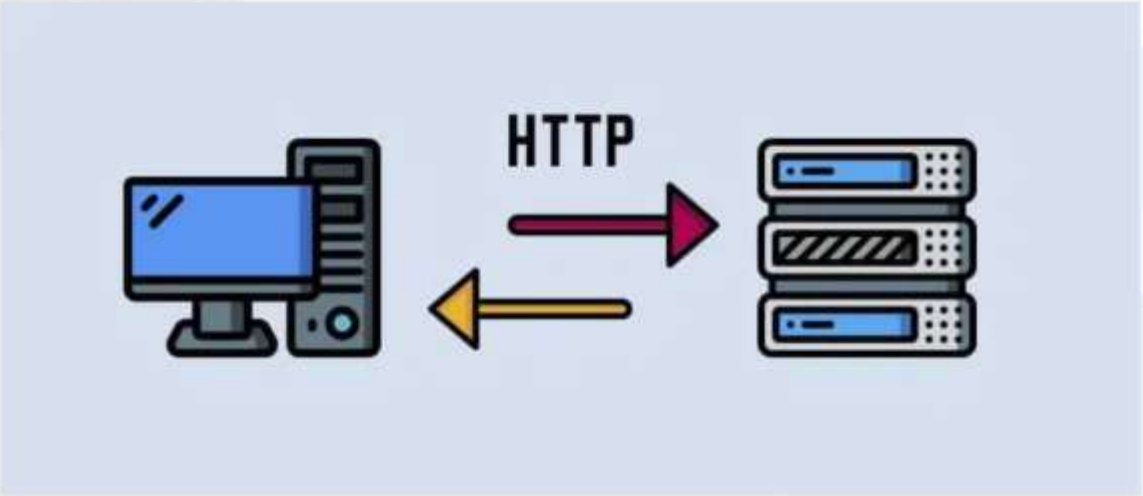
CAMPUS LANDS

Solicitudes HTTP en JavaScript

Peticiones HTTP mediante AJAX

AJAX, que significa **Asynchronous JavaScript and XML**, representa un enfoque en el que las peticiones HTTP se realizan desde JavaScript sin afectar directamente la experiencia del usuario.



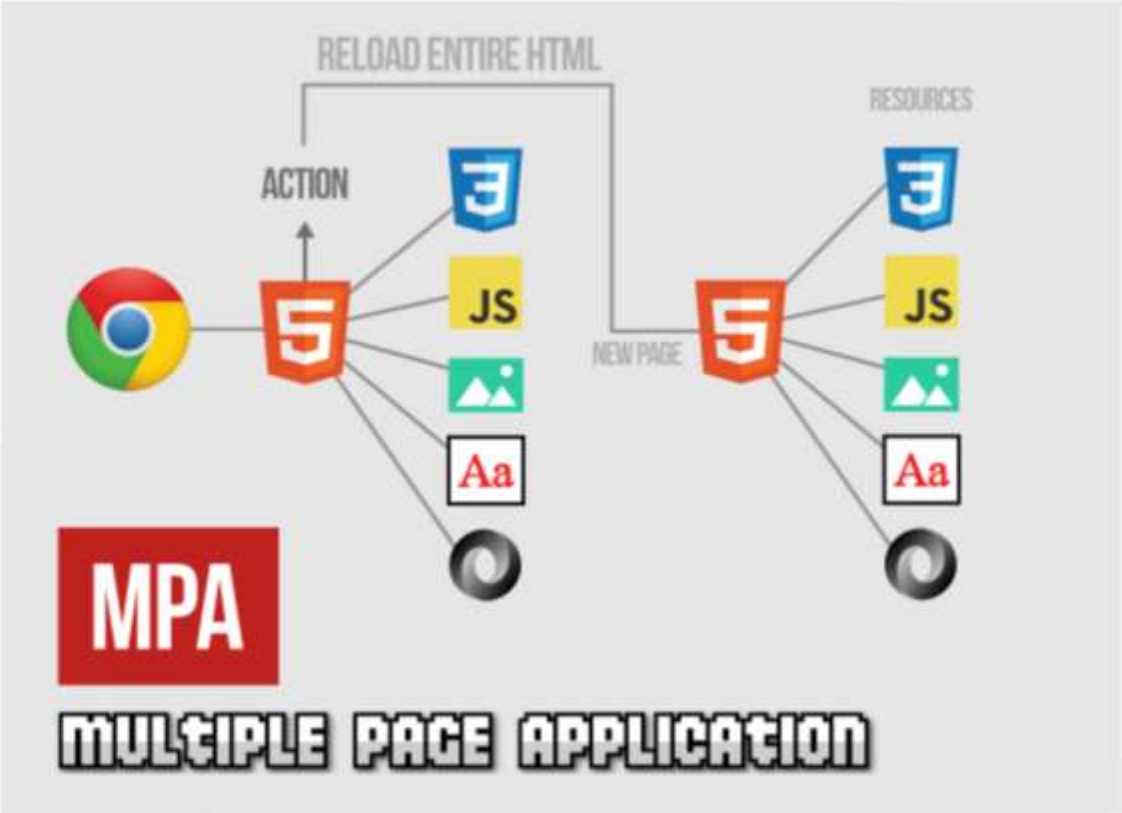
Peticiones HTTP mediante AJAX

Esta técnica permite descargar información de manera transparente, posibilitando el procesamiento y actualización parcial de una página en tiempo real. Con AJAX, es factible realizar actualizaciones de contenido sin recargar toda la página, permitiendo cambios en partes específicas y aprovechando JavaScript para implementar lógica adicional.

MPA: Multiple Page Application

La creación tradicional de páginas o aplicaciones web solía seguir el modelo de páginas MPA (Multiple Page Application). En este enfoque, el navegador descarga el archivo .html, lo lee y luego realiza solicitudes para los archivos adicionales relacionados que encuentra en el documento HTML. Cuando el usuario hace clic en un enlace, se descarga el .html correspondiente (recargando la página completa) y se repite el proceso.

MPA: Multiple Page Application

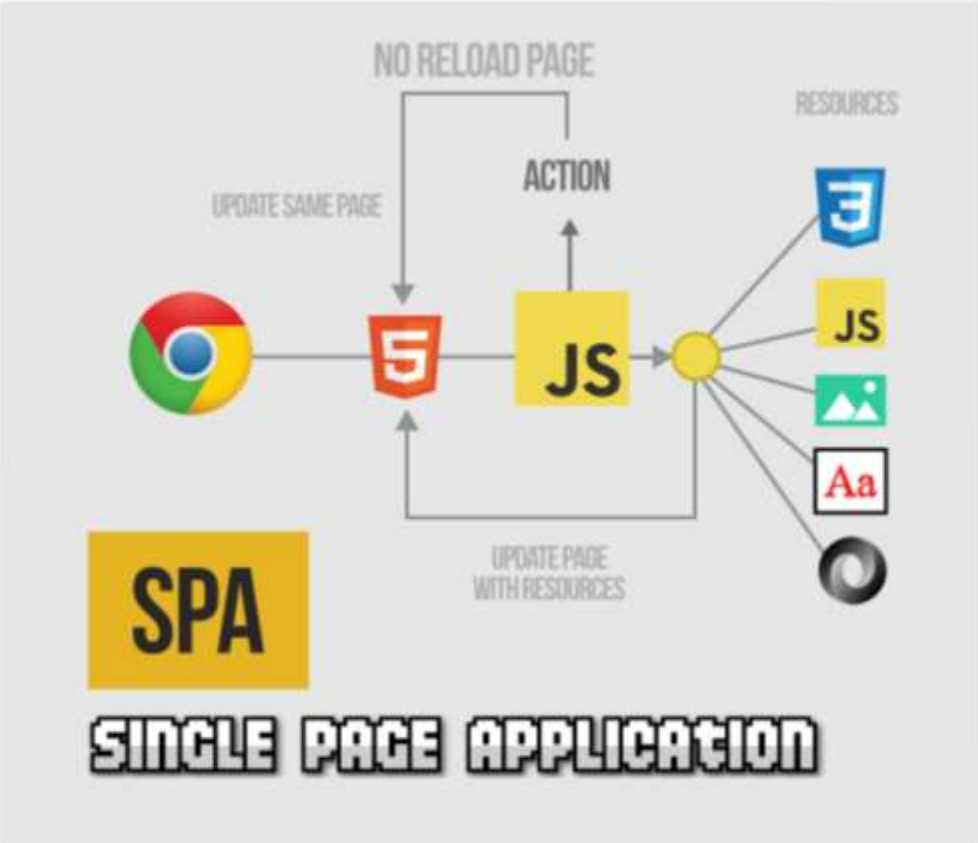


SPA: Single Page Application

Las páginas SPA (Single Page Application) representan un enfoque donde el navegador inicialmente descarga un archivo .html básico junto con un archivo .js encargado de gestionar toda la aplicación. A través de peticiones, obtiene archivos relacionados y datos adicionales (.json o .js), actualizando la página parcial o completamente sin necesidad de recargarla por completo.

Ejemplos de SPA incluyen la versión web de WhatsApp, Twitter o Google Drive.

SPA: Single Page Application



XHR: XMLHttpRequest

XMLHttpRequest (XHR) es un objeto especial de JavaScript que se creó originalmente para realizar peticiones ***HTTP a ficheros .xml*** externos, aunque actualmente se utiliza comúnmente para operaciones con archivos JSON, siendo estos más habituales en el ecosistema JavaScript como almacenamiento ligero de datos.

XHR: XMLHttpRequest

Este mecanismo de peticiones HTTP con XMLHttpRequest es relativamente sencillo en su enfoque principal, ***aunque puede volverse más complejo*** a medida que se incorporan comprobaciones y detalles adicionales.

Es importante tener en cuenta que, a diferencia de su equivalente moderno, **fetch**, **XMLHttpRequest** opera a un nivel más bajo. A continuación veamos un ejemplo de una petición con XMLHttpRequest.

XHR: XMLHttpRequest

```
var xhr = new XMLHttpRequest();
```

Configurar la solicitud:

Puedes configurar la solicitud mediante métodos como open y setRequestHeader . Por ejemplo, para realizar una solicitud GET a una URL específica, podrías hacer lo siguiente:

```
xhr.open("GET", "https://api.ejemplo.com/data", true);
xhr.setRequestHeader("Content-Type", "application/json");
```

Definir el manejo de eventos:

Puedes especificar funciones que se ejecutarán cuando haya cambios en el estado de la solicitud. Por ejemplo, para manejar la respuesta del servidor, podrías hacer lo siguiente:

```
xhr.onload = function () {
  if (xhr.status >= 200 && xhr.status < 300) {
    // La solicitud fue exitosa
    console.log(xhr.responseText);
  } else {
    // La solicitud falló
    console.error("Error en la solicitud:", xhr.statusText);
  }
};
```

Enviar la solicitud:

```
xhr.send();
```

Este ejemplo muestra cómo utilizar XMLHttpRequest para realizar una solicitud GET a una API. Ten en cuenta que este es un **enfoque antiguo** y, en la actualidad, se recomienda el uso de la **API fetch**, que proporciona una interfaz más moderna y basada en promesas para realizar solicitudes HTTP.

Otro ejemplo

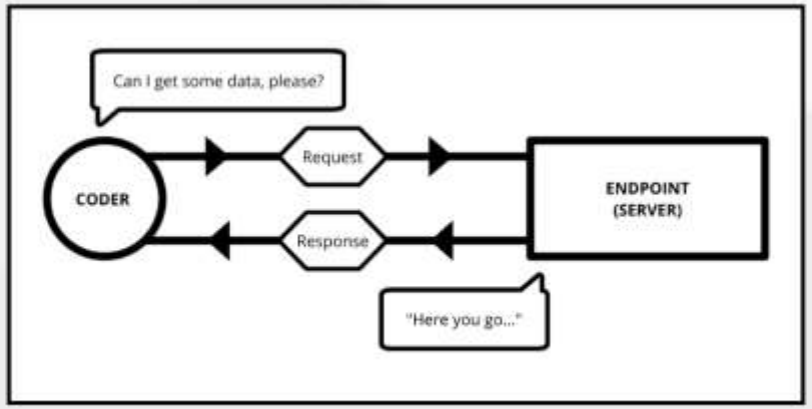
```
// funcion para cuando la llamada es exitosa
function exito() {
  var datos = JSON.parse(this.responseText); //convertir a JSON
  console.log(datos);
}

// funcion para la llamada fallida
function error(err) {
  console.log("Solicitud fallida", err); //los detalles en el objecto "err"
}

var xhr = new XMLHttpRequest(); //invocar nueva instancia de XMLHttpRequest
xhr.onload = exito; // llamar a la funcion exito si exitosa
xhr.onerror = error; // llamar a la funcion error si fallida
xhr.open("GET", "https://api.github.com/users/manishmshiva"); // Abrir solicitud GET
xhr.send(); // mandar la solicitud al vervidor.
```

Fetch: Peticiones Asíncronas

Fetch: Peticiones Asíncronas

Fetch es una novedosa API de Javascript que facilita las solicitudes HTTP asíncronas mediante el uso de promesas, simplificando el código y haciéndolo menos verboso.

La realización de una petición se simplifica al llamar simplemente a "fetch" y proporcionar la URL como parámetro.



Fetch: Peticiones Asíncronas

Ejemplo:

```
const promise = fetch("/robots.txt");

promise.then(function (response) {
  /* ... */
});
```

La función **fetch()** devuelve una promesa que se resolverá al recibir una respuesta y solo se rechazará en caso de un fallo de red o si la petición no se pudo completar.

Fetch: Peticiones Asíncronas

Una alternativa más concisa y clara implica reescribir el código, evitando el uso de constantes o variables temporales de un solo uso.

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

Ejemplo:

```
let fetchRes = fetch("https://jsonplaceholder.typicode.com/todos/1");

// leer el id 1
fetchRes
  .then((res) => res.json())
  .then((d) => {
    console.log(d);
  });
```

Opciones de fetch()

A la función fetch(), al margen de la url a la que hacemos petición, se le puede pasar un segundo parámetro de opciones de forma opcional, un con opciones de la petición HTTP.👀

```
const options = {
  method: "GET",
};

fetch("/robots.txt", options)
  .then((response) => response.text())
  .then((data) => {
    /** Procesar los datos **/
  });
```

Opciones de fetch()

En este objeto 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.

Opciones de fetch()

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

Se efectúa una solicitud POST con la indicación de contenido JSON en la cabecera.

- En el cuerpo de la solicitud, enviamos el objeto "jsonData", credenciales.

Ejemplo petición GET con opciones

```
// Opciones
const options = {
  method: "GET",
  headers: { "Content-type": "application/json;charset=UTF-8" },
};

fetch("https://jsonplaceholder.typicode.com/todos", options)
  .then((res) => res.json())
  .then((d) => {
    console.table(d);
  });
```

Respuesta (Response)

La respuesta (Response) de `fetch` es un objeto que representa la respuesta a una solicitud HTTP realizada mediante la función `fetch` en JavaScript. Este objeto **contiene información sobre el estado de la respuesta**, como el *código de estado*, las *cabeceras* y el *cuerpo de la respuesta*. Permite a los desarrolladores interactuar y trabajar con la respuesta recibida del servidor web.

Respuesta (Response)

```
fetch("/robots.txt", options)
  .then((response) => response.text())
  .then((data) => {
    /** Procesar los datos **/
  });
```

En este ejemplo, utilizamos una **arrow function** que realiza un retorno implícito de la promesa proporcionada por el método .**text()**. Sin embargo, el objeto **response** cuenta con **propiedades** y **métodos** adicionales que pueden ser útiles al desarrollar nuestro código.

Tabla con algunas propiedades del Response

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.

Respuesta (Response)

Ejemplo:

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

Ejemplo de propieades del Response

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then((res) => {
    const opciones = {
      status: res.status,
      statusText: res.statusText,
      ok: res.ok,
      url: res.url,
      type: res.type,
      headers: res.headers,
    };
    console.table(opciones);

    if (res.ok) {
      return res.json();
    }
  })
  .then((d) => {
    console.log("\nDatos de la respuesta: ");
    console.table(d);
  });
```

Procesar la respuesta

response cuenta con varios métodos útiles, para procesar los datos recibidos mediante el uso de promesas y simplificar su manipulación:

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).

Procesar la respuesta

Método	Descripción
.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.

Procesar la respuesta

Método	Descripción
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.

Procesar la respuesta

En el siguiente **fragmento de código**, después de procesar la respuesta con **response.text()**, retornamos una promesa con el contenido en texto plano. Esta promesa se gestiona en el segundo **.then()**, donde trabajamos con el contenido almacenado en la variable **"data"**.

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

Procesar la respuesta

- 👀 **Ejemplo resonse.json**

```
fetch("/contents.json")
  .then((response) => response.json())
  .then((data) => console.log(json));
```

Fetch usando .then()

En el siguiente ejemplo, empleamos `fetch` utilizando `.then`, presentando un resumen integral de los conceptos explorados hasta ahora con `fetch`. Además, se agrega un bloque `try/catch` para gestionar posibles excepciones.

```
fetch("/robots.txt")
  .then((response) => {
    if (response.ok) return response.text();
    throw new Error(response.status);
  })
  .then((data) => {
    console.log("Datos: " + data);
  })
  .catch((err) => {
```

Fetch usando .then()

👀 **Ejemplos refactorizado**

```
const isResponseOk = (response) => {
  if (!response.ok) throw new Error(response.status);
  return response.text();
};

fetch("/robots.txt")
  .then((response) => isResponseOk(response))
  .then((data) => console.log("Datos: ", data))
  .catch((err) => console.error("ERROR: ", err.message));
```

Fetch usando async/await

El uso de **async/await** es simplemente un "**azúcar sintáctico**", una forma visualmente más agradable de realizar tareas que, en el fondo, son equivalentes a las ejecutadas con **.then().** Es importante recordar que **await solo puede ejecutarse** dentro de una función definida **como async**.

Ejemplo fecth con asyn / await

```
const peticion = async (url) => {
  const response = await fetch(url);
  if (!response.ok) {
    const error = new Error(`[OCURRIO UN ERROR]: ${response.status}`);
    error.status = response.status; // Agregar el status al objeto Error
    throw error;
  }
  const data = await response.text();
  return data;
};
const main = async () => {
  const resultOk = await peticion("datos.csv");
  console.log("[OK]\n", resultOk);

  const resultError = await peticion("archivoNoExiste.txt");
};

main();
```

Ejemplo 2 de Fetch con async y await

```
const fetchData = async () => {
  const res = await fetch("https://jsonplaceholder.typicode.com/todos/1");
  if (res.ok) {
    const data = await res.json();
    console.log("\nDatos de la respuesta: ");
    console.table(data);
  }
};

fetchData();
```

Ejercicios

- Realizar una página que dibuje los personajes de la serie animada Ricky and Morty
- Realizar una página que de conversión de Euros a Dolares.

Recursos

- Tutorial de solicitud AJAX con Javascript