Desarrollador Frontend

clase 4 |  JSON

Javascript Object Notation - JSON

¿Qué es JSON?

- Es un acrónimo de Javascript Object Notation
- Algo así como Notación de objetos javascript
- Su implementación es parte del lenguaje javascript
- Nos permite utilizar una sintaxis específica para crear objetos javascript
- La mayoría de los lenguajes actuales dispone de bibliotecas para codificar / decodificar JSON, utilizando Strings para su serialización
- Es una alternativa más liviana en cuanto a procesamiento que XML para transferencia de datos entre cliente / servidor
- Actualmente motores de bases de datos admiten JSON como tipo a almacenar

¿Qué **no** es JSON?

- No es un lenguaje
- No es una biblioteca de funciones (como jQuery)

Para conocer más sobre lo que dice la comunidad sobre JSON podemos visitar su entrada oficial en wikipedia

JSON sintaxis básica

```
var numeros = ['uno', 'dos', 'tres'];

var clima = { dia: 'lunes', temp: 24, desc: 'despejado'};

var pronostico = [
    { dia: 'martes', min: 5, max: 24, desc: 'despejado'},
    { dia: 'miercoles', min: -3, max: 12, desc: 'nublado'},
    { dia: 'jueves', min: 4, max: 18, desc: 'nublado'}
  ];

var todo = {
  hoy: clima,
  pronostico: pronostico
};
```

Para definir un Array basta con utilizar los corchetes **[ ]** y separar con comas sus valores.

Para definir un objeto se utilizan las llaves **{ }**

Las propiedades de un objeto (par clave/valor) se definen separando la clave del valor con dos puntos **:**

Si existen varias propiedades se las separa con comas.

Puedo generar objetos complejos como Arrays de objetos (variable **pronostico** en el ejemplo)

Puedo generar propiedades que refieran a variables ya definidas (variable **todo** en el ejemplo)

El uso de saltos de línea y tabulaciones favorece la lectura de JSON, aunque en objetos de pocas propiedades (caso **numeros** o **clima** en el ejemplo) generalmente se utiliza una sola línea.

JSON vs javascript plano

```
var persona = new Object();
persona.nombre = "Roberto";
persona.apellido = "Sanchez";
persona.apodo = "Sandro";
persona.canciones = new Array();

var cancion = new Object();
cancion.titulo = "rosa rosa";
persona.canciones.push(cancion);
```

```
var persona = {
  "nombre": "Roberto",
  "apellido": "Sanchez",
  "apodo": "Sandro",
  "canciones": [
    { "titulo": "rosa rosa" }
  ]
}
```

Utilizando javascript plano, cada declaración de variable y de propiedad se realiza en una línea de código. Muchas veces se generan también variables temporales en el espacio global que luego formarán parte del objeto padre.

Utilizando JSON, la variable persona es exactamente la misma pero declarada en una sola línea de código (los saltos de línea y tabulaciones los utilizamos para mejorar la legibilidad).

serializar y deserializar JSON

```
var clima = { dia: 'lunes', temp: 24, desc: 'despejado'};

//serializamos el objeto en un String
var jsonString = JSON.stringify(clima);

clima.temp = 33;
console.log(clima.temp);

//deserializamos el String y reconstruimos el objeto
clima = JSON.parse(jsonString);
console.log(clima.temp);
```

Javascript dispone del objeto JSON con dos métodos principales:

- **JSON.stringify()** convierte cualquier objeto javascript en un String JSON
- **JSON.parse()** intenta generar un objeto javascript a partir de un String

Si **JSON.parse** puede recibir un String, esto nos habilita a obtener el String utilizando un pedido AJAX e intentando convertir los caracteres almacenados en **ajax.responseText**.

De la misma manera, lenguajes de programación del lado del servidor disponen de funciones de codificación y decodificación de JSON (generando objetos nativos de ese lenguaje).

JSON.parse() vs window.eval()

Comprendiendo **eval()**

El método **window.eval** permite ejecutar un script utilizando un String, como podemos probar escribiendo este código

```
var operacion = '2 + 2 * 4';
var resultado = eval(operacion);
eval('var persona = { nombre: "Pepe" }');
console.log(resultado, persona);
```

Como vemos, window.eval puede

- obtener resultados
- declarar variables
- llamar funciones

Es por eso que, si vamos a trabajar con JSON, estamos abriendo la puerta a posibles ejecuciones indeseadas, lo que lo hace inseguro.

Utilizando **JSON.parse()**

El método **JSON.parse** lanza un error en caso de no poder interpretar el String proporcionado como un JSON *estricto* válido.

Un JSON *estricto*, a diferencia de la notación literal en JS, debe:

- utilizar comillas dobles al describir las claves de un objeto
- utilizar comillas dobles en los valores de tipo String

```
JSON.parse('{ nombre: "Pepe", edad: 22 }'); //error!
JSON.parse('{ "nombre": "Pepe", "edad": 22 }'); //OK
```

El formato estricto deber ser utilizado tanto para generar un String JSON, almacenar archivos JSON en archivos de texto, generar JSON como respuesta desde un servidor.

AJAX: JSON vs XML

```xml
<persona>
  <nombre>Roberto</nombre>
  <apellido>Sanchez</apellido>
  <apodo>Sandro</apodo>
  <cancion titulo="rosa rosa" />
  <cancion titulo="tengo" />
</persona>
```

```json
{
  "nombre": "Roberto",
  "apellido": "Sanchez",
  "apodo": "Sandro",
  "cancion": [
    { "titulo": "rosa rosa" },
    { "titulo": "tengo" }
  ]
}
```

Si bien se dice que XML es más verboso, por lo que ocupa más caracteres que JSON, esto va a depender mucho de la estructura descripta. Por otro lado el método de compresión gzip elimina las redundancias, lo que hace que este argumento no sea una *ley*.

Utilizar XML permite utilizar la API XML, lo que puede ser cómodo para algunas situaciones (obtener un nodo por id o nombre).

Utilizar JSON permite evaluar directamente el texto como un objeto javascript, lo que puede ser cómodo si estoy emulando una estructura de objetos.

Actualmente JSON es el método principal de intercambio de datos, por lo que muchos desarrolladores decimos que la X de AJAX debería ser una J (Asynchronous Javascript and JSON) :D

AJAX, cross origin y JSON

¿Qué es el JSON con padding (o JSONP)?

Es una estrategia para poder buscar objetos JSON a servidores que se encuentren en un dominio diferente al de la página actual* .

La idea es la siguiente: si bien sabemos que un objeto **XHR** no tiene permitido realizar pedidos en otros dominios**, las etiquetas **<script>** de HTML no tienen esta limitación (de otra forma, ¿cómo es posible que se incluya código externo?)

También sabemos que mediante la API DOM puedo crear elementos en tiempo de ejecución. Entonces, podríamos crear un **<script>** en el momento deseado, cuyo **src** apunte a un servidor en otro dominio y obtener datos de esa forma, siempre recordando que el navegador *ejecuta* el script obtenido.

* ¿Recuerdan el **Cross Origin Request Error** de la lección pasada?
** A menos que se implemente **CORS** en el servidor, tal como vimos

Implementando JSONP

Si bien la respuesta del servidor podría ser algo así como

```
var clima = { dia: 'lunes', temp: 24, desc: 'despejado'};
```

Corremos el riesgo de que este script *pise* una variable **clima** previamente definida. Es por eso que la respuesta JSONP generalmente ejecuta una función que deberá estar definida en el cliente:

```
recibirClima({ dia: 'lunes', temp: 24, desc: 'despejado'});
```

Muchos servidores que implementan JSONP permiten enviar como parámetro qué función *callback* queremos que se ejecute de nuestro lado con los datos esperados

Ejemplo de implementación JSONP

```
var boton = document.querySelector('#miboton');
boton.addEventListener('click', pedidoJSONP);

function pedidoJSONP(){
  var script = document.createElement('script');
  script.src = 'http://otro-dominio/datos/?fn=respuesta';
  document.querySelector('body').appendChild(script);
}

function respuesta(datos){
  console.log('Funciona!', datos);
}
```

Suponiendo que exista un servidor en **http://otro-dominio/datos** que espere un parámetro GET **fn** donde defino la función callback que quiero ejecutar, este ejemplo genera por cada click un **<script>** que es añadido al **<body>** del HTML.

Al ser una etiqueta **<script>** nunca se dispara el **Cross Origin Error** y el server se encargará de generar un resultado similar a este (pseudo código para dar una idea)

```
respuesta([{...}, {...}, {...}]);
```

Conclusión: como puede observarse, JSONP es una *técnica* y requiere ponerse de acuerdo de alguna forma entre cliente y servidor para poder implementarlo.

Actualmente la implementación CORS trata de ofrecer una solución estandarizada a este problema, aunque sólo la adoptan navegadores modernos.

Ejercicios en clase

- En el siguiente link se encuentran los ejercicios para trabajar en clase, con sus soluciones. Plunker nos permite ejecutar código online y compartirlo, sin necesidad de tener nada instalado más que el navegador web.