- ✓Ajax: Peticiones HTTP
- ✓Fetch: Peticiones Asíncronas
- ✓ Fetch: método .then
- ✓Fetch: async/await

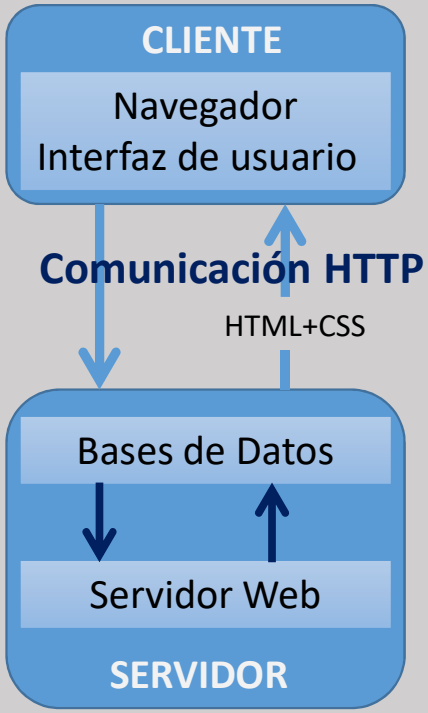
Modelos de comunicación

COMUNICACIÓN CLÁSICA

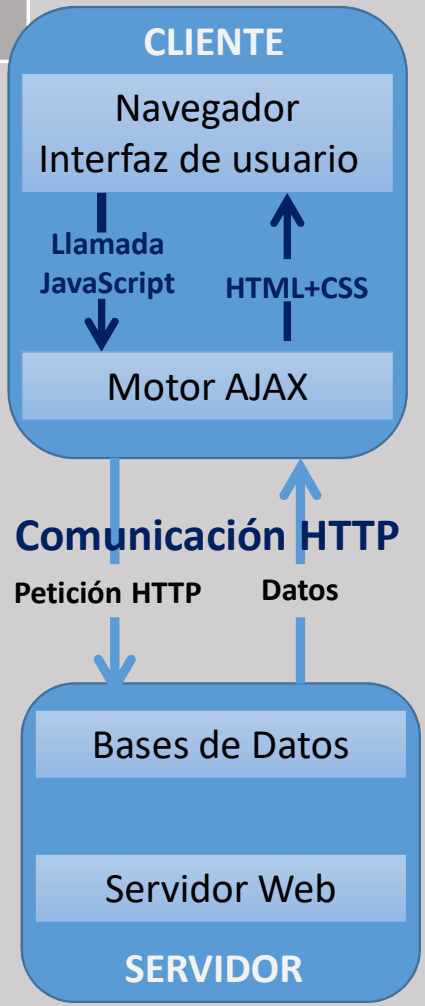
- Continuas recargas de la página
- Información actualizada a partir del comportamiento del usuario.
- Condiciona la petición realizada al servidor
- Envío continuo de formularios con datos al servidor lo que conlleva sobrecarga en la red
- Imposibilidad de interactuar con otro elemento de la página hasta que no se ha cargado nuevamente

COMUNICACIÓN MEDIANTE AJAX

- Procesos en segundo plano
- Se elimina la necesidad de recarga de la página
- Cambios y/o actualizaciones parciales
- El motor AJAX se encarga de gestionar las peticiones del usuario, y de comunicarse con el servidor. Permite que la interacción suceda de forma asíncrona



COMUNICACIÓN CLÁSICA



COMUNICACIÓN MEDIANTE AJAX

AJAX: Peticiones HTTP

Un **navegador**, durante la carga de una página, suele realizar múltiples **peticiones HTTP** a un servidor para solicitar los archivos que necesita renderizar en la página.
Ejemplo:
- El documento .html de la página (*aquí encontraremos referencias a otros archivos*)
- La hoja de estilos .css (*probablemente existan aquí más referencias a otros archivos*)
- Imágenes .jpg, .png, .webp u otras
- Scripts .js (*aquí nuevamente, más referencias a otros archivos*)
- Tipografías .ttf, .woff o .woff2

¿Qué es una petición HTTP?

Una **petición HTTP** es la acción por parte del navegador de solicitar a un servidor web un documento o archivo, ya sea un fichero .html, una imagen, una tipografía, un archivo .js, etc.
Gracias a dicha petición, el navegador puede descargar ese archivo, almacenarlo en un **caché temporal de archivos del navegador** y, finalmente, mostrarlo en la página actual que lo ha solicitado.
Además de la petición de carga de página, hay peticiones de forma transparente al usuario, sin que visualmente se reflejen en la página actual en la que estamos.

AJAX: Peticiones HTTP

Peticiones HTTP mediante AJAX

AJAX (*Asynchronous Javascript and XML*) es una modalidad de peticiones HTTP basada en que la **petición HTTP** se realiza desde Javascript, de forma transparente al usuario, descargando la información y pudiendo tratarla **sin necesidad de mostrarla directamente en la página**.
Es una novedad, puesto que podemos hacer actualizaciones de contenidos **de forma parcial**, de modo que se actualice una página «**en vivo**», sin necesidad de recargar toda la página, sino solamente actualizado una pequeña parte de ella.

Originalmente, a este sistema de realización de peticiones HTTP se le llamó **AJAX**, donde la X significa **XML**, el formato ligero de datos que más se utilizaba en aquel entonces. Actualmente, sobre todo en el mundo Javascript, se utiliza más el formato **JSON**, aunque por razones fonéticas evidentes se sigue manteniendo el término **AJAX** en lugar del horrible correspondiente **AJAJ**.
Posteriormente, y debido a una evolución mayor, se ha pasado de crear páginas de tipo **MPA**, a crear páginas de tipo **SPA**, mucho más frecuentes en entornos empresariales hoy en día

AJAX: Peticiones HTTP

MPA: Multiple Page Application

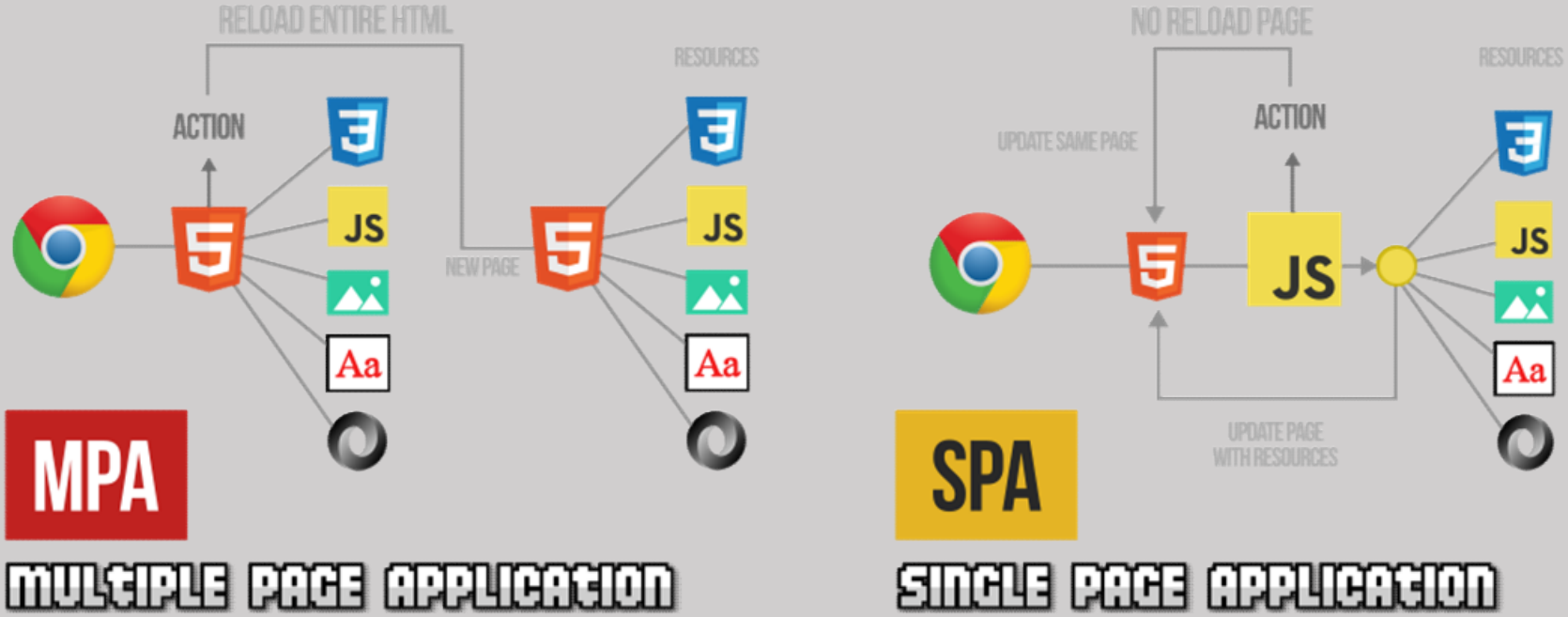
Tradicionalmente, se creaban webs dentro de la categoría de páginas **MPA** (*Multiple Page Application*). En ellas, el navegador se descarga el fichero .html, lo lee y luego realiza las peticiones de los restantes archivos relacionados que encuentra en el documento HTML. Si el usuario pulsa en algún enlace, se descarga el .html de dicho enlace (*recargando la página completa*) y se repite el proceso. La navegación se producía mediante enlaces, y al hacer click en ellos, se recarga la página completa. Generalmente, es el que se utiliza frecuentemente en sitios web más tradicionales, los que usan mayoritariamente backend o necesitan SEO (*posicionamiento en buscadores*).

SPA: Single Page Application

En el lado opuesto se encuentran las páginas de tipo **SPA** (*Single Page Application*). Enfoque posterior, donde el navegador se descarga una versión básica .html junto a un .js que se encargará de controlar toda la web. Realizará peticiones de los archivos relacionados junto a peticiones a archivos .json o .js con más información o nuevos contenidos, que mostrará en el navegador parcial o completamente, pero sin la necesidad obligatoria de recargar la página completamente, conservando el contenido descargado previamente.

AJAX: Peticiones HTTP

Este sistema se utiliza mayoritariamente para construir aplicaciones web como sitios de gestión en los que no necesitamos «navegar» a través de una serie de páginas. Ejemplos de este tipo de páginas podrían ser la versión web de WhatsApp, Twitter o Google Drive podrían ser ejemplos de **SPA**, que van más enfocados en la funcionalidad y acciones, que en el contenido.
Las páginas de tipo **SPA** son las que utilizan en la mayoría de los frameworks de Javascript, como por ejemplo, **React**, **Vue**, **Angular** o **Svelte**.



AJAX: Peticiones HTTP

Métodos de petición AJAX

Existen varias formas de realizar **peticiones HTTP mediante AJAX**, pero las principales suelen ser XMLHttpRequest y fetch (*nativas, incluidas en el navegador por defecto*), además de liberías externas como por ejemplo axios o superagent:

Método	Descripción
AJAX:	
	Métodos nativos del navegador
XMLHttpRequest	Abreviado como XHR. El más antiguo, y también más verbose. Nativo.
fetch	Nuevo sistema nativo de peticiones basado en promesas. Nativo.
	Librerías externas
Axios	Librería basada en promesas para realizar peticiones en Node o en navegadores.
superagent	Librería para realizar peticiones HTTP tanto en Node como en navegadores.
frisbee	Librería basada en fetch. Suele usarse junto a React Native.

Fetch: Peticiones asíncronas

La forma de trabajar con objetos XMLHttpRequest, aunque es muy potente requiere mucho trabajo que hace que el código no sea tan legible y práctico como quizás debería. Entonces es cuando surge fetch, un sistema más moderno, basado en promesas de Javascript, para realizar peticiones HTTP asíncronas de una forma más legible y cómoda.

Peticiones con el método fetch()

Fetch es el nombre de una nueva API para Javascript con la cuál podemos realizar peticiones HTTP asíncronas utilizando promesas y de forma que el código sea un poco más sencillo y menos verbose.

¿Qué son las promesas?

Las **promesas** son un concepto para simplificar el manejo de la asincronía en Javascript de una forma más sencilla que con las callbacks.

Como en la vida real, una **promesa** es algo que en principio pensamos que se cumplirá en el futuro, pero a medida que pasa el tiempo pueden ocurrir varias cosas:

- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa rechazada*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Fetch: Peticiones asíncronas

Entendiendo las promesas

Hasta ahora, trabajábamos con código síncrono, que se ejecuta de forma secuencial, es decir, una línea detrás de otra:

```
primera_funcion();
segunda_funcion();
tercera_funcion();
```

Las promesas son una especie de condición que «se queda en espera» y se ejecutará en el futuro. De esta forma, podemos manejar código asíncrono sin bloquear la ejecución del resto del programa. Al ejecutar la función asíncrona, el programa continua con la siguiente línea de código. Realmente, no ha terminado de ejecutar el código de la función anterior, pero se ha quedado en espera, hasta que se cumpla el criterio o condición de la promesa.

Las **promesas** en Javascript son un tipo de dato. Son un OBJECT de tipo PROMISE . Cuando obtengamos uno de estos objetos (*generalmente, devuelto por una función*) sabemos que tenemos una promesa que puede que se cumpla en un futuro cercano o lejano.

Fetch: Peticiones asíncronas

Peticiones con el método fetch()

La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a fetch y pasarle por parámetro la URL de la petición a realizar:

```
const promise = fetch("/robots.txt");
promise.then(function(response) {
    /* ... */
});
```

O escrita de una forma más legible:

```
fetch("/robots.txt")
    .then(function(response) {
        /** Código que procesa la respuesta **/
        });
```

El fetch() hará una petición a una url, y se devolverá una PROMISE que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición. Será gestionada mediante el then.

El modo más habitual de manejar las promesas es utilizando **.then()**.
También se puede utilizar **async/await** (LO VEREMOS MÁS ADELANTE)

Fetch: Peticiones asíncronas

Opciones de fetch()

Fetch puede tener un segundo parámetro de opciones de forma opcional, un OBJECT con opciones de la petición HTTP:

```
const options = {
        method: "GET"
};
fetch("/robots.txt", options)
        .then(response => response.text())
        ……
```

En este Objeto options podemos definir varios detalles:

Campo	Descripción
method	Método HTTP de la petición. Por defecto, GET. Otras opciones: HEAD, POST, etc...
headers	Cabeceras HTTP. Por defecto, {}.
body	Cuerpo de la petición HTTP. Puede ser de varios tipos: String, FormData, Blob, etc...
credentials	Modo de credenciales. Por defecto, omit. Otras opciones: same-origin e include.

Fetch: Peticiones asíncronas

En options definimos varias opciones: 1º) método HTTP a realizar en la petición. Por defecto, será GET, pero podemos cambiarlos a HEAD, POST, PUT o cualquier otro tipo de método. 2º) podemos indicar objetos para enviar en el body de la petición, así como modificar las cabeceras en el campo headers:

```
const options = {
        method: "POST",
        headers: {
                "Content-Type": "application/json"
        },
        body: JSON.stringify(jsonData)
};
```

Enviamos una petición POST, indicando en la cabecera que se envía contenido JSON y en el cuerpo de los datos, enviando el objeto jsonData, codificado como texto mediante stringify().

Fetch: Peticiones asíncronas

Diferencia entre métodos GET y POST

La diferencia entre ambos es la forma del envío de datos

- **GET** → los parámetros se concatenan a la URL accedida
  - Suele utilizarse para peticiones de lectura
  - Se utiliza cuando se accede a un recurso que depende de la información proporcionada por el usuario
  - Tiene límite en la cantidad de datos a enviar (512 bytes)

  Los parámetros se envían como una serie de pares clave/valor concatenados por símbolos &

```
https://localhost/aplication.php? parametro1=valor&parametro2=valor
```

- **POST** → los datos se envían de forma que no pueden verse (en un segundo plano u "ocultos" al usuario)
  - Se utiliza en operaciones que crean, borran o actualizan información

  La cadena que contiene los parámetros se debe construir manualmente

```
xmlhttp.setRequestHeader("Content-type", "application/json");
xmlhttp.send(cadenaParams);
```

Fetch: Peticiones asíncronas

La respuesta Response

En el ejemplo anterior, en el primer .then() tenemos un **objeto response**. Se trata de la respuesta que nos llega del servidor web al momento de recibir nuestra petición:

```
fetch("/robots.txt", options)
        .then(response => response.text())
        ……
```

El objeto response tiene una serie de propiedades y métodos que pueden resultarnos útiles al implementar nuestro código.

Por el lado de las **propiedades**, tenemos las siguientes:

Propiedad	Descripción
.status	Código de error HTTP de la respuesta (100-599).
.statusText	Texto representativo del código de error HTTP anterior.
.ok	Devuelve true si el código HTTP es 200 (o empieza por 2).
.headers	Cabeceras de la respuesta.
.url	URL de la petición HTTP.

Fetch: Peticiones asíncronas

Con la propiedad .ok tenemos una forma práctica y sencilla de comprobar si todo ha ido bien al realizar la petición:

```
fetch("/robots.txt")
        .then(response => {
                if (response.ok) return response.text()
                })
```

Fetch: Peticiones asíncronas

La instancia response también tiene **métodos**, la mayoría de ellos para procesar mediante una promesa los datos recibidos y facilitar el trabajo con ellos:

Método	Descripción
.text()	Devuelve una promesa con el texto plano de la respuesta.
.json()	Idem, pero con un objeto json. Equivale a usar JSON.parse().
.blob()	Idem, pero con un objeto Blob (binary large object).
.arrayBuffer()	Idem, pero con un objeto ArrayBuffer (buffer binario puro).
.formData()	Idem, pero con un objeto FormData (datos de formulario).
.clone()	Crea y devuelve un clon de la instancia en cuestión.
Response.error()	Devuelve un nuevo objeto Response con un error de red asociado.
Response.redirect(url, code)	Redirige a una url, opcionalmente con un code de error.

En el ejemplo anterior, response.text() indica que queremos procesar la respuesta como datos textuales, por lo que dicho método devolverá una PROMISE con los datos en texto plano, facilitando trabajar con ellos de forma manual

Fetch: Peticiones asíncronas

Ampliamos el ejemplo anterior, añadiendo un segundo .then:

```
fetch("/robots.txt")
        .then(response => response.text())
        .then(data => console.log(data));
```

Tras procesar la respuesta, con response.text(), devolvemos una PROMISE con el contenido en texto plano. Esta PROMISE se procesa en el segundo .then(), donde gestionamos dicho contenido almacenado en data.

¿Por qué dos .then()?
Se utilizan dos o más .then() en una cadena de promesas para manejar diferentes etapas de una operación asíncrona. En el caso de fetch, estas etapas suelen ser:

1.Obtener la respuesta: El primer .then() se utiliza para interceptar la respuesta del servidor. Esta respuesta contiene información sobre el estado de la petición (si fue exitosa, si hubo un error, etc.) y los datos que se han recibido.

2.Procesar los datos: El segundo .then() se utiliza para procesar los datos de la respuesta. Esto puede incluir convertir la respuesta a un formato más usable, extraer información específica, o realizar cualquier otra operación necesaria con los datos.

Fetch: Peticiones asíncronas

¿Por qué no un solo .then()?

- **Separación de responsabilidades:** Separar la obtención de la respuesta de su procesamiento hace el código más legible y mantenible.
- **Manejo de errores:** Cada .then() puede manejar diferentes tipos de errores. El primer .then() puede manejar errores de red, mientras que el segundo puede manejar errores en el procesamiento de los datos.
- **Encadenamiento:** Los .then() permiten encadenar múltiples operaciones asíncronas de forma secuencial.

Fetch: Peticiones asíncronas

Como vemos en la tabla anterior, tenemos varios métodos similares para procesar las respuestas. Por ejemplo, el caso anterior utilizando el método response.json() en lugar de response.text() sería equivalente al siguiente fragmento:

```
fetch("/contents.json")
        .then(response => response.text())
        .then(data => {
                const json = JSON.parse(data);
                console.log(json);
        });
```

Con response.json() nos ahorraríamos tener que hacer el JSON.parse() de forma manual, por lo que el código es algo más directo.

Fetch: Peticiones asíncronas

Ejemplo completo:

```
fetch('https://api.example.com/data')
    .then(response => {
        if (!response.ok) {
            throw new Error('Ha habido un error de envío en la red');
        }
        return response.json(); // Convertimos la respuesta a JSON
    })
    .then(data => { // Procesamos los datos console.log(data); })
    .catch(error => { console.error('Ha habido un problema con la operación Fetch:', error); });
```

- **Primer .then():**
  - Verifica si la respuesta del servidor es exitosa.
  - Convierte la respuesta a formato JSON .
  - Devuelve la promesa resuelta con los datos JSON.
- **Segundo .then():**
  - Recibe los datos JSON y los procesa, en este caso, simplemente los imprime en la consola.
- **. catch():** se ejecutará en los siguientes casos:
  - **Si hay un error de red:** La promesa será rechazada directamente y el catch se ejecutará con un objeto error que contiene información sobre el error de red.
  - **Si la respuesta no es exitosa:** El primer then lanzará un error manualmente si response.ok es false, lo que hará que el catch se ejecute.
    - response.ok se evaluará como falso en los casos de errores HTTP como 404 Not Found y 500 Internal Server Error, entre otros. Así:
      - Los códigos que comienzan con 4 (como 404) indican errores del cliente (por ejemplo, la página no se encontró), y los que comienzan con 5 (como 500) indican errores del servidor.
      - Si el código está fuera del rango de 200-299 (que indica éxito), ok será false.
  - **Si hay un error al parsear el JSON:** Si response.json() lanza un error, la promesa será rechazada y el catch se ejecutará.

Fetch: Método then

Código no bloqueante

El código que hemos ejecutado en los ejemplos anteriores es un **código asíncrono no bloqueante**.

- **Asíncrono**: Porque generalmente no se ejecuta de inmediato, sino que tardará en completarse.
- **No bloqueante**: Porque mientras espera ser ejecutado, no bloquea y continua el resto del programa.

Observa el siguiente ejemplo:

```
fetch("/robots.txt")
    .then(response => { console.log("Código asíncrono"); });
    console.log("Código síncrono");
```

Aunque el console.log("Código asíncrono") figura unas líneas antes del console.log("Código síncrono"), no se garantiza que se muestre antes, de hecho, normalmente aparecerá después.
Las promesas en JavaScript, tanto gestionadas con then() como con async/await son no bloqueantes. Sin embargo, con async/await, la ejecución dentro de la función async se "pausa" en los puntos donde se utiliza await, aunque el resto del programa sigue siendo no bloqueante.

Api Fetch. EJEMPLO 0

FETCH: Leer el contenido de un fichero de texto

```
<html>
<head>
   <title>Fetch sin Servidor</title>
</head>
<body>
   <div>
      <button id="pulsar">pulsar</button>
      <p id="contenido"></p>
      <img id="imagen"></img>
   </div>
   <script src="Fetch1.js"></script>
</body>
</html>
```

```
const boton = document.getElementById('pulsar');
boton.addEventListener('click',()=>{
// EJEMPLO FECH -0-
   //Leer archivo de texto
   const contenido=document.getElementById('contenido');
   fetch('Texto.txt')
      .then(response => response.text()) //transforma los datos recibidos al formato text
      .then(response =>{
         console.log(response);
         contenido.innerHTML = `${response}`;
      })
      .catch(err => {
         console.log(err);
   })
});
```

Api Fetch. EJEMPLO 1

FETCH CON GET. Buscar datos en una base de datos

```
const dni = '12345678'; // Reemplaza con el DNI que quieras buscar
fetch(`buscar_empleado.php?dni=${dni}`)
        .then(response => {
                if (!response.ok) {
                        throw new Error('La solicitud no fue exitosa');
                }
                 return response.json();
        })
        .then(data => {
                console.log('Información del empleado:', data);
        })
        .catch(error => {
                console.log('Error en la solicitud:', error);
        });
```

Cuando se ejecuta la declaración throw, se interrumpe inmediatamente la ejecución del bloque de código actual y se pasa el control al primer bloque catch.
Al lanzar una excepción con throw new Error, estás indicando que algo salió mal y permites que el código que maneja las Promesas (a través del bloque .catch en este caso) maneje ese error.

Api Fetch. EJEMPLO 1

```
<?php
// Incluir conexión a la base de datos
include_once “conectar.php";
// Obtener el DNI del parámetro de la solicitud GET
$dni = $_GET['dni'];
// Consulta SQL para buscar empleados por DNI
$sql = "SELECT * FROM empleados WHERE dni = '$dni'";
$result = $conn->query($sql);
// Verificar si se encontraron resultados
if ($result->num_rows > 0) {
          // Convertir resultados a un array asociativo
          $row = $result->fetch_assoc();
          // Devolver los resultados como JSON
          echo json_encode($row);
          } else {
          // Devolver un mensaje si no se encontraron resultados
          echo json_encode(['mensaje' => 'No se encontró ningún empleado con ese
DNI']); }
          // Cerrar la conexión a la base de datos
$conn->close(); ?>
```

Api Fetch. EJEMPLO 3

FETCH CON POST y json. Buscar datos en una base de datos

```
var dni = document.getElementById('dni‘).value;
// Crear un objeto JSON con el DNI
var datosEmpleado = { dni: dni };
// Reemplaza con el DNI que quieras buscar
const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json', },
        body: JSON.stringify(datosEmpleado), // Enviar el DNI como JSON en el cuerpo de la solicitud
        };
fetch('buscar_empleado.php', requestOptions)
        .then(response => {
                if (!response.ok) {
                        throw new Error('La solicitud no fue exitosa');
                }
                return response.json();///OJO!! JSON.parse(response) no funcionaría, ya que no genera una nueva
        })                                                                                    promesa
        .then(data => {
                if (data.mensaje){
                        console.log(data.mensaje);
                }else{
                console.log(‘Nombre del empleado:', data.Nombre);
                }
        })
        .catch(error => { console.log('Error en la solicitud:', error); });
```

Se envía la información al servidor en formato json, y se recibe en el mismo formato

Api Fetch. EJEMPLO 3

```php
<?php
// Incluir conexión a la base de datos
include_once “conectar.php";
// Obtener el DNI del cuerpo de la solicitud POST
$data = json_decode(file_get_contents('php://input'), true);
$dni = $data['dni'];
// Consulta SQL para buscar empleados por DNI
$sql = "SELECT * FROM empleados WHERE dni = '$dni'";
$result = $conn->query($sql);
// Verificar si se encontraron resultados
if ($result->num_rows > 0) {
                    // Convertir resultados a un array asociativo
                    $row = $result->fetch_assoc(); // Devolver los resultados como JSON
                    echo json_encode($row);
          } else {
                    // Devolver un mensaje si no se encontraron resultados
                    echo json_encode(['mensaje' => 'No se encontró ningún empleado con ese DNI']);
          }
// Cerrar la conexión a la base de datos
$conn->close();
?>
```

- **file_get_contents():** se utiliza para leer el contenido de un archivo o de un flujo de datos.
- **'php://input':** Este es un flujo especial que permite leer el cuerpo de una petición HTTP. En este caso, estamos leyendo los datos enviados por el cliente a través de la petición fetch.

json_decode(): toma una cadena JSON y la convierte en un valor de datos PHP.

- **Primer parámetro:** La cadena JSON a decodificar
- **Segundo parámetro (true):** Indica que se desea obtener un array asociativo como resultado. Si se omite o se establece en false, se obtendrá un objeto.

Api Fetch. EJEMPLO 2 a)

FETCH CON POST y www.form.urlencoded.

El Content-Type application/x-www-urlencoded es un formato estándar utilizado para codificar datos enviados en solicitudes HTTP, especialmente en formularios HTML.

Esta codificación funciona así:

- **Pares clave-valor:** Los datos se representan como pares clave-valor, donde la clave es el nombre del campo y el valor es el dato introducido por el usuario.
- **Separadores:** Los pares clave-valor se separan entre sí por el carácter & (ampersand) y los nombres de los campos se separan de sus valores por el carácter = (igual).
- **Codificación de caracteres especiales:** Los caracteres especiales, como espacios, signos de puntuación y caracteres no alfanuméricos, se codifican utilizando el formato URL encoding. Por ejemplo, un espacio en blanco se codifica como %20.

Api Fetch. EJEMPLO 2 a)

```
const requestOptions = {

   method: 'POST',
   headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
   },
   body: `dni=1111&nombre=Juan&apellido=Garcia&edad=35`
};

fetch('buscar_empleado2.php', requestOptions)
   .then(response => {
      if (!response.ok) {
         throw new Error('La solicitud no fue exitosa');
      }
      return response.json();
   })
   .then(data => { console.log(data.mensaje,' Dni: ',data.dni,' Nombre: ', data.nombre, ' Apellido: ',data.apellido,' edad: ',data.edad); })
   .catch(error => {
      console.log('Error en la solicitud:', error);
   });
```

Api Fetch. EJEMPLO 2 a)

```
<?php
// Incluir conexión a la base de datos
include_once "conectar.php";
// Obtener los datos enviados en la petición
$dni = $_POST['dni'];
$nombre = $_POST['nombre'];
$apellido = $_POST['apellido'];
$edad = $_POST['edad'];
// Crear una respuesta JSON
$respuesta = ['mensaje' => 'Datos recibidos correctamente','dni'=>$dni,'nombre'=>$nombre,'apellido'=>$apellido,'edad'=>$edad ];
echo json_encode($respuesta);
?>
```

Api Fetch. EJEMPLO 2 b)

FETCH CON POST y www.form.urlencoded

```
const datos = {
   dni:'1111',
   nombre: 'Juan',
   apellido:'García',
   edad: 30
};
// Convertimos los datos a formato x-www-form-urlencoded
const formulario = new URLSearchParams(datos);
const requestOptions = {
   method: 'POST',
   headers: { 'Content-Type': 'application/x-www-form-urlencoded', },
   body: formulario.toString(), // Datos a enviar en el cuerpo de la solicitud
   };
fetch('buscar_empleado2.php', requestOptions)
   .then(response => {
      if (!response.ok) {
         throw new Error('La solicitud no fue exitosa');
      }
      return response.json();
   })
   .then(data => { console.log(data.mensaje,' Dni: ',data.dni,' Nombre: ', data.nombre, ' Apellido: ',data.apellido,' edad: ',data.edad); })
   .catch(error => {
      console.log('Error en la solicitud:', error);
   });
```

Este es otro modo de formar la cadena anterior nombre=Juan¬edad=edad=30, con el objeto URLSearchParams

El fichero php sería igual al del Ejemplo 2 a)

Api Fetch. EJEMPLO 2 b)

```php
<?php
// Incluir conexión a la base de datos
include_once "conectar.php";
// Obtener los datos enviados en la petición
$dni = $_POST['dni'];
$nombre = $_POST['nombre'];
$apellido = $_POST['apellido'];
$edad = $_POST['edad'];
// Crear una respuesta JSON
$respuesta = ['mensaje' => 'Datos recibidos correctamente','dni'=>$dni,'nombre'=>$nombre,'apellido'=>$apellido,'edad'=>$edad ];
echo json_encode($respuesta);
?>
```

Api Fetch. EJEMPLO 4

FETCH CON POST y FormData

Este es alternativo al modo www-form-urlencoded, pero para formularios más complejos

```
function enviarDatos() {
   const formulario = document.getElementById('myForm');
   const datos = new FormData(formulario);

const requestOptions = {
                method: 'POST',
                headers: {'Content-Type': 'multipart/form-data'},
                body: datos, // Se envia el objeto FormData
     };
fetch('buscar_empleado2.php', requestOptions)
   .then(response => {
      if (!response.ok) {
         throw new Error('La solicitud no fue exitosa');
      }
      return response.json();
   })
   .then(data => { console.log(data.mensaje,' Nombre: ', data.nombre, ' email: ',data.email); })
   .catch(error => {
      console.log('Error en la solicitud:', error);
   });
}
```

```
<form id="myForm" method='POST' enctype="multipart/form-data">
      <input type="text" name="nombre" placeholder="Nombre">
      <input type="email" name="email" placeholder="Correo electrónico">
      <button type="button" onclick="enviarDatos()">Enviar</button>
   </form>
```

OJO!! Podría omitirse la cabecera form-data, porque el navegador entendería el tipo da datos que está enviando. Pero es conveniente indicarlo

Api Fetch. EJEMPLO 4

```php
<?php
// Incluir conexión a la base de datos
include_once "conectar.php";
// Recibir los datos enviados por POST
$nombre = $_POST['nombre'];
$email = $_POST['email'];

// Crear una respuesta JSON
$respuesta = ['mensaje' => 'Datos recibidos correctamente','nombre'=>$nombre,'email'=>$email];
// Establecer la cabecera HTTP. header('Content-Type: application/json');
echo json_encode($respuesta);
?>
```

Si no hubiese especificado en la cabecera de js que era form-data no se hubiera que tenido que especificar en el php el envío de información json.
Pero como se ha indicado, es necesario especificar aquí con la cabecera que la información a enviar es json