


UT4: VANILLA JS III - Avanzado




JS



ÍNDICE

1. CommonJS vs ES modules
2. Destructuring
3. Spread operator
4. Template strings
5. POO
6. Asincronía
7. Fetch
8. Try catch
9. Apis
10. POST
11. Json server
12. APIS de almacenamiento web



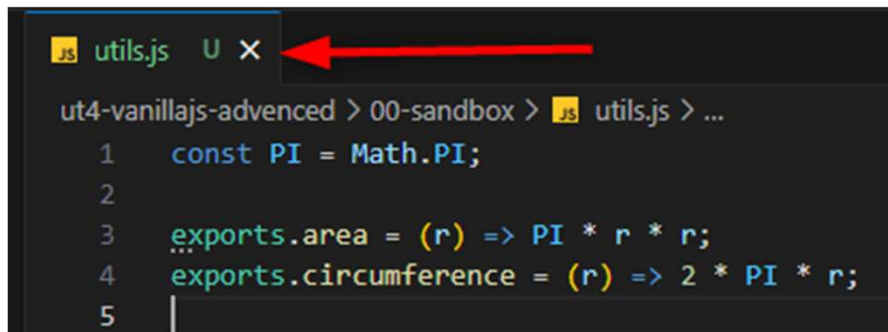
JS

1. CommonJS vs ES modules

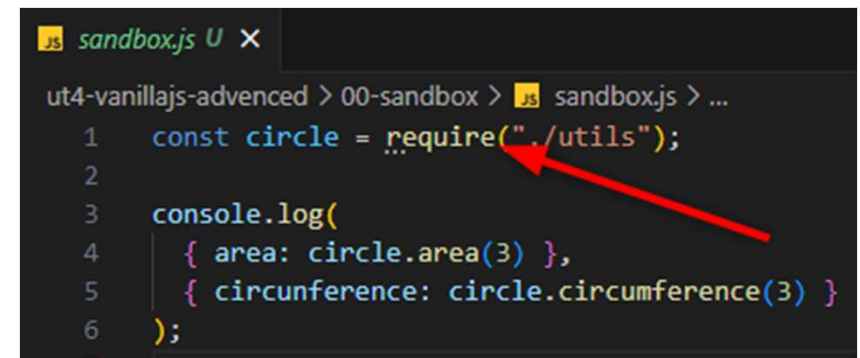
Estos dos conceptos hacen referencia al sistema de módulos utilizados, básicamente a como realizamos la importación desde otros archivos.

CommonJS(CJS)

CommonJS surge cerca de 2009 como una serie de pautas para crear un sistema de módulos en el ecosistema Javascript. Más tarde **NodeJS** adopta e implementa una versión mejorada de este sistema, haciéndolo muy popular. Ejemplo:



```
JS utils.js U X
ut4-vanillajs-advanced > 00-sandbox > JS utils.js > ...
1  const PI = Math.PI;
2
3  exports.area = (r) => PI * r * r;
4  exports.circumference = (r) => 2 * PI * r;
5
```



```
JS sandbox.js U X
ut4-vanillajs-advanced > 00-sandbox > JS sandbox.js > ...
1  const circle = require("../utils");
2
3  console.log(
4    { area: circle.area(3) },
5    { circumference: circle.circumference(3) }
6  );
```

1. CommonJs vs ES modules

CommonJS(CJS)

Palabras claves:

- ▶ **`module.exports`** -> exporta todo lo que hay en su interior

```
module.exports = {  
  area: (r) => PI * r * r,  
  circumference: (r) => 2 * PI * r,  
};
```

- ▶ **`exports.functionName`** -> exporta solo la función

```
exports.area = (r) => PI * r * r;
```

1. CommonJs vs ES modules

CommonJS(CJS)

Palabras claves:

► **Require** -> nos permite importar módulo CJS

```
const circle = require("../utils");

console.log(
  { area: circle.area(3) },
  { circumference: circle.circumference(3) }
);
```

Importante los require son creados por NodeJS por lo tanto no son directamente compatibles con el navegador. Si queremos usar importación/exportación usando CommonJS se deberá usar un **Bundler** para que haga las transformaciones necesarias.

1. CommonJs vs ES modules

ES Module (ESM)

En 2015, se libera **ECMAScript 2015** y con ella multitud de novedades en Javascript. Una de ellas, el sistema de módulos nativos de Javascript (ESM), que es una evolución de CommonJS.

```
export const PI = Math.PI;  
  
export const area = (r) => PI * r * r;  
export const circumference = (r) => 2 * PI * r;
```

```
import * as circle from "../utils.js";  
  
console.log(  
  { area: circle.area(3) },  
  { circumference: circle.circumference(3) }  
);
```

```
<script src="sandbox.js" type="module"></script>
```

En el proyecto del cuestionario sacar el mock de datos y la creación de la ventana modal a módulos independientes.

2. Destructuring

El **Destructuring** es una técnica de JS moderno que nos permite extraer de forma limpia **valores de arrays o propiedades de un objetos**.

Destrucuturación de Arrays

```
elements = [5, 4, 3, 2];  
[first, second] = elements; // first = 5, second = 4, rest = discard  
  
elements = [5, 4, 3, 2];  
[first, , third] = elements; // first = 5, third = 3, rest = discard  
  
elements = [4];  
[first, second] = elements; // first = 4, second = undefined
```


2. Destructuring

Destructuración de Objetos

```
const person = {  
  name: "Pedro",  
  surname: "Perez",  
  age: 32,  
  job: "Doctor",  
};  
  
const { name, surname, age, job } = person;  
console.log(name, surname, age, job);
```



```
$ node sandbox.js  
Pedro Perez 32 Doctor
```


2. Destructuring

Renombrados de propiedades

La destructuración también nos permite renombrar las variables donde se almacenan las propiedades extraídas

```
const person = {  
  name: "Pedro",  
  surname: "Perez",  
  age: 32,  
  job: "Doctor",  
};  
  
const { surname: apellido, age: edad, job: trabajo } = person;  
console.log(apellido, edad, trabajo);  
console.log({ apellido, edad, trabajo });
```



```
$ node sandbox.js  
Perez 32 Doctor  
{ apellido: 'Perez', edad: 32, trabajo: 'Doctor' }
```

2. Destructuring

Destructuración en parámetros de una función

La destructuración también nos permite usar las propiedades de un objeto en una función

```
const person = {  
  name: "Pedro",  
  surname: "Perez",  
  age: 32,  
  job: "Doctor",  
};  
  
const getPerson = ({ surname, job }) => {  
  console.log(surname, job);  
};  
  
getPerson(person);
```



```
$ node sandbox.js  
Perez Doctor
```

2. Destructuring

Destructuración en el return de una función

```
const cars = ["Audi", "Mercedes", "BMW", "Tesla"];

const getCars = (cars) => {
  return cars;
};

[Primero, , , Ultimo] = getCars(cars);
console.log(Primero, Ultimo);
```

\$ node sandbox.js
Audi Tesla

2. Destructuring

Destructuración en importación de módulos.

Ya hemos visto que al usar la siguiente sintaxis, importamos todo lo que esté precedido de la palabra reserva **export**, pero podemos aplicar **destrucción** para importar solo lo que queramos.

```
export const PI = Math.PI;  
export const area = (r) => PI * r * r;  
export const circumference = (r) => 2 * PI * r;
```

`import * as circle from "../utils.js";`

`import { PI, circumference } from "../utils.js";`

Hacer Ejercicio

3. Spread Operator

Este operador fue incorporado con la versión **ECMAScript 6**, su sintaxis son los tres puntos “...” y su función principal es convertir un array en el conjunto de sus valores.

```
let arr_1 = [1, 2, 3];  
console.log(arr_1);  
console.log(arr_1[0], arr_1[1], arr_1[2]);  
console.log(...arr_1);
```



```
[ 1, 2, 3 ]  
1 2 3  
1 2 3
```

3. Spread Operator

Spread operator en arrays

- Copia de array por **valor**

```
let arr_1 = [1, 2, 3];  
let arr_2 = [...arr_1];
```

- Concatenación de arrays:

```
var midArray = [3, 4];  
var arr = [1, 2, ...midArray, 5, 6];  
//arr → [1, 2, 3, 4, 5, 6]
```

3. Spread Operator

Spread operator en arrays

- Spread como parámetros en un función

```
let arr_1 = [1, 2, 3];  
const sumarTresNumeros = (num1, num2, num3) => {  
  return num1 + num2 + num3;  
};  
  
console.log(sumarTresNumeros(arr_1[0], arr_1[1], arr_1[2])); // Sin Spread  
console.log(sumarTresNumeros(...arr_1)); // Con Spread
```



6
6
6

3. Spread Operator

Spread operator en objetos

- Copiar objetos por valor

```
const person_1 = {  
  name: "Pedro",  
  surname: "Lopez",  
  age: 26,  
};  
const person_2 = { ...person_1 };  
person_1.age = 33;  
console.log({ person_1 });  
console.log({ person_2 });
```



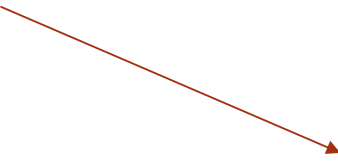
```
{ person_1: { name: 'Pedro', surname: 'Lopez', age: 33 } }  
{ person_2: { name: 'Pedro', surname: 'Lopez', age: 26 } }  
[pedemon] clean_exit: waiting for changes before restart
```

3. Spread Operator

Spread operator en objetos

- Unión de objetos por valor

```
const person_1 = {  
  name: "Pedro",  
  surname: "Lopez",  
  age: 26,  
};  
  
const job = {  
  name: "Doctor",  
  salary: "36000$",  
};  
  
const person = {  
  ...person_1,  
  ...job,  
};  
console.log({ person });
```



```
{ name: 'Doctor', surname: 'Lopez', age: 26, salary: '36000$' }
```

3. Spread Operator

Importante

Hay que tener en cuenta que el **spread operator** realiza un copia **débil** entre objetos, esto quiere decir que la copia **por valor** solo la hace para las propiedades de **primer nivel**, las propiedades de mayor profundidad las hace por **referencia**,

```
const person 1 = {  
  name: "Pedro",  
  surname: "Lopez",  
  age: 26,  
  job: {  
    name: "Doctor",  
    salary: "36000$",  
  },  
};
```

1º Nivel

2º Nivel

4. Template strings

- ▶ Los **template strings** son cadenas literales que habilitan el uso de expresiones incrustadas.
- ▶ Estas cadenas literales van delimitadas mediante las **backtick** (```)
- ▶ Para incrustar expresiones debemos usar **`${expresión}`**

```
const cars = ["Audi", "Mercedes", "BMW", "Tesla"];
const getCars = () => {
  return cars;
};

let [, secondCar] = cars;
let str = `Literal strings: ${getCars()[0]} ${secondCar}`;
console.log(str);
```

Literal strings: Audi Mercedes

5. Programación orientada a objetos

La Programación Orientada a Objetos (POO) es un estilo de programación muy utilizado, donde creas y utilizas estructuras de datos de una forma muy similar a la **vida real**, lo que facilita considerablemente la forma de planificar y estructurar el código.

Clase

Una clase en POO es una estructura de datos que se usa como **plantilla** para crear objetos.

Palabras reservadas en POO:

- ▶ **class** -> palabra reservada que se pone justo delante del nombre de mi estructura para indicar que ese elemento es una **clase**
- ▶ **constructor** -> función que nos permite crear e inicializar una **instancia de una clase**
- ▶ **this** -> el objeto que hace referencia a la propia clase
- ▶ **atributos/propiedades** -> son las variables y constantes definidas dentro de una clase y proporcionan características de la estructura que se está definiendo
- ▶ **objeto** -> la instancia de una clase
- ▶ **métodos** -> son funciones asociadas a la instancia de una clase.

5. Programación orientada a objetos

Ejemplo de Clase

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
  
  sayHello = () => {  
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);  
  }  
}
```

```
let person_1 = new Person("Pedro", 25)  
console.log(person_1.name)  
console.log(person_1.age)  
person_1.sayHello()
```



```
Pedro  
25  
Hello, my name is Pedro and I am 25 years old.
```

5. Programación orientada a objetos

Clase

En el ejemplo anterior debemos tener en cuenta ciertos aspectos:

- ▶ En nombre de una clase debe seguir notación **PascalCase**, la primera letra siempre en mayúscula. **class Person**.
- ▶ El archivo js que contiene a la clase también se suele nombrar en **PascalCase**. **Person.js**
- ▶ Una clase siempre debe tener el método **constructor**, este puede tener o no parámetros.
- ▶ Los **atributos** de una clase se pueden definir dentro o fuera del constructor

```
constructor(name, age) {  
  this.name = name;  
  this.age = age;  
}
```

- ▶ Una clase en JS no puede tener **más de un** constructor.
- ▶ Para **crear/instanciar** un objeto se debe usar la sintaxis: **let person_1 = new Person()**

5. Programación orientada a objetos

Buenas prácticas

- ▶ Generalmente, para tener el código lo más organizado posible, las clases se suelen almacenar en ficheros individuales.
- ▶ A la hora de crear objetos, lo habitual suele ser importar el fichero de la clase en cuestión y crear el objeto a partir de la clase.
- ▶ Si nuestra aplicación se complica mucho, podríamos comenzar a crear carpetas para organizar mejor nuestros ficheros de clases, dentro de una carpeta **classes**
- ▶ Las **propiedades de una clase** deberían tener nombre de **sustantivos**
- ▶ los **métodos de una clase** deberían tener nombre de **verbos**
- ▶ Intentar escribir código en inglés
- ▶ El número de líneas de una clase debería estar entre 100 y 300, siempre que esto sea posible

5. Programación orientada a objetos

Propiedades de clase

Las **propiedades**, son variables/constantas definidas dentro de una clase. Y al definirlas nos aportan ciertas características de dicha clase.

```
class Person {  
    #name;  
    age;  
    constructor(name, age) {  
        this.#name = name;  
        this.age = age;  
    }  
}
```

Visibilidad de una propiedad

- ▶ **Pública** -> las propiedades pueden ser leídas o modificadas tanto desde dentro de la clase como desde fuera
- ▶ **Privada** -> las propiedades sólo pueden ser leídas o modificadas desde el interior de la clase

5. Programación orientada a objetos

Visibilidad de una propiedad

Las propiedades de una clase siempre van a tener una **visibilidad** establecida, pública por defecto, o privada, añadiendo el carácter '#' justo delante del nombre.

Hay que tener en cuenta que la visibilidad **privada** se añadió en la versión ES2020 de ECMAScript por lo que se deberá tener en cuenta si esta característica está soportado por el navegador.

Si intentamos acceder a una propiedad privada fuera de la clase obtendremos el siguiente error:

```
class Person {  
  #name;  
  age;  
  constructor(name, age) {  
    this.#name = name;  
    this.age = age;  
  }  
}  
  
const person = new Person("Peter", 26)  
console.log(person.#name)
```



```
console.log(person.#name)  
                ^
```

```
SyntaxError: Private field '#name' must be declared in an enclosing class  
    at internalCompileFunction (node:internal/vm:73:18)
```

5. Programación orientada a objetos

Propiedad computadas (getters and setters)

Son propiedades que se declaran como **función** y que se ejecutan cuando accedemos a la propiedad con el nombre de la función. Existen dos tipos de propiedades computadas, los **getter** y los **setter**.

Propiedades get (getters)

- ▶ Para definirla solo se debe añadir la palabra reservada **get** justo antes de la función.
- ▶ Observar como se accede a la propiedad: **person.fullName**.

```
class Person {
  #name;
  surname;
  age;

  constructor(name, age, surname) {
    this.#name = name;
    this.surname = surname;
    this.age = age;
  }

  get fullName() {
    return `${this.#name} ${this.surname}`;
  }
}

const person = new Person("Peter", 26, "Doe");
console.log(person.fullName)
```

5. Programación orientada a objetos

Propiedades set (setters)

- ▶ Para definirla solo se debe añadir la palabra reservada **set** justo antes de la función.
- ▶ Observar como se **setea** la propiedad: **person.name = "John"**

Nota:

- ▶ Tener en cuenta que aunque parezcan funciones en realidad son **propiedades de clase**
- ▶ La idea de estas propiedades es permitir pequeñas modificaciones sobre propiedades ya existentes.
- ▶ Otro uso muy habitual es tener accesos a propiedades **privadas**

```
class Person {  
  #name;  
  surname;  
  age;  
  
  constructor(name, age, surname) {  
    this.#name = name;  
    this.surname = surname  
    this.age = age;  
  }  
  
  get name() {  
    return this.#name  
  }  
  
  set name(name) {  
    this.#name = name;  
  }  
  
  get fullName() {  
    return `${this.#name} ${this.surname}`  
  }  
}  
  
const person = new Person("Peter", 26, "Doe");  
person.name = "John"  
console.log(person.name)
```

5. Programación orientada a objetos

Métodos

Un método es una **función** declarada dentro de una clase.

```
class Person {  
  #name;  
  surname;  
  
  constructor(name, age) {  
    this.#name = name;  
    this.age = age;  
  }  
  
  getInfoPeson() {  
    return `Name: ${this.#name} \nAge: ${this.age}`;  
  }  
}  
  
const person_1 = new Person("Peter", 26, );  
console.log(person_1.getInfoPeson())
```



```
Name: Peter  
Age: 26
```

5. Programación orientada a objetos

Static

En el apartado anterior el método creado, se usa una vez que hemos creado una instancia de clase mediante la palabra reservada **new**.

Sin embargo, existen ocasiones en las que queremos acceder a una **propiedad o a un método**, sin que hayamos instanciado una clase, ahí es donde se usa la palabra reservada **static**.

Propiedades y métodos declarados como estáticos se denominan **propiedades y métodos de clase**.

```
class Person {  
  #name;  
  age;  
  static adult = 18;  
  
  constructor(name, age) {  
    this.#name = name;  
    this.age = age;  
  }  
}  
  
const person_1 = new Person("Peter", 26);  
console.log(Person.adult)  
console.log(person_1.adult)
```



18
undefined

5. Programación orientada a objetos

Visibilidad en los métodos

Al igual que en las propiedades, la visibilidad en los métodos es pública o privada, por defecto todos serán públicos, si queremos hacerlos privados debemos añadirle el carácter '#'

Si intentamos acceder a un método privado desde fuera de una clase se lanzará un error.

```
class Person {  
  #name;  
  age;  
  static adult = 18;  
  
  constructor(name, age) {  
    this.#name = name;  
    this.age = age;  
  }  
  
  #getName() {  
    return this.#name;  
  }  
}  
  
const person_1 = new Person("Peter", 26);  
console.log(person_1.#getName());
```

console.log(person_1.#getName())

SyntaxError: Private field '#getName' must be declared in an enclosing class
at internalCompileFunction (node:internal/vm:73:18)

Hacer ejercicios 2 y 3

5. Programación orientada a objetos

Herencia

A grandes rasgos, se puede denominar **herencia de clases** a la característica donde una **clase hija** obtiene las propiedades y métodos de una **clase padre** porque se ha establecido una relación entre ambas. Esa relación se establece a través de la palabra clave **extends**.

Desde **ECMaScript ES2015** javascript permite la herencia entre clases.

Ver sintaxis en la siguiente diapositiva

5. Programación orientada a objetos

Herencia - Sintaxis

```
class Person {  
  #name;  
  surname;  
  
  constructor(name, surname) {  
    this.#name = name;  
    this.surname = surname;  
  }  
  
  get name(){  
    return this.#name  
  }  
}
```



```
class Student extends Person {  
  #group;  
  constructor(name, surname, group){  
    super(name, surname);  
    this.#group = group;  
  }  
  get info() {  
    return `The ${this.name} ${this.surname} studies ${this.#group}`;  
  }  
}  
  
let student_1 = new Student("Peter", "Doe", "DAW");  
console.log(student_1.info);
```



The Peter Doe studies DAW

5. Programación orientada a objetos

Herencia

Como hemos visto en la diapositiva anterior:

- Para que una clase hija pueda heredar de una clase padre se debe usar la palabra **extends**

```
class Student extends Person
```

- La clase hija debe tener el método **super()** dentro de su constructor, cuya función es llamar al constructor de la clase padre

```
constructor(name, surname, group){  
  super(name, surname);  
  this.#group = group;  
}
```

- La clase hija hereda todos los atributos y métodos de la clase padre siempre que estos sean **públicos**.

¿Si esto es así porque se está heredando la propiedad name del ejemplo anterior?

Porque se implementa un getter como propiedad computada

5. Programación orientada a objetos

Herencia

- ▶ La clase hija puede **sobrescribir** los atributos y métodos de la clase padre

```
class Student extends Person {
  #group;

  constructor(name, surname, group) {
    super(name, surname);
    this.#group = group;
  }

  get name() {
    return "Hola"
  }

  get info() {
    return `The ${this.name} ${this.surname} studies ${this.#group}`;
  }
}

let student = new Student('Juan', 'Reina', 'DAW');
console.log(student.info);
```

→ The Hola Reina studies DAW

- ▶ La clase hija puede añadir los atributos y métodos que necesite
- ▶ Una clase solo puede extender de una única clase

5. Programación orientada a objetos

Herencia

- ▶ Explicar los ejercicios 4 - Herencias
 - ▶ Enseñar clases ejercicio 4
- ▶ Explicar los ejercicios 5 – Clases desde un JSON
 - ▶ Herramientas online para ver objetos JSON
 - ▶ Centrar elementos con FlexBox
- ▶ Explicar los ejercicios 6 – Ampliando ejercicio 5
 - ▶ Añadir ventana modal - Demo

6. Asincronía

El concepto de **asincronía** aparece para solucionar el problema de lenguajes **bloqueantes**, es decir lenguajes que no son capaces de seguir ejecutando código hasta que su tarea actual no haya terminado. Algunos ejemplos son:

- ▶ Peticiones HTTP
- ▶ Timers
- ▶ Tratamiento de eventos
- ▶ Consultas a base de datos

Javascript proporciona diferentes formas de gestionar la asincronía, donde las más importantes son:

- ▶ Callbacks
- ▶ Promesas
- ▶ Async/Await

6. Asincronía

Callbacks

Las callbacks son funciones que se pasan como parámetro a otras funciones. Con la particularidad de que esta función será llamada en un momento determinado.

Un callback por si solo no tiene porque ser **asíncrono**, se convierte en asíncrono cuando la función que le pasamos como parámetro es asíncrona.

Ejemplo de callback **síncrono**:

```
// Callback - sincrono
function addElement(array, callback) {
  array.push("Mercedes");
  callback()
  console.log("Estoy después del callback")
}

let cars = ["BMW", "KIA"];
addElement(cars, () => {
  console.log(`La longitud del array es ${cars.length}`);
});
```



```
La longitud del array es 3
Estoy después de la callback
```

6. Asincronía

Callbacks

El ejemplo más popular de callback **asíncrono** en javascript es la gestión de eventos del DOM.

Otro ejemplo muy claro donde podemos ver la asincronía mediante callback, es en el uso de la función **setTimeout**

```
// Asincronía - Callback
console.log("Tarea Sincrona 1")
setTimeout(() => {
  console.log('Me ejecuto paso 3 segundos')
}, 3000)
console.log("Tarea Sincrona 2")
```



```
Tarea Sincrona 1
Tarea Sincrona 2
Me ejecuto pasodo 3 segundos
```

6. Asincronía

Callbacks

Otro ejemplo

```
function addElementAsinc(array, callback) {  
  console.log(`Sincrono - La logitud del array es ${array.length}`);  
  setTimeout(() => {  
    callback(array)  
  }, 3000)  
  console.log(`Sincrono - La logitud del array es ${array.length}`);  
}  
  
let cars = ["BMW", "KIA"];  
addElementAsinc(cars, function (array) {  
  array.push("Mercedes");  
  console.log(`Asincrono - La logitud del array es ${array.length}`);  
});
```

Sincrono - La logitud del array es 2
Sincrono - La logitud del array es 2
Asincrono - La logitud del array es 3


6. Asincronía

Callbacks

Las funciones callback tienen ciertas desventajas:

- ▶ En primer lugar el código creado con las funciones callbacks suele ser algo confuso
- ▶ Y en segundo lugar y más importante lo que se conoce como el **Callback Hell**, que aparece cuando anidamos varias funciones callbacks.

Para evitar estos problemas se crearon las **Promesas**



```
function hell(win) {  
  // for listener purpose  
  return function() {  
    loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {  
      loadLink(win, REMOTE_SRC+'/lib/async.js', function() {  
        loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {  
          loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {  
            loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {  
              loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {  
                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {  
                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {  
                    loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {  
                      async.eachSeries(SRIPTS, function(src, callback) {  
                        loadScript(win, BASE_URL+src, callback);  
                      });  
                    });  
                  });  
                });  
              });  
            });  
          });  
        });  
      });  
    });  
  });  
}
```



6. Asincronía

Callbacks

Ejercicio – T4_P7 - Callbacks

6. Asincronía

Promesas

Nace para resolver el problema de asincronía de una forma mucho más elegante y práctica que, por ejemplo, utilizando funciones callbacks.

Una **Promise** (promesa en castellano) es un objeto que representa la **terminación** o el **fracaso** de una operación asíncrona.

Las promesas tienen 3 estados

- ▶ Incierto -> promesa pendiente
- ▶ Cumplida -> promesa resuelta
- ▶ No cumplida -> promesa rechazada.

Las promesas podemos.

- ▶ Consumirlas
- ▶ Crearlas

6. Asincronía

Promesas

La clase **Promise** nos ofrece los siguientes métodos:

- ▶ **then(resolve)** → Ejecuta la función **callback resolve** cuando la promesa se cumple.
- ▶ **catch(reject)** → Ejecuta la función **callback reject** cuando la promesa se rechaza.
- ▶ **then(resolve, reject)** → Método equivalente a los dos anteriores en el mismo `.then()`.
- ▶ **finally(end)** → Ejecuta la función **callback end** tanto si se cumple como si se rechaza.

6. Asincronía

Consumir Promesas

La forma general de consumir una promesa es utilizando el **.then()** con un sólo parámetro.

```
fetch("recurso a consumir").then((response) => {  
  /* Código a realizar cuando la promesa se  
  cumpla*/  
});
```

En el ejemplo anterior estamos usando la función **fetch**, que devuelve una promesa que se cumple cuando obtiene respuesta desde el parámetro **"recurso a consumir"**

6. Asincronía

Consumir Promesas

También podemos tratar si una promesa es **rechazada** añadiendo la función **catch**

```
fetch("recurso a consumir")
  .then((response) => {
    /* Código a realizar cuando la promesa se cumpla*/
  })
  .catch((error) => {
    /* Código a realizar cuando la promesa se rechace*/
  });
```

6. Asincronía

Consumir Promesas

Además es posible añadir varios *then's* si lo que devuelve el *then anterior* es una *promesa devuelta con return*

```
fetch("recurso a consumir")
  .then((response) => {
    return response.data
  })
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.log(error)
  });
```

6. Asincronía

Consumir Promesas

Por último, tenemos el método **finally**, que se ejecutará una vez la promesa haya terminado, por el **then** o por el **catch**.

Esta función se usa para realizar tareas de limpieza o acciones que deben ocurrir sin importar el resultado de la promesa

```
fetch("recurso a consumir")
  .then(response => { return response.data })
  .then(data => { console.log(data) })
  .catch(error => { console.log(error) })
  .finally(() => { console.log("Promesa terminada") })
```

6. Asincronía

Crear Promesas

Para implementar una promesa debemos **instanciar** un objeto de la clase **Pomise**, este constructor recibe como parámetro una función (callback) que a su vez recibe dos **parámetros**:

```
new Promise((resolve, reject) => {});
```

resolve -> contiene el código de cuando se cumple la promesa

reject -> contiene el código de cuando se rechaza la promesa

Una vez la promesa está implementada, esta se podrá consumir como hemos vistos en las diapositivas anteriores, usando **then, catch y finally**

6. Asincronía

Crear Promesas

Ejemplo de promesa:

```
const p = (value) => {  
  return new Promise((resolve, reject) => {  
    if (value < 5) {  
      resolve("All ok!!");  
    } else {  
      reject("All bad!!");  
    }  
  });  
};
```

6. Asincronía

Crear Promesas

```
console.log("antes")
p(2)
  .then((response) => console.log(response) )
  .catch((error) => console.log(error) )
  .finally(() => console.log("Fin" ) );
console.log("despues")
```



```
antes
despues
All ok!!
Fin
```

```
console.log("antes")
p(5)
  .then((response) => console.log(response) )
  .catch((error) => console.log(error) )
  .finally(() => console.log("Fin" ) );
console.log("despues")
```



```
antes
despues
All bad!!
Fin
```

Importante: Las promesas siempre son **asíncronas**.

6. Asincronía

Promesas – Ejemplo

```
const myPromise = (array) => {  
  return new Promise((resolve) => {  
    let newArray = array.map((item) => item + 1);  
    resolve(newArray);  
  });  
};  
  
myPromise([1, 2, 3])  
  .then((result) => result.map((item) => item + 2))  
  .then((result) => {  
    let newArray = result.map((item) => item + 3);  
    console.log(`Result: ${newArray}`);  
  })  
  .finally(() => console.log("All operations finished"));
```

Ejercicios

Result: 7,8,9
All operations finished

6. Asincronía

Async/Await

Las palabras **async/await** se introdujeron con la única finalidad de hacer algo más fácil la sintaxis de las promesas.

- ▶ Con **async/await** desaparecen los **then**, lo que se hace ahora es poner **await** justo delante:

```
myPromise([1, 2, 3])  
  .then((result) => console.log(result));
```



```
console.log(await myPromise([1, 2, 3]));
```

- ▶ Pero esto tal cual no funciona, nos daría el siguiente error, ya que **await es bloqueante**

```
SyntaxError: await is only valid in async functions and the top level bodies of modules  
    at internalCompileFunction (node:internal/vm:73:18)
```


6. Asincronía

Async/Await

Para solucionar el problema anterior debemos envolver el **await** en una función con la palabra reservada **async**

```
async function res() {  
  console.log(await myPromise([1, 2, 3]));  
}  
res();
```

6. Asincronía

Async/Await

Ejemplo

```
const myPromise = (array) => {  
  return new Promise((resolve) => {  
    let newArray = array.map((item) => item + 1);  
    resolve(newArray);  
  });  
};
```

```
myPromise([1, 2, 3])  
  .then((result) => myPromise(result))  
  .then((result) => myPromise(result))  
  .then((result) => console.log("Promise: " + result));
```

```
async function res() {  
  let result = await myPromise([1, 2, 3]);  
  result = await myPromise(result);  
  result = await myPromise(result);  
  console.log("Async/Await: " + result);  
}  
res();
```

Async/Await: 4,5,6
Promise: 4,5,6

6. Asincronía

Async/Await

Explicar ejercicios 9 y 10

7. Fetch

Fetch, es un sistema más moderno, basado en promesas de Javascript, para realizar peticiones **HTTP asíncronas** de una forma más legible y cómoda y que sustituye a la antigua librería **XMLHttpRequest**.

Fetch devuelve una promesa, y como ya hemos visto estas se consumen con **then/catch** o con **async/await**. Aunque el modo más habitual es usar then.

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

7. Fetch

Parámetros de fetch

La función fetch acepta dos parámetros de entrada:

- ▶ El primero es la **url** a la que se le hará la petición
- ▶ El segundo es opcional y es un **objeto literal** con la siguientes posibilidades:
 - ▶ **method** -> Método HTTP de la petición por defecto **GET**
 - ▶ **headers** -> Cabecera HTTP, por defecto {}
 - ▶ **body** -> Cuerpo de la petición HTTP, puede ser String, FormData, etc
 - ▶ **credentials** -> Modo de credenciales, por defecto sin credenciales.

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

7. Fetch

Ejemplo real del parámetro options

```
const url = 'https://car-data.p.rapidapi.com/cars?limit=10&page=0';
const options = {
  method: 'GET',
  headers: {
    'X-RapidAPI-Key': '6185a78aa2mshb04ae8991085691p1d093bjsnae8623070f4a',
    'X-RapidAPI-Host': 'car-data.p.rapidapi.com'
  }
};
```

7. Fetch

Parámetro method

Indica el método HTTP con el que se realiza la petición, si no se especifica por defecto es GET, pero estos pueden ser:

- ▶ **GET** -> Solicita la representación de un recurso específico
- ▶ **HEAD** -> Idéntica al GET pero sin que se devuelvan datos, solo se devuelve es **status** de la petición
- ▶ **POST** -> Se usa para enviar datos al servidor
- ▶ **PUT** -> Se suele usar para reemplazar información existente en el servidor
- ▶ **DELETE** -> Para eliminar un recurso de un servidor

7. Fetch

Parámetro method

Respuestas obtenidas en peticiones HTTP:

- ▶ ***Respuestas informativas (100–199)***
- ▶ ***Respuestas satisfactorias (200–299)***
- ▶ ***Redirecciones (300–399)***
- ▶ ***Errores de los clientes (400–499)***
- ▶ ***Errores de los servidores (500–599)***

7. Fetch

Objeto Response

El primer then de una petición fetch, devuelve el objeto **response**, que será la respuesta que nos llega desde el servidor web.

Este objeto tiene las siguiente **propiedades**:

- ▶ **status** -> Código HTTP de la respuesta (100-599).
- ▶ **statusText** -> Texto representativo del código 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.

7. Fetch

Objeto Response

```
fetch(url, options)
  .then((response) => {
    console.log(response.status);
    console.log(response.statusText);
    console.log(response.ok);
    console.log(response.url);
    console.log(response.headers);
  })
```

```
200
OK
true
https://car-data.p.rapidapi.com/cars?limit=10&page=0
HeadersList {
  cookies: null,
  [Symbol(headers map)]: Map(17) {
    'date' => { name: 'Date', value: 'Mon, 13 Nov 2023 11:51:20 GMT' },
    'content-type' => { name: 'Content-Type', value: 'application/json' },
    'content-length' => { name: 'Content-Length', value: '748' },
    'connection' => { name: 'Connection', value: 'keep-alive' },
    'x-amzn-requestid' => {
      name: 'x-amzn-RequestId',
      value: 'd5648678-e2d2-4864-ab38-a618f36628af'
    },
  },
  'x-ratelimit-requests-limit' => { name: 'X-RateLimit-Requests-Limit', value: '1000' },
  'x-ratelimit-requests-remaining' => { name: 'X-RateLimit-Requests-Remaining', value: '986' },
  'x-ratelimit-requests-reset' => { name: 'X-RateLimit-Requests-Reset', value: '2260461' },
  'x-ratelimit-rapid-free-plans-hard-limit-limit' => {
    name: 'X-RateLimit-rapid-free-plans-hard-limit-Limit',
    value: '500000'
  },
  'x-ratelimit-rapid-free-plans-hard-limit-remaining' => {
    name: 'X-RateLimit-rapid-free-plans-hard-limit-Remaining',
    value: '499986'
  },
}
```

7. Fetch

Objeto Response

Este objeto además de **propiedades** tiene los siguientes métodos:

- ▶ **text()** -> Devuelve una promesa con el texto plano de la respuesta.
- ▶ **json()** -> Devuelve una promesa con un objeto json.
- ▶ **blob()** -> Devuelve una promesa pero con un objeto Blob (binary large object).
- ▶ **arrayBuffer()** -> Devuelve una promesa con un objeto ArrayBuffer (buffer binario puro).
- ▶ **formData()** -> Devuelve una promesa 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.

7. Fetch

Objeto Response

```
fetch(url, options)
  .then((response) => response.text())
  .then((data) => console.log(data))
```

[{"id":9582,"year":2008,"make":"Buick","model":"Enclave","type":"SUV"}, {"id":9583,"year":2006,"make":"MINI","model":"Convertible","type":"Convertible"}]

```
fetch(url, options)
  .then((response) => response.json())
  .then((data) => console.log(data))
```

```
[
  {
    id: 9582,
    year: 2008,
    make: 'Buick',
    model: 'Enclave',
    type: 'SUV'
  },
  {
    id: 9583,
    year: 2006,
    make: 'MINI',
    model: 'Convertible',
    type: 'Convertible'
  }
]
```

7. Fetch

Objeto Response

```
fetch(url, options)
  .then((response) => response.blob())
  .then((data) => console.log(data))
```

Blob { size: 152, type: 'application/json' }

```
fetch(url, options)
  .then((response) => response.arrayBuffer())
  .then((data) => console.log(data))
```

```
ArrayBuffer {
  [Uint8Contents]: <5b 7b 22 69 64 22 3a 39 35 38 32 2c 22 79 65 61 72 22 3a 32 30 30 38 2c 22 6d 61 6b 65 22 3a 22 42 75
69 63 6b 22 2c 22 6d 6f 64 65 6c 22 3a 22 45 6e 63 6c 61 76 65 22 2c 22 74 79 70 65 22 3a 22 53 55 56 22 7d 2c 7b 22 69 64
22 3a 39 35 38 33 2c 22 79 65 61 72 22 3a 32 30 30 36 2c 22 6d 61 6b 65 22 ... 52 more bytes>,
  byteLength: 152
}
```

7. Fetch

Fetch usando async/await

```
fetch(url, options)
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => {
    console.error(error);
  });
```



```
async function getCars() {
  try {
    const response = await fetch(url, options);
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

getCars();
```

Explicar ejercicio 11

Explicar Rapidapi:

- ▶ <https://rapidapi.com/principalapis/api/car-data/>
- ▶ <https://rapidapi.com/weatherapi/api/weatherapi-com/>

8. Try/Catch

El tratamiento de errores/excepciones se realiza mediante el bloque **try..catch..finally**

- ▶ En el bloque **try** se añade el código que se quiere probar
- ▶ En el **catch** se trata el error en caso de que lo haya
- ▶ Y en el **finally** se ejecuta siempre, detrás del try o detrás del catch

En JS, las excepciones lanzan errores que pertenecen a la clase **Error**, esta clase tiene tres propiedades, **name**, **message**, **stack**:

```
const error = new Error("My error");  
error.name = "PaseError";  
throw error;
```

```
Error [PaseError]: My error  
    at Object <anonymous> (E:\Dev\gs-daw-dwec\error.js:1:15)  
    at Module._compile (node:internal/modules/cjs/loader:1254:14)  
    at Module._extensions..js (node:internal/modules/cjs/loader:1308:10)  
    at Module.load (node:internal/modules/cjs/loader:1117:32)  
    at Module._load (node:internal/modules/cjs/loader:958:12)  
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)  
    at node:internal/main/run_main_module:23:47  
Node.js v18.16.0
```

Message

Stack

Name

8. Try/Catch

Las excepciones definidas por el usuario se pueden lanzar con la función **throw**. **JS** permite lanzar cualquier cosa como excepción pero no es buena práctica:

```
throw "Error2"; // genera una excepción con un valor cadena
throw 42; // genera una excepción con un valor 42
throw true; // genera una excepción con un valor true
```

Lo recomendado es crear un objeto error y asignarle un nombre y un mensaje

```
try {
  const error = new Error("My error");
  error.name = "PaseError";
  throw error;
} catch (e) {
  console.log(e.message);
}
```


9. APIS

Las API son mecanismos que permiten a dos componentes software comunicarse entre sí mediante un conjunto de definiciones y protocolos.

En el ecosistemas JavaScript nos interesan las **API WebSocket** y las **API Rest**

► **API Rest**

Rest usa el protocolo **http/https** que es **unidireccional** entre el cliente-servidor, esto significa que el servidor solo nos responderá si el cliente ha hecho una solicitud.

► **API WebSocket**

WS es un protocolo de comunicación informática que proporciona canales de comunicación **full-duplex** a través de una única conexión TCP.

En esta situación siempre hay una conexión activa entre el cliente y servidor, por lo tanto el servidor podrá mandar información al cliente sin que este le haya preguntado.

WS se suele implementar en proyectos como chats, notificaciones push, o respuestas en tiempo real.

10. POST

POST

- ▶ Como ya vimos el método http Post envía datos al servidor, pero este puede que nos devuelva información haciendo uso de estos datos. Por ejemplo si creamos un nuevo usuario el servidor nos contestará con alguna respuesta.
- ▶ Además POST usa el parámetro **body** para enviar esta información

```
fetch(url, {  
  method: "POST", // or 'PUT'  
  body: JSON.stringify(data), // data can be `string` or {object}!  
  headers: {  
    "Content-Type": "application/json",  
  },  
})
```

- ▶ A veces si lo que se va a enviar en el **body** es algo más complejo se puede usar el objeto **URLSearchParams**

10. POST

Objeto *URLSearchParams*

- Esta api básicamente nos proporciona métodos útiles para trabajar con los parámetros de una URL.

```
const options_my_apy = {
  method: "POST",
  headers: {
    "content-type": "application/x-www-form-urlencoded",
    "Accept-Encoding": "gzip, deflate, br",
    "x-random": Math.random(),
    "x-rapidapi-key": "myapikey",
  },
  body: new URLSearchParams({
    param1: "value1",
    param2: "value2",
    param3: { nombre: "Pedro", apellido: "Perez" },
  })),
};
```

10. POST

Objeto URLSearchParams

- Si tuviéramos que modificar los valores de body sería

```
options_my_apy.body.set("param1", "new value");  
options_my_apy.body.set("param2", "new value");  
options_my_apy.body.set("param3", { nombre: "Juan", apellido: "Gomez" });
```

- Si tuviéramos que modificar los valores de body sería

```
options_my_apy.body.get("param1");  
options_my_apy.body.get("param2");  
options_my_apy.body.get("param3").nombre;  
options_my_apy.body.get("param3").apellido;
```

11. Json Server

JSON Server es una herramienta que permite crear una **API RESTful falsa** de manera rápida y sencilla. Está diseñada para usarse en entornos de desarrollo o pruebas, donde necesitas simular una API sin tener que implementar un **backend** completo

Cómo usar JSON Server:

1. Crear un proyecto npm -> ***npm init***
2. Instalar el paquete json-server vía npm -> ***npm install json-server***
3. Crear un archivo llamado ***db.json***. Este archivo contendrá los datos que se quieren exponer a través de la API.
4. Añadir el script para poder exponer el **mock server** -> ***"start": "json-server --watch db.json"***
5. Ejecutar el proyecto npm -> ***npm run start***

11. Json Server

Ejemplo del archivo db.json

```
{
  "posts": [
    { "id": 1, "title": "Primer post", "author": "Juan" },
    { "id": 2, "title": "Segundo post", "author": "Ana" }
  ],
  "comments": [
    { "id": 1, "body": "Comentario 1", "postId": 1 },
    { "id": 2, "body": "Comentario 2", "postId": 2 }
  ]
}
```

Crear proyecto para consumir datos