto plazo: las variables de JavaScript se borran al recargar la página. Las cookies y el almacenamiento local ayudan un poco, pero viven en la computadora del visitante, no en tu servidor, y cualquier persona puede verlos o modificarlos.

Una base de datos MySQL es el cuaderno del servidor: un lugar permanente, ordenado y rápido donde tu aplicación puede guardar información y recuperarla meses después. La tienda en línea guarda sus productos ahí. El sistema de la escuela guarda las calificaciones ahí. Tu red social favorita guarda millones de publicaciones ahí. Si tu aplicación necesita recordar algo entre una visita y otra, necesita una base de datos. Punto.

Qué es PDO y por qué usarlo en vez de las funciones viejas `mysql_*`

En el PHP antiguo existían unas funciones que empezaban con `mysql_` (por ejemplo `mysql_connect` y `mysql_query`). Fueron la forma estándar de conectarse a MySQL durante muchos años. El problema es que tenían defectos graves de diseño: no obligaban a usar sentencias preparadas (así que era fácil dejar huecos de seguridad), solo servían para MySQL y nada más, y su manejo de errores era torpe. PHP las marco como obsoletas en la versión 5.5 y las eliminó por completo en PHP 7.0. Si hoy ves un tutorial que usa `mysql_connect`, ese tutorial es un fósil: ciérralo y sigue con este libro.

La forma moderna de conectarse se llama PDO, que significa PHP Data Objects (Objetos de Datos de PHP). Piensa en PDO como un enchufe universal de esos que usan los viajeros: el mismo enchufe te sirve en México, en España o en Japón porque el adaptador se encarga de traducir. PDO es igual: aprendes una sola forma de trabajar y con ella puedes hablar con MySQL, PostgreSQL, SQLite, SQL Server y otras bases de datos, cambiando apenas un par de líneas. Si mañana tu proyecto cambia de MySQL a otra base, tu código casi no se entera.

Además, PDO te obliga (de buena manera) a usar sentencias preparadas, que son la defensa número uno contra la inyección SQL, el ataque que veremos a fondo en esta unidad. También maneja los errores con excepciones, que son mucho más claras que los viejos mensajes crípticos.

En resumen: las funciones `mysql_*` son historia, `mysqli` es una opción válida pero atada solo a MySQL, y PDO es la herramienta profesional, portable y segura que todo desarrollador PHP debe dominar. Por eso esta unidad entera gira alrededor de él.

Que vas a ver en esta unidad

Primero entenderás que es MySQL y cómo se relaciona con phpMyAdmin dentro de XAMPP, el paquete que instalaste en unidades anteriores. Después crearemos juntos una base de datos de ejemplo -una tiendita con productos y clientes- que usarás en todos los ejercicios. Luego aprenderás el DSN (la "dirección" de la base de datos), a abrir la conexión con `new PDO` dentro de un `try/catch`, y a configurar el modo de errores. Después dominarás las consultas: `SELECT` para leer, `INSERT/UPDATE/DELETE` para escribir, con `fetch` y `fetchAll` para recorrer resultados. A la mitad de la unidad llegará el tema estrella: la inyección SQL explicada como la historia de un ladrón, y las sentencias preparadas como la cerradura que lo deja fuera. Cerraremos con errores comunes, buenas prácticas y mucha práctica con código real.

Toma nota de algo: esta unidad tiene más código que las anteriores, y eso es intencional. Las bases de datos se aprenden escribiendo consultas, equivocandote con ellas y corrigiéndolas. No te limites a leer los ejemplos: copialos, ejecútalos en tu XAMPP y rompelos a propósito para ver qué pasa. Esa es la forma más rápida de aprender.

Un aviso honesto antes de empezar: vas a ver mensajes de error. Muchos. La conexión fallara porque escribiste mal la contraseña, una consulta fallara porque olvidaste una coma, un fetch devolvera false porque no había resultados. Todo eso es normal y es parte del oficio. Cada error que entiendas hoy es un error que no te asustara mañana en un proyecto real.

Explicación detallada

Que es MySQL/MariaDB y que es phpMyAdmin en XAMPP

MySQL es un sistema manejador de bases de datos relacionales. Traducido a lenguaje humano: es un programa que se encarga de guardar tus datos en tablas ordenadas (filas y columnas, como una hoja de calculo) y de responder preguntas sobre esos datos usando un lenguaje llamado SQL.

MariaDB es un hermano gemelo de MySQL: nació del mismo código, funciona casi identico y es el que hoy viene incluido en XAMPP. Para efectos de este curso, MySQL y MariaDB son lo mismo. Cuando alguien diga MySQL, tú puedes pensar MariaDB sin problema.

XAMPP es el paquete que instalaste antes: trae Apache (el servidor web), PHP, y MariaDB/MySQL, todo junto y ya configurado para trabajar en equipo. Es como comprar un kit de herramientas: en vez de instalar cada programa por separado y pelearte con la configuración, XAMPP te da todo listo.

phpMyAdmin es una aplicación web que viene dentro de XAMPP y sirve para administrar tus bases de datos con clicks en lugar de comandos. Para abrirlo: enciende Apache y MySQL desde el panel de control de XAMPP, abre tu navegador y visita <http://localhost/phpmyadmin>. Ahí verás la lista de bases de datos, podrás crear tablas, insertar filas de prueba y ejecutar consultas SQL a mano. Es tu laboratorio: antes de escribir una consulta en PHP, pruebala en phpMyAdmin para asegurarte de que funciona.

Analogía: si la base de datos es una bodega enorme llena de cajas ordenadas, MySQL es el encargado de la bodega que sabe exactamente donde esta cada caja, SQL es el idioma en el que le hablas ("traeme las cajas rojas de la sección 3"), y phpMyAdmin es la ventanilla con un empleado amable que te deja pedir las cosas con formularios en vez de gritarle ordenes al encargado.

Datos de conexión típicos en XAMPP: el servidor está en "localhost" (tu propia computadora), el usuario es "root" y la contraseña está vacía (cadena vacía ""). El puerto por defecto es 3306. En un servidor real de producción estos datos serán distintos y la contraseña NO estara vacía, pero para practicar en tu máquina esos son los valores.

Crear la base de datos y las tablas de ejemplo

Vamos a crear la base de datos de la tiendita que usaremos en toda la unidad. Tendrá dos tablas: productos y clientes. Puedes crearla de dos formas: con clicks en phpMyAdmin o con código SQL. Te muestro el código SQL porque así aprendes el lenguaje de una vez; copialo en la pestaña "SQL" de phpMyAdmin y ejecútalo.

Primero creamos la base de datos y la seleccionamos:

```
CREATE DATABASE IF NOT EXISTS tienda CHARACTER SET utf8mb4; USE tienda;
```

La primera línea crea la base de datos llamada "tienda" (si ya existe, no hace nada gracias al IF NOT EXISTS). CHARACTER SET utf8mb4 significa "guarda los caracteres en formato que acepta acentos, ñes y hasta emojis". La segunda línea dice "de aquí en adelante trabajo dentro de la base tienda".

Ahora la tabla de productos:

```
CREATE TABLE productos ( id INT AUTO_INCREMENT PRIMARY KEY, nombre VARCHAR(100) NOT NULL, precio DECIMAL(10,2) NOT NULL, stock INT NOT NULL DEFAULT 0, categoría VARCHAR(50) );
```

Lee esto como una receta: la tabla se llama productos y tiene cinco columnas. id es un número entero que crece solo (AUTO_INCREMENT) y es la llave primaria (PRIMARY KEY), o sea, el identificador único de cada fila, como la CURP de cada producto. nombre es texto de hasta 100 caracteres y no puede quedar vacío (NOT NULL). precio es un número decimal con 2 decimales, ideal para dinero. stock es entero y vale 0 si no dices otra cosa. categoría es texto opcional.

La tabla de clientes:

```
CREATE TABLE clientes ( id INT AUTO_INCREMENT PRIMARY KEY, nombre VARCHAR(100) NOT NULL, email VARCHAR(150) NOT NULL UNIQUE, teléfono VARCHAR(20) );
```

Aquí email lleva UNIQUE, que significa "no puede haber dos clientes con el mismo correo". Es una regla que la base de datos vigila por ti.

Y unos datos de ejemplo para practicar:

```
INSERT INTO productos (nombre, precio, stock, categoría) VALUES ('Laptop Gamer X', 15999.99, 5, 'Computo'), ('Mouse inalámbrico', 349.50, 40, 'Accesorios'), ('Teclado mecánico RGB', 1299.00, 15, 'Accesorios'), ('Monitor 27 pulgadas', 4999.99, 8, 'Computo'), ('Silla ergonómica', 2799.00, 12, 'Muebles');
```

```
INSERT INTO clientes (nombre, email, teléfono) VALUES ('Ana Lopez', 'ana@example.com', '5551234567'), ('Bruno Diaz', 'bruno@example.com', '5559876543'), ('Carla Ruiz', 'carla@example.com', '5554567890');
```

Ejecuta todo esto en phpMyAdmin y verifica en el panel izquierdo que aparezca la base "tienda" con sus dos tablas y sus filas. Si algo salió mal, phpMyAdmin te mostrara el error exacto con el número de línea: léelo con calma, casi siempre es una coma faltante o una comilla sin cerrar.

Consejo práctico: guarda este script SQL en un archivo tienda.sql dentro de tu carpeta de prácticas. Cada vez que rompas la base haciendo pruebas (y la vas a romper, es parte del aprendizaje), podrás borrarla y recrearla en segundos ejecutando el script de nuevo. Los profesionales hacen exactamente eso con scripts de "seed" o datos semilla.

El DSN explicado parte por parte

DSN significa Data Source Name, o "nombre de la fuente de datos". Es una cadena de texto que le dice a PDO a que base de datos conectarse, en que servidor vive y con que reglas hablarle. Es como la dirección completa escrita en un sobre: país, ciudad, calle y número, todo en una línea.

Un DSN típico para MySQL se ve así:

```
mysql:host=localhost;dbname=tienda;charset=utf8mb4
```

Vamos parte por parte:

- `mysql:` es el "apellido" del manejador. Le dice a PDO "vamos a hablar con MySQL". Si usaras SQLite diría `sqlite:`, si fuera PostgreSQL diría `pgsql:`. Esta es la única parte que cambia cuando cambias de base de datos: la magia del enchufe universal.
- `host=localhost` significa que la base de datos está en tu misma computadora. En un servidor real aquí iría una dirección IP o un nombre de dominio, por ejemplo `host=192.168.1.50` o `host=db.misitio.com`.
- `dbname=tienda` es el nombre de la base de datos a la que quieres entrar. Recuerda: un servidor MySQL puede tener muchas bases de datos (tienda, escuela, blog); con `dbname` eliges la puerta correcta.
- `charset=utf8mb4` le dice a la conexión "vamos a hablar en utf8mb4", para que los acentos y las enes viajen sin corromperse. Si omites esto, es común ver símbolos raros en lugar de "nino" o "camión". Siempre inclúyelo.

Puedes agregar más piezas separadas por punto y coma, por ejemplo port=3306 si tu MySQL escucha en un puerto distinto. Pero con esas cuatro tienes el 99 por ciento de los casos cubiertos.

Error típico de principiante: escribir mal el nombre de la base (dbname=tiedna) o el host. PDO te respondera con un error de conexión; no entres en panico, revisa letra por letra el DSN contra lo que ves en phpMyAdmin.

new PDO() con try/catch: abrir la puerta con llave

Crear la conexión es instanciar la clase PDO pasandole el DSN, el usuario y la contraseña. Pero ojo: si algo sale mal (servidor apagado, credenciales incorrectas, base inexistente), el constructor de PDO lanza una excepción. Una excepción es como una alarma: si no la atrapas, el script muere de golpe mostrando un mensaje feo. Por eso la conexión SIEMPRE va dentro de un try/catch.

Analogía: conectarte a la base de datos es como intentar abrir una puerta con una llave. El try es "intentó meter la llave y girar"; si la llave es correcta, la puerta se abre y entras. El catch es "si la llave no gira, no rompo la puerta a patadas: me quedó tranquilo en el pasillo y aviso con calma que no pude entrar". El programa no debe explotar jamás por un error de conexión: debe atraparlo y responder con elegancia.

El esqueleto básico, que usaras cientos de veces en tu carrera:

```
try { $pdo = new PDO("mysql:host=localhost;dbname=tienda;charset=utf8mb4", "root", "");  
echo "Conexión exitosa"; } catch (PDOException $e) { die("Error de conexión: " .  
$e->getMessage()); }
```

Línea por línea: el try intenta crear el objeto PDO. Si lo logra, \$pdo queda guardado y lo usaras para todas las consultas. Si algo falla, PHP salta automáticamente al catch, donde \$e es el objeto de la excepción y getMessage() trae la descripción del problema. die() detiene el script mostrando un mensaje.

Nota importante: en este ejemplo mostramos el mensaje de error con getMessage() porque estamos practicando en local. En producción JAMÁS le muestras al visitante el error crudo (puede revelar nombres de tablas o rutas de archivos a un atacante). Más adelante, en buenas prácticas, verás qué hacer en su lugar.

PDOException es una clase especial de excepción dedicada a errores de base de datos. Atrapar PDOException en vez de la generica Exception te permite distinguir "falló la base de datos" de "falló otra cosa".

Modos de error: ERRMODE_EXCEPTION

PDO puede comportarse de tres maneras cuando una consulta falla, y tu eliges cual con el atributo PDO::ATTR_ERRMODE:

- PDO::ERRMODE_SILENT (el modo por defecto): no avisa nada, solo guarda el código de error para que lo consultes a mano. Es como un empleado que rompe algo y se queda callado esperando que tu lo descubras. Es el peor modo para aprender y para trabajar.
- PDO::ERRMODE_WARNING: emite una advertencia de PHP pero el script sigue corriendo. Mejor que el silencio, pero tu código puede seguir adelante con datos incompletos sin que te des cuenta.
- PDO::ERRMODE_EXCEPTION: lanza una excepción en cuanto algo falla. El error no pasa desapercibido: tu try/catch lo atrapa y decides que hacer. Este es el modo profesional y el que usaras siempre.

Se configura justo después de crear la conexión:

```
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

Hazlo una sola vez, en tu archivo de conexión reutilizable, y olvídate. Con este modo activado, cada consulta mal escrita, cada tabla inexistente y cada violación de reglas te avisara de inmediato con un mensaje claro. Es como ponerle alarma a la bodega: prefieres enterarte en el momento a descubrir el desastre semanas después.

Advertencia: si no configuras este modo, PDO trabaja en silencio y pasaras horas preguntandote por qué tu INSERT "no hace nada". Cuando una consulta parezca no tener efecto y no veas ningun error, lo primero que debes revisar es si activaste ERRMODE_EXCEPTION.

query() vs prepare()/execute(): dos formas de preguntar

PDO te ofrece dos métodos para enviar SQL a la base de datos, y elegir el correcto es una de las decisiones más importantes de esta unidad.

query() ejecuta directamente una consulta escrita a mano en el código. Sirve para consultas fijas, sin datos del usuario:

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

Es rápido y comodo para consultas que nunca cambian. Pero tiene una regla de oro: JAMÁS concatenes datos que vengan del usuario dentro de un query(). Hacer eso es dejar la puerta abierta al ataque de inyección SQL que veremos más adelante.

`prepare()` + `execute()` es el método profesional: primero preparas una plantilla de la consulta con marcadores (placeholders), y después la ejecutas pasándole los valores por separado. La base de datos trata la estructura y los datos como cosas distintas, y así ningún dato malicioso puede convertirse en código.

```
$stmt = $pdo->prepare("SELECT * FROM productos WHERE categoría = :cat");  
$stmt->execute(["cat" => "Computo"]);
```

Analogía: `query()` es como dictarle una carta completa a tu secretaria por teléfono, incluyendo los datos del cliente en medio del dictado; si alguien te susurra datos falsos mientras dictas, la carta sale corrupta. `prepare()/execute()` es como entregarle un formato impreso con espacios en blanco: la estructura ya está fija en el papel y los datos solo rellenan los espacios, sin poder cambiar el formato. El formato (la consulta) y los datos (los valores) viajan por caminos separados.

Regla práctica que te simplifica la vida: si la consulta incluye aunque sea UN dato que venga de fuera (formulario, URL, cookie, archivo), usa `prepare()/execute()`. Si la consulta es 100 por ciento fija y escrita por ti, `query()` está bien. Muchos programadores, para no pensarlo dos veces, usan `prepare()` siempre. No es mala idea.

`fetch()` y `fetchAll()`: recoger los resultados

Cuando ejecutas un `SELECT`, la base de datos no te devuelve un arreglo de PHP directamente: te devuelve un objeto `PDOStatement`, que es como una bandeja con las filas encontradas. Para convertirlas en datos de PHP usas `fetch()` o `fetchAll()`.

`fetch()` saca UNA fila de la bandeja cada vez que lo llamas. Es ideal para recorrer resultados grandes con un `while`, porque solo mantiene una fila en memoria a la vez:

```
$stmt = $pdo->query("SELECT nombre, precio FROM productos"); while ($fila =  
$stmt->fetch(PDO::FETCH_ASSOC)) { echo $fila["nombre"] . " cuesta " . $fila["precio"] . "<br>";  
}
```

`fetchAll()` saca TODAS las filas de golpe y te las entrega como un arreglo de arreglos. Es cómodo para resultados pequeños (una lista de productos, por ejemplo), pero cuidado con tablas de millones de filas: te puedes comer toda la memoria del servidor.

```
$todos = $stmt->fetchAll(PDO::FETCH_ASSOC); foreach ($todos as $fila) { echo  
$fila["nombre"] . "<br>"; }
```

El parámetro PDO::FETCH_ASSOC es el modo de lectura más usado: cada fila llega como un arreglo asociativo donde las llaves son los nombres de las columnas (\$fila["nombre"], \$fila["precio"]). Es legible y directo.

Existen otros modos que conviene conocer aunque uses FETCH_ASSOC el 95 por ciento del tiempo:

- PDO::FETCH_NUM: cada fila es un arreglo numerado (\$fila[0], \$fila[1]). Rápido pero ilegible: si cambias el orden de las columnas en el SELECT, tu código se rompe en silencio.
- PDO::FETCH_BOTH (el modo por defecto si no especificas nada): cada fila trae AMBAS llaves, numericas y asociativas, duplicando la memoria. Por eso siempre conviene pedir FETCH_ASSOC explícitamente.
- PDO::FETCH_OBJ: cada fila es un objeto (\$fila->nombre). A algunos les gusta este estilo orientado a objetos.
- PDO::FETCH_COLUMN: trae una sola columna de todas las filas como arreglo simple. Útil para listas de nombres o ids.

Puedes fijar el modo por defecto una sola vez en la conexión para no repetirlo en cada fetch:

```
$pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
```

Detalle que causa confusiones: fetch() devuelve false cuando ya no hay más filas. Por eso el while (\$fila = \$stmt->fetch(...)) funciona: en la última vuelta \$fila vale false y el ciclo termina. Y si la consulta no encontró NADA, fetchAll() devuelve un arreglo vacío (no false), así que puedes contar con count() sin miedo.

INSERT, UPDATE y DELETE desde PHP

Leer datos es la mitad del trabajo; la otra mitad es escribirlos. Las tres operaciones de escritura usan prepare()/execute() porque casi siempre incluyen datos del usuario.

INSERT: agregar una fila nueva.

```
$stmt = $pdo->prepare("INSERT INTO productos (nombre, precio, stock, categoría) VALUES (:nombre, :precio, :stock, :categoría)");  
$stmt->execute([":nombre" => "Audifonos Bluetooth",  
":precio" => 899.99, ":stock" => 25, ":categoría" => "Accesorios" ]);
```

Después de un INSERT, es común necesitar el id que la base asignó automáticamente. lastInsertId() te lo da:

```
$nuevold = $pdo->lastInsertId(); echo "Producto guardado con id " . $nuevold;
```

UPDATE: modificar filas existentes. La cláusula WHERE es obligatoria y sagrada: sin WHERE, el UPDATE cambia TODAS las filas de la tabla. Lee eso de nuevo: sin WHERE actualizas toda la tabla. Es uno de los accidentes más caros en la historia de las bases de datos.

```
$stmt = $pdo->prepare("UPDATE productos SET precio = :precio WHERE id = :id");  
$stmt->execute([":precio" => 329.50, ":id" => 2]);
```

DELETE: borrar filas. Aplica la misma advertencia del WHERE: un DELETE sin WHERE vacía la tabla completa. En aplicaciones reales muchos prefieren el "borrado lógico" (una columna activo que se pone en 0) en vez de borrar de verdad, pero para practicar usaremos DELETE real.

```
$stmt = $pdo->prepare("DELETE FROM productos WHERE id = :id"); $stmt->execute([":id" => 9]);
```

Para saber cuantas filas afecto una operación de escritura, usa rowCount():

```
$afectadas = $stmt->rowCount(); echo "Filas modificadas: " . $afectadas;
```

Nota honesta: rowCount() en SELECT no es confiable en todos los manejadores (en MySQL suele funcionar, en otros no). Para contar resultados de un SELECT, haz fetchAll() y usa count(), o mejor aun, pide el conteo a la base con SELECT COUNT(*).

Consejo de oficio: antes de ejecutar un UPDATE o DELETE con WHERE en tu código, prueba el mismo WHERE en un SELECT dentro de phpMyAdmin. Si el SELECT muestra exactamente las filas que quieres cambiar, el WHERE está bien. Es un habito de 30 segundos que evita desastres.

Que es la inyección SQL: la historia del ladron

Llegó el momento del tema más importante de la unidad. La inyección SQL es el ataque más famoso contra aplicaciones web y lleva decadas en el top de vulnerabilidades, no porque sea sofisticado, sino porque muchisimos programadores siguen dejando la puerta abierta.

La historia del ladron

Imagina que eres el dueño de una bodega y contratas a un guardia para la entrada. Le das una instrucción escrita: "Deja pasar a quien diga la contraseña correcta". Un día llega un ladron muy astuto. El guardia le pregunta "cual es la contraseña" y el ladron responde: "la contraseña correcta O ... bueno, tu ya estas de guardia, así que dejame pasar".

Si el guardia es ingenuo y solo verifica que la frase contenga las palabras magicas, el ladron entra. El truco del ladron fue meter instrucciones nuevas DENTRO de su respuesta, aprovechando que el guardia mezcla la instrucción original con lo que el visitante dice.

En una aplicación vulnerable pasa exactamente eso. Supon este código de login, MAL escrito, que concatena directamente lo que el usuario escribió:

```
$usuario = $_POST["usuario"]; $clave = $_POST["clave"]; $sql = "SELECT * FROM clientes WHERE email = '$usuario' AND password = '$clave'"; $resultado = $pdo->query($sql);
```

Si un usuario normal escribe ana@example.com y su clave, la consulta queda:

```
SELECT * FROM clientes WHERE email = 'ana@example.com' AND password = 'secreta123'
```

Todo bien. Pero el ladron, en el campo de usuario, escribe esto:

```
' OR '1'='1
```

Y en la clave escribe lo mismo. La consulta resultante es:

```
SELECT * FROM clientes WHERE email = " OR '1'='1' AND password = " OR '1'='1'
```

Vamos paso a paso como lo "piensa" la base de datos. Primero sustituye los valores tal cual llegaron, sin preguntar si son legitimos. La condición email = " es falsa (nadie tiene email vacío), pero luego viene OR '1'='1'. La expresión '1'='1' compara el texto 1 con el texto 1: siempre es verdadera. En lógica, FALSO OR VERDADERO da VERDADERO. Lo mismo pasa con la clave. Resultado: la condición completa es verdadera para TODAS las filas, y la consulta devuelve el primer cliente de la tabla. El ladron entro sin conocer ninguna contraseña.

Y eso es solo el principio. Con la misma técnica un atacante puede agregar UNION SELECT para robar tablas enteras, o encadenar consultas para borrar datos. Todo porque el programa mezclo código (la consulta) con datos (lo que escribió el usuario) en una sola cadena.

Por qué funciona el ataque: la base de datos no distingue entre "el SQL que escribió el programador" y "el texto que escribió el visitante" cuando todo llega pegado en una sola cadena. Para ella todo es una orden. La inyección SQL es, en el fondo, un problema de confundir datos con instrucciones.

Sentencias preparadas: el molde que deja fuera al ladron

La solución son las sentencias preparadas con placeholders (marcadores de posición), que ya viste en acción. La idea es separar de forma estricta la estructura de la consulta de los datos:

```
$stmt = $pdo->prepare("SELECT * FROM clientes WHERE email = :email AND password = :clave"); $stmt->execute([":email" => $usuario, ":clave" => $clave]);
```

Analogía del molde: imagina un molde para hacer gelatinas con forma de estrella. Eches lo que echas dentro (gelatina de fresa, de limon o hasta tierra), la forma que sale SIEMPRE es una estrella: el contenido jamás puede cambiar el molde. La consulta preparada es el molde (la estructura SQL queda fija cuando llamas a prepare) y los datos son la gelatina (entran después, con execute, y solo pueden rellenar los huecos, nunca romper el molde).

Qué pasa si el ladron intenta su truco contra una consulta preparada: el texto ' OR '1'='1 llega como DATO, y la base lo compara literalmente contra la columna email. Busca un cliente cuyo email sea exactamente el texto raro ' OR '1'='1. No existe ninguno, la consulta devuelve cero filas y el ladron se queda fuera. El ataque muere porque los datos ya no pueden disfrazarse de código.

Hay dos estilos de placeholders:

1. Placeholders con nombre: :email, :clave, :nombre. Se ven así en la consulta y se pasan como arreglo asociativo en execute. Son más legibles, puedes repetirlos y el orden no importa.

```
$stmt = $pdo->prepare("INSERT INTO productos (nombre, precio) VALUES (:nombre, :precio)"); $stmt->execute([":nombre" => "Webcam HD", ":precio" => 749.00]);
```

2. Placeholders de posición (signos de interrogación): ?, ?, ?. Más cortos, pero los valores deben ir en el MISMO orden en un arreglo numerado. Si los desordenas, los datos caen en las columnas equivocadas sin que nadie te avise.

```
$stmt = $pdo->prepare("INSERT INTO productos (nombre, precio) VALUES (?, ?)"); $stmt->execute(["Webcam HD", 749.00]);
```

Puedes elegir el estilo que prefieras; lo importante es no mezclarlos en la misma consulta. Con nombre es más claro para consultas largas, con ? es más rápido para consultas cortas. En este libro usaremos sobre todo los de nombre porque el código se lee mejor.

Detalle técnico que vale oro: con placeholders NO debes poner comillas alrededor del marcador. Esto está MAL: WHERE email = ':email'. PDO se encarga de citar y escapar los valores por ti; si tu agregas comillas, buscará literalmente el texto ":email". Deja el placeholder limpio, sin comillas.

Otro detalle: los placeholders solo sirven para VALORES (lo que va después de = en un WHERE, los VALUES de un INSERT). No puedes usar un placeholder para el nombre de una tabla o de una columna (por ejemplo ORDER BY :columna no funciona como esperarías). Si

necesitas un nombre de columna dinámico, validalo contra una lista blanca de nombres permitidos en tu código PHP.

Errores comunes y como evitarlos

1. No capturar la excepción de conexión. Si `new PDO()` falla fuera de un `try/catch`, el usuario ve un error fatal con detalles técnicos. Toda conexión va en `try/catch`, sin excepciones a esta regla.
2. Mezclar placeholders con nombre y de posición en la misma consulta. PDO lo prohíbe y lanza error. Elige un estilo por consulta y mantenlo.
3. Olvidar llamar a `execute()` después de `prepare()`. `prepare()` solo deja la consulta lista; sin `execute()` nada viaja a la base de datos. Síntoma clásico: "mi INSERT no guarda nada y no hay error". Revisa que el `execute` exista.
4. Credenciales mal escritas. Contraseña con un espacio de más, usuario "Root" con mayúscula en vez de "root", dbname con el nombre equivocado. El mensaje dirá algo como "Access denied" o "Unknown database". Copia los datos desde phpMyAdmin en vez de escribirlos de memoria.
5. Poner comillas alrededor de los placeholders, como vimos arriba. El marcador va desnudo.
6. Usar `query()` con datos del usuario concatenados. Es el pecado original de esta unidad. Si ves una variable de `$_POST` o `$_GET` pegada dentro de una cadena SQL, eso es una vulnerabilidad esperando a ser explotada.
7. No activar `ERRMODE_EXCEPTION` y luego preguntarse por qué "no pasa nada". Configuralo siempre en tu conexión.
8. Confiar en `rowCount()` para SELECT. Usa `COUNT(*)` en SQL o `count()` sobre `fetchAll()`.
9. Olvidar el WHERE en UPDATE o DELETE durante las pruebas. Prueba primero el WHERE con un SELECT en phpMyAdmin.
10. Guardar contraseñas en texto plano. PDO no te protege de esto: es tu responsabilidad guardar hashes con `password_hash()` (lo viste en la unidad de seguridad... y si no, repasa: jamás guardes la clave tal cual la escribió el usuario).

Buenas prácticas

Un archivo `conexión.php` reutilizable. No repitas el DSN, el usuario y la contraseña en cada script: centralízalos en un solo archivo e incluyelo con `require` donde lo necesites. Si mañana cambian las credenciales, las cambias en un solo lugar.

En producción, nunca muestres el error crudo al usuario. En el catch de conexión, guarda el detalle en un archivo de log (`error_log()`) y muestra al visitante un mensaje generico: "Estamos teniendo problemas técnicos, intenta más tarde". Los mensajes técnicos son para ti, no para el público ni para los atacantes.

Cierra la conexión cuando termines, asignando null a la variable: `$pdo = null;`. En scripts web cortos PHP la cierra sola al terminar, pero hacerlo explícito es una buena costumbre, sobre todo en scripts largos o que abren varias conexiones.

Valida antes de consultar: revisa en PHP que los datos tengan el formato esperado (un email parece email, un precio es numerico) ANTES de mandarlos a la base. Las sentencias preparadas te protegen de la inyección, pero no de datos absurdos como un precio de -500.

Usa transacciones cuando varias operaciones deben funcionar juntas o no funcionar: por ejemplo, descontar stock y registrar la venta. Si la segunda falla después de que la primera funcionó, tus datos quedan inconsistentes. `$pdo->beginTransaction()`, luego `commit()` si todo salio bien o `rollBack()` si algo falló. Es un tema avanzado que verás a fondo en la siguiente unidad, pero que sepas que existe.

Nombra tus variables con claridad: `$stmt` para la sentencia preparada, `$pdo` para la conexión, `$fila` para cada registro. Tu yo del futuro te lo agradecerá cuando lea el código dentro de seis meses.

Ejemplos paso a paso

En esta sección cada ejemplo es un script completo que puedes copiar a tu carpeta de XAMPP (`htdocs/tienda`, por ejemplo) y ejecutar en el navegador. Todos usan el mismo archivo `conexión.php` del ejemplo 1. Después de cada bloque de código viene la explicación línea por línea.

Ejemplo 1: conexión.php reutilizable

Este es el archivo que todos los demás van a incluir. Guardalo como `conexión.php`.

```

<?php
$host = "localhost";
$dbname = "tienda";
$usuario = "root";
$clave = "";

try {
    $pdo = new PDO(
        "mysql:host=$host;dbname=$dbname;charset=utf8mb4",
        $usuario,
        $clave
    );
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
} catch (PDOException $e) {
    error_log("Error de BD: " . $e->getMessage());
    die("Estamos teniendo problemas tecnicos. Intenta mas tarde.");
}

```

Explicación línea por línea:

- Las primeras cuatro líneas guardan los datos de conexión en variables con nombres claros. Si cambias de servidor, solo editas aquí.
- El try intenta crear el objeto PDO con el DSN armado con esas variables. Observa que el DSN usa comillas dobles para que PHP sustituya \$host y \$dbname dentro de la cadena.
- setAttribute con ERRMODE_EXCEPTION activa el modo de errores con excepciones: cualquier falló futuro avisara de inmediato.
- setAttribute con DEFAULT_FETCH_MODE deja FETCH_ASSOC como modo de lectura por defecto, así no tienes que escribirlo en cada fetch.
- Si algo falla, el catch registra el detalle técnico en el log del servidor con error_log() (solo tu lo verás) y muestra al visitante un mensaje generico con die(). Así cumples la buena práctica de no exponer errores crudos.
- Nota: este archivo NO cierra el PHP con ?> al final. Omitir la etiqueta de cierre en archivos que solo tienen código es una práctica recomendada: evita espacios o saltos de línea accidentales que romperian las cabeceras HTTP.

Para usarlo en cualquier script: require "conexión.php"; y automáticamente tendrás la variable \$pdo lista.

Ejemplo 2: listar productos en una tabla HTML

Guardalo como lista.php. Lee todos los productos y los muestra en una tabla.

```
<?php
require "conexion.php";

$stmt = $pdo->query("SELECT id, nombre, precio, stock, categoria FROM productos
ORDER BY nombre");
$productos = $stmt->fetchAll();
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Productos</title></head>
<body>
<h1>Catalogo de productos</h1>
<table border="1" cellpadding="6">
<tr><th>ID</th><th>Nombre</th><th>Precio</th><th>Stock</th><th>Categoria</th></tr>
<?php foreach ($productos as $p): ?>
<tr>
<td><?php echo $p["id"]; ?></td>
<td><?php echo htmlspecialchars($p["nombre"]); ?></td>
<td><?php echo number_format($p["precio"], 2); ?></td>
<td><?php echo $p["stock"]; ?></td>
<td><?php echo htmlspecialchars($p["categoria"]); ?></td>
</tr>
<?php endforeach; ?>
</table>
<p>Total de productos: <?php echo count($productos); ?></p>
</body>
</html>
```

Explicación:

- require "conexión.php" nos da \$pdo ya configurado.
- query() ejecuta un SELECT fijo: no hay datos del usuario, así que es seguro usarlo directamente. ORDER BY nombre los ordena alfabéticamente.
- fetchAll() trae todas las filas como arreglos asociativos (gracias al modo por defecto que fijamos en conexión.php).
- El HTML mezcla PHP con la sintaxis alternativa foreach (...): ... endforeach;, que es más legible dentro de plantillas.

- htmlspecialchars() al mostrar nombre y categoría evita ataques XSS: si un producto se llamara "<script>alert(1)</script>", se mostraria como texto y no se ejecutaria. Regla: todo dato que salga a HTML pasa por htmlspecialchars().
- number_format(\$p["precio"], 2) muestra el precio con 2 decimales y formato bonito.
- count(\$productos) cuenta las filas porque fetchAll devolvio un arreglo normal.

Ejemplo 3: insertar un producto con formulario

Guardalo como agregar.php. Muestra el formulario y, al enviarse, guarda el producto.

```
<?php
require "conexion.php";
$mensaje = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"]);
    $precio = (float) $_POST["precio"];
    $stock = (int) $_POST["stock"];
    $categoria = trim($_POST["categoria"]);

    if ($nombre === "" || $precio <= 0) {
        $mensaje = "Nombre y precio valido son obligatorios.";
    } else {
        $stmt = $pdo->prepare(
            "INSERT INTO productos (nombre, precio, stock, categoria)
            VALUES (:nombre, :precio, :stock, :categoria)"
        );
        $stmt->execute([
            ":nombre" => $nombre,
            ":precio" => $precio,
            ":stock" => $stock,
            ":categoria" => $categoria
        ]);
        $mensaje = "Producto guardado con id " . $pdo->lastInsertId();
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Agregar producto</title></head>
<body>
<h1>Agregar producto</h1>
<?php if ($mensaje !== ""): ?>
<p><?php echo htmlspecialchars($mensaje); ?></p>
```

```

<?php endif; ?>
<form method="post">
Nombre: <input type="text" name="nombre" required><br><br>
Precio: <input type="number" step="0.01" name="precio" required><br><br>
Stock: <input type="number" name="stock" value="0"><br><br>
Categoria: <input type="text" name="categoria"><br><br>
<button type="submit">Guardar</button>
</form>
<p><a href="lista.php">Ver catalogo</a></p>
</body>
</html>

```

Explicación:

- Detectamos el envío con `REQUEST_METHOD === "POST"`: el mismo script muestra el formulario (GET) y procesa los datos (POST).
- `trim()` quita espacios al inicio y final; `(float)` y `(int)` convierten los textos del formulario a números. Nunca confíes en que el navegador mandó números: conviértelos tu.
- Validamos antes de tocar la base: nombre no vacío y precio mayor que cero. Las sentencias preparadas evitan la inyección, pero la validación evita datos absurdos.
- `prepare()` con placeholders con nombre y `execute()` con el arreglo asociativo. Los datos y la estructura viajan separados: blindado contra inyección.
- `lastInsertId()` recupera el id automatico para confirmarle al usuario que se guardó.
- El formulario usa `method="post"` para que los datos no viajen en la URL.

Pruebalo: agrega un producto y luego abre `lista.php` para verlo. Después intenta escribir en el nombre algo con comilla simple, como "Teclado de Ana's": con la sentencia preparada se guarda perfecto; con concatenación habría roto la consulta (o peor).

Ejemplo 4: buscar productos por nombre con placeholder

Guardalo como `buscar.php`. Un buscador que filtra por texto parcial usando LIKE.

```

<?php
require "conexion.php";
$resultados = [];
$texto = "";

if (isset($_GET["q"])) {
    $texto = trim($_GET["q"]);
    $stmt = $pdo->prepare(
        "SELECT id, nombre, precio FROM productos WHERE nombre LIKE :busqueda ORDER
        BY nombre"
    );
    $stmt->execute([":busqueda" => "%" . $texto . "%"]);
    $resultados = $stmt->fetchAll();
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Buscar</title></head>
<body>
<h1>Buscar productos</h1>
<form method="get">
<input type="text" name="q" value="<?php echo htmlspecialchars($texto); ?>">
<button type="submit">Buscar</button>
</form>
<?php if (isset($_GET["q"])): ?>
<h2>Resultados para "<?php echo htmlspecialchars($texto); ?>" (<?php echo
count($resultados); ?>)</h2>
<ul>
<?php foreach ($resultados as $p): ?>
<li><?php echo htmlspecialchars($p["nombre"]); ?> - $<?php echo
number_format($p["precio"], 2); ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>
</body>
</html>

```

Explicación:

- El formulario usa GET porque una búsqueda es una operación de lectura: así la URL se puede compartir y marcar como favorita.
- LIKE :busqueda busca coincidencias parciales. Los signos % se agregan en PHP, alrededor del texto: "%" . \$texto . "%" significa "que contenga el texto en cualquier parte".

Importante: los % van en el VALOR, no en la consulta; el placeholder sigue siendo un simple marcador.

- Aunque el texto viene de la URL (dato totalmente controlado por el visitante), al pasar por el placeholder es inofensivo: si alguien busca ' OR '1'='1, la base buscara literalmente un producto con ese nombre raro y no encontrara nada.
- isset(\$_GET["q"]) distingue "aun no buscó nada" de "buscó y no hubo resultados": solo mostramos la sección de resultados después de una búsqueda.
- De nuevo htmlspecialchars() en todo lo que se imprime y viene del usuario.

Ejemplo 5: actualizar el precio de un producto

Guardalo como editar.php. Recibe el id por URL, muestra el precio actual y permite cambiarlo.

```

<?php
require "conexion.php";
$id = isset($_GET["id"]) ? (int) $_GET["id"] : 0;
$mensaje = "";

$stmt = $pdo->prepare("SELECT nombre, precio FROM productos WHERE id = :id");
$stmt->execute(["id" => $id]);
$producto = $stmt->fetch();

if (!$producto) {
    die("Producto no encontrado.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nuevoPrecio = (float) $_POST["precio"];
    if ($nuevoPrecio <= 0) {
        $mensaje = "El precio debe ser mayor que cero.";
    } else {
        $upd = $pdo->prepare("UPDATE productos SET precio = :precio WHERE id = :id");
        $upd->execute(["precio" => $nuevoPrecio, "id" => $id]);
        $mensaje = "Precio actualizado. Filas afectadas: " . $upd->rowCount();
        $producto["precio"] = $nuevoPrecio;
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Editar precio</title></head>
<body>
<h1>Editar: <?php echo htmlspecialchars($producto["nombre"]); ?></h1>
<?php if ($mensaje !== ""): ?>
<p><?php echo htmlspecialchars($mensaje); ?></p>
<?php endif; ?>
<form method="post">
Precio actual: $<?php echo number_format($producto["precio"], 2); ?><br><br>
Nuevo precio: <input type="number" step="0.01" name="precio" required><br><br>
<button type="submit">Actualizar</button>
</form>
<p><a href="lista.php">Volver al catalogo</a></p>
</body>
</html>

```

Explicación:

- (int) \$_GET["id"] convierte el id a entero: si alguien escribe ?id=hola, se vuelve 0 y no rompe nada. Convertir a int es una defensa extra barata para identificadores numericos.
- Primero hacemos SELECT del producto. Si fetch() devuelve false, el producto no existe y detenemos con die(). Nunca muestres un formulario de edición para algo que no existe.
- El UPDATE lleva su WHERE id = :id sagrado. Sin el, cambiarías los precios de TODA la tienda. Repite conmigo: UPDATE siempre lleva WHERE.
- rowCount() aquí SI es confiable porque es una operación de escritura: te dice cuantas filas se modificaron.
- Actualizamos \$producto["precio"] en memoria para que el formulario muestre el valor nuevo sin recargar.

Para probarlo, entra a `editar.php?id=2` (cambia el número por un id real de tu tabla).

Ejemplo 6: eliminar con confirmación

Borrar es peligroso, así que este ejemplo pide confirmación en dos pasos. Guardalo como `eliminar.php`.

```

<?php
require "conexion.php";
$id = isset($_GET["id"]) ? (int) $_GET["id"] : 0;

$stmt = $pdo->prepare("SELECT nombre FROM productos WHERE id = :id");
$stmt->execute(["id" => $id]);
$producto = $stmt->fetch();
if (!$producto) {
    die("Producto no encontrado.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST" && isset($_POST["confirmar"])) {
    $del = $pdo->prepare("DELETE FROM productos WHERE id = :id");
    $del->execute(["id" => $id]);
    header("Location: lista.php");
    exit;
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Eliminar</title></head>
<body>
<h1>Eliminar producto</h1>
<p>Seguro que quieres eliminar "<?php echo htmlspecialchars($producto["nombre"]);
?>" (id <?php echo $id; ?>)? Esta accion no se puede deshacer.</p>
<form method="post">
<button type="submit" name="confirmar" value="1">Si, eliminar</button>
<a href="lista.php">Cancelar</a>
</form>
</body>
</html>

```

Explicación:

- El borrado real solo ocurre con POST y con el campo "confirmar" presente. Si alguien visita eliminar.php?id=3 con GET, solo ve la pregunta, no se borra nada. Esto evita que un buscador o un enlace malicioso borre datos con solo cargarse.
- El DELETE con WHERE id = :id y placeholder: el id viaja como dato, no como código.
- header("Location: lista.php") redirige al catálogo después de borrar, y exit detiene el script. Es el patrón "POST-Redirect-GET": evita que al recargar la página se repita el borrado.
- El mensaje advierte que no se puede deshacer: en sistemas reales considera el borrado lógico (columna activo) en vez de DELETE físico.

Ejemplo 7: inyección SQL vulnerable vs protegida, lado a lado

Este ejemplo no es para usarlo en producción: es una demostración para que VEAS el ataque con tus propios ojos. Crea login_vulnerable.php y login_seguro.php con el mismo formulario, cambiando solo la consulta.

Version VULNERABLE (no la uses en proyectos reales):

```
<?php
require "conexion.php";
$mensaje = "";
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $email = $_POST["email"];
    $sql = "SELECT * FROM clientes WHERE email = '$email'";
    $stmt = $pdo->query($sql);
    $cliente = $stmt->fetch();
    $mensaje = $cliente ? "Bienvenido, " . $cliente["nombre"] : "No encontrado";
}
?>
<form method="post">
Email: <input type="text" name="email">
<button type="submit">Entrar</button>
<p><?php echo htmlspecialchars($mensaje); ?></p>
</form>
```

Version PROTEGIDA:

```
<?php
require "conexion.php";
$mensaje = "";
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $email = $_POST["email"];
    $stmt = $pdo->prepare("SELECT * FROM clientes WHERE email = :email");
    $stmt->execute(["email" => $email]);
    $cliente = $stmt->fetch();
    $mensaje = $cliente ? "Bienvenido, " . $cliente["nombre"] : "No encontrado";
}
?>
<form method="post">
Email: <input type="text" name="email">
<button type="submit">Entrar</button>
<p><?php echo htmlspecialchars($mensaje); ?></p>
</form>
```

Explicación comparativa, paso a paso:

- La única diferencia entre ambas es como viaja \$email: pegado dentro de la cadena SQL (vulnerable) o como valor separado en execute() (protegida). Todo lo demás es idéntico.
- Prueba de ataque: en la versión vulnerable escribe en el campo email exactamente esto: ' OR '1'='1. La consulta se convierte en `SELECT * FROM clientes WHERE email = " OR '1'='1'`, la condición siempre es verdadera y el sistema te da la bienvenida como el primer cliente de la tabla. Acabas de "entrar" sin ser nadie.
- Haz la misma prueba en la versión protegida: la base busca un cliente cuyo email sea literalmente el texto ' OR '1'='1, no existe, y responde "No encontrado". El ataque se estrella contra el molde.
- Conclusión que debes grabarte: la seguridad no está en "revisar que el usuario no escriba cosas raras" (los atacantes siempre encuentran algo raro nuevo), sino en separar estructura y datos con sentencias preparadas. Es una solución estructural, no un parche.

Ejemplo 8: paginación simple con LIMIT

Cuando una tabla tiene cientos de productos, no los muestras todos de golpe: los divides en páginas. Guardalo como `páginas.php`.

```

<?php
require "conexion.php";
$porPagina = 3;
$pagina = isset($_GET["pag"]) ? max(1, (int) $_GET["pag"]) : 1;
$inicio = ($pagina - 1) * $porPagina;

$total = $pdo->query("SELECT COUNT(*) AS n FROM productos")->fetch()["n"];
$totalPaginas = (int) ceil($total / $porPagina);

$stmt = $pdo->prepare(
    "SELECT id, nombre, precio FROM productos ORDER BY id LIMIT :inicio, :cantidad"
);
$stmt->bindValue(":inicio", $inicio, PDO::PARAM_INT);
$stmt->bindValue(":cantidad", $porPagina, PDO::PARAM_INT);
$stmt->execute();
$productos = $stmt->fetchAll();
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Productos por pagina</title></head>
<body>
<h1>Pagina <?php echo $pagina; ?> de <?php echo $totalPaginas; ?></h1>
<ul>
<?php foreach ($productos as $p): ?>
<li><?php echo htmlspecialchars($p["nombre"]); ?> - $<?php echo
number_format($p["precio"], 2); ?></li>
<?php endforeach; ?>
</ul>
<p>
<?php if ($pagina > 1): ?>
<a href="?pag=<?php echo $pagina - 1; ?>">Anterior</a>
<?php endif; ?>
<?php if ($pagina < $totalPaginas): ?>
<a href="?pag=<?php echo $pagina + 1; ?>">Siguiete</a>
<?php endif; ?>
</p>
</body>
</html>

```

Explicación:

- \$porPagina define el tamaño de cada página. \$pagina se lee de la URL y max(1, ...) evita números menores a 1.

- \$inicio es el desplazamiento: página 1 empieza en 0, página 2 en 3, página 3 en 6. Es la cuenta (página - 1) * porPagina.
- SELECT COUNT(*) AS n cuenta el total de filas sin traerlas: barato y rápido. ceil() redondea hacia arriba para saber cuantas páginas hay.
- LIMIT :inicio, :cantidad le pide a la base solo el pedazo necesario. Aquí usamos bindValue() con PDO::PARAM_INT en vez de execute() con arreglo, porque LIMIT necesita números enteros garantizados: PARAM_INT le dice a PDO "esto es un entero, mandalo como entero". Es el detalle fino que evita errores sutiles con LIMIT y placeholders.
- Los enlaces Anterior/Siguiente solo aparecen cuando tiene sentido: no hay "Anterior" en la página 1 ni "Siguiente" en la última.
- Pruebalo con ?pag=1, ?pag=2 y también con ?pag=999 o ?pag=-5 para ver como el código se defiende solo.

Con estos 8 ejemplos tienes el kit completo de operaciones: conectar, leer, crear, buscar, actualizar, borrar, defenderte y paginar. Todo lo demás que harás con bases de datos en PHP son variaciones de estos patrones.

Práctica

Es hora de ensuciarte las manos. Cada ejercicio usa la base "tienda" y el archivo conexión.php de los ejemplos. Intenta resolverlo por tu cuenta antes de mirar la solución: la frustración de atorarte 20 minutos vale más que leer 20 soluciones.

Ejercicio 1

Enunciado: crea un script top_caros.php que muestre los 3 productos más caros de la tienda en una lista HTML ordenada, con su nombre y precio. Si hay menos de 3 productos, muestra los que haya sin errores.

Solución 1

```

<?php
require "conexion.php";

$stmt = $pdo->query(
    "SELECT nombre, precio FROM productos ORDER BY precio DESC LIMIT 3"
);
$top = $stmt->fetchAll();
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Los 3 mas caros</title></head>
<body>
<h1>Los 3 productos mas caros</h1>
<?php if (count($top) === 0): ?>
<p>No hay productos registrados.</p>
<?php else: ?>
<ol>
<?php foreach ($top as $p): ?>
<li><?php echo htmlspecialchars($p["nombre"]); ?> - $<?php echo
number_format($p["precio"], 2); ?></li>
<?php endforeach; ?>
</ol>
<?php endif; ?>
</body>
</html>

```

Explicación paso a paso:

- ORDER BY precio DESC ordena de mayor a menor precio. DESC significa descendente; ASC (el valor por defecto) seria ascendente.
- LIMIT 3 corta el resultado a las primeras 3 filas. La consulta es fija, sin datos del usuario, así que query() es correcto.
- fetchAll() trae las filas; si la tabla está vacía devuelve un arreglo vacío, no un error.
- count(\$top) === 0 detecta el caso "sin productos" y muestra un mensaje amable en vez de una lista vacía. Programar pensando en los casos borde (tablas vacías, un solo registro) es lo que separa al principiante del profesional.
- La lista ordenada numera automáticamente del 1 al 3: el ranking visual sale gratis.

Ejercicio 2

Enunciado: crea registrar_cliente.php con un formulario que pida nombre, email y teléfono. Al enviarlo, debe validar que el email tenga formato valido y que no exista ya otro cliente con ese email (recuerda que la columna es UNIQUE). Si todo está bien, guarda el cliente y muestra su id; si el email ya existe, muestra un mensaje claro sin romper el programa.

Solución 2

```
<?php
require "conexion.php";
$mensaje = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"]);
    $email = trim($_POST["email"]);
    $telefono = trim($_POST["telefono"]);

    if ($nombre === "" || !filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $mensaje = "Revisa los datos: nombre obligatorio y email valido.";
    } else {
        $check = $pdo->prepare("SELECT id FROM clientes WHERE email = :email");
        $check->execute(["email" => $email]);
        if ($check->fetch()) {
            $mensaje = "Ese email ya esta registrado.";
        } else {
            $stmt = $pdo->prepare(
                "INSERT INTO clientes (nombre, email, telefono)
                VALUES (:nombre, :email, :telefono)"
            );
            $stmt->execute([
                "nombre" => $nombre,
                "email" => $email,
                "telefono" => $telefono
            ]);
            $mensaje = "Cliente registrado con id " . $pdo->lastInsertId();
        }
    }
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Registrar cliente</title></head>
<body>
<h1>Registrar cliente</h1>
```

```

<?php if ($mensaje != ""): ?>
<p><?php echo htmlspecialchars($mensaje); ?></p>
<?php endif; ?>
<form method="post">
Nombre: <input type="text" name="nombre" required><br><br>
Email: <input type="email" name="email" required><br><br>
Telefono: <input type="text" name="telefono"><br><br>
<button type="submit">Registrar</button>
</form>
</body>
</html>

```

Explicación paso a paso:

- `filter_var($email, FILTER_VALIDATE_EMAIL)` es la forma estandar de PHP para validar un email: devuelve false si el formato no es válido. Es validación, no protección contra inyección (de eso se encarga la sentencia preparada).
- Antes de insertar, hacemos un `SELECT` buscando el email. Si `fetch()` devuelve una fila, el email ya existe y mostramos el mensaje sin intentar el `INSERT`.
- Por qué verificar antes si la columna ya es `UNIQUE`: porque el mensaje de error de la base ("Duplicate entry...") es técnico y feo para el usuario. Verificar primero nos permite responder con un mensaje humano. La restricción `UNIQUE` queda como red de seguridad por si dos personas registran el mismo email al mismo tiempo.
- Todo viaja con placeholders con nombre: ni el `SELECT` de verificación ni el `INSERT` concatenan datos.
- `lastInsertId()` confirma el guardado con el id generado.

Ejercicio 3

Enunciado: crea `stock_bajo.php` que muestre en una tabla los productos con stock menor a 10, ordenados del stock más bajo al más alto. Además, al final debe mostrar el valor total del inventario de esos productos (precio * stock sumado). El umbral (10) debe estar en una variable al inicio del script para cambiarlo fácilmente.

Solución 3

```

<?php
require "conexion.php";
$umbral = 10;

$stmt = $pdo->prepare(
    "SELECT nombre, precio, stock FROM productos WHERE stock < :umbral ORDER BY
stock ASC"
);
$stmt->execute(["umbral" => $umbral]);
$productos = $stmt->fetchAll();

$total = 0;
foreach ($productos as $p) {
    $total += $p["precio"] * $p["stock"];
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Stock bajo</title></head>
<body>
<h1>Productos con stock menor a <?php echo $umbral; ?></h1>
<table border="1" cellpadding="6">
<tr><th>Nombre</th><th>Precio</th><th>Stock</th><th>Valor</th></tr>
<?php foreach ($productos as $p): ?>
<tr>
<td><?php echo htmlspecialchars($p["nombre"]); ?></td>
<td><?php echo number_format($p["precio"], 2); ?></td>
<td><?php echo $p["stock"]; ?></td>
<td><?php echo number_format($p["precio"] * $p["stock"], 2); ?></td>
</tr>
<?php endforeach; ?>
</table>
<p>Valor total de este inventario: <?php echo number_format($total, 2); ?></p>
</body>
</html>

```

Explicación paso a paso:

- \$umbral = 10 al inicio: si mañana el jefe dice "ahora avisame con menos de 20", cambias un número y listo. Los valores mágicos regados por el código son una deuda que siempre se cobra.
- Aunque \$umbral lo escribimos nosotros, usamos prepare() con placeholder por disciplina: cuesta lo mismo y mantiene el hábito de nunca concatenar.

- WHERE stock < :umbral filtra, ORDER BY stock ASC ordena del más escaso al menos escaso: lo urgente aparece primero.
- El total se calcula en PHP recorriendo el arreglo: precio * stock por producto, acumulado en \$total. También podría calcularlo SQL con SUM(precio * stock), y sería un buen reto extra intentarlo.
- La columna "Valor" por fila muestra cuanto dinero hay inmovilizado en ese producto: información útil para decidir que resurtir primero.

Ejercicio 4

Enunciado: crea descontar.php que aplique un descuento del 10 por ciento a todos los productos de la categoría "Accesorios". Debe mostrar cuantos productos fueron afectados y listar sus nuevos precios. Ojo: el descuento debe calcularse en SQL, no trayendo los precios a PHP uno por uno.

Solución 4

```

<?php
require "conexion.php";
$categoria = "Accesorios";
$descuento = 0.10;

$stmt = $pdo->prepare(
    "UPDATE productos SET precio = precio * (1 - :descuento) WHERE categoria = :categoria"
);
$stmt->execute([":descuento" => $descuento, ":categoria" => $categoria]);
$afectados = $stmt->rowCount();

$lista = $pdo->prepare(
    "SELECT nombre, precio FROM productos WHERE categoria = :categoria ORDER BY nombre"
);
$lista->execute([":categoria" => $categoria]);
$productos = $lista->fetchAll();
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Descuento aplicado</title></head>
<body>
<h1>Descuento del 10% en <?php echo htmlspecialchars($categoria); ?></h1>
<p>Productos afectados: <?php echo $afectados; ?></p>
<ul>
<?php foreach ($productos as $p): ?>
<li><?php echo htmlspecialchars($p["nombre"]); ?> - $<?php echo
number_format($p["precio"], 2); ?></li>
<?php endforeach; ?>
</ul>
</body>
</html>

```

Explicación paso a paso:

- La clave del ejercicio: SET precio = precio * (1 - :descuento). La base de datos hace la matemática fila por fila, en una sola operación. Traer cientos de precios a PHP, calcular y devolverlos uno por uno sería lento y propenso a errores: deja que la base haga lo que sabe hacer.
- El WHERE categoría = :categoría limita el descuento a la categoría pedida. Sin ese WHERE, toda la tienda quedaría con 10 por ciento de descuento: el accidente clásico.

- :descuento viaja como dato (0.10), así que el porcentaje también podría venir de un formulario sin riesgo.
- rowCount() después del UPDATE dice exactamente cuantos productos cambiaron: es tu comprobante de la operación.
- El segundo SELECT con la misma categoría muestra el resultado para verificar visualmente. En un sistema real, antes de correr este UPDATE en producción lo probarías con un SELECT usando el mismo WHERE, como recomienda la buena práctica.

Ejercicio 5

Enunciado: el dueño de la tienda quiere borrar todos los productos con stock en 0 porque "estorban". Crea limpiar.php que primero muestre cuantos productos tienen stock 0 y sus nombres, y solo los borre si se confirma con un botón (POST). Después de borrar, debe mostrar cuantos se eliminaron. Además, explica en un comentario del código por qué este DELETE es seguro contra el accidente del "WHERE olvidado".

Solución 5

```

<?php
require "conexion.php";
$mensaje = "";

// Paso 1: ver que se va a borrar ANTES de borrarlo
$ver = $pdo->query("SELECT id, nombre FROM productos WHERE stock = 0 ORDER BY
nombre");
$candidatos = $ver->fetchAll();

// Paso 2: borrar solo si hubo confirmacion por POST
if ($_SERVER["REQUEST_METHOD"] === "POST" && isset($_POST["confirmar"])) {
    $del = $pdo->prepare("DELETE FROM productos WHERE stock = :cero");
    $del->execute(["cero" => 0]);
    $mensaje = "Se eliminaron " . $del->rowCount() . " productos.";
    // Recargar la lista para mostrar el estado actual
    $ver = $pdo->query("SELECT id, nombre FROM productos WHERE stock = 0 ORDER BY
nombre");
    $candidatos = $ver->fetchAll();
}
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="utf-8"><title>Limpiar sin stock</title></head>
<body>
<h1>Productos con stock en 0 (<?php echo count($candidatos); ?>)</h1>
<?php if ($mensaje !== ""): ?>
<p><?php echo htmlspecialchars($mensaje); ?></p>
<?php endif; ?>
<ul>
<?php foreach ($candidatos as $c): ?>
<li><?php echo htmlspecialchars($c["nombre"]); ?> (id <?php echo $c["id"]; ?>)</li>
<?php endforeach; ?>
</ul>
<?php if (count($candidatos) > 0): ?>
<form method="post">
<p>Esta accion no se puede deshacer.</p>
<button type="submit" name="confirmar" value="1">Borrar todos los de stock
0</button>
</form>
<?php else: ?>
<p>No hay productos con stock en 0.</p>
<?php endif; ?>
</body>
</html>

```

Explicación paso a paso:

- Paso 1: el SELECT previo muestra exactamente que filas están en la mira. Es la buena práctica de "prueba el WHERE con un SELECT primero", convertida en parte del programa: el usuario VE lo que se va a borrar antes de confirmar.
- Paso 2: el DELETE solo corre con POST + confirmación. Una visita normal con GET jamás borra nada.
- Por qué este DELETE es seguro contra el "WHERE olvidado": el WHERE esta escrito como parte fija de la cadena SQL ("DELETE FROM productos WHERE stock = :cero"), no se construye concatenando variables. El placeholder :cero solo aporta el valor 0, no puede modificar la estructura. Además, como el script muestra primero los candidatos, cualquier error en la condición se detectaría visualmente antes de confirmar. El accidente del WHERE olvidado ocurre cuando el WHERE se arma dinamicamente o se omite por descuido; aquí es estático y visible.
- Después de borrar, se vuelve a consultar para mostrar la lista actualizada (que debería estar vacía) junto con el conteo de eliminados via rowCount().
- Nota de criterio: en un negocio real, borrar productos con stock 0 puede ser mala idea (pierdes el historial y la ficha del producto para cuando vuelva a haber stock). Muchas tiendas prefieren ocultarlos o marcarlos como "agotado". El ejercicio es técnico; la decisión de negocio es otro tema.

Resumen de la unidad

- Las variables de PHP viven solo mientras se ejecuta el script: sin base de datos, tu aplicación olvida todo al recargar. MySQL/MariaDB es el "cuaderno permanente" del servidor donde los datos sobreviven entre visitas.
- XAMPP incluye MariaDB (gemelo de MySQL) y phpMyAdmin, tu laboratorio visual para crear bases, tablas y probar consultas antes de escribirlas en PHP. Datos típicos en local: host localhost, usuario root, contraseña vacía.
- PDO (PHP Data Objects) es la forma moderna, portable y segura de conectarse: un "enchufe universal" que sirve para MySQL y otras bases cambiando apenas el DSN. Las funciones viejas mysql_* están muertas desde PHP 7: no las uses ni en broma.
- El DSN describe la conexión en una cadena: mysql:host=localhost;dbname=tienda;charset=utf8mb4. Cada pieza (manejador, host, base, charset) tiene su función y debes revisarla letra por letra cuando la conexión falle.

- La conexión se abre con `new PDO()` SIEMPRE dentro de `try/catch`, como intentar abrir una puerta con llave: si no abre, el programa avisa con calma en vez de explotar.
- Configura `PDO::ATTR_ERRMODE` en `PDO::ERRMODE_EXCEPTION` justo después de conectar: los errores avisan de inmediato con excepciones en vez de fallar en silencio.
- `query()` es para consultas fijas escritas por ti; `prepare()` + `execute()` es para todo lo que incluya datos externos. Regla de oro: jamás concatenes datos del usuario dentro del SQL.
- `fetch()` saca una fila por vez (ideal con `while` para resultados grandes); `fetchAll()` trae todo de golpe (comodo para listas pequeñas). Usa `PDO::FETCH_ASSOC` para recibir arreglos asociativos con los nombres de las columnas, o fijalo como modo por defecto en la conexión.
- `INSERT` agrega filas (`lastInsertId()` te da el id nuevo), `UPDATE` modifica (siempre con `WHERE`) y `DELETE` borra (siempre con `WHERE`). Prueba el `WHERE` con un `SELECT` en phpMyAdmin antes de ejecutar: un `UPDATE` o `DELETE` sin `WHERE` afecta a toda la tabla.
- La inyección SQL ocurre cuando datos del usuario se pegan dentro de la consulta y la base los confunde con instrucciones: el clásico `' OR '1'='1` convierte la condición en siempre verdadera y el atacante entra sin clave.
- Las sentencias preparadas son la defensa estructural: la consulta es un molde fijo (`prepare`) y los datos solo rellenan los huecos (`execute`) sin poder cambiar la forma. El ataque muere porque los datos ya no pueden disfrazarse de código.
- Placeholders con nombre (`:email`) o de posición (`?`): no los mezcles en la misma consulta, no les pongas comillas alrededor, y recuerda que solo sirven para valores, no para nombres de tablas o columnas.
- Para paginar usa `SELECT COUNT(*)` para el total y `LIMIT` con `bindValue()` y `PDO::PARAM_INT` para traer solo la página necesaria.
- Buenas prácticas que te salvan la carrera: un solo `conexión.php` reutilizable; en producción guarda el error en log y muestra un mensaje generico al usuario; cierra con `$pdo = null`; valida los datos antes de consultar; y muestra con `htmlspecialchars()` todo dato que salga a HTML.
- Errores típicos que ya sabes detectar: excepción sin atrapar, placeholders mezclados, `execute()` olvidado, credenciales mal escritas, comillas alrededor del placeholder, `query()` con datos concatenados, `ERRMODE` sin configurar, `rowCount()` usado en `SELECT`, `WHERE` olvidado y contraseñas en texto plano.

Con esta unidad ya puedes construir aplicaciones que recuerdan: catálogos, registros de usuarios, buscadores y paneles de administración. En la siguiente unidad subiremos la apuesta con transacciones, relaciones entre tablas y consultas más potentes. Práctica estos patrones hasta que te salgan sin pensar: son el pan de cada día del desarrollador PHP.

Una última recomendación antes de cerrar: arma tu propio "kit de supervivencia" con los scripts de esta unidad. Guarda `conexión.php`, `lista.php`, `agregar.php`, `buscar.php`, `editar.php` y `eliminar.php` en una carpeta y reutilízalos como plantilla cada vez que empieces un proyecto nuevo. La mayoría de los sistemas (un blog, un inventario, un directorio de clientes) son variaciones de estos mismos seis scripts con distintas tablas y columnas. Cuando domines este kit, descubrirás que el 80 por ciento del trabajo con bases de datos en PHP es combinar estas piezas. Y cada vez que sientas la tentación de concatenar una variable dentro de un SQL "solo por esta vez", recuerda al ladrón de la historia: la puerta que dejas entreabierta hoy es la que alguien cruzará mañana.

UNIDAD 8: CRUD COMPLETO

CRUD. Cuatro letras que suenan a algo complicado pero que esconden la idea más simple y más poderosa de todo el desarrollo web. Si llegaste hasta aquí, ya sabes crear conexiones a MySQL, insertar datos y consultarlos con PHP. Ahora toca unir todas esas piezas para construir aplicaciones de verdad: programas donde el usuario puede crear, leer, actualizar y borrar registros. Eso es exactamente lo que significa CRUD: Create, Read, Update, Delete. En español: crear, leer, actualizar y eliminar.

En esta unidad vas a construir, paso a paso, un sistema completo de manejo de productos para una tiendita: el clásico ejemplo del negocio que vende abarrotes. Pero no te dejes enganar por lo modesto del ejemplo. El mismo esqueleto que uses aquí para registrar productos con su nombre y precio es el que usan los sistemas reales para manejar alumnos de una escuela, pacientes de una clínica, empleados de una empresa, artículos de un blog o platillos de un restaurante. Si entiendes el CRUD a fondo, entiendes el 90 por ciento de las aplicaciones que existen en la web.

Piensa en la libreta de la tiendita. En cualquier tiendita de barrio hay una libreta, física o mental, donde el dueño anota lo que llega, consulta lo que tiene, corrige los precios cuando cambian y tacha lo que ya no vende. Anotar es crear. Consultar es leer. Corregir es actualizar. Tachar es eliminar. Un sistema CRUD no es otra cosa que esa libreta llevada a la computadora, pero con superpoderes: puede buscar en miles de registros en un segundo, nadie la puede borrar sin permiso, y varias personas pueden usarla al mismo tiempo.

A lo largo de esta unidad vas a ver lo siguiente. Primero, la arquitectura de un CRUD: que archivos lo componen y para que sirve cada uno, con la analogía de las estaciones de una cocina para que entiendas cómo se reparten las tareas. Después, cada operación explicada a fondo: el CREATE con su formulario y su INSERT protegido con sentencias preparadas; el READ que muestra todo en una tabla bonita con botones para editar y eliminar, más el detalle de un solo registro; el UPDATE que precarga el formulario con los datos que ya existen; y el DELETE con su confirmación de seguridad. También vas a aprender dos técnicas que separan a los programadores novatos de los profesionales: los mensajes flash que le avisan al usuario que todo salió bien, y el patrón POST-Redirect-GET que evita que se guarden registros duplicados cuando el usuario refresca la página.

Luego pasaremos a ocho ejemplos completos de código, explicados línea por línea, que forman un mini-CRUD de productos funcional: conexión, listado, crear, editar, eliminar, mensajes flash, página de detalle y una versión mejorada con funciones reutilizables. Después tendrás cinco ejercicios prácticos con sus soluciones: un CRUD de alumnos, un CRUD de

libros, un buscador integrado al CRUD, un campo de foto y paginación de resultados. Para terminar, un resumen con los puntos clave de la unidad.

Una advertencia importante antes de empezar. En esta unidad vas a ver operaciones que borran y modifican datos. Eso significa que una línea mal escrita puede borrar todos tus registros de golpe. No es para asustarte: es para que desde el primer día tomes en serio los buenos hábitos que aquí enseño. El error más famoso de los principiantes, olvidar el WHERE en un UPDATE o un DELETE, ha destruido bases de datos enteras en empresas reales. Aquí lo vas a aprender a evitar de forma casi instintiva. La buena noticia es que sí prácticas en tu base de datos de pruebas, sin miedo, vas a salir de esta unidad con la confianza de construir sistemas que la gente usa de verdad.

Prepara tu editor, tu servidor local con PHP y MySQL, y la base de datos de pruebas que ya usaste en las unidades anteriores. Esta unidad es larga porque el tema lo merece, pero cada sección construye sobre la anterior. Cuando termines, tendrás entre manos un CRUD completo y funcional que puedes presumir como tu primer proyecto real.

Explicación detallada

Que es un CRUD y por qué es el corazón de las apps

CRUD es un acrónimo de las cuatro operaciones básicas que se pueden hacer con la información guardada en una base de datos. Crear un registro nuevo. Leer los registros que existen. Actualizar un registro que cambió. Eliminar un registro que ya no sirve. Eso es todo. Pero detrás de esa simpleza hay una verdad profunda: casi cualquier aplicación que uses a diario es, en el fondo, un CRUD disfrazado.

Piénsalo. Una lista de tareas: anotas una tarea (crear), la consultas (leer), la marcas como hecha o la editas (actualizar) y la borras (eliminar). Un carrito de compras: agregas productos (crear), ves lo que llevas (leer), cambias la cantidad (actualizar) y quitas un producto (eliminar). Las redes sociales: publicas (crear), lees tu muro (leer), editas tu publicación (actualizar) y la borras (eliminar). El panel de administración de cualquier negocio: exactamente lo mismo con sus propios datos.

Por eso decimos que el CRUD es el corazón del 90 por ciento de las aplicaciones. Todo lo demás (gráficos bonitos, buscadores inteligentes, reportes en PDF) son adornos que se le ponen encima. El corazón late igual: crear, leer, actualizar, eliminar. Cuando domines este patrón, vas a reconocerlo en todas partes y construir sistemas nuevos se volverá mucho más fácil, porque siempre empezarás con la misma pregunta: cuáles son los datos y qué operaciones necesita el usuario sobre ellos.

Volvamos a la libreta de la tiendita, porque esa analogía te va a acompañar toda la unidad. El dueño de la tiendita tiene su libreta con columnas: nombre del producto, precio, existencias. Cuando llega mercancía nueva, abre la libreta y anota una línea nueva (crear). Cuando un cliente pregunta el precio de algo, consulta la libreta (leer). Cuando el proveedor sube los precios, borra el precio viejo y escribe el nuevo (actualizar). Cuando un producto se discontinúa, tacha toda la línea (eliminar). Nuestro proyecto de esta unidad hace exactamente eso, pero en un navegador web, con botones y formularios, y con una base de datos MySQL que hace de libreta infinita.

La tabla que usaremos se llamara productos y tendrá estas columnas: id (número automático que identifica cada producto), nombre (texto), precio (número con decimales) y existencia (cantidad en almacén). Cuatro columnas, nada más. Con eso basta para aprender todo lo importante. Cuando termines, podrás cambiar las columnas por las de alumnos, libros o pacientes sin cambiar la lógica.

Antes de escribir código, crea la tabla en tu base de datos con este SQL. Ejecútalo en phpMyAdmin o en tu cliente de MySQL:

```
CREATE TABLE productos ( id INT AUTO_INCREMENT PRIMARY KEY, nombre VARCHAR(100) NOT NULL, precio DECIMAL(10,2) NOT NULL, existencia INT NOT NULL DEFAULT 0 );
```

Y tu conexión a la base de datos, si la olvidaste de la unidad anterior, es así de simple:

```
$conexión = new mysqli("localhost", "root", "", "tiendita"); if ($conexión->connect_error) { die("Error de conexión: " . $conexión->connect_error); }
```

En los ejemplos de esta unidad usare la variable \$conexión para la conexión y el estilo orientado a objetos de mysqli, igual que en las unidades anteriores. Si usas otra base de datos o credenciales distintas, ajusta esa línea en todos los archivos, por eso el archivo de conexión va aparte: para cambiarlo en un solo lugar.

Arquitectura de un CRUD: las estaciones de una cocina

Un CRUD no es un solo archivo gigante con todo el código revuelto. Es un conjunto de archivos pequeños, cada uno con una responsabilidad clara. Piensa en una cocina de restaurante. Hay una estación para picar verduras, otra para cocinar, otra para emplatar y otra para lavar. Ninguna estación hace todo, pero juntas sacan los platillos completos. Un CRUD se organiza igual.

Estos son los archivos de nuestro mini-CRUD de productos:

- `conexión.php`: solo crea la conexión a la base de datos. Es la bodega de la cocina: nadie cocina ahí, pero todos los demás archivos van a pedirle ingredientes (la conexión) con un `require`.
- `index.php`: la lista de productos. Es el menú que ve el cliente: muestra todo lo que hay en una tabla HTML con botones para editar y eliminar cada fila, más un botón grande para agregar un producto nuevo.
- `crear.php`: el formulario para dar de alta un producto nuevo y el código que guarda los datos con `INSERT`. Es la estación donde se prepara un platillo nuevo.
- `editar.php`: parecido a `crear.php`, pero primero lee los datos actuales del producto (con el `id` que llega por la URL) y precarga el formulario. Al guardar, hace `UPDATE` en vez de `INSERT`.
- `eliminar.php`: recibe el `id` del producto y lo borra con `DELETE`, pero solo después de una confirmación.
- `detalle.php`: muestra la ficha completa de un producto. Es opcional pero muy útil en sistemas reales.

Todos estos archivos se comunican con hipervínculos y formularios. El `index` enlaza a `crear.php` con un botón, enlaza a `editar.php?id=5` para editar el producto 5, y tiene un formulario pequeño que manda el `id` a `eliminar.php`. Así de simple: la navegación del CRUD es una red de enlaces entre estaciones.

Organizar el proyecto en carpetas también importa. Para este mini-proyecto basta con una carpeta llamada `crud-productos` que contenga todos los archivos. En proyectos más grandes se usan carpetas como estas: `config` para la conexión, `includes` para funciones reutilizables, y vistas separadas. No te compliques ahora: una sola carpeta ordenada y archivos con nombres claros (`crear`, `editar`, `eliminar`, `index`) ya es arquitectura profesional.

Una regla de oro: cada archivo hace una sola cosa y la hace bien. Si tu `crear.php` empieza a crecer porque también lista, edita y borra, estas regresando a la cocina donde una sola

persona pica, cocina, lava y atiende. Funciona con 3 clientes, colapsa con 30. Los archivos separados también facilitan encontrar errores: si algo falla al guardar, sabes que el problema está en crear.php, no tienes que leer 500 líneas de código revuelto.

CREATE: formulario más INSERT con sentencia preparada

Crear un registro tiene dos mitades que trabajan juntas: el formulario HTML que captura los datos del usuario, y el código PHP que los recibe y los guarda en la base de datos con INSERT. Son dos momentos distintos de la misma historia. Primero el usuario ve el formulario vacío y lo llena. Después lo envía, y el servidor guarda lo que llegó. En el código, esa diferencia se detecta así: si la petición es POST, hay datos que guardar; si es GET, solo hay que mostrar el formulario.

Vamos con el formulario. Necesitas los campos que corresponden a las columnas de la tabla: nombre, precio y existencia. El id no se pide porque es automático. Un formulario básico se ve así:

```
<form method="post" action="crear.php"> <label>Nombre del producto:</label> <input
type="text" name="nombre" required> <label>Precio:</label> <input type="number"
name="precio" step="0.01" min="0" required> <label>Existencia:</label> <input type="number"
name="existencia" min="0" value="0" required> <button type="submit">Guardar
producto</button> </form>
```

Fíjate en varios detalles. El atributo name de cada input es el nombre con el que PHP recibirá el dato en \$_POST. El atributo required hace que el navegador no deje enviar el formulario vacío, eso es validación del lado del cliente, rápida y cómoda pero no suficiente. El input de precio usa step="0.01" para aceptar centavos y min="0" para no aceptar precios negativos.

Ahora la parte seria: la validación del lado del servidor. Nunca, jamás, confíes en lo que llega del navegador. Un usuario malicioso (o un formulario manipulado) puede enviar el formulario sin pasar por tu HTML, con datos vacíos, textos larguísimos o hasta código malicioso. Así que en PHP, antes de guardar, validas todo: que los campos no vengan vacíos, que el precio sea un número mayor o igual a cero, que la existencia sea un número entero. Ejemplo:

```
$nombre = trim($_POST["nombre"] ?? ""); $precio = $_POST["precio"] ?? ""; $existencia =
$_POST["existencia"] ?? ""; $errores = []; if ($nombre === "") { $errores[] = "El nombre es
obligatorio."; } if (!is_numeric($precio) || $precio < 0) { $errores[] = "El precio debe ser un
número mayor o igual a cero."; } if (!ctype_digit(strval($existencia))) { $errores[] = "La
existencia debe ser un número entero."; }
```

La función trim quita espacios al inicio y al final, para que un nombre de puros espacios no pase como válido. El operador ?? es el "si no existe, usa esto" que ya conoces: evita errores cuando alguien entra directo a la página sin enviar el formulario.

Una vez validados los datos, viene el INSERT. Y aquí hay una regla que no se negocia: usa sentencias preparadas. Ya las viste en la unidad de seguridad, pero vale la pena repetirlo porque en un CRUD las vas a usar en cada operación de escritura. La sentencia preparada separa el SQL de los datos, de modo que aunque el nombre del producto contenga comillas o texto raro, la base de datos lo trata como un simple valor y no como código. Así:

```
$stmt = $conexión->prepare("INSERT INTO productos (nombre, precio, existencia) VALUES (?, ?, ?)");  
$stmt->bind_param("sdi", $nombre, $precio, $existencia);  
$stmt->execute();
```

La "sdi" significa: string, decimal (double), integer. Cada signo de interrogación es un lugar reservado que luego se llena con bind_param. Si el INSERT se ejecuta bien, el producto queda guardado y puedes mostrar un mensaje de éxito. Si falla, muestras el error de forma amable, sin revelar detalles técnicos al usuario final.

Errores comunes al crear. El más típico: olvidar que los nombres de los inputs (name="nombre") deben coincidir exactamente con las llaves que lees en \$_POST. Si el input se llama "nombre" y tu lees \$_POST["Nombre"], PHP te dará un aviso de índice indefinido y el dato llegará vacío. Otro error frecuente: poner el código de guardado después del HTML y luego intentar redirigir con header(), lo cual falla porque ya se envió salida al navegador. Por eso el código de guardado va siempre al inicio del archivo, antes de cualquier HTML, y la redirección se hace ahí mismo. Otro más: confiar solo en required del navegador y no validar en PHP. Recuerda: required es cortesía, la validación en PHP es seguridad.

READ: listar en tabla y ver el detalle de un registro

Leer es la operación más amable del CRUD: no modifica nada, solo muestra. Y sin embargo es la que más diseño requiere, porque es lo que el usuario ve todo el tiempo. Un buen listado es una tabla HTML clara, con una fila por producto y columnas para cada dato, más una columna de acciones con los botones de editar y eliminar.

El flujo es así: PHP hace un SELECT a la tabla, obtiene todos los registros con fetch_assoc en un ciclo while, y por cada registro imprime una fila <tr> de la tabla. Así de directo. Ejemplo de la consulta:

```
$resultado = $conexión->query("SELECT id, nombre, precio, existencia FROM productos  
ORDER BY id DESC");
```

El ORDER BY id DESC pone los más recientes primero, que es lo que el usuario espera: lo último que agregó aparece arriba.

Dentro del ciclo, cada dato se imprime con `htmlspecialchars`. Esta función convierte caracteres especiales de HTML (<, >, &, comillas) en su versión segura, para que si alguien guardó un nombre con código malicioso como `<script>`, el navegador lo muestre como texto y no lo ejecute. Es la defensa contra ataques XSS y es obligatoria en todo dato que venga de la base de datos y se muestre en pantalla. Regla simple: todo lo que imprimas con `echo` y venga de la base o del usuario, pasalo por `htmlspecialchars`.

La columna de acciones lleva dos enlaces. Uno a `editar.php?id=3` (donde 3 es el id del producto de esa fila) y otro que dispara el borrado. El enlace de editar es un simple `<a href>`. El de eliminar, como verás más adelante, es mejor hacerlo con un formulario POST y una confirmación de JavaScript. También es buena idea agregar un botón bien visible al inicio de la página: "Agregar producto", que lleve a `crear.php`. Así el usuario siempre sabe donde empezar.

Qué pasa si no hay productos? Un CRUD bien hecho no muestra una tabla vacía y muda: muestra un mensaje amable como "Aun no hay productos. Agrega el primero con el botón de arriba." Ese detalle marca la diferencia entre un programa hecho con prisa y uno hecho con cariño. Para saber si hay registros, cuentas las filas con `$resultado->num_rows`.

Además del listado, casi todo CRUD necesita la vista de detalle: la ficha completa de un solo registro. Se llega con un enlace como `detalle.php?id=3`. El archivo toma el id de `$_GET`, lo valida como número entero, hace un SELECT con WHERE id = ?, y muestra los datos en grande, con un diseño más legible que la tabla. Si el id no existe en la base de datos, se muestra un mensaje de "producto no encontrado" en vez de una página rota. El detalle es útil cuando un registro tiene muchos campos o cuando quieres mostrar información extra, como la fecha de creación o una descripción larga.

Un consejo práctico sobre el id en la URL. El id viaja visible en la dirección, y cualquier usuario puede cambiarlo a mano: escribir `detalle.php?id=999` aunque ese producto no exista, o `detalle.php?id=abc` con letras. Por eso siempre validas: que sea un número entero positivo, y que el registro exista. Nunca confíes en que "nadie va a tocar la URL". La gente la toca, por curiosidad o por malicia. Validar el id es barato y te evita errores feos y agujeros de seguridad.

Errores comunes al leer. El primero: olvidar `htmlspecialchars` y mostrar los datos "crudos". Funciona perfecto hasta el día que alguien guarda un nombre con < o con comillas y tu tabla se rompe o, peor, se ejecuta código ajeno. El segundo: no manejar el caso de cero registros, dejando una tabla vacía que confunde al usuario. El tercero: hacer el SELECT sin ORDER BY y que los productos aparezcan en un orden aparentemente aleatorio cada vez que se recarga

la página. El cuarto: en el detalle, no verificar que el registro exista y mostrar una página con campos vacíos o errores de PHP.

UPDATE: precargar el formulario y actualizar con WHERE

Actualizar es cómo crear, pero con memoria. El formulario ya no aparece vacío: aparece lleno con los datos actuales del producto, para que el usuario solo cambie lo necesario. Y al guardar, en vez de INSERT se ejecuta un UPDATE con un WHERE que diga exactamente que fila modificar.

El flujo tiene dos pasos. Paso uno: el usuario llega desde el listado con un enlace como `editar.php?id=3`. El archivo toma ese id, lo valida, hace un SELECT para traer los datos del producto 3, y muestra el formulario con cada campo precargado usando el atributo value. Así:

```
<input type="text" name="nombre" value="<?php echo htmlspecialchars($producto["nombre"]);
?>">
```

Fíjate que aquí `htmlspecialchars` es doblemente importante: el valor va dentro de un atributo HTML entre comillas, y si el nombre contiene una comilla doble, podría romper el HTML o inyectar código. Siempre escapa.

Paso dos: el usuario envía el formulario (por POST, igual que en crear) y PHP ejecuta el UPDATE:

```
$stmt = $conexión->prepare("UPDATE productos SET nombre = ?, precio = ?, existencia = ?
WHERE id = ?"); $stmt->bind_param("sdii", $nombre, $precio, $existencia, $id);
$stmt->execute();
```

Ese `WHERE id = ?` es la línea más importante de todo el UPDATE. Sin el WHERE, MySQL actualiza todas las filas de la tabla. Imagina la tiendita: quieres cambiar el precio de la leche de 25 a 27 pesos, olvidas el WHERE, y de pronto todos los productos cuestan 27 pesos. El arroz, los frijoles, el jabón: todo a 27. Esa es la catástrofe del WHERE olvidado, y le ha pasado a programadores profesionales en sistemas reales, con pérdidas de dinero de verdad. Por eso la regla se tatúa en la mente: todo UPDATE lleva su WHERE, y antes de ejecutar un UPDATE en producción, primero pruebas el WHERE en un SELECT para ver cuántas filas afectaría.

Como en `crear.php`, aquí también validas los datos en el servidor antes de guardar: nombre no vacío, precio numérico no negativo, existencia entera. Y el id también se valida: debe ser un número entero positivo y, buena práctica, verificas que el producto exista antes de mostrar el formulario. Si alguien manipula la URL y pone un id que no existe, le muestras "producto no encontrado" en lugar de un formulario vacío que al guardar crearía confusión.

Un detalle elegante: el formulario de editar suele llevar un campo oculto con el id.

```
<input type="hidden" name="id" value="<?php echo $id; ?>">
```

Así, cuando el formulario se envía por POST, el id viaja junto con los demás datos y el código de guardado sabe que fila actualizar. Alternativa válida: leer el id de la URL también en el POST (action="editar.php?id=3"). Ambas funcionan; el campo oculto es más común porque el formulario queda autocontenido.

Errores comunes al actualizar. El campeón indiscutible: olvidar el WHERE. El segundo: precargar el formulario sin htmlspecialchars y que un dato con comillas rompa el HTML. El tercero: validar el id solo como "que exista en \$_GET" sin comprobar que sea número, lo que abre la puerta a inyecciones si no usas sentencia preparada (con sentencia preparada el riesgo baja muchísimo, pero validar de todos modos es lo correcto). El cuarto: confundir los archivos y hacer INSERT en editar.php, duplicando el producto en vez de actualizarlo. Por eso conviene que crear.php y editar.php sean archivos separados con nombres claros, aunque compartan parte del código del formulario.

DELETE: confirmar antes de borrar y usar POST

Borrar es la operación más peligrosa del CRUD porque no hay deshacer fácil. En la libreta de la tiendita, tachar una línea con pluma es definitivo; en la base de datos, un DELETE también lo es (a menos que tengas respaldos). Por eso el borrado se rodea de dos protecciones: una confirmación visible para el usuario y el uso del método POST en vez de GET.

Primero, la confirmación. Antes de borrar, el usuario debe decir explícitamente "sí, estoy seguro". La forma más simple es la función confirm() de JavaScript, que muestra un cuadro del navegador con Aceptar y Cancelar:

```
<form method="post" action="eliminar.php" onsubmit="return confirm('Eliminar este producto? No se puede deshacer.');"> <input type="hidden" name="id" value="<?php echo $p["id"]; ?>"> <button type="submit">Eliminar</button> </form>
```

Si el usuario pulsa Cancelar, el formulario no se envía. Es una protección sencilla contra clics accidentales. En sistemas más avanzados se usa una página intermedia de confirmación ("seguro que quieres eliminar la leche? [Sí, eliminar] [Cancelar]"), que es aun más clara. Para nuestro mini-CRUD, el confirm() es suficiente y muy común en la práctica.

Segundo, y esto es crucial: el borrado se hace con POST, nunca con GET. ¿Por qué? Imagina que eliminar fuera un simple enlace: eliminar.php?id=3. Ese enlace podría ser visitado sin querer de muchas formas: el usuario lo guarda en marcadores y lo abre meses después, un robot de buscador lo rastrea, alguien lo comparte por chat y otra persona le da clic, el

navegador lo precarga. Cada visita borraría el producto, sin confirmación posible. Es como tener un botón rojo de autodestrucción sin tapa protectora, al alcance de cualquiera que pase.

Con POST, en cambio, borrar requiere enviar un formulario, lo que solo ocurre cuando el usuario pulsa el botón a propósito (y confirma en el cuadrado). La regla general del desarrollo web es: GET es para leer, POST es para cambiar cosas. Crear, actualizar y eliminar siempre por POST. Grábatelo: nunca pongas una acción destructiva detrás de un simple enlace.

El código de eliminar.php es corto. Recibe el id por POST, lo valida como entero, ejecuta el DELETE con sentencia preparada y su WHERE (otra vez el WHERE, la misma catástrofe aplica: un DELETE sin WHERE borra toda la tabla), y redirige al listado con un mensaje flash:

```
$stmt = $conexión->prepare("DELETE FROM productos WHERE id = ?");  
$stmt->bind_param("i", $id); $stmt->execute();
```

Buena práctica extra: después de borrar, verifica cuántas filas se afectaron con \$stmt->affected_rows. Si fue 1, todo bien. Si fue 0, el producto ya no existía (quizás alguien más lo borró antes) y puedes mostrar un mensaje acorde en vez de un éxito falso.

Errores comunes al eliminar. El más grave: borrar con un enlace GET. El segundo: olvidar el WHERE en el DELETE y vaciar la tabla completa. El tercero: no pedir confirmación y que un clic accidental borre información. El cuarto: no validar el id, permitiendo que valores raros lleguen a la consulta. El quinto: mostrar detalles técnicos del error de SQL al usuario en vez de un mensaje amable.

Mensajes flash: avisarle al usuario que todo salió bien

Imagina que guardas un producto y la página simplemente recarga el listado sin decir nada. El usuario se queda pensando: se guardó o no? Tuvo que buscar el producto en la tabla para comprobarlo. Esa incertidumbre es mala experiencia de usuario. La solución son los mensajes flash: avisos breves que aparecen una sola vez después de una acción ("Producto guardado correctamente", "Producto eliminado") y desaparecen al recargar.

Se llaman flash porque destellan una vez y se apagan. Se implementan con sesiones, que ya conoces: guardas el mensaje en \$_SESSION antes de redirigir, y en la página de destino lo muestras y lo borras inmediatamente. Así:

```
// En crear.php, después de guardar: session_start(); $_SESSION["flash"] = "Producto  
guardado correctamente."; header("Location: index.php"); exit;
```

```
// En index.php, al inicio del contenido: session_start(); if (isset($_SESSION["flash"])) { echo
"<p      class='aviso'" .      htmlspecialchars($_SESSION["flash"]) .      "</p>";
unset($_SESSION["flash"]); }
```

El unset es lo que lo hace flash: si no borras el mensaje de la sesión, aparecería en todas las páginas siguientes hasta que el usuario cierre el navegador. También puedes guardar el tipo de mensaje (éxito o error) para pintarlo de verde o rojo con CSS. Por ejemplo, `$_SESSION["flash_tipo"] = "éxito"`. Es un detalle pequeño que hace que tu CRUD se sienta profesional.

Nota importante: `session_start()` debe ir al inicio del archivo, antes de cualquier salida HTML, igual que `header()`. Si tu archivo ya imprimió algo (aunque sea un espacio en blanco antes de `<?php`), la sesión y la redirección fallarán. Acostúmbrate a poner el bloque PHP de lógica siempre primero y el HTML después.

POST-Redirect-GET: el patrón que evita duplicados

Este es uno de esos trucos que separan al aficionado del profesional, y es muy fácil de entender con un ejemplo. Sin el patrón, el flujo es: el usuario envía el formulario (POST), el servidor guarda el producto y muestra una página de "guardado con éxito". Hasta aquí todo bien. Pero si el usuario pulsa F5 para recargar, el navegador le pregunta "quieres reenviar el formulario?" y si dice que sí, el producto se guarda otra vez. Y otra vez. Cada F5 crea un duplicado. En la tiendita terminarias con la leche registrada 14 veces.

El patrón POST-Redirect-GET (PRG) lo evita con un paso intermedio. El flujo correcto es: el usuario envía el formulario (POST), el servidor guarda, y en vez de mostrar una página, responde con una redirección (`header("Location: index.php")`). El navegador entonces hace una petición GET normal al listado. Si el usuario pulsa F5 ahora, solo recarga el listado (un GET inofensivo), no reenvía el formulario. Problema resuelto.

El código es de tres líneas, siempre igual, después de guardar con éxito:

```
$_SESSION["flash"] = "Producto guardado correctamente."; header("Location: index.php");
exit;
```

El `exit` después del `header` es obligatorio: detiene la ejecución para que no se siga procesando código ni se imprima nada más. Sin el `exit`, PHP seguiría ejecutando el resto del archivo, lo cual puede causar efectos raros. Y recuerda la condición para que `header()` funcione: no debe haberse enviado ninguna salida antes, ni siquiera un espacio o un salto de línea fuera de las etiquetas PHP. Si ves el famoso error "headers already sent", el culpable casi siempre es un espacio en blanco antes de `<?php` o un `echo` olvidado arriba del archivo.

El patrón PRG se aplica después de crear, actualizar y eliminar. Siempre. Es un hábito que, una vez adquirido, aplicarás en todos tus proyectos sin pensarlo. Y combina perfecto con los mensajes flash: la redirección lleva al usuario al listado, y el mensaje flash le confirma que su acción funcionó. POST guarda, Redirect mueve, GET muestra, Flash avisa. Así de redondo.

Organizar el proyecto en carpetas

Cuando tu CRUD es de 6 archivos, una sola carpeta basta. Pero los proyectos crecen, y conviene conocer la organización que usan los proyectos reales para que tu código escale sin convertirse en un caos. La idea es agrupar los archivos por su papel, como los departamentos de una empresa.

Una estructura típica para un proyecto mediano sería:

- `config/conexión.php`: los datos de conexión a la base de datos. Solo este archivo sabe el usuario y la contraseña.
- `includes/funciones.php`: funciones reutilizables (validar datos, mostrar mensajes flash, escapar texto).
- `public/index.php`, `public/crear.php`, etc.: las páginas que el usuario visita.
- `assets/estilos.css`: la hoja de estilos para que todo se vea bonito.

No necesitas aplicar toda esta estructura en el mini-CRUD de esta unidad. Pero si aplica el principio: separa la configuración (conexión) de la lógica (guardar, borrar) y de la presentación (HTML). Cuando algo falla, sabrás en qué "departamento" buscar. Y cuando el proyecto crezca, mover archivos a carpetas será natural porque ya pensaste en responsabilidades separadas.

Otro hábito sano: ponle nombres descriptivos a todo. `crear.php` es mejor que `alta.php` o que `pagina2.php`. `$producto` es mejor que `$p` o `$x`. El código se lee muchas más veces de las que se escribe, y tu yo del futuro (o tu compañero de trabajo) te lo agradecerá.

Errores comunes del CRUD (y como evitarlos)

Esta sección es oro puro. Cada error de esta lista le ha costado tiempo, dinero o sueños a algún programador. Aprendetelos aquí, en terreno seguro, y no tendrás que aprenderlos a la mala.

- Olvidar el `WHERE` en `UPDATE` o `DELETE`. La catástrofe. Un `UPDATE` sin `WHERE` cambia todas las filas; un `DELETE` sin `WHERE` las borra todas. Prevención: escribe primero el `WHERE` y luego el resto, o prueba la condición en un `SELECT` antes. Algunos

programadores hasta se acostumbran a escribir "WHERE 1=0" como marcador mientras redactan la consulta, para que sea imposible ejecutarla sin condición por accidente.

- Id manipulado en la URL. El usuario cambia ?id=3 por ?id=9999 o por ?id=hola. Prevencion: valida que el id sea un entero positivo con `filter_var($id, FILTER_VALIDATE_INT)` y verifica que el registro exista antes de operar. Si no es válido, muestra un mensaje de error amable y detente.
- No escapar la salida con `htmlspecialchars`. Los datos guardados pueden contener `<`, `>` o comillas que rompen tu HTML o ejecutan scripts ajenos. Prevencion: pasa por `htmlspecialchars` todo dato que imprimas y venga de la base de datos o del usuario. Sin excepciones.
- `header()` después de haber impreso HTML. PHP no puede enviar cabeceras si ya mandó contenido al navegador. Prevencion: pon toda la lógica PHP (validar, guardar, redirigir) al inicio del archivo, antes de cualquier HTML, y cuidado con los espacios en blanco antes de `<?php`.
- Olvidar `exit` después de `header("Location: ...")`. El script sigue ejecutandose y puede guardar dos veces o mostrar contenido extraño. Prevencion: `header` y `exit` siempre van juntos, como tortilla y sal.
- Confiar solo en la validación del navegador. El atributo `required` se salta fácil. Prevencion: valida todo en PHP, siempre. La validación del navegador es comodidad; la de PHP es seguridad.
- Borrar con GET en vez de POST. Un enlace de borrado puede activarse sin querer. Prevencion: el borrado siempre viaja en un formulario POST con confirmación.
- No comprobar si el registro existe antes de editar o borrar. Si el id no existe, el UPDATE o DELETE no hace nada y el usuario cree que funcionó. Prevencion: haz un SELECT primero o revisa `affected_rows` después.
- Mezclar lógica y presentación en un solo archivo gigante. Al principio funciona, pero se vuelve imposible de mantener. Prevencion: un archivo por responsabilidad, como las estaciones de la cocina.
- Mostrar errores técnicos de SQL al usuario. Mensajes como "Duplicate entry" o rutas de archivos le dicen al atacante como esta hecho tu sistema. Prevencion: registra el error técnico en un log y muestra al usuario un mensaje generico y amable.

Buenas prácticas del CRUD

Para cerrar la teoría, una lista de hábitos que distinguen un CRUD bien hecho:

- Valida el id como número antes de usarlo. Un `filter_var` de una línea te ahorra muchos dolores.
- Confirma antes de borrar. Un `confirm()` de JavaScript o una página intermedia de confirmación.
- Usa sentencias preparadas en cada consulta que reciba datos del usuario. Sin excepciones, ni siquiera "es que este dato es seguro".
- Escapa la salida con `htmlspecialchars` en cada dato que muestres.
- Aplica el patrón POST-Redirect-GET después de cada escritura, con su mensaje flash.
- Muestra mensajes claros al usuario: que paso, y que puede hacer ahora. "Producto guardado" es mejor que nada; "Producto 'Leche' guardado. Puedes agregar otro o volver al listado." es aun mejor.
- Maneja el caso de lista vacía con un mensaje amable y un botón de acción.
- Ordena los listados de forma predecible (los más recientes primero, o alfabeticamente).
- Haz respaldos de tu base de datos antes de probar operaciones de borrado masivo. En desarrollo, trabaja siempre sobre datos de prueba que puedas regenerar.

Con la teoría completa, pasemos a la práctica: ocho ejemplos de código que forman el mini-CRUD de productos, explicados línea por línea.

Ejemplos paso a paso

Ejemplo 1: `conexión.php`, la base de todo

Este archivo es el más corto y el más importante: todos los demás lo incluyen con `require`. Si algún día cambias de servidor o de contraseña, solo editas este archivo.

```

<?php
$servidor = "localhost";
$usuario = "root";
$clave = "";
$base = "tiendita";

$conexion = new mysqli($servidor, $usuario, $clave, $base);

if ($conexion->connect_error) {
    die("Error de conexión: " . $conexion->connect_error);
}

$conexion->set_charset("utf8");
?>

```

Línea por línea. Las primeras cuatro líneas guardan los datos de conexión en variables con nombres claros; así, si cambian, se editan en un solo lugar visible. La quinta crea el objeto de conexión mysqli con esos cuatro datos: servidor, usuario, clave y base de datos. Las líneas del if verifican si hubo error de conexión: connect_error contiene el mensaje del error, y die detiene el programa mostrando ese mensaje. Sin conexión no hay CRUD, así que detenerse aquí es lo correcto. La última línea, set_charset("utf8"), le dice a MySQL que hable en UTF-8 para que los acentos y las ñes se guarden y se lean bien. Sin esto, un producto llamado "Pina" podría guardarse con caracteres raros.

Cada archivo del CRUD empezara con require "conexión.php"; para tener la variable \$conexion disponible. Usa require (no include) porque si la conexión falla al cargarse, el programa no debe continuar a medias.

Ejemplo 2: index.php, el listado de productos

Este es el corazon visual del CRUD: la tabla con todos los productos y los botones de acción. También muestra los mensajes flash.

```

<?php
session_start();
require "conexion.php";

$resultado = $conexion->query("SELECT id, nombre, precio, existencia FROM productos
ORDER BY id DESC");
?>
<!DOCTYPE html>
<html lang="es">
<head>

```

```

<meta charset="utf-8">
<title>Productos - Mi tiendita</title>
</head>
<body>
<h1>Productos de la tiendita</h1>

<?php if (isset($_SESSION["flash"])): ?>
<p><?php echo htmlspecialchars($_SESSION["flash"]); ?></p>
<?php unset($_SESSION["flash"]); endif; ?>

<p><a href="crear.php">Agregar producto</a></p>

<?php if ($resultado->num_rows == 0): ?>
<p>Aun no hay productos. Agrega el primero con el boton de arriba.</p>
<?php else: ?>
<table border="1" cellpadding="8">
<tr>
<th>Nombre</th>
<th>Precio</th>
<th>Existencia</th>
<th>Acciones</th>
</tr>
<?php while ($p = $resultado->fetch_assoc()): ?>
<tr>
<td><?php echo htmlspecialchars($p["nombre"]); ?></td>
<td>$<?php echo number_format($p["precio"], 2); ?></td>
<td><?php echo (int)$p["existencia"]; ?></td>
<td>
<a href="detalle.php?id=<?php echo $p["id"]; ?>">Ver</a>
<a href="editar.php?id=<?php echo $p["id"]; ?>">Editar</a>
<form method="post" action="eliminar.php" style="display:inline"
onsubmit="return confirm('Eliminar este producto? No se puede deshacer.');">
<input type="hidden" name="id" value="<?php echo $p["id"]; ?>">
<button type="submit">Eliminar</button>
</form>
</td>
</tr>
<?php endwhile; ?>
</table>
<?php endif; ?>
</body>
</html>

```

Vamos por partes. El bloque PHP inicial abre la sesión (para los mensajes flash) e incluye la conexión. Después ejecuta el SELECT ordenado por id descendente: lo más nuevo primero. Luego empieza el HTML normal.

El bloque del mensaje flash usa la sintaxis alternativa de PHP (if: ... endif;) que es más legible cuando se mezcla con HTML. Muestra el mensaje escapado con htmlspecialchars y lo borra con unset para que solo aparezca una vez. Recuerda: flash que no se borra, no es flash.

El botón "Agregar producto" es un simple enlace a crear.php, bien visible arriba de la tabla. Si no hay productos (num_rows === 0), se muestra el mensaje amable en vez de una tabla vacía. Si si hay, se dibuja la tabla: una fila de encabezados y luego un while que por cada producto imprime una fila.

Dentro de la fila, cada dato se escapa con htmlspecialchars. El precio se formatea con number_format(\$p["precio"], 2) para que siempre muestre dos decimales: \$25.50 en vez de \$25.5. La existencia se convierte a entero con (int) por seguridad. La columna de acciones tiene tres cosas: el enlace Ver a detalle.php?id=, el enlace Editar a editar.php?id=, y el formulario de eliminar por POST con su confirm() de JavaScript. Fíjate que el formulario de eliminar va dentro de la celda, con style="display:inline" para que no rompa el renglon, y lleva el id en un campo oculto.

Ejemplo 3: crear.php, formulario y guardado

Este archivo hace dos trabajos: mostrar el formulario (cuando se llega por GET) y guardar los datos (cuando se recibe un POST). La lógica de guardado va primero, antes del HTML, para poder redirigir.

```
<?php
session_start();
require "conexion.php";

$errores = [];
$nombre = "";
$precio = "";
$existencia = "0";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"] ?? "");
    $precio = $_POST["precio"] ?? "";
    $existencia = $_POST["existencia"] ?? "";

    if ($nombre === "") {
        $errores[] = "El nombre es obligatorio.";
    }
}
```



```

}
if (!is_numeric($precio) || $precio < 0) {
$errores[] = "El precio debe ser un numero mayor o igual a cero.";
}
if (!ctype_digit(strval($existencia))) {
$errores[] = "La existencia debe ser un numero entero no negativo.";
}

if (empty($errores)) {
$stmt = $conexion->prepare("INSERT INTO productos (nombre, precio, existencia) VALUES
(?, ?, ?)");
$stmt->bind_param("sdi", $nombre, $precio, $existencia);
if ($stmt->execute()) {
$_SESSION["flash"] = "Producto guardado correctamente.";
header("Location: index.php");
exit;
} else {
$errores[] = "No se pudo guardar el producto. Intenta de nuevo.";
}
}
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="utf-8">
<title>Agregar producto</title>
</head>
<body>
<h1>Agregar producto</h1>

<?php if (!empty($errores)): ?>
<ul>
<?php foreach ($errores as $e): ?>
<li><?php echo htmlspecialchars($e); ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>

<form method="post" action="crear.php">
<label>Nombre del producto:</label><br>
<input type="text" name="nombre" value="<?php echo htmlspecialchars($nombre); ?>"
required><br><br>
<label>Precio:</label><br>
<input type="number" name="precio" step="0.01" min="0" value="<?php echo

```

```

htmlspecialchars($precio); ?>" required><br><br>
<label>Existencia:</label><br>
<input      type="number"      name="existencia"      min="0"      value="php      echo
htmlspecialchars($existencia); ?" required><br><br>
<button type="submit">Guardar producto</button>
</form>

<p><a href="index.php">Volver al listado</a></p>
</body>
</html>

```

Analicemos. El bloque inicial prepara variables vacías para los campos y un arreglo de errores. La condición `$_SERVER["REQUEST_METHOD"] === "POST"` detecta si el formulario fue enviado: solo entonces se leen y validan los datos. `trim` limpia el nombre; el operador `??` evita avisos si falta algún campo.

Las validaciones: nombre no vacío después de `trim`, precio numérico y no negativo, existencia con `ctype_digit` (que verifica que todos los caracteres sean dígitos, perfecto para enteros no negativos). Cada falló agrega un mensaje al arreglo `$errores`.

Si no hay errores, se prepara el `INSERT` con los tres signos de interrogación y `bind_param("sdi", ...)`: `s` de string para el nombre, `d` de double para el precio decimal, `i` de integer para la existencia. Si `execute()` tiene éxito, se guarda el mensaje flash, se redirige al listado con header y se detiene con `exit`: el patrón POST-Redirect-GET en acción. Si falla, se agrega un mensaje de error genérico (sin detalles técnicos para el usuario).

Si hubo errores de validación, el script no redirige: sigue hacia el HTML, que primero muestra la lista de errores y luego el formulario. Fíjate en un detalle de buena experiencia de usuario: los inputs conservan lo que el usuario ya escribió (`value` con las variables), para que no tenga que reescribir todo por un solo error. Eso se llama "mantener el estado del formulario" y los usuarios lo agradecen muchísimo.

El enlace "Volver al listado" al final es cortesía básica de navegación: el usuario nunca debe sentirse atrapado en una página.

Ejemplo 4: editar.php, precargar y actualizar

Muy parecido a `crear.php`, pero con un paso extra al inicio: leer el producto actual para precargar el formulario. Y al guardar, `UPDATE` con `WHERE` en vez de `INSERT`.

```

<?php
session_start();
require "conexion.php";

```

```

$id = filter_var($_GET["id"] ?? 0, FILTER_VALIDATE_INT);
if (!$id || $id < 1) {
    die("Producto no valido.");
}

$stmt = $conexion->prepare("SELECT nombre, precio, existencia FROM productos WHERE id
= ?");
$stmt->bind_param("i", $id);
$stmt->execute();
$producto = $stmt->get_result()->fetch_assoc();
if (!$producto) {
    die("Producto no encontrado.");
}

$errores = [];
$nombre = $producto["nombre"];
$precio = $producto["precio"];
$existencia = $producto["existencia"];

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $id = filter_var($_POST["id"] ?? 0, FILTER_VALIDATE_INT);
    $nombre = trim($_POST["nombre"] ?? "");
    $precio = $_POST["precio"] ?? "";
    $existencia = $_POST["existencia"] ?? "";

    if (!$id || $id < 1) {
        $errores[] = "Producto no valido.";
    }
    if ($nombre === "") {
        $errores[] = "El nombre es obligatorio.";
    }
    if (!is_numeric($precio) || $precio < 0) {
        $errores[] = "El precio debe ser un numero mayor o igual a cero.";
    }
    if (!ctype_digit(strval($existencia))) {
        $errores[] = "La existencia debe ser un numero entero no negativo.";
    }

    if (empty($errores)) {
        $upd = $conexion->prepare("UPDATE productos SET nombre = ?, precio = ?, existencia = ?
WHERE id = ?");
        $upd->bind_param("sdii", $nombre, $precio, $existencia, $id);
        if ($upd->execute()) {
            $_SESSION["flash"] = "Producto actualizado correctamente.";
        }
    }
}

```

```

header("Location: index.php");
exit;
} else {
$errores[] = "No se pudo actualizar el producto. Intenta de nuevo.";
}
}
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="utf-8">
<title>Editar producto</title>
</head>
<body>
<h1>Editar producto</h1>

<?php if (!empty($errores)): ?>
<ul>
<?php foreach ($errores as $e): ?>
<li><?php echo htmlspecialchars($e); ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>

<form method="post" action="editar.php?id=<?php echo $id; ?>">
<input type="hidden" name="id" value="<?php echo $id; ?>">
<label>Nombre del producto:</label><br>
<input type="text" name="nombre" value="<?php echo htmlspecialchars($nombre); ?>"
required><br><br>
<label>Precio:</label><br>
<input type="number" name="precio" step="0.01" min="0" value="<?php echo
htmlspecialchars($precio); ?>" required><br><br>
<label>Existencia:</label><br>
<input type="number" name="existencia" min="0" value="<?php echo
htmlspecialchars($existencia); ?>" required><br><br>
<button type="submit">Guardar cambios</button>
</form>

<p><a href="index.php">Volver al listado</a></p>
</body>
</html>

```

Línea por línea de lo nuevo. Primero se valida el id que llega por GET con filter_var y FILTER_VALIDATE_INT: si no es un entero valido o es menor que 1, se detiene con un

mensaje. Esto neutraliza el "id manipulado en la URL". Después se hace un SELECT con sentencia preparada para traer los datos actuales; si `fetch_assoc` devuelve falso, el producto no existe y se detiene con "Producto no encontrado".

Las variables del formulario se inicializan con los datos del producto: así el formulario aparece precargado. Si luego llega un POST, las variables se rellenan con lo que el usuario escribió (para no perder sus cambios si hay un error de validación) y se repite la validación igual que en `crear.php`.

El UPDATE lleva sus cuatro parámetros: nombre, precio, existencia y el id para el WHERE, con `bind_param("sdii", ...)`. Observa el orden: los tres valores primero, el id al final, coincidiendo con el orden de los ? en la consulta. Tras guardar, mensaje flash, redirección y exit: el mismo patrón PRG.

El formulario incluye el id de dos formas (campo oculto y en la URL del action) por redundancia segura; el código de guardado lo lee del POST y lo vuelve a validar. Los valores de los inputs usan `htmlspecialchars` porque van dentro de atributos HTML. El botón dice "Guardar cambios" en vez de "Guardar producto": detalles de redacción que orientan al usuario.

Ejemplo 5: eliminar.php, borrado seguro con POST

El archivo más corto del CRUD, pero con las protecciones más importantes: solo acepta POST, valida el id y borra con WHERE.

```

<?php
session_start();
require "conexion.php";

if ($_SERVER["REQUEST_METHOD"] !== "POST") {
$_SESSION["flash"] = "Operacion no permitida.";
header("Location: index.php");
exit;
}

$id = filter_var($_POST["id"] ?? 0, FILTER_VALIDATE_INT);
if (!$id || $id < 1) {
$_SESSION["flash"] = "Producto no valido.";
header("Location: index.php");
exit;
}

$stmt = $conexion->prepare("DELETE FROM productos WHERE id = ?");
$stmt->bind_param("i", $id);
$stmt->execute();

if ($stmt->affected_rows === 1) {
$_SESSION["flash"] = "Producto eliminado correctamente.";
} else {
$_SESSION["flash"] = "El producto ya no existia.";
}

header("Location: index.php");
exit;
?>

```

Análisis. La primera condición es el guardián: si alguien entra a eliminar.php directamente por GET (escribiendo la URL o con un enlace), se le redirige al listado con un mensaje de "Operación no permitida". Así se cumple la regla de nunca borrar con GET: el archivo simplemente se niega a borrar si no viene por POST.

Luego se valida el id del POST igual que antes. El DELETE usa sentencia preparada con su WHERE id = ?. Después de ejecutar, se revisa affected_rows: si fue exactamente 1, el borrado ocurrió; si fue 0, el producto ya no existía (alguien lo borró antes, o el id era inventado) y el mensaje lo refleja con honestidad en vez de mentir con un éxito falso.

Finalmente, mensaje flash y redirección al listado. Nota que este archivo no tiene HTML: su único trabajo es procesar y redirigir. Es un buen ejemplo del principio "cada archivo hace una cosa".

Un consejo de programador experimentado: en sistemas reales, muchas empresas no borran de verdad. En vez de DELETE, agregan una columna como activo (1 o 0) y el "borrado" solo la pone en 0. Así los datos se conservan para historial y se pueden recuperar. Eso se llama borrado lógico. Para nuestro mini-CRUD el DELETE físico está bien, pero ya sabes que existe la alternativa profesional.

Ejemplo 6: detalle.php, la ficha de un producto

La vista de detalle muestra un solo registro con calma y claridad. Útil cuando el registro tiene muchos campos o quieres una presentación más legible que la tabla.

```

<?php
session_start();
require "conexion.php";

$id = filter_var($_GET["id"] ?? 0, FILTER_VALIDATE_INT);
if (!$id || $id < 1) {
    die("Producto no valido.");
}

$stmt = $conexion->prepare("SELECT id, nombre, precio, existencia FROM productos
WHERE id = ?");
$stmt->bind_param("i", $id);
$stmt->execute();
$producto = $stmt->get_result()->fetch_assoc();
if (!$producto) {
    die("Producto no encontrado.");
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="utf-8">
<title>Detalle del producto</title>
</head>
<body>
<h1><?php echo htmlspecialchars($producto["nombre"]); ?></h1>

<p><strong>Precio:</strong>  <?php echo number_format($producto["precio"], 2);
?></p>
<p><strong>Existencia:</strong>  <?php echo (int)$producto["existencia"]; ?>
piezas</p>

<?php if ((int)$producto["existencia"] === 0): ?>
<p>Sin existencias por el momento.</p>
<?php elseif ((int)$producto["existencia"] < 5): ?>
<p>Quedan pocas piezas. Conviene resurtir pronto.</p>
<?php endif; ?>

<p>
<a href="editar.php?id=<?php echo $producto["id"]; ?>">Editar</a>
<a href="index.php">Volver al listado</a>
</p>
</body>
</html>

```


El inicio es idéntico al de `editar.php`: validar el id, buscar el producto, detenerse si no existe. La diferencia está en la presentación: en vez de una tabla compacta, cada dato tiene su renglón con etiqueta clara. El nombre del producto es el título de la página.

Fíjate en el detalle inteligente del final: según la existencia, se muestra un aviso. Si es cero, "Sin existencias"; si es menor de 5, una sugerencia de resurtir. Esto muestra algo importante: la vista de detalle no es solo "los mismos datos en grande", es el lugar donde pones la lógica de presentación que en la tabla estorbaría. En un sistema real aquí pondrías la foto del producto, su descripción larga, su historial de precios o sus ventas del mes.

Los enlaces al final permiten editar directamente o volver al listado. La navegación circular (listado -> detalle -> editar -> listado) es la marca de un CRUD bien pensado: el usuario nunca se pierde.

Nota sobre las etiquetas `<strong>` en este ejemplo: se usan solo para resaltar las etiquetas de cada dato dentro del HTML de presentación, no para dar formato al texto del libro. En tu CSS real las sustituirías por clases con estilo.

Ejemplo 7: mensajes flash mejorados con tipos

El ejemplo 2 mostró el flash básico. Ahora una versión más completa: mensajes de éxito y de error con estilos distintos, usando una pequeña función auxiliar que puedes incluir en todos tus archivos.

```

<?php
// Guardar en includes/flash.php e incluirlo con require en cada pagina.

function flash_set($mensaje, $tipo = "exito") {
$_SESSION["flash"] = $mensaje;
$_SESSION["flash_tipo"] = $tipo;
}

function flash_mostrar() {
if (!isset($_SESSION["flash"])) {
return;
}
$tipo = $_SESSION["flash_tipo"] ?? "exito";
$clase = ($tipo === "error") ? "aviso-error" : "aviso-exito";
$mensaje = htmlspecialchars($_SESSION["flash"]);
unset($_SESSION["flash"]);
unset($_SESSION["flash_tipo"]);
echo "<p class=\"{$clase}\">{$mensaje}</p>";
}
?>

```

Explicación. `flash_set` guarda el mensaje y su tipo ("éxito" o "error") en la sesión. El tipo por defecto es "éxito" gracias al valor por defecto del parámetro. `flash_mostrar` revisa si hay mensaje; si no hay, termina sin hacer nada (return temprano, una técnica muy limpia). Si hay, elige la clase CSS según el tipo con un operador ternario, escapa el mensaje, borra ambas variables de sesión y lo imprime.

El uso en el resto del código queda elegantemente corto. En `crear.php`, después de guardar:

```
flash_set("Producto guardado correctamente.", "éxito");
```

Y si algo falló:

```
flash_set("No se pudo guardar el producto.", "error");
```

En `index.php`, donde antes había cuatro líneas de manejo manual, ahora basta con:

```
flash_mostrar();
```

Las clases CSS `aviso-éxito` y `aviso-error` las defines en tu hoja de estilos: verde suave para éxito, rojo suave para error. Con esto, todos tus mensajes del CRUD se ven consistentes y profesionales, y si algún día quieres cambiar el diseño, lo cambias en un solo lugar: la función.

Este ejemplo también te enseña un principio más grande: cuando repites el mismo bloque de código en varios archivos, conviértelo en una función y ponla en un archivo compartido. Esa es la semilla de las bibliotecas y los frameworks.

Ejemplo 8: versión mejorada con funciones reutilizables

Si miras los ejemplos anteriores, notarás código repetido: validar el id aparece en editar, eliminar y detalle; escapar texto con `htmlspecialchars` aparece docenas de veces; la redirección con flash aparece en crear, editar y eliminar. La versión mejorada centraliza todo eso en un archivo de funciones. Así se ve un CRUD cuando empieza a madurar.

```

<?php
// Archivo: includes/funciones.php

function e($texto) {
return htmlspecialchars($texto ?? "", ENT_QUOTES, "UTF-8");
}

function validar_id($valor) {
$id = filter_var($valor, FILTER_VALIDATE_INT);
if (!$id || $id < 1) {
return 0;
}
return $id;
}

function redirigir_con_mensaje($destino, $mensaje, $tipo = "exito") {
$_SESSION["flash"] = $mensaje;
$_SESSION["flash_tipo"] = $tipo;
header("Location: " . $destino);
exit;
}

function validar_producto($nombre, $precio, $existencia) {
$errores = [];
if (trim($nombre) === "") {
$errores[] = "El nombre es obligatorio.";
}
if (!is_numeric($precio) || $precio < 0) {
$errores[] = "El precio debe ser un numero mayor o igual a cero.";
}
if (!ctype_digit(strval($existencia))) {
$errores[] = "La existencia debe ser un numero entero no negativo.";
}
return $errores;
}
?>

```

Analicemos cada función. `e()` es un atajo para `htmlspecialchars` con los parámetros correctos: `ENT_QUOTES` escapa también las comillas simples (importante dentro de atributos HTML) y `"UTF-8"` fija la codificación. Con esta función, escapar se vuelve tan corto que no hay excusa para omitirlo: `e($p["nombre"])` en vez de la llamada larga. El operador `?? ""` evita avisos si el valor es nulo.

`validar_id()` recibe cualquier valor y devuelve el entero validado o 0 si no sirve. Así, en `editar.php` el inicio se reduce a:

```
$id = validar_id($_GET["id"] ?? 0); if ($id === 0) { die("Producto no valido."); }
```

`redirigir_con_mensaje()` combina el flash y la redirección en una sola llamada. El final de `crear.php` quedaría así:

```
redirigir_con_mensaje("index.php", "Producto guardado correctamente.");
```

Una sola línea en vez de tres, y es imposible olvidar el `exit` porque la función lo incluye. Fíjate como una buena función no solo ahorra escritura: hace imposible cometer ciertos errores.

`validar_producto()` centraliza las reglas de validación que antes estaban duplicadas en `crear.php` y `editar.php`. Devuelve el arreglo de errores (vacío si todo está bien). Si mañana agregas una regla nueva, por ejemplo que el nombre tenga máximo 100 caracteres, la agregas una vez aquí y ambos formularios la heredan.

Con estas funciones, tus archivos del CRUD se vuelven mucho más cortos y legibles. El `index.php` mostraría filas así:

```
<td><?php echo e($p["nombre"]); ?></td>
```

Y el mensaje flash con la función del ejemplo 7. Este es el camino natural de todo programador: primero escribes código repetido para entenderlo, luego lo refactorizas en funciones cuando el patrón se vuelve obvio. No empieces por las funciones abstractas; empieza por el código directo de los ejemplos 1 a 6, y cuando sientas que repites lo mismo por tercera vez, crea la función. Esa es la regla práctica.

Un último consejo de arquitectura: en proyectos grandes, estas funciones crecen hasta convertirse en clases (una clase `Producto` con métodos guardar, actualizar, eliminar, buscar). Eso ya es programación orientada a objetos aplicada, tema de unidades posteriores. Lo importante es que el camino es gradual: código directo, luego funciones, luego clases. Cada nivel organiza mejor el anterior.

Práctica

Ejercicio 1

Construye un CRUD completo de alumnos para una escuela. La tabla se llamara alumnos con columnas: id (automatico), nombre (texto), edad (entero) y promedio (decimal). Debe tener los 6 archivos: conexión, listado, crear, editar, eliminar y detalle. Aplica todo lo aprendido:

sentencias preparadas, validación en servidor, htmlspecialchars en la salida, confirmación al borrar con POST, mensajes flash y redirección POST-Redirect-GET.

Solución 1

La estructura es idéntica a la del CRUD de productos; solo cambian los campos y sus validaciones. Aquí va el código completo.

Primero la tabla:

```
CREATE TABLE alumnos (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  nombre VARCHAR(100) NOT NULL,  
  edad INT NOT NULL,  
  promedio DECIMAL(4,2) NOT NULL  
);
```

El archivo de validación adaptado (puedes ponerlo en includes/funciones.php):

```
<?php  
function validar_alumno($nombre, $edad, $promedio) {  
  $errores = [];  
  if (trim($nombre) === "") {  
    $errores[] = "El nombre es obligatorio.";  
  }  
  if (!ctype_digit(strval($edad)) || (int)$edad < 5 || (int)$edad > 100) {  
    $errores[] = "La edad debe ser un número entero entre 5 y 100.";  
  }  
  if (!is_numeric($promedio) || $promedio < 0 || $promedio > 10) {  
    $errores[] = "El promedio debe ser un número entre 0 y 10.";  
  }  
  return $errores;  
}
```

El crear.php adaptado (la lógica de guardado; el formulario sigue el mismo patrón):

```

<?php
session_start();
require "conexion.php";
require "includes/funciones.php";

$errores = [];
$nombre = "";
$edad = "";
$promedio = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"] ?? "");
    $edad = $_POST["edad"] ?? "";
    $promedio = $_POST["promedio"] ?? "";

    $errores = validar_alumno($nombre, $edad, $promedio);

    if (empty($errores)) {
        $stmt = $conexion->prepare("INSERT INTO alumnos (nombre, edad, promedio) VALUES (?, ?, ?)");
        $stmt->bind_param("sid", $nombre, $edad, $promedio);
        if ($stmt->execute()) {
            redirigir_con_mensaje("index.php", "Alumno registrado correctamente.");
        } else {
            $errores[] = "No se pudo registrar el alumno. Intenta de nuevo.";
        }
    }
}
}
?>

```

Explicación paso a paso. La tabla alumnos replica la estructura de productos con campos propios de una escuela. La función `validar_alumno` encapsula las reglas del negocio escolar: edad entera entre 5 y 100 (rango razonable para un alumno), promedio decimal entre 0 y 10 (escala mexicana de calificaciones). El `bind_param` usa "sid": string para el nombre, integer para la edad, double para el promedio.

El resto de los archivos (`index.php`, `editar.php`, `eliminar.php`, `detalle.php`) se adaptan igual: cambia los nombres de tabla y columnas, ajusta las etiquetas del formulario ("Edad", "Promedio") y los tipos de input. El `editar.php` usa `UPDATE alumnos SET nombre = ?, edad = ?, promedio = ? WHERE id = ?`. El `eliminar.php` usa `DELETE FROM alumnos WHERE id = ?` con POST y confirmación. La lección del ejercicio: una vez que dominas el patrón CRUD, cambiar de "productos" a "alumnos" es mecánico. El 90 por ciento del código se conserva; solo cambian los datos.

Ejercicio 2

Construye un CRUD de libros para una biblioteca. Tabla libros: id automatico, título (texto), autor (texto) y año (entero de 4 dígitos). Incluye listado, crear, editar, eliminar y detalle. Agrega una validación extra: el año no puede ser mayor que el año actual.

Solución 2

Tabla:

```
CREATE TABLE libros (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  titulo VARCHAR(150) NOT NULL,  
  autor VARCHAR(100) NOT NULL,  
  año INT NOT NULL  
);
```

Función de validación con la regla del año actual:

```
<?php  
function validar_libro($titulo, $autor, $año) {  
  $errores = [];  
  if (trim($titulo) === "") {  
    $errores[] = "El título es obligatorio.";  
  }  
  if (trim($autor) === "") {  
    $errores[] = "El autor es obligatorio.";  
  }  
  $año_actual = (int)date("Y");  
  if (!ctype_digit(strval($año)) || strlen(strval($año)) !== 4) {  
    $errores[] = "El año debe tener 4 dígitos.";  
  } elseif ((int)$año > $año_actual) {  
    $errores[] = "El año no puede ser mayor que $año_actual.";  
  } elseif ((int)$año < 1000) {  
    $errores[] = "El año no parece válido.";  
  }  
  return $errores;  
}
```

Listado con columna extra de antigüedad (ejemplo de lógica de presentación en el READ):


```

<?php
session_start();
require "conexion.php";
$resultado = $conexion->query("SELECT id, titulo, autor, anio FROM libros ORDER BY
titulo ASC");
?>
<!-- dentro del while, por cada libro $l: -->
<tr>
<td><?php echo e($l["titulo"]); ?></td>
<td><?php echo e($l["autor"]); ?></td>
<td><?php echo (int)$l["anio"]; ?></td>
<td><?php echo ((int)date("Y") - (int)$l["anio"]); ?> años</td>
<td>
<a href="detalle.php?id=<?php echo $l["id"]; ?>">Ver</a>
<a href="editar.php?id=<?php echo $l["id"]; ?>">Editar</a>
</td>
</tr>

```

Explicación paso a paso. La validación del anio usa date("Y") para obtener el anio actual del servidor, así la regla se mantiene sola con el paso del tiempo: en 2027 seguira funcionando sin editar código. Se verifica que sean exactamente 4 digitos con strlen, que no sea futuro y que no sea absurdamente antiguo (menor a 1000).

El listado ordena por título ASC (alfabetico) en vez de por id: en una biblioteca el orden alfabetico es más útil que el orden de registro. Eso ensena que el ORDER BY se elige pensando en el usuario, no por costumbre. La columna de antigüedad calcula los años del libro restando: es lógica de presentación que vive en la vista, no en la base de datos. No guardes la antigüedad en la tabla porque cambiaria cada año; calculala al mostrar.

El crear, editar y eliminar siguen el patrón ya conocido con sentencias preparadas: INSERT INTO libros (título, autor, anio), UPDATE libros SET título = ?, autor = ?, anio = ? WHERE id = ?, DELETE FROM libros WHERE id = ?.

Ejercicio 3

Agrega un buscador al CRUD de productos. En la parte superior del index.php, un formulario con un campo de texto "Buscar" que filtre los productos por nombre. El buscador debe funcionar con GET (porque es una lectura, no modifica nada), debe usar sentencia preparada con LIKE, y debe mostrar cuantos resultados encontró. Si no se busca nada, muestra todos los productos como antes.

Solución 3

Formulario del buscador (arriba de la tabla en index.php):

```
<form method="get" action="index.php">
<input type="text" name="q" value="<?php echo e($_GET["q"]) ?? ""; ?>"
placeholder="Buscar producto...">
<button type="submit">Buscar</button>
<?php if (isset($_GET["q"]) && $_GET["q"] !== ""): ?>
<a href="index.php">Limpiar</a>
<?php endif; ?>
</form>
```

La consulta adaptada en el PHP inicial:

```
<?php
session_start();
require "conexion.php";
require "includes/funciones.php";

$busqueda = trim($_GET["q"] ?? "");

if ($busqueda === "") {
$resultado = $conexion->query("SELECT id, nombre, precio, existencia FROM productos
ORDER BY id DESC");
$total = $resultado->num_rows;
} else {
$stmt = $conexion->prepare("SELECT id, nombre, precio, existencia FROM productos
WHERE nombre LIKE ? ORDER BY id DESC");
$params = "%" . $busqueda . "%";
$stmt->bind_param("s", $params);
$stmt->execute();
$resultado = $stmt->get_result();
$total = $resultado->num_rows;
}
?>
<p>Se encontraron <?php echo $total; ?> producto(s).</p>
```

Explicación paso a paso. El buscador usa GET a propósito: buscar no modifica datos, y con GET la búsqueda queda en la URL (index.php?q=leche), lo que permite compartirla, guardarla en marcadores y usar el botón atrás del navegador sin advertencias. Esa es la regla: GET para leer, POST para cambiar.

La sentencia preparada con LIKE necesita los signos % alrededor del término, y esos se agregan en PHP al construir \$param, no dentro del SQL. Así el usuario puede escribir "leche" y encuentra "Leche entera", "Leche deslactosada", etc. Nunca concatenes directamente \$_GET dentro del SQL: aunque sea una lectura, un LIKE inyectado también es peligroso.

El campo de búsqueda conserva lo que el usuario escribió (value con e(\$_GET["q"])) para que vea que buscó. El enlace "Limpiar" solo aparece cuando hay una búsqueda activa y lleva al index sin parámetros, mostrando todo de nuevo. El contador de resultados con num_rows le dice al usuario cuantos coincidieron; si es cero, el mensaje de "no hay productos" del listado normal se encarga del resto, aunque conviene personalizarlo: "No se encontraron productos con 'xyz'".

Detalle importante: el mensaje flash y el buscador conviven sin problema porque usan mecanismos distintos (sesión vs URL). Y la tabla, los botones de editar y eliminar no cambian nada: el buscador solo filtra que filas llegan al while.

Ejercicio 4

Agrega un campo de foto al CRUD de productos. Cada producto podrá tener una imagen. Necesitaras: agregar la columna foto (VARCHAR con el nombre del archivo) a la tabla, poner enctype="multipart/form-data" en los formularios de crear y editar, procesar la subida con \$_FILES validando que sea una imagen JPG o PNG de máximo 2 MB, guardar el archivo en una carpeta "fotos/" con un nombre único, y mostrar la miniatura en el listado y en el detalle.

Solución 4

Primero, la columna nueva:

```
ALTER TABLE productos ADD COLUMN foto VARCHAR(255) DEFAULT NULL;
```

Formulario de crear.php (solo cambian estas dos líneas respecto al original):

```
<form method="post" action="crear.php" enctype="multipart/form-data">
<!-- ... los demas campos ... -->
<label>Foto del producto (JPG o PNG, maximo 2 MB):</label><br>
<input type="file" name="foto" accept="image/jpeg,image/png"><br><br>
```

Procesamiento de la subida (función reutilizable para crear y editar):

```

<?php
function subir_foto($campo = "foto") {
if (!isset($_FILES[$campo]) || $_FILES[$campo]["error"] === UPLOAD_ERR_NO_FILE) {
return [null, null];
}
if ($_FILES[$campo]["error"] !== UPLOAD_ERR_OK) {
return [null, "Error al subir la foto."];
}
$permitidos = ["image/jpeg" => "jpg", "image/png" => "png"];
$tipo = $_FILES[$campo]["type"];
if (!isset($permitidos[$tipo])) {
return [null, "La foto debe ser JPG o PNG."];
}
if ($_FILES[$campo]["size"] > 2 * 1024 * 1024) {
return [null, "La foto no debe pesar mas de 2 MB."];
}
$extension = $permitidos[$tipo];
$nombre_unico = "prod_" . time() . "_" . bin2hex(random_bytes(4)) . "." .
$extension;
$destino = "fotos/" . $nombre_unico;
if (!move_uploaded_file($_FILES[$campo]["tmp_name"], $destino)) {
return [null, "No se pudo guardar la foto."];
}
return [$nombre_unico, null];
}
?>

```

Uso en crear.php después de validar los demás campos:

```

<?php
list($foto, $error_foto) = subir_foto();
if ($error_foto) {
$errores[] = $error_foto;
}

if (empty($errores)) {
$stmt = $conexion->prepare("INSERT INTO productos (nombre, precio, existencia,
foto) VALUES (?, ?, ?, ?)");
$stmt->bind_param("sdis", $nombre, $precio, $existencia, $foto);
if ($stmt->execute()) {
redirigir_con_mensaje("index.php", "Producto guardado correctamente.");
}
}
?>

```

Mostrar la miniatura en el listado (dentro del while):

```
<?php if (!empty($p["foto"])): ?>
" width="60" alt="Foto">
<?php else: ?>
<span>Sin foto</span>
<?php endif; ?>
```

Explicación paso a paso. El enctype="multipart/form-data" es obligatorio: sin el, el archivo nunca llega al servidor y \$_FILES queda vacío. Es uno de los errores más comunes y más difíciles de detectar, porque el formulario "funciona" pero la foto no se sube.

La función subir_foto devuelve un arreglo de dos elementos: el nombre del archivo guardado (o null) y un mensaje de error (o null). Así el código que la usa decide que hacer. Las validaciones son en capas: primero que el archivo exista (si no se subió foto, no es error: la foto es opcional), luego que no haya error de subida, luego el tipo MIME contra una lista permitida, luego el tamaño máximo de 2 MB.

El nombre único combina un prefijo, la hora actual y 4 bytes aleatorios: así dos fotos subidas en el mismo segundo no se pisan. Nunca guardes el archivo con el nombre original que puso el usuario: podría contener caracteres raros, espacios o hasta rutas maliciosas. move_uploaded_file es la única forma segura de mover el archivo temporal a su destino.

Nota de seguridad importante: la carpeta fotos/ debe estar fuera del alcance de ejecución de PHP si es posible, o al menos validar el tipo real del archivo. La validación por tipo MIME es básica; en producción se verificaría también la extensión y se impediría la ejecución de scripts en esa carpeta. Para nuestro mini-CRUD, la lista de tipos permitidos más el nombre generado son una base sólida.

En editar.php, la lógica extra es: si se sube una foto nueva, borrar la anterior con unlink("fotos/" . \$foto_anterior) para no acumular archivos huérfanos, y actualizar la columna foto. Si no se sube foto nueva, se conserva la anterior. En eliminar.php, también conviene borrar el archivo físico de la foto al eliminar el producto, para no dejar basura en el servidor.

Ejercicio 5

Agrega paginación al listado de productos. En vez de mostrar todos los productos de golpe, muestra 5 por página, con enlaces de "Anterior", números de página y "Siguiente". La página actual viaja por GET (?página=2). Debes calcular el total de páginas, validar que el número de página sea válido, y usar LIMIT con OFFSET en la consulta.

Solución 5

Código del inicio de index.php adaptado:

```
<?php
session_start();
require "conexion.php";
require "includes/funciones.php";

$por_pagina = 5;
$pagina = validar_id($_GET["pagina"] ?? 1);
if ($pagina === 0) {
    $pagina = 1;
}

$total_registros = (int)$conexion->query("SELECT COUNT(*) AS total FROM
productos")->fetch_assoc()["total"];
$total_paginas = max(1, (int)ceil($total_registros / $por_pagina));

if ($pagina > $total_paginas) {
    $pagina = $total_paginas;
}

$offset = ($pagina - 1) * $por_pagina;

$stmt = $conexion->prepare("SELECT id, nombre, precio, existencia FROM productos
ORDER BY id DESC LIMIT ? OFFSET ?");
$stmt->bind_param("ii", $por_pagina, $offset);
$stmt->execute();
$resultado = $stmt->get_result();
?>
```

Enlaces de paginación (debajo de la tabla):

```

<p>
<?php if ($pagina > 1): ?>
<a href="index.php?pagina=<?php echo $pagina - 1; ?>">Anterior</a>
<?php endif; ?>

<?php for ($i = 1; $i <= $total_paginas; $i++): ?>
<?php if ($i === $pagina): ?>
<strong><?php echo $i; ?></strong>
<?php else: ?>
<a href="index.php?pagina=<?php echo $i; ?>"><?php echo $i; ?></a>
<?php endif; ?>
<?php endfor; ?>

<?php if ($pagina < $total_paginas): ?>
<a href="index.php?pagina=<?php echo $pagina + 1; ?>">Siguiente</a>
<?php endif; ?>
</p>
<p>Pagina <?php echo $pagina; ?> de <?php echo $total_paginas; ?> (<?php echo $total_registros; ?> productos en total).</p>

```

Explicación paso a paso. La paginación resuelve un problema real: si la tiendita tiene 2,000 productos, cargarlos todos de golpe es lento y la página se vuelve imposible de navegar. Mostrar 5 (o 10, o 20) por página es el estándar.

Primero se valida la página con la función `validar_id` que ya tenemos: si alguien escribe `?pagina=hola` o `?pagina=-3`, se corrige a 1. Luego se cuenta el total de registros con `COUNT(*)`, que es rapidísimo porque no trae los datos, solo el número. Con `ceil()` se calcula cuantas páginas se necesitan: `ceil` redondea hacia arriba, porque 12 productos a 5 por página necesitan 3 páginas (la última con 2). `max(1, ...)` garantiza que haya al menos una página aunque la tabla este vacía, para no dividir entre cero ni mostrar "página 1 de 0".

Si la página pedida es mayor que el total (por ejemplo `?pagina=99` con solo 3 páginas), se ajusta a la última: así nunca se muestra una página vacía por un número inventado. El `OFFSET` se calcula como `(página - 1) * por_pagina`: página 1 salta 0, página 2 salta 5, página 3 salta 10. `LIMIT ? OFFSET ?` traen solo el pedazo correspondiente, con sentencia preparada y parámetros enteros.

Los enlaces usan `GET`, como debe ser: la página es una lectura. "Anterior" solo aparece si no estas en la primera; "Siguiente" solo si no estas en la última. La página actual se muestra como texto (no como enlace) para que el usuario sepa donde esta. El `for` genera un enlace por página; con muchas páginas (50 o más) convendría mostrar solo algunas cercanas a la actual, pero para este ejercicio el rango completo está bien.

Combinacion con el buscador del ejercicio 3: si implementas ambos, los enlaces de paginación deben conservar el término de búsqueda (`index.php?q=leche&página=2`) y el conteo total debe contar solo los que coinciden con la búsqueda. Es un excelente reto extra para practicar.

Resumen de la unidad

- CRUD significa Crear, Leer, Actualizar y Eliminar: las cuatro operaciones básicas sobre los datos, y el corazón del 90 por ciento de las aplicaciones web.
- La analogía de la libreta de la tiendita resume el CRUD: anotar (crear), consultar (leer), corregir (actualizar) y tachar (eliminar).
- Un CRUD se organiza en archivos con responsabilidades claras, como las estaciones de una cocina: `conexión.php`, `index.php` (listar), `crear.php`, `editar.php`, `eliminar.php` y `detalle.php`.
- CREATE combina un formulario HTML (method POST) con validación en el servidor y un INSERT con sentencia preparada. Nunca confíes solo en la validación del navegador.
- READ muestra los datos en una tabla HTML con `htmlspecialchars` en cada valor, botones de editar y eliminar, manejo del caso "sin registros" y una vista de detalle por id validado.
- UPDATE precarga el formulario con los datos actuales y ejecuta un UPDATE siempre con `WHERE id = ?`. Olvidar el WHERE actualiza toda la tabla: la catástrofe clásica.
- DELETE se hace siempre con POST y confirmación (`confirm()` de JavaScript o página intermedia), nunca con un enlace GET. Olvidar el WHERE borra toda la tabla.
- Los mensajes flash usan la sesión para mostrar un aviso una sola vez después de una acción; se guardan antes de redirigir y se borran al mostrarse.
- El patrón POST-Redirect-GET evita registros duplicados al recargar: después de guardar, `header("Location: ...")` seguido de `exit`, sin haber enviado salida antes.
- Valida el id de la URL como entero positivo y verifica que el registro exista; nunca confíes en que nadie manipulara la dirección.
- Escapa toda salida con `htmlspecialchars`, usa sentencias preparadas en cada consulta con datos del usuario y no muestres errores técnicos de SQL al usuario final.
- Centraliza el código repetido en funciones (escapar, validar id, redirigir con mensaje, validar datos): primero código directo, luego funciones cuando el patrón se repita.
- Extensiones naturales del CRUD: buscador con LIKE por GET, subida de fotos con `$_FILES` validado, y paginación con LIMIT y OFFSET.

UNIDAD 9: SUBIDA DE ARCHIVOS Y SEGURIDAD

Introducción

Imagina que vives en una casa y todos los días llega el repartidor con paquetes. Algunos los pediste tú: libros, ropa, una licuadora. Otros llegan de sorpresa: un vecino te manda tamales, un primo te regresa una herramienta. Tú no abres la puerta y metes todo a la sala sin mirar. Primero revisas: quien lo manda, que pesa, que suena adentro, si la caja viene rota. Si algo huele raro, no lo metes a tu casa. Así de simple.

Un sitio web que permite subir archivos es exactamente esa casa. Cada vez que un visitante sube una foto de perfil, un comprobante de pago o un documento, está metiendo un paquete a tu servidor. Y aquí viene lo importante: en internet no solo llegan paquetes buenos. También llegan paquetes trampa. Alguien puede disfrazar un programa malicioso con nombre de foto, alguien puede intentar subir un archivo gigante para tumbar tu servidor, alguien puede poner código peligroso donde tú solo esperabas un nombre.

Por eso en esta unidad la subida de archivos y la seguridad van de la mano. No se pueden separar. Aprender a recibir archivos sin aprender a revisarlos es como poner una puerta sin chapa: tarde o temprano alguien entra sin permiso.

En las clases 25, 26 y 27 vas a aprender todo el ciclo completo. Primero, como construir un formulario que acepte archivos y como PHP los recibe en el arreglo `$_FILES`. Vas a conocer cada campo de ese arreglo como si fuera la etiqueta de un paquete: el nombre original, el archivo temporal, el peso, el código de error y el tipo declarado. Después aprenderás a mover el archivo a su lugar definitivo con `move_uploaded_file()`, que es como llevar el paquete del recibidor al cuarto donde debe guardarse.

Luego viene la parte de detective: las validaciones. Vas a aprender a poner un límite de tamaño para que nadie te suba un elefante disfrazado de foto, a usar listas blancas de extensiones para aceptar solo lo que tú quieres, y a comprobar con `getimagesize()` que una imagen sea realmente una imagen y no un lobo con piel de oveja. También aprenderás a renombrar los archivos con `uniqid()`, que es como ponerle un número de folio único a cada paquete que entra a tu bodega: así evitas que dos archivos se llamen igual y se pisen, y evitas que un atacante adivine rutas.

Después conectaremos la subida con la base de datos: no se guarda el archivo dentro de la base, se guarda su ruta, como quien guarda en una libreta en que pasillo de la bodega quedó cada caja.

Y entonces entra la seguridad, que es el corazon de esta unidad. Vas a entender qué es un ataque XSS con un ejemplo clarísimo: un comentario con una etiqueta `<script>` que roba datos. Y vas a aprender la defensa: `htmlspecialchars()`, que convierte el código peligroso en texto inofensivo. Repasaremos la inyección SQL y las sentencias preparadas, porque un formulario de subida suele vivir al lado de un formulario de registro o de login, y un descuido ahí es una puerta abierta.

Hablaremos de la regla de oro: nunca confíes en el usuario. Ni en lo que escribe, ni en lo que sube, ni en lo que dice que es su archivo. Todo se revisa. Todo se valida. Usaremos la analogía del cadenero del antro: el no deja pasar a nadie solo porque trae traje; revisa identificación, revisa la lista, y si algo no cuadra, no entra.

En la parte de contraseñas aprenderas por qué guardarlas en texto plano es una de las peores ideas de la historia de la programación. Te contare la historia de las filtraciones: cuando una base de datos se filtra (y pasa más seguido de lo que crees), si las contraseñas están en texto plano el atacante ya ganó. En cambio, si usas `password_hash()`, lo que se filtra es inservible. Aprenderas cómo funciona `PASSWORD_DEFAULT`, que es como una caja fuerte que se mejora sola con el tiempo, y como verificar al usuario en el login con `password_verify()`. Armaremos juntos un sistema de registro y login seguro, paso a paso, con sesiones.

También verás qué es el ataque CSRF explicado de forma simple: alguien que te hace hacer clic en algo que ejecuta una acción en otro sitio donde ya iniciaste sesión. Y cerraremos con los errores más comunes que cometen los principiantes (olvidar el `enctype`, confiar en el tipo MIME que manda el navegador, guardar archivos con su nombre original, comparar contraseñas a mano) y con buenas prácticas profesionales: carpetas de subida protegidas, límites de tamaño y mensajes de error genéricos en el login para no darle pistas al atacante.

Al terminar esta unidad no solo sabras subir archivos: sabras recibir paquetes en tu casa como un profesional, revisando cada uno antes de meterlo. Esa es la diferencia entre un sitio que funciona y un sitio que funciona y es seguro. Vamos a ello.

Explicación detallada

El formulario de subida: enctype="multipart/form-data"

Un formulario normal envía texto: nombres, correos, mensajes. Todo eso viaja como texto plano dentro de la petición. Pero un archivo no es texto: es una secuencia de bytes que puede pesar megabytes. Para decirle al navegador "oye, este formulario va a llevar archivos, empacalos de forma especial", se usa el atributo `enctype="multipart/form-data"`.

La palabra "multipart" significa "en varias partes": el navegador divide el envío en partes, una por cada campo, y los archivos viajan como datos binarios dentro de su propia parte. Sin este atributo, el campo de archivo llega vacío al servidor y `$_FILES` queda vacío. Es el error número uno de los principiantes: el formulario se ve bien, el botón de "examinar" funciona, pero al enviar, PHP no recibe nada. Si tu subida no funciona, lo primero que revisas es el `enctype`.

El campo para elegir el archivo es `<input type="file" name="foto">`. El atributo `name` es el que usaras en PHP: `$_FILES["foto"]`. Puedes permitir varios archivos con el atributo múltiple y un `name` con corchetes, pero en esta unidad nos concentramos en un archivo a la vez para dominar bien la base.

Un detalle importante: el formulario debe usar `method="post"`. Con GET no se pueden enviar archivos, porque GET mete los datos en la URL y una foto no cabe en una URL.

Analogía: el `enctype` es como decirle a la paquetería "este envío lleva cosas frágiles, usen caja especial". Si no lo dices, te lo mandan en un sobre normal y llega roto, o de plano no llega.

El arreglo `$_FILES` explicado campo por campo

Cuando el formulario llega al servidor, PHP guarda la información del archivo en el arreglo superglobal `$_FILES`. Si tu input se llama "foto", la información vive en `$_FILES["foto"]`, que a su vez es un arreglo con cinco campos. Piensa en ellos como la etiqueta pegada al paquete:

- `name`: el nombre original del archivo en la computadora del usuario. Ejemplo: "mis-vacaciones.jpg". Ojo: este nombre lo pone el usuario, así que no es confiable. Puede traer caracteres raros, espacios, acentos, o incluso intentos de ataque como `"../../../../truco.php"`. Nunca lo uses tal cual para guardar el archivo.
- `tmp_name`: la ruta temporal donde PHP guardó el archivo al recibirlo. Ejemplo: `"/tmp/phpA1b2C3"`. Este es el archivo real, el que vas a mover a su destino final. Si no lo mueves antes de que termine el script, PHP lo borra automáticamente. Es como el paquete que el repartidor deja en el recibidor: si no lo recoges rápido, se lo llevan de vuelta.

- `size`: el peso del archivo en bytes. Ejemplo: 204800 significa 200 KB. Lo usas para validar que no te suban archivos gigantescos.
- `error`: un número que indica si hubo problema en la subida. El 0 significa "todo bien". Los demás son errores: 1 significa que el archivo supera el límite `upload_max_filesize` del servidor; 2 que supera el límite `MAX_FILE_SIZE` del formulario; 3 que llegó incompleto; 4 que no se subió ningún archivo; 6 que falta la carpeta temporal; 7 que no se pudo escribir en disco; 8 que una extensión de PHP detuvo la subida. Siempre revisa este campo antes de hacer cualquier otra cosa.
- `type`: el tipo MIME que declara el navegador, por ejemplo "image/jpeg". Suena útil, pero es mentira piadosa: lo manda el navegador del usuario, y el usuario puede mandar lo que quiera. Un atacante puede subir un programa malicioso diciendo que es "image/jpeg". Regla de oro: nunca confíes en `$_FILES["foto"]["type"]` para validar. Más adelante verás como verificar el tipo de verdad.

Para ver todo esto con tus propios ojos, puedes hacer `var_dump($_FILES)` en tu script de prueba. Ver la estructura real una vez vale más que leer diez explicaciones.

`move_uploaded_file()` paso a paso

Ya recibiste el paquete en el recibidor (la carpeta temporal). Ahora hay que llevarlo a la bodega (tu carpeta de uploads). Para eso existe `move_uploaded_file()`, una función hecha exactamente para esto y con una característica de seguridad extra: solo mueve archivos que realmente fueron subidos por HTTP. Si alguien intenta enganarla pasándole la ruta de un archivo del sistema como `"/etc/passwd"`, la función se niega. Por eso siempre debes usar esta función y nunca `copy()` o `rename()` para archivos subidos.

Recibe dos parámetros: el origen (el `tmp_name`) y el destino (la ruta completa donde quieres guardarlo, incluyendo el nombre final que tu decidiste). Devuelve `true` si todo salió bien y `false` si falló.

El paso a paso en tu script es así:

1. Revisas que no haya error: `$_FILES["foto"]["error"]` debe ser 0.
2. Validas el archivo (tamaño, extensión, contenido real). Esto lo veremos a fondo en las siguientes secciones.
3. Decides el nombre final del archivo. No uses el nombre original; genera uno único.
4. Armas la ruta destino, por ejemplo `"uploads/img_64f2a1b3c8d9e.jpg"`.
5. Llamas a `move_uploaded_file($_FILES["foto"]["tmp_name"], $destino)`.

6. Revisas el valor de retorno. Si es false, muestras un mensaje de error generico.

Si la carpeta "uploads" no existe, la función falla. Así que antes, en tu instalación o en el script, asegurate de crearla. Y dale los permisos correctos: que PHP pueda escribir en ella, pero sin permisos de ejecución innecesarios.

Analogía: `move_uploaded_file` es el montacargas de tu bodega. Solo mueve cajas que llegaron por la puerta de recepción (subidas HTTP reales) y las coloca en el pasillo que tu le indiques. No mueve cajas que ya estaban adentro de la bodega, y eso te protege.

Validar el tamaño máximo

Sin límite de tamaño, un atacante puede subirte un archivo de 2 GB y llenar tu disco duro, o subir muchos archivos grandes hasta tumbar el servidor. Esto se llama ataque de denegación de servicio por agotamiento de recursos, pero tú puedes pensarlo así: si no pones límite de peso en tu recibidor, alguien te manda un piano y bloquea la puerta.

Hay dos niveles de límite. El primero lo pone el servidor en el archivo `php.ini` con `upload_max_filesize` y `post_max_size`. El segundo lo pones tu en tu código, comparando `$_FILES["foto"]["size"]` con tu límite. Por ejemplo, si solo aceptas fotos de perfil, 2 MB es más que suficiente: $2 * 102 * 1024 = 2097152$ bytes.

Importante: valida el tamaño ANTES de mover el archivo. No tiene sentido guardar algo para después darte cuenta de que pesa demasiado. El orden correcto es: revisar error, revisar tamaño, revisar extensión, revisar contenido real, y hasta el final mover.

Si el archivo supera el límite del servidor (`upload_max_filesize`), PHP ni siquiera lo guarda completo y `$_FILES["foto"]["error"]` valdrá 1. Por eso conviene revisar el código de error primero: si vale 1 o 2, ya sabes que el problema fue el tamaño.

Lista blanca de extensiones permitidas

Hay dos formas de filtrar: lista negra ("prohibo .php, .exe, .sh...") y lista blanca ("solo permito .jpg, .jpeg, .png, .gif"). La lista negra es un juego de nunca acabar: siempre habrá una extensión rara que se te olvide prohibir (.phtml, .phar, .htaccess...). La lista blanca es la forma profesional: defines exactamente lo que aceptas y todo lo demás se rechaza.

Para obtener la extensión del nombre original usas `pathinfo($_FILES["foto"]["name"], PATHINFO_EXTENSION)` y la conviertes a minúsculas con `strtolower()`, porque "JPG" y "jpg" deben tratarse igual. Luego verificas con `in_array()` si está en tu lista permitida.

Pero ojo: la extensión también la pone el usuario. Un archivo llamado "foto.jpg" puede no ser una imagen. La extensión es solo el primer filtro, el de la entrada. El filtro de verdad viene en la siguiente sección.

Ejemplo de lista blanca para imágenes: ["jpg", "jpeg", "png", "gif", "webp"]. Para documentos: ["pdf"]. Define tu lista según lo que tu aplicación necesite, ni más ni menos. Si tu sitio solo necesita fotos de perfil, no aceptes pdf ni zip "por si acaso". Cada tipo extra que aceptas es una puerta extra que hay que vigilar.

Verificar que sea una imagen real con getimagesize()

Aquí está el truco que separa al aficionado del profesional. Un atacante puede tomar un script malicioso de PHP, llamarlo "foto.jpg" y subirlo. La extensión dice jpg, el navegador dijo "image/jpeg", todo parece bien... pero el contenido es código. Si tu servidor llega a ejecutar ese archivo, el atacante toma el control.

La función `getimagesize()` lee los primeros bytes del archivo y verifica si corresponden a una imagen real. Si le pasas la ruta temporal del archivo subido y devuelve false, no es una imagen: rechazas la subida. Si devuelve un arreglo, es una imagen de verdad, y además te da el ancho, el alto y el tipo real.

El tipo real viene en el índice 2 del arreglo (constantes como `IMAGETYPE_JPEG`, `IMAGETYPE_PNG`) o puedes usar el índice "mime". Con esto ya no dependes de lo que el usuario declare: tú mismo abres la caja y miras que hay adentro.

Analogía: es como cuando el cadenero no se conforma con ver la identificación por fuera, sino que la pone bajo la luz para ver si es auténtica. La extensión es la portada de la identificación; `getimagesize()` es la prueba de autenticidad.

Para archivos que no son imágenes (por ejemplo un PDF), la verificación equivalente es revisar los primeros bytes del archivo, conocidos como "números mágicos". Un PDF real empieza con "%PDF". Pero en esta unidad nos concentramos en imágenes, que es el caso más común.

Combinando las tres validaciones (tamaño, extensión con lista blanca, contenido real) ya tienes un filtro serio. Ninguna es suficiente sola; juntas forman una defensa en capas, como las tres chapas de una puerta blindada.

Renombrar archivos con `uniqid()`: el folio de cada paquete

Guardar el archivo con su nombre original es una mala idea por tres razones. Primera: dos usuarios pueden subir "foto.jpg" y el segundo pisa al primero. Segunda: el nombre original puede traer caracteres problematicos (espacios, acentos, símbolos) que rompen URLs o rutas. Tercera y más grave: un atacante puede usar un nombre como ".././../config.php" o "truco.php.jpg" para intentar escribir fuera de tu carpeta o confundir al servidor.

La solución profesional es ignorar el nombre original y generar tu uno único con `uniqid()`. Esta función genera un identificador basado en la hora actual con precisión de microsegundos, así que es prácticamente imposible que se repita. La forma típica es: `uniqid("img_", true) . "." . $extensión`. El primer parámetro es un prefijo para que el nombre sea legible ("img_"), el segundo (true) agrega más entropía para hacerlo aun más único.

Así, "mis-vacaciones.jpg" se convierte en algo como "img_64f2a1b3c8d9e12.34567890.jpg". Feo para un humano, perfecto para un servidor: único, seguro, sin caracteres raros.

Analogía: en una bodega profesional ninguna caja se guarda con el nombre que traía escrito con plumon. Al entrar, cada caja recibe un folio único del sistema: "CAJA-004821". Con ese folio la encuentras siempre, nunca hay duplicados y nadie puede adivinar folios para robar cajas ajenas. `uniqid()` es tu máquina de folios.

Guarda el nombre original en la base de datos si lo necesitas para mostrarselo al usuario ("tu archivo mis-vacaciones.jpg se subió correctamente"), pero el archivo en disco siempre lleva el nombre generado por ti.

Guardar la ruta en la base de datos

Un error comun de principiantes es intentar guardar el archivo completo dentro de la base de datos como un BLOB. Tecnicamente se puede, pero es lento, infla la base y complica los respaldos. La práctica profesional es: el archivo vive en el disco (carpeta uploads) y en la base de datos guardas solo su ruta, como texto.

Por ejemplo, tu tabla "fotos" puede tener columnas `id`, `usuario_id`, `nombre_original`, `nombre_archivo` y `fecha_subida`. Cuando el usuario sube su foto, mueves el archivo a "uploads/img_64f2a1b3.jpg" y guardas esa ruta en la columna `nombre_archivo`. Cuando quieras mostrar la foto, haces un `SELECT`, obtienes la ruta y la pones en una etiqueta ``.

Analogía: la base de datos es la libreta de la bodega. En la libreta no metes las cajas físicas; anotas "folio CAJA-004821, pasillo 3, estante B". Cuando necesitas la caja, consultas la libreta y vas al pasillo indicado. Rápido, ordenado y escalable.

Y por supuesto, ese INSERT a la base de datos debe hacerse con sentencias preparadas, como repasaremos más adelante. La ruta del archivo también es un dato que viene del exterior y debe tratarse con cuidado.

XSS: el ataque del comentario malicioso

XSS significa Cross-Site Scripting (letras cruzadas porque CSS ya estaba ocupado por las hojas de estilo). Es uno de los ataques más comunes de la web y funciona así: tu sitio permite a los usuarios escribir algo (un comentario, un nombre, una biografía) y luego lo muestras a otros visitantes sin revisarlo. El atacante escribe en el comentario algo como:

```
<script>alert("te hackee");</script>
```

Si tu página imprime ese texto tal cual, el navegador de cada visitante lo interpreta como código JavaScript real y lo ejecuta. El alert es solo la demostración inofensiva. En un ataque real, ese script puede robar la cookie de sesión del visitante y enviarla al servidor del atacante. Con esa cookie, el atacante entra a la cuenta de la víctima como si fuera ella. Todo porque tu imprimiste sin filtrar lo que un usuario escribió.

Hay tres tipos principales de XSS, pero para esta unidad concéntrate en el reflejado y el almacenado: el almacenado es el más peligroso porque el código malicioso queda guardado en tu base de datos (en un comentario, por ejemplo) y se ejecuta cada vez que alguien abre la página. Un solo comentario malicioso puede atacar a miles de visitantes.

La defensa es `htmlspecialchars()`. Esta función convierte los caracteres especiales de HTML en entidades inofensivas: `<` se convierte en `<`, `>` en `>`, `"` en `"`, `&` en `&`. Así, el texto `<script>` se muestra en pantalla literalmente como `"<script>"` pero el navegador ya no lo interpreta como código. Es como ponerle una vitrina al cuchillo: se ve, pero no corta.

Usala SIEMPRE que imprimas datos que vinieron del usuario: `echo htmlspecialchars($nombre, ENT_QUOTES, "UTF-8");`. El parámetro `ENT_QUOTES` convierte también las comillas simples, y `"UTF-8"` indica la codificación. Hazlo un hábito: si el dato lo escribió un usuario, pasa por `htmlspecialchars` antes de mostrarse. Sin excepciones.

Analogía: imagina que en tu restaurante los clientes pueden dejar notas en un pizarrón. Un gracioso escribe "ignoren el menú, la cocina está cerrada, vayan a tal dirección". Si tu dejas la nota tal cual, los demás clientes le creen al pizarrón. `htmlspecialchars` es como enmarcar cada nota entre comillas visibles: todos ven que es solo la opinión de un cliente, nadie la confunde con un anuncio oficial.

Inyeccion SQL repasada: sentencias preparadas

La inyección SQL es el primo hermano del XSS: en vez de inyectar código en la página, el atacante inyecta código en tu consulta a la base de datos. Funciona cuando armas una consulta pegando directamente lo que el usuario escribió. Ejemplo del login clásico:

```
SELECT * FROM usuarios WHERE email = "$email" AND clave = "$clave"
```

Si el atacante escribe en el campo de clave: " OR "1"="1, la consulta se convierte en:

```
SELECT * FROM usuarios WHERE email = "x" AND clave = "" OR "1"="1"
```

Como "1"="1" siempre es verdadero, la condición se cumple para todos los usuarios y el atacante entra sin saber ninguna contraseña. Es como decirle al cadenero "dejame pasar a mi o a cualquier persona del mundo": con ese "o" magico, la puerta se abre.

La defensa son las sentencias preparadas (prepared statements). En vez de pegar los datos dentro del SQL, mandas primero la estructura de la consulta con marcadores ?, y después mandas los datos por separado. La base de datos los trata siempre como datos, nunca como código. Así:

```
$stmt = $conexión->prepare("SELECT * FROM usuarios WHERE email = ?");  
$stmt->bind_param("s", $email); $stmt->execute();
```

Aunque el atacante escriba código SQL en el email, la base lo buscara literalmente como texto: buscara un usuario cuyo email sea "" OR "1"="1", no lo encontrara y listo. El ataque muere solo.

Regla: toda consulta que incluya datos del usuario (GET, POST, cookies, rutas de archivo, lo que sea) debe usar sentencias preparadas. Sin excepciones. Si ves código que pega variables directamente en el SQL, es código vulnerable, punto.

Nunca confies en el usuario: el cadenero del antro

Esta es la regla de oro de la seguridad web y merece su propia sección porque lo explica todo: nunca confies en nada que venga del cliente. Nada. Ni en los campos del formulario, ni en el tipo MIME del archivo, ni en la extensión, ni en el tamaño declarado, ni en las cookies, ni en los headers. Todo lo que llega del navegador puede ser falsificado.

Piensa en el cadenero del antro. El cadenero no deja pasar a alguien porque "se ve buena gente" o porque "trae traje caro". El cadenero revisa: identificación oficial (¿es autentica?), lista de invitados (¿vienes anotado?), estado (¿vienes en condiciones de entrar?). Y si algo no cuadra, no entras, aunque protestes. Tu código debe ser ese cadenero.

Cada validación de esta unidad es una revisión del cadenero:

- Revisar `$_FILES["foto"]["error"]` es pedir la identificación.
- Validar el tamaño es revisar que no traigas una maleta sospechosa.
- La lista blanca de extensiones es la lista de invitados: si no estas en la lista, no entras.
- `getimagesize()` es la prueba de autenticidad de la identificación.
- `htmlspecialchars()` es revisar que no traigas nada peligroso en los bolsillos antes de mostrarte al público.
- Las sentencias preparadas son hablar por radio con la base para verificar cada dato.

El error mental del principiante es pensar "¿quien querría atacar mi sitio chiquito?". La respuesta: los ataques automatizados no eligen victimas, barren miles de sitios por hora buscando puertas abiertas. Tu sitio chiquito es exactamente el que encuentran. Se el cadenero, siempre.

Contraseñas: por qué NUNCA se guardan en texto plano

Te voy a contar una historia que se ha repetido decenas de veces. Un sitio de recetas, un foro de videojuegos, una tienda en línea: un día un atacante encuentra una vulnerabilidad y descarga la tabla de usuarios completa. Miles de emails y contraseñas quedan en sus manos. Si esas contraseñas estaban guardadas en texto plano, el atacante ya ganó el premio mayor: no solo entra a ese sitio, sino que prueba esas mismas combinaciones en Gmail, Facebook, bancos y todo lo demás, porque la gente reuse sus contraseñas. Una sola filtración se convierte en cientos de cuentas robadas.

Esto no es teoría: ha pasado con sitios gigantes, con millones de usuarios. Y la lección es siempre la misma: debes programar asumiendo que tu base de datos algún día puede filtrarse. No es pesimismo, es profesionalismo. Tu trabajo es que, si eso pasa, lo robado no sirva para nada.

Guardar contraseñas en texto plano es como guardar las llaves de tus clientes en una caja con etiquetas que dicen "llave de la casa de Juan". Si alguien abre la caja, tiene todo. La alternativa es guardar solo una "huella" irreversible de la contraseña: algo que te permite verificar que la contraseña es correcta cuando el usuario la escribe, pero que no permite reconstruir la contraseña original aunque el atacante tenga la base completa.

Esa huella se llama hash, y en PHP la herramienta correcta es `password_hash()`. Ojo: no uses `md5()` ni `sha1()` para contraseñas. Esos algoritmos son rapidísimos, y la velocidad es enemiga aquí: un atacante puede probar miles de millones de combinaciones por segundo con `md5`.

Además no usan "sal" automática. `password_hash()` sí: cada hash incluye una sal aleatoria única, así que dos usuarios con la misma contraseña tienen hashes distintos, y es intencionalmente lento para que probar combinaciones sea carísimo.

`password_hash()` con `PASSWORD_DEFAULT` explicado

La función se usa así:

```
$hash = password_hash($clave, PASSWORD_DEFAULT);
```

Recibe la contraseña en texto plano (la que el usuario escribió en el registro) y devuelve una cadena de unos 60 caracteres que es el hash. Ese hash es lo que guardas en la base de datos, en una columna VARCHAR de al menos 255 caracteres (para que quepa aunque el algoritmo cambie en el futuro).

`PASSWORD_DEFAULT` es una constante mágica: le dice a PHP "usa el mejor algoritmo disponible". Hoy es `bcrypt`, mañana podría ser otro más fuerte, y tu código no tiene que cambiar: PHP se encarga. Es como una caja fuerte que se actualiza sola: cada vez que sale un modelo más seguro, tus contraseñas nuevas ya usan el nuevo modelo sin que muevas un dedo.

Detalles importantes que debes saber:

- Cada vez que llamas a `password_hash()` con la misma contraseña, obtienes un hash DISTINTO, porque la sal aleatoria cambia. Esto es normal y es bueno. Por eso nunca compares hashes directamente.
- El hash ya incluye dentro de sí el algoritmo usado, la sal y el costo. Todo lo necesario para verificar después viaja dentro del propio hash. No necesitas guardar nada más.
- La función tarda unas décimas de segundo a propósito. En un registro eso no se nota; para un atacante que quiere probar millones de contraseñas, es una muralla.

Analogía: `password_hash` es como tomar la huella digital de la contraseña y guardar solo la huella en tu archivo. Cuando el usuario regresa, tomas de nuevo su huella y la comparas con la archivada. Si alguien roba tu archivo de huellas, no puede reconstruir los dedos de nadie.

`password_verify()` y el login seguro

Si los hashes son distintos cada vez, ¿cómo verificas la contraseña en el login? Con `password_verify()`, la pareja inseparable de `password_hash()`:

```
if (password_verify($clave_escrita, $hash_guardado)) { /* acceso permitido */ }
```

Esta función toma la contraseña que el usuario escribió ahorita, le aplica el mismo algoritmo con la misma sal (que lee desde el hash guardado) y compara el resultado. Devuelve true o false. Simple, seguro, y resistente a ataques de tiempo porque la comparación interna esta diseñada para no dar pistas.

Error típico de principiante: guardar el hash y luego en el login hacer `password_hash($clave, PASSWORD_DEFAULT) == $hash_guardado`. Esto NUNCA funciona, porque el hash nuevo tiene una sal nueva y será distinto aunque la contraseña sea correcta. Siempre usa `password_verify()`. Grabatelo: para crear, `password_hash`; para comprobar, `password_verify`.

Login seguro completo paso a paso

Vamos a armar el flujo completo de un login seguro, pieza por pieza, como si fuera una receta:

1. El usuario envia su email y su contraseña por POST desde un formulario con `method="post"`. Nunca por GET, porque GET deja la contraseña en la URL, en el historial y en los registros del servidor.
2. En el servidor, buscas al usuario por su email usando una sentencia preparada. Solo por email: no pongas la contraseña en la consulta SQL. Si no existe ningun usuario con ese email, muestras un mensaje generico (más sobre esto abajo).
3. Si el usuario existe, tomas el hash guardado y lo comparas con `password_verify($clave_escrita, $hash)`. Si devuelve false, mensaje generico de error.
4. Si devuelve true, el login es correcto. Aquí regeneras el ID de sesión con `session_regenerate_id(true)` para evitar el ataque de fijación de sesión (que un atacante te plante una sesión conocida). Luego guardas en `$_SESSION` los datos minimos: el id del usuario y quizá su nombre. Nunca guardes la contraseña ni el hash en la sesión.
5. Rediriges al usuario a su página privada con `header("Location: perfil.php")` y `exit`. El `exit` es obligatorio: sin el, el script sigue ejecutandose después del header.
6. En cada página protegida, al inicio del script verificas: si no existe `$_SESSION["usuario_id"]`, rediriges al login. Así de simple y así de importante.
7. El logout destruye la sesión: `$_SESSION = []`, `session_destroy()`, y borrar la cookie de sesión.

Sobre los mensajes de error: en el login nunca digas "el email no existe" o "la contraseña es incorrecta". Esos mensajes le dicen al atacante cuál de los dos datos es válido, y con eso puede enumerar cuentas. Usa siempre un mensaje generico: "Email o contraseña incorrectos". Al usuario legítimo le basta; al atacante no le das pistas.

CSRF explicado de forma simple

CSRF significa Cross-Site Request Forgery, o falsificación de petición entre sitios. Suena complicado, pero la idea es simple: tu ya iniciaste sesión en tu banco (sitio A). Sin cerrar sesión, abres otra pestaña y visitas un sitio malicioso (sitio B). El sitio B tiene un formulario oculto que se autoenvía al sitio A diciendo "transfiere dinero a esta cuenta". Como tu navegador todavía tiene tu sesión del banco activa, el banco cree que la petición la hiciste tu y la ejecuta. Tu solo visitaste una página, y sin darte cuenta hiciste una transferencia.

La defensa es el token CSRF: un valor aleatorio secreto que tu servidor genera, guarda en la sesión y mete como campo oculto en cada formulario importante. Cuando el formulario llega, el servidor verifica que el token recibido coincida con el de la sesión. El sitio malicioso no conoce ese token (no puede leer tu sesión), así que su formulario falsificado será rechazado.

En la práctica: al mostrar el formulario, generas `$_SESSION["csrf"] = bin2hex(random_bytes(32));` y lo pones en `<input type="hidden" name="csrf" value="...">`. Al procesar, comparas con `hash_equals($_SESSION["csrf"], $_POST["csrf"])`. La función `hash_equals` compara cadenas de forma segura, sin dar pistas de tiempo. Si no coinciden, rechazas la petición.

Aplica tokens CSRF en toda acción que cambie algo: subir archivos, cambiar contraseña, borrar cuenta, publicar. Las acciones de solo lectura no lo necesitan.

Errores comunes

Vamos a repasar los tropiezos clásicos, para que los reconozcas antes de cometerlos:

- Olvidar el `enctype="multipart/form-data"` en el formulario. Sintoma: `$_FILES` llega vacío aunque todo "se ve bien". Es el primer sospechoso siempre.
- Confiar en `$_FILES["foto"]["type"]`. Ese valor lo manda el navegador y se falsifica en dos segundos. Valida con `getimagesize()` o revisando el contenido real, nunca con el `type` declarado.
- Guardar archivos con su nombre original. Colisiones entre usuarios, caracteres raros que rompen rutas, y nombres maliciosos tipo `../truco.php`. Genera siempre el nombre con `uniqid()` y conserva la extensión validada.
- Comparar hashes con `==` en vez de usar `password_verify()`. Como cada hash tiene sal distinta, la comparación directa siempre falla aunque la contraseña sea correcta. Y si comparas texto plano con `==`, estas haciendo todo mal desde el inicio.

- Guardar la contraseña en texto plano "temporalmente" o "solo en este proyecto chico". No hay proyecto tan chico que justifique eso. El habito se construye aquí, en la práctica.
- Mostrar errores detallados al usuario en producción ("Error en la línea 42: Access denied for user..."). Esos mensajes son un mapa del tesoro para el atacante. En producción: mensajes genericos para el usuario, detalles solo en tus registros internos.
- No poner exit después de header("Location: ..."). El navegador redirige, pero tu script sigue corriendo en el servidor y puede ejecutar código que no debia.

Buenas prácticas

Cerramos la teoría con la lista de habitos profesionales. No son opcionales si quieres que tu sitio sobreviva en internet de verdad:

- Guarda los uploads fuera de la raíz publica del sitio cuando puedas, y sirvelos con un script PHP que verifique permisos. Si deben estar en una carpeta publica, bloquea la ejecución de scripts ahí con un archivo .htaccess que niegue la ejecución de PHP, o configura tu servidor para que esa carpeta solo sirva archivos estaticos.
- Pon limites de tamaño en todos los niveles: en php.ini (upload_max_filesize, post_max_size) y en tu código (\$_FILES["foto"]["size"]). Define el limite según lo que necesites: 2 MB para fotos de perfil, quizá más para documentos, pero siempre con un número concreto.
- En el login, mensajes de error genericos: "Email o contraseña incorrectos". Nunca reveles cuál de los dos falló.
- Regenera el ID de sesión al iniciar sesión (session_regenerate_id(true)) y destruye la sesión completa al cerrar sesión.
- Usa HTTPS en producción. Sin HTTPS, las contraseñas y las cookies de sesión viajan legibles por la red y cualquiera en el mismo WiFi las puede leer.
- Valida en el servidor, no solo en el navegador. El atributo accept="image/*" del input y el JavaScript son comodidad para el usuario honesto; el atacante los ignora por completo. La validación real siempre vive en PHP.
- Manten PHP actualizado. PASSWORD_DEFAULT mejora con las versiones nuevas; cada versión también cierra agujeros de seguridad. Un PHP viejo es una puerta vieja.

Con esto tienes la teoría completa. Ahora vamos a ver todo esto convertido en código real, ejemplo por ejemplo.

Ejemplos paso a paso

Ejemplo 1: formulario de subida

Este es el formulario HTML que el usuario ve. Sin el enctype correcto, nada de lo demás funciona.

```
<form action="subir.php" method="post" enctype="multipart/form-data">
  <label for="foto">Elige tu foto de perfil (JPG o PNG, maximo 2 MB):</label>
  <input type="file" name="foto" id="foto" accept="image/*" required>
  <button type="submit">Subir foto</button>
</form>
```

Línea por línea:

- `<form action="subir.php" ...>`: los datos se enviaran al script subir.php, que hara todo el trabajo de validación.
- `method="post"`: obligatorio para enviar archivos; con GET es imposible.
- `enctype="multipart/form-data"`: la pieza clave. Le dice al navegador que empaque el archivo como datos binarios en varias partes. Sin esto, `$_FILES` llega vacío.
- `<input type="file" name="foto">`: el campo para elegir el archivo. El `name="foto"` es la llave que usaras en PHP: `$_FILES["foto"]`.
- `accept="image/*"`: sugerencia visual para que el selector de archivos muestre imagenes. Es comodidad, no seguridad: el atacante lo ignora.
- `required`: el navegador exige elegir un archivo antes de enviar. También es solo comodidad; la validación real está en subir.php.
- `<button type="submit">`: envia el formulario.

Ejemplo 2: script que valida y guarda la imagen

Este es el corazon de la unidad: subir.php recibe el archivo, lo revisa como cadenero y lo guarda con nombre único.

```

<?php
// 1. Configuración: a donde van los archivos y que se permite
$carpeta = "uploads/";
$tamano_maximo = 2 * 1024 * 1024; // 2 MB en bytes
$extensiones_permitidas = ["jpg", "jpeg", "png", "gif", "webp"];

// 2. Verificar que si llegó un archivo y no hubo error
if (!isset($_FILES["foto"]) || $_FILES["foto"]["error"] !== 0) {
    die("No se recibió el archivo o hubo un error en la subida.");
}

$archivo = $_FILES["foto"];

// 3. Validar el tamaño antes de hacer cualquier otra cosa
if ($archivo["size"] > $tamano_maximo) {
    die("El archivo es demasiado grande. Máximo 2 MB.");
}

// 4. Validar la extensión con lista blanca
$extension = strtolower(pathinfo($archivo["name"], PATHINFO_EXTENSION));
if (!in_array($extension, $extensiones_permitidas)) {
    die("Tipo de archivo no permitido. Solo imágenes JPG, PNG, GIF o WEBP.");
}

// 5. Verificar que sea una imagen REAL, no un disfraz
$info = getimagesize($archivo["tmp_name"]);
if ($info === false) {
    die("El archivo no es una imagen válida.");
}

// 6. Generar un nombre único y armar el destino final
$nombre_nuevo = uniqid("img_", true) . "." . $extension;
$destino = $carpeta . $nombre_nuevo;

// 7. Mover el archivo de la carpeta temporal a su hogar definitivo
if (!move_uploaded_file($archivo["tmp_name"], $destino)) {
    die("No se pudo guardar el archivo. Intenta de nuevo.");
}

// 8. Éxito: aquí guardarías $destino en la base de datos
echo "Foto subida correctamente: " . htmlspecialchars($nombre_nuevo, ENT_QUOTES, "UTF-8");
?>

```


Línea por línea:

- Las variables de configuración arriba del todo facilitan cambiar limites sin cazar números por el código.
- El paso 2 revisa el código de error antes que nada. Si error no es 0, no hay nada que validar: se rechaza.
- El paso 3 compara size con el limite. Va antes de mover el archivo: no guardamos algo que ya sabemos que pesa de más.
- El paso 4 extrae la extensión con pathinfo(), la pasa a minusculas y la busca en la lista blanca con in_array(). Todo lo que no este en la lista se rechaza.
- El paso 5 es la prueba de autenticidad: getimagesize() lee el contenido real del archivo temporal. Si devuelve false, era un disfraz.
- El paso 6 genera el folio único con uniqid("img_", true) y le pega la extensión ya validada. El nombre original del usuario se ignora por completo.
- El paso 7 usa move_uploaded_file(), que además verifica que el archivo haya llegado por HTTP de verdad.
- El paso 8 muestra el resultado usando htmlspecialchars() incluso para el nombre generado: habito de seguridad, siempre.

Ejemplo 3: galeria que muestra las imagenes subidas

Una vez guardadas las rutas en la base de datos, mostrarlas es sencillo. Este script lee la tabla "fotos" y genera una galeria.

```

<?php
$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

$resultado = $conexion->query("SELECT nombre_original, nombre_archivo FROM fotos
ORDER BY id DESC");

while ($fila = $resultado->fetch_assoc()) {
    $ruta = htmlspecialchars($fila["nombre_archivo"], ENT_QUOTES, "UTF-8");
    $titulo = htmlspecialchars($fila["nombre_original"], ENT_QUOTES, "UTF-8");
    echo "<figure>";
    echo "<img src=\"\" . $ruta . \"\" alt=\"\" . $titulo . \"\" width=\"200\">";
    echo "<figcaption>" . $titulo . "</figcaption>";
    echo "</figure>";
}

$conexion->close();
?>

```

Línea por línea:

- La conexión mysqli es la base; en producción estas credenciales no van quemadas en el código, pero para el ejemplo es claro.
- El SELECT no lleva datos del usuario, así que puede ir directo. Si llevara un filtro (por ejemplo WHERE usuario_id = ?), usarías sentencia preparada.
- Dentro del ciclo, cada ruta y cada nombre pasan por htmlspecialchars() antes de imprimirse. La ruta viene de tu propio sistema, pero el nombre original lo escribió un usuario: podría contener código. Nunca imprimas datos de usuario sin escapar.
- Las comillas del atributo src se escapan en el código PHP con \" porque el echo usa comillas dobles.
- Mostrar la imagen es solo poner la ruta en el src: el archivo vive en disco, la base solo presta la dirección.

Ejemplo 4: ataque XSS y su corrección

Primero la versión VULNERABLE, para que veas el peligro con tus propios ojos. Nunca uses este código en un sitio real.

```
<?php
// VERSION VULNERABLE: NO LA USES EN PRODUCCION
$comentario = $_POST["comentario"];
echo "<p>Comentario: " . $comentario . "</p>";
?>
```

Si un atacante envía como comentario `<script>alert(document.cookie);</script>`, ese script se ejecuta en el navegador de cada visitante y puede robar sus cookies de sesión. El servidor lo guardó y lo sirve tal cual: es un XSS almacenado.

Ahora la versión CORREGIDA. El cambio es de una sola función, pero lo cambia todo:

```
<?php
// VERSION SEGURA: así sí
$comentario = $_POST["comentario"];
$seguro = htmlspecialchars($comentario, ENT_QUOTES, "UTF-8");
echo "<p>Comentario: " . $seguro . "</p>";
?>
```

Línea por línea de la corrección:

- `htmlspecialchars()` convierte `<` en `<` y `>` en `>`. El navegador muestra el texto `"<script>"` en pantalla, pero ya no lo interpreta como código: es solo texto inofensivo dentro del párrafo.
- `ENT_QUOTES` convierte también las comillas simples y dobles, cerrando la puerta a trucos que rompen atributos HTML.
- `"UTF-8"` le dice a la función que codificación usar para no corromper acentos ni ñes.

La lección: el problema nunca fue guardar el comentario; fue mostrarlo sin escapar. Todo dato que venga del usuario y se muestre en HTML pasa por `htmlspecialchars()`. Sin excepciones, sin "es que este campo es de confianza".

Ejemplo 5: registro de usuarios con `password_hash`

El registro es donde nace la contraseña. Aquí se crea el hash que vivirá en la base de datos.

```

<?php
$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

$nombre = trim($_POST["nombre"]);
$email = trim($_POST["email"]);
$clave = $_POST["clave"];

// Validaciones basicas del cadenero
if (empty($nombre) || empty($email) || empty($clave)) {
    die("Todos los campos son obligatorios.");
}
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    die("El email no es valido.");
}
if (strlen($clave) < 8) {
    die("La clave debe tener al menos 8 caracteres.");
}

// Crear el hash: la huella irreversible de la clave
$hash = password_hash($clave, PASSWORD_DEFAULT);

// Guardar con sentencia preparada: estructura y datos por separado
$stmt = $conexion->prepare("INSERT INTO usuarios (nombre, email, clave_hash) VALUES
(?, ?, ?)");
$stmt->bind_param("sss", $nombre, $email, $hash);

if ($stmt->execute()) {
    echo "Registro exitoso. Ya puedes iniciar sesion.";
} else {
    echo "No se pudo registrar. El email quiza ya existe.";
}

$stmt->close();
$conexion->close();
?>

```

Línea por línea:

- trim() quita espacios al inicio y al final: evita emails con espacios accidentales.
- Las validaciones basicas rechazan datos vacios, emails mal formados y claves cortas. Es el cadenero pidiendo identificación.

- `password_hash($clave, PASSWORD_DEFAULT)` crea el hash con sal aleatoria. La variable `$clave` en texto plano no se vuelve a usar después de esto.
- La sentencia preparada separa la estructura SQL de los datos: `prepare()` manda el esqueleto con `?`, `bind_param("sss", ...)` dice "tres textos", y `execute()` los envía. Aunque el nombre contenga código SQL, la base lo trata como texto.
- El mensaje de error es genérico: no revelamos si el email ya existía o fue otro problema. Menos pistas, mejor.

Ejemplo 6: login seguro con `password_verify` y sesiones

El login es donde se comprueba la huella. Nota que la clave nunca aparece en el SQL.

```

<?php
session_start();

$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

$email = trim($_POST["email"]);
$clave = $_POST["clave"];

// Buscar al usuario SOLO por email, con sentencia preparada
$stmt = $conexion->prepare("SELECT id, nombre, clave_hash FROM usuarios WHERE email
= ?");
$stmt->bind_param("s", $email);
$stmt->execute();
$resultado = $stmt->get_result();

if ($fila = $resultado->fetch_assoc()) {
    // El usuario existe: verificar la clave contra el hash guardado
    if (password_verify($clave, $fila["clave_hash"])) {
        // Clave correcta: renovar la sesion y guardar lo minimo
        session_regenerate_id(true);
        $_SESSION["usuario_id"] = $fila["id"];
        $_SESSION["usuario_nombre"] = $fila["nombre"];
        header("Location: perfil.php");
        exit;
    }
}

// Si llegamos aqui, algo fallo: mensaje generico, sin pistas
echo "Email o clave incorrectos.";

$stmt->close();
$conexion->close();
?>

```

Línea por línea:

- session_start() va antes que cualquier salida HTML: las sesiones usan headers.
- El SELECT busca por email unicamente. La clave no va en el WHERE porque los hashes con sal distinta nunca coinciden con ==.

- password_verify() toma la clave recién escrita y el hash guardado, y devuelve true o false. Es la única forma correcta de comparar.
- session_regenerate_id(true) cambia el ID de sesión al entrar: evita que un atacante que planto una sesión la siga usando.
- En \$_SESSION se guarda solo el id y el nombre. Jamás la clave ni el hash.
- header("Location: perfil.php") redirige, y exit detiene el script de inmediato. Sin exit, el código de abajo seguiría corriendo.
- El mensaje final es genérico a propósito: no dice si falló el email o la clave. Así el atacante no puede enumerar cuentas.

Ejemplo 7: página de perfil protegida

Esta página solo debe verse si hay sesión activa. El guardia va al inicio del archivo.

```
<?php
session_start();

// Guardia: sin sesion, de regreso al login
if (!isset($_SESSION["usuario_id"])) {
    header("Location: login.html");
    exit;
}

$nombre = htmlspecialchars($_SESSION["usuario_nombre"], ENT_QUOTES, "UTF-8");
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="UTF-8"><title>Mi perfil</title></head>
<body>
    <h1>Hola, <?php echo $nombre; ?></h1>
    <p>Esta es tu zona privada.</p>
    <p><a href="subir-foto.html">Subir foto de perfil</a></p>
    <p><a href="cambiar-clave.html">Cambiar mi clave</a></p>
    <p><a href="salir.php">Cerrar sesion</a></p>
</body>
</html>
```

Línea por línea:

- El bloque del guardia es lo primero del archivo, antes de cualquier HTML. Si no hay usuario_id en la sesión, redirige al login y detiene todo con exit.

- El nombre se escapa con htmlspecialchars() aunque venga de tu propia sesión: el nombre lo escribió el usuario en el registro y podría contener código.
- El resto es HTML normal con enlaces a las acciones privadas. Cada una de esas páginas debe tener su propio guardia igual a este.
- El enlace "Cerrar sesión" apunta a salir.php, que destruye la sesión por completo.

Ejemplo 8: cambió de contraseña

Cambiar la clave exige verificar la clave actual primero: si alguien dejó la sesión abierta en una computadora ajena, el atacante no podrá cambiar la clave sin saber la anterior.

```
<?php
session_start();
if (!isset($_SESSION["usuario_id"])) {
    header("Location: login.html");
    exit;
}

$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

$actual = $_POST["clave_actual"];
$nueva = $_POST["clave_nueva"];
$usuario_id = $_SESSION["usuario_id"];

// 1. Obtener el hash actual de ESTE usuario (sentencia preparada)
$stmt = $conexion->prepare("SELECT clave_hash FROM usuarios WHERE id = ?");
$stmt->bind_param("i", $usuario_id);
$stmt->execute();
$fila = $stmt->get_result()->fetch_assoc();
$stmt->close();

// 2. Verificar que la clave actual sea correcta
if (!$fila || !password_verify($actual, $fila["clave_hash"])) {
    die("La clave actual no es correcta.");
}

// 3. Validar la clave nueva
if (strlen($nueva) < 8) {
    die("La clave nueva debe tener al menos 8 caracteres.");
}
```



```
// 4. Crear el hash nuevo y guardarlo
$hash_nuevo = password_hash($nueva, PASSWORD_DEFAULT);
$stmt = $conexion->prepare("UPDATE usuarios SET clave_hash = ? WHERE id = ?");
$stmt->bind_param("si", $hash_nuevo, $usuario_id);

if ($stmt->execute()) {
    echo "Clave actualizada correctamente.";
} else {
    echo "No se pudo actualizar la clave.";
}

$stmt->close();
$conexion->close();
?>
```

Línea por línea:

- El guardia de sesión protege la página: solo el dueño de la cuenta puede cambiar su clave.
- El SELECT trae el hash del usuario autenticado usando su id de sesión, no un id que venga del formulario. Así nadie puede cambiar la clave de otro.
- password_verify() comprueba la clave actual: si el que esta frente al teclado no la sabe, se acabo el intentó.
- La clave nueva se valida (mínimo 8 caracteres) y se convierte en hash nuevo con password_hash(). La vieja queda inservible.
- El UPDATE usa sentencia preparada con dos marcadores: el hash (texto) y el id (entero). bind_param("si", ...) refleja esos tipos.
- Nota que en ningun momento la clave viaja en texto plano a la base: solo hashes.

Práctica

Ejercicio 1

Enunciado: Crea un sistema para subir documentos PDF (por ejemplo, comprobantes o tareas). Requisitos: formulario con enctype correcto, limite de 5 MB, solo extensión .pdf, verificación de que el archivo sea un PDF real revisando que sus primeros 4 bytes sean "%PDF", renombrado con uniqid() y guardado en la carpeta "documentos/". Si algo falla, muestra un mensaje claro.

Solución 1

Formulario (subir-pdf.html):

```
<form action="subir-pdf.php" method="post" enctype="multipart/form-data">
  <label for="doc">Sube tu documento PDF (maximo 5 MB):</label>
  <input type="file" name="doc" id="doc" accept=".pdf" required>
  <button type="submit">Subir documento</button>
</form>
```

Script (subir-pdf.php):

```

<?php
$carpeta = "documentos/";
$tamano_maximo = 5 * 1024 * 1024; // 5 MB

if (!isset($_FILES["doc"]) || $_FILES["doc"]["error"] !== 0) {
    die("No se recibio el archivo o hubo un error en la subida.");
}
$sarchivo = $_FILES["doc"];

if ($sarchivo["size"] > $tamano_maximo) {
    die("El archivo supera los 5 MB permitidos.");
}

$extension = strtolower(pathinfo($sarchivo["name"], PATHINFO_EXTENSION));
if ($extension !== "pdf") {
    die("Solo se permiten archivos PDF.");
}

// Verificar el "numero magico": un PDF real empieza con %PDF
$inicio = file_get_contents($sarchivo["tmp_name"], false, null, 0, 4);
if ($inicio !== "%PDF") {
    die("El archivo no es un PDF valido.");
}

$nombre_nuevo = uniqid("doc_", true) . ".pdf";
$destino = $carpeta . $nombre_nuevo;

if (!move_uploaded_file($sarchivo["tmp_name"], $destino)) {
    die("No se pudo guardar el documento.");
}

echo "Documento guardado como: " . htmlspecialchars($nombre_nuevo, ENT_QUOTES, "UTF-8");
?>

```

Explicación paso a paso:

- El formulario usa enctype="multipart/form-data" y accept=".pdf" como ayuda visual. La seguridad real está en el PHP.
- Se revisa el código de error y el tamaño (5 MB) antes de cualquier otra validación.
- La extensión se compara de forma estricta: solo "pdf" pasa. Es una lista blanca de un solo elemento.

- La verificación del número magico es la estrella: `file_get_contents()` lee solo los primeros 4 bytes del archivo temporal y se comparan con `"%PDF"`. Un script disfrazado de PDF no empieza así y queda rechazado. Es el equivalente de `getimagesize()` pero para documentos.
- El archivo se renombra con `uniqid("doc_", true)` y se mueve con `move_uploaded_file()`. El nombre original se descarta por seguridad.
- El mensaje final escapa el nombre con `htmlspecialchars()` por habito.

Ejercicio 2

Enunciado: Crea un libro de visitas con protección contra XSS. El visitante escribe su nombre y un mensaje; se guardan en la base de datos con sentencia preparada y se muestran en la misma página escapando la salida con `htmlspecialchars()`. Prueba escribir `<script>alert("xss")</script>` como mensaje y comprueba que se muestra como texto, no se ejecuta.

Solución 2

```

<?php
$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

// Guardar el mensaje si llego por POST
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"]);
    $mensaje = trim($_POST["mensaje"]);
    if ($nombre !== "" && $mensaje !== "") {
        $stmt = $conexion->prepare("INSERT INTO visitas (nombre, mensaje) VALUES
(?, ?)");
        $stmt->bind_param("ss", $nombre, $mensaje);
        $stmt->execute();
        $stmt->close();
    }
}

// Mostrar todos los mensajes, escapando la salida
$resultado = $conexion->query("SELECT nombre, mensaje FROM visitas ORDER BY id
DESC");
?>
<!DOCTYPE html>
<html lang="es">
<head><meta charset="UTF-8"><title>Libro de visitas</title></head>
<body>
    <h1>Libro de visitas</h1>
    <form method="post">
        <input type="text" name="nombre" placeholder="Tu nombre" required><br>
        <textarea name="mensaje" placeholder="Tu mensaje" required></textarea><br>
        <button type="submit">Firmar</button>
    </form>
    <hr>
    <?php while ($fila = $resultado->fetch_assoc()): ?>
        <p><strong><?php echo htmlspecialchars($fila["nombre"], ENT_QUOTES, "UTF-8");
?></strong> dice:</p>
        <p><?php echo htmlspecialchars($fila["mensaje"], ENT_QUOTES, "UTF-8"); ?></p>
        <hr>
    <?php endwhile; ?>
</body>
</html>
<?php $conexion->close(); ?>

```

Explicación paso a paso:

- Al guardar, la sentencia preparada protege contra inyección SQL: el mensaje se trata como dato, nunca como código.
- Al mostrar, htmlspecialchars() protege contra XSS: convierte <, >, comillas y & en entidades inofensivas.
- Nota que se aplican LAS DOS defensas, cada una en su lugar: preparada al entrar a la base, escape al salir al HTML. Son defensas distintas para ataques distintos; una no sustituye a la otra.
- Si pruebas con <script>alert("xss")</script>, verás el texto literal en pantalla y nada se ejecuta. Esa es la prueba de que la defensa funciona.
- El dato guardado en la base queda intacto (con los <script> originales). Eso está bien: la base guarda la verdad, y el escape se aplica al mostrar. Así, si mañana muestras esos datos en otro formato (un PDF, un correo), decides el escape adecuado para cada salida.

Ejercicio 3

Enunciado: Arma un mini sistema de registro y login en dos archivos. registro.php valida nombre, email y clave (mínimo 8 caracteres), crea el hash con password_hash() y guarda con sentencia preparada. login.php busca por email con sentencia preparada, verifica con password_verify(), regenera la sesión y redirige a bienvenida.php, que muestra "Hola, NOMBRE" solo si hay sesión. Los errores del login deben ser genericos.

Solución 3

Archivo registro.php:

```

<?php
$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"]);
    $email = trim($_POST["email"]);
    $clave = $_POST["clave"];

    if ($nombre === "" || !filter_var($email, FILTER_VALIDATE_EMAIL) ||
    strlen($clave) < 8) {
        die("Datos invalidos: revisa nombre, email y que la clave tenga 8+
    caracteres.");
    }

    $hash = password_hash($clave, PASSWORD_DEFAULT);
    $stmt = $conexion->prepare("INSERT INTO usuarios (nombre, email, clave_hash)
VALUES (?, ?, ?)");
    $stmt->bind_param("sss", $nombre, $email, $hash);
    if ($stmt->execute()) {
        echo "Registro ok. <a href='login.php'>Ir al login</a>";
    } else {
        echo "No se pudo registrar.";
    }
    $stmt->close();
    $conexion->close();
    exit;
}
?>

<form method="post">
    <input type="text" name="nombre" placeholder="Nombre" required><br>
    <input type="email" name="email" placeholder="Email" required><br>
    <input type="password" name="clave" placeholder="Clave (8+ caracteres)"
required><br>
    <button type="submit">Registrarme</button>
</form>

```

Archivo login.php:

```

<?php
session_start();
$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $email = trim($_POST["email"]);
    $clave = $_POST["clave"];

    $stmt = $conexion->prepare("SELECT id, nombre, clave_hash FROM usuarios WHERE
email = ?");
    $stmt->bind_param("s", $email);
    $stmt->execute();
    $fila = $stmt->get_result()->fetch_assoc();
    $stmt->close();

    if ($fila && password_verify($clave, $fila["clave_hash"])) {
        session_regenerate_id(true);
        $_SESSION["usuario_id"] = $fila["id"];
        $_SESSION["usuario_nombre"] = $fila["nombre"];
        header("Location: bienvenida.php");
        exit;
    }
    echo "Email o clave incorrectos.";
    $conexion->close();
    exit;
}
?>
<form method="post">
    <input type="email" name="email" placeholder="Email" required><br>
    <input type="password" name="clave" placeholder="Clave" required><br>
    <button type="submit">Entrar</button>
</form>

```

Archivo bienvenida.php:


```
<?php
session_start();
if (!isset($_SESSION["usuario_id"])) {
    header("Location: login.php");
    exit;
}
echo "Hola, " . htmlspecialchars($_SESSION["usuario_nombre"], ENT_QUOTES, "UTF-8");
echo "<br><a href='salir.php'>Cerrar sesion</a>";
?>
```

Explicación paso a paso:

- En el registro, las tres validaciones (nombre no vacío, email valido, clave de 8+) son el cadenero en la puerta. password_hash() crea la huella y la sentencia preparada la guarda sin riesgo de inyección.
- En el login, el SELECT solo filtra por email. La comparación de la clave ocurre en PHP con password_verify(), nunca en el SQL.
- session_regenerate_id(true) renueva la sesión al entrar; en \$_SESSION solo viajan id y nombre.
- El mensaje "Email o clave incorrectos" es generico: no revela si el email existe.
- bienvenida.php tiene el guardia al inicio: sin sesión no hay saludo. El nombre se escapa al mostrarlo porque lo escribió el usuario en el registro.
- Prueba el flujo completo: registra un usuario, revisa la tabla y confirma que la columna clave_hash contiene un hash largo (no la clave), luego entra con la clave correcta y con una incorrecta para ver el mensaje generico.

Ejercicio 4

Enunciado: Extiende el sistema del ejercicio 3: el usuario con sesión activa puede subir su foto de perfil. Requisitos: solo con sesión iniciada, validaciones completas (error, tamaño máximo 2 MB, lista blanca de extensiones, getimagesize()), renombrado con uniqid(), guardar la ruta en la tabla usuarios (columna avatar) con sentencia preparada usando el id de la sesión, y mostrar el avatar en bienvenida.php con la ruta escapada.

Solución 4

Archivo subir-avatar.php (con guardia de sesión incluido):

```
<?php
```

```

session_start();
if (!isset($_SESSION["usuario_id"])) {
    header("Location: login.php");
    exit;
}

$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $permitidas = ["jpg", "jpeg", "png", "gif", "webp"];
    $maximo = 2 * 1024 * 1024;

    if (!isset($_FILES["avatar"]) || $_FILES["avatar"]["error"] !== 0) {
        die("Error en la subida del archivo.");
    }
    $archivo = $_FILES["avatar"];

    if ($archivo["size"] > $maximo) {
        die("La imagen supera los 2 MB.");
    }

    $extension = strtolower(pathinfo($archivo["name"], PATHINFO_EXTENSION));
    if (!in_array($extension, $permitidas)) {
        die("Extension no permitida.");
    }

    if (getimagesize($archivo["tmp_name"]) === false) {
        die("El archivo no es una imagen real.");
    }

    $nombre_nuevo = uniqid("avatar_", true) . "." . $extension;
    $destino = "uploads/" . $nombre_nuevo;

    if (!move_uploaded_file($archivo["tmp_name"], $destino)) {
        die("No se pudo guardar la imagen.");
    }

    // Guardar la ruta en la BD, ligada al id de la SESSION (no de un formulario)
    $stmt = $conexion->prepare("UPDATE usuarios SET avatar = ? WHERE id = ?");
    $stmt->bind_param("si", $destino, $_SESSION["usuario_id"]);
    $stmt->execute();
    $stmt->close();
}

```

```

        header("Location: bienvenida.php");
        exit;
    }
    $conexion->close();
?>
<form action="subir-avataar.php" method="post" enctype="multipart/form-data">
    <label>Elige tu foto de perfil:</label>
    <input type="file" name="avataar" accept="image/*" required>
    <button type="submit">Subir avataar</button>
</form>

```

Cambié en bienvenida.php para mostrar el avatar:

```

<?php
session_start();
if (!isset($_SESSION["usuario_id"])) {
    header("Location: login.php");
    exit;
}
$conexion = new mysqli("localhost", "root", "", "mi_sitio");
$stmt = $conexion->prepare("SELECT nombre, avatar FROM usuarios WHERE id = ?");
$stmt->bind_param("i", $_SESSION["usuario_id"]);
$stmt->execute();
$fila = $stmt->get_result()->fetch_assoc();
$stmt->close();
$conexion->close();

echo "Hola, " . htmlspecialchars($fila["nombre"], ENT_QUOTES, "UTF-8") . "<br>";
if (!empty($fila["avatar"])) {
    $ruta = htmlspecialchars($fila["avatar"], ENT_QUOTES, "UTF-8");
    echo "<img src=\"\" . $ruta . \"\" alt=\"Mi avatar\" width=\"150\"><br>";
}
echo "<a href='subir-avataar.php'>Cambiar avataar</a> | <a href='salir.php'>Cerrar
sesion</a>";
?>

```

Explicación paso a paso:

- El guardia de sesión va primero: nadie sin login puede subir avatares. La subida queda ligada a una cuenta real.
- Las cuatro validaciones del cadenero están todas: error, tamaño, lista blanca y getimagesize() sobre el archivo temporal. En ese orden.

- El UPDATE usa el id que viene de \$_SESSION, no de un campo oculto del formulario. Si el id viniera del formulario, un atacante podría cambiar el avatar de otro usuario manipulando el campo. El id de la sesión no lo puede falsificar.
- En bienvenida.php la ruta del avatar se lee con sentencia preparada y se escapa con htmlspecialchars() antes de ponerla en el src. La ruta la generó tu sistema, pero el hábito de escapar la salida se aplica siempre.
- Este ejercicio une las tres piezas de la unidad: subida de archivos, sesiones y base de datos, todas con sus defensas.

Ejercicio 5

Enunciado: Protege contra CSRF el cambio de contraseña del Ejemplo 8. Genera un token aleatorio con random_bytes(), guárdalo en la sesión, inclúyelo como campo oculto en el formulario y verifícalo con hash_equals() al procesar. Si el token no coincide, rechaza la petición.

Solución 5

Archivo cambiar-clave.php completo y protegido:

```
<?php
session_start();
if (!isset($_SESSION["usuario_id"])) {
    header("Location: login.php");
    exit;
}

// Crear el token CSRF si aun no existe en esta sesion
if (empty($_SESSION["csrf"])) {
    $_SESSION["csrf"] = bin2hex(random_bytes(32));
}

$conexion = new mysqli("localhost", "root", "", "mi_sitio");
if ($conexion->connect_error) {
    die("Error de conexion.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    // Verificar el token ANTES de hacer cualquier otra cosa
    if (!isset($_POST["csrf"]) || !hash_equals($_SESSION["csrf"], $_POST["csrf"])) {
        die("Petición no válida.");
    }
}
```

```

$actual = $_POST["clave_actual"];
$nueva = $_POST["clave_nueva"];
$usuario_id = $_SESSION["usuario_id"];

$stmt = $conexion->prepare("SELECT clave_hash FROM usuarios WHERE id = ?");
$stmt->bind_param("i", $usuario_id);
$stmt->execute();
$fila = $stmt->get_result()->fetch_assoc();
$stmt->close();

if (!$fila || !password_verify($actual, $fila["clave_hash"])) {
    die("La clave actual no es correcta.");
}
if (strlen($nueva) < 8) {
    die("La clave nueva debe tener al menos 8 caracteres.");
}

$hash_nuevo = password_hash($nueva, PASSWORD_DEFAULT);
$stmt = $conexion->prepare("UPDATE usuarios SET clave_hash = ? WHERE id = ?");
$stmt->bind_param("si", $hash_nuevo, $usuario_id);
$stmt->execute();
$stmt->close();

// Rotar el token despues de usarlo: cada formulario importante usa uno fresco
$_SESSION["csrf"] = bin2hex(random_bytes(32));

echo "Clave actualizada correctamente.";
$conexion->close();
exit;
}
$conexion->close();
?>
<form method="post">
    <input type="hidden" name="csrf" value="<?php echo
htmlspecialchars($_SESSION["csrf"], ENT_QUOTES, "UTF-8"); ?>">
    <input type="password" name="clave_actual" placeholder="Clave actual" required><br>
    <input type="password" name="clave_nueva" placeholder="Clave nueva (8+ caracteres)"
required><br>
    <button type="submit">Cambiar clave</button>
</form>

```

Explicación paso a paso:

- `random_bytes(32)` genera 32 bytes aleatorios criptograficamente seguros; `bin2hex()` los convierte en texto hexadecimal para meterlos en el formulario. Adivinarlo es prácticamente imposible.
- El token vive en la sesión del usuario y viaja también como campo oculto en el formulario. Al procesar, ambas copias deben coincidir.
- La verificación con `hash_equals()` es lo primero que hace el bloque POST, antes de tocar la base. Si un sitio malicioso autoenvia este formulario, no conocerá el token y la petición muere aquí.
- `hash_equals()` compara las cadenas en tiempo constante: no da pistas sobre cuantos caracteres coinciden, cerrando la puerta a ataques de medición de tiempo.
- Después de un cambio exitoso se genera un token nuevo (rotación): así cada token se usa una sola vez para acciones críticas.
- El token también se escapa con `htmlspecialchars()` al imprimirlo en el value: es un dato que viaja al HTML y el hábito no se rompe nunca.
- Con este ejercicio tu cambio de clave queda blindado contra tres ataques a la vez: CSRF (token), inyección SQL (preparadas) y robo de sesión (guardia + id de sesión).

Resumen de la unidad

- Subir archivos y seguridad van de la mano: cada archivo que entra a tu servidor es un paquete que debe revisarse antes de guardarse, como los paquetes que llegan a tu casa.
- El formulario de subida necesita `method="post"` y `enctype="multipart/form-data"`; sin el `enctype`, `$_FILES` llega vacío.
- `$_FILES` guarda por cada archivo: `name` (nombre original, no confiable), `tmp_name` (archivo temporal real), `size` (peso en bytes), `error` (0 significa todo bien) y `type` (declarado por el navegador, no confiable).
- `move_uploaded_file()` mueve el archivo temporal a su destino final y solo acepta archivos subidos por HTTP de verdad; es la única función correcta para esta tarea.
- Valida siempre en este orden: código de error, tamaño máximo, extensión con lista blanca y contenido real con `getimagesize()`.
- Usa listas blancas ("solo permito esto"), nunca listas negras ("prohibo aquello"): la lista negra siempre deja huecos.
- Renombra cada archivo con `uniqid()` para evitar colisiones, caracteres problemáticos y ataques con nombres maliciosos; el nombre original se descarta.

- En la base de datos guarda la ruta del archivo, no el archivo completo; el INSERT o UPDATE debe usar sentencias preparadas.
- XSS es inyectar código en tu página a través de datos de usuario; la defensa es `htmlspecialchars()` en TODA salida que incluya datos del usuario.
- Inyección SQL es inyectar código en tus consultas; la defensa son las sentencias preparadas en TODA consulta con datos externos.
- Regla de oro: nunca confíes en el usuario. Se el cadenero del antro: revisa todo, siempre.
- Nunca guardes contraseñas en texto plano; usa `password_hash()` con `PASSWORD_DEFAULT` al registrar y `password_verify()` al iniciar sesión.
- Un login seguro: buscar por email con preparada, verificar con `password_verify()`, regenerar la sesión, guardar lo mínimo en `$_SESSION`, redirigir con `exit`, y mensajes de error genéricos.
- CSRF te hace ejecutar acciones sin darte cuenta; la defensa es un token aleatorio en la sesión verificado con `hash_equals()`.
- Errores que ya no vas a cometer: olvidar el `enctype`, confiar en `$_FILES["type"]`, guardar con el nombre original, comparar hashes con `==`, y olvidar el `exit` después de `header()`.
- Buenas prácticas: carpeta uploads protegida, límites de tamaño en todos los niveles, HTTPS en producción, validación siempre en el servidor y PHP actualizado.

Con esta unidad cierras el ciclo: ya sabes recibir archivos como un profesional y defender tu sitio como un cadenero. En la siguiente unidad seguiras construyendo sobre esta base sólida. Sigue practicando: la seguridad no es un tema que se aprende una vez, es un hábito que se entrena todos los días.

UNIDAD 10: PROYECTO FINAL Y DESPLIEGUE

Llegaste a la última unidad del curso. Tómate un segundo para pensarlo: empezaste sin saber qué era PHP y ahora manejas variables, formularios, sesiones, bases de datos MySQL, consultas con PDO, subida de archivos y validaciones. Eso no es poca cosa. Pero falta la parte más importante, la que convierte todo ese conocimiento suelto en algo real: construir un proyecto completo de principio a fin y subirlo a internet para que cualquiera pueda verlo.

Esta unidad tiene tres clases. En la clase 28 vas a planear y construir un blog completo con registro de usuarios, inicio de sesión, publicaciones con imagen y comentarios. En la clase 29 vas a desplegarlo, es decir, vas a subirlo a un hosting real con su dominio para que tu blog exista en internet de verdad, no solo en tu computadora. Y en la clase 30 cerramos el curso, repasamos lo esencial y te muestro el mapa de lo que sigue después de PHP.

Antes de empezar, déjame explicarte por qué un proyecto completo vale más que diez tutoriales sueltos. Imagina que quieres aprender carpintería. Puedes pasar semanas practicando con tornillos: atornillar, desatornillar, conocer cada tipo de tornillo que existe. Al final sabrás mucho de tornillos, pero no sabrás hacer un mueble. En cambio, si te propones armar un librero completo, vas a usar los tornillos, sí, pero también vas a medir la madera, a cortarla, a lijarla, a ensamblar las piezas en el orden correcto y a resolver los problemas que salen cuando algo no encaja. El librero terminado te enseña algo que los tornillos solos nunca te van a enseñar: cómo se juntan todas las piezas para hacer algo que sirve.

Con la programación pasa exactamente lo mismo. Ver tutoriales sueltos es practicar con tornillos: hoy aprendes un formulario, mañana una consulta SQL, pasado una sesión. Todo eso es útil, pero mientras no juntes las piezas, no sabes construir nada completo. Este proyecto es tu librero. Vas a tomar todo lo que aprendiste en las unidades anteriores y lo vas a ensamblar en una aplicación real que funciona: un blog donde la gente se registra, publica artículos con imágenes y comenta.

Y hay una razón práctica muy poderosa para hacer esto: nadie contrata tornillos, contratan libreros. Cuando busques trabajo o clientes como desarrollador, nadie te va a pedir que recites la sintaxis de un foreach de memoria. Te van a pedir que muestres algo que hayas construido. Un blog funcionando en internet, con su dominio y su base de datos real, vale más en una entrevista que cualquier certificado. Este proyecto es la primera pieza de tu portafolio.

Qué vas a construir exactamente. Un blog personal con estas funciones: cualquier visitante puede ver la lista de publicaciones y leerlas completas. Para participar hay que registrarse con nombre, correo y contraseña. Una vez dentro, el usuario puede iniciar sesión, publicar artículos con título, contenido e imagen, ver sus propias publicaciones en un panel personal,

editarlas y eliminarlas. Además, los usuarios registrados pueden dejar comentarios en las publicaciones de otros. Suena como mucho, y lo es, pero cada pieza usa cosas que ya viste: el registro usa formularios y validación con `password_hash`, el login usa sesiones, las publicaciones usan `INSERT` y `SELECT` con PDO, las imágenes usan el upload de archivos, y los comentarios usan relaciones entre tablas.

Cómo vamos a trabajar esta unidad. No te voy a pedir que memorices nada nuevo y complicado. La estrategia es reutilizar: cada parte del blog es una versión un poco más grande de ejercicios que ya hiciste. El registro del blog es el formulario de registro de la unidad 7 con una tabla de usuarios. El login es el mismo sistema de sesiones que ya practicaste. Crear una publicación es un formulario con un campo de archivo, como el upload de la unidad 8, más un `INSERT` como los de la unidad 6. Cuando lo veas así, el proyecto deja de verse como una montaña y se convierte en una serie de pasos que ya sabes dar.

Un consejo antes de empezar a programar: escribe tú el código, no solo lo leas. Sé que da flojera y que es más rápido copiar y pegar, pero la diferencia entre leer código y escribirlo es la misma que hay entre ver a alguien armar el librero y armarlo tú. Los errores que cometes escribiendo son los que más te van a enseñar. Cuando algo no funcione, que va a pasar varias veces, respira, lee el mensaje de error con calma y revisa. Depurar es la habilidad más importante de un programador, más que memorizar sintaxis.

Otra advertencia honesta: esta unidad es larga. Es la más larga del libro y es a propósito. Un proyecto real no cabe en dos páginas. Tómate tu tiempo, trabaja por secciones y celebra cada parte que funcione: cuando el registro guarde tu primer usuario, cuando el login te deje entrar, cuando tu primera publicación aparezca en el inicio. Cada una de esas pequeñas victorias es una prueba de que ya sabes construir software.

Qué necesitas para esta unidad. Lo mismo de siempre: tu servidor local (XAMPP o el que uses), un editor de código y tu navegador. Para la clase 29 del despliegue vas a necesitar además una cuenta en un servicio de hosting. No te preocupes por eso ahora, en su momento te explicó las opciones, incluyendo algunas gratuitas para practicar. Por ahora concéntrate en construir el blog en tu computadora.

El plan de archivos que vamos a crear es este, para que lo tengas en la cabeza desde el inicio: conexión.php para conectarnos a la base de datos, funciones.php con las funciones que se repiten, registro.php y login.php para las cuentas de usuario, crear-post.php para publicar, index.php como página principal del blog, post.php para leer una publicación completa con sus comentarios, mis-posts.php como panel del autor con opciones de editar y eliminar, editar-post.php y eliminar-post.php para esas acciones, logout.php para cerrar sesión, y las carpetas includes/ para el encabezado y pie de página compartidos, y uploads/ para guardar las imágenes. No te asustes por la lista, iremos archivo por archivo.

Vamos a armar el librero. Empieza la clase 28.

Explicación detallada

Planeación del proyecto: primero el plano, luego la construcción

Ningún carpintero serio empieza a cortar madera sin un plano. En programación pasa igual: antes de escribir la primera línea de código, hay que planear. Planear no significa hacer documentos aburridos de cien páginas, significa responder tres preguntas en una hoja: qué datos voy a guardar, qué páginas va a tener mi sitio y qué puede hacer cada tipo de usuario. Con esas tres respuestas claras, programar se vuelve seguir instrucciones en lugar de improvisar.

Pregunta uno: qué datos voy a guardar. Nuestro blog necesita guardar usuarios (los que se registran), publicaciones (los artículos del blog) y comentarios (lo que la gente opina de cada artículo). Cada uno de estos es una tabla en la base de datos. Las tablas se relacionan entre sí: cada publicación pertenece a un usuario (su autor) y cada comentario pertenece a una publicación y a un usuario (quien lo escribió). Estas relaciones son el corazón del proyecto, así que vamos a diseñarlas con cuidado en la siguiente sección.

Pregunta dos: qué páginas va a tener el sitio. Ya te adelanté la lista en la introducción, pero vamos a ver qué hace cada una: index.php es la portada del blog, muestra la lista de publicaciones con su imagen, título y un extracto. registro.php y login.php son los formularios de cuenta. crear-post.php es el formulario para escribir un artículo nuevo con su imagen. post.php muestra un artículo completo con su lista de comentarios y el formulario para comentar. mis-posts.php es el panel personal donde el autor ve solo sus publicaciones con botones de editar y eliminar. editar-post.php reutiliza el formulario de crear pero con los datos ya cargados. eliminar-post.php procesa el borrado. logout.php cierra la sesión. Además tendremos includes/header.php e includes/footer.php con el diseño compartido, para no repetir el HTML del menú en cada página.

Pregunta tres: qué puede hacer cada tipo de usuario. Hay dos tipos: el visitante que no ha iniciado sesión y el usuario registrado. El visitante puede ver la lista de publicaciones y leerlas completas, pero si quiere comentar o publicar, el sistema lo manda al login. El usuario registrado puede hacer todo lo del visitante más comentar, crear publicaciones, editar sus propias publicaciones y eliminarlas. Ojo con un detalle importante de seguridad: un usuario solo puede editar o eliminar SUS publicaciones, no las de otros. Esa verificación la vamos a programar en cada acción, porque nunca debes confiar en que el usuario solo va a dar clic en sus propios botones. Alguien con malas intenciones podría escribir la dirección manualmente en el navegador, y tu código tiene que detenerlo.

Un consejo práctico de planeación: dibuja las páginas en papel antes de programarlas. No necesitas ser diseñador, basta con rectángulos que digan "aquí va el título, aquí la imagen, aquí el botón". Ese dibujo te ahorra horas de estar moviendo cosas sin rumbo en el código. Los programadores profesionales hacen esto todo el tiempo, con herramientas más elegantes, pero la idea es la misma: primero el plano, luego la construcción.

Las tablas de la base de datos: usuarios, posts y comentarios

Vamos a diseñar las tres tablas con calma, porque una base de datos bien diseñada hace que todo lo demás sea fácil, y una mal diseñada te persigue con problemas hasta el final del proyecto.

La tabla usuarios guarda a las personas registradas. Necesita: un id que sea la llave primaria autoincremental (el número único de cada usuario), nombre para mostrar quién escribe, email que debe ser único porque lo usamos para iniciar sesión, password donde guardaremos la contraseña cifrada con password_hash (nunca en texto plano, eso ya lo viste en la unidad 7), y fecha_registro para saber cuándo se unió. El email único es importante: si dos personas pudieran registrarse con el mismo correo, el login no sabría cuál es cuál. En la base de datos lo marcamos como UNIQUE y además lo verificamos en el código antes de registrar.

La tabla posts guarda las publicaciones. Necesita: id como llave primaria, usuario_id que es llave foránea apuntando a usuarios.id (así sabemos quién escribió cada artículo), título, contenido (usamos tipo TEXT porque los artículos pueden ser largos), imagen donde guardamos el nombre del archivo de imagen que se subió, y fecha_publicacion. La llave foránea usuario_id es lo que conecta cada post con su autor. Cuando borremos un usuario, habría que decidir qué pasa con sus posts, pero para este proyecto lo simple es no permitir borrar usuarios desde la interfaz, así evitamos ese problema.

La tabla comentarios guarda los comentarios. Necesita: id como llave primaria, post_id como llave foránea a posts.id (a qué artículo pertenece el comentario), usuario_id como llave foránea

a usuarios.id (quién lo escribió), comentario de tipo TEXT, y fecha. Fíjate que esta tabla tiene dos llaves foráneas porque un comentario siempre está atado a dos cosas: el artículo donde se publicó y la persona que lo escribió.

Un error típico de principiantes es querer guardar todo en una sola tabla gigante, por ejemplo meter el nombre del autor como texto dentro de cada post en lugar de usar el usuario_id. Eso parece más fácil al principio, pero se vuelve un desastre cuando el usuario cambia su nombre: tendrías que actualizar todos sus posts uno por uno. Con las llaves foráneas, el nombre vive en un solo lugar (la tabla usuarios) y todo lo demás lo referencia. Esa es la gracia de las bases de datos relacionales: cada dato vive en un solo lugar.

Otro detalle de diseño: usamos el motor InnoDB en las tablas, que es el que permite llaves foráneas de verdad en MySQL. Y usamos utf8mb4 como cotejamiento para que los acentos, las eñes y hasta los emojis se guarden bien. Si alguna vez ves signos de interrogación raros en lugar de acentos en tu sitio, el problema casi siempre es el cotejamiento.

El mapa de archivos: cómo se organiza el proyecto

Antes de programar, definamos la estructura de carpetas. Vas a crear una carpeta llamada blog/ dentro de tu servidor local (en XAMPP sería dentro de htdocs). Dentro de blog/ van los archivos PHP principales, y dos subcarpetas: includes/ para las piezas de diseño que se repiten (el encabezado con el menú y el pie de página) y uploads/ para las imágenes que suban los usuarios.

El mapa queda así: blog/conexión.php, blog/funciones.php, blog/registro.php, blog/login.php, blog/logout.php, blog/crear-post.php, blog/editar-post.php, blog/eliminar-post.php, blog/index.php, blog/post.php, blog/mis-posts.php, blog/includes/header.php, blog/includes/footer.php, y la carpeta blog/uploads/ que empieza vacía. Son catorce elementos, pero no te asustes: varios son muy cortos y varios comparten código a través de funciones.php y los includes.

La regla de oro de la organización es esta: si un pedazo de código aparece en más de un archivo, debe convertirse en una función o en un include. El encabezado del sitio con el menú de navegación aparece en todas las páginas, así que vive en includes/header.php y cada página lo incluye con require. La conexión a la base de datos se usa en casi todo, así que vive en conexión.php. Las funciones como "obtener el usuario actual" o "verificar que haya sesión" se usan en varias páginas, así que viven en funciones.php. Esta regla te ahorra un dolor de cabeza enorme cuando quieras cambiar algo: cambias el menú en un solo archivo y se actualiza en todo el sitio.

Advertencia de seguridad sobre la carpeta uploads/: es una de las partes más delicadas del proyecto. Como ahí se guardan archivos que suben los usuarios, un atacante podría intentar subir un archivo PHP disfrazado de imagen para luego ejecutarlo en tu servidor. Por eso vamos a validar la extensión del archivo, a renombrar cada imagen con un nombre único generado por nosotros (el usuario no decide el nombre final) y a guardar solo la referencia en la base de datos. Nunca confíes en el nombre original del archivo que manda el navegador.

La base de datos del blog: el SQL completo

Con el diseño claro, es hora de crear la base de datos. Abre phpMyAdmin, crea una base de datos llamada blog_db con cotejamiento utf8mb4_unicode_ci, y ejecuta el SQL que te doy en la sección de ejemplos. Vamos a repasarlo parte por parte para que entiendas cada decisión.

Primero el CREATE TABLE de usuarios: id INT AUTO_INCREMENT PRIMARY KEY, nombre VARCHAR(100) NOT NULL, email VARCHAR(150) NOT NULL UNIQUE, password VARCHAR(255) NOT NULL, fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP. El VARCHAR(255) para el password es porque password_hash genera textos largos, de unos 60 caracteres, y conviene dejar espacio de sobra. El TIMESTAMP con valor por defecto nos ahorra tener que mandar la fecha desde PHP: la base de datos la pone sola al insertar.

Luego la tabla posts: id INT AUTO_INCREMENT PRIMARY KEY, usuario_id INT NOT NULL, título VARCHAR(200) NOT NULL, contenido TEXT NOT NULL, imagen VARCHAR(255), fecha_publicacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP, y la llave foránea FOREIGN KEY (usuario_id) REFERENCES usuarios(id). Fíjate que imagen permite NULL porque podríamos decidir que la imagen sea opcional, aunque en nuestro formulario la vamos a pedir obligatoria. Dejarla opcional en la base de datos nos da flexibilidad.

Y la tabla comentarios: id INT AUTO_INCREMENT PRIMARY KEY, post_id INT NOT NULL, usuario_id INT NOT NULL, comentario TEXT NOT NULL, fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP, con sus dos llaves foráneas a posts y usuarios.

Un consejo: después de crear las tablas, inserta a mano un usuario y un post de prueba desde phpMyAdmin. Así, cuando programes la página index.php que lista las publicaciones, ya tendrás datos para ver si funciona, en lugar de estar adivinando con la página vacía. Los programadores llamamos a esto "datos semilla", y es una práctica muy común.

Error típico número uno en esta etapa: crear la base de datos con el cotejamiento latin1 por defecto y después pelearse con los acentos. Si ya te pasó, no hay drama: en phpMyAdmin puedes cambiar el cotejamiento de la base de datos y las tablas a utf8mb4_unicode_ci en la pestaña de operaciones. Error típico número dos: olvidar las llaves foráneas y después no poder hacer los JOINS. Si ya creaste las tablas sin ellas, puedes agregarlas después con ALTER TABLE, pero es más fácil hacerlo bien desde el inicio.

Registro y login: reutilizando lo que ya aprendiste

El registro y el login del blog son los mismos que practicaste en la unidad 7, adaptados a nuestra tabla usuarios. Vamos a repasar el flujo completo porque es el sistema de seguridad de todo el proyecto y tiene que quedar perfecto.

El registro funciona así: el usuario llena nombre, email y contraseña en registro.php. Al enviar el formulario, el código valida que ningún campo venga vacío, que el email tenga formato válido con filter_var, y que la contraseña tenga al menos 6 caracteres. Después verifica que el email no esté ya registrado con un SELECT. Si todo está bien, cifra la contraseña con password_hash y hace el INSERT. Finalmente, inicia la sesión automáticamente y redirige al index. Iniciar la sesión justo después del registro es un detalle de experiencia de usuario: obligar a la persona a ir al login después de registrarse es una fricción innecesaria.

El login funciona así: el usuario escribe email y contraseña en login.php. El código busca el email en la base de datos con un SELECT. Si no existe, muestra error. Si existe, compara la contraseña escrita con la guardada usando password_verify. Si coincide, guarda el id y el nombre del usuario en la sesión con \$_SESSION y redirige al index. Si no coincide, muestra un mensaje de error genérico.

Ojo con el mensaje de error del login: debe ser genérico, algo como "Correo o contraseña incorrectos", sin decir cuál de los dos falló. Si dices "ese correo no existe", le estás regalando a un atacante la lista de correos registrados: puede probar correos hasta que el mensaje cambie. Es un detalle pequeño de seguridad que los profesionales siempre aplican.

Las sesiones son las que mantienen al usuario "adentro" mientras navega. En cada página protegida (crear-post.php, mis-posts.php, etc.) la primera línea de lógica verifica si existe \$_SESSION['usuario_id']. Si no existe, redirige al login con header('Location: login.php') seguido de exit. El exit después del header es obligatorio: sin él, PHP seguiría ejecutando el resto de la página aunque ya mandó la redirección. Es un error clásico que deja páginas protegidas expuestas.

El logout es simple: logout.php destruye la sesión con session_destroy() y redirige al index. Parece trivial, pero no olvides ofrecer el enlace de "Cerrar sesión" en el menú cuando el

usuario está adentro, y los enlaces de "Registrarse" e "Iniciar sesión" cuando está afuera. Ese menú dinámico vive en `includes/header.php` y usa la sesión para decidir qué mostrar.

Crear publicaciones con imagen: el upload paso a paso

Crear una publicación combina dos cosas que ya conoces: un formulario y un INSERT, más la subida de la imagen. Vamos a desmenuzar el proceso porque el upload es la parte donde más errores cometen los principiantes.

El formulario de `crear-post.php` lleva el atributo `enctype="multipart/form-data"`, sin el cual el archivo nunca llega al servidor. Es el error número uno: todo el código PHP está perfecto pero el formulario no tiene el `enctype` y `$_FILES` llega vacío. Si tu upload "no funciona" y no sabes por qué, revisa el `enctype` primero. Los campos son título (texto), contenido (un textarea grande) e imagen (`input type="file"` que acepta imágenes).

Cuando se envía el formulario, el archivo llega a PHP en el arreglo `$_FILES['imagen']`, que trae el nombre original, el tipo, el tamaño, un código de error y la ruta temporal donde PHP guardó el archivo. El flujo de validación es este: primero revisar que no haya error de subida (`$_FILES['imagen']['error']` debe ser 0), luego verificar que la extensión sea una imagen permitida (jpg, jpeg, png, gif, webp) usando `pathinfo`, luego verificar el tamaño (vamos a poner un límite de 2 MB para no llenar el servidor), y opcionalmente verificar con `getimagesize` que de verdad sea una imagen y no un archivo disfrazado.

Después de validar, generamos un nombre único para el archivo. Usamos algo como `uniqid('img_')` más la extensión, por ejemplo `img_64f2a1b3c8d9e.jpg`. Esto evita dos problemas: que dos usuarios suban archivos con el mismo nombre y se sobrescriban, y que un nombre malicioso con caracteres raros cause problemas. Luego movemos el archivo de su ubicación temporal a nuestra carpeta `uploads/` con `move_uploaded_file`, que es la única función segura para esto porque verifica que el archivo realmente se subió por HTTP.

Finalmente hacemos el INSERT en la tabla `posts` con el `usuario_id` de la sesión, el título, el contenido y el nombre final de la imagen. Si el INSERT falla después de haber movido el archivo, conviene borrar el archivo huérfano con `unlink` para no acumular basura, aunque para este proyecto lo manejaremos de forma simple.

Advertencia importante: la carpeta `uploads/` debe tener permisos de escritura para el servidor web. En tu computadora con XAMPP casi siempre funciona sin hacer nada, pero en el hosting real a veces hay que ajustar los permisos a 755 en la carpeta. Lo veremos en la parte de despliegue. Y recuerda: nunca muestres el nombre original del archivo al usuario ni lo uses para guardar, el nombre final siempre lo decides tú con `uniqid`.

El inicio del blog: listar publicaciones

El `index.php` es la portada del blog y su trabajo es mostrar la lista de publicaciones de la más reciente a la más antigua. La consulta es un `SELECT` con `JOIN` a la tabla usuarios para traer también el nombre del autor, ordenado por fecha descendente con `ORDER BY fecha_publicacion DESC`. El `JOIN` lo veremos a fondo en la sección de detalle, pero la idea es simple: en una sola consulta traemos los datos del post y el nombre de quien lo escribió.

Cada publicación se muestra como una tarjeta con su imagen (la etiqueta `img` apuntando a `uploads/` más el nombre del archivo), el título como enlace a `post.php?id=...`, el nombre del autor, la fecha y un extracto del contenido. El extracto lo generamos con `substr` del contenido, unos 150 caracteres seguidos de puntos suspensivos, para no mostrar el artículo completo en la portada. La fecha la formateamos bonita con `date()` a partir del `TIMESTAMP`.

Un detalle de seguridad que aplica en todo el listado: cada dato que venga de la base de datos y se muestre en HTML debe pasar por `htmlspecialchars`. Los títulos y contenidos los escribieron usuarios, y un usuario malicioso podría escribir código JavaScript en su publicación para robar sesiones de otros visitantes. Eso se llama XSS y `htmlspecialchars` lo previene convirtiendo los caracteres especiales en entidades HTML. Acostúmbrate: todo lo que imprimas con `echo` y venga de la base de datos o de un formulario, pasa por `htmlspecialchars`.

Si el blog no tiene publicaciones todavía, mostramos un mensaje amable invitando a ser el primero en publicar, en lugar de una página vacía que parece rota. Los detalles de experiencia de usuario como este son los que separan un proyecto de estudiante de un proyecto profesional.

La página de detalle: una publicación con sus comentarios

La página `post.php` recibe el id de la publicación por GET, por ejemplo `post.php?id=5`. Lo primero que hace es validar que el id sea un número válido; si no lo es, muestra un error o redirige al `index`. Nunca uses un valor de `$_GET` directamente en una consulta sin validarlo, aunque uses consultas preparadas, porque un id inválido puede causar comportamientos raros.

Con el id validado, hacemos un `SELECT` del post con `JOIN` a usuarios para traer el título, contenido, imagen, fecha y nombre del autor. Si la consulta no devuelve nada (porque el id no existe), mostramos un mensaje de "publicación no encontrada" en lugar de una página rota. Después hacemos otro `SELECT` para los comentarios de ese post, también con `JOIN` a usuarios para mostrar quién comentó, ordenados por fecha ascendente (los más antiguos primero, como en una conversación normal).

Debajo del artículo mostramos la lista de comentarios, cada uno con el nombre de quien comentó, la fecha y el texto. Y al final, el formulario para dejar un comentario. Aquí viene una decisión importante: solo los usuarios registrados pueden comentar. Si no hay sesión, en lugar del formulario mostramos un mensaje que invita a iniciar sesión con un enlace al login. Esto evita el spam de comentarios anónimos y nos obliga a usar las sesiones, que es justo lo que queremos practicar.

Cuando se envía un comentario, el código verifica la sesión otra vez (porque alguien podría mandar el formulario directamente sin pasar por la página), valida que el comentario no venga vacío, y hace el INSERT en la tabla comentarios con el post_id, el usuario_id de la sesión y el texto. Después redirige a la misma página `post.php?id=...` para evitar el problema del reenvío del formulario al recargar. Esa técnica se llama POST-Redirect-GET y ya la usaste en unidades anteriores: después de procesar un POST, siempre redirige con header en lugar de mostrar HTML directamente.

Vamos a explicar el JOIN de forma simple, porque es el concepto nuevo más importante de esta unidad. Imagina dos listas en papel: una con los posts (cada uno con su usuario_id) y otra con los usuarios (cada uno con su id y nombre). El JOIN es como poner las dos listas lado a lado y trazar líneas entre el usuario_id del post y el id del usuario que coincide. El resultado es una tabla combinada donde cada post ya trae el nombre de su autor pegado. En SQL se escribe: `SELECT posts.*, usuarios.nombre FROM posts JOIN usuarios ON posts.usuario_id = usuarios.id`. La parte del ON es la regla para trazar las líneas: una donde el usuario_id del post sea igual al id del usuario. Sin el JOIN tendrías que hacer una consulta por cada post para averiguar el nombre del autor, lo cual es lento y torpe. Con el JOIN, una sola consulta trae todo.

Mi panel: editar y eliminar mis publicaciones

La página `mis-posts.php` muestra solo las publicaciones del usuario que inició sesión, con un SELECT filtrado por `WHERE usuario_id = el id de la sesión`. Cada publicación aparece con botones o enlaces de Editar y Eliminar. Esta página es el "escritorio" del autor y es muy satisfactoria de construir porque es la primera vez que el usuario siente que el sistema es suyo.

Editar funciona en dos pasos. Primero, `editar-post.php` recibe el id por GET, verifica que la publicación exista Y que pertenezca al usuario de la sesión con una consulta que filtra por id y usuario_id al mismo tiempo. Si no pertenece al usuario, muestra error y no deja continuar. Esta verificación es crítica: sin ella, cualquiera podría editar los posts de otros cambiando el id en la dirección del navegador. Después muestra el formulario con los valores actuales precargados en los campos value y dentro del textarea. Al enviar, procesa el UPDATE. La imagen en la

edición la vamos a manejar de forma simple: si el usuario sube una imagen nueva, se reemplaza (y se borra la anterior con unlink); si no sube nada, se conserva la actual.

Eliminar también lleva su verificación de propiedad antes del DELETE. Además, al eliminar un post debemos borrar sus comentarios primero (por la llave foránea) y borrar el archivo de imagen de la carpeta uploads/ para no dejar basura. El orden es: verificar propiedad, borrar comentarios del post, borrar el archivo de imagen, borrar el post. Una buena práctica es pedir confirmación con JavaScript (un simple confirm) antes de eliminar, porque no hay forma de deshacer un DELETE. En la base de datos real del hosting, lo borrado está borrado.

Un consejo sobre los enlaces de eliminar: usa un formulario pequeño con método POST en lugar de un simple enlace GET para la acción de eliminar. Los enlaces GET se pueden activar sin querer (un clic accidental, un robot de buscador que sigue enlaces) y provocar borrados no deseados. Con POST y un botón de confirmación reduces mucho ese riesgo. Es un detalle que distingue al código cuidadoso.

Organización del código: includes, uploads y funciones reutilizables

A estas alturas tu carpeta blog/ tiene varios archivos y es buen momento para hablar de orden. Un proyecto desorganizado funciona igual que uno organizado, pero cuando algo falla, en el desorganizado tardas el triple en encontrar el problema. Y cuando quieras agregar funciones nuevas, el organizado te lo pone fácil.

La carpeta includes/ guarda header.php y footer.php. El header abre el HTML, incluye el menú de navegación que cambia según haya sesión o no, y abre el contenido principal. El footer cierra el contenido, pone el pie de página y cierra el HTML. Cada página del blog empieza con require 'includes/header.php' y termina con require 'includes/footer.php'. Así el diseño es consistente y si quieres cambiar el menú, lo cambias una vez.

El archivo funciones.php concentra la lógica que se repite: una función usuarioActual() que devuelve los datos del usuario de la sesión o null, una función requiereLogin() que redirige al login si no hay sesión, una función esDueno(\$postId) que verifica si una publicación pertenece al usuario actual, y funciones pequeñas como formatearFecha() o extracto(). Cada vez que sientas la tentación de copiar y pegar un bloque de código de un archivo a otro, detente y conviértelo en función. La repetición de código es la raíz de la mitad de los errores en proyectos reales: cambias algo en un lugar y se te olvida cambiarlo en los otros cinco donde lo pegaste.

La carpeta uploads/ empieza vacía y se llena con las imágenes. Dos reglas para ella: nunca subas ahí archivos PHP del proyecto (solo imágenes generadas por el sistema) y en el despliegue verifica sus permisos. Algunos hostings permiten protegerla con un archivo

.htaccess que impide ejecutar scripts dentro de ella, lo cual es una excelente medida de seguridad extra que puedes investigar cuando publiques.

El archivo conexión.php merece mención especial: es el único lugar donde aparecen el servidor, el usuario, la contraseña y el nombre de la base de datos. Toda la aplicación lo incluye y nadie más necesita saber esos datos. Esto no es solo orden, es seguridad: cuando despliegues, solo tendrás que editar este archivo para apuntar al hosting, y si algún día subes tu código a un repositorio público, este es el archivo que debes proteger para no regalar tus credenciales.

Qué es el despliegue: tu blog sale a internet

Hasta ahora tu blog vive en tu computadora. Solo tú puedes verlo, escribiendo localhost en tu navegador. El despliegue es el proceso de mudar tu blog a una computadora que está encendida las 24 horas, conectada a internet y preparada para recibir visitas: eso es un hosting. Y para que la gente lo encuentre sin memorizar números raros, le pones un nombre fácil: eso es un dominio, como miblog.com.

La analogía clásica es la de rentar un local para un negocio. Tu computadora es como cocinar en tu casa: está bien para practicar, pero ningún cliente puede entrar. El hosting es rentar un local en una avenida transitada: pagas una renta mensual o anual, te dan un espacio con luz, agua y seguridad (el servidor configurado, la conexión a internet, los respaldos), y tú solo llevas tus muebles, o sea, tus archivos y tu base de datos. El dominio es el letrero luminoso con el nombre de tu negocio: sin él, los clientes tendrían que llegar con las coordenadas GPS (la dirección IP del servidor, algo como 192.168.1.45), y nadie se aprende eso.

Qué necesitas contratar. Un plan de hosting compartido es suficiente para este proyecto y es lo más barato: tu sitio comparte el servidor con otros sitios, como un edificio de departamentos donde cada quien tiene el suyo pero comparten la estructura. Busca uno que incluya PHP, MySQL, phpMyAdmin y acceso FTP, que es prácticamente cualquier hosting comercial. El dominio se compra por separado, normalmente por un año, y luego lo "apuntas" al hosting siguiendo las instrucciones del proveedor. Muchos proveedores venden paquetes que incluyen dominio y hosting juntos para simplificar.

Opciones para practicar sin gastar. Si solo quieres probar el proceso de despliegue sin pagar, hay hostings gratuitos con PHP y MySQL que sirven para aprender, aunque son más lentos y a veces muestran publicidad. Otra opción es pedir un periodo de prueba en un hosting de pagó. Mi recomendación: práctica una vez en un hosting gratuito para entender el proceso sin presión, y cuando tu blog esté listo de verdad, invierte en un hosting de pagó barato y un dominio propio. Un dominio propio (tunombre.com) se ve infinitamente más profesional en tu portafolio que un subdominio gratuito.

El despliegue tiene cuatro pasos grandes: exportar tu base de datos local, subir tus archivos por FTP, crear la base de datos en el hosting e importar tu SQL, y ajustar el archivo conexión.php con los datos nuevos. Vamos uno por uno con todo el detalle.

Paso 1: exportar la base de datos desde phpMyAdmin

Tu base de datos local blog_db tiene las tablas y probablemente algunos datos de prueba. Para llevarla al hosting necesitas un archivo .sql con todas las instrucciones para recrearla. Eso se llama exportar, y phpMyAdmin lo hace en un par de clics.

Abre phpMyAdmin en tu computadora, selecciona la base de datos blog_db en la lista de la izquierda (importante: selecciona la base de datos, no una tabla suelta), y haz clic en la pestaña "Exportar". Elige el método "Rápido" y el formato "SQL", y presiona "Continuar". Se descargará un archivo llamado blog_db.sql. Guárdalo en un lugar seguro porque lo vas a usar en el paso 3.

Abre ese archivo con tu editor de texto y échale un vistazo. Verás sentencias CREATE TABLE para cada tabla y sentencias INSERT con tus datos. Es exactamente el mismo SQL que escribiste para crear la base de datos, más los datos que metiste después. Entender que una base de datos se puede representar como un archivo de texto con instrucciones es un concepto poderoso: significa que puedes respaldarla, moverla y versionarla como cualquier archivo.

Decisión importante: ¿exportas con tus datos de prueba o limpias primero? Para tu primera práctica de despliegue, deja los datos de prueba, así puedes verificar que todo funciona viendo contenido real. Para el despliegue "de verdad" de tu portafolio, es mejor limpiar las tablas de prueba y empezar con la base vacía pero con tu cuenta de usuario creada, o incluso empezar totalmente de cero y registrarte desde el sitio ya publicado. Un blog de portafolio con publicaciones de prueba que dicen "asdf" se ve descuidado.

Error típico: exportar solo una tabla en lugar de toda la base de datos, porque se hizo clic en la tabla antes de ir a la pestaña Exportar. Si tu archivo .sql solo tiene un CREATE TABLE, repite el proceso seleccionando la base de datos completa.

Paso 2: subir los archivos por FTP con FileZilla

FTP es el protocolo para transferir archivos entre tu computadora y el servidor del hosting, como un servicio de mudanza para tus archivos. FileZilla es un programa gratuito que hace esto con una interfaz de dos paneles: a la izquierda tus archivos locales, a la derecha los del servidor. Arrastras de un lado al otro y listo.

Primero consigue tus datos de acceso FTP: el proveedor de hosting te los da en el panel de control o por correo cuando contratas. Son cuatro datos: servidor (algo como ftp.tudominio.com o una dirección IP), usuario, contraseña y a veces el puerto (normalmente 21). Guárdalos bien.

Abre FileZilla y conéctate con esos datos. Al conectar verás del lado derecho las carpetas del servidor. Busca la carpeta pública, que según el hosting se llama public_html, htdocs o www. Esa carpeta es el equivalente a tu htdocs local: todo lo que pongas ahí será visible en internet. Si pones tus archivos en otra carpeta, el sitio no se verá.

Ahora sube tu proyecto: del lado izquierdo navega hasta tu carpeta blog/ local, selecciónala completa y arrástrala a public_html. FileZilla empezará a subir archivo por archivo. La primera vez tarda unos minutos según tu conexión. Cuando termine, tu sitio debería estar en tudominio.com/blog/. Si prefieres que el blog sea la página principal del dominio (tudominio.com directamente), sube el contenido de la carpeta blog (los archivos, no la carpeta) directo a public_html.

Dos advertencias críticas para este paso. Primera: sube también la carpeta uploads/ aunque esté vacía, porque el código espera que exista para guardar las imágenes. FileZilla a veces no crea carpetas vacías al arrastrar, así que créala a mano del lado del servidor con clic derecho y "Crear directorio". Segunda: revisa que los nombres de archivo conserven sus mayúsculas y minúsculas exactas. En tu Windows local, Registro.php y registro.php son el mismo archivo; en el servidor Linux del hosting son dos archivos distintos. Si tu código incluye registro.php pero el archivo se llama Registro.php, en local funciona y en el hosting falla. Este es el error de despliegue más común del mundo y el más difícil de detectar porque "en mi computadora sí funcionaba".

Paso 3: crear la base de datos en el hosting e importar tu SQL

Los archivos ya están arriba, pero sin base de datos el blog no funciona. En el hosting tienes que crear una base de datos nueva y llenarla con tu archivo .sql. El proceso varía un poco según el proveedor, pero la idea siempre es la misma.

Entra al panel de control de tu hosting (normalmente cPanel) y busca la sección de bases de datos MySQL. Ahí creas una base de datos nueva: elige un nombre, por ejemplo `tuusuario_blog`. Luego creas un usuario de MySQL con una contraseña segura y lo asocias a la base de datos con todos los privilegios. Anota en un papel estos tres datos: nombre de la base de datos, usuario y contraseña. Los vas a necesitar en el paso 4. El servidor de la base de datos casi siempre es `localhost`, igual que en tu computadora, porque la base de datos vive en la misma máquina que tu sitio.

Ahora abre el phpMyAdmin del hosting (hay un enlace en el cPanel), selecciona tu base de datos nueva y ve a la pestaña "Importar". Elige tu archivo `blog_db.sql` y presiona continuar. Si todo sale bien, verás tus tres tablas creadas con sus datos. Si el archivo es muy grande y la importación falla a la mitad, prueba exportar de nuevo sin los datos de prueba o pide ayuda al soporte del hosting; para nuestro blog pequeño esto casi nunca pasa.

Verificación rápida: en el phpMyAdmin del hosting, haz clic en la tabla usuarios y confirma que tus datos están ahí. Si ves tus usuarios de prueba, la importación funcionó. Este paso de verificar te ahorra confusiones después: si el sitio falla, ya sabes que la base de datos está bien y el problema está en otro lado.

Error típico: importar el `.sql` en la base de datos equivocada o sin haber seleccionado ninguna base de datos, lo que crea las tablas en un lugar donde tu sitio no las busca. Siempre verifica que el nombre de la base de datos seleccionada arriba sea el correcto antes de importar.

Paso 4: ajustar conexión.php con los datos del hosting

Tu archivo `conexión.php` todavía tiene los datos de tu computadora: servidor `localhost`, usuario `root`, contraseña vacía y base de datos `blog_db`. En el hosting esos datos son diferentes, así que tienes que actualizarlos. Edita el archivo en tu computadora, cambia los cuatro valores por los que anotaste en el paso 3, guárdalo y súbelo por FTP sobrescribiendo el del servidor.

Este es el momento de entender por qué centralizamos la conexión en un solo archivo: si hubieras escrito los datos de conexión en cada página, tendrías que editar catorce archivos y seguro se te olvidaría alguno. Con `conexión.php` es un solo cambio y todo el sitio se conecta a la base de datos nueva.

Después de subir el `conexión.php` actualizado, abre tu sitio en el navegador: `tudominio.com/blog/`. Si ves la portada del blog con tus publicaciones, felicidades, el despliegue funcionó. Prueba el flujo completo: regístrate con una cuenta nueva, inicia sesión, crea una publicación con imagen y deja un comentario. Si todo eso funciona en el hosting, tu aplicación está viva en internet de verdad.

Consejo de seguridad para este paso: nunca dejes el archivo .sql de respaldo dentro de public_html. Si alguien adivina la dirección tudominio.com/blog_db.sql, se descarga toda tu base de datos con los correos y las contraseñas cifradas de tus usuarios. Guarda los respaldos en tu computadora o en una carpeta fuera de la zona pública del servidor. Lo mismo aplica para cualquier archivo con credenciales que no sea conexión.php.

Errores típicos del despliegue y cómo resolverlos

Vamos a repasar los errores que casi todos cometen en su primer despliegue, para que cuando te topes con uno no entres en pánico. Todos tienen solución y todos te enseñan algo.

Error uno: "página en blanco" al abrir el sitio. Casi siempre es un error de PHP que el hosting oculta por seguridad. Para diagnosticarlo, revisa el archivo de registro de errores del hosting (en cPanel hay una sección de errores) o activa temporalmente la muestra de errores. Las causas más comunes son un error de sintaxis que en local no notaste o una ruta de include incorrecta.

Error dos: "Access denied" o "no se puede conectar a la base de datos". Significa que los datos de conexión.php están mal. Revisa letra por letra el nombre de la base de datos, el usuario y la contraseña. Ojo: en muchos hostings el nombre de la base de datos y el usuario llevan un prefijo automático, por ejemplo tuusuario_blog en lugar de blog. Copia los nombres exactos desde el cPanel, no los escribas de memoria.

Error tres: las imágenes no se ven o no se pueden subir. Primero revisa que la carpeta uploads/ exista en el servidor y tenga permisos de escritura (755 para la carpeta suele bastar). En FileZilla puedes cambiar los permisos con clic derecho sobre la carpeta y "Permisos de archivo". Segundo, revisa que las rutas en tu código sean relativas y no absolutas de tu computadora: uploads/foto.jpg funciona en todos lados, pero C:/xampp/htdocs/blog/uploads/foto.jpg solo funciona en tu máquina.

Error cuatro: "funciona en local pero en el hosting una página dice no encontrada". El culpable casi siempre son las mayúsculas y minúsculas en los nombres de archivo, como ya te advertí. Revisa que cada require e include use exactamente el mismo nombre que el archivo real. Una forma de prevenirlo desde el inicio es usar siempre minúsculas con guiones en los nombres de archivo, cómo crear-post.php.

Error cinco: los acentos se ven como símbolos raros en el hosting aunque en local se veían bien. Es el tema del cotejamiento: verifica que la base de datos del hosting use utf8mb4 y que tu conexión PDO incluya el charset utf8mb4 en el DSN. Si importaste el .sql y el cotejamiento se heredó bien, con el charset en la conexión suele bastar.

Error seis: el sitio se ve pero "a medias", sin diseño o con enlaces rotos. Revisa las rutas de tus CSS e imágenes en el header: si usaste rutas absolutas como `/blog/estilos.css` y en el hosting el blog está directo en la raíz, las rutas cambian. Por eso conviene definir una constante con la URL base del sitio en un archivo de configuración y usarla para construir todos los enlaces.

La regla general para depurar en el hosting es: reproduce el error, lee el mensaje exacto, y ataca la causa más probable primero (conexión, rutas, mayúsculas, permisos). El 90 por ciento de los errores de despliegue caen en esas cuatro categorías.

Checklist final antes de publicar

Antes de presumir tu blog en redes o ponerlo en tu portafolio, pasa esta lista de verificación. Es lo que hacen los profesionales antes de cada lanzamiento, adaptado a tu proyecto.

- El sitio abre correctamente en el dominio final, sin "localhost" en ninguna dirección.
- El registro crea usuarios nuevos y el login permite entrar y salir sin errores.
- Se puede crear una publicación con imagen y aparece en el inicio.
- Los comentarios funcionan: se publican y se muestran con el nombre del autor.
- El panel "mis publicaciones" muestra solo las del usuario actual, y editar y eliminar funcionan con su confirmación.
- Un usuario no puede editar ni eliminar publicaciones de otro (pruébalo cambiando el id en la dirección).
- Las imágenes subidas se ven correctamente y la carpeta uploads/ tiene los permisos adecuados.
- No hay archivos de respaldo .sql ni archivos de prueba dentro de las carpetas públicas.
- El archivo conexión.php tiene los datos del hosting y funciona.
- Los acentos y eñes se ven bien en todo el sitio.
- Probaste el sitio en tu celular, porque la mitad de tus visitas llegarán desde ahí.

Si algo de la lista falla, no lo dejes "para después". El "después" en proyectos publicados se convierte en "nunca", y un portafolio con detalles rotos dice más de ti que uno pequeño pero impecable. Mejor un blog sencillo que funcione perfecto que uno ambicioso lleno de errores.

Próximos pasos: el mapa después de este libro

Terminar este libro no significa terminar de aprender, significa que ya tienes los cimientos para aprender todo lo demás por tu cuenta. Vamos a trazar el mapa de lo que sigue, en orden de prioridad.

Primero: Programación Orientada a Objetos (POO) en PHP. Hasta ahora programaste de forma procedural: funciones sueltas y código secuencial. La POO organiza el código en clases y objetos, que es como trabajan los proyectos grandes y cómo funcionan los frameworks modernos. Con la base que ya tienes, la POO te tomará unas semanas y te abrirá la puerta a todo lo demás. En lucasegaacademy.com encontrarás material para seguir.

Segundo: un framework PHP, y el rey es Laravel. Un framework es como una caja de herramientas profesional: trae soluciones ya hechas para el login, las rutas, la base de datos y la seguridad, para que tú te concentres en lo único de tu proyecto. Todo lo que hiciste a mano en este blog (sesiones, validaciones, uploads) Laravel lo hace en minutos. Aprenderlo con tu base actual es totalmente alcanzable y es lo que más piden los empleos de PHP.

Tercero: APIs REST. Es la forma en que las aplicaciones modernas se comunican: tu PHP devuelve datos en formato JSON y una aplicación de celular o un frontend en JavaScript los consume. Con lo que sabes de PHP y MySQL, crear tu primera API es un paso natural y muy valioso en el mercado.

Cuarto: JavaScript para el frontend. Tu blog actual recarga la página en cada acción. Con JavaScript puedes hacer que los comentarios aparezcan sin recargar, validar formularios al instante y crear interfaces más vivas. PHP y JavaScript son la pareja clásica del desarrollo web: PHP en el servidor, JavaScript en el navegador.

Cómo seguir aprendiendo sin estancarte. La regla es simple: un proyecto nuevo por cada tema nuevo. Aprendes POO reescribiendo este blog con clases. Aprendes Laravel construyendo el blog otra vez en Laravel y comparando. Aprendes APIs convirtiendo tu blog en una API que alimenta una app. Cada vuelta al mismo proyecto con herramientas nuevas te muestra cuánto avanzaste, y al final tendrás un portafolio con la evolución completa.

Y para armar portafolio de verdad: publica cada proyecto en internet como hiciste con este blog, escribe una descripción breve de qué hace y qué tecnologías usa, y guarda el código ordenado. Tres o cuatro proyectos publicados y funcionales valen más que veinte ejercicios a medias. Tu blog de este curso es el proyecto número uno. Ya empezaste.

Ejemplos paso a paso

En esta sección tienes el código completo del blog, archivo por archivo, con explicación línea por línea. Te recomiendo escribir cada archivo en tu editor mientras lees la explicación, en el orden en que aparecen aquí, porque cada uno se apoya en los anteriores. Al final tendrás el proyecto completo funcionando en tu computadora.

Ejemplo 1: el SQL de las tres tablas

Este es el script que crea la base de datos completa. Ejecútalo en phpMyAdmin en la pestaña SQL, con la base de datos blog_db seleccionada.

```
CREATE TABLE usuarios (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    email VARCHAR(150) NOT NULL UNIQUE,  
    password VARCHAR(255) NOT NULL,  
    fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;  
  
CREATE TABLE posts (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    usuario_id INT NOT NULL,  
    titulo VARCHAR(200) NOT NULL,  
    contenido TEXT NOT NULL,  
    imagen VARCHAR(255),  
    fecha_publicacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (usuario_id) REFERENCES usuarios(id)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;  
  
CREATE TABLE comentarios (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    post_id INT NOT NULL,  
    usuario_id INT NOT NULL,  
    comentario TEXT NOT NULL,  
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (post_id) REFERENCES posts(id),  
    FOREIGN KEY (usuario_id) REFERENCES usuarios(id)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Explicación línea por línea. Cada CREATE TABLE define una tabla con sus columnas. El id INT AUTO_INCREMENT PRIMARY KEY es el identificador único que se genera solo. En usuarios, el email lleva UNIQUE para que no se repita, y el password es VARCHAR(255)

porque el hash cifrado es largo. La fecha_registro usa TIMESTAMP con DEFAULT CURRENT_TIMESTAMP para que la base de datos ponga la fecha sola. En posts, usuario_id es la llave foránea que conecta con usuarios(id), contenido es TEXT porque los artículos son largos, e imagen guarda solo el nombre del archivo. En comentarios hay dos llaves foráneas: post_id apunta al artículo y usuario_id a quien comentó. El ENGINE=InnoDB permite usar llaves foráneas de verdad, y el CHARSET=utf8mb4 guarda bien acentos y eñes.

Ejemplo 2: conexión.php y funciones.php

Estos dos archivos son la base de todo el proyecto. La conexión centraliza los datos de la base de datos y las funciones evitan repetir código.

```
<?php
$servidor = "localhost";
$usuario_db = "root";
$clave_db = "";
$nombre_db = "blog_db";

try {
    $pdo = new PDO(
        "mysql:host=$servidor;dbname=$nombre_db;charset=utf8mb4",
        $usuario_db,
        $clave_db
    );
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    die("Error de conexion: " . $e->getMessage());
}
?>
```

Explicación: las primeras cuatro líneas guardan los datos de conexión en variables, para cambiarlos en un solo lugar cuando despliegues. El bloque try crea el objeto PDO con el DSN que incluye host, base de datos y charset utf8mb4. La línea setAttribute hace que PDO lance excepciones cuando algo falle, en lugar de fallar en silencio, lo cual es vital para detectar errores. El catch atrapa cualquier problema de conexión y detiene el programa con un mensaje claro. Este archivo se incluye con require en todas las páginas que usan la base de datos.

```

<?php
session_start();

function usuarioActual() {
    if (isset($_SESSION["usuario_id"])) {
        return [
            "id" => $_SESSION["usuario_id"],
            "nombre" => $_SESSION["usuario_nombre"]
        ];
    }
    return null;
}

function requiereLogin() {
    if (!isset($_SESSION["usuario_id"])) {
        header("Location: login.php");
        exit;
    }
}

function esDueno($pdo, $post_id, $usuario_id) {
    $stmt = $pdo->prepare("SELECT id FROM posts WHERE id = ? AND usuario_id = ?");
    $stmt->execute([$post_id, $usuario_id]);
    return $stmt->fetch() !== false;
}

function extracto($texto, $largo = 150) {
    $texto = strip_tags($texto);
    if (strlen($texto) > $largo) {
        return substr($texto, 0, $largo) . "...";
    }
    return $texto;
}

function formatearFecha($fecha) {
    return date("d/m/Y H:i", strtotime($fecha));
}
?>

```

Explicación: `session_start()` va primero porque todas las funciones trabajan con la sesión. La función `usuarioActual` devuelve los datos del usuario en sesión o `null` si no hay nadie adentro, así las páginas pueden decidir qué mostrar. La función `requiereLogin` es el guardián de las páginas protegidas: si no hay sesión, redirige al login y detiene todo con `exit`. La función `esDueno` verifica en la base de datos que una publicación pertenezca al usuario actual,

buscando por id y usuario_id al mismo tiempo; devuelve verdadero solo si encuentra la fila. La función `extracto` quita etiquetas HTML con `strip_tags` y corta el texto al largo indicado agregando puntos suspensivos, para las tarjetas del inicio. La función `formatearFecha` convierte el `TIMESTAMP` de la base de datos a un formato legible como 19/09/2026 10:30.

Ejemplo 3: registro.php

El formulario de registro con todas sus validaciones. Observa cómo cada validación protege un aspecto distinto.

```
<?php
require "conexion.php";
require "funciones.php";

$errores = [];

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $nombre = trim($_POST["nombre"]);
    $email = trim($_POST["email"]);
    $clave = $_POST["password"];

    if ($nombre === "" || $email === "" || $clave === "") {
        $errores[] = "Todos los campos son obligatorios.";
    }
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $errores[] = "El correo no tiene un formato valido.";
    }
    if (strlen($clave) < 6) {
        $errores[] = "La contraseña debe tener al menos 6 caracteres.";
    }

    if (empty($errores)) {
        $stmt = $pdo->prepare("SELECT id FROM usuarios WHERE email = ?");
        $stmt->execute([$email]);
        if ($stmt->fetch()) {
            $errores[] = "Ese correo ya esta registrado.";
        }
    }

    if (empty($errores)) {
        $hash = password_hash($clave, PASSWORD_DEFAULT);
        $stmt = $pdo->prepare("INSERT INTO usuarios (nombre, email, password) VALUES
        (?, ?, ?)");
        $stmt->execute([$nombre, $email, $hash]);
    }
}
```

```

        $_SESSION["usuario_id"] = $pdo->lastInsertId();
        $_SESSION["usuario_nombre"] = $nombre;
        header("Location: index.php");
        exit;
    }
}
require "includes/header.php";
?>

<h2>Crear cuenta</h2>

<?php foreach ($errores as $error): ?>
    <p class="error"><?php echo htmlspecialchars($error); ?></p>
<?php endforeach; ?>

<form method="POST" action="registro.php">
    <label>Nombre:</label>
    <input type="text" name="nombre" required>

    <label>Correo:</label>
    <input type="email" name="email" required>

    <label>Contraseña:</label>
    <input type="password" name="password" required>

    <button type="submit">Registrarme</button>
</form>

<p>Ya tienes cuenta? <a href="login.php">Inicia sesion</a></p>

<?php require "includes/footer.php"; ?>

```

Explicación línea por línea. Los dos require traen la conexión y las funciones. El arreglo \$errores acumula todos los problemas para mostrarlos juntos. Si el formulario se envió por POST, limpiamos los datos con trim y validamos: campos no vacíos, email con formato válido usando filter_var, y contraseña de mínimo 6 caracteres. Después verificamos que el email no exista ya con un SELECT; si existe, agregamos el error. Si no hay errores, ciframos la contraseña con password_hash y hacemos el INSERT con consulta preparada. Luego iniciamos la sesión guardando el id nuevo (con lastInsertId) y el nombre, y redirigimos al index. En el HTML, mostramos los errores con htmlspecialchars por seguridad, el formulario usa method POST, y cada campo lleva required como primera barrera en el navegador. Al final incluimos el footer compartido.

Ejemplo 4: login.php

El inicio de sesión. Fíjate en el mensaje de error genérico, que es una medida de seguridad deliberada.

```
<?php
require "conexion.php";
require "funciones.php";

$error = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $email = trim($_POST["email"]);
    $clave = $_POST["password"];

    $stmt = $pdo->prepare("SELECT id, nombre, password FROM usuarios WHERE email = ?");
    $stmt->execute([$email]);
    $usuario = $stmt->fetch(PDO::FETCH_ASSOC);

    if ($usuario && password_verify($clave, $usuario["password"])) {
        $_SESSION["usuario_id"] = $usuario["id"];
        $_SESSION["usuario_nombre"] = $usuario["nombre"];
        header("Location: index.php");
        exit;
    } else {
        $error = "Correo o contraseña incorrectos.";
    }
}
require "includes/header.php";
?>

<h2>Iniciar sesión</h2>

<?php if ($error !== ""): ?>
    <p class="error"><?php echo htmlspecialchars($error); ?></p>
<?php endif; ?>

<form method="POST" action="login.php">
    <label>Correo:</label>
    <input type="email" name="email" required>

    <label>Contraseña:</label>
    <input type="password" name="password" required>

    <button type="submit">Entrar</button>
```

```

</form>

<p>No tienes cuenta? <a href="registro.php">Registrate</a></p>

<?php require "includes/footer.php"; ?>

```

Explicación: al recibir el POST, buscamos el email en la base de datos con un SELECT que trae id, nombre y el hash de la contraseña. La condición clave es doble: el usuario debe existir Y password_verify debe confirmar la contraseña. Solo si ambas se cumplen guardamos el id y el nombre en la sesión y redirigimos al index. Si algo falla, mostramos el mensaje genérico "Correo o contraseña incorrectos" sin decir si el problema fue el correo o la contraseña, para no regalar información a atacantes. El formulario es simple: email y contraseña por POST. Nota que este archivo no usa requiereLogin porque, por definición, es para quienes aún no han entrado.

Ejemplo 5: crear-post.php

El formulario para publicar un artículo con imagen. Esta es la parte del upload explicada con todo detalle en código.

```

<?php
require "conexion.php";
require "funciones.php";
requireLogin();

$errores = [];

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $titulo = trim($_POST["titulo"]);
    $contenido = trim($_POST["contenido"]);

    if ($titulo === "" || $contenido === "") {
        $errores[] = "El titulo y el contenido son obligatorios.";
    }

    $nombre_final = "";
    if (isset($_FILES["imagen"]) && $_FILES["imagen"]["error"] === 0) {
        $permitidas = ["jpg", "jpeg", "png", "gif", "webp"];
        $extension = strtolower(pathinfo($_FILES["imagen"]["name"],
PATHINFO_EXTENSION));

        if (!in_array($extension, $permitidas)) {
            $errores[] = "Solo se permiten imagenes jpg, png, gif o webp.";
        }
    }
}

```



```

        if ($_FILES["imagen"]["size"] > 2 * 1024 * 1024) {
            $errores[] = "La imagen no debe pesar mas de 2 MB.";
        }

        if (empty($errores)) {
            $nombre_final = uniqid("img_") . "." . $extension;
            $destino = "uploads/" . $nombre_final;
            if (!move_uploaded_file($_FILES["imagen"]["tmp_name"], $destino)) {
                $errores[] = "No se pudo guardar la imagen.";
            }
        }
    } else {
        $errores[] = "Debes subir una imagen para la publicacion.";
    }

    if (empty($errores)) {
        $stmt = $pdo->prepare(
            "INSERT INTO posts (usuario_id, titulo, contenido, imagen) VALUES (?, ?,
?, ?)"
        );
        $stmt->execute([$SESSION["usuario_id"], $titulo, $contenido, $nombre_final]);
        header("Location: index.php");
        exit;
    }
}
require "includes/header.php";
?>

<h2>Nueva publicacion</h2>

<?php foreach ($errores as $error): ?>
    <p class="error"><?php echo htmlspecialchars($error); ?></p>
<?php endforeach; ?>

<form method="POST" action="crear-post.php" enctype="multipart/form-data">
    <label>Titulo:</label>
    <input type="text" name="titulo" required>

    <label>Contenido:</label>
    <textarea name="contenido" rows="8" required></textarea>

    <label>Imagen:</label>
    <input type="file" name="imagen" accept="image/*" required>

    <button type="submit">Publicar</button>

```

```
</form>
```

```
<?php require "includes/footer.php"; ?>
```

Explicación línea por línea. La llamada a `requiereLogin()` al inicio protege la página: sin sesión no se puede publicar. Validamos que título y contenido no vengan vacíos. Para la imagen, primero confirmamos que `$_FILES` la traiga sin errores. La lista `$permitidas` define las extensiones aceptadas y `pathinfo` extrae la extensión real del archivo, convertida a minúsculas para comparar bien. Verificamos el tamaño máximo de 2 MB multiplicando 2 por 1024 por 1024 para obtener bytes. Si todo está bien, generamos el nombre único con `uniqid("img_")` más la extensión, y movemos el archivo temporal a `uploads/` con `move_uploaded_file`. Solo cuando la imagen ya está guardada hacemos el INSERT con el `usuario_id` de la sesión, el título, el contenido y el nombre final de la imagen. El formulario lleva `enctype="multipart/form-data"` sin el cual el archivo nunca llegaría, y el input file usa `accept="image/*"` para que el navegador sugiera solo imágenes.

Ejemplo 6: index.php del blog

La portada que lista todas las publicaciones con el nombre de su autor, usando un JOIN.

```

<?php
require "conexion.php";
require "funciones.php";

$stmt = $pdo->query(
    "SELECT posts.id, posts.titulo, posts.contenido, posts.imagen,
       posts.fecha_publicacion, usuarios.nombre AS autor
    FROM posts
    JOIN usuarios ON posts.usuario_id = usuarios.id
    ORDER BY posts.fecha_publicacion DESC"
);
$posts = $stmt->fetchAll(PDO::FETCH_ASSOC);

require "includes/header.php";
?>

<h2>Ultimas publicaciones</h2>

<?php if (empty($posts)): ?>
    <p>Todavia no hay publicaciones. Se el primero en escribir una.</p>
<?php else: ?>
    <?php foreach ($posts as $post): ?>
        <article>
            "
alt="Imagen del post">
            <h3>
                <a href="post.php?id=<?php echo $post["id"]; ?>">
                    <?php echo htmlspecialchars($post["titulo"]); ?>
                </a>
            </h3>
            <p>Por <?php echo htmlspecialchars($post["autor"]); ?>
                el <?php echo formatearFecha($post["fecha_publicacion"]); ?></p>
            <p><?php echo htmlspecialchars(extracto($post["contenido"])); ?></p>
            <p><a href="post.php?id=<?php echo $post["id"]; ?>">Leer mas</a></p>
        </article>
    <?php endforeach; ?>
<?php endif; ?>

<?php require "includes/footer.php"; ?>

```

Explicación: la consulta usa JOIN para traer en un solo viaje los datos del post y el nombre del autor desde la tabla usuarios. La condición ON posts.usuario_id = usuarios.id es la regla que empareja cada post con su autor. El alias usuarios.nombre AS autor nos permite usar \$post["autor"] de forma clara. El ORDER BY fecha_publicacion DESC pone lo más reciente

primero. Traemos todo con `fetchAll` porque vamos a recorrer la lista. En el HTML, si no hay posts mostramos un mensaje amable en lugar de una página vacía. Cada tarjeta muestra la imagen desde `uploads/`, el título como enlace a `post.php` con su id, el autor y la fecha formateada con nuestra función, y el extracto del contenido. Todo lo que viene de la base de datos pasa por `htmlspecialchars` para prevenir ataques XSS.

Ejemplo 7: `post.php` con comentarios

La página de lectura completa: muestra el artículo, sus comentarios y el formulario para comentar.

```
<?php
require "conexion.php";
require "funciones.php";

$id = isset($_GET["id"]) ? (int)$_GET["id"] : 0;
if ($id <= 0) {
    header("Location: index.php");
    exit;
}

$stmt = $pdo->prepare(
    "SELECT posts.titulo, posts.contenido, posts.imagen, posts.fecha_publicacion,
        usuarios.nombre AS autor
    FROM posts
    JOIN usuarios ON posts.usuario_id = usuarios.id
    WHERE posts.id = ?"
);
$stmt->execute([$id]);
$post = $stmt->fetch(PDO::FETCH_ASSOC);

if (!$post) {
    die("Publicacion no encontrada.");
}

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    requiereLogin();
    $texto = trim($_POST["comentario"]);
    if ($texto !== "") {
        $stmt = $pdo->prepare(
            "INSERT INTO comentarios (post_id, usuario_id, comentario) VALUES (?, ?,
            ?)"
        );
        $stmt->execute([$id, $_SESSION["usuario_id"], $texto]);
    }
}
```

```

    }
    header("Location: post.php?id=" . $id);
    exit;
}

$stmt = $pdo->prepare(
    "SELECT comentarios.comentario, comentarios.fecha, usuarios.nombre AS autor
    FROM comentarios
    JOIN usuarios ON comentarios.usuario_id = usuarios.id
    WHERE comentarios.post_id = ?
    ORDER BY comentarios.fecha ASC"
);
$stmt->execute([$id]);
$comentarios = $stmt->fetchAll(PDO::FETCH_ASSOC);

require "includes/header.php";
?>

<article>
    <h2><?php echo htmlspecialchars($post["titulo"]); ?></h2>
    <p>Por <?php echo htmlspecialchars($post["autor"]); ?>
        el <?php echo formatearFecha($post["fecha_publicacion"]); ?></p>
    " alt="Imagen
del post">
    <p><?php echo nl2br(htmlspecialchars($post["contenido"])); ?></p>
</article>

<h3>Comentarios (<?php echo count($comentarios); ?>)</h3>

<?php foreach ($comentarios as $c): ?>
    <div class="comentario">
        <p><strong><?php echo htmlspecialchars($c["autor"]); ?></strong>
            - <?php echo formatearFecha($c["fecha"]); ?></p>
        <p><?php echo nl2br(htmlspecialchars($c["comentario"])); ?></p>
    </div>
<?php endforeach; ?>

<?php if (usuarioActual()): ?>
    <h3>Deja tu comentario</h3>
    <form method="POST" action="post.php?id=<?php echo $id; ?>">
        <textarea name="comentario" rows="4" required></textarea>
        <button type="submit">Comentar</button>
    </form>
<?php else: ?>
    <p><a href="login.php">Inicia sesion</a> para dejar un comentario.</p>

```

```
<?php endif; ?>
```

```
<?php require "includes/footer.php"; ?>
```

Explicación línea por línea. El id se lee de \$_GET y se convierte a entero con (int); si no es válido, redirigimos al index. La primera consulta trae el post con el nombre del autor mediante JOIN, filtrando con WHERE posts.id = ?. Si no existe, detenemos con un mensaje claro. El bloque POST procesa el comentario: exige login con requiereLogin(), valida que el texto no esté vacío, hace el INSERT con el post_id, el usuario_id de la sesión y el texto, y redirige a la misma página (técnica POST-Redirect-GET) para que recargar no duplique el comentario. La segunda consulta trae los comentarios del post con otro JOIN a usuarios, ordenados por fecha ascendente para leer la conversación en orden. En el HTML mostramos el artículo completo, usando nl2br para que los saltos de línea del textarea se vean como párrafos. La lista de comentarios muestra autor, fecha y texto. Al final, si hay sesión mostramos el formulario; si no, un enlace al login. Fíjate que la verificación de sesión se hace dos veces: una para mostrar el formulario y otra al procesarlo, porque un atacante podría enviar el POST directamente sin pasar por la página.

Ejemplo 8: mis-posts.php con editar y eliminar

El panel personal del autor con la lista de sus publicaciones y las acciones de editar y eliminar.

```
<?php
require "conexion.php";
require "funciones.php";
requiereLogin();

$usuario_id = $_SESSION["usuario_id"];

$stmt = $pdo->prepare(
    "SELECT id, titulo, imagen, fecha_publicacion
    FROM posts WHERE usuario_id = ? ORDER BY fecha_publicacion DESC"
);
$stmt->execute([$usuario_id]);
$posts = $stmt->fetchAll(PDO::FETCH_ASSOC);

require "includes/header.php";
?>

<h2>Mis publicaciones</h2>
<p><a href="crear-post.php">Escribir nueva publicacion</a></p>

<?php if (empty($posts)): ?>
```

```

        <p>Aun no tienes publicaciones.</p>
    <?php else: ??>
        <?php foreach ($posts as $post): ??>
            <article>
                "
alt="">
                <h3><?php echo htmlspecialchars($post["titulo"]); ?></h3>
                <p><?php echo formatearFecha($post["fecha_publicacion"]); ?></p>
                <p>
                    <a href="post.php?id=<?php echo $post["id"]; ?>">Ver</a> |
                    <a href="editar-post.php?id=<?php echo $post["id"]; ?>">Editar</a> |
                    <form method="POST" action="eliminar-post.php" style="display:inline;"
                        onsubmit="return confirm('Seguro que quieres eliminar esta
publicacion?');">
                        <input type="hidden" name="id" value="<?php echo $post["id"]; ?>">
                        <button type="submit">Eliminar</button>
                    </form>
                </p>
            </article>
        <?php endforeach; ??>
    <?php endif; ??>

<?php require "includes/footer.php"; ?>

```

Explicación: requiereLogin() protege el panel. La consulta filtra con WHERE usuario_id = ? usando el id de la sesión, así cada autor ve solo lo suyo. Cada publicación muestra su imagen, título, fecha y tres acciones: Ver (lleva a post.php), Editar (lleva a editar-post.php con el id) y Eliminar. Fíjate que Eliminar no es un enlace sino un formulario POST con el id en un campo oculto, más una confirmación con JavaScript mediante onsubmit y confirm(). Esto evita borrados accidentales por clics o por robots que siguen enlaces. El archivo eliminar-post.php que recibe este formulario debe verificar con esDueno() que el post pertenezca al usuario, borrar primero los comentarios de ese post, borrar la imagen con unlink() y finalmente hacer el DELETE del post. El archivo editar-post.php sigue la misma lógica de verificación, muestra el formulario con los datos actuales precargados y procesa el UPDATE, reemplazando la imagen solo si se sube una nueva.

Práctica

Ya tienes el blog funcionando. Pero un proyecto nunca está terminado de verdad: siempre hay funciones nuevas que agregar. Estos cinco ejercicios te proponen mejoras reales que harían tu blog más completo y más atractivo para tu portafolio. Intenta resolver cada uno por tu cuenta antes de leer la solución. Si te atorras más de treinta minutos, lee la solución, entiéndela y escríbela tú mismo.

Ejercicio 1

Agrega categorías al blog. Crea una tabla categorías con id y nombre, agrega una columna categoria_id a la tabla posts, y modifica crear-post.php para que el autor elija una categoría de una lista desplegable. En el index.php, muestra la categoría de cada publicación y agrega enlaces para filtrar los posts por categoría.

Solución 1

Primero el SQL para la tabla nueva y la columna nueva. La columna permite NULL para no romper los posts que ya existen.

```
CREATE TABLE categorias (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL UNIQUE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;  
  
ALTER TABLE posts ADD COLUMN categoria_id INT NULL,  
ADD FOREIGN KEY (categoria_id) REFERENCES categorias(id);  
  
INSERT INTO categorias (nombre) VALUES  
('Programacion'), ('Diseno'), ('Noticias'), ('Tutoriales');
```

Explicación paso a paso. Creamos la tabla categorías con id y nombre único. Con ALTER TABLE agregamos categoria_id a posts como llave foránea opcional. Insertamos cuatro categorías de ejemplo para probar. Ahora modificamos crear-post.php: antes del formulario, consultamos las categorías y las mostramos en un select.


```

<?php
$stmt = $pdo->query("SELECT id, nombre FROM categorias ORDER BY nombre");
$categorias = $stmt->fetchAll(PDO::FETCH_ASSOC);
?>

<label>Categoria:</label>
<select name="categoria_id">
    <option value="">Sin categoria</option>
    <?php foreach ($categorias as $cat): ?>
        <option value="<?php echo $cat["id"]; ?>">
            <?php echo htmlspecialchars($cat["nombre"]); ?>
        </option>
    <?php endforeach; ?>
</select>

```

Al procesar el formulario, leemos `$categoria_id = $_POST["categoria_id"] === "" ? null : (int)$_POST["categoria_id"]`; y lo incluimos en el INSERT. En el index.php agregamos un JOIN más para traer el nombre de la categoría y mostramos un menú de filtros.

```

$stmt = $pdo->query(
    "SELECT posts.id, posts.titulo, posts.contenido, posts.imagen,
        posts.fecha_publicacion, usuarios.nombre AS autor,
        categorias.nombre AS categoria
    FROM posts
    JOIN usuarios ON posts.usuario_id = usuarios.id
    LEFT JOIN categorias ON posts.categoria_id = categorias.id
    ORDER BY posts.fecha_publicacion DESC"
);

```

Usamos LEFT JOIN porque algunos posts pueden no tener categoría y con un JOIN normal desaparecerían de la lista. Para el filtro por categoría, leemos `$_GET["categoría"]`, lo convertimos a entero, y si es mayor que cero agregamos `WHERE posts.categoria_id = ?` a la consulta usando una consulta preparada. Los enlaces del filtro se generan recorriendo las categorías: `<a href="index.php?categoría=2">Diseno</a>`. Con esto el blog ya se puede navegar por temas.

Ejercicio 2

Agrega un buscador al blog. Pon un formulario de búsqueda en el index.php y muestra solo las publicaciones cuyo título o contenido contengan el texto buscado. El buscador debe ser seguro contra inyección SQL.

Solución 2

El formulario va arriba de la lista de publicaciones en index.php. Usamos método GET para que la búsqueda quede en la dirección y se pueda compartir el enlace.

```
<form method="GET" action="index.php">
    <input type="text" name="q" placeholder="Buscar publicaciones..."
        value="<?php echo htmlspecialchars($_GET["q"] ?? ""); ?>">
    <button type="submit">Buscar</button>
</form>
```

Explicación paso a paso. El campo se llama q y conserva el texto buscado usando el operador ?? para evitar errores cuando no hay búsqueda. En el PHP, construimos la consulta de forma condicional.

```
$sql = "SELECT posts.id, posts.titulo, posts.contenido, posts.imagen,
        posts.fecha_publicacion, usuarios.nombre AS autor
        FROM posts
        JOIN usuarios ON posts.usuario_id = usuarios.id";
$parametros = [];

if (!empty($_GET["q"])) {
    $sql .= " WHERE posts.titulo LIKE ? OR posts.contenido LIKE ?";
    $busqueda = "%" . $_GET["q"] . "%";
    $parametros = [$busqueda, $busqueda];
}

$sql .= " ORDER BY posts.fecha_publicacion DESC";
$stmt = $pdo->prepare($sql);
$stmt->execute($parametros);
$posts = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

El truco está en LIKE con los signos de porcentaje: %texto% encuentra el texto en cualquier parte del título o del contenido. Lo importante de seguridad es que el texto del usuario nunca se pega directamente en el SQL: va como parámetro ? en la consulta preparada, así PDO lo trata como dato y no como código. Ni siquiera un texto malicioso con comillas puede romper la consulta. Si no hay búsqueda, la consulta trae todo como antes. Además mostramos un mensaje con el número de resultados:

Se encontraron 3 resultados para "php".

 Este patrón de construir el SQL por partes con un arreglo de parámetros te servirá para filtros avanzados en cualquier proyecto.

Ejercicio 3

Agrega paginación al inicio del blog. Muestra solo 5 publicaciones por página y pon enlaces de "Anterior" y "Siguiente" con los números de página. Debe funcionar combinada con el buscador del ejercicio anterior.

Solución 3

La paginación se logra con LIMIT y OFFSET en SQL: LIMIT dice cuántos traer y OFFSET desde cuál empezar. La página 1 trae del 0 al 4, la página 2 del 5 al 9, y así.

```
$por_pagina = 5;
$pagina = isset($_GET["pag"]) ? max(1, (int)$_GET["pag"]) : 1;
$offset = ($pagina - 1) * $por_pagina;

$sql = "SELECT posts.id, posts.titulo, posts.contenido, posts.imagen,
           posts.fecha_publicacion, usuarios.nombre AS autor
        FROM posts
        JOIN usuarios ON posts.usuario_id = usuarios.id";
$parametros = [];

if (!empty($_GET["q"])) {
    $sql .= " WHERE posts.titulo LIKE ? OR posts.contenido LIKE ?";
    $busqueda = "%" . $_GET["q"] . "%";
    $parametros = [$busqueda, $busqueda];
}

$sql .= " ORDER BY posts.fecha_publicacion DESC LIMIT $por_pagina OFFSET $offset";
$stmt = $pdo->prepare($sql);
$stmt->execute($parametros);
$posts = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Explicación paso a paso. Calculamos la página actual desde `$_GET["pag"]`, asegurando que sea al menos 1 con `max()`. El offset es (página - 1) por registros por página. Agregamos LIMIT y OFFSET al final del SQL. Nota que `$por_pagina` y `$offset` se pegan directo al SQL sin parámetros porque ya los convertimos a enteros nosotros mismos, así no hay riesgo de inyección. Para los enlaces de navegación necesitamos saber cuántas páginas hay en total, así que contamos los registros con la misma condición de búsqueda.

```

$sql_total = "SELECT COUNT(*) FROM posts";
$params_total = [];
if (!empty($_GET["q"])) {
    $sql_total .= " WHERE titulo LIKE ? OR contenido LIKE ?";
    $params_total = [$busqueda, $busqueda];
}
$stmt = $pdo->prepare($sql_total);
$stmt->execute($params_total);
$total = $stmt->fetchColumn();
$total_paginas = ceil($total / $por_pagina);

```

Y generamos los enlaces conservando la búsqueda en la dirección para que no se pierda al cambiar de página.

```

<?php for ($i = 1; $i <= $total_paginas; $i++): ?>
    <a href="index.php?pag=<?php echo $i; ?>&q=<?php echo urlencode($_GET["q"]) ??
    """); ?>">
        <?php echo $i; ?>
    </a>
<?php endfor; ?>

```

La función urlencode prepara el texto de búsqueda para viajar seguro en la dirección. Con esto el blog escala: aunque tengas mil publicaciones, cada página carga rápido porque solo trae cinco. La paginación es una de esas funciones que todo sitio real necesita y que ahora ya sabes construir.

Ejercicio 4

Crea una página de perfil de autor. Al hacer clic en el nombre del autor en cualquier publicación, se debe abrir `perfil.php?id=...` mostrando su nombre, desde cuándo está registrado, cuántas publicaciones tiene y la lista de sus publicaciones con enlaces.

Solución 4

Creamos el archivo `perfil.php`. La página recibe el id del usuario, valida que sea un número y trae sus datos más sus publicaciones.

```

<?php
require "conexion.php";
require "funciones.php";

$id = isset($_GET["id"]) ? (int)$_GET["id"] : 0;

```

```

if ($id <= 0) {
    header("Location: index.php");
    exit;
}

$stmt = $pdo->prepare("SELECT nombre, fecha_registro FROM usuarios WHERE id = ?");
$stmt->execute([$id]);
$autor = $stmt->fetch(PDO::FETCH_ASSOC);

if (!$autor) {
    die("Autor no encontrado.");
}

$stmt = $pdo->prepare(
    "SELECT id, titulo, imagen, fecha_publicacion
    FROM posts WHERE usuario_id = ? ORDER BY fecha_publicacion DESC"
);
$stmt->execute([$id]);
$posts = $stmt->fetchAll(PDO::FETCH_ASSOC);

require "includes/header.php";
?>

<h2><?php echo htmlspecialchars($autor["nombre"]); ?></h2>
<p>Miembro desde: <?php echo formatearFecha($autor["fecha_registro"]); ?></p>
<p>Publicaciones: <?php echo count($posts); ?></p>

<h3>Sus articulos</h3>
<?php foreach ($posts as $post): ?>
    <article>
        " alt="">
        <h4>
            <a href="post.php?id=<?php echo $post["id"]; ?>">
                <?php echo htmlspecialchars($post["titulo"]); ?>
            </a>
        </h4>
        <p><?php echo formatearFecha($post["fecha_publicacion"]); ?></p>
    </article>
<?php endforeach; ?>

<?php require "includes/footer.php"; ?>

```

Explicación paso a paso. Validamos el id igual que en post.php. La primera consulta trae el nombre y la fecha de registro del autor; si no existe, mostramos el error. La segunda trae sus publicaciones ordenadas por fecha. El conteo de publicaciones lo obtenemos con count()

sobre el arreglo, sin necesidad de otra consulta. En el HTML mostramos los datos del perfil y la lista de artículos con enlaces a post.php. El último paso es convertir los nombres de autor en enlaces en index.php y post.php: cambiamos el texto del autor por . Para eso necesitamos también el usuario_id en esas consultas, así que agregamos posts.usuario_id al SELECT. Con este ejercicio practicaste el patrón completo de "página de detalle por id", que es el mismo que usarías para perfiles, productos o cualquier entidad.

Ejercicio 5

Agrega un sistema de "me gusta" a las publicaciones. Los usuarios registrados pueden dar "me gusta" a un post una sola vez, y en post.php se muestra el total. Si ya le dio me gusta, el botón debe permitir quitarlo.

Solución 5

Primero la tabla que guarda los "me gusta". Cada fila es un voto: quién votó y a qué post. La restricción UNIQUE evita votos duplicados a nivel de base de datos.

```
CREATE TABLE likes (
  id INT AUTO_INCREMENT PRIMARY KEY,
  post_id INT NOT NULL,
  usuario_id INT NOT NULL,
  fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  UNIQUE KEY unico_voto (post_id, usuario_id),
  FOREIGN KEY (post_id) REFERENCES posts(id),
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Explicación paso a paso. La clave UNIQUE sobre la pareja (post_id, usuario_id) garantiza que un usuario solo pueda votar una vez por post, incluso si alguien intenta mandar el formulario dos veces rápido. En post.php, antes del HTML, procesamos el voto.

```

if ($_SERVER["REQUEST_METHOD"] === "POST" && isset($_POST["like"])) {
    requiereLogin();
    $uid = $_SESSION["usuario_id"];

    $stmt = $pdo->prepare("SELECT id FROM likes WHERE post_id = ? AND usuario_id = ?");
    $stmt->execute([$id, $uid]);

    if ($stmt->fetch()) {
        $stmt = $pdo->prepare("DELETE FROM likes WHERE post_id = ? AND usuario_id = ?");
        $stmt->execute([$id, $uid]);
    } else {
        $stmt = $pdo->prepare("INSERT INTO likes (post_id, usuario_id) VALUES (?, ?)");
        $stmt->execute([$id, $uid]);
    }
    header("Location: post.php?id=" . $id);
    exit;
}

$stmt = $pdo->prepare("SELECT COUNT(*) FROM likes WHERE post_id = ?");
$stmt->execute([$id]);
$total_likes = $stmt->fetchColumn();

$ya_voto = false;
if (usuarioActual()) {
    $stmt = $pdo->prepare("SELECT id FROM likes WHERE post_id = ? AND usuario_id = ?");
    $stmt->execute([$id, $_SESSION["usuario_id"]]);
    $ya_voto = $stmt->fetch() !== false;
}

```

La lógica es un interruptor: si ya existe el voto, lo borramos; si no existe, lo creamos. Después redirigimos para evitar votos duplicados al recargar. Contamos el total con COUNT(*) y verificamos si el usuario actual ya votó para cambiar el texto del botón.

```

<?php if (usuarioActual()): ?>
    <form method="POST" action="post.php?id=<?php echo $id; ?>">
        <button type="submit" name="like">
            <?php echo $ya_voto ? "Quitar me gusta" : "Me gusta"; ?>
            (<?php echo $total_likes; ?>)
        </button>
    </form>
<?php else: ?>
    <p>Esta publicacion tiene <?php echo $total_likes; ?> me gusta.
    <a href="login.php">Inicia sesion</a> para votar.</p>
<?php endif; ?>

```

El botón cambia su texto según si ya votaste o no, y siempre muestra el total. Solo los usuarios registrados pueden votar. Este ejercicio te enseñó tres cosas valiosas: las restricciones UNIQUE como defensa en la base de datos, el patrón de interruptor (toggle) tan común en interfaces, y el conteo con COUNT(*). Con esto tu blog ya tiene interacción social básica, y en tu portafolio puedes presumir que implementaste un sistema de votos completo.

Resumen de la unidad

- Un proyecto completo vale más que diez tutoriales sueltos: es donde las piezas sueltas se convierten en una aplicación real y en la primera pieza de tu portafolio.
- Planear antes de programar significa responder qué datos vas a guardar, qué páginas tendrá el sitio y qué puede hacer cada tipo de usuario.
- La base de datos del blog usa tres tablas relacionadas: usuarios, posts y comentarios, unidas por llaves foráneas.
- El registro cifra contraseñas con password_hash, el login las verifica con password_verify y usa mensajes de error genéricos por seguridad.
- Las sesiones mantienen al usuario adentro: cada página protegida verifica \$_SESSION y redirige al login si no hay sesión, siempre con exit después del header.
- Subir imágenes requiere enctype="multipart/form-data", validación de extensión y tamaño, nombre único con uniqid y move_uploaded_file.
- El JOIN combina tablas en una sola consulta: la condición ON indica cómo emparejar las filas de cada tabla.
- Todo dato que venga del usuario o de la base de datos y se muestre en HTML debe pasar por htmlspecialchars para prevenir XSS.

- Un usuario solo puede editar o eliminar sus propias publicaciones: la verificación se hace en el servidor, nunca se confía en los botones.
- Después de procesar un POST, siempre redirige con header (técnica POST-Redirect-GET) para evitar duplicados al recargar.
- El despliegue tiene cuatro pasos: exportar la base de datos, subir archivos por FTP, crear e importar la base de datos en el hosting, y ajustar conexión.php.
- Los errores de despliegue más comunes son las credenciales mal copiadas, las mayúsculas en nombres de archivo en Linux, las rutas absolutas de tu computadora y los permisos de la carpeta uploads.
- Después de PHP sigue la POO, luego Laravel, luego APIs REST y JavaScript: cada tema nuevo se aprende mejor construyendo un proyecto nuevo.

Cierre: lo que sigue es tuyo

Llegaste al final del libro, pero en realidad llegaste al inicio de algo más grande. Piensa en dónde estabas en la primera página: PHP era solo una palabra rara que habías escuchado por ahí. Ahora tienes un blog completo publicado en internet, con usuarios reales que pueden registrarse, publicar con imágenes y comentar. Eso lo construiste tú, línea por línea, error por error. Nadie te lo regaló.

Te voy a decir algo que casi ningún curso dice: los errores que cometiste en estas diez unidades valen tanto como los aciertos. Cada página en blanco que te sacó canas, cada "Access denied" que te hizo revisar letra por letra, cada upload que no funcionaba por el enctype, todo eso te entrenó en la habilidad real de un programador, que no es memorizar sintaxis sino no rendirse cuando algo no funciona. La sintaxis se busca en internet en diez segundos. La persistencia no se busca, se construye, y tú ya la tienes.

No compares tu inicio con el final de otros. Vas a ver programadores con diez años de experiencia haciendo cosas que te parecerán magia, y es fácil sentir que vas atrasado. Pero ellos también empezaron con un blog sencillo, también se pelearon con una conexión a la base de datos, también subieron su primer archivo por FTP con el corazón acelerado. La única diferencia entre ellos y tú es el tiempo y los proyectos. Y el tiempo va a pasar de todos modos, así que mejor que pase construyendo.

Tu tarea de ahora en adelante es simple y es para siempre: sigue construyendo. Reescribe este blog con Programación Orientada a Objetos. Vuelve a hacerlo en Laravel y compara cuánto más rápido sale. Convierte tu blog en una API y hazle una app. Cada proyecto nuevo te va a enseñar algo que ningún libro puede enseñarte, porque los libros enseñan el camino y los proyectos enseñan a caminar.

Guarda este blog, publícalo, presúmelo. Es tu proyecto número uno y algún día, cuando tengas veinte, vas a volver a verlo y te va a dar ternura lo sencillo que era. Ese día vas a entender que el programador que eres entonces nació aquí, en estas diez unidades, escribiendo código que a veces no funcionaba y arreglándolo hasta que sí.

Nos vemos en el siguiente curso, en lucasegaacademy.com. Lo que sigue es tuyo. A construir.

EXAMEN FINAL

Felicidades por llegar hasta aquí. Completaste el Curso Completo de PHP de lucasegaacademy.com y este examen final es tu prueba de fuego. Hazlo SIN apuntes y SIN ver el libro: la honestidad contigo mismo es lo que hace que el conocimiento se quede de verdad y para siempre. Tiempo sugerido: 90 minutos. Para calificarte, cada pregunta vale 2 puntos, así que el máximo es 100. Si sacas 90 o más, excelente: dominas PHP y estas listo para proyectos reales y clientes de verdad. Entre 70 y 88, bien, pero repasa las unidades donde fallaste. Menos de 70, vuelve a las unidades y práctica más antes de intentarlo de nuevo. Las primeras 40 preguntas son de opción múltiple y verdadero/falso; las últimas 10 son abiertas y debes escribir código PHP. Al final encontraras las respuestas con explicaciones detalladas: usalas para aprender de tus errores, no solo para contar aciertos.

Estrategia para el examen: no te estanques más de 3 minutos en una pregunta difícil; marca tu mejor respuesta y sigue adelante, al final regresas a las dudosas. En las preguntas abiertas escribe el código aunque no estes seguro de cada detalle: un intentó con la estructura correcta (prepare, execute, fetch) vale más que una hoja en blanco, y es así cómo se programa en la vida real, iterando. Lee cada pregunta dos veces antes de responder, porque muchas trampas están en una sola palabra (por ejemplo, "antes" contra "después" en lo de `session_start`). Y respira: ya hiciste el trabajo durante el curso, este examen solo lo pone en orden.

Contexto para las preguntas abiertas: usaras una mini-tienda con estas tablas. Aquí esta el SQL para crearlas:

```

CREATE TABLE productos (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  precio DECIMAL(10,2) NOT NULL,
  stock INT NOT NULL DEFAULT 0
);

CREATE TABLE usuarios (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  email VARCHAR(150) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL
);

CREATE TABLE pedidos (
  id INT AUTO_INCREMENT PRIMARY KEY,
  usuario_id INT NOT NULL,
  producto_id INT NOT NULL,
  cantidad INT NOT NULL,
  fecha DATETIME NOT NULL,
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id),
  FOREIGN KEY (producto_id) REFERENCES productos(id)
);

```

PREGUNTA 1: En PHP, cómo se escribe correctamente una variable que guarda el nombre "Lucas"?

a) \$nombre = "Lucas"; b) nombre = "Lucas"; c) var \$nombre = "Lucas"; d) string \$nombre = "Lucas";

PREGUNTA 2: Que imprime el siguiente código?

```

$edad = 25;
echo "Tengo $edad años";

```

a) Tengo \$edad años b) Tengo 25 años c) Error de sintaxis d) Tengo "25" años

PREGUNTA 3: Cual de estas afirmaciones sobre PHP es verdadera?

a) PHP es un lenguaje de tipado estático como Java b) PHP se ejecuta en el navegador del usuario c) PHP es de tipado dinámico y se ejecuta en el servidor d) Todo código PHP debe compilarse antes de ejecutarse

PREGUNTA 4: Que valor tiene \$c después de ejecutar este código?

```
$a = 10;  
$b = "5";  
$c = $a + $b;
```

a) 105 b) 15 c) Error porque no se pueden sumar número y texto d) "105"

PREGUNTA 5: Cual es la diferencia entre comillas dobles y simples en PHP?

a) No hay diferencia, son intercambiables b) Las dobles interpretan variables dentro del texto, las simples no c) Las simples interpretan variables, las dobles no d) Las dobles solo sirven para HTML

PREGUNTA 6: Que resultado da la expresión (8 % 3)?

a) 2 b) 2.66 c) 1 d) 0

PREGUNTA 7: Que imprime este código?

```
$x = 5;  
$y = $x++;  
echo $y . " - " . $x;
```

a) 6 - 6 b) 5 - 6 c) 6 - 5 d) 5 - 5

PREGUNTA 8: Cual operador se usa para comparar valor Y tipo de dato al mismo tiempo?

a) == b) = c) === d) !=

PREGUNTA 9: Que evalua la expresión (true && false || true)?

a) false b) true c) Error lógico d) null

PREGUNTA 10: Que valor toma \$total en este código?

```
$precio = 100;  
$descuento = 20;  
$total = $precio - ($precio * $descuento / 100);
```

a) 80 b) 20 c) 98 d) 60

PREGUNTA 11: Que imprime este código?

```
$nota = 7;
if ($nota >= 9) {
    echo "Excelente";
} elseif ($nota >= 6) {
    echo "Aprobado";
} else {
    echo "Reprobado";
}
```

a) Excelente b) Aprobado c) Reprobado d) Nada, hay error

PREGUNTA 12: En un switch, para que sirve la palabra break?

a) Para terminar el programa completo b) Para evitar que se ejecuten los casos siguientes c) Para regresar al caso anterior d) No sirve para nada, es opcional decorativa

PREGUNTA 13: Qué hace el operador ternario en esta línea?

```
$mensaje = ($edad >= 18) ? "Mayor" : "Menor";
```

a) Suma 18 a la edad b) Asigna "Mayor" si \$edad es 18 o más, si no asigna "Menor" c) Compara dos variables llamadas Mayor y Menor d) Genera un error porque falta el if

PREGUNTA 14: Verdadero o falso: En PHP, un if puede llevar condición sin paréntesis, así: if \$x > 5 { ... }.

a) Verdadero b) Falso

PREGUNTA 15: Que imprime este código con el operador de fusión null?

```
$nombre = $_GET["nombre"] ?? "Invitado";
echo $nombre;
```

Supon que la URL no trae el parámetro nombre. a) Error: índice no definido b) Invitado c) Una cadena vacía d) null

PREGUNTA 16: Cuantas veces se ejecuta el echo en este bucle?

```
for ($i = 1; $i <= 5; $i++) {
    echo $i;
}
```

a) 4 veces b) 5 veces c) 6 veces d) Infinitas veces

PREGUNTA 17: Que imprime este bucle while?

```
$n = 3;
while ($n > 0) {
    echo $n;
    $n--;
}
```

a) 321 b) 210 c) 123 d) 3210

PREGUNTA 18: Para que sirve la instrucción continue dentro de un bucle?

a) Termina el bucle por completo b) Salta a la siguiente iteración sin terminar la actual c) Reinicia el bucle desde cero d) Pausa el bucle 1 segundo

PREGUNTA 19: Verdadero o falso: Un bucle do...while siempre ejecuta su cuerpo al menos una vez, aunque la condición sea falsa desde el inicio.

a) Verdadero b) Falso

PREGUNTA 20: Que imprime este código?

```
foreach ([10, 20, 30] as $valor) {
    if ($valor == 20) {
        break;
    }
    echo $valor . " ";
}
```

a) 10 20 30 b) 10 c) 20 30 d) 10 20

PREGUNTA 21: Que imprime este código?

```
function saludar($nombre = "amigo") {
    return "Hola, $nombre";
}
echo saludar();
```

a) Hola, b) Hola, amigo c) Error: falta el argumento d) Hola, \$nombre

PREGUNTA 22: Cual es la diferencia entre echo y return dentro de una función?

a) No hay diferencia b) echo imprime en pantalla y return devuelve un valor a quien llamó la función c) return imprime y echo devuelve d) echo solo funciona fuera de funciones

PREGUNTA 23: Que imprime este código sobre alcance de variables?

```
$numero = 10;
function duplicar() {
    $numero = $numero * 2;
    return $numero;
}
echo duplicar();
```

a) 20 b) Error o advertencia porque \$número no existe dentro de la función c) 10 d) 0

PREGUNTA 24: Verdadero o falso: Una función en PHP puede recibir otra función como argumento (funciones de orden superior con callable).

a) Verdadero b) Falso

PREGUNTA 25: Que devuelve esta función recursiva al llamarla con factorial(4)?

```
function factorial($n) {
    if ($n <= 1) {
        return 1;
    }
    return $n * factorial($n - 1);
}
```

a) 10 b) 24 c) 12 d) Bucle infinito

PREGUNTA 26: Como accedes al valor "rojo" en este arreglo?

```
$colores = ["rojo", "verde", "azul"];
```

a) \$colores[1] b) \$colores[0] c) \$colores["rojo"] d) \$colores->rojo

PREGUNTA 27: Que imprime este código?

```
$alumno = ["nombre" => "Ana", "edad" => 22];
echo $alumno["edad"];
```

a) Ana b) 22 c) Error: sintaxis de arreglo invalida d) nombre

PREGUNTA 28: Qué hace la función count() en este código?


```
$frutas = ["manzana", "pera", "uva"];  
echo count($frutas);
```

a) Suma los valores del arreglo b) Devuelve 3, el número de elementos c) Devuelve el último elemento d) Ordena el arreglo

PREGUNTA 29: Verdadero o falso: array_push(\$lista, "nuevo") agrega "nuevo" al final del arreglo \$lista.

a) Verdadero b) Falso

PREGUNTA 30: Que imprime este código con un arreglo multidimensional?

```
$tienda = [  
    ["pan", 20],  
    ["leche", 35]  
];  
echo $tienda[1][0] . " cuesta " . $tienda[1][1];
```

a) pan cuesta 20 b) leche cuesta 35 c) Error de índices d) Array cuesta Array

PREGUNTA 31: Un formulario HTML usa method="get". Donde llegan los datos en PHP?

a) En \$_POST b) En \$_GET c) En \$_REQUEST solamente si es POST d) En \$GLOBALS["form"]

PREGUNTA 32: Cual es la diferencia principal entre GET y POST al enviar un formulario?

a) No hay diferencia real b) GET muestra los datos en la URL y tiene limite de tamaño; POST los oculta del URL y admite más datos c) POST es más rápido que GET d) GET solo sirve para números

PREGUNTA 33: Qué hace esta línea de validación?

```
$email = filter_var($_POST["email"], FILTER_VALIDATE_EMAIL);
```

a) Encripta el email b) Devuelve el email si tiene formato valido, o false si no lo tiene c) Envía un correo de confirmación d) Guarda el email en la base de datos

PREGUNTA 34: Verdadero o falso: Siempre debes validar y sanear los datos que vienen de \$_GET y \$_POST porque el usuario puede enviar lo que quiera.

a) Verdadero b) Falso

PREGUNTA 35: Qué hace htmlspecialchars() y por qué es importante al mostrar datos del usuario?

a) Encripta contraseñas b) Convierte caracteres especiales como < y > en entidades HTML, previniendo ataques XSS c) Valida direcciones de correo d) Comprime el texto para que pese menos

PREGUNTA 36: Cual es la diferencia entre una cookie y una sesión?

a) Son lo mismo b) La cookie se guarda en el navegador del usuario; la sesión guarda los datos en el servidor y solo una identificación viaja en la cookie c) La sesión se guarda en el navegador d) Las cookies no pueden guardar texto

PREGUNTA 37: Qué hace session_start() y donde debe colocarse?

a) Cierra la sesión; al final del script b) Inicia o reanuda la sesión; debe ir antes de cualquier salida HTML c) Borra todas las variables de sesión; donde sea d) Crea una cookie; después del HTML

PREGUNTA 38: Como guardas el nombre del usuario logueado en la sesión?

a) `$SESSION["usuario"] = $nombre;` b) `$_SESSION["usuario"] = $nombre;` c) `session("usuario", $nombre);` d) `$_COOKIE["usuario"] = $nombre;`

PREGUNTA 39: Verdadero o falso: Para cerrar la sesión de un usuario basta con hacer `$_SESSION = []` sin llamar `session_destroy()`.

a) Verdadero b) Falso

PREGUNTA 40: En un login seguro, como debes verificar la contraseña que escribe el usuario contra la guardada en la base de datos?

a) Comparando el texto plano con `==` b) Con `password_verify($claveEscrita, $hashGuardado)`, porque la contraseña se guardó con `password_hash()` c) Encriptando ambas con md5 y comparando d) No se puede verificar, solo el usuario la conoce

PREGUNTA 41: Escribe el código PHP completo para conectarte a una base de datos MySQL llamada "tienda" (usuario root, sin contraseña, servidor localhost) usando PDO, con manejo de errores que muestre el mensaje si falla la conexión.

PREGUNTA 42: Escribe una sentencia preparada con PDO que inserte un producto en la tabla productos (nombre, precio, stock). Los valores vienen en las variables \$nombre, \$precio y \$stock. Usa marcadores con nombre (:nombre, etc.).

PREGUNTA 43: Escribe el código que lea TODOS los productos de la tabla productos y los muestre en una lista HTML , donde cada diga "nombre - \$precio" y el nombre vaya protegido contra XSS. Usa PDO y sentencias preparadas.

PREGUNTA 44: Escribe una función PHP llamada registrarPedido(\$pdo, \$usuarioid, \$productoid, \$cantidad) que: 1) inserte el pedido en la tabla pedidos con la fecha actual (usa NOW() de MySQL), 2) reste la cantidad del stock del producto, 3) devuelva true si todo salió bien o false si algo falló. Usa sentencias preparadas.

PREGUNTA 45: Escribe el código PHP que recibe un archivo subido desde un formulario (campo "foto"), verifica que sea una imagen JPG o PNG de máximo 2 MB, y si pasa la validación la guarda en la carpeta "uploads/" con un nombre único. Si algo falla, muestra un mensaje de error.

PREGUNTA 46: Escribe el código de login completo: recibe email y password por POST, busca al usuario en la tabla usuarios con una sentencia preparada, verifica la contraseña con password_verify(), y si todo es correcto guarda el id y el nombre en la sesión y redirige a "panel.php". Si falla, muestra "Credenciales invalidas". No olvides session_start().

PREGUNTA 47: Escribe el código que actualiza el precio de un producto: recibe por POST el id y el nuevo precio, valida que el precio sea un número mayor que cero, y hace el UPDATE con sentencia preparada. Muestra cuantas filas fueron afectadas.

PREGUNTA 48: Escribe una función eliminarProducto(\$pdo, \$id) que borre el producto con ese id usando sentencia preparada y devuelva el número de filas eliminadas. Además explica en un comentario dentro del código por qué una sentencia preparada protege contra inyección SQL.

PREGUNTA 49: Escribe el código que muestra los pedidos de un usuario logueado (su id está en \$_SESSION["usuario_id"]): haz un JOIN entre pedidos y productos para mostrar el nombre del producto, la cantidad y la fecha de cada pedido, ordenados del más reciente al más antiguo. Protege la salida contra XSS.

PREGUNTA 50: Escribe un checklist en comentarios con los 6 pasos para subir tu proyecto PHP a un hosting compartido (archivos, base de datos, credenciales, etc.), y debajo un código que lea las credenciales de la base de datos desde variables de entorno (getenv()) en lugar de escribirlas directas en el código, explicando en un comentario por qué esto es más seguro.

Respuestas y soluciones

RESPUESTA 1: a) \$nombre = "Lucas";

En PHP toda variable empieza obligatoriamente con el signo \$, así que la opción a) es la única sintaxis válida: \$nombre = "Lucas";. La opción b) falla porque sin el signo \$ PHP interpreta "nombre" como una constante no definida y lanza un error. La opción c) usa la palabra var, que es sintaxis de JavaScript para declarar variables, no existe en PHP. La opción d) intenta declarar el tipo como en Java o TypeScript (string \$nombre), pero en PHP las variables simples no se declaran con tipo de esa forma. Recuerda: signo \$, nombre descriptivo, operador = para asignar y punto y coma al final. Olvidar el punto y coma es uno de los errores más comunes cuando empiezas.

RESPUESTA 2: b) Tengo 25 años

Las comillas dobles en PHP interpretan (interpolan) las variables que encuentran dentro del texto, así que \$edad se sustituye por su valor 25 y el resultado es "Tengo 25 años". La opción a) sería correcta si el texto usara comillas simples ('Tengo \$edad años'), porque las simples muestran todo literal. La opción c) es incorrecta porque no hay ningún error de sintaxis en el código. La opción d) es incorrecta porque PHP no agrega comillas alrededor del valor al interpolarlo. Este detalle de comillas dobles contra simples aparece en muchísimos exámenes y proyectos reales, así que vale la pena tenerlo clarísimo. Un truco útil: con comillas dobles puedes usar llaves para delimitar la variable cuando va pegada a texto, como "Hola {\$nombre}s", lo que evita ambigüedades. Y si necesitas comillas dobles dentro de un texto con comillas dobles, escápalas con backslash: "El dijo \"hola\"". Con las simples, solo necesitas escapar la comilla simple y el backslash.

RESPUESTA 3: c) PHP es de tipado dinámico y se ejecuta en el servidor

PHP es de tipado dinámico: no declaras el tipo de una variable y esta puede guardar un número, luego un texto y luego un arreglo sin problema. Además es un lenguaje del lado del servidor: tu código PHP corre en el servidor (Apache, Nginx, etc.) y el navegador del usuario solo recibe el HTML ya generado. Por eso la opción a) es falsa (el tipado estático es de Java o C#), la opción b) es falsa (en el navegador corre JavaScript, no PHP) y la opción d) es falsa (PHP es interpretado: el servidor lo ejecuta línea por línea sin una compilación previa como la de Java o C++).

RESPUESTA 4: b) 15

PHP hace conversión automática de tipos (coerción): cuando sumas un número con una cadena que contiene un número válido, convierte la cadena a número. Entonces "5" se vuelve

5 y $10 + 5 = 15$. La opción a) y la d) confunden la suma con la concatenación: si el código fuera `$a . $b` (con punto), el resultado si sería "105" como texto. La opción c) es falsa porque PHP no se queja de mezclar número y cadena numerica en una suma. Ojo: si la cadena no fuera numerica (por ejemplo "hola"), PHP la convertiría a 0 y el resultado sería 10, posiblemente con una advertencia según la versión.

RESPUESTA 5: b) Las dobles interpretan variables dentro del texto, las simples no

Esta es una de las reglas de oro de PHP: con comillas dobles, "Hola \$nombre" muestra el valor de la variable; con comillas simples, 'Hola \$nombre' muestra literalmente el texto \$nombre. Por eso la opción a) es falsa: no son intercambiables cuando hay variables de por medio. La opción c) invierte la regla y es falsa. La opción d) no tiene sentido: ambos tipos de comillas sirven para cualquier texto, incluido HTML. Como regla práctica, usa comillas simples para texto fijo (es un poco más rápido porque PHP no busca variables dentro) y dobles cuando necesites interpolar valores.

RESPUESTA 6: a) 2

El operador % es el módulo: devuelve el residuo de una división entera, no el cociente. 8 dividido entre 3 cabe 2 veces ($2 \times 3 = 6$) y sobran 2, así que $8 \% 3 = 2$. La opción b) confunde el módulo con la división decimal: $8 / 3$ si daría 2.66, pero eso es con la diagonal, no con %. La opción c) y la d) son residuos de otras divisiones. El módulo es muy útil en la práctica: por ejemplo, `$numero \% 2 == 0` te dice si un número es par, y se usa mucho para alternar colores en tablas o repartir elementos en columnas. Otro uso común es limitar valores a un rango: `$segundos \% 60` te da los segundos dentro del minuto actual. Y en paginación, `($pagina - 1) \% $porPagina` ayuda a calcular posiciones. Siempre que necesites "cada N elementos haz algo distinto", el módulo es tu herramienta.

RESPUESTA 7: b) 5 - 6

Aquí está la trampa clásica del post-incremento: `$y = $x++` primero asigna a \$y el valor ACTUAL de \$x (qué es 5) y DESPUÉS incrementa \$x a 6. Por eso al hacer echo, \$y vale 5 y \$x vale 6, imprimiendo "5 - 6". La opción a) sería correcta con pre-incremento (`++$x`), donde primero se incrementa y luego se asigna, dejando ambos en 6. La opción c) invierte los valores y la d) ignora el incremento por completo. Recuerda: el ++ después de la variable significa "usame primero, incrementame después"; antes de la variable significa "incrementame primero, usame después".

RESPUESTA 8: c) ===

El operador === se llama "identico" y compara valor Y tipo de dato al mismo tiempo: `5 === "5"` es falso porque uno es entero y el otro es cadena, aunque "valgan" lo mismo. El operador ==

(opción a) solo compara el valor después de convertir tipos, así que `5 == "5"` es verdadero. El `=` (opción b) no compara nada: es el operador de asignación. El `!=` (opción d) es desigualdad simple y tampoco revisa el tipo (para eso existe `!==`). En código real, usar `===` te salva de errores sutiles, por ejemplo al comparar el resultado de funciones que pueden devolver 0 o false.

RESPUESTA 9: b) true

La clave es la precedencia de operadores: `&&` se evalúa antes que `||`. Primero se resuelve `(true && false)`, que da false porque `&&` exige que AMBAS partes sean verdaderas. Luego queda `(false || true)`, y como `||` se conforma con que UNA parte sea verdadera, el resultado final es true. La opción a) saldría si toda la expresión fuera una sola conjunción `&&`. No hay ningún error lógico (opción c) ni valor null (opción d): es una expresión booleana perfectamente válida. Cuando mezcles `&&` y `||`, usa paréntesis para dejar clara tu intención y no depender de memorizar la precedencia.

RESPUESTA 10: a) 80

Se resuelve por orden de operaciones: primero el paréntesis. Dentro del paréntesis, $\$precio * \$descuento / 100 = 100 * 20 / 100 = 20$ (la multiplicación y división se hacen de izquierda a derecha). Luego $100 - 20 = 80$. Es simplemente un descuento del 20 por ciento sobre 100. La opción b) es solo el monto del descuento, no el total. La opción c) saldría de un cálculo mal hecho y la d) de restar mal. Fíjate que los paréntesis son importantes: sin ellos, la expresión $\$precio - \$precio * \$descuento / 100$ daría lo mismo por precedencia, pero escribirlos hace la intención obvia y evita errores. En PHP (como en matemáticas) la multiplicación y la división se resuelven antes que la suma y la resta, y entre ellas se va de izquierda a derecha. Cuando dudes, pon paréntesis de más: no cuestan nada y documentan tu intención. El código se lee más veces de las que se escribe, así que la claridad siempre gana.

RESPUESTA 11: b) Aprobado

`$nota` vale 7. La primera condición `( $nota >= 9 )` es falsa, así que se salta su bloque. La segunda condición `( $nota >= 6 )` es verdadera, por lo que imprime "Aprobado" y el resto (el `else`) ya no se evalúa. El `if/elseif/else` funciona en cascada: entra al PRIMER bloque cuya condición sea verdadera y ignora los demás. La opción a) requeriría una nota de 9 o más, y la c) una nota menor que 6. La opción d) es falsa: la sintaxis es correcta. Un error común es poner las condiciones en desorden (por ejemplo, revisar `>= 6` antes que `>= 9`); si lo hicieras así, un 9 entraría al primer bloque y nunca llegaría al de "Excelente".

RESPUESTA 12: b) Para evitar que se ejecuten los casos siguientes

El break sale del switch en cuanto termina el caso que coincidió. Sin break ocurre el "fall-through": PHP sigue ejecutando los case de abajo aunque no coincidan con el valor, lo cual casi siempre es un error y produce salidas duplicadas o incorrectas. La opción a) es falsa: para terminar el programa se usa exit o die, no break. La opción c) no existe: break nunca regresa atrás. La opción d) es peligrosa: omitir el break cambia el comportamiento del programa, así que no es decorativo. Solo se omite a propósito cuando quieres que varios casos compartan el mismo código, y aun así conviene dejar un comentario que lo aclare.

RESPUESTA 13: b) Asigna "Mayor" si \$edad es 18 o más, si no asigna "Menor"

El operador ternario es un if compacto con la forma condición ? valor_si_verdadero : valor_si_falso. Aquí evalúa (\$edad >= 18): si es verdadero, \$mensaje recibe "Mayor"; si es falso, recibe "Menor". La opción a) es falsa: no se suma nada, el 18 es parte de la comparación. La opción c) es falsa: "Mayor" y "Menor" son cadenas de texto, no variables. La opción d) es falsa: el ternario es sintaxis válida y no necesita un if alrededor. El ternario es ideal para asignaciones simples; si la lógica crece, mejor usa un if normal para que el código siga legible.

RESPUESTA 14: b) Falso

En PHP la condición del if SIEMPRE debe ir entre paréntesis: if (\$x > 5) { ... }. Escribir if \$x > 5 { ... } produce un error de sintaxis (parse error) y el script no corre. Esta regla es estricta en PHP, aunque en otros lenguajes los paréntesis sean opcionales. Lo mismo aplica para while, for, foreach y switch: la expresión de control va entre paréntesis. Es uno de esos detalles que el intérprete no perdona, así que acostúmbrate a escribir los paréntesis automáticamente. La única excepción aparente es la sintaxis alternativa con dos puntos (if (\$x > 5): ... endif;), donde los paréntesis siguen siendo obligatorios y lo que cambia son las llaves. Si algún día ves código sin paréntesis, es de otro lenguaje, no de PHP.

RESPUESTA 15: b) Invitado

El operador ?? se llama "fusión null" (null coalescing): devuelve el valor de la izquierda si existe y no es null; de lo contrario devuelve el de la derecha. Como la URL no trae el parámetro "nombre", \$_GET["nombre"] no existe y la expresión devuelve "Invitado" sin lanzar ninguna advertencia. Sin el ??, acceder a \$_GET["nombre"] generaría el aviso "Undefined array key". La opción c) es falsa: no devuelve cadena vacía, devuelve el valor de respaldo que tú definas. La opción d) es falsa: justamente el ?? existe para evitar que te quedes con null. Es el operador favorito para leer parámetros opcionales de formularios y URLs.

RESPUESTA 16: b) 5 veces

El bucle for tiene tres partes: $i = 1$ (inicio), $i \leq 5$ (condición para seguir) y $i++$ (incremento). Los valores que toma i son 1, 2, 3, 4 y 5: cinco iteraciones, cinco echos. La opción a) olvida que el 1 también cuenta. La opción c) asumiría que el bucle llega hasta 6, pero cuando i vale 6 la condición $i \leq 5$ es falsa y el bucle ya terminó. La opción d) sería cierta solo si la condición nunca se volviera falsa (por ejemplo, si olvidaras el $i++$), pero aquí el incremento garantiza que el bucle termina. Para contar iteraciones, lista mentalmente los valores: del 1 al 5 inclusive hay 5.

RESPUESTA 17: a) 321

Sigamos el bucle paso a paso: n empieza en 3. Vuelta 1: $3 > 0$ es verdadero, imprime "3", n baja a 2. Vuelta 2: $2 > 0$ es verdadero, imprime "2", n baja a 1. Vuelta 3: $1 > 0$ es verdadero, imprime "1", n baja a 0. Vuelta 4: $0 > 0$ es falso, el bucle termina. Resultado: "321" todo pegado porque el echo no agrega espacios ni saltos de línea. La opción b) empezaría en 2 y la c) iría en orden ascendente, lo cual no corresponde a un decremento. La opción d) imprimiría el 0, pero el 0 nunca se imprime porque la condición se revisa ANTES de entrar al cuerpo.

RESPUESTA 18: b) Salta a la siguiente iteración sin terminar la actual

continue abandona lo que falta del cuerpo en la vuelta actual y salta directo a la siguiente iteración del bucle. Es útil para "saltarse" casos: por ejemplo, dentro de un foreach puedes poner `if ($valor < 0) { continue; }` para ignorar los negativos. La opción a) describe a break, que si termina el bucle por completo. La opción c) no existe: ningún bucle se reinicia solo con una palabra. La opción d) confunde continue con sleep(1), que si pausa la ejecución un segundo. Recuerda el par: break rompe el bucle, continue salta la vuelta.

RESPUESTA 19: a) Verdadero

El do...while tiene una característica única: ejecuta el cuerpo PRIMERO y revisa la condición DESPUÉS. Eso garantiza al menos una ejecución, aunque la condición sea falsa desde el inicio. Ejemplo: $n = 10$; `do { echo $n; } while ($n < 5)`; imprime "10" una vez aunque $10 < 5$ sea falso. En cambio, un while normal revisa antes de entrar y podría no ejecutarse nunca. Usa do...while cuando necesites que algo ocurra por lo menos una vez, como mostrar un menu antes de preguntar si el usuario quiere continuar. Un ejemplo típico: pedir una contraseña y repetir mientras sea incorrecta, mostrando el prompt al menos una vez. Otro: lanzar un dado y seguir mientras salga 6, tirando siempre la primera vez. Si usas un while normal en esos casos, tendrías que duplicar el código antes del bucle, lo cual es más propenso a errores.

RESPUESTA 20: b) 10

Recorramos el foreach: primera vuelta, \$valor = 10, no es igual a 20, así que imprime "10 ". Segunda vuelta, \$valor = 20, la condición del if se cumple y el break sale del bucle DE INMEDIATO, sin imprimir el 20 y sin llegar al 30. Por eso la salida es solo "10 ". La opción a) ignoraría el break, la c) empezaría donde no debe y la d) imprimiría el 20 antes de romper, pero el break esta antes del echo. Fíjate en el orden: el if con break va primero, así que el 20 nunca llega a imprimirse.

RESPUESTA 21: b) Hola, amigo

El parámetro \$nombre tiene un valor por defecto ("amigo"), así que llamar a saludar() sin argumentos es perfectamente valido: PHP usa el valor por defecto. La función devuelve "Hola, amigo" (las comillas dobles interpolan la variable) y el echo lo imprime. La opción a) saldría si el valor por defecto fuera una cadena vacía. La opción c) es falsa: solo habría error si el parámetro NO tuviera valor por defecto y lo omitieras. La opción d) saldría con comillas simples, que no interpretan variables. Los parámetros con valor por defecto siempre van al final de la lista de parámetros; poner uno obligatorio después de uno opcional es un error.

RESPUESTA 22: b) echo imprime en pantalla y return devuelve un valor a quien llamó la función

Son dos mecanismos con propósitos distintos: echo envía texto a la salida (lo que verá el navegador), mientras que return entrega un valor de vuelta al código que invocó la función, para que lo guarde en una variable o lo use en una expresión. Una función puede hacer echo sin devolver nada útil (devuelve null implícitamente), o puede devolver un valor con return sin imprimir nada. La opción c) invierte los roles. La opción d) es falsa: echo funciona donde sea. Como buena práctica, las funciones que calculan algo deben usar return (así son reutilizables y fáciles de probar), y el echo se deja para la capa de presentación. Un síntoma de código mal organizado es una función que hace echo a la mitad de un cálculo: después no puedes reusarla en otro contexto (por ejemplo, en una API que devuelve JSON) sin que imprima basura. Separa siempre la lógica (return) de la presentación (echo) y tu código crecerá sano.

RESPUESTA 23: b) Error o advertencia porque \$numero no existe dentro de la función

Cada función en PHP tiene su propio alcance (scope): las variables definidas fuera NO se ven dentro. Dentro de duplicar(), \$numero no está definida, así que la línea \$numero = \$numero * 2 genera una advertencia de variable no definida y el resultado no es 20. La opción a) asumiría que la función ve la variable externa, lo cual es falso. Para usar el valor de afuera hay dos caminos correctos: pasarlo como parámetro (function duplicar(\$numero)) o declararlo global dentro de la función (global \$numero;), aunque pasar parámetros es la práctica recomendada porque evita efectos secundarios sorpresa.

RESPUESTA 24: a) Verdadero

PHP soporta funciones de orden superior: puedes pasar el nombre de una función como cadena (`array_map("strtoupper", $lista)`), usar funciones anonimas (`function($x) { return $x * 2; }`) o funciones flecha (`fn($x) => $x * 2`). El tipo callable en un parámetro indica que se espera algo invocable. Esto es la base de los callbacks: por ejemplo, `usort($datos, function($a, $b) { ... })` recibe una función que define como ordenar. Así que la afirmación es verdadera y es una técnica que verás en frameworks y librerías reales. Otro ejemplo cotidiano: `array_filter($edades, fn($e) => $e >= 18)` usa una función flecha para quedarse solo con los mayores de edad. Los frameworks modernos están llenos de callbacks para rutas, eventos y validaciones, así que esta no es una curiosidad teorica: es una habilidad que usaras a diario.

RESPUESTA 25: b) 24

Desenredemos la recursión: `factorial(4) = 4 * factorial(3)`; `factorial(3) = 3 * factorial(2)`; `factorial(2) = 2 * factorial(1)`; `factorial(1)` devuelve 1 porque cumple el caso base (`$n <= 1`). Sustituyendo hacia atras: `4 * 3 * 2 * 1 = 24`. La opción a) y la c) son productos parciales mal calculados. La opción d) es falsa: el caso base detiene la recursión, así que no hay bucle infinito. Toda función recursiva necesita dos cosas: un caso base que frene y un paso que se acerque a ese caso base; si falta el caso base, ahí si tendrías recursión infinita y un error de desbordamiento.

RESPUESTA 26: b) \$colores[0]

Los arreglos numericos en PHP usan índices que empiezan en 0: el primer elemento ("rojo") está en la posición 0, "verde" en la 1 y "azul" en la 2. Por eso la opción a) devolveria "verde", no "rojo". La opción c) usa sintaxis de arreglo asociativo con clave de texto, pero este arreglo no tiene claves de texto, así que daría un aviso de índice no definido. La opción d) usa la sintaxis de flecha de los objetos (`->`), que no aplica a los arreglos. El famoso "error por uno" (off-by-one) nace justo de olvidar que se empieza en cero: si quieres el último de 3 elementos, es el índice 2, no el 3.

RESPUESTA 27: b) 22

Es un arreglo asociativo: cada valor se guarda bajo una clave de texto y se recupera con esa misma clave. `$alumno["edad"]` devuelve 22 y `$alumno["nombre"]` devolveria "Ana". La opción a) corresponderia a la clave "nombre". La opción c) es falsa: la sintaxis con `=>` es la forma correcta de crear arreglos asociativos en PHP. La opción d) confunde la clave con el valor. Los arreglos asociativos son la estructura más usada en PHP: `$_POST`, `$_GET` y las filas que trae PDO con `FETCH_ASSOC` son todos arreglos asociativos.

RESPUESTA 28: b) Devuelve 3, el número de elementos

count() cuenta cuantos elementos contiene un arreglo, sin importar que valores tengan. Aquí hay 3 frutas, así que imprime 3. La opción a) describe a array_sum(), que si suma los valores. La opción c) describe a end(), que devuelve el último elemento. La opción d) describe a sort(), que ordena. count() es indispensable para validar ("si el carrito tiene 0 productos, muestra un mensaje") y para controlar bucles for que recorren arreglos por índice. Ojo: count() sobre una variable que no es arreglo puede dar advertencia en versiones recientes de PHP.

RESPUESTA 29: a) Verdadero

array_push(\$lista, "nuevo") agrega "nuevo" al final del arreglo \$lista, aumentando su tamaño en uno. Es equivalente a escribir \$lista[] = "nuevo";, que es más corto y más común en código moderno. También puedes agregar varios de una vez: array_push(\$lista, "a", "b", "c"). La afirmación es verdadera sin matices. Lo contrario sería array_pop(), que quita y devuelve el último elemento. Estas dos funciones convierten un arreglo en una pila (stack): push mete, pop saca. Las pilas sirven para el historial de "deshacer", para evaluar expresiones matemáticas y para recorrer estructuras anidadas. Su contraparte es la cola (queue): array_shift() saca del inicio y array_unshift() mete al inicio. Push/pop trabajan al final del arreglo y son rapidísimos; shift/unshift trabajan al inicio y son más lentos porque deben reindexar. Elegir el extremo correcto es un detalle de rendimiento que los buenos programadores conocen.

RESPUESTA 30: b) leche cuesta 35

En un arreglo multidimensional el primer índice elige la "fila" y el segundo la "columna". \$tienda[1] es el segundo sub-arreglo, ["leche", 35]; de ahí, [0] es "leche" y [1] es 35. Resultado: "leche cuesta 35". La opción a) correspondería a \$tienda[0][0] y \$tienda[0][1] (la primera fila). La opción c) es falsa: los índices 1, 0 y 1 existen. La opción d) saldría si intentaras imprimir un arreglo completo con echo (PHP muestra la palabra "Array"), pero aquí se accede a valores escalares, no a arreglos. Los arreglos multidimensionales son la forma natural de representar tablas, matrices y datos anidados como respuestas JSON.

RESPUESTA 31: b) En \$_GET

Cuando un formulario usa method="get", el navegador pega los datos en la URL (página.php?nombre=Ana&edad=22) y PHP los pone automáticamente en el arreglo superglobal \$_GET. La opción a) es falsa: \$_POST solo se llena cuando el formulario usa method="post". La opción c) es confusa: \$_REQUEST combina GET, POST y COOKIE, pero decir "solamente si es POST" es incorrecto. La opción d) inventa una variable \$GLOBALS["form"] que no existe. Regla de memoria: el method del formulario dicta el arreglo superglobal; si no coinciden, tus variables llegarán vacías y pasarás un buen rato depurando.

RESPUESTA 32: b) GET muestra los datos en la URL y tiene limite de tamaño; POST los oculta del URL y admite más datos

GET viaja en la dirección: es visible, queda en el historial y tiene límite de longitud, por eso sirve para búsquedas y filtros que quieres poder compartir o marcar. POST viaja en el cuerpo de la petición: no se ve en la URL y admite mucho más volumen, por eso es el adecuado para registros, logins y subida de archivos. La opción a) es falsa: la diferencia es real e importante. La opción c) es un mito: ninguno es inherentemente más rápido. La opción d) es falsa: GET puede enviar texto sin problema. Nota de seguridad: POST no es "seguro" por sí solo, solo es menos visible; los datos sensibles siempre deben viajar por HTTPS.

RESPUESTA 33: b) Devuelve el email si tiene formato válido, o false si no lo tiene

`filter_var()` con el filtro `FILTER_VALIDATE_EMAIL` revisa que el texto tenga estructura de correo (`algo@dominio.com`). Si pasa la validación, devuelve el mismo email; si no, devuelve `false`, que puedes usar en un `if` para rechazar el dato. La opción a) es falsa: no encripta nada (eso lo hace `password_hash` para contraseñas). La opción c) es falsa: validar el formato no envía ningún correo. La opción d) es falsa: no guarda nada en la base de datos. `filter_var` es tu navaja suiza para validación: también tiene `FILTER_VALIDATE_INT`, `FILTER_VALIDATE_URL` y `FILTER_SANITIZE_STRING` entre otros.

RESPUESTA 34: a) Verdadero

Todo lo que llega por `$_GET` y `$_POST` lo controla el usuario: puede manipular la URL, editar el HTML del formulario o enviar peticiones falsas con herramientas. Ahí pueden venir scripts maliciosos, SQL inyectado o valores fuera de rango. Por eso la regla de oro es: nunca confíes en la entrada del usuario; siempre valida (verifica que sea lo esperado: tipo, formato, rango) y sanea (limpia o escapa caracteres peligrosos) antes de usarla o mostrarla. La opción b) es falsa y es justo la mentalidad que provoca las vulnerabilidades más comunes: XSS e inyección SQL nacen de confiar en datos sin revisar. Piénsalo así: tu formulario HTML es solo una sugerencia amable; el atacante ni siquiera usa tu formulario, manda la petición directamente con los valores que quiera. Por eso la validación del lado del servidor no es opcional: el JavaScript del navegador se puede desactivar, pero tu PHP siempre está ahí para poner orden. Valida tipo, formato, longitud y rango; luego sanea o escapa según a donde vayan los datos.

RESPUESTA 35: b) Convierte caracteres especiales como < y > en entidades HTML, previniendo ataques XSS

`htmlspecialchars()` transforma `<` en `<`, `>` en `>`, `&` en `&`; y las comillas en sus entidades. Así, si un usuario escribe `<script>alert("x")</script>` en un comentario, el navegador lo muestra

como texto inofensivo en vez de ejecutarlo. Eso bloquea el XSS (cross-site scripting), donde un atacante inyecta JavaScript en tu página para robar sesiones o redirigir usuarios. La opción a) es falsa: las contraseñas se protegen con `password_hash()`, no con esto. La opción c) es falsa: los emails se validan con `filter_var()`. La opción d) es falsa: no comprime nada. Regla: todo dato de origen dudoso que imprimas en HTML debe pasar por `htmlspecialchars()`.

RESPUESTA 36: b) La cookie se guarda en el navegador del usuario; la sesión guarda los datos en el servidor y solo una identificación viaja en la cookie

La cookie vive en el equipo del cliente: el usuario puede verla, editarla y borrarla, así que jamás debes guardar ahí datos sensibles como `"es_admin = true"`. La sesión, en cambio, guarda los datos en el servidor (archivos o base de datos) y el navegador solo conserva un ID de sesión en una cookie; con ese ID, PHP recupera tus variables de `$_SESSION`. Por eso el login usa sesiones. La opción a) es falsa: son mecanismos distintos. La opción c) invierte los lugares. La opción d) es falsa: las cookies si guardan texto (hasta unos 4 KB). Usa cookies para preferencias (tema oscuro, idioma) y sesiones para estado sensible (usuario logueado). Técnicamente, la sesión también usa una cookie, pero solo para el ID: se llama cookie de sesión y normalmente se borra al cerrar el navegador. Si un atacante roba ese ID (por ejemplo, en una red sin HTTPS), podría suplantar al usuario, por eso los sitios serios regeneran el ID al iniciar sesión y usan HTTPS siempre. Entender esta división cliente/servidor es la base de toda la seguridad web.

RESPUESTA 37: b) Inicia o reanuda la sesión; debe ir antes de cualquier salida HTML

`session_start()` activa el manejo de sesiones: si el navegador trae un ID de sesión válido, reanuda esa sesión; si no, crea una nueva. Cómo trabaja con cabeceras HTTP, debe ejecutarse ANTES de imprimir cualquier cosa: ni un espacio, ni un echo, ni HTML antes. Si lo pones después, PHP lanza el clásico error "headers already sent". La opción a) es falsa: para cerrar se usa `session_destroy()`. La opción c) es falsa: para borrar variables se vacía `$_SESSION`. La opción d) es falsa: crear cookies se hace con `setcookie()`, y tampoco puede ir después de la salida. Pon `session_start()` como primera línea de tus scripts y te evitaras dolores de cabeza.

RESPUESTA 38: b) `$_SESSION["usuario"] = $nombre;`

El arreglo superglobal de sesión se escribe `$_SESSION`, con signo \$ y guion bajo. La opción a) omite el guion bajo (`$SESSION`) y PHP lo trataría como una variable normal que se pierde al terminar el script. La opción c) inventa una función `session()` que no existe con esa firma. La opción d) guarda el dato en `$_COOKIE`, que vive en el navegador del cliente: es insegura para datos de sesión y además requiere `setcookie()` con sus reglas de cabeceras. Recuerda: `$_SESSION` solo funciona después de llamar a `session_start()` al inicio del script.

RESPUESTA 39: b) Falso

Hacer `$_SESSION = []` solo vacía el arreglo de variables en memoria durante esa petición, pero la sesión sigue existiendo en el servidor y la cookie con el ID de sesión sigue en el navegador del usuario. Un cierre de sesión correcto incluye: vaciar `$_SESSION`, llamar a `session_destroy()` para eliminar los datos del servidor y, idealmente, borrar la cookie del ID de sesión. Sin `session_destroy()`, la sesión queda técnicamente viva y podría reutilizarse. Es como salir de una cuenta solo cerrando la pestaña: los datos de la sesión siguen ahí.

RESPUESTA 40: b) Con `password_verify($claveEscrita, $hashGuardado)`, porque la contraseña se guardó con `password_hash()`

Las contraseñas nunca se guardan en texto plano: al registrar al usuario se guardan con `password_hash()`, que genera un hash irreversible con sal automática. Para verificar el login, `password_verify()` toma la clave que escribió el usuario, la procesa con el mismo algoritmo y la compara con el hash guardado, devolviendo true o false. La opción a) es un grave error de seguridad: comparar texto plano significa que guardaste la contraseña legible. La opción c) es obsoleta: md5 es rapidísimo de romper con tablas arcoiris y no debe usarse para contraseñas. La opción d) es falsa: claro que se puede verificar, justamente para eso existen las funciones de hash. Un detalle importante: `password_hash()` genera un hash distinto cada vez aunque la contraseña sea la misma (usa una sal aleatoria), por eso no puedes comparar dos hashes directamente con `==`; siempre debes usar `password_verify()`. Y si algún día PHP mejora el algoritmo, `password_needs_rehash()` te avisa que debes actualizar el hash guardado en el próximo login del usuario.

RESPUESTA 41: Código de conexión PDO con manejo de errores:

```
<?php
try {
    $pdo = new PDO("mysql:host=localhost;dbname=tienda;charset=utf8", "root", "");
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    echo "Conexion exitosa";
} catch (PDOException $e) {
    echo "Error de conexion: " . $e->getMessage();
}
?>
```

Explicación línea por línea:

- El bloque try encierra el código que puede fallar; si algo sale mal, el flujo salta al catch en vez de romper la página con un error fatal sin control.

- new PDO(...) crea la conexión. El primer argumento es el DSN (Data Source Name): mysql indica el driver, host=localhost el servidor, dbname=tienda la base de datos y charset=utf8 el juego de caracteres para que los acentos y la ñ se guarden bien.
- El segundo argumento es el usuario ("root") y el tercero la contraseña ("" vacía en este ejercicio, como en un XAMPP recién instalado).
- setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION) es crucial: hace que PDO lance excepciones ante cualquier error SQL en vez de fallar en silencio, lo que te permite detectar y manejar los problemas.
- El catch atrapa la excepción de tipo PDOException y getMessage() devuelve el motivo del fallo (servidor apagado, base inexistente, credenciales malas, etc.).
- Errores comunes que debes evitar: escribir mal el nombre de la base en el DSN, olvidar el charset (los acentos se ven como símbolos raros) y no usar try/catch, dejando que el usuario vea un error técnico crudo.

Para practicar más: modifica el código para conectarte a una base llamada "blog" con usuario "admin" y contraseña "secreta123", y agrega una línea que configure el modo de obtención por defecto a FETCH_ASSOC con \$pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);. Así ya no tendrás que escribir el modo en cada fetch. Después provoca un error a propósito (por ejemplo, escribe mal el nombre de la base) y observa el mensaje que muestra el catch: aprender a leer esos mensajes es la mitad de la depuración en PHP.

RESPUESTA 42: Inserción con sentencia preparada y marcadores con nombre:

```
<?php
$sql = "INSERT INTO productos (nombre, precio, stock) VALUES (:nombre, :precio, :stock)";
$stmt = $pdo->prepare($sql);
$stmt->execute([
    ":nombre" => $nombre,
    ":precio" => $precio,
    ":stock" => $stock
]);
echo "Producto insertado con id: " . $pdo->lastInsertId();
?>
```

Explicación línea por línea:

- El SQL usa marcadores con nombre (:nombre, :precio, :stock) en lugar de pegar los valores directamente en la cadena. Los dos puntos identifican cada marcador.

- `$pdo->prepare($sql)` compila la sentencia en el servidor de base de datos: MySQL entiende la estructura del INSERT antes de conocer los valores.
- `execute()` recibe un arreglo asociativo que liga cada marcador con su variable; PDO se encarga de enviar los valores de forma segura, escapando todo lo necesario.
- Esta separación entre código SQL y datos es exactamente lo que impide la inyección SQL: aunque `$nombre` contenga comillas o código malicioso, se tratara como un simple texto a guardar.
- `lastInsertId()` devuelve el id autoincremental que MySQL asigno a la fila nueva; es muy útil para confirmar la inserción o para usar ese id en operaciones siguientes.
- Error comun: olvidar los dos puntos en el arreglo de `execute()` o escribir un marcador en el SQL que no existe en el arreglo (o viceversa); PDO lanzara una excepción y la inserción no se hara.

Para practicar más: escribe la variante que inserta un usuario en la tabla usuarios (nombre, email, password), donde la contraseña llegue ya procesada con `password_hash()` en una variable `$hash`. Luego escribe otra versión usando marcadores de posición (?) en vez de marcadores con nombre, pasando los valores como un arreglo simple en `execute([$nombre, $precio, $stock])`. Compara ambas: los marcadores con nombre son más legibles cuando hay muchos parámetros, pero los de posición son más cortos. Domina las dos formas porque las verás en código ajeno.

RESPUESTA 43: Listar productos con protección XSS:

```
<?php
$stmt = $pdo->prepare("SELECT nombre, precio FROM productos");
$stmt->execute();
$productos = $stmt->fetchAll(PDO::FETCH_ASSOC);
?>
<ul>
<?php foreach ($productos as $p): ?>
    <li><?php echo htmlspecialchars($p["nombre"]); ?> - $<?php echo
htmlspecialchars($p["precio"]); ?></li>
<?php endforeach; ?>
</ul>
```

Explicación línea por línea:

- Se prepara el SELECT y se ejecuta; aunque no lleva parámetros, usar `prepare()` mantiene un estilo consistente y seguro en todo el proyecto.

- `fetchAll(PDO::FETCH_ASSOC)` trae TODAS las filas de una vez como un arreglo de arreglos asociativos, donde cada clave es el nombre de la columna ("nombre", "precio").
- El `foreach` recorre cada producto; se usa la sintaxis alternativa con `:` y `endforeach`, muy cómoda cuando mezclas PHP con HTML.
- `htmlspecialchars($p["nombre"])` convierte `<`, `>` y comillas en entidades HTML: si un producto se llamara `<script>alert(1)</script>`, se mostraria como texto en vez de ejecutarse. Esa es la defensa contra XSS.
- El precio también se protege por disciplina: todo dato que venga de la base de datos y se imprima en HTML debe pasar por `htmlspecialchars()`, sin excepciones.
- Error comun: imprimir `$p["nombre"]` directo con `echo`. Funciona con datos "normales", pero el día que un dato traiga HTML o JavaScript, tu página queda comprometida.

Para practicar más: modifica el código para mostrar los productos en una tabla HTML con columnas de nombre, precio y stock, y agrega un buscador: un formulario GET con un campo "q" que filtre con `WHERE nombre LIKE :q`, ligando `"%" . $q . "%"` al marcador. Recuerda proteger también el terminó buscado cuando lo muestres de vuelta en el campo del formulario con `value="<?php echo htmlspecialchars($q); ?>"`. Este ejercicio combina formularios, consultas con parámetros y XSS en un solo caso real.

RESPUESTA 44: Función para registrar un pedido y actualizar el stock:

```

<?php
function registrarPedido($pdo, $usuarioId, $productoId, $cantidad) {
    try {
        $sql = "INSERT INTO pedidos (usuario_id, producto_id, cantidad, fecha)
                VALUES (:uid, :pid, :cant, NOW())";
        $stmt = $pdo->prepare($sql);
        $stmt->execute([
            ":uid" => $usuarioId,
            ":pid" => $productoId,
            ":cant" => $cantidad
        ]);

        $sql2 = "UPDATE productos SET stock = stock - :cant WHERE id = :pid";
        $stmt2 = $pdo->prepare($sql2);
        $stmt2->execute(["cant" => $cantidad, "pid" => $productoId]);

        return true;
    } catch (PDOException $e) {
        return false;
    }
}
?>

```

Explicación línea por línea:

- La función recibe la conexión PDO y los tres datos del pedido como parámetros, lo que la hace reutilizable desde cualquier parte del proyecto.
- El INSERT guarda el pedido en la tabla pedidos; NOW() es una función de MySQL que pone la fecha y hora actual del servidor, así no tienes que generar la fecha en PHP ni preocuparte por el formato.
- El UPDATE resta la cantidad directamente en la base de datos (stock = stock - :cant): hacerlo en SQL es atómico y evita condiciones de carrera que ocurrirían si leyeras el stock en PHP, restaras y luego guardarás.
- Ambas sentencias usan marcadores con nombre y arreglos en execute(), quedando blindadas contra inyección SQL.
- Todo va dentro de try/catch: si alguna consulta falla (producto inexistente, cantidad inválida, base caída), se atrapa la PDOException y se devuelve false; si ambas salen bien, se devuelve true.

- Mejora profesional: envolver las dos operaciones en una transacción (beginTransaction/commit/rollback) para que si el UPDATE falla después del INSERT, el pedido se deshaga y no queden datos a medias.

Para practicar más: reescribe la función usando transacciones de verdad: \$pdo->beginTransaction() antes del INSERT, \$pdo->commit() al final si todo salio bien y \$pdo->rollBack() dentro del catch. Luego agrega una validación previa: consulta el stock actual del producto y devuelve false con un mensaje si la cantidad pedida es mayor que el stock disponible. Así tu función ya no solo registra, sino que protege la lógica del negocio, que es lo que distingue a un programador que copia código de uno que entiende el problema.

RESPUESTA 45: Subida de imagen con validaciones:

```
<?php
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $archivo = $_FILES["foto"];
    $permitidos = ["image/jpeg", "image/png"];
    $maxBytes = 2 * 1024 * 1024;

    if ($archivo["error"] !== UPLOAD_ERR_OK) {
        echo "Error al subir el archivo.";
    } elseif (!in_array($archivo["type"], $permitidos)) {
        echo "Solo se permiten imagenes JPG o PNG.";
    } elseif ($archivo["size"] > $maxBytes) {
        echo "El archivo supera los 2 MB.";
    } else {
        $extension = pathinfo($archivo["name"], PATHINFO_EXTENSION);
        $nuevoNombre = "uploads/" . uniqid("img_") . "." . $extension;
        if (move_uploaded_file($archivo["tmp_name"], $nuevoNombre)) {
            echo "Imagen guardada como: " . htmlspecialchars($nuevoNombre);
        } else {
            echo "No se pudo guardar el archivo.";
        }
    }
}
?>
```

Explicación línea por línea:

- Primero se verifica que la petición sea POST; sin esto, el código intentaría procesar un archivo que no existe cuando alguien abre la página directamente.
- \$_FILES["foto"] es un arreglo con cinco claves: name (nombre original), type (tipo MIME), size (tamaño en bytes), tmp_name (ruta temporal) y error (código de error).

- Se revisa que error sea UPLOAD_ERR_OK (cero significa "todo bien"); cualquier otro valor indica que la subida falló (archivo muy grande para el servidor, subida parcial, etc.).
- in_array() verifica que el tipo MIME este en la lista de permitidos (jpeg o png); el tamaño se compara contra 2 MB expresados en bytes ($2 * 1024 * 1024 = 2097152$).
- pathinfo() extrae la extensión del nombre original y uniqid("img_") genera un nombre único con prefijo, evitando que dos archivos con el mismo nombre se sobrescriban.
- move_uploaded_file() es la ÚNICA forma segura de guardar el archivo: verifica que realmente venga de una subida HTTP antes de moverlo del temporal a uploads/.
- El nombre mostrado pasa por htmlspecialchars() porque el nombre original lo eligió el usuario y podría contener HTML malicioso.
- Nota de seguridad extra: en producción también conviene verificar la extensión contra una lista blanca y guardar fuera de la carpeta pública o renombrar siempre, porque el tipo MIME también puede falsificarse.

Para practicar más: mejora el código verificando la extensión con una lista blanca (solo "jpg", "jpeg" y "png" en minúsculas usando strtolower()), y rechaza nombres de archivo con doble extensión como "foto.php.jpg". Después investiga la función getimagesize(): devuelve las dimensiones reales de una imagen y falla si el archivo no es una imagen de verdad, lo cual es una validación mucho más fuerte que el tipo MIME. Con esas tres capas (MIME, extensión y contenido real), tu subida queda bien protegida.

RESPUESTA 46: Login completo y seguro:

```

<?php
session_start();

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $email = $_POST["email"];
    $password = $_POST["password"];

    $stmt = $pdo->prepare("SELECT id, nombre, password FROM usuarios WHERE email = :email");
    $stmt->execute(["email" => $email]);
    $usuario = $stmt->fetch(PDO::FETCH_ASSOC);

    if ($usuario && password_verify($password, $usuario["password"])) {
        $_SESSION["usuario_id"] = $usuario["id"];
        $_SESSION["usuario_nombre"] = $usuario["nombre"];
        header("Location: panel.php");
        exit;
    } else {
        echo "Credenciales invalidas";
    }
}
?>

```

Explicación línea por línea:

- session_start() va en la primera línea, antes de cualquier salida HTML, para poder leer y escribir \$_SESSION.
- Se leen email y password del POST; en un sistema real aquí también validarías que el email tenga formato válido.
- La sentencia preparada busca al usuario por email con el marcador :email, así un email malicioso como ' OR '1'='1 no puede alterar la consulta.
- fetch() trae una sola fila como arreglo asociativo, o false si el email no existe en la tabla.
- La condición verifica dos cosas: que el usuario exista (\$usuario no sea false) y que password_verify() confirme la clave contra el hash guardado. El && en cortocircuito evita llamar a password_verify con datos inexistentes.
- Si todo coincide, se guardan el id y el nombre en la sesión: esas variables identificarán al usuario en las siguientes páginas sin pedirle la clave de nuevo.
- header("Location: panel.php") redirige al panel; el exit inmediato es obligatorio para que el script no siga ejecutándose después de la redirección.

- El mensaje de error es generico a proposito ("Credenciales invalidas"): no revela si falló el email o la contraseña, para no darle pistas a un atacante sobre que cuentas existen.

Para practicar más: agrega protección contra intentos masivos agregando un contador en la sesión (\$_SESSION["intentos"]) que se incremente con cada falló y bloquee el formulario por 5 minutos al llegar a 5 intentos. También agrega session_regenerate_id(true) después del login exitoso: genera un ID de sesión nuevo y evita ataques de fijación de sesión. Y en el registro de usuarios, nunca olvides guardar la contraseña con password_hash(\$password, PASSWORD_DEFAULT) en vez del texto plano: el login solo funciona si el registro se hizo bien.

RESPUESTA 47: Actualizar el precio con validación:

```
<?php
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $id = $_POST["id"];
    $precio = $_POST["precio"];

    if (!is_numeric($precio) || $precio <= 0) {
        echo "El precio debe ser un numero mayor que cero.";
    } else {
        $stmt = $pdo->prepare("UPDATE productos SET precio = :precio WHERE id = :id");
        $stmt->execute([":precio" => $precio, ":id" => $id]);
        echo "Filas afectadas: " . $stmt->rowCount();
    }
}
?>
```

Explicación línea por línea:

- Se reciben el id y el nuevo precio por POST, como indica el ejercicio.
- La validación usa is_numeric() para comprobar que el precio sea un número (acepta "19.99" como texto numerico) y exige que sea mayor que cero; si no cumple, se muestra el error y la base de datos ni se toca.
- Esta validación previa es la que impide guardar precios absurdos: cero, negativos o texto como "gratis" que romperian la lógica de la tienda.
- El UPDATE usa los marcadores :precio y :id para que los datos viajen separados del SQL, eliminando el riesgo de inyección SQL aunque el id venga manipulado.

- rowCount() informa cuantas filas se modificaron: será 1 si el producto con ese id existía, y 0 si no había ningún producto con ese id (o si el precio nuevo era idéntico al anterior, según el driver).
- Mostrar las filas afectadas le da retroalimentación al usuario: sabe si su cambio realmente se aplicó o si escribió un id inexistente.

Para practicar más: extiende el código para actualizar nombre y precio a la vez, validando que el nombre no venga vacío (usa trim() para quitar espacios y verifica que no sea ""). Luego escribe la versión que recibe los datos por GET en vez de POST y piensa en que cambia: ¿es correcto modificar datos con GET? (Pista: no, porque las URLs se pueden visitar por accidente, se guardan en el historial y los buscadores las siguen; las operaciones que cambian datos deben usar POST.) Entender el porqué de cada método es tan importante como saber usarlos.

RESPUESTA 48: Eliminar producto con explicación de inyección SQL:

```
<?php
function eliminarProducto($pdo, $id) {
    // Una sentencia preparada separa el SQL de los datos: el servidor compila
    // "DELETE FROM productos WHERE id = ?" primero y después recibe el valor
    // como dato puro. Así, si alguien manda '1 OR 1=1' como id, la base busca
    // un producto con ese texto literal en vez de ejecutar el OR como código,
    // y la inyección SQL queda completamente neutralizada.
    $stmt = $pdo->prepare("DELETE FROM productos WHERE id = :id");
    $stmt->execute(["id" => $id]);
    return $stmt->rowCount();
}
?>
```

Explicación línea por línea:

- La función recibe la conexión PDO y el id del producto a borrar, y devuelve el número de filas eliminadas para que quien la llame sepa si el borrado ocurrió.
- El comentario dentro del código explica el mecanismo de defensa: prepare() envía la estructura del SQL al servidor ANTES de conocer los valores, así MySQL compila "borra donde id sea un valor" como plan fijo.
- Cuando execute() envía el valor, la base de datos lo trata únicamente como dato, jamás como parte de la instrucción; no hay forma de que "escape" y se convierta en código SQL.

- El ataque clásico id = "1 OR 1=1" funciona solo cuando el valor se pega directo en la cadena SQL, porque el OR se vuelve parte de la condición y borra toda la tabla; con sentencia preparada, ese texto se busca literalmente como id y no coincide con nada.
- rowCount() devuelve 1 si el producto existía y se borró, o 0 si no había producto con ese id; la función lo retorna directamente.
- Lección general: jamás concatenes variables en un SQL. Prepare + execute es la forma correcta siempre, sin excepciones.

Para practicar más: escribe una función borrarPedidosAntiguos(\$pdo, \$fecha) que elimine los pedidos anteriores a una fecha dada usando un marcador para la fecha, y devuelva cuantos se borraron. Luego intenta "atacarla" tú mismo pasándole como fecha el texto "2020-01-01' OR '1'=1" y comprueba que no borra nada indebido: la sentencia preparada lo trata como un texto literal que no coincide con ninguna fecha. Hacerte tu propio atacante es la mejor forma de confiar de verdad en esta protección.

RESPUESTA 49: Pedidos del usuario logueado con JOIN:

```
<?php
session_start();

$stmt = $pdo->prepare(
    "SELECT p.nombre AS producto, pe.cantidad, pe.fecha
    FROM pedidos pe
    INNER JOIN productos p ON pe.producto_id = p.id
    WHERE pe.usuario_id = :uid
    ORDER BY pe.fecha DESC"
);
$stmt->execute(["uid" => $_SESSION["usuario_id"]]);
$pedidos = $stmt->fetchAll(PDO::FETCH_ASSOC);
?>
<ul>
<?php foreach ($pedidos as $ped): ?>
    <li>
        <?php echo htmlspecialchars($ped["producto"]); ?> -
        Cantidad: <?php echo htmlspecialchars($ped["cantidad"]); ?> -
        Fecha: <?php echo htmlspecialchars($ped["fecha"]); ?>
    </li>
<?php endforeach; ?>
</ul>
```

Explicación línea por línea:

- `session_start()` al inicio permite leer `$_SESSION["usuario_id"]`, que se guardó al momento del login y identifica al usuario actual.
- El `INNER JOIN` une la tabla pedidos (alias `pe`) con productos (alias `p`) a través de `pe.producto_id = p.id`, así cada fila del resultado trae el nombre del producto junto con los datos del pedido; sin el `JOIN` solo tendrías el número de producto, no su nombre.
- Los alias `AS producto` y los alias de tabla (`pe`, `p`) hacen la consulta más corta y el código PHP más legible (`$ped["producto"]` en vez de `$ped["nombre"]`).
- El `WHERE` filtra con el marcador `:uid` ligado al id de la sesión: cada usuario ve **UNICAMENTE** sus propios pedidos, lo cual es un requisito de privacidad básico.
- `ORDER BY pe.fecha DESC` ordena del pedido más reciente al más antiguo, como pide el ejercicio (`ASC` sería al revés).
- Todo lo que se imprime en el HTML pasa por `htmlspecialchars()`, incluyendo cantidad y fecha, porque la regla es no confiar en ningún dato que salga a pantalla.
- Si la lista sale vacía, el `foreach` simplemente no genera `<li>` y convendría mostrar un mensaje como "Aun no tienes pedidos".

Para practicar más: agrega al `SELECT` el total de cada pedido calculado en SQL (cantidad * precio `AS total`) y muestra una suma final con `array_sum()` o con `SUM()` en otra consulta. Después escribe la versión para el administrador: sin el `WHERE` de usuario, mostrando el nombre del usuario con otro `JOIN` a la tabla usuarios, pero protegida con una verificación de sesión (si `$_SESSION["rol"]` no es "admin", redirige al login). Así prácticas `JOINS` múltiples y control de acceso en el mismo ejercicio.

RESPUESTA 50: Checklist de despliegue y credenciales con variables de entorno:

```

<?php
// CHECKLIST PARA SUBIR TU PROYECTO PHP A UN HOSTING COMPARTIDO:
// 1. Sube todos los archivos por FTP a la carpeta public_html del hosting.
// 2. Crea la base de datos MySQL desde el panel (cPanel) y anota su nombre.
// 3. Crea un usuario MySQL con contraseña segura y dale permisos sobre esa base.
// 4. Importa tu archivo .sql con las tablas usando phpMyAdmin.
// 5. Configura las credenciales como variables de entorno en el panel del hosting,
//     nunca escritas directo en el código.
// 6. Prueba el sitio en la URL real y revisa el log de errores si algo falla.

$host = getenv("DB_HOST");
$db    = getenv("DB_NAME");
$user  = getenv("DB_USER");
$pass  = getenv("DB_PASS");

// Leer las credenciales con getenv() es mas seguro porque:
// - Si el código se sube por error a un repositorio público, las claves no van incluidas.
// - Cada ambiente (local, pruebas, producción) usa sus propias variables sin cambiar código.
// - El código fuente puede compartirse sin exponer los accesos a la base de datos.
$pdo = new PDO("mysql:host=$host;dbname=$db;charset=utf8", $user, $pass);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
?>

```

Explicación línea por línea:

- Los comentarios numeran los 6 pasos del despliegue en orden lógico: primero los archivos (FTP a public_html), luego la base de datos y su usuario dedicado con contraseña segura, después la importación de las tablas con phpMyAdmin, la configuración de credenciales y finalmente las pruebas en la URL real.
- Crear un usuario MySQL exclusivo para la app (paso 3) es mejor que usar root: si esas credenciales se filtran, el dano queda limitado a esa base.
- `getenv()` lee cada credencial desde el entorno del servidor; en el hosting se configuran en el panel o en un archivo `.env` que nunca se sube al repositorio.
- El comentario explica las tres ventajas de seguridad: las claves no viajan con el código, cada ambiente tiene las suyas sin modificar archivos y el proyecto puede compartirse o versionarse sin exponer accesos.

- La conexión PDO se arma exactamente igual que en desarrollo, solo que las variables vienen del entorno; el resto de tu código no necesita ningún cambio entre tu computadora y el servidor.
- Como cierre del curso: este patrón (código portable + configuración por entorno) es el estándar profesional para desplegar aplicaciones PHP.

Para practicar más: crea en tu computadora un archivo .env de prueba con las cuatro variables y un script que las lea con getenv() después de cargarlas (investiga como lo hace la librería vlucas/phpdotenv, la más usada en proyectos reales). Luego agrega al checklist un paso 7: configurar HTTPS con certificado gratuito, porque sin HTTPS las contraseñas y sesiones viajan legibles por la red. Y un paso 8: desactivar la muestra de errores en producción con display_errors = Off en el php.ini, para que un fallo no le regale información técnica a un atacante.

Guía para calificarte y repasar por unidad

Ya tienes tus respuestas. Ahora cuenta 2 puntos por cada acierto: 100 es la calificación perfecta. Si llegaste a 90 o más, excelente: dominas los fundamentos de PHP y puedes seguir al siguiente nivel con la conciencia tranquila. Si quedaste entre 70 y 88, vas bien pero tienes huecos: usa la guía de abajo para ubicar en qué unidades fallaste y repasalas antes de dar el tema por cerrado. Si sacaste menos de 70, no te desanimes: significa que necesitas otra vuelta al libro con más práctica de código, y eso es normal. Lo importante es que ahora sabes exactamente qué repasar.

Unidad 1 - Sintaxis y variables (preguntas 1 a 5): aquí se evalúa si escribes variables correctamente con el signo \$, si entiendes la interpolación en comillas dobles, las características del lenguaje (tipado dinámico, ejecución en servidor) y la conversión automática de tipos. Si fallaste aquí, repasa cómo se declaran las variables, la diferencia entre comillas simples y dobles, y qué hace PHP cuando mezcla números con cadenas numéricas. Son los cimientos: sin esto, todo lo demás se tambalea.

Unidad 2 - Operadores (preguntas 6 a 10): se prueban el módulo, el post-incremento contra el pre-incremento, la diferencia entre == y ===, la precedencia de && sobre || y el orden de las operaciones aritméticas. Si fallaste aquí, practica expresiones en un script de prueba e imprime los resultados: ver con tus ojos que \$x++ asigna antes de incrementar vale más que leerlo diez veces. Pon atención especial al ===, porque en código real evita errores difíciles de depurar.

Unidad 3 - Condicionales (preguntas 11 a 15): se evalúa la cascada if/elseif/else, el uso del break en switch, el operador ternario, la sintaxis obligatoria de paréntesis y el operador de fusión null ?. Si fallaste aquí, repasa el orden de las condiciones (la más restrictiva primero) y

práctica el ?? leyendo parámetros de URL, porque lo usaras en casi todos tus formularios. Recuerda que un switch sin break ejecuta los casos siguientes por accidente.

Unidad 4 - Bucles (preguntas 16 a 20): se prueban el conteo de iteraciones del for, el seguimiento paso a paso del while, la diferencia entre break y continue, la garantía de ejecución del do...while y la interrupción de un foreach. Si fallaste aquí, toma lápiz y papel y simula los bucles iteración por iteración: es la técnica más efectiva para dominarlos. Confundir break con continue es un error clásico que en programas reales causa bucles que no terminan o datos que se saltan.

Unidad 5 - Funciones (preguntas 21 a 25): se evalúa el uso de parámetros con valores por defecto, la diferencia entre echo y return, el alcance de las variables (scope), las funciones como argumentos y la recursión con su caso base. Si fallaste aquí, repasa sobre todo el alcance: es la causa número uno de "mi variable está vacía dentro de la función". Acostumbrate a que las funciones reciban lo que necesitan por parámetros y devuelvan resultados con return; así tu código será reutilizable y fácil de probar.

Unidad 6 - Arreglos (preguntas 26 a 30): se prueban los índices base cero, los arreglos asociativos, la función count(), array_push() y la navegación en arreglos multidimensionales. Si fallaste aquí, práctica creando arreglos de cada tipo y accediendo a sus elementos, porque los arreglos son la estructura de datos que más usaras en PHP: \$_POST, \$_GET y los resultados de la base de datos son todos arreglos. El error por uno (off-by-one) con los índices se corrige contando desde cero en voz alta hasta que se vuelva automatico.

Unidad 7 - Formularios y validación (preguntas 31 a 35): se evalúa la correspondencia entre method del formulario y el arreglo superglobal, las diferencias entre GET y POST, la validación con filter_var(), la regla de nunca confiar en la entrada del usuario y la protección contra XSS con htmlspecialchars(). Si fallaste aquí, repasa con un formulario real: envíalo por GET y por POST, valida un email falso y prueba escribir <script> en un campo para ver como htmlspecialchars() lo neutraliza. Esta unidad es la frontera entre un sitio que funciona y un sitio seguro.

Unidad 8 - Cookies, sesiones y login (preguntas 36 a 40): se prueban donde vive cada mecanismo (navegador contra servidor), la colocación de session_start() antes de cualquier salida, la sintaxis de \$_SESSION, el cierre correcto de sesión y la verificación de contraseñas con password_verify(). Si fallaste aquí, monta un login de juguete y juega con el: entra, recarga páginas, cierra sesión y verifica que ya no tengas acceso. Entender que la sesión vive en el servidor es lo que te hará guardar ahí los datos sensibles y no en cookies.

Unidad 9 - PDO, sentencias preparadas y CRUD (preguntas 41 a 45 y 47 a 49): se evalúa la conexión con manejo de errores, el INSERT, SELECT, UPDATE y DELETE con marcadores,

la lectura de resultados con `fetchAll()`, el JOIN entre tablas, el ordenamiento y la subida de archivos con validaciones. Si fallaste aquí, vuelve a escribir cada operación CRUD contra tu base de práctica hasta que te salga sin mirar. La sentencia preparada no es opcional ni "para después": es la forma correcta de hablar con la base de datos desde el día uno, y es lo que separa al código aficionado del código profesional.

Unidad 10 - Seguridad y despliegue (preguntas 46 y 50): se prueban el flujo completo de un login seguro y el checklist de despliegue con credenciales en variables de entorno. Si fallaste aquí, repasa por qué `password_hash()` y `password_verify()` van juntos, por qué el mensaje de error del login debe ser generico y por qué las claves nunca van escritas en el código. Desplegar es el último paso del curso y el primero de tu vida como desarrollador: hazlo bien desde tu primer proyecto.

Cierre: si este examen te costó, es buena señal: significa que te exigio de verdad. Vuelve a las unidades marcadas, escribe el código con tus manos (no solo lo leas) y repite el examen en una semana. La programación se aprende programando, y cada error que entendiste hoy es un error que no cometeras en tu primer proyecto real. Guarda tu calificación y la fecha de hoy: cuando repitas el examen y veas el número subir, sabras que el esfuerzo valio la pena. Nos vemos en el siguiente curso de lucasegaacademy.com.

Colibrí Editorial EGA

Curso Completo de PHP

Curso completo de PHP: variables, tipos, formularios, cookies, sesiones, PDO, CRUD, seguridad y proyecto final, con prácticas y examen final.

lucasegaacademy.com

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México