



Programador Web Inicial Front End Developer JavaScript ES 6

Bloques temáticos:

- ES6 – ECMAScript 6
- Javascript – Template Strings
- Javascript – Let && Const
- Javascript – Función Arrow
- Javascript – Deconstructor
- Javascript – Valores por defecto
- Javascript – Import && Export
- Javascript – Rest y Spread
- Javascript – Promises
- Javascript – Clases
- Instalar node js

ES6 – ECMAScript 6

ECMAScript es el estándar que define cómo debe de ser el lenguaje Javascript.

Javascript es interpretado y procesado por multitud de plataformas, entre las que se encuentran los navegadores web, así como NodeJS y otros ámbitos más específicos como el desarrollo de aplicaciones para Windows y otros sistemas operativos. Todos los responsables de cada una de esas tecnologías se encargan de interpretar el lenguaje tal como dice el estándar ECMAScript.

ES5 estuvo con nosotros durante muchos años y a día de hoy es la versión de Javascript más extendida en todo tipo de plataformas. Cuando alguien dice que conoce o usa Javascript es común entender que lo que usa es ES5, el Javascript con mayor índice de compatibilidad. De esto también se entiende que, cuando queremos escribir un código compatible con todos los navegadores o sistemas, lo normal es que ese código sea ES5.

Transpiladores

Conscientes de los problemas de compatibilidad o soporte a ES6 en las distintas plataformas, un desarrollador poco informado podría pensar que:

- Es poco aconsejable usar hoy ES6, debido a la falta de compatibilidad.
- Lo correcto sería esperar a que todos los navegadores se pongan al día para empezar a usar ES6 con todas las garantías.

Afortunadamente, ninguna de esas suposiciones se ajusta a la realidad. Primero porque si ES6 nos aporta diversas ventajas, lo aconsejable es usarlo ya. Luego porque es absurdo quedarse esperando a que todos los navegadores soporten Javascript en la versión ES6. Quizás nunca llegue ese momento de total compatibilidad o posiblemente para entonces hayan sacado nuevas versiones del lenguaje que también deberías usar.

Así que, para facilitar nuestra vida y poder comenzar a usar ES6 en cualquier proyecto, han surgido los transpiladores, una herramienta que ha venido a nuestro kit de desarrollo para quedarse.

Los transpiladores son programas capaces de traducir el código de un lenguaje para otro, o de una versión para otra. Por ejemplo, el código escrito en ES6, traducirlo a ES5. Dicho de otra manera, el código con posibles problemas de compatibilidad, hacerlo compatible con cualquier plataforma.

El transpilador es una herramienta que se usa durante la fase de desarrollo. En esa fase el programador escribe el código y el transpilador lo convierte en un proceso de "traducción/compilación = transpilación". El código transpilado, compatible, es el que realmente se distribuye o se despliega para llevar a producción. Por tanto, todo el trabajo

de traducción del código se queda solo en la etapa de desarrollo y no supone una carga mayor para el sistema donde se va a ejecutar de cara al público.

Hoy tenemos transpiladores para traducir ES6 a ES5, pero también los hay para traducir de otros lenguajes a Javascript. Quizás hayas oído hablar de TypeScript, o de CoffeeScript o Flow. Son lenguajes que una vez transpilados se convierten en Javascript ES5, compatible con cualquier plataforma.

Javascript – Template Strings

Con ES6 podemos interpolar Strings de una forma más sencilla que como estábamos haciendo hasta ahora. Fíjate en este ejemplo:

```
/*IN ES5*/  
var userName = 'Hello World';  
var message = 'Hey' + userName + ',';  
  
/*IN ES6*/  
let userName = 'Hello World';  
let message = 'Hey ${userName},';
```

En el ejemplo de ES6 utilizamos el **backtick (`)** en lugar de las comillas. Luego colocamos el placeholder `${...}` para identificar el contenido a ser interpretado.

De esta forma evitamos tener que concatenar las variables al string definido con comillas.

Let & Const

En Javascript ES 6 podemos definir una constante con la palabra reservada **const**

```
const PI = 3.141593;  
alert (PI > 3.0);
```

Las constantes no podrán ser redefinidas luego.

Var - Let

El scope (alcance) de **var** abarca toda la función en la cual está definida

```
function helloworld() {  
    for (var x = 0; x < 2; x++) {  
        // x should only be scoped to this block because this is where  
        we have defined x.  
    }  
    // But it turns out that x is available here as well!  
    console.log(x);  
    // 2  
}
```

En el ejemplo vemos que “x” (definida en el for) tiene un alcance a la función, por lo cual al realizar un `console.log(x)` luego del for accedemos al último valor de dicha variable.

Con **let** el alcance de la variable pasa a ser las llaves en la cual está definida:

```
function helloworld() {  
    for (let x = 0; x < 2; x++) {  
        // With the "let" keyword, now x is only accessible in this  
        block.  
    }  
    // x is out of the scope here  
    console.log(x); // x is not defined  
}
```

En este ejemplo vemos que al querer realizar un `console.log(x)` nos arroja “undefined”. Esto se debe a que **let** quedó declarada dentro de las llaves del **for** y no para toda la función.

Tanto **let** como **const** no crean una propiedad global si se utilizan en el nivel superior

Función Arrow

La función **arrow** es una simplificación sintáctica de la función en ES5

```
//ES5
const add = function (num) {
  return num + num;
}
//ES6
const add = (num) => {
  return num + num;
}

//en caso de ser una unica sentencia y sin {} aplica return
const add = (num) => num + num;
//Si recibe un unico parametro no hace falta los ()
const add = num => num + num;
//Si no recibe parametros se debe colocar ()
const add = () => num + num;
```

Vemos en los ejemplos que no debemos colocar la palabra reservada **function**, quedando su declaración de la siguiente manera:

```
const add = () => {} (el = y > forman la "flecha" o "arrow")
```

En los ejemplos podemos ver que en caso de no colocar **{}** y solo tener una única sentencia aplica el **return** de forma implícita

Deconstructor

Es una expresión que permite extraer propiedades de un objeto o ítems de un array:

```
//Objeto
const address = {
  street: 'Pallimon',
  city: 'Kollam',
  state: 'Kerala',
};
//ES5
```

```
var street = address.street;
var city = address.city;
var state = address.state;
//ES6
const { street, city, state } = address;
```

En este caso vemos que podemos acceder y crear una constante “Street”, “city”, “state” relacionada con las propiedades que tienen el mismo nombre en el objeto address

En caso de ser un array:

```
//ES5
var values = ['Hello', 'World'];
var first = values[0];
var last = values[1];
//ES6
const values = ['Hello', 'World'];
const [first, last] = values;
```

La constante “first” tendrá asignado el valor del array en el índice 0, mientras que “last” el valor del array en el índice 1

Valores por defecto

Otra novedad es asignar valores por defecto a las variables que se pasan por parámetros en las funciones. Antes teníamos que comprobar si la variable ya tenía un valor. Ahora con ES6 se la podemos asignar según creemos la función.

```
//ES5
function getUser(name, year) {
  year = (typeof year !== 'undefined') ? year : 2018;
  // reminder of the function.
}
//ES6
function getUser(name, year = 2018) {
  // function body here.
}
```

Como vemos el parámetro “year” cuando no reciba valor tomará por default el valor 2018

Import & Export

Ahora JavaScript se empieza a parecer a lenguajes como Python o Ruby. Llamamos a las funciones desde los propios Scripts, sin tener que importarlos en el HTML, si usamos JavaScript en el navegador.

```
// MyClass.js
class MyClass {
  constructor() { }
}
export default MyClass;
// Main.js
import MyClass from 'MyClass';
```

En el archivo “MyClass.js” tenemos la declaración de la clase y el export de la misma.

En el archivo “Main.js” importamos MyClass (debe tener el export previamente)

En caso de no realizar export **default** (es decir no declarar el artefacto como default) debemos realizar el import con {}:

```
// MyClass.js
class MyClass {
  constructor() { }
}
export MyClass;
// Main.js
import { MyClass } from 'MyClass';
```

Solo se puede declarar un solo artefacto como **default** por módulo.

Rest y Spread

Spread: Propaga los elementos de un array de forma individual

```
function getSum(x, y, z) {
  console.log(x + y + z);
}
let sumArray = [10, 20, 30];
getSum(...sumArray);
```


Podemos utilizarlo para concatenar 2 arrays:

```
var a = [1, 2];  
var b = [3, 4];  
var c = [...a, ...b]  
console.log(c);
```

En este caso “c” tendrá los valores de a y b. Este operador también puede ser utilizados para objetos:

```
let alumno = {  
  nombre: "Leandro",  
  apellido: "Gil"  
}  
let cursoAlumno = { ...alumno, { curso: "php"}}
```

Promises

Una Promise (promesa en castellano) es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona. Una promesa puede ser creada usando su constructor. Sin embargo, la mayoría de la gente son consumidores de promesas ya creadas devueltas desde funciones.

Esencialmente, una promesa es un objeto devuelto al cual enganchas las funciones callback, en vez de pasar funciones callback a una función.

Por ejemplo, en vez de una función del viejo estilo que espera dos funciones callback, y llama a una de ellas en caso de terminación o fallo:

```
function exitoCallback(resultado) {  
  console.log("Tuvo éxito con " + resultado);  
}  
function falloCallback(error) {  
  console.log("Falló con " + error);  
}  
hazAlgo(exitoCallback, falloCallback);
```

Las funciones modernas devuelven una promesa a la que puedes enganchar tus funciones de retorno.

```
let p = new Promise(function (resolve, reject) {
  if (/* condition */) {
    resolve(/* value */);
    // fulfilled successfully
  } else {
    reject(/* reason */);
    // error, rejected
  }
});
p.then((val) => console.log("fulfilled:", val)) //10
  .catch((err) => console.log("rejected:", err));
```

La promesa es un objeto en este caso lo asignamos a la variable “p”. Este objeto recibe como parámetro una función de callback la cual recibe el parámetro **resolve** y **reject**.

En caso de que se resuelva la promesa de forma correcta se retornara **resolve(valor_retorno)** esto indicará que la ejecución fue correcta y retorna el valor deseado. En caso de que ocurra un error o excepción se retornada **reject(valor_error)**, en este caso se indica un error o excepción y la causa como parámetro.

Luego tratamos la promesa (como consumidores) aplicando el método “then” y “catch” Cuando se resuelva la misma de forma correcta (resolve) ejecuta el then y recibe el parámetro devuelto. En caso de error o excepción (reject) ejecuta el catch.

Características:

- Las funciones callback nunca serán llamadas antes de la terminación de la ejecución actual del bucle de eventos de JavaScript.
- Las funciones callback añadidas con **.then** serán llamadas después del éxito o fracaso de la operación asíncrona, como arriba.
- Pueden ser añadidas múltiples funciones callback llamando a **.then** varias veces, para ser ejecutadas independientemente en el orden de inserción.
- Pero el beneficio más inmediato de las promesas es el encadenamiento.

Async / Await

Las incorporaciones más recientes al lenguaje JavaScript, son las funciones `async` y la palabra clave `await`, parte de la edición ECMAScript 2017. Estas características, básicamente, actúan como azúcar sintáctico, haciendo el código asíncrono fácil de escribir y leer más tarde. Hacen que el código asíncrono se parezca más al código síncrono de la vieja escuela, por lo que merece la pena aprenderlo.

Primero tenemos la palabra clave "async", que se coloca delante de la declaración de una función, para convertirla en función "async" (asíncrona). Una función "async", es una función que sabe cómo esperar la posibilidad de que la palabra clave "await" sea utilizada para invocar código asíncrono.

await solo trabaja dentro de las funciones async. Esta puede ser puesta frente a cualquier función async basada en una promesa para pausar tu código en esa línea hasta que se cumpla la promesa, entonces retorna el valor resultante. Mientras tanto, otro código que puede estar esperando una oportunidad para ejecutarse, puede hacerlo.

Ejemplo de promise con async / await:

```
//Promise
fetch('coffee.jpg')
  .then(response => {
    if (!response.ok) {
      throw new Error('HTTP error! status: ${response.status}');
    } else {
      return response.blob();
    }
  }).then(myBlob => {
    let objectURL = URL.createObjectURL(myBlob);
    let image = document.createElement('img');
    image.src = objectURL;
    document.body.appendChild(image);
  })
  .catch(e => {
    console.log('There has been a problem with your fetch operation: ' +
e.message);
  });
//ASync / await
async function myFetch() {
```

```
try {
  let response = await fetch('coffee.jpg');
  if (!response.ok) {
    throw new Error('HTTP error! status: ${response.status}');
  } else {
    let myBlob = await response.blob();
    let objectURL = URL.createObjectURL(myBlob);
    let image = document.createElement('img');
    image.src = objectURL;
    document.body.appendChild(image);
  }
} catch (e) {
  console.log('There has been a problem with your fetch operation: ' +
e.message);
}
}
```

Javascript – Clases

Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objetos que realizamos en el estándar anterior, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como, por ejemplo, herencia.

```
class LibroTecnico extends Libro {
  constructor(tematica, paginas) {
    super(tematica, paginas);
    this.capitulos = [];
    this.precio = " ";
    // ..
  }
  metodo() {
    //..
  }
}
```

This

La variable `this` muchas veces se vuelve un dolor de cabeza. antiguamente teníamos que cachearlo en otra variable ya que solo hace referencia al contexto en el que nos encontremos. Por ejemplo, en el siguiente código si no hacemos `var that = this` dentro de la función `document.addEventListener`, `this` haría referencia a la función que pasamos por Callback y no podríamos llamar a `foo()`

```
//ES3
var obj = {
  foo: function () {...},
  bar: function () {
    var that = this;
    document.addEventListener("click", function (e) {
      that.foo();
    });
  }
}
```

Con ECMAScript5 la cosa cambió un poco, y gracias al método `bind` podíamos indicarle que `this` hace referencia a un contexto y no a otro.

```
//ES5
var obj = {
  foo: function () {...},
  bar: function () {
    document.addEventListener("click", function (e) {
      this.foo();
    }.bind(this));
  }
}
```

Ahora con ES6 y la función Arrow `=>` la cosa es todavía más visual y sencilla.

```
//ES6
var obj = {
```

```
foo: function () {...},  
bar: function () {  
    document.addEventListener("click", (e) => this.foo());  
}  
}
```

Instalar node js

Node.js es un entorno de ejecución JavaScript multiplataforma de código abierto. Ejecuta código JavaScript fuera de un navegador.

Cada navegador tiene un motor JavaScript incorporado para procesar archivos JavaScript contenidos en sitios web. Google Chrome usa el motor V8, que está construido usando C + +. Node.js también utiliza este motor ultrarrápido para interpretar archivos JavaScript.

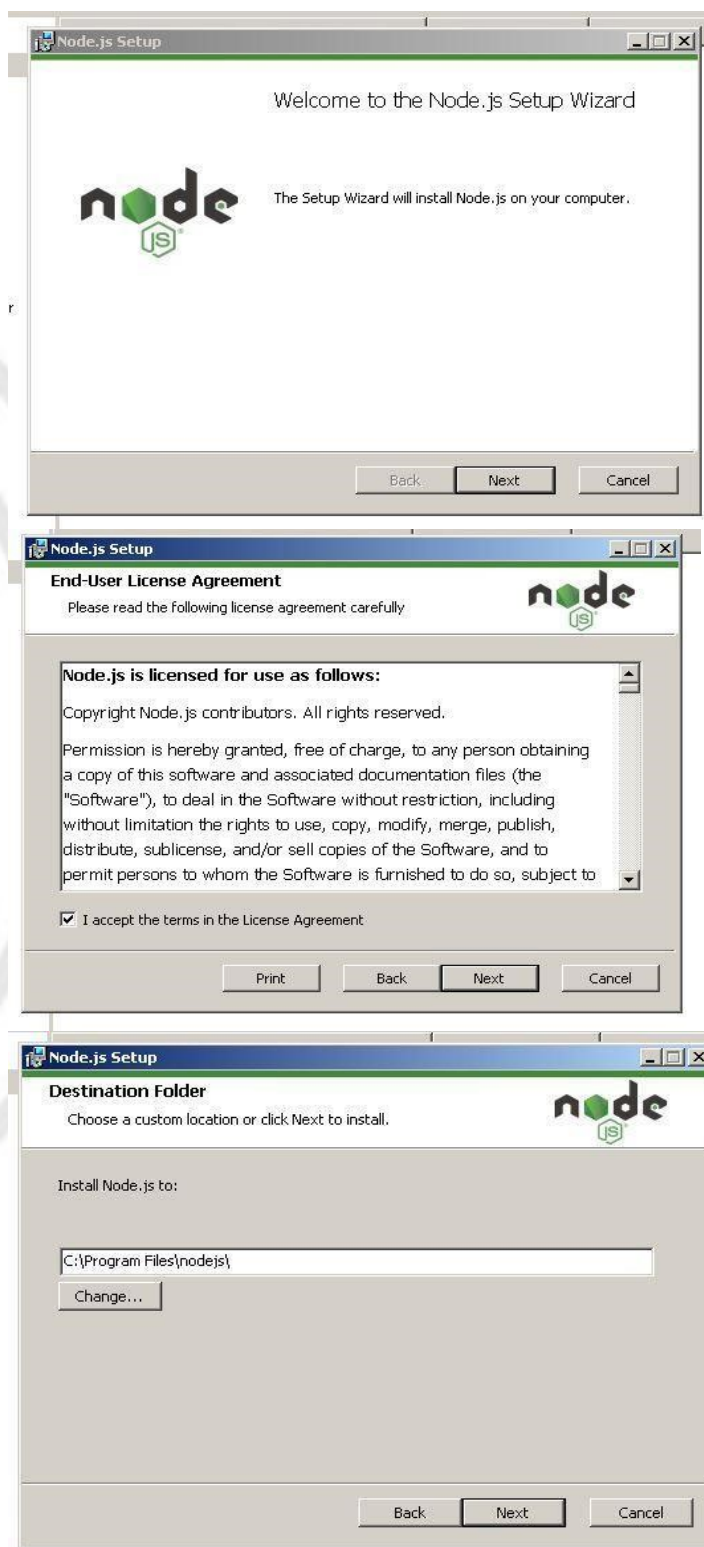
Node.js utiliza un modelo basado en eventos. Esto significa que Node.js espera a que se lleven a cabo ciertos eventos. Luego actúa sobre esos eventos. Los eventos pueden ser cualquier cosa, desde un clic hasta una solicitud HTTP. También podemos declarar nuestros propios eventos personalizados y hacer que Node.js escuche esos eventos.

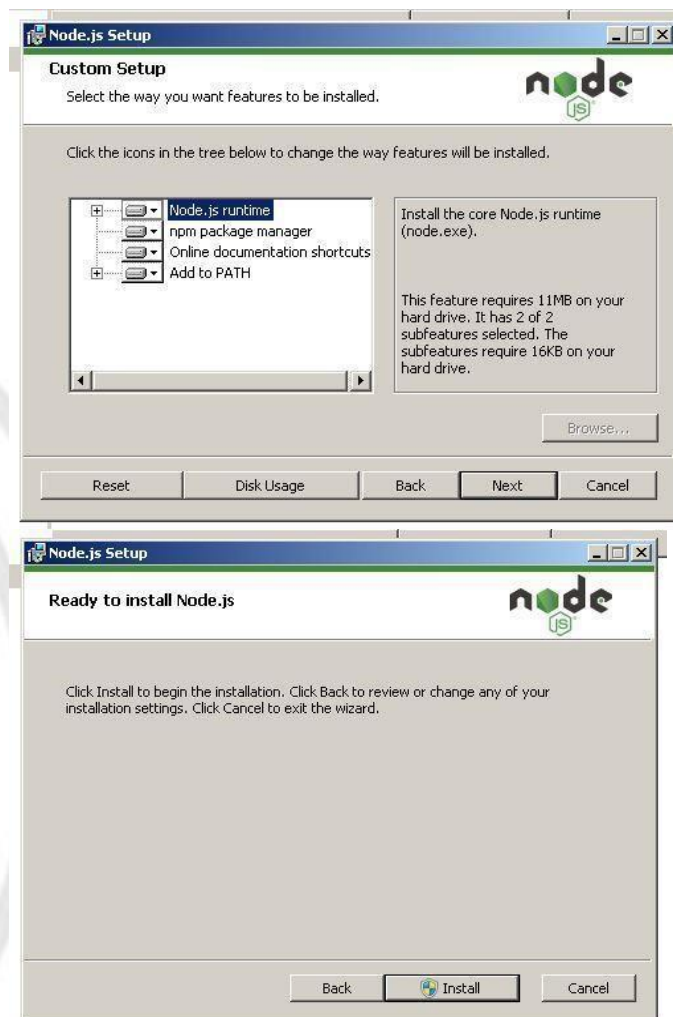
Node.js utiliza un modelo de E/S sin bloqueo. Sabemos que las tareas de E/S toman mucho más tiempo que las tareas de procesamiento. Node.js usa funciones de devolución de llamada para manejar tales solicitudes.

Instalación

Para poder instalar una aplicación de react js desde el CLI, debemos previamente instalar node js.

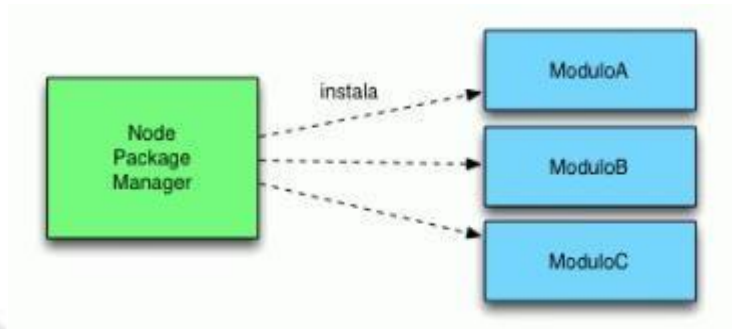
1. Ingresar a: <https://nodejs.org/en/>
2. Descargar la última versión LTS de node Js
3. Ejecutar el archivo descargado y seguir los siguientes pasos:





¿Qué es NPM?

Cuando usamos Node.js rápidamente tenemos que instalar módulos nuevos (librerías) ya que Node al ser un sistema fuertemente modular viene prácticamente vacío. Así que para la mayoría de las operaciones deberemos instalar módulos adicionales. Esta operación se realiza de forma muy sencilla con la herramienta npm (Node Package Manager).



¿Por qué necesita un **Administrador de Paquetes**?

Supongamos que no hubiera administradores de paquetes. En ese caso, tendría que hacer lo siguiente manualmente:

- Encuentre todos los paquetes correctos para su proyecto
- Verifique que los paquetes no tengan vulnerabilidades conocidas
- Descarga los paquetes
- Instálelos en el lugar apropiado
- Mantenga un registro de las nuevas actualizaciones para todos sus paquetes
- Actualice cada paquete cada vez que haya una nueva versión
- Elimina los paquetes que ya no necesites

Un administrador de paquetes es una herramienta que usan los desarrolladores para buscar, descargar, instalar, configurar, actualizar y desinstalar automáticamente los paquetes de una computadora.

NPM (Node Package Manager) e Yarn (Yet Another Resource Negotiator) son dos administradores de paquetes de uso popular.

Un registro de paquetes es una base de datos (almacenamiento) para miles de paquetes (bibliotecas, complementos, marcos o herramientas).

En otras palabras, un registro de paquetes es el lugar desde el que se publican y se instalan los paquetes.

```
tecadmin@linux:~/nodeapp$ npm install cowsay@1.1.0
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN nodeapp@1.0.0 No description
npm WARN nodeapp@1.0.0 No repository field.

+ cowsay@1.1.0
added 4 packages from 3 contributors and audited 4 packages in 0.366s
found 0 vulnerabilities

tecadmin@linux:~/nodeapp$
```

¿Qué es un Módulo en JavaScript?

En términos simples, un módulo es una pieza de código JavaScript reutilizable. Podría ser un .js archivo o un directorio que contenga .js archivos. Puede exportar el contenido de estos archivos y usarlos en otros archivos.

Los módulos ayudan a los desarrolladores a adherirse al principio DRY (Don't Repeat Yourself) en la programación. También ayudan a dividir la lógica compleja en fragmentos pequeños, simples y manejables.

Tipos de módulos de Node

Hay tres tipos principales de módulos de Node con los que trabajará como desarrollador de Node.js. Incluyen lo siguiente.

- Módulos integrados
- Módulos locales
- Módulos de terceros

¿Por qué exportar módulos?

Querrá exportar módulos para poder usarlos en otras partes de su aplicación.

Los módulos pueden servir para diferentes propósitos. Pueden proporcionar utilidades simples para modificar cadenas. Pueden proporcionar métodos para realizar solicitudes de API. O incluso pueden proporcionar constantes y valores primitivos.

Cuando exporta un módulo, puede importarlo a otras partes de sus aplicaciones y consumirlo.

Tipos de exportaciones de archivos en JavaScript

Exportaciones predeterminadas

Aquí se explica cómo realizar una exportación predeterminada en un archivo JavaScript:

```
function getAllUser():  
export default getAllUser
```

Tenga en cuenta que solo puede usar una exportación predeterminada en un archivo JavaScript. También puede exportar antes de la declaración, así:

```
export default function getAllUser():
```

Bibliografía y Webgrafía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

https://www.tutorialspoint.com/es6/es6_syntax.htm

<http://www.desarrolloweb.com/manuales/20/>

<http://www.desarrolloweb.com/articulos/826.php>

<http://www.desarrolloweb.com/articulos/827.php>

<http://www.desarrolloweb.com/articulos/846.php>

<http://www.desarrolloweb.com/articulos/861.php>

<https://carlosazaustre.es/ecmascript-6-el-nuevo-estandar-de-javascript/> <http://es6-features.org/>