Laboratorio de Computación II

Unidad 5: JavaScript Avanzado

Tecnicatura Universitaria en Programación - UTN

JavaScript Avanzado - Introducción

En la **Unidad 4** se han dado las características básicas y más importantes de JavaScript. En esta unidad, seguiremos profundizando en este potente y amplio lenguaje, en nuevos conceptos y en nuevas funcionalidades que han aparecido con ECMAScript 6 y versiones posteriores del estándar que nos permitirán lograr mejores rendimientos no sólo de nuestras aplicaciones Web sino también que nos facilitarán la tarea a la hora de codificar.

JavaScript - Asincronía

La asincronía es uno de los pilares fundamentales de Javascript, ya que es un lenguaje de programación de un solo hilo (single thread), lo que significa que sólo puede ejecutar una cosa a la vez.

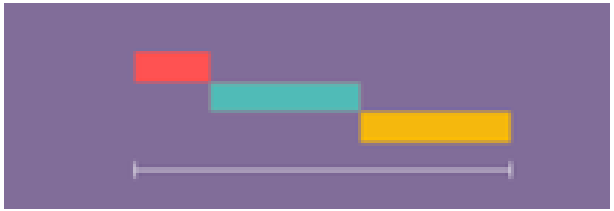
Imagina que solicitas datos de una API, dependiendo de la situación, el servidor puede tardar un tiempo en procesar la solicitud mientras bloquea el hilo principal y hace que la página web no responda. Ahí es donde entra en juego la asincronía que permite realizar largas solicitudes de red sin bloquear el hilo principal.

JS fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones de usuario, al tiempo que mantiene una interfaz fluida.

JS usa un modelo asíncrono y no bloqueante, con un loop de eventos implementado en un sólo hilo, (single thread) para operaciones de entrada y salida (input/output).

Gracias a esta solución, Javascript es áltamente concurrente a pesar de emplear un sólo hilo.

Sincronía



Asincronía

JavaScript - Asincronía

Control de la asincronía

Para controlar la asincronía JavaScript cuenta con diferentes mecanismos, ellos son:

- Callbacks
- Promesas
- Async/Await

Callbacks

Este concepto ha sido profundizado en la unidad anterior. Resumiendo, una **callback** es una función que se ejecutará después de que otra la haga.

Es un mecanismo para asegurar que cierto código no se ejecute hasta que otro haya terminado.

Al ser JavaScript un lenguaje orientado a eventos, las callbacks son una buena técnica para controlar la asincronía.

JavaScript - Asincronía

Callback - Ejemplo simulando conexión a API

En el siguiente ejemplo, la función “simulateApiConnection” simula ser una conexión a un API, es decir un proceso que demora X cantidad de tiempo y que, sólo luego de que se procesa queremos ejecutar código JS.

```
// Simula conexión a API con tiempo de respuesta aleatorio
function simulateApiConnection(callback) {
    setTimeout(function() {
        let response = "respuesta del servidor"
        callback(response)
    }, randomIntFromInterval(1000, 10000));
}
// Genera números random
function randomIntFromInterval(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min)
}
// Llamo a la API y le paso una función (callback)
simulateApiConnection(function(response) {
    console.log(response);
});
```

Promesas

Una promesa es un objeto que representa el resultado de una operación asíncrona y tiene 3 estados posibles:

1. Pendiente.
2. Resuelta.
3. Rechazada.

Tienen la particularidad de que se pueden encadenar (**then**), siendo el resultado de una promesa, los datos de entrada de otra posible función.

Las **promesas** mantienen un código más legible y mantenible que las callbacks, además tienen un mecanismo para la detección de errores (**catch**) que es posible usar en cualquier parte del flujo de datos. Además, cuentan con mecanismo “**finally**” que se ejecuta siempre, sea que salga todo como lo esperado (**then**) o por error (**catch**).

JavaScript - Asincronía

Promesa - Ejemplo simulando conexión a API

En este ejemplo se realiza la simulación de una conexión a una API pero utilizando una Promesa.

```
// Declaración función que devuelve una promesa
function simulateApiConnection() {
    return new Promise((resolve, reject) => {
        setTimeout(function() {
            let response = "respuesta del servidor"
            resolve(response)
        }, randomIntFromInterval(1000, 10000));
    });
}
// Llamo a la promesa que simula conexión a la API
simulateApiConnection()
    .then((response) => {
        console.log(response);
    }).catch((error) => {
        console.error(error);
    });
```

Async/await

Las promesas fueron una gran mejora respecto a las callbacks para controlar la asincronía en JavaScript, sin embargo pueden llegar a ser muy tediosas de manejar a medida que se requieran más y más métodos **.then()** o promesas desencadenadas.

Las funciones asíncronas (**async / await**) surgen para simplificar el manejo de las promesas.

La palabra **async** declara una función como asíncrona e indica que una promesa será automáticamente devuelta.

Podemos declarar como async cualquier tipo de función (con nombres,, anónimas o funciones flecha).

La palabra **await** debe ser usada siempre dentro de una función declarada como async y esperará de forma asíncrona y no bloqueante a que una promesa se resuelva o rechace.

JavaScript - Asincronía

Async/await - Ejemplo simulando conexión a API

En este ejemplo se realiza la simulación de una conexión a una API utilizando una Promesa pero con async y await.

```
// Declaración función que devuelve una promesa
function simulateApiConnection() {
    return new Promise((resolve, reject) => {
        setTimeout(function() {
            let response = "respuesta del servidor"
            resolve(response)
        }, randomIntFromInterval(1000, 10000));
    });
}
// Declaro función async donde se ejecuta la promesa con await
async function test() {
    try {
        // Utilización de await para ejecutar la promesa
        let response = await simulateApiConnection();
        console.log(response);
    } catch (error) {
        console.error(error);
    }
}
```

Callbacks VS Promesas VS Async/await

A continuación, se presenta un ejemplo de acciones desencadenadas, utilizando las 3 alternativas vistas:

```
// Callbacks
hazAlgo(function(resultado) {
    hazAlgoMas(resultado, function(nuevoResultado) {
        hazLaTerceraCosa(nuevoResultado, function(resultadoFinal) {
            console.log(`Obtenido el resultado final: ${resultadoFinal}`);
        }, falloCallback);
    }, falloCallback);
}, falloCallback);
// Promesas
hazAlgo().then(function(resultado) {
        return hazAlgoMas(resultado);
    })
    .then(function(nuevoResultado) {
        return hazLaTerceraCosa(nuevoResultado);
    })
    .then(function(resultadoFinal) {
        console.log(`Obtenido el resultado final: ${resultadoFinal}`);
    })
    .catch(falloCallback);
// Async/await
async function test() {
    try {
        let resultado = await hazAlgo();
        let nuevoResultado = await hazAlgoMas(resultado);
        let resultadoFinal = await hazLaTerceraCosa(nuevoResultado);
        console.log(`Obtenido el resultado final: ${resultadoFinal}`);
    } catch(error) {
        falloCallback(error);
    }
}
```

JavaScript - API Fetch

AJAX

JavaScript puede enviar peticiones de red al servidor y cargar nueva información siempre que se necesite.

Por ejemplo, podemos utilizar una petición de red para:

- Crear una orden,
- Cargar información de usuario,
- Recibir las últimas actualizaciones desde un servidor,
- …etc.

…Y todo esto sin la necesidad de refrescar la página.

Se utiliza el término global "**AJAX**" (abreviado **A**synchronous **J**avaScript **A**nd **XML**) para referirse a las peticiones de red originadas desde JavaScript. Sin embargo, no estamos necesariamente condicionados a utilizar XML dado que el término es antiguo y es por esto que el acrónimo **XML** se encuentra aquí, actualmente los datos se envían en formato **JSON** y es prácticamente un estandar en todas las API.

Fetch API

La interfaz de Fetch API permite al navegador web realizar solicitudes HTTP a los servidores web.

```
fetch(url, [options])
```

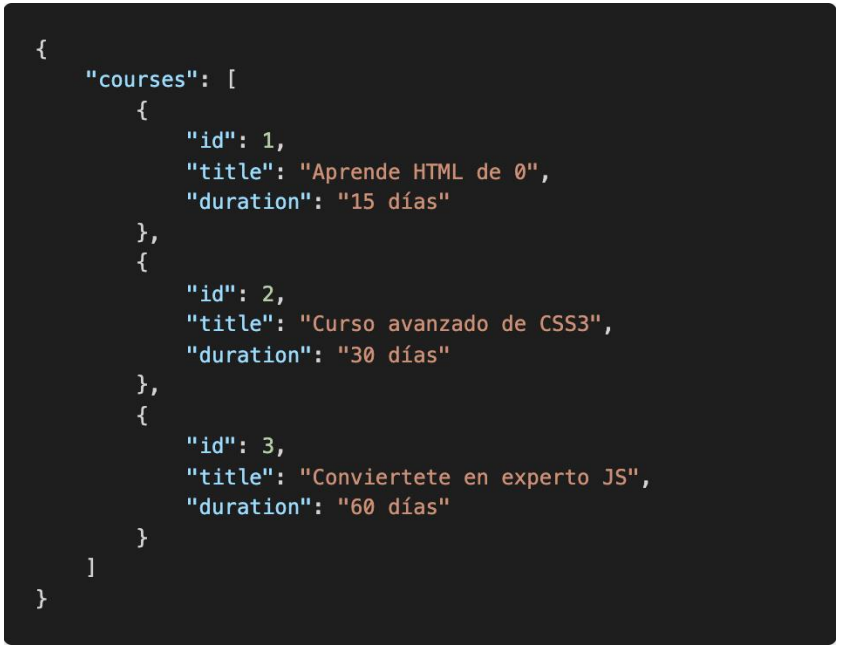
La implementación del método **fetch()** está basado en las promesas, por lo que podemos utilizarlo como tal, con then y catch, o bien, con async/await. En otras palabras, al instanciar el método fetch, se devolverá una promesa la cual podremos manejar por el success (then) o el error (catch). Ver el siguiente ejemplo:

```
fetch('https://my-servidor.com/noticias')
    .then((response) => {
        // Respuesta satisfactoria
        console.log(response) // Muestro el listado de noticias
    }).catch((error) => {
        // Se produce un error
        console.error(error); // Muestro el error
    });
```

JavaScript - JSON

JSON

JSON (por sus siglas JavaScript Object Notation) es un formato para almacenar y transportar información. Se asemeja mucho a lo que son los objetos en JavaScript y eso se debe a que fue creado para este lenguaje convirtiéndose luego en un estándar para todos los lenguajes de programación.



```
{
    "courses": [
        {
            "id": 1,
            "title": "Aprende HTML de 0",
            "duration": "15 días"
        },
        {
            "id": 2,
            "title": "Curso avanzado de CSS3",
            "duration": "30 días"
        },
        {
            "id": 3,
            "title": "Conviertete en experto JS",
            "duration": "60 días"
        }
    ]
}
```

Preguntas?

JavaScript - ECMAScript 6

ECMAScript 6, también conocido como **ECMAScript 2015** o **ES6,** es una nueva versión de Javascript, aprobada en Junio 2015.

Se podría considerar que es una auténtica revolución en la sintaxis de Javascript. Su buque insignia es, probablemente, una clara orientación a **clases y herencia**, pero la verdad es que hay muchas otras novedades interesantes, como el uso de **módulos**, los **parámetros por defecto,** las variables **let** y **const**, o la novedosa sintaxis de las **funciones flecha**, entre otros cambios que veremos a continuación.

ES6 - Principales novedades

Variables let y const

Como hemos visto en la unidad anterior, al momento de declarar una variable lo podemos hacer con **var**, **let** o **const.**

Tanto **let** como **const** se incorporaron en esta nueva versión de JS y sus características principales son las siguientes:

- **let**: es muy similar a **var**, aunque se recomienda utilizar siempre **let** por las consideraciones que aquí se detallan.
- **const**: esta palabra reservada nos permite declarar variables constantes, por lo que debemos asignar su valor al momento de la declaración sabiendo que el mismo no se va a poder modificar.

Funciones Flecha (Arrow Functions)

Las arrow proporcionan una sintaxis más compacta para la definición de funciones. Las principales diferencias entre éstas y las funciones convencionales, además de la manera en que se declaran, son:

- Las arrow tienen un return implícito en caso de que la función sea de única línea
- El valor de la palabra reservada **this** dentro de las arrow es diferente al de las funciones convencionales.

Ver aquí todas las diferencias y ejemplos entre ambas funciones.

La palabra reservada **this** hace referencia al contexto donde se está ejecutando nuestro código. Ver este artículo para comprender this y sus usos.

ES6 - Principales novedades

Parámetros por defecto

En los parámetros de entrada de una función, podemos asignarle un valor por defecto, el cual será asignado cuando la función sea llamada y no se le envíe valor para el parámetro en cuestión.

```
function greet(name="Desconocido") {
    console.log('Hola ' + name);
}
greet("Michael"); // Resultado: 'Hola Michael'
greet(); // Resultado: 'Hola Desconocido'
```

Parámetros REST

Los parámetros REST nos proporcionan una manera de pasar un conjunto indeterminado de parámetros que la función agrupa en forma de Array. Como detalle (de lógica), solo puede ser parámetro REST el último argumento de la función.

```
function printName(name, ...fancyNames){
    var fullName = name;
    fancyNames.forEach(fancyN => fullName += ' ' + fancyN);
    console.log(fullName);
};

printName('Felipe'); // Felipe
printName('Felipe', 'Juan', 'Miguel', 'Luis'); // Felipe Juan Miguel Luis
```

ES6 - Principales novedades

Desestructuración

La sintaxis de desestructuración JavaScript permite extraer datos de matrices y objetos dentro de variables, resultando en una sintaxis ágil y cómoda.

Este concepto es el mismo que se aplica en los parámetros REST de las funciones que se ha visto en el slide anterior.

```
let a, b, rest;
// Matriz/arreglo #1
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20
// Matriz/arreglo #2
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]
// Objeto y propiedades #1
({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20
// Objeto y propiedades #2
({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});
console.log(a); // 10
console.log(b); // 20
console.log(rest); // {c: 30, d: 40}
```

ES6 - Principales novedades

Template String Literals

Las plantillas literales o template string literals son cadenas de texto o strings que habilitan el uso de expresiones incrustadas, es decir, de código JavaScript (variables, funciones, operadores ternarios, for, etc.).

Con ellas, es posible utilizar cadenas de caracteres de más de una línea, y funcionalidades de interpolación de cadenas de caracteres.

Cuando dentro del Template Literal se coloca **${ }** se abre una especie de ventana a nuestro script desde el cual podemos llamar a funciones, variables o realizar cálculos, etc.

```
console.log(`Línea 1 de la cadena de texto
Línea 2 de la cadena de texto`);
// Línea 1 de la cadena de texto
// Línea 2 de la cadena de texto

let a = 5;
let b = 10;
console.log(`La suma es: ${a + b}`);
// La suma es: 15

let isEnable = true;
const classes = `my-class-examp ${ isActive ? 'enable' :
'disable' }`;
// my-class-examp enable
```

ES6 - Principales novedades

Nuevos métodos iterables

A la hora de trabajar con arreglos se dispone de nuevos métodos para sacar el mayor provecho al lenguaje. Hasta incluso a algunos métodos los podemos utilizar para recorrer las propiedades de un objeto o bien, los caracteres de un string. A continuación, veremos sólo algunos:

array.find()

Devuelve el valor del primer elemento del array que cumple la condición proporcionada.

```
const array1 = [5, 12, 8, 130, 44];
const found = array1.find(element => element > 10);
console.log(found); // Resultado: 12
```

array.includes()

Permite determinar si un arreglo incluye o no un elemento determinado.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const isThereMango = fruits.includes("Mango");
console.log(isThereMango); // Resultado: true
```

array.filter()

Crea un nuevo array con todos los elementos que cumplan la condición implementada.

```
const words = ['exuberante', 'talco', 'pera', 'elite',
'construcción', 'presente'];
const result = words.filter(word => word.length > 6);
console.log(result);
// Resultado: ['exuberante', 'construcción', 'presente']
```

array.map()

Este método crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(x) {
   return x * 2;
});
// doubles es: [2, 10, 20, 30]
// numbers sigue siendo [1, 5, 10, 15]
```

ES6 - Principales novedades

Clases y Herencia

Hasta antes de ES6, las clases y herencias se manejaban de una manera un tanto complicada, a partir de esta versión se implementaron las clases (**class**) propiamente dichas, con conceptos como constructores, herencia, etc. propios de la programación orientada a objetos. En sí, para JS las clases son funciones “complejas”, pero para nosotros son clases de las cuales se puede heredar, generar objetos, etc.

Para seguir profundizando sobre clases y sus utilidades, ver el siguiente artículo.

```
// Creación de clase
class Car {
    // Constructor
    constructor(brand) {
        this.carname = brand;
    }
    // Método
    present() {
        return `Mi auto es ${this.carname}`;
    }
}
// Herencia de clase
class Model extends Car {
    constructor(brand, mod) {
        super(brand);
        this.model = mod;
    }
    show() {
        return `${this.present()}, modelo ${this.model}`;
    }
    // Getter
    get cmodel() {
        return this.model;
    }
    // Setter
    set cmodel(newModel) {
        this.model = newModel;
    }
}
// Instanciación / creación del objeto
let myCar = new Model("Ford", "Focus");
console.log(myCar.show()); // Mi auto es Ford, modelo Focus
```

ES6 - Principales novedades

Módulos

A medida que nuestra aplicación crece, necesitaremos dividir/organizar el código en múltiples archivos, llamados “módulos”. Un módulo puede contener una clase o una biblioteca de funciones, variables, etc. para un propósito específico.

Un módulo es solo un archivo. Un script es un módulo. Los módulos pueden cargarse entre sí y usar directivas especiales **export** e **import** para intercambiar funcionalidad e información.

Los módulos deben hacer **export** de lo que ellos quieren que esté accesible desde afuera y hacer **import** de lo que necesiten de otros módulos.

Utilizar módulos entre archivos **.js** nos permite evitar agregar a estos archivos en nuestros **.html**. En caso de que necesitemos trabajar con módulo en un archivo **.html**, debemos colocar al elemento script la propiedad **type=”module”**.

Ver esta guía completa sobre módulos.

math.js

```
export const PI = 3.14159265358979323;

export function doAddition(num1, num2) {
    return num1 + num2;
}

export function doSubtraction(num1, num2) {
    return num1 - num2;
}
```

index.js

```
import { doAddition, PI } from './math.js';

console.log(`El valor de PI es: ${PI}`);
// Resultado: El valor de PI es: 3.141592653589793
console.log(`5 + 5 es: ${doAddition(5,5)}`);
// Resultado: 5 + 5 es: 10
```

Preguntas?