UT 10 – SOLICITUDES DE RED (RESUMEN)

Contenido

1. JSON[1]

JSON es un estándar que permite representar valores y objetos en una cadena. Este estándar es reconocido por muchos lenguajes (JavaScript, PHP, Ruby, Java…).

Se utiliza para:

- Intercambiar datos entre distintos lenguajes si estos lenguajes reconocen el mismo formato JSON. Un caso en el que se utiliza es para intercambiar datos con el servidor.
- Almacenar datos en formato cadena. En algunos casos, sólo se pueden almacenar cadenas. Utilizando el formato JSON podríamos almacenar cualquier valor.

Qué valores se pueden representar en cadenas JSON:

- Un valor primitivo: número, boolean, cadena
- Un objeto
- Un array
- El valor `null`

Estos valores pueden estar anidados (por ejemplo, un array cuyos elementos son objetos).

Los siguientes valores **no** se pueden serializar:

- Una función, un método
- El valor undefined
- El tipo de datos SYMBOL

Ejemplo: el siguiente valor

```
let student = {  name: 'John',
                 age: 30,
                 isAdmin: false,
                 courses: ['html', 'css', 'js'],
                 wife: null
               };
```

Se puede representar con el siguiente formato JSON:

```
{  "name": "John",
   "age": 30,
   "isAdmin": false,
   "courses": ["html", "css", "js"],
   "wife": null
}
```

Vemos que:

- Las propiedades y las cadenas se escriben entre dobles comillas.
- Los valores numéricos, booleanos y null se escriben directamente, sin comillas.
- Los arrays se representan entre corchetes [ ] *(como en JavaScript)*.
- Los objetos se representan entre llaves { } *(como en JavaScript)*

[1] Más detallado en: https://es.javascript.info/json

Obtener la cadena JSON a partir de un valor (serializar)

```
cadena = JSON.stringyfy( valor )
```

Este método devuelve una cadena que representa al `valor` en formato JSON.

Ejemplo:

```
let student = {  name: 'John',
                 age: 30,
                 isAdmin: false,
                 courses: ['html', 'css', 'js'],
                 wife: null
              };

let cadenaJSON = JSON.stringify(student);

alert(typeof cadenaJSON); // ¡obtenemos un string!

alert(cadenJSON);
/*{  "name": "John",
   "age": 30,
   "isAdmin": false,
   "courses": ["html", "css", "js"],
   "wife": null
}
*/
```

Obtener un valor a partir de la cadena JSON

```
valor = JSON.parse( cadena )
```

Este método obtiene un dato a partir de una cadena que representa un `valor` en formato JSON.

Ejemplo: Seguimos con el ejemplo anterior.

```
estudiante2 = JSON.parse(cadenaJSON);
alert( estudiante2.name);              // John
```

Cuidado con las fechas

Ojo! Se puede serializar una fecha, pero para recuperar su valor, tendremos que crear un objeto Date a partir de la cadena que devuelve JSON.

Ejemplo: En este ejemplo, creamos un objeto de tipo fecha (`hoy`). Lo serializamos, y luego intentamos recuperar su valor. En `fecha1` recuperamos una cadena, no una fecha. Sin embargo, en `fecha2` vemos que se ha convertido la fecha en una cadena.

```
let hoy = new Date()
hoy_JSON = JSON.stringify(hoy) // "\"2022-02-01T08:40:35.826Z\""

let fecha1 = JSON.parse(hoy_JSON)             // String
let fecha2 = new Date(JSON.parse(hoy_JSON))   // Objeto Date
```

2. PROMESAS

Las promesas son objetos de JavaScript 6 (ES2015) que nos van a permitir gestionar operaciones asíncronas.

Las promesas nos van a servir para hacer llamadas a servidor y recibir datos del servidor sin necesidad de recargar la página. Van a comportarse como objetos intermediarios entre el código del cliente y el código del servidor.

En el siguiente resumen, sólo se verán algunas características de las promesas, para centrarnos en su utilización en las llamadas al servidor.

Crear una promesa[2]

Los objetos de tipo promesa se crean de este modo:

```
let promise = new Promise(function(resolve, reject) {
  // Ejecutor (el código productor, "cantante")
});
```

resolve y reject son dos funciones que pasa automáticamente JavaScript a la función. No hay que crearlas.

Dentro de la función está el código que se quiere ejecutar. Cuando el código obtiene un valor correcto, dentro de la función se llama a resolve(), pudiendo pasar un valor.

Si en la ejecución se quiere dar algún error, se llama a reject(), pudiendo pasar un valor como parámetro.

Ejemplo de promesa que termina correctamente: La promesa se ejecuta cuando se construye la promesa. Al cabo de 1 segundo, indica que la tarea está realizada, con el resultado "hecho".

```
let promise = new Promise(
     function(resolve, reject) {
          setTimeout(() => resolve("hecho"), 1000);
     }
);
```

Ejemplo de promesa que termina con error: Ejemplo similar al anterior, pero en este caso se quiere indicar que la tarea ha tenido un error, y se crea un objeto de tipo error con el texto "¡Vaya".

```
let promise = new Promise(
     function(resolve, reject) {
          setTimeout(() => reject(new Error("¡Vaya!")), 1000);
     }
);
```

El objeto promise tiene estas propiedades internas (no se puede acceder desde fuera de la promesa):

- state: puede tener los siguientes valores:
  - pending: pendiente
  - fulfilled: cumplido (cuando se llama a resolve)
  - rejected: rechazado (cuando se llama a reject)
- result: puede tener los siguientes valores:
  - undefined: valor inicial

[2] https://es.javascript.info/promise-basics

- *valor*: si se llama a resolve(*valor*)
- *error*: si se llama a rejected(*error*)

Tener en cuenta:

- La función que pasamos a la promesa se ejecuta automáticamente cuando se crea el objeto de tipo promesa.
- Sólo puede haber un único resultado o un error. Todas las llamadas adicionales a resolve() y reject() son ignoradas.
- resolve() y reject() sólo esperan un argumento (o ninguno) e ignorarán los argumentos adicionales.
- Cuando se llama a reject(), se recomienda usar un objeto de tipo Error.

Consumidores: then, catch, finally

Las promesas tienen unos métodos (.then(), .catch() y .finally().) donde se registran las funciones que se quieren ejecutar cuando la promesa pase al estado fulfilled (terminado correctamente), rejected (rechazado), o cualquiera de los dos.

.then(funciónOk, funciónError)

La sintaxis es:

```
promise.then(
  function(result) { /* manejar un resultado exitoso */ },
  function(error) { /* manejar un error */ }
);
```

El primer argumento es una función que se ejecuta cuando la promesa ha terminado con resolve(). result es el valor pasado a resolve().

El segundo argumento es una función que se ejecuta cuando la promesa ha terminado con reject().error es el valor pasado a resolve().

Ejemplo:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// reject ejecuta la segunda función en .then
promise.then(
  result => alert(result), // no se ejecuta
  error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo
);
```

.catch(funciónError)

Permite registrar una función que se ejecutará si se produce un error

```
promise.catch(
  function(error) { /* manejar un error */ }
);
```

La llamada .catch(f) es un análogo completo de .then(null, f), es solo una abreviatura.

.finally(función)

Función que se ejecuta siempre, se resuelva la promesa o se rechace.

1. Un manejador `finally` no tiene argumentos. En `finally` no sabemos si la promesa es exitosa o no. Eso está bien, ya que nuestra tarea generalmente es realizar procedimientos de finalización "generales".
2. Un manejador `finally` traspasa resultados y errores al siguiente manejador.

Encadenamiento de promesas

El método `promise.then` devuelve una nueva promesa.

Si la función (manejador) que se pasa al then:

- devuelve un valor: la promesa devuelta por el then, se considera resuelta con el valor devuelto como su valor (como si se hubiera hecho resolve(valor).
- no devuelve nada: la promesa devuelta por el then, se considera resuelta con el valor undefined.
- se produce un error: la promesa devuelta por el then, se considera rechazada con el error como su valor.
- devuelve una promesa resuelta: la promesa devuelta por el then se considera finalizada con el valor de la promesa como su valor.
- devuelve una promesa rechazada: la promesa devuelta por el then se considera rechazada con el valor de la promesa como su valor.
- devuelve una promesa pendiente de resolver: el manejador espera a que se estabilice y luego obtiene su resultado; la resolución o el rechazo de la promesa devuelta por el then será lo que corresponda a la resolución/rechazo de la promesa.

Ejemplo1: Encadenando promesas que devuelven valores.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

Ejemplo 2: Encadenando promesas que devuelven promesas.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

async/await

JavaScript 8 (ES2017) ha proporcionado una sintaxis especial para trabajar con promesas de una forma más confortable, llamada `"asyn/await"`.

Funciones async

La palabra reservada `async` se coloca delante de la declaración de una función. De este modo se indica que la función siempre devolverá una promesa. Otros valores serán envueltos y resueltos en una promesa automáticamente.

Ejemplo 1: Si la función devuelve un valor, ese valor se considera el valor con el que se resuelve la promesa.

```
async function f() {
  return 1;
}
f().then(alert);    // 1
```

Ejemplo 2: La función puede devolver una promesa. El siguiente ejemplo es equivalente al anterior.

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

await

Esta palabra reservada sólo puede utilizarse dentro de funciones async.

`await` hace que JavaScript espere a que una promesa responda y devuelve su resultado.

Ejemplo:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Hecho!"), 1000)
  });

  let result = await promise; // espera hasta que la promesa se resuelva (*)

  alert(result); // "¡Hecho!"
}

f();
```

La ejecución de la función es pausada en la línea (*) y se reanuda cuando la promesa se resuelve devolviendo un valor que se asigna a `result`. Ese valor se muestra con un `alert`.

Se podría haber hecho también con un `then`:

```
function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Hecho!"), 1000)
  });
```

```
  promise.then( function(result) {
       alert(result);
  }
}

f();
```

`async`/`await` nos proporcionan una sintaxis más sencilla para tener el resultado de una promesa que promise.`then`.

Manejo de errores con await

Si una promesa se resuelve, `await` `promise` devuelve su resultado. En caso de error se dispara un erro que puede ser capturado con un `try..catch`. En el bloque `try..catch` puede haber varios `await`.

Si no tenemos `try..catch`, entonces la promesa generada por `async f()` se vuelve rechazada.

Ejemplo: llamamos a `fetch` que devuelve una promesa con los datos para acceder al servidor, o devuelve un error si no se ha podido hacer la conexión con el servidor.

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

Ejemplo 2: en el bloque `try` podría haber varios `await`.

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // atrapa errores tanto en fetch como en response.json
    alert(err);
  }
}

f();
```

Ejemplo 3: Cómo gestionar los errores si no se utiliza el bloque `try..catch`.

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() se vuelve una promesa rechazada
f().catch(alert); // TypeError: failed to fetch // (*)
```

3. PETICIONES AL SERVIDOR CON PROMESAS: FETCH[3]

La función fetch

fetch() es un método global que inicia el proceso de obtener un recurso de la red, devolviendo una promesa que es resuelta cuando la respuesta está disponible.

La promesa se resuelve con un objeto de tipo Response que representa la respuesta a la petición.

La promesa se resuelve aunque haya errores HTTP (ej. 404). Por tanto, el manejador del then() debe comprobar Response.ok y/o Response.status.

La promesa sólo termina con un rechazo si se encuentra un error al acceder al servidor (ej. el servidor no existe, problemas de red...)

Sintaxis

```
let promise = fetch(url, [options])
```

donde:

- **`url`** – representa la dirección URL a la que deseamos acceder.
- **`options`** – objeto que contiene la configuración que se quiere aplicar a la conexión. Si se omite, se ejecuta una petición GET.

  Algunas de las propiedades que puede tener ese objeto son:

  - method: "GET", "POST", "PUT"; "DELETE", ..
  - headers: cabeceras que se quieren enviar.
  - body: un cuerpo que se quiere añadir a la petición. Sólo para peticiones POST.
  - mode: puede ser "cors", "no-cors" o "same-origin"
  - credentials: controla que tiene que hacer el navegador con las credenciales. Puede ser una de las siguientes cadenas:
    - "omit"
    - "same-origin"
    - "include"

Cómo hacer una petición al servidor

Obtener una respuesta es un proceso de dos pasos:

1. Llamada a **fetch()**. Fech devuelve una promesa que se resuelve con un objeto de la clase Response tan pronto como el servidor responde con los encabezados de la petición.
   Este objeto tiene dos valores que nos permiten conocer el resultado de la petición hecha al servidor:
   - **status**: Se puede chequear para comprobar el código de estado HTTP (ej. 200)
   - **ok**: true si el código de estado es 200 a 299

2. Obtener el cuerpo de la respuesta. Si no se ha producido error http, (response.ok==true), se puede llamar a un método de la respuesta, como:

[3] Más detallado en: https://es.javascript.info/fetch

- **`response.text()`** – devuelve una promesa que se resuelve con la representación en texto del cuerpo de la respuesta.
- **`response.json()`** – devuelve una promesa que se resuelve con el resultado de parsear el texto del cuerpo de la respuesta como un JSON.

IMPORTANTE

- Podemos elegir un solo método de lectura para el cuerpo de la respuesta.
- Si ya obtuvimos la respuesta con `response.text()`, entonces `response.json()` no funcionará, dado que el contenido del cuerpo ya ha sido procesado.

Petición GET que devolverá un JSON

Ejemplo: hacer una llamada a una url que devolverá un json;

```
let response = await fetch(url);
if (response.ok) { // si el HTTP-status es 200-299
  let json = await response.json();
} else {
  alert("Error-HTTP: " + response.status);
}
```

Petición GET enviando parámetros

Hay que agregar los parámetros a la URL como `?nombre=valor`, y asegurar la codificación adecuada. Podemos hacerlo manualmente (creando una cadena) o utilizando el objeto URL:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'pruébame!');

// el parámetro 'q' está codificado
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

Petición POST enviando un JSON

En este apartado vemos un ejemplo de cómo enviar datos de tipo JSON.

Ejemplo: enviar información al servidor como un objeto JSON.

```
let user = {
  nombre: 'Juan',
  apellido: 'Perez'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Recuperar los datos en el servidor

```
<?php
$json = file_get_contents('php://input');
$data = json_decode($json);

$nombre = $data->nombre;
$apellido = $data->apellido;

echo "El usuario " . $nombre." ".$apellido . " ha sido registrado." ;
?>
```

Petición POST enviando datos con FormData[4]

Los objetos FormData pueden facilitar la tarea de enviar datos al servidor.

Crear un FormData

```
let formData = new FormData([form]);
```

Si le pasamos un elemento HTML de tipo formulario, el objeto automáticamente capturará sus campos.

Envío de un formulario simple

Ejemplo 1: En el siguiente ejemplo vemos cómo se pueden enviar al servidor los datos de un formulario simple.

```
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('recibirUsuario.php', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

Cómo se pueden recuperar los datos en un programa php:

[4] https://es.javascript.info/formdata

```php
<?php
$nombre = $_POST["name"];
$apellido = $_POST["surname"];

echo $nombre." ".$apellido;
?>
```

Ejemplo 2: Los campos <input type="file"> también son enviados, como sucede en un envío normal.

Métodos de FormData

Existen métodos para modificar los campos del FormData:

- `formData.append(name, value)` – agrega un campo al formulario con el nombre name y el valor `value`,
- `formData.append(name, blob, fileName)` – agrega un campo tal como si se tratara de un `<input type="file">`, el tercer argumento fileName establece el nombre del archivo (no el nombre del campo), tal como si se tratara del nombre del archivo en el sistema de archivos del usuario,
- `formData.delete(name)` – elimina el campo de nombre `name`,
- `formData.get(name)` – obtiene el valor del campo con el nombre `name`,
- `formData.has(name)` – en caso de que exista el campo con el nombre `name`, devuelve `true`, de lo contrario `false`

Un formulario técnicamente tiene permitido contar con muchos campos con el mismo atributo `name`, por lo que múltiples llamadas a append agregarán más campos con el mismo nombre.

Por otra parte existe un método `set`, con la misma sintaxis que `append`. La diferencia está en que `.set` remueve todos los campos con el name que se le ha pasado, y luego agrega el nuevo campo. De este modo nos aseguramos de que exista solamente un campo con determinado `name`, el resto es tal como en `append`:

- `formData.set(name, value)`,
- `formData.set(name, blob, fileName)`.

También es posible iterar por los campos del objeto formData utilizando un bucle `for..of`:

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// Se listan los pares clave/valor
for(let [name, value] of formData) {
  alert(`${name} = ${value}`);                                    // key1 = value1,
luego key2 = value2
}
```

4. PETICIONES AL SERVIDOR CON XMLHttpRequest[5]

`XMLHttpRequest` es un objeto nativo del navegador que permite hacer solicitudes HTTP desde JavaScript.

A pesar de tener la palabra "XML" en su nombre, se puede operar sobre cualquier dato, no solo en formato `XML`. Podemos cargar/descargar archivos, dar seguimiento y mucho más.

Ahora hay un método más moderno `fetch` que en algún sentido hace obsoleto a `XMLHttpRequest`.

En el desarrollo web moderno `XMLHttpRequest` se usa por tres razones:

1. Razones históricas: necesitamos soportar scripts existentes con `XMLHttpRequest`.
2. Necesitamos soportar navegadores viejos, y no queremos polyfills (p.ej. para mantener los scripts pequeños).
3. Necesitamos hacer algo que `fetch` no puede todavía, ej. rastrear el progreso de subida.

¿Te suena familiar? Si es así, está bien, adelante con `XMLHttpRequest`. De otra forma, por favor, dirígete a `Fetch`.

Lo básico

Para hacer la petición, necesitamos seguir varios pasos:

1. Crear el objeto XMTHttpRequest

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

2. Inicializarlo, normalmente justo después de crearlo.

```
xhr.open(method, URL, [async, user, password])
```

siendo:

- `method`: Método HTTP. Normalmente, "GET" o "POST"
- `URL`
- `async`: si la petición será síncrona o no (por defecto es asíncrona, valor utilizado casi siempre)
- `user`, `password`: usuario y contraseña para atenticación HTTP básica

La llamada a `open`, no abre la conexión. Sólo configura la solicitud, pero la actividad de red sólo comienza con la llamada al método `send`.

3. Establecer el formato de respuesta esperada *(SÓLO EN ALGUNOS CASOS, por ejemplo, si la respuesta no es texto).*

```
xhr.responseType = "..."
```

Posibles tipos:

- `""` (default) – obtiene una cadena,
- `"text"` – obtiene una cadena,
- `"arraybuffer"` – obtiene un ArrayBuffer
- `"blob"` – obtiene un Blob

[5] https://es.javascript.info/xmlhttprequest

- `"document"` – obtiene un documento XML (puede usar XPath y otros métodos XML) o un documento HTML (en base al tipo MIME del dato recibido),
- `"json"` – obtiene un JSON (automáticamente analizado).

4. Enviar cabeceras *(SÓLO EN ALGUNOS CASOS, por ejemplo, si enviamos datos json).*

```
xhr.setRequestHeader(nombre, valor);
```

5. Enviar la petición.

```
xhr.send([body])
```

Este método abre la conexión y envía la solicitud al servidor. El parámetro adicional body contiene el cuerpo de la solicitud.

Algunos métodos como GET no tienen un cuerpo. Y otros como POST usan el parámetro body para enviar datos al servidor.

6. Escuchar los eventos de respuesta del objeto

Los eventos más comunes son:

- `xhr.onload = function()...` – cuando la solicitud está; completa (incluso si el estado HTTP es 400 o 500), y la respuesta se descargó por completo.
  Una vez el servidor haya respondido, podemos recibir el resultado en las siguientes propiedades de xhr:
  - **`xhr.status:`** Código del estado HTTP (un número): 200, 404, 403, etc., puede ser 0 en caso de un fallo no HTTP.
  - **`xhr.statusText:`** Mensaje del estado HTTP (una cadena): usualmente OK para 200, Not found para 404, Forbidden para 403.
  - **`xhr.response`** (scripts antiguos utilizan `responseText`)
- `xhr.onerror = function()...` – cuando la solicitud no pudo ser realizada satisfactoriamente, ej. red caída o una URL inválida.
- `xhr.onprogress = function()...` – se dispara periódicamente mientras la respuesta está siendo descargada, reporta cuánto se ha descargado.

Ejemplo1: obtener una respuesta de texto

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.send();

xhr.onload = function() {
   if (xhr.status != 200) {
       alert(`Error ${xhr.status}: ${xhr.statusText}`);
```

```
   } else {
       texto = xhr.response;
       alert(texto);
   }
}

xhr.onerror = function() { // solo se activa si la solicitud no se puede realizar
  alert(`Error de red`);
};
```

Métodos y propiedades obsoletas

En los scripts antiguos se pueden encontrar las propiedades xhr.responseText y xhr.responseXML. Existen por razones históricas, pero hoy en día, debemos seleccionar el formato en xhr.responseType y obtener la respuesta en xhr.response.

xhr.onreadystatechange se utilizaba cuando no existían onload y otros eventos. Hoy load/error/progress lo hacen obsoleto.

Abortar solicitudes

Se puede terminar la solicitud en cualquier momento.

```
xhr.abort();  // termina la solicitud
```

Peticiones POST

Peticiones POST con formularios: FormData

Se puede crear un objeto FormData desde un formulario o manualmente. Después se envía al servidor. No es necesario enviar cabeceras porque el tipo de datos enviado es `multipart/form-data`.

```
<form name="person">
  <input name="name" value="John">
  <input name="surname" value="Smith">
</form>

<script>
  // pre llenado del objeto FormData desde el formulario
  let formData = new FormData(document.forms.person);

  // agrega un campo más
  formData.append("middle", "Lee");

  // lo enviamos
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);

  xhr.onload = () => alert(xhr.response);
</script>
```

Peticiones POST con JSON

En este caso, sí que hay que enviar cabeceras.

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
  name: "John",
  surname: "Smith"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```