RA3. Escribe bloques de sentencias embebidos en lenguajes de marcas, seleccionando y utilizando las estructuras de programación.

Resultados de Aprendizaje y Conceptos básicos

3. Escribe bloques de sentencias embebidos en lenguajes de marcas, seleccionando y utilizando las estructuras de programación.

Criterios de evaluación:
a) Se han utilizado mecanismos de decisión en la creación de bloques de sentencias.
b) Se han utilizado bucles y se ha verificado su funcionamiento.
c) Se han utilizado “arrays” para almacenar y recuperar conjuntos de datos. (+ JSON)
d) Se han creado y utilizado funciones.
e) Se han utilizado formularios Web para interactuar con el usuario del navegador Web.
f) Se han empleado métodos para recuperar la información introducida en el formulario.
g) Se han añadido comentarios al código.

Conceptos básicos

Programación estructurada o secuencial

Sentencias: Tipos y Bloques.
Comentarios.
Tomas de decisión.
Bucles.
Tipos de datos compuestos: Arrays.

Programación orientada a objetos

Funciones: Parámetros.

Interacción con el usuario

Recuperación y utilización de información proveniente del cliente Web.
Procesamiento de la información introducida en un formulario: Métodos POST y GET.

Procesamiento de la Información Introducida en un Formulario: Métodos POST y GET

Cuando se trabaja en aplicaciones web, el procesamiento de la información introducida en un formulario se realiza mediante la comunicación entre el cliente (navegador) y el servidor, utilizando los métodos HTTP, principalmente **POST** y **GET**. Estos métodos son los más comunes para enviar datos desde el navegador web al servidor y cada uno tiene características y usos específicos. A continuación, se explica ampliamente cada uno de ellos:

Método GET

El método **GET** se utiliza para solicitar datos de un servidor web. En el contexto de formularios, los datos introducidos por el usuario en el formulario se envían como parte de la URL. Los parámetros del formulario se adjuntan a la URL como una cadena de consulta (query string), que sigue a un signo de interrogación ?.

Funcionamiento de GET:

1. Cuando un formulario se envía mediante el método GET, el navegador construye una URL que contiene todos los datos del formulario.
    - Ejemplo de URL con datos: `https://example.com/search?query=ordenadores&cattegory=tecnologia`
    - En este caso, `query` y `category` son los nombres de los campos del formulario, y los valores introducidos por el usuario son "ordenadores" y "tecnología".
2. El servidor recibe la solicitud con esta URL, extrae los datos de la cadena de consulta, y procesa la información.
3. **GET es idempotente**, lo que significa que hacer múltiples solicitudes GET al mismo recurso no debería cambiar el estado del servidor (es decir, no provoca efectos secundarios). Por eso se usa principalmente para recuperar información, no para modificarla.

Características de GET:

- **Visibilidad:** Los datos se envían como parte de la URL, lo que significa que son visibles para el usuario y se almacenan en el historial del navegador. Esto hace que GET sea inapropiado para el envío de información sensible como contraseñas.
- **Longitud limitada:** Los navegadores tienen límites en la longitud de las URLs (generalmente unos 2000 caracteres), por lo que GET no es adecuado para formularios con gran cantidad de datos.
- **Caché y marcadores:** Dado que las URLs con parámetros pueden ser almacenadas en caché y agregadas a marcadores, GET es útil para solicitudes que deben ser recordadas o compartidas.

Usos comunes de GET:

- Búsquedas en la web (como consultas de búsqueda en Google).
- Navegación por categorías o filtros en tiendas online.
- Recuperación de recursos (imágenes, documentos, etc.).

Método POST

El método **POST** se utiliza para enviar datos al servidor con el fin de procesarlos. A diferencia de GET, los datos no se adjuntan a la URL, sino que se envían en el cuerpo de la solicitud HTTP. Esto lo convierte en una opción más segura y adecuada para formularios que contienen grandes cantidades de datos o información sensible.

Funcionamiento de POST:

1. Cuando un formulario se envía mediante el método POST, el navegador envía los datos del formulario en el cuerpo de la solicitud HTTP.
    - Ejemplo de solicitud POST (simplificada):

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

username=usuario&password=contraseña
```

1. El servidor recibe los datos y los procesa de acuerdo con la lógica programada (por ejemplo, registrar un nuevo usuario o validar una sesión de inicio).
2. **POST no es idempotente**, lo que significa que enviar la misma solicitud POST varias veces puede provocar efectos secundarios (como crear múltiples cuentas o registros).

Características de POST:

- **Mayor seguridad:** Dado que los datos no aparecen en la URL, no son visibles en el historial del navegador ni en los marcadores. Esto lo hace más adecuado para enviar datos sensibles, aunque no reemplaza la necesidad de cifrar la comunicación con HTTPS.
- **Sin límite de tamaño:** Al enviar los datos en el cuerpo de la solicitud, POST permite manejar formularios con grandes volúmenes de datos, como archivos o información extensa.
- **No cacheable:** A diferencia de GET, las solicitudes POST no se almacenan en caché, lo que garantiza que siempre se procesen en el servidor.

Usos comunes de POST:

- Envío de formularios de registro o inicio de sesión.

- Procesamiento de pedidos de compra en tiendas en línea.
- Carga de archivos o imágenes.
- Envío de comentarios o mensajes en foros o redes sociales.

Diferencias Clave entre GET y POST:

Característica	GET	POST
Visibilidad de los datos	En la URL, visibles para el usuario.	En el cuerpo de la solicitud, no visibles.
Tamaño de los datos	Limitado (2000 caracteres aprox.)	Ilimitado (depende del servidor).
Idempotencia	Sí, no debería alterar el estado del servidor.	No, puede alterar el estado del servidor.
Caché y marcadores	Puede ser almacenado y marcado.	No es almacenado ni marcado.
Seguridad	Menos seguro para datos sensibles.	Más seguro, pero requiere HTTPS para confidencialidad.
Uso típico	Recuperar datos (búsquedas, navegación).	Enviar datos, como formularios o archivos.

Tipos de Llamadas GET y POST

Llamadas GET

Las llamadas GET pueden variar según la naturaleza de la solicitud y cómo se estructuran los parámetros de la URL:

1. **Simple GET Request:**
   - Se usa para solicitar recursos o datos estáticos desde el servidor.
   - Ejemplo: Solicitar una imagen o página web.
   - URL: `https://example.com/image.jpg`
2. **GET con Query String:**
   - Se utiliza para pasar parámetros al servidor, generalmente para filtrar o buscar información.
   - Ejemplo: Búsqueda de productos.
   - URL: `https://example.com/search?query=smartphone&category=electronics`
3. **GET con Paginación:**
   - En aplicaciones que requieren mostrar datos en múltiples páginas, como catálogos de productos.
   - Ejemplo: `https://example.com/products?page=2&size=20`

Llamadas POST

Las llamadas POST suelen diferenciarse por el tipo de datos que envían o cómo se estructuran:

1. **Formulario con Datos URL-Encoded:**
   - Es el tipo más común para el envío de formularios simples. Los datos se envían en un formato de clave-valor, codificados de forma similar a la query string de GET.
   - Ejemplo: Envío de un formulario de contacto.
2. **POST con Datos Multipart:**
   - Se utiliza para cargar archivos o enviar múltiples partes de un formulario, que pueden incluir datos binarios.
   - Ejemplo: Subida de imágenes o archivos.
3. **POST con JSON o XML:**
   - Se emplea principalmente en APIs RESTful, donde los datos estructurados (como JSON o XML) se envían en el cuerpo de la solicitud.
   - Ejemplo: Creación de un nuevo usuario en un sistema mediante una API:

```
POST /api/users HTTP/1.1
{
  "username": "johndoe",
  "email": "johndoe@example.com"
}
```

El método **GET** se usa para recuperar información, donde los datos del formulario son visibles en la URL, y es ideal para acciones no críticas como búsquedas o filtrado. Por otro lado, **POST** es adecuado para enviar información que no debe ser visible ni almacenada en caché, como datos confidenciales o grandes volúmenes de información, y es la opción correcta para acciones que alteran el estado del servidor, como la creación o modificación de registros.

No solo tenemos GET Y POST

En las aplicaciones web, además de los métodos **POST** y **GET**, existen otros métodos HTTP como **PUT**, **DELETE**, **PATCH**, entre otros. Cada uno de estos métodos tiene un propósito específico dentro de la comunicación entre cliente y servidor. Juntos, forman los llamados **verbos HTTP**, que son esenciales en las APIs RESTful para la interacción con recursos en el servidor. Aquí te explico en detalle los métodos más comunes:

Explicación de API RESTful y los Métodos HTTP:

En las aplicaciones web modernas, las **APIs RESTful** juegan un papel fundamental en la comunicación entre el **cliente** (navegador, aplicación móvil, etc.) y el **servidor**. Estas APIs permiten que diferentes sistemas se comuniquen entre sí de manera eficiente. En una API RESTful, la interacción con los datos y recursos en el servidor se realiza utilizando los llamados **verbos HTTP**, como **POST**, **GET**, **PUT**, **DELETE**, **PATCH**, entre otros.

¿Qué es una API RESTful?

- **API** significa "Interfaz de Programación de Aplicaciones" (Application Programming Interface), lo que permite que diferentes aplicaciones se comuniquen.
- **REST** significa "Transferencia de Estado Representacional" (Representational State Transfer), que es un estilo arquitectónico para diseñar servicios web. En una API RESTful, se siguen principios que permiten una interacción simple y escalable con los recursos del servidor.

Principios Básicos de REST:

1. **Recursos**: Todo en una API RESTful se trata como un recurso. Un recurso puede ser cualquier entidad, como usuarios, productos, pedidos, etc. Estos recursos se identifican mediante una **URL** única.
2. **Operaciones HTTP**: Para interactuar con los recursos, se utilizan los verbos HTTP (GET, POST, PUT, DELETE, etc.), donde cada verbo tiene un propósito específico.
3. **Stateless**: Las solicitudes RESTful son **sin estado**, lo que significa que cada solicitud del cliente al servidor es independiente y no mantiene información entre solicitudes. El cliente debe enviar toda la información necesaria en cada solicitud.
4. **Respuesta en formato JSON**: La mayoría de las APIs RESTful devuelven datos en formato **JSON, CASI NUNCA EN XML**, que es ligero y fácil de interpretar por diferentes lenguajes de programación.

Idempotencia

El concepto de **idempotencia** en programación, y más específicamente en las **APIs RESTful** y los **verbos HTTP**, se refiere a la propiedad de algunas operaciones que pueden ser repetidas muchas veces sin cambiar el resultado después de la primera ejecución.

Idempotencia: Definición

- **Una operación es idempotente** si, al repetirla varias veces, produce el mismo resultado que si se hubiera hecho solo una vez.
- Es decir, realizar la operación una o más veces no tiene efectos adicionales en el estado del sistema o en los datos.

Ejemplo sencillo:

Imagina que estás actualizando la dirección de un usuario en una base de datos.

- Si haces una solicitud **PUT** para cambiar la dirección del usuario a "Calle Ejemplo 123", y la haces 10 veces seguidas, la dirección sigue siendo "Calle Ejemplo 123" después de cada solicitud. No importa cuántas veces repitas la operación, el resultado es el mismo.
    - **Conclusión**: **PUT** es idempotente porque no genera cambios adicionales si se ejecuta varias veces con los mismos datos.

1. GET

El método **GET** se usa para **recuperar** datos de un servidor. Como mencionamos anteriormente, GET no altera el estado del servidor y es idempotente, lo que significa que puedes hacer la misma solicitud GET varias veces sin que cambie el resultado ni tenga efectos secundarios en el servidor.

Características:

- **Idempotente:** Sí.
- **Uso:** Recuperar información de un recurso.
- **Seguridad:** Los datos son visibles en la URL, no se recomienda para datos sensibles.

```
GET /api/products HTTP/1.1
Host: example.com
```

2. POST

El método **POST** se utiliza para **enviar** datos al servidor y **crear** nuevos recursos. No es idempotente, lo que significa que repetir una solicitud POST puede generar múltiples registros o resultados diferentes en el servidor. Es adecuado para formularios de registro, envío de datos, o cualquier operación que modifique el estado del servidor.

Características:

- **Idempotente:** No.
- **Uso:** Crear nuevos recursos o enviar datos para su procesamiento.
- **Seguridad:** Los datos van en el cuerpo de la solicitud, pero debe usarse con HTTPS para evitar su intercepción.

Ejemplo:

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "username": "johndoe",
  "password": "securepassword"
}
```

3. PUT

El método **PUT** se utiliza para **actualizar** un recurso existente o **reemplazarlo** completamente. PUT es idempotente, lo que significa que puedes hacer la misma solicitud PUT varias veces y el resultado será el mismo, siempre y cuando los datos sean los mismos. Si el recurso no existe, puede crear uno nuevo, pero esta funcionalidad depende de la implementación del servidor.

Características:

- **Idempotente:** Sí.
- **Uso:** Actualizar o reemplazar completamente un recurso existente.
- **Seguridad:** Los datos van en el cuerpo de la solicitud, como en POST.

Ejemplo:

```
PUT /api/users/123 HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "username": "johndoe",
  "email": "johndoe@example.com"
}
```

En este caso, el usuario con ID 123 será actualizado con los nuevos datos. Si el usuario no existiera, algunos sistemas pueden optar por crearlo.

Diferencia entre PUT y POST explicada con ejemplos:

Ejemplo de PUT (Operación Idempotente):

Imagina que tienes un perfil de usuario en una plataforma y quieres actualizar la dirección de tu cuenta.

Solicitud PUT:

Cuando envías una solicitud **PUT**, la idea es **actualizar** completamente un recurso. En este caso, el recurso es la información del perfil del usuario. Si ya existe un usuario con ID 123, la solicitud **PUT** lo **actualiza**. Si no existiera, en algunos casos el servidor podría crearlo.

Lo que hace el servidor:

1. Si el usuario con ID 123 ya existe, **toda la información se reemplaza**. Si había otro nombre o dirección antes, ahora serán exactamente los valores enviados en la solicitud.
2. **Idempotencia**: Si haces esta misma solicitud PUT 100 veces, el resultado será siempre el mismo: El nombre será "Juan Pérez" y la dirección será "Calle Ejemplo 123" después de cada solicitud.

Ejemplo práctico para entender PUT:

Piensa en **PUT** como si estuvieras **reemplazando una foto de perfil**. Si subes la misma foto varias veces, el resultado es el mismo: tu foto de perfil se mantiene igual. No importa cuántas veces subas la misma imagen, el resultado final siempre será la misma foto.

Ejemplo de POST (No Idempotente):

Ahora, imagina que estás creando una nueva cuenta en la misma plataforma.

Solicitud POST:

Cuando envías una solicitud **POST**, la idea es **crear** un recurso nuevo. Cada vez que haces una solicitud POST, se crea un **nuevo** recurso en el servidor.

Lo que hace el servidor:

1. El servidor crea un **nuevo usuario** con los datos enviados. Genera un nuevo ID para el usuario (por ejemplo, `ID = 456`).

2. **No idempotente**: Si haces esta misma solicitud POST 100 veces, el servidor creará 100 usuarios nuevos, cada uno con un ID diferente.

Ejemplo práctico para entender POST:

Piensa en **POST** como si estuvieras **subiendo fotos nuevas a un álbum**. Si subes la misma foto varias veces, acabarás con muchas copias de la misma imagen. Cada solicitud POST genera algo nuevo, por lo que no es idempotente.

4. DELETE

El método **DELETE** se utiliza para **eliminar** un recurso existente en el servidor. Es idempotente, ya que realizar la misma solicitud varias veces no tendrá efectos adicionales. Si el recurso ya ha sido eliminado, una nueva solicitud DELETE no cambiará nada.

Características:

- **Idempotente:** Sí.
- **Uso:** Eliminar un recurso.
- **Seguridad:** Dado que elimina datos, se debe usar con cuidado, y preferiblemente con mecanismos de autenticación y autorización adecuados.

Ejemplo:

```
DELETE /api/users/123 HTTP/1.1
Host: example.com
```

Esto eliminaría al usuario con ID 123 del sistema.

5. PATCH

El método **PATCH** es similar a **PUT**, pero la diferencia principal es que PATCH se utiliza para realizar **actualizaciones parciales** a un recurso existente, en lugar de reemplazar el recurso por completo. Con PATCH, solo se actualizan los campos proporcionados en la solicitud.

Características:

- **Idempotente:** Depende de la implementación, pero en general, sí.
- **Uso:** Actualizar parcialmente un recurso existente.

- **Seguridad:** Los datos van en el cuerpo de la solicitud, como en PUT y POST.

Ejemplo:

```
PATCH /api/users/123 HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "email": "newemail@example.com"
}
```

En este caso, solo se actualizará el correo electrónico del usuario con ID 123, sin afectar otros datos como su nombre o contraseña.

6. HEAD

El método **HEAD** es similar a **GET**, pero en lugar de recuperar todo el contenido de un recurso, solo recupera los **encabezados** HTTP. Es útil para verificar si un recurso existe o para comprobar metadatos (como el tamaño o la fecha de modificación) sin descargar el contenido completo.

Características:

- **Idempotente:** Sí.
- **Uso:** Obtener encabezados HTTP sin el cuerpo de la respuesta.
- **Seguridad:** Similar a GET, pero más ligero en términos de carga.

Ejemplo:

```
HEAD /api/products/123 HTTP/1.1
Host: example.com
```

Esto devolverá los encabezados del producto con ID 123 sin incluir el contenido del producto.

7. OPTIONS

El método **OPTIONS** se usa para **preguntar** al servidor qué métodos HTTP son **soportados** o permitidos en un recurso en particular. Es útil para saber qué operaciones puedes realizar en un recurso, y también se usa comúnmente en solicitudes **CORS** (Cross-Origin Resource Sharing) para indicar qué dominios pueden interactuar con una API.

Características:

- **Idempotente:** Sí.
- **Uso:** Obtener los métodos permitidos para un recurso o dominio.

Ejemplo:

```
OPTIONS /api/products HTTP/1.1
Host: example.com
```

La respuesta del servidor incluirá una lista de los métodos HTTP permitidos, como GET, POST, PUT, DELETE, etc.

8. TRACE

El método **TRACE** se utiliza para hacer un **eco** de la solicitud que ha llegado al servidor. Esto permite a los desarrolladores ver cómo se está transmitiendo la solicitud desde el cliente al servidor y si ha sido modificada en el camino. Sin embargo, por motivos de seguridad, este método rara vez se usa y se desactiva en muchos servidores.

Características:

- **Idempotente:** Sí.
- **Uso:** Diagnosticar problemas con las solicitudes HTTP.

Ejemplo:

```
TRACE /api/products HTTP/1.1
Host: example.com
```

Esto devolverá un eco de la solicitud tal como llegó al servidor.

9. CONNECT

El método **CONNECT** se utiliza para establecer un **túnel** hacia un servidor a través de un proxy, y es comúnmente utilizado para habilitar conexiones seguras **SSL** o **TLS** a través de un proxy HTTP.

Características:

- **Idempotente:** No.
- **Uso:** Crear una conexión segura a través de un proxy.
- **Seguridad:** Generalmente utilizado en conexiones HTTPS.

Ejemplo:

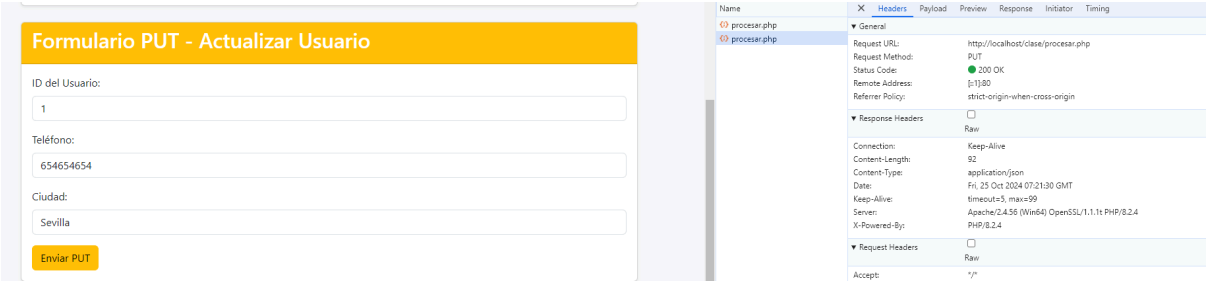
```
CONNECT www.example.com:443 HTTP/1.1
Host: www.example.com
```

Resumen de los Métodos HTTP:

Método	Uso	Idempotente	Uso común
GET	Recuperar recursos	Sí	Obtener datos, páginas web, imágenes
POST	Enviar datos y crear nuevos recursos	No	Enviar formularios, crear registros
PUT	Reemplazar o actualizar un recurso	Sí	Actualizar o crear recursos
DELETE	Eliminar recursos	Sí	Eliminar registros
PATCH	Actualizar parcialmente un recurso	Depende	Actualizaciones parciales
HEAD	Obtener los encabezados HTTP	Sí	Verificar la existencia o metadatos del recurso
OPTIONS	Consultar métodos permitidos para un recurso	Sí	Verificar los métodos soportados
TRACE	Hacer eco de la solicitud HTTP para diagnóstico	Sí	Diagnóstico de problemas HTTP
CONNECT	Establecer un túnel para una conexión segura	No	Conexiones HTTPS a través de proxies

Cada método HTTP tiene su función dentro de una API o aplicación web. Los más utilizados son **GET** y **POST**, aunque **PUT**, **DELETE** y **PATCH** son cruciales en sistemas RESTful para la manipulación de recursos. Otros métodos, como **HEAD** y **OPTIONS**, juegan roles importantes en la optimización y seguridad de las aplicaciones.

VER LOS EJEMPLOS DE LLAMADAS DESDE CLIENTE A PHP. Se desarrollan a continuación.



Creación del archivo HTML:

- Desarrolla un archivo HTML que incluya un formulario y botones para enviar datos utilizando las solicitudes GET, POST, PUT y DELETE.
- Utiliza el siguiente esquema de diseño como base para la interfaz:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Práctica de Solicitudes HTTP</title>
    <script src="script.js"></script>
</head>
<body>
    <h1>Práctica: Métodos HTTP</h1>

    <!-- Formulario GET -->
    <h2>Solicitud GET</h2>
    <form id="form-get" action="procesar.php" method="get">
        <label for="nombre-get">Nombre (GET):</label>
        <input type="text" id="nombre-get" name="nombre">
        <button type="submit">Enviar GET</button>
    </form>

    <!-- Formulario POST -->
```

```
    <h2>Solicitud POST</h2>
    <form id="form-post" action="procesar.php" method="post">
        <label for="email-post">Email (POST):</label>
        <input type="email" id="email-post" name="email">
        <button type="submit">Enviar POST</button>
    </form>

    <!-- Botón PUT -->
    <h2>Solicitud PUT</h2>
    <button id="put-button">Enviar PUT</button>

    <!-- Botón DELETE -->
    <h2>Solicitud DELETE</h2>
    <button id="delete-button">Enviar DELETE</button>
</body>
</html>
```

Script JavaScript:

- Crea un archivo `script.js` que gestione las solicitudes PUT y DELETE mediante `fetch` y envíe los datos de manera correcta al servidor.

```
document.addEventListener('DOMContentLoaded', function () {
    const putButton = document.getElementById('put-button');
    const deleteButton = document.getElementById('delete-button');

    // Ejemplo PUT
    putButton.addEventListener('click', function () {
        fetch('procesar.php', {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ nombre: 'Nuevo Nombre' }),
        })
        .then(response => response.text())
        .then(data => alert('Respuesta PUT: ' + data))
        .catch(error => console.error('Error:', error));
    });

    // Ejemplo DELETE
    deleteButton.addEventListener('click', function () {
        fetch('procesar.php', {
```

```
        method: 'DELETE'
      })
      .then(response => response.text())
      .then(data => alert('Respuesta DELETE: ' + data))
      .catch(error => console.error('Error:', error));
  });
});
```

Nuevos conceptos:

1. **`document.addEventListener('DOMContentLoaded', function() { ... });`:**

   - **¿Qué es?**: Esta línea espera a que toda la página HTML haya sido completamente cargada antes de ejecutar el código dentro de la función. Esto es importante porque queremos asegurarnos de que todos los elementos, como los botones, estén disponibles antes de intentar hacer algo con ellos.
   - **Por qué es útil**: Si intentamos acceder a un botón antes de que el navegador haya terminado de cargar la página, podríamos obtener errores porque ese botón aún no existe.

2. **`const putButton = document.getElementById('put-button');`:**

   - **¿Qué es?**: Aquí estamos utilizando la función `getElementById` para seleccionar un botón específico que tiene el atributo `id="put-button"` en el archivo HTML. En este caso, es el botón que usaremos para hacer la solicitud **PUT**.
   - **¿Qué hace?**: Básicamente, le estamos diciendo al navegador: "Encuentra en el documento (la página HTML) un elemento que tenga el `id put-button` y guárdalo en la variable `putButton`".

3. **`const deleteButton = document.getElementById('delete-button');`:**

   - **¿Qué es?**: Similar al paso anterior, aquí seleccionamos el botón con el `id="delete-button"` en el archivo HTML. Este es el botón que usaremos para la solicitud **DELETE**.

4. **`putButton.addEventListener('click', function() { ... });`:**

   - **¿Qué es?**: Ahora estamos diciendo que queremos escuchar un "evento" en el botón `putButton`. El evento que estamos esperando es el clic del ratón. Cuando el usuario hace clic en el botón, ejecutamos el código que está dentro de la función.
   - **¿Qué hace?**: Cuando se hace clic en el botón de **PUT**, se envía una solicitud **PUT**

al servidor.

5. `fetch('procesar.php', { method: 'PUT', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ nombre: 'Nuevo Nombre' }) }):`

- **¿Qué es?**: La función `fetch` se utiliza para enviar solicitudes al servidor. En este caso, estamos enviando una solicitud **PUT** a un archivo en el servidor llamado `procesar.php`.
  - `method: 'PUT'`: Aquí indicamos que estamos utilizando el método **PUT**.
  - `headers: { 'Content-Type': 'application/json' }`: Esto indica que estamos enviando los datos en formato JSON.
  - `body: JSON.stringify({ nombre: 'Nuevo Nombre' })`: Los datos que estamos enviando son `{ nombre: 'Nuevo Nombre' }`, que significa que queremos actualizar algo con el nombre "Nuevo Nombre".
- **¿Qué hace?**: Envía una solicitud al servidor y le dice que queremos actualizar un recurso (por ejemplo, un nombre) con el método **PUT**.

6. `then(response => response.text()):`

- **¿Qué es?**: Esta parte del código se ejecuta cuando el servidor ha respondido a nuestra solicitud. El servidor nos devuelve una respuesta, y `then()` se utiliza para manejar lo que haremos con esa respuesta.
- **¿Qué hace?**: Convierte la respuesta que viene del servidor en texto para que podamos mostrarla o usarla de alguna manera.

7. `.then(data => alert('Respuesta PUT: ' + data)):`

- **¿Qué es?**: Después de convertir la respuesta en texto, aquí mostramos la respuesta en una ventana emergente (un "alert").
- **¿Qué hace?**: Por ejemplo, si el servidor responde con "Datos actualizados por PUT: Nuevo Nombre", el usuario verá un mensaje que dice: "Respuesta PUT: Datos actualizados por PUT: Nuevo Nombre".

8. `deleteButton.addEventListener('click', function() { ... }):`

- **¿Qué es?**: Similar al botón **PUT**, este es el código que se ejecuta cuando se hace clic en el botón de **DELETE**.
- **¿Qué hace?**: Al hacer clic en este botón, enviamos una solicitud **DELETE** al servidor para eliminar un recurso.

9. `fetch('procesar.php', { method: 'DELETE' }):`

- **¿Qué es?**: Aquí estamos utilizando la función `fetch` nuevamente, pero esta vez el método es **DELETE**.
- **¿Qué hace?**: Esto le indica al servidor que queremos eliminar algo, pero no enviamos datos adicionales, ya que solo necesitamos decirle al servidor que

elimine un recurso.

10. `.catch(error => console.error('Error:', error));`

- **¿Qué es?**: Esta parte del código se utiliza para manejar errores. Si algo sale mal durante la solicitud (por ejemplo, si no se puede contactar con el servidor), el error se mostrará en la consola del navegador.
- **¿Qué hace?**: Evita que el programa se detenga si ocurre un error y nos permite ver cuál fue el problema.

Backend en PHP:

- Crea un archivo `procesar.php` que gestione las solicitudes GET, POST, PUT y DELETE y envíe respuestas apropiadas.

```
<?php
// Procesar GET
if ($_SERVER['REQUEST_METHOD'] === 'GET') {
    $nombre = $_GET['nombre'];
    echo "Datos recibidos por GET: " . htmlspecialchars($nombre);
}

// Procesar POST
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email = $_POST['email'];
    echo "Datos recibidos por POST: " . htmlspecialchars($email);
}

// Procesar PUT
if ($_SERVER['REQUEST_METHOD'] === 'PUT') {
    $data = json_decode(file_get_contents('php://input'), true);
    echo 'Datos actualizados por PUT: ' . htmlspecialchars($data['nombre']);
}

// Procesar DELETE
if ($_SERVER['REQUEST_METHOD'] === 'DELETE') {
    echo 'Recurso eliminado con DELETE';
}
?>
```

Conceptos nuevos:

1. `if ($_SERVER['REQUEST_METHOD'] === 'PUT'):`

- **¿Qué es?**: Esta línea verifica el tipo de solicitud HTTP que ha recibido el servidor. En este caso, estamos comprobando si la solicitud que llegó es de tipo **PUT**.
- **¿Por qué es importante?**: En una aplicación web, pueden llegar diferentes tipos de solicitudes (GET, POST, PUT, DELETE, etc.). Este condicional asegura que el código que sigue solo se ejecutará si el método es **PUT**, que generalmente se usa para **actualizar** datos en el servidor.

2. `$data = json_decode(file_get_contents('php://input'), true);:`

- **¿Qué es?**:
  - `file_get_contents('php://input')`: Esta función de PHP obtiene los datos crudos que fueron enviados al servidor en la solicitud PUT. En las solicitudes PUT, los datos no se envían en las variables $_GET o $_POST, sino que están en el cuerpo de la solicitud.
  - `json_decode()`: Como los datos que enviamos desde JavaScript estaban en formato JSON, esta función los **decodifica** para que PHP los convierta en un formato que pueda usar, como un array asociativo.
  - `true`: Este parámetro indica que queremos que el resultado de `json_decode` sea un **array asociativo** en lugar de un objeto.
- **¿Qué hace?**: Básicamente, esta línea toma los datos enviados en formato JSON (como `{ nombre: 'Nuevo Nombre' }`), los convierte a un array de PHP, y los almacena en la variable `$data`. Después, podemos acceder a cada valor dentro de `$data`, como `$data['nombre']`.

3. `echo 'Datos actualizados por PUT: ' . htmlspecialchars($data['nombre']);:`

- **¿Qué es?**:
  - **`echo`**: Muestra un mensaje al usuario, en este caso, el resultado de la operación.
  - **`htmlspecialchars($data['nombre'])`**: Esta función convierte caracteres especiales en entidades HTML. Esto es útil para prevenir posibles vulnerabilidades como XSS (Cross-Site Scripting). Si el nombre enviado por el cliente contiene caracteres especiales, como `<` o `>`, se convierten en entidades seguras como `<` o `>`.
- **¿Qué hace?**: Muestra un mensaje que incluye el nombre actualizado por la solicitud PUT. Por ejemplo, si la solicitud incluía `{ nombre: 'Nuevo Nombre' }`, el servidor responderá con el mensaje:
  `Datos actualizados por PUT: Nuevo Nombre.`

4. `if ($_SERVER['REQUEST_METHOD'] === 'DELETE'):`

   - **¿Qué es?**: Similar al caso de PUT, esta línea verifica si la solicitud HTTP que llega al servidor es de tipo **DELETE**.
   - **¿Por qué es importante?**: En este caso, estamos controlando si la solicitud es **DELETE**, que generalmente se usa para eliminar recursos.

5. `echo 'Recurso eliminado con DELETE';:`

   - **¿Qué es?**: Muestra un mensaje que confirma que se ha eliminado algo en el servidor.
   - **¿Qué hace?**: Al procesar una solicitud **DELETE**, el servidor responde al cliente con este mensaje. Aunque este es un ejemplo sencillo que solo imprime un mensaje, en un caso real podríamos eliminar un registro de una base de datos o realizar otras acciones relacionadas con la eliminación de recursos.

Ver ejemplo: POST_GET_PUT_Delete.zip

Uso de Formularios en Aplicaciones Web: Tipos de Envío y Procesamiento en el Servidor

Los formularios son una de las herramientas principales para interactuar con los usuarios en aplicaciones web, permitiendo el envío de datos al servidor para su procesamiento. Existen distintos métodos para enviar estos datos, cada uno adecuado para diferentes tipos de información y necesidades.

Uso de Formularios en Aplicaciones Web

Los formularios son elementos fundamentales en las aplicaciones web, ya que permiten a los usuarios interactuar con el sitio enviando datos al servidor para su procesamiento. Los formularios son utilizados para múltiples propósitos, como la autenticación de usuarios, la recolección de datos, la subida de archivos, entre otros. A continuación, se detalla la estructura básica de un formulario, las partes principales y los tipos de campos más comunes.

1. Estructura de un Formulario HTML

Un formulario en HTML se define mediante la etiqueta `<form>`, que contiene varios atributos que configuran su comportamiento y una serie de campos para la entrada de datos por parte del usuario. La estructura general de un formulario es la siguiente:

```
<form action="URL_destino" method="GET o POST" enctype="tipo_de_codificación">
   <!-- Campos de entrada de datos -->
</form>
```

1.1. Atributos Principales de la Etiqueta `<form>`

- **`action`**:
  - Especifica la URL a la que se enviarán los datos del formulario cuando se realice el envío. Esta URL puede ser una página PHP, un script en el servidor o una API que procese los datos.
  - Por ejemplo, `action="procesar.php"` indica que los datos serán enviados al archivo `procesar.php` para su procesamiento.
- **`method`**:
  - Indica el método HTTP que se usará para enviar los datos. Los métodos más comunes son:
    - **GET**: Los datos del formulario se envían como parte de la URL (en la cadena de consulta). Es útil para formularios de búsqueda, pero no es recomendable para enviar datos sensibles.
    - **POST**: Los datos se envían en el cuerpo de la solicitud HTTP, lo que permite enviar información más grande y sensible.
- **`enctype`** (solo cuando `method="POST"`):
  - Especifica cómo se deben codificar los datos del formulario antes de enviarse al servidor.
  - **Valores comunes**:
    - `application/x-www-form-urlencoded` (por defecto): Los datos se codifican como pares clave-valor, lo que es adecuado para la mayoría de los formularios.
    - `multipart/form-data`: Se usa para enviar archivos o datos binarios.
    - `text/plain`: Envía los datos en texto plano.

2. Principales Tipos de Campos en un Formulario

2.1. Etiqueta `<input>`

La etiqueta `<input>` se utiliza para crear campos de entrada de datos. Tiene varios atributos que determinan su tipo y comportamiento.

- **Atributo `type`**: Define el tipo de campo de entrada. Algunos de los tipos más comunes son:
    - **`type="text"`**:
        - Crea un campo de entrada de texto de una sola línea.
        - Ejemplo:

```
<input type="text" name="nombre" placeholder="Introduce tu nombre">
```

`type="password"`:

- Crea un campo de entrada para contraseñas, ocultando los caracteres.
- Ejemplo:

```
<input type="password" name="clave" placeholder="Introduce tu contraseña">
```

`type="email"`:

- Crea un campo de entrada para direcciones de correo electrónico. Los navegadores modernos validan automáticamente el formato del correo.
- Ejemplo:

```
<input type="email" name="correo" placeholder="correo@ejemplo.com">
```

`type="radio"`:

- Crea botones de opción (radio buttons), que permiten seleccionar una única opción de un conjunto. Todos los botones del mismo grupo deben tener el mismo atributo `name`.
- Ejemplo:

```
<input type="radio" name="genero" value="hombre"> Hombre
<input type="radio" name="genero" value="mujer"> Mujer
```

`type="checkbox"`:

- Crea una casilla de verificación (checkbox) que permite seleccionar múltiples opciones.
- Ejemplo:

```
<input type="checkbox" name="intereses" value="musica"> Música
<input type="checkbox" name="intereses" value="deportes"> Deportes
```

`type="number"`:

- Crea un campo de entrada para números, con controles incrementales.
- Ejemplo

```
<input type="number" name="edad" min="1" max="100">
```

`type="file"`:

- Permite al usuario seleccionar un archivo para subir.
- Ejemplo:

```
<input type="file" name="archivo">
```

`type="submit"`:

- Crea un botón para enviar el formulario.
- Ejemplo:

```
<input type="submit" value="Enviar">
```

2.2. Etiqueta `<label>`

La etiqueta `<label>` proporciona una descripción para un campo de entrada y mejora la accesibilidad del formulario. Puede estar asociada a un campo utilizando el atributo `for`, que debe coincidir con el `id` del campo correspondiente.

Ejemplo:

```
<label for="nombre">Nombre:</label>
<input type="text" id="nombre" name="nombre">
```

2.3. Etiqueta `<textarea>`

La etiqueta `<textarea>` permite la entrada de texto en varias líneas, a diferencia de `<input type="text">`.

Ejemplo:

```
<textarea name="comentarios" rows="4" cols="50" placeholder="Escribe tus comentarios aquí"></textarea>
```

2.4. Etiqueta `<select>`

La etiqueta `<select>` crea un menú desplegable (dropdown) con opciones. Cada opción se define con la etiqueta `<option>`.

Ejemplo:

```
<select name="pais">
    <option value="es">España</option>
    <option value="mx">México</option>
    <option value="us">Estados Unidos</option>
</select>
```

3. Buenas Prácticas para el Uso de Formularios

1. **Usar `label` siempre que sea posible** para mejorar la accesibilidad.
2. **Validar los datos tanto en el lado del cliente como en el servidor**. La validación del cliente mejora la experiencia de usuario, pero no debe ser la única.
3. **Utilizar `method="POST"` para enviar datos sensibles o largos**. El método GET tiene limitaciones de tamaño y muestra los datos en la URL.
4. **Configurar el atributo `autocomplete`** cuando sea necesario. Ayuda a los usuarios a rellenar el formulario más rápidamente.

Ver ejemplos de formularios html-post-php: formulario_ejemplo.zip

1. Acceso a los Elementos del Formulario desde JavaScript

Para acceder y manipular los valores de los elementos de un formulario en JavaScript, podemos usar varias técnicas, dependiendo del tipo de elemento (como campos de texto, radio buttons, checkboxes, etc.). La manera más común de acceder a los elementos de un formulario es utilizando el método **`document.getElementById`**, **`document.querySelector`**, o **`document.forms`**.

1.1. Campos de Entrada de Texto (`<input type="text">`)

- **Acceso y obtención de valores**:

```
const nombre = document.getElementById('nombre').value; // Obtener el valor del campo de texto
```

Establecer un valor:

```
document.getElementById('nombre').value = 'Juan Pérez'; // Establecer un nuevo valor
```

1.2. Campo de Contraseña (`<input type="password">`)

Acceso y obtención de valores:

```
const password = document.getElementById('password').value;
```

1.3. Campo de Correo Electrónico (`<input type="email">`)

- **Acceso y validación de valores**:

```
const email = document.getElementById('email').value;
if (!email.includes('@')) {
    alert('Por favor, ingresa un correo electrónico válido.');
}
```

1.4. Radio Buttons (`<input type="radio">`)

- **Obtener el valor del radio button seleccionado**:

```
const generoSeleccionado = document.querySelector('input[name="genero"]:checked').value;
```

1.5. Checkboxes (`<input type="checkbox">`)

- **Verificar si una casilla está seleccionada**:

```
const musicaCheckbox = document.getElementById('musica').checked; // Devuelve true o false
```

Obtener todos los valores seleccionados:

```
const interesesSeleccionados = [];
document.querySelectorAll('input[name="intereses"]:checked').forEach((checkbox) => {
    interesesSeleccionados.push(checkbox.value);
});
```

1.6. Área de Texto (`<textarea>`)

- **Acceso y manipulación del valor del área de texto**:

```
const mensaje = document.getElementById('mensaje').value; // Obtener el texto del textarea
```

1.7. Menú Desplegable (`<select>`)

- **Obtener el valor seleccionado**:

```
const paisSeleccionado = document.getElementById('pais').value;
```

Obtener el texto del elemento seleccionado:

```
const indiceSeleccionado = document.getElementById('pais').selectedIndex;
const textoSeleccionado = document.getElementById('pais').options[indiceSeleccionado].text;
```

1.8. Archivos (`<input type="file">`)

- **Obtener el archivo seleccionado**:

```
const archivo = document.getElementById('archivo').files[0];
console.log('Nombre del archivo:', archivo.name);
```

2. Acceso a los Elementos del Formulario desde PHP

En PHP, los datos enviados desde un formulario se pueden acceder utilizando los arrays superglobales `$_POST`, `$_GET` y `$_FILES`, dependiendo del método de envío del formulario.

2.1. Uso del Array $_POST

Si el formulario se envía con el método **POST**, los datos estarán disponibles en el array `$_POST`.

- **Acceso a un campo de texto**:

```
$nombre = $_POST['nombre'];
echo "Nombre: " . htmlspecialchars($nombre);
```

Acceso a un radio button:

```
$genero = $_POST['genero'];
echo "Género seleccionado: " . htmlspecialchars($genero);
```

Acceso a un checkbox:

```
if (isset($_POST['intereses'])) {
    $intereses = $_POST['intereses'];
    foreach ($intereses as $interes) {
        echo "Interés: " . htmlspecialchars($interes) . "<br>";
    }
} else {
    echo "No se seleccionaron intereses.";
}
```

Acceso a un menú desplegable (`<select>`):

```
$pais = $_POST['pais'];
echo "País seleccionado: " . htmlspecialchars($pais);
```

2.2. Uso del Array `$_FILES` para Subida de Archivos

Cuando se usa un formulario con `enctype="multipart/form-data"`, los archivos se envían a través del array `$_FILES`.

- **Acceso a la información del archivo**:

```
$archivo = $_FILES['archivo'];
$nombreArchivo = basename($archivo['name']);
$rutaTemporal = $archivo['tmp_name'];
```

Mover el archivo a un directorio específico:

```
if (move_uploaded_file($rutaTemporal, 'uploads/' . $nombreArchivo)) {
    echo "Archivo subido con éxito.";
} else {
    echo "Error al subir el archivo.";
}
```

Los 3 tipos de formularios

A continuación, se presentan tres formas comunes de enviar datos desde un formulario HTML a un archivo PHP en el servidor, detallando sus características, casos de uso y ejemplos prácticos.

1. Formulario con Datos URL-Encoded

1.1. Concepto

El envío de formularios con datos **URL-Encoded** es el método más común para el envío de datos simples al servidor. En este formato, los datos son codificados en pares de clave-valor, utilizando el formato `clave=valor`, donde cada par es separado por el carácter &. Este formato es similar a cómo se manejan las cadenas de consulta en las solicitudes **GET**, pero en este caso, los datos se envían en el cuerpo de la solicitud HTTP, lo que es típico en solicitudes **POST**.

La codificación **URL-Encoded** convierte ciertos caracteres especiales en códigos seguros para transmitir a través de la URL (por ejemplo, el espacio se convierte en `%20`). Este método es adecuado para formularios simples donde se envían textos y números, como formularios de contacto o registro.

1.2. Ejemplo de Implementación: Envío de un Formulario de Contacto

HTML - Formulario de Contacto

El siguiente formulario permite al usuario enviar su nombre, correo electrónico y un mensaje. Los datos serán enviados al servidor mediante el método **POST** en el formato URL-Encoded.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Formulario de Contacto</title>
</head>
<body>
    <h1>Contacto</h1>
    <form action="procesar.php" method="post">
        <div>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre" required>
        </div>
```

```
        <div>
            <label for="email">Correo Electrónico:</label>
            <input type="email" id="email" name="email" required>
        </div>
        <div>
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" required></textarea>
        </div>
        <button type="submit">Enviar</button>
    </form>
</body>
</html>
```

PHP - Procesamiento del Formulario (procesar.php)

El archivo `procesar.php` recibe los datos enviados por el formulario y los procesa. Los valores son accedidos a través de la variable global `$_POST`, que contiene los datos enviados en el formato URL-Encoded.

```
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $nombre = $_POST['nombre'];
    $email = $_POST['email'];
    $mensaje = $_POST['mensaje'];

    // Validación básica de campos
    if (!empty($nombre) && !empty($email) && !empty($mensaje)) {
        echo "Formulario enviado con éxito.<br>";
        echo "Nombre: " . htmlspecialchars($nombre) . "<br>";
        echo "Correo Electrónico: " . htmlspecialchars($email) . "<br>";
        echo "Mensaje: " . nl2br(htmlspecialchars($mensaje)) . "<br>";
    } else {
        echo "Error: Todos los campos son obligatorios.";
    }
}
?>
```

2. POST con Datos Multipart

2.1. Concepto

El método **POST con datos Multipart** se utiliza para el envío de formularios que contienen archivos o datos binarios. En este formato, el contenido del formulario se divide en varias partes, cada una con un encabezado que indica el tipo de dato y el nombre del campo. Esto permite enviar tanto datos de texto como archivos adjuntos.

Para usar este formato, es necesario especificar el atributo `enctype` en el formulario como `multipart/form-data`. Es el formato más adecuado para la carga de archivos, como imágenes o documentos.

2.2. Ejemplo de Implementación: Subida de Archivos

HTML - Formulario para Subir una Imagen

Este formulario permite al usuario seleccionar una imagen para subir al servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Subida de Imagen</title>
</head>
<body>
    <h1>Subir Imagen</h1>
    <form action="procesar.php" method="post" enctype="multipart/form-data">
        <div>
            <label for="archivo">Selecciona una imagen:</label>
            <input type="file" id="archivo" name="archivo" accept="image/*" required>
        </div>
        <button type="submit">Subir</button>
    </form>
</body>
</html>
```

PHP - Procesamiento de la Subida de Imagen (procesar.php)

El archivo `procesar.php` recibe el archivo y lo guarda en un directorio del servidor. La información del archivo se encuentra en la variable global `$_FILES`.

```php
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST' && isset($_FILES['archivo'])) {
    $archivo = $_FILES['archivo'];
    $nombreArchivo = basename($archivo['name']);
    $rutaTemporal = $archivo['tmp_name'];
    $directorioDestino = 'uploads/' . $nombreArchivo;

    // Crear el directorio si no existe
    if (!is_dir('uploads')) {
        mkdir('uploads', 0777, true);
    }

    // Mover el archivo a la ubicación final
    if (move_uploaded_file($rutaTemporal, $directorioDestino)) {
        echo "Imagen subida correctamente: " . htmlspecialchars($nombreArchivo);
    } else {
        echo "Error al subir la imagen.";
    }
}
?>
```

3. POST con JSON o XML * Accediendo a los formularios desde cliente para enviarlo a servidor (El método habitual)

3.1. Concepto

El uso de **POST con JSON o XML** es común en las APIs RESTful para el envío de datos estructurados. En este caso, el contenido del formulario se envía en el cuerpo de la solicitud HTTP en formato **JSON** o **XML**. Esto es especialmente útil cuando se trabaja con datos que deben ser procesados por una API en lugar de un simple formulario web.

Para enviar datos en este formato, se debe establecer el encabezado `Content-Type` de la solicitud a `application/json` o `application/xml`, según el caso.

3.2. Ejemplo de Implementación: Creación de un Nuevo Usuario mediante JSON

HTML y JavaScript - Formulario con Envío de JSON

Este ejemplo muestra un formulario simple y una función en JavaScript que envía los datos en formato JSON al servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Crear Usuario</title>
</head>
<body>
    <h1>Crear Nuevo Usuario</h1>
    <form id="form-usuario">
        <div>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre" required>
        </div>
        <div>
            <label for="email">Correo Electrónico:</label>
            <input type="email" id="email" name="email" required>
        </div>
        <button type="button" onclick="enviarFormulario()">Crear Usuario</button>
    </form>

    <script>
        function enviarFormulario() {
            const nombre = document.getElementById('nombre').value;
            const email = document.getElementById('email').value;

            const datos = {
                nombre: nombre,
                email: email
            };

            fetch('procesar.php', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(datos)
            })
            .then(response => response.json())
            .then(data => {
                alert('Respuesta del servidor: ' + JSON.stringify(data));
            })
            .catch(error => console.error('Error:', error));
        }
    </script>
</body>
</html>
```

PHP - Procesamiento de Datos en JSON (procesar.php)

El archivo `procesar.php` maneja la solicitud JSON y responde con un mensaje de éxito o error.

```
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $contenido = file_get_contents('php://input');
    $datos = json_decode($contenido, true);

    if ($datos && isset($datos['nombre']) && isset($datos['email'])) {
        $nombre = $datos['nombre'];
        $email = $datos['email'];

          echo json_encode(["mensaje" => "Usuario creado con éxito", "nombre" => $nombre,
"email" => $email]);
    } else {
        echo json_encode(["error" => "Datos inválidos"]);
    }
}
?>
```

Ver ejemplos de formularios formularios_completos.zip

La importancia del objeto Fetch. (Mi primera api restFull)

El Objeto `fetch` *Esto es cliente, pero lo introducimos para comenzar a conocerlo.

1. ¿Qué es `fetch`?

El objeto **`fetch`** es una API moderna de JavaScript que permite realizar solicitudes HTTP (como **GET**, **POST**, **PUT**, **DELETE**, etc.) de forma asíncrona para interactuar con servidores web. Fue introducido para simplificar y mejorar la experiencia de los desarrolladores al trabajar con solicitudes HTTP, proporcionando una alternativa más limpia y manejable en comparación con el objeto **`XMLHttpRequest`**.

El uso de **`fetch`** facilita la obtención de recursos y datos de un servidor y la carga dinámica de contenido en una página web sin necesidad de recargarla por completo, lo que es común en aplicaciones web modernas y **APIs RESTful**.

2. Características de `fetch`

- **Basado en promesas**: `fetch` utiliza promesas para manejar el flujo asíncrono de solicitudes y respuestas. Esto permite un manejo más claro y organizado de las operaciones asíncronas utilizando métodos como `.then()` para manejar respuestas exitosas y `.catch()` para manejar errores.
- **Sintaxis limpia y moderna**: La sintaxis de `fetch` es más simple y legible que la de `XMLHttpRequest`, lo que facilita la lectura y mantenimiento del código.
- **Soporte para solicitudes HTTP completas**: Permite realizar operaciones con cualquier método HTTP (`GET`, `POST`, `PUT`, `DELETE`, etc.), además de poder configurar encabezados personalizados y enviar datos en distintos formatos (JSON, texto, formularios, etc.).

- **Manejo de errores más intuitivo**: Los errores relacionados con la red (como la imposibilidad de contactar con el servidor) se manejan fácilmente utilizando el método `.catch()`.

3. Sintaxis Básica de `fetch`

El uso básico de `fetch` sigue la siguiente sintaxis:

```
fetch(url, opciones)
    .then(response => {
        // Manejo de la respuesta
    })
    .catch(error => {
        // Manejo del error
    });
```

Donde:

- **`url`**: Es la URL a la que se realizará la solicitud.
- **`opciones`**: Es un objeto opcional que puede incluir configuraciones adicionales, como el método HTTP, los encabezados, el cuerpo de la solicitud, etc.

4. Ejemplos de Uso del Objeto `fetch`

4.1. Solicitud GET

Para realizar una solicitud **GET** y obtener datos de un servidor, se puede usar la siguiente sintaxis:

```
fetch('https://api.example.com/datos')
    .then(response => {
        if (!response.ok) {
            throw new Error('Error en la solicitud: ' + response.status);
        }
```

```
    return response.json();
  })
  .then(data => {
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

En este ejemplo:

- Se realiza una solicitud **GET** a la URL `https://api.example.com/datos`.
- Si la respuesta no es satisfactoria (el código de estado no está en el rango 200-299), se lanza un error.
- Si la respuesta es exitosa, se convierte a formato JSON usando `.json()` y los datos se manejan en el siguiente bloque `.then()`.

4.2. Solicitud POST

Para enviar datos al servidor (por ejemplo, al crear un nuevo recurso), se puede utilizar el método **POST** de la siguiente forma:

```
const datos = {
  nombre: 'Juan',
  email: 'juan@example.com'
};

fetch('https://api.example.com/usuarios', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(datos)
})
  .then(response => {
    if (!response.ok) {
```

```
        throw new Error('Error en la solicitud: ' + response.status);
    }
    return response.json();
})
.then(data => {
    console.log('Respuesta del servidor:', data);
})
.catch(error => {
    console.error('Error:', error);
});
```

En este ejemplo:

- Se configura la solicitud con el método **POST** y se especifican los **encabezados** para indicar que el contenido enviado es de tipo JSON.
- El cuerpo de la solicitud (`body`) contiene los datos, que deben ser convertidos a una cadena JSON con `JSON.stringify()` antes de enviarlos.
- La respuesta se maneja de manera similar al ejemplo de **GET**.

5. Opciones del Objeto `fetch`

El objeto de opciones permite configurar diversos aspectos de la solicitud. Algunas de las configuraciones más comunes son:

- **`method`**: Especifica el método HTTP (`GET`, `POST`, `PUT`, `DELETE`, etc.).
- **`headers`**: Permite establecer los encabezados HTTP personalizados (como `Content-Type`).
- **`body`**: Contiene el cuerpo de la solicitud. Es necesario para métodos como **POST** o **PUT**.
- **`mode`**: Controla el modo de la solicitud, como `cors`, `no-cors`, o `same-origin`.
- **`credentials`**: Indica si se deben enviar cookies junto con la solicitud (`omit`, `same-origin`, `include`).

Ejemplo con algunas de estas opciones:

```
fetch('https://api.example.com/usuarios', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer token123'
    },
    body: JSON.stringify({ nombre: 'Ana', edad: 30 }),
    mode: 'cors',
    credentials: 'include'
})
```

6. Manejo de Errores en `fetch`

A diferencia de **`XMLHttpRequest`**, **`fetch`** no lanza errores para códigos de estado HTTP que indican fallos (como 404 o 500). En cambio, la propiedad `response.ok` se debe verificar para confirmar si la solicitud fue exitosa.

Por ejemplo:

```
fetch('https://api.example.com/datos')
    .then(response => {
        if (!response.ok) {
            throw new Error('Error HTTP: ' + response.status);
        }
        return response.json();
    })
    .catch(error => {
        console.error('Hubo un problema con la solicitud:', error);
    });
```

7. Uso de `async/await` con `fetch`

El uso de `async/await` simplifica aún más el código, haciéndolo más parecido al manejo síncrono:

```
async function obtenerDatos() {
    try {
        const response = await fetch('https://api.example.com/datos');
        if (!response.ok) {
            throw new Error('Error HTTP: ' + response.status);
        }
        const data = await response.json();
        console.log('Datos recibidos:', data);
    } catch (error) {
        console.error('Error en la solicitud:', error);
    }
}

obtenerDatos();
```

Ventajas de `fetch` sobre `XMLHttpRequest`

1. **API más moderna y fácil de usar**:
    - **`fetch`** es una API moderna que se introdujo para simplificar el manejo de solicitudes HTTP en JavaScript. La sintaxis es más limpia y legible en comparación con **`XMLHttpRequest`**, lo que facilita la comprensión y el mantenimiento del código.
    - Con **`fetch`**, las solicitudes se realizan mediante promesas, lo que permite manejar las respuestas de forma más intuitiva utilizando `.then()` y `.catch()` para el control de errores.
2. **Soporte para promesas**:
    - **`fetch`** está basado en promesas, lo que facilita trabajar con flujos de control asíncronos. Esto evita la necesidad de manejar eventos o callbacks complejos como en **`XMLHttpRequest`**.
    - Las promesas en **`fetch`** permiten un flujo de control más natural cuando se combina con `async/await`, lo que simplifica aún más la lectura del código.
3. **Mayor compatibilidad con APIs modernas**:
    - **`fetch`** tiene soporte incorporado para la configuración de solicitudes con `Content-Type: application/json`, lo que lo hace ideal para trabajar

con **APIs RESTful** que requieren el envío de datos en formatos JSON o similares.
    - Además, **`fetch`** permite especificar de manera sencilla métodos HTTP como `POST`, `PUT`, `DELETE`, y otras configuraciones en un solo objeto de configuración.
4. **Manejo simplificado de la respuesta**:
    - Con **`fetch`**, la respuesta se maneja como un flujo (stream), lo que permite trabajar fácilmente con formatos de respuesta como JSON, texto, o blobs, usando métodos como `.json()`, `.text()`, o `.blob()` para convertir la respuesta.
5. **Menos configuración para solicitudes básicas**:
    - Para realizar solicitudes básicas, **`fetch`** requiere menos configuración en comparación con **`XMLHttpRequest`**, que necesita configurar manualmente muchos aspectos de la solicitud (como el método HTTP y los encabezados).

Comparación de Ejemplo entre `fetch` y `XMLHttpRequest`

Para entender mejor las diferencias, veamos cómo se realizaría la misma solicitud para enviar datos JSON usando ambas APIs.

```
function enviarFormulario() {
    const nombre = document.getElementById('nombre').value;
    const email = document.getElementById('email').value;

    const datos = {
        nombre: nombre,
        email: email
    };

    fetch('procesar.php', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(datos)
    })
    .then(response => response.json())
    .then(data => {
        alert('Respuesta del servidor: ' + JSON.stringify(data));
    })
    .catch(error => console.error('Error:', error));
}
```

```
function enviarFormulario() {
    const nombre = document.getElementById('nombre').value;
    const email = document.getElementById('email').value;

    const datos = {
        nombre: nombre,
        email: email
    };

    const xhr = new XMLHttpRequest();
    xhr.open('POST', 'procesar.php', true);
    xhr.setRequestHeader('Content-Type', 'application/json');

    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4) {
            if (xhr.status === 200) {
                const response = JSON.parse(xhr.responseText);
                alert('Respuesta del servidor: ' + JSON.stringify(response));
            } else {
                console.error('Error:', xhr.statusText);
            }
        }
    };

    xhr.send(JSON.stringify(datos));
}
```

Diferencias Clave

1. **Simplicidad y legibilidad**:
    - El ejemplo con **`fetch`** es más fácil de leer y mantener. En comparación, el código de **`XMLHttpRequest`** necesita más líneas y una configuración más explícita.
2. **Manejo de Promesas vs. Callbacks**:
    - **`fetch`** utiliza promesas, lo que permite un flujo de control más limpio con `.then()` y `.catch()`. En cambio, **`XMLHttpRequest`** depende de callbacks y el uso del estado de la solicitud (`readyState`) para manejar las respuestas.
3. **Manejo de Errores**:

- Con **`fetch`**, el manejo de errores se realiza fácilmente mediante `.catch()`, mientras que con **`XMLHttpRequest`**, se debe verificar manualmente el estado de la respuesta (`xhr.status`).

Consideraciones sobre `XMLHttpRequest`

Aunque **`fetch`** es la opción recomendada para la mayoría de los casos, **`XMLHttpRequest`** aún puede ser útil en escenarios específicos:

- Soporte para navegadores muy antiguos (Internet Explorer 11 o anteriores).
- Necesidad de usar funcionalidades que no están disponibles en **`fetch`**, como el seguimiento de progreso de carga de archivos (`xhr.upload`).

Ver ejemplo: api_restful_simulation.zip

Taller: Mi Primera API RESTful

En este taller, vamos a aprender los conceptos básicos sobre cómo construir y consumir una API RESTful simulada. El objetivo es que los alumnos comprendan cómo interactuar con una API utilizando la herramienta `fetch` en JavaScript y cómo crear una API básica en el servidor con PHP. Vamos a desarrollar una simulación de una API RESTful con las funcionalidades de login, obtención de datos, creación y eliminación de recursos.

1. ¿Qué es una API RESTful?

Una **API RESTful** es una interfaz que permite que diferentes sistemas se comuniquen entre sí usando el protocolo HTTP. El término "REST" significa "Representational State Transfer" y se basa en las operaciones básicas de los métodos HTTP: **GET**, **POST**, **PUT**, **DELETE**, etc.

- **GET**: Obtener datos de un recurso.
- **POST**: Crear un nuevo recurso.
- **PUT/PATCH**: Actualizar un recurso existente.
- **DELETE**: Eliminar un recurso.

1.1. ¿Por qué usar una API RESTful?

- Facilita la comunicación entre aplicaciones.
- Estandariza las operaciones para interactuar con los recursos.
- Permite la integración con múltiples clientes (navegadores, aplicaciones móviles, etc.).

2. Estructura del Proyecto

El proyecto consistirá en los siguientes archivos:

1. **`api.php`**: Archivo PHP que simula la API RESTful.
2. **`login.html`**: Página de inicio de sesión.
3. **`dashboard.html`**: Panel de administración donde se visualizan, agregan y eliminan usuarios.
4. **`fetch` en JavaScript**: Para consumir la API desde el lado del cliente.

2.1. Simulación de la API con `api.php`

El archivo `api.php` actuará como la API, ofreciendo los siguientes servicios:

- **Login** (`POST /api.php?action=login`): Simula la autenticación de usuarios.
- **Obtener Usuarios** (`GET /api.php`): Devuelve una lista de usuarios.
- **Agregar Usuario** (`POST /api.php?action=add`): Agrega un nuevo usuario.
- **Eliminar Usuario** (`DELETE /api.php`): Elimina un usuario.

Código de `api.php`:

```
<?php
header('Content-Type: application/json');

$usuarios = [
    ["id" => 1, "nombre" => "Juan", "apellidos" => "Pérez", "direccion" => "Calle A"],
    ["id" => 2, "nombre" => "Ana", "apellidos" => "García", "direccion" => "Calle B"],
    ["id" => 3, "nombre" => "Luis", "apellidos" => "Martínez", "direccion" => "Calle C"],
];

session_start();

switch ($_SERVER['REQUEST_METHOD']) {
    case 'POST':
        if (isset($_GET['action']) && $_GET['action'] == 'login') {
            $contenido = file_get_contents('php://input');
            $datos = json_decode($contenido, true);

            if ($datos['username'] === 'admin' && $datos['password'] === '1234') {
                $_SESSION['loggedin'] = true;
                echo json_encode(["mensaje" => "Login exitoso"]);
            } else {
                http_response_code(401);
                echo json_encode(["error" => "Credenciales incorrectas"]);
```

```
        }
    } elseif (isset($_GET['action']) && $_GET['action'] == 'add') {
        $contenido = file_get_contents('php://input');
        $datos = json_decode($contenido, true);
        $nuevoUsuario = [
            "id" => count($usuarios) + 1,
            "nombre" => $datos['nombre'],
            "apellidos" => $datos['apellidos'],
            "direccion" => $datos['direccion']
        ];
        $usuarios[] = $nuevoUsuario;
            echo json_encode(["mensaje" => "Usuario agregado con éxito", "usuario" =>
$nuevoUsuario]);
    }
    break;

case 'GET':
    if (isset($_SESSION['loggedin']) && $_SESSION['loggedin']) {
        echo json_encode($usuarios);
    } else {
        http_response_code(401);
        echo json_encode(["error" => "No autorizado"]);
    }
    break;

case 'DELETE':
    parse_str(file_get_contents("php://input"), $datos);
    $id = $datos['id'];
    $index = array_search($id, array_column($usuarios, 'id'));

    if ($index !== false) {
        array_splice($usuarios, $index, 1);
        echo json_encode(["mensaje" => "Usuario eliminado con éxito"]);
    } else {
        http_response_code(404);
        echo json_encode(["error" => "Usuario no encontrado"]);
    }
    break;

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

2.2. Frontend con `login.html` y `dashboard.html`

Código de `login.html`

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
        <link  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">
    <div class="container mt-5">
        <div class="row justify-content-center">
            <div class="col-md-6">
                <div class="card p-4 shadow-sm">
                    <h2 class="text-center mb-4">Iniciar Sesión</h2>
                    <form id="form-login">
                        <div class="mb-3">
                            <label for="username" class="form-label">Usuario</label>
                            <input type="text" id="username" class="form-control" required>
                        </div>
                        <div class="mb-3">
                            <label for="password" class="form-label">Contraseña</label>
                            <input type="password" id="password" class="form-control" required>
                        </div>
                                                <button type="button" class="btn btn-primary w-100"
onclick="login()">Iniciar Sesión</button>
                    </form>
                </div>
            </div>
        </div>
    </div>

    <script>
        function login() {
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;

            fetch('api.php?action=login', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ username, password })
            })
            .then(response => response.json())
            .then(data => {
                if (data.mensaje) {
                    window.location.href = 'dashboard.html';
                } else {
```

```
                alert('Error en el login: ' + data.error);
            }
        })
        .catch(error => console.error('Error:', error));
    }
  </script>
</body>
</html>
```

Código de `dashboard.html`

El archivo `dashboard.html` es el panel principal, donde se muestra el listado de usuarios y permite añadir o eliminar usuarios.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Dashboard</title>
      <link  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">
  <div class="container mt-5">
    <h2 class="text-center mb-4">Listado de Usuarios</h2>
                    <button  class="btn  btn-success  mb-3"  data-bs-toggle="modal"
data-bs-target="#addUserModal">Agregar Usuario</button>
    <div id="usuarios-listado" class="card p-4 shadow-sm"></div>
  </div>

                <div  class="modal  fade"  id="addUserModal"  tabindex="-1"
aria-labelledby="addUserModalLabel" aria-hidden="true">
    <div class="modal-dialog">
      <div class="modal-content">
        <div class="modal-header">
          <h5 class="modal-title" id="addUserModalLabel">Agregar Usuario</h5>
                    <button  type="button"  class="btn-close"  data-bs-dismiss="modal"
aria-label="Close"></button>
        </div>
        <div class="modal-body">
          <form id="form-add-user">
            <div class="mb-3">
              <label for="nombre" class="form-label">Nombre</label>
              <input type="text" id="nombre" class="form-control" required>
```

```
                </div>
                <div class="mb-3">
                    <label for="apellidos" class="form-label">Apellidos</label>
                    <input type="text" id="apellidos" class="form-control" required>
                </div>
                <div class="mb-3">
                    <label for="direccion" class="form-label">Dirección</label>
                    <input type="text" id="direccion" class="form-control" required>
                </div>
                                                    <button type="button" class="btn btn-success"
onclick="addUser()">Agregar</button>
            </form>
        </div>
    </div>
</div>

<script>
    document.addEventListener('DOMContentLoaded', function() {
        fetchUsuarios();
    });

    function fetchUsuarios() {
        fetch('api.php')
            .then(response => response.json())
            .then(data => {
                let html = '<ul class="list-group">';
                data.forEach(usuario => {
                                html += `<li class="list-group-item d-flex justify-content-between
align-items-center">
                            ${usuario.nombre} ${usuario.apellidos} - ${usuario.direccion}
                                                        <button class="btn btn-danger btn-sm"
onclick="deleteUser(${usuario.id})">Eliminar</button>
                            </li>`;
                });
                html += '</ul>';
                document.getElementById('usuarios-listado').innerHTML = html;
            })
            .catch(error => console.error('Error:', error));
    }

    function deleteUser(id) {
        fetch('api.php', {
            method: 'DELETE',
            headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
            body: `id=${id}`
        })
        .then(response => response.json())
        .then(data => {
```

```
            alert(data.mensaje);
            fetchUsuarios();
        })
        .catch(error => console.error('Error:', error));
    }

    function addUser() {
        const nombre = document.getElementById('nombre').value;
        const apellidos = document.getElementById('apellidos').value;
        const direccion = document.getElementById('direccion').value;

        fetch('api.php?action=add', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ nombre, apellidos, direccion })
        })
        .then(response => response.json())
        .then(data => {
            alert(data.mensaje);
            fetchUsuarios();
            document.getElementById('form-add-user').reset();
                                                    var addUserModal = new
bootstrap.Modal(document.getElementById('addUserModal'));
            addUserModal.hide();
        })
        .catch(error => console.error('Error:', error));
    }
</script>
                                                                    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

3. Explicación del Taller

3.1. Interacción Cliente-Servidor

1. **Inicio de sesión**: El usuario ingresa sus credenciales en `login.html`, y la función `login()` envía una solicitud POST a la API simulada (`api.php?action=login`). Si el inicio de sesión es exitoso, se redirige al usuario a `dashboard.html`.
2. **Listado de usuarios**: Una vez en el dashboard, se ejecuta la función `fetchUsuarios()`, que realiza una solicitud GET a la API (`api.php`) para obtener el listado de usuarios y mostrarlos en pantalla.

3. **Agregar un usuario**: Cuando se envía el formulario para agregar un usuario, se llama a la función `addUser()`, que realiza una solicitud POST a la API (`api.php?action=add`) para agregar el nuevo usuario.
4. **Eliminar un usuario**: Al hacer clic en el botón "Eliminar" de un usuario, se llama a la función `deleteUser()`, que realiza una solicitud DELETE a la API con el ID del usuario.
5. Cuando una API empieza a crecer y manejar muchas entidades, utilizar múltiples `if` para gestionar las diferentes operaciones puede volverse complicado y difícil de mantener. Más adelante veremos el concepto de Routing…..

Anexo JSON

Interacción con JSON en PHP: Serialización y Envío de Datos en Formato JSON

JSON (JavaScript Object Notation) es un formato de datos ligero que se utiliza para el intercambio de datos entre un servidor y una aplicación web, o entre aplicaciones en general. En PHP, JSON es muy útil para enviar y recibir datos estructurados, ya que facilita la interacción entre el frontend y el backend.

1. ¿Qué es JSON?

- **JSON** es un formato de texto sencillo y fácil de leer para estructurar datos. Representa datos como pares clave-valor.
- Muy utilizado en APIs y servicios web, permite la comunicación entre el servidor y el cliente (por ejemplo, entre PHP y JavaScript).

Ejemplo de un objeto JSON:

```
{
   "nombre": "Juan",
   "edad": 25,
   "email": "juan@example.com"
}
```

2. Serialización y Deserialización en PHP

Serialización convierte un objeto PHP o un array en una cadena JSON, mientras que **deserialización** convierte una cadena JSON en un objeto o array PHP.

2.1. Funciones de Serialización (`json_encode`)

La función `json_encode` convierte datos en formato JSON, útil para:

- Enviar datos JSON desde el servidor al cliente.
- Guardar datos en formato JSON para futuras consultas.

Sintaxis:

```
json_encode($datos, opciones, profundidad);
```

`$datos`: Array o variable a convertir en JSON.

`opciones`: Opcional; permite definir opciones como `JSON_PRETTY_PRINT`.

`profundidad`: Opcional; establece la profundidad máxima del objeto.

Ejemplo Básico de json_encode:

```
$usuario = [
    "nombre" => "Juan",
    "edad" => 25,
    "email" => "juan@example.com"
];

$jsonUsuario = json_encode($usuario);
echo $jsonUsuario;
```

Salida:

```
{"nombre":"Juan","edad":25,"email":"juan@example.com"}
```

Ejemplo de `json_encode` con JSON bonito (formato de salida más legible):

```
$jsonUsuarioBonito = json_encode($usuario, JSON_PRETTY_PRINT);
echo $jsonUsuarioBonito;
```

Salida

```
{
    "nombre": "Juan",
    "edad": 25,
    "email": "juan@example.com"
}
```

2.2. Funciones de Deserialización (`json_decode`)

La función `json_decode` convierte una cadena JSON en un objeto o array PHP, útil cuando se reciben datos JSON (por ejemplo, desde el frontend) y se necesita acceder a ellos en PHP.

Sintaxis:

```
json_decode($cadenaJson, $asociativo, profundidad, opciones);
```

- `cadenaJson`: La cadena JSON a deserializar.
- `$asociativo`: Cuando es `true`, convierte el JSON en un array asociativo; cuando es `false` (predeterminado), convierte el JSON en un objeto.
- `profundidad` y `opciones`: Opcionales; permiten definir detalles adicionales.

Ejemplo de `json_decode` en un Array Asociativo:

```
$jsonDatos = '{"nombre":"Juan","edad":25,"email":"juan@example.com"}';
$arrayDatos = json_decode($jsonDatos, true);

echo "Nombre: " . $arrayDatos["nombre"];
```

Salida

```
Nombre: Juan
```

Ejemplo de `json_decode` en un Objeto:

```
$objetoDatos = json_decode($jsonDatos);
echo "Nombre: " . $objetoDatos->nombre;
```

3. Envío de Datos en Formato JSON desde el Servidor

Cuando desarrollamos una API en PHP, es común que el servidor responda con datos en formato JSON.

Ejemplo: Envío de una Respuesta en JSON

Para enviar datos JSON desde el servidor al cliente, configuramos la cabecera HTTP y luego usamos `json_encode` para devolver los datos.

```
header('Content-Type: application/json');

$datos = [
    "status" => "success",
```

```
    "message" => "Operación completada correctamente"
];

echo json_encode($datos);
```

4. Ejemplo Completo: API Básica con JSON

A continuación, se presenta un ejemplo de API en PHP que permite:

- Obtener una lista de usuarios.
- Agregar un nuevo usuario.

Código PHP para el API

```
header('Content-Type: application/json');

$usuarios = [
    ["id" => 1, "nombre" => "Juan", "email" => "juan@example.com"],
    ["id" => 2, "nombre" => "Ana", "email" => "ana@example.com"]
];

if ($_SERVER['REQUEST_METHOD'] === 'GET') {
    echo json_encode($usuarios);
} elseif ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $datosRecibidos = json_decode(file_get_contents('php://input'), true);
    $nuevoUsuario = [
        "id" => count($usuarios) + 1,
        "nombre" => $datosRecibidos["nombre"],
        "email" => $datosRecibidos["email"]
    ];
    $usuarios[] = $nuevoUsuario;
    echo json_encode([
        "status" => "success",
        "usuario" => $nuevoUsuario
    ]);
} else {
    http_response_code(405);
    echo json_encode(["error" => "Método no permitido"]);
}
```

- La API responde a solicitudes GET y POST.

- Para GET, devuelve la lista de usuarios en JSON.
- Para POST, decodifica el JSON recibido con `json_decode`, agrega un nuevo usuario al arreglo y devuelve la información del nuevo usuario en JSON.

5. Envío de Datos JSON desde el Cliente a PHP

El cliente puede enviar datos en formato JSON al servidor mediante `fetch` o `XMLHttpRequest` desde JavaScript.

Ejemplo de Solicitud POST desde JavaScript

```
const nuevoUsuario = {
    nombre: "Carlos",
    email: "carlos@example.com"
};

fetch("http://localhost/api/usuarios", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify(nuevoUsuario)
})
.then(response => response.json())
.then(data => console.log("Respuesta del servidor:", data))
.catch(error => console.error("Error:", error));
```

7. Buenas Prácticas al Trabajar con JSON en PHP

1. **Configurar cabeceras adecuadamente**: Asegúrate de usar `header('Content-Type: application/json')` para que el cliente interprete correctamente la respuesta JSON.
2. **Validar datos recibidos**: Usa `json_decode` con cuidado y verifica que los datos no estén vacíos o malformados antes de procesarlos.
3. **JSON_PRETTY_PRINT para depuración**: Úsalo solo en desarrollo o depuración; en producción, omite esta opción para reducir el tamaño de la respuesta JSON.