

Javascript Moderno


1. Para crear la aplicación de React se utiliza el comando: **npx create-react-app nombreApp.**

2. Variables y constantes

- No es recomendado utilizar var.
- Es recomendado utilizar const y let.
- Debemos utilizar const cuando no vayamos a cambiar el valor de la variable.
- Si el valor de la variable va a cambiar, utilizamos let.
- Scope

A screenshot of a code editor with a dark background. At the top left, there are three colored circles: red, yellow, and green. The code is as follows:

```
1  let valorDado = 4;
2
3  console.log(valorDado);           // 4
4
5
6  if(true){
7      let valorDado = 6;
8
9      console.log(valorDado);       // 6
10 }
11
12 console.log(valorDado);           // 4
```



```

1  let valorDado = 4;
2
3  console.log(valorDado);           // 4
4
5
6  if(true){
7      // let valorDado = 6;
8
9      console.log(valorDado);       // 4
10 }
11
12 console.log(valorDado);           // 4

```

- Dentro de un bloque podemos repetir el nombre de las variables porque no va a afectar a las variables con el mismo nombre que estan fuera de ese bloque.



```

1  const nombre = 'Robert';
2  const apellido = 'Szekeres';
3
4  let edad = 27;
5      edad = 28;
6
7
8  console.log(nombre, apellido, edad); // Robert Szekeres 28
9
10 if(true){
11     const nombre = 'Maria';
12     console.log(nombre);             // Maria
13 }
14
15 console.log(edad);                  //28

```

3. Template String

- El template string es una manera interesante para concatenar strings, para poner variables dentro de los strings o resultados de operaciones, poner caracteres o strings multi-lineas.
- Para escribir el template string se utiliza: ``${variable}``.

```
1  const nombre = 'Robert';
2  const apellido = 'Szekeres';
3
4  // const nombreCompleto = nombre + ' ' + apellido;
5  const nombreCompleto = `${nombre} ${apellido} ${1+1}`
6
7  console.log(nombreCompleto); // Robert Szekeres 2
8
9  // Funcion
10 function getSaludo(nombre){
11     return 'Hola ' + nombre;
12 }
13
14 console.log(`Este es un texto: ${getSaludo(nombre)}`); //Este es un texto: Hola Robert
```

4. Objetos literales y operado spread

- Las llaves en js significan que es un objeto: `{}`.
- El operado spread se utiliza para clonar. Se escribe con tres puntos: `...`

```
1  const persona = {
2    nombre: 'Robert',
3    apellido: 'Szekeres',
4    edad: 27,
5    direccion: {
6      ciudad: 'Paris',
7      cp: 94120,
8      lat: 12.4324,
9      lng: 23.12344
10   }
11 };
12
13 // console.table({persona});
14 console.log({persona});
15
16 // Operador Spread: ..., se utiliza para clonar.
17 const persona2 = { ... persona };
18 persona2.nombre = 'Jack';
19 persona2.apellido = 'Sparrow';
20 persona2.edad = '100';
21
22 console.log(persona2);
```

5. Arreglos (Arrays), map()

- Un array es una colección de información que se encuentra dentro de una misma variable.
- El array se escribe utilizando los corchetes: [].
- El método map() se utiliza para transformar un array, pero manteniendo el array original.
- El método map() ejecuta una función (callback) dada, en cada elemento del array original y crea un nuevo array con el resultado de la función dada. La

funcion debe tener un argumento (x), y este argumento va a ser cada elemento del array original. El argumento se puede llamar como queramos.

- El map lo que hace es parcurir el array original.

```
1 // Utilizar esta manera SOLO cuando se crea un array con un tamaño fijo
2 // const array = new Array(100);
3
4 // Utilizar esta manera: []
5 const array = [1,2,3,4];
6 // el metodo push() se utiliza para añadir informacion al array
7 // array.push(1);
8
9 // Operador spread para clonar el array original y añadirle el 5
10 let array2 = [ ... array, 5];
11
12 // map() es un metodo que parcorre un array
13 const array3 = array2.map(function(x){
14     return x * 2;
15 });
16
17
18 console.log(array); // 1,2,3,4
19 console.log(array2); // 1,2,3,4,5
20
21 // 2*1, 2*2, 2*3, 2*4, 2*5
22 console.log(array3); // 2,4,6,8,10
```

6. Funcion tradicional y funcion flecha

```
1 // Funcion tradicional;
2 const saludar = function(nombre){
3     return `Hola ${nombre}`;
4 }
5
6 // Funcion flecha
7 const saludar2 = (nombre) => {
8     return `Hola ${nombre}`;
9 }
10 // Funcion flecha en una sola linea
11 const saludar3 = (nombre) => `Hola ${nombre}`;
12
13
14 console.log(saludar('Robert'));
15 console.log(saludar2('Juan'));
16 console.log(saludar3('Pablo'));
17
18 // Si quitamos el return, hay que poner los parentesis ()
19 const getUser = () => ({
20     id: 'ABC123',
21     username: 'Elpapi'
22 })
23
24
25 const user = getUser();
26 console.log(user)
27
28 // Funcion flecha + return objeto implicito
29 // el objeto implicito esta entre parentesis
30 const getUsuarioActivo = (nombre) => ({
31     id: 'ABC789',
32     username: nombre
33 });
34
35 const usuarioActivo = getUsuarioActivo('Robert');
36 console.log(usuarioActivo);
```

7. Desestructuración de Objetos

- La desestructuración o Destructuring es una nueva característica de ES6 para javascript que nos da la posibilidad de poder coger los datos de objetos o arrays directamente y de manera múltiple, para extraerlos a variables o constantes.
- Nos permite desempaquetar valores de arrays u objetos en grupos de variables, permitiéndonos simplificar y crear código más legible.
- Algo que es muy usado en una función, es la desestructuración directamente en el argumento.

```
1  const persona = {
2    nombre: 'Robert',
3    edad: 27,
4    personaje: 'Goku',
5    rango: 'Capitan'
6  };
7
8  // Forma de desestructurar
9  // const {personaje, edad, nombre} = persona;
10
11 // console.log(nombre);
12 // console.log(edad);
13 // console.log(clave);
14
15 // Asignación de una propiedad + valor directamente en el argumento
16 const {nombre, edad, personaje, rango = 'Experto'} => {
17
18   // console.log(nombre, edad, clave, rango);
19
20   return {
21     tipo: personaje,
22     años: edad,
23     latlng: {
24       lat: 12.1234,
25       lng: -12.12424
26     },
27     nivel: 'Dios',
28     pseudo: 'robirobi'
29   }
30 }
31
32 const {tipo, años, latlng: {lat, lng}, nivel, pseudo} = persona(persona);
33
34 console.log(tipo, años, lat, lng, nivel, pseudo);
```

8. Desestructuración de Arrays

```
1  const personajes = ['Goku', 'Vegetta', 'Trunks'];
2
3  // Ponemos la coma para decirle que ignore los otros personajes
4  const [, , p3] = personajes;
5  console.log(p3);
6
7
8  const returnArray = () => {
9      return ['ABC', 123];
10 }
11
12 const [letras, numeros] = returnArray();
13
14 console.log(letras, numeros);
15
16
17 //
18
19 const state = (valor) => {
20     return [valor, () => {console.log('Hola Mundo')}]
21 }
22
23 const [nombre, apellido] = state('Robert');
24
25 console.log(nombre);
26 apellido();
```


9. Import, export y funciones comunes de arrays

- Import se usa para usar informacion de otros archivos.
- Para que funcione el import, en el archivo de donde estamos extrayendo la informacion, hay que poner el export.
- find()
- filter()

```
1 // Poner .js al ultimo archivo es Opcional
2 // import {heroes} from './data/heroes';
3 // import {heroes} from './data/heroes';
4 import { heroes } from './data/heroes';
5
6
7 // const getHeroeById = (id)⇒{
8 //     return heroes.find((heroes)⇒{
9 //         if(heroes.id === id){
10 //             return true;
11 //         } else {
12 //             return false;
13 //         }
14 //     });
15 // }
16
17
18 // Manera simplificada
19 // Si hay solo un argumento, no se ponen los parentesis
20 const getHeroeById = (id)⇒{
21     return heroes.find((hero) ⇒ hero.id === id)
22 }
23
24 console.log(getHeroeById(2));
25
26 // filter()
27 const getHeroesByOwner = (owner)⇒{
28     return heroes.filter((hero) ⇒ hero.owner === owner)
29 }
30
31 console.log(getHeroesByOwner('Marvel'));
```

10. Múltiples exportaciones y exportaciones por defecto


- Para hacer una exportación por defecto se utiliza: **export default**.
Y para importar esa exportación por defecto se utiliza: **import heroes (sin llaves) from './data/heroes';**.
- Al utilizar la importación por defecto, se puede poner el nombre que queramos: **import nombreQueQueramos ...**
- Se recomienda poner el **export default nombreQueQueramos** al final del archivo.
- Si queremos añadir otra importación que esta en el mismo archivo, a parte de la de por defecto, la segunda importación se pone entre llaves.
Import heroes, {owners} ...
- También se puede simplificar la exportación y poner los objetos en un mismo export, utilizando las llaves: **export{heroes, owners}**.
Y para importarlas, se pone todo entre llaves: **Import {heroes, owners}...**
- Pero si queremos hacer algún objeto por defecto, en el export se pone: **heroes as default**. Y en el import se saca el objeto por defecto de las llaves.

export:



```
1      const heroes = [  
2      {  
3          id: 1,  
4          name: 'Batman',  
5          owner: 'DC'  
6      },  
7      {  
8          id: 2,  
9          name: 'Spiderman',  
10         owner: 'Marvel'  
11     },  
12     {  
13         id: 3,  
14         name: 'Superman',  
15         owner: 'DC'  
16     },  
17     {  
18         id: 4,  
19         name: 'Flash',  
20         owner: 'DC'  
21     },  
22     {  
23         id: 5,  
24         name: 'Wolverine',  
25         owner: 'Marvel'  
26     },  
27 ];  
28  
29 export const owners = ['DC', 'Marvel'];  
30  
31 export default heroes;  
32  
33 // export{  
34 //     heroes as default,  
35 //     owners  
36 // }
```

Import:



```
1  import heroes, {owners} from './data/heroes';
```

11. Promesas

- Para crear una promesa se utiliza: **new Promise()**.
- Las promesas por naturaleza son asincronas, las tareas pueden ejecutarse una detrás de otra, esto significa que podemos indicar que algunas tareas se pasen a segundo plano y esperen su turno para ser reanudadas y ejecutadas.
- Una promesa tiene 3 estados:
 1. **Pending**: estado inicial, ni cumplido ni rechazado.
 2. **Fulfilled**: La operación se cumplió con éxito.
 3. **Rejected**: La operación falló.
- Una promesa pendiente puede cumplirse con un valor o rechazarse con un motivo (error). Cuando ocurre cualquiera de estas dos opciones, se llama a

los controladores asociados en cola por el metodo **then** de una promesa.

- El método .then() toma dos argumentos.
 1. Una funcion de devolucion de llamada(callback) para el caso resuelto de la promesa.
 2. Una funcion de devolucion de llamada para el caso rechazado.
- El metodo then devuelve una promesa que permite encadenar metodos.
- **catch** permite manejar errores de tiempo de ejecucion.
- Cuando una promesa se resuelve entonces se ejecuta la funcion que pasamos al metodo then, pero si la promesa es rechazada entonces se ejecuta la funcion que pasamos a catch, de esta forma podemos controlar el flujo de datos.
- El metodo **finally()** devuelve una promesa. Cuando la promesa se resuelve, sea exitosa o rechazada, la funcion de callback especificada sera ejecutada. El finally() es lo ultimo que se ejecuta.
- El metodo **throw()** lanza una excepcion durante la ejecucion del programa si aparece una situacion extraña.

- Las promesas se crean con un argumento que es el callback.
- Este callback recibe dos argumentos: resolve y reject.

Resolve es otro callback que se va a ejecutar cuando la promesa es exitosa.

Reject se va a ejecutar cuando la promesa falla.

- `setTimeout()`: Permite ejecutar una tarea en cierto tiempo, y recibe un callback.

```
1  import { getHeroeById } from './bases/08-imp-exp';
2
3
4
5  // const promesa = new Promise((resolve, reject)⇒{
6
7  //      setTimeout(()⇒{
8
9  //          const p1 = getHeroeById(2);
10 //          resolve(p1);
11
12 //          reject('No se pudo encontrar el heroe.');
```

12. Fetch API

- La API Fetch proporciona una interfaz javascript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas.
- Con Fetch podemos realizar peticiones. HTTP asincronas utilizando promesas y de forma mas sencilla.
- La forma de realizar una peticion es llamar al metodo `fetch()` y pasarle por parametro la URL de la peticion a realizar.
- `fetch()` devuelve una promesa que sera aceptada cuando reciba una respuesta y solo sera rechazada si hay un fallo de red o si por alguna razon no se pudo completar la peticion.
- El modo mas habitual de manejar las promesas es. Utilizando `.then()`.
- Al metodo `.then()` se le pasa una funcion callback donde su parametro `response` es el objeto de respuesta de la peticion que hemos realizado, y en su interior realizamos la logica que queramos hacer con la respuesta a nuestra peticion.
- `res.json()` se utiliza para obtener los datos que necesitamos del objeto de respuesta. Devuelve una promesa.



```
1  const apiKey = 'nSbUXA56oAqM0VfChA9xcE0mfuc50bAo';
2
3  const peticion = fetch(`https://api.giphy.com/v1/gifs/random?api_key=${apiKey}`);
4
5  // Como el fetch() retorna una promesa, podemos utilizar el then
6  peticion
7  .then((res)⇒res.json())
8  .then(({data})⇒{
9      const {url} = data.images.original;
10
11
12      const img = document.createElement('img');
13      img.src = url;
14
15      document.body.append(img);
16  })
17  .catch(console.warn);
```

13. Async – Await / try...catch

- Async y Await es una manera de trabajar con las promesas.
- La palabra reservada Async se pone delante de la funcion.
- Cuando se llama a una funcion async, esta devuelve una promesa.
- Una funcion async puede contener una expresion await, la cual pausa la ejecucion de la funcion asincrona y espera la resolucio de la promesa pasada y, a continuacion, reanuda la ejecucion de la funcion async y devuelve el valor resuelto.

- La finalidad de las funciones `async/await` es simplificar el comportamiento del uso sincrónico de promesas y realizar algún comportamiento específico en un grupo de promesas.
- El `async` puede ser independiente. Pero el `await` va de la mano con `async`.
- `Await` nos ayuda a trabajar todo el código como si fuera sincrónico.
- El `await` tiene que estar en una función `async`.
- Para manejar errores en el `async` y `await` se utiliza el **`try...catch`**.

```
1  const getImage = async()=>{
2
3    try{
4
5      const apiKey = 'nSbUXA56oAqM0VfChA9xcE0mfuc50bAo';
6      const res = await fetch(`https://api.giphy.com/v1/gifs/random?api_key=${apiKey}`);
7      const {data} = await res.json();
8
9      const {url} = data.images.original;
10
11     const img = document.createElement('img');
12     img.src = url;
13     document.body.append(img);
14
15   } catch(err){
16     // manejo del error
17     console.error(err);
18   }
19 }
20
21 getImage();
```

14. Operador condicional ternario



```
1  const activo = true;
2
3  // let mensaje = '';
4
5  // if(!activo){
6  //     mensaje = 'Activo';
7  // } else {
8  //     mensaje = 'Inactivo';
9  // }
10
11 // Operador condicional ternario
12 // const mensaje = (!activo) ? 'Activo' : 'Inactivo';
13 // const mensaje = (!activo) ? 'Activo' : null;
14 const mensaje = (activo === true) && 'Activo';
15
16 console.log(mensaje);
```