

Javascript

El siguiente texto es una guía teórica-práctica sobre javascript escrita para la materia de desarrollo de aplicaciones en la nube de la carrera de ingeniería en sistemas de información - UTN FRSF

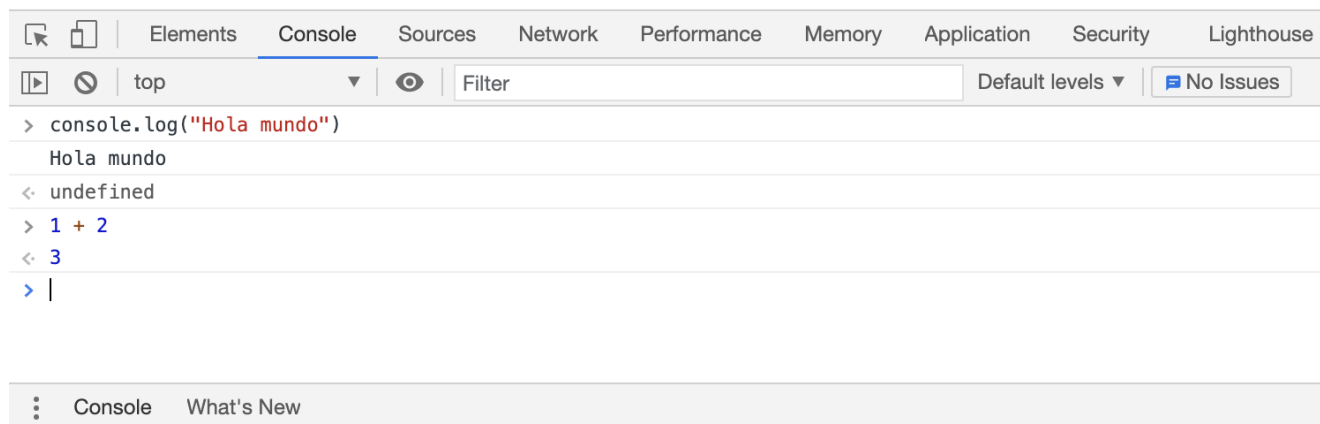
[Leer o descargar la versión en PDF](#)



Intro

Javascript es un lenguaje de programación que ejecuta en todos los browsers.

Por ejemplo, en Chrome podemos utilizar las herramientas de desarrollador para abrir una consola presionando Option + ⌘ +

J (macOS), o Shift + CTRL + J (Windows/Linux)



Dentro la consola de Chrome ingresamos expresiones de js como nuestro input  que luego se ejecutan y su resultado se imprime como output  en la terminal misma.

TIP: La consola de Chrome es útil para realizar pruebas mientras estamos aprendiendo, ya que nos permite ver resultados inmediatos y sus herramientas de autocompletado e inspección son potentes.

Para poder escribir nuestro código y luego poder ejecutarlo vamos a necesitar un poco de ayuda de NodeJS, un entorno de ejecución de javascript, el cual está basado en el motor que utiliza Chrome.

Utilizando Node es que podemos ejecutar javascript en todos lados, incluso en servidores.

Para instalar node basta con dirigirnos a <https://nodejs.org/es/> para descargar e instalar la

última version **LTS**. Otra alternativa es utilizar **nvm** (node version manager), para cuando necesitamos administrar múltiples versiones, el cual se encuentra disponible para [linux/mac](#) y [windows](#)

Para comprobar si se instaló correctamente abrimos alguna terminal y ejecutamos

```
$ node -v
```

Deberíamos ver una respuesta indicando el número de versión de node indicándonos que todo salió bien

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
AR0FVFCH1NHL415:javascript leandamarill$ node -v  
v14.17.0  
AR0FVFCH1NHL415:javascript leandamarill$
```

Con node instalado solo nos falta un editor de texto, vscode es una buena opción, pero también existen otros. Ahora solo nos falta abrir nuestro editor de texto de preferencia y estamos listos para empezar a escribir javascript.

0 - Hola mundo

Creando un archivo de extensión **.js** podemos empezar a escribir código, lo que escribamos aquí se ejecutara de arriba hacia abajo hasta finalizar.

Al escribir

```
console.log("Hola mundo");
```

Suponiendo que nuestro archivo se llama **holamundo.js** podemos abrir una terminal en el mismo directorio que el archivo y ejecutar

```
node holamundo.js
```

Y observamos

```
AR0FVCH1NHL415:resoluciones leandamarill$ node holamundo.js  
Hola mundo  
AR0FVCH1NHL415:resoluciones leandamarill$
```

Sintaxis básica de javascript

Esta sección es un pequeño resumen de sintaxis moderna de javascript (ES6), no es necesario revisar, pero la idea es que resulte familiar a lo ya conocido por el alumno o que sirva de punto de partida para explorar

Declarando variables

Existen 3 formas de hacerlo

```
var foo = "asd"; // Variable mutable de scope de función o global (No  
recomendado)  
let foo = "asd"; // Variable mutable de scope de bloque  
const foo = "asd"; // Variable inmutable de scope de bloque
```

Para ver la diferencia de porque es recomendable `let` sobre `var` pueden leer este link de [let vs var](#)

En javascript no declaramos el tipo de dato al momento de crear variables, de hecho aunque no es para nada recomendable, el tipo de dato de la variable puede modificarse en tiempo de ejecución.

El siguiente código es javascript válido y nos muestra los tipos de dato más comunes

```
let miVariable;  
console.log("Type: ", typeof miVariable); // Type: undefined  
  
miVariable = "asd";  
console.log("Type: ", typeof miVariable); // Type: string  
  
miVariable = 100;  
console.log("Type: ", typeof miVariable); // Type: number  
  
miVariable = 0.212312;  
console.log("Type: ", typeof miVariable); // Type: number (no hay distinción  
de enteros)
```

```

miVariable = true;
console.log("Type: ", typeof miVariable); // Type: boolean

// Muchas cosas en javascript son 'object'
miVariable = [1, 2, 3]; // Arrays
console.log("Type: ", typeof miVariable); // Type: object

miVariable = {
  id: 1,
  nombre: "juan",
  apellido: "carlos",
  habilitado: true
}; // Objetos
console.log("Type: ", typeof miVariable); // Type: object

miVariable = null; // Null
console.log("Type: ", typeof miVariable); // Type: object

// También podemos declarar variables definidas

let miVariableDefinida = miVariable;
const miConstanteDefinida = miVariable;

```

Bloques de control

Javascript también soporta los bloques de control similares a los que ya conocemos de otros lenguajes

- [switch](#)
- [while](#)
- [for](#)
- [if...else](#)

Comparando variables

```

// Operador '==' - Igualdad 'de valor' (no recomendable)
if (100 == "100") {
  console.log("Iguales (?) :thonk:");
}

// Operador '===' - Igualdad de valor y de tipo
if (100 === "100") {
  console.log("Esto no se ejecuta");
}

```

```
}

if (100 === 100.0) { // Recordemos que ambos son "number"
    console.log("Esto se ejecuta");
}
```

Valores Truthy & Falsy de variables

Los valores en javascript se consideran verdaderos o falsos dependiendo de su contenido al momento de ser evaluado en un “contexto booleano”, esto es conveniente para chequeos.

truthy	falsy
true	false
'false' (el string false)	0
'0' (el string 0)	" (string vacío)
() (función vacía)	null
[] (array vacío)	undefined
{ } (objeto vacío)	NaN (Not a number)
Los demás valores	

Algunos ejemplos

```
let nombre = undefined;

// Usamos !! para negar dos veces e imprimir el valor booleano de la
// variable
// también podríamos haber escrito
//
// if(nombre){
//     console.log("true")
// }
// console.log("false")
// }

console.log(!nombre); // false

nombre = null;
console.log(!nombre); // false
```

```
nombre = "";  
console.log(!!nombre); // false  
  
nombre = "Ricardo";  
console.log(!!nombre); // true
```

El objeto global 'process' de Node

Node nos proporciona el objeto global [process](#) para obtener información e interactuar con el proceso actual.

Como `process` es un objeto global solo basta referenciarlo en nuestro programa para poder utilizarlo, sin embargo nuestro editor de texto a veces puede no reconocerlo. Si quisiéramos ayudar al autocompletado del editor podríamos escribir:

```
const process = require('process');
```

Utilizando process podemos recuperar los argumentos de consola accediendo a `process.argv`

```
resoluciones >  nodeprocess.js
```

```
1 console.log(`process.argv: ${process.argv}`);  
2
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
AR0FVFCH1NHL415:resoluciones leandamarill$ node nodeprocess.js primero segundo tercero  
process.argv: [  
  '/usr/local/Cellar/node/15.0.1/bin/node',  
  '/Users/leandamarill/Documents/javascript/resoluciones/nodeprocess.js',  
  'primero',  
  'segundo',  
  'tercero'  
]  
AR0FVFCH1NHL415:resoluciones leandamarill$
```

Ejercicios

Si querés resolver los ejercicios pensados para esta sección podés dirigirte a la hoja de [ejercicios-00](#)

1 - Funciones y asincronía en javascript

Definición

Las funciones existen en casi todos los lenguajes de programación modernos, nos permiten encapsular comportamiento y reutilizar código.

```
function saludador(nombre) {  
    return `Hola ${nombre}!`;  
}  
  
const greeter = function (name) {  
    return `Hello ${name}!`;  
}  
  
// Invocaciones  
saludador("Ricardo");  
greeter("Richard");
```

Desde **ES6** también podemos declarar nuestras funciones con la sintaxis de *funciones flecha*

```
const saludador = (nombre) => {  
    return `Hola ${nombre}!`;  
}  
  
// Podemos comprimir quitando el return y los {}  
const saludador = (nombre) => `Hola ${nombre}!`;  
  
// Si solo recibe un parámetro también podemos quitar el ()  
const saludador = nombre => `Hola ${nombre}!`;  
  
// Si no recibe parametros tenemos que usar ()  
const saludador = () => `Hola Extraño!`;
```

Funciones como first class citizens

En javascript las funciones van un paso más allá, se consideran “first-class citizen” (o ciudadanos de primera clase),

esto quiere decir que son tratadas como cualquier otra variable. Por ejemplo, una función puede recibir otra función como parámetro, retornar una función y que esta sea luego asignada a una variable.

```
// Asignando funciones a variables
const f1 = function foo() {
  console.log("foo");
};
const f2 = function () {
  console.log("anónima");
};
const f3 = () => console.log("anónima");

// Declarando funciones y ejecutándolas inmediatamente (se asigna el resultado)
const hola = function () {
  return "hola"
}()
const chau = (() => "chau")() // En este caso tenemos que agregar paréntesis extras

// Funciones como parámetros
const sumar = (a, b) => a + b;
const operar = (operacion, n1, n2) => console.log(operacion(n1, n2))
operar(sumar, 1, 2) // Imprime: 3

// Funciones que retornan funciones (también llamadas thunks)
const restar = (n1) => (n2) => console.log(n1 - n2)
restar(10)(8) // Imprime: 2
```

Clausuras

Un detalle a mencionar es que javascript, a diferencia de lenguajes como java, no ofrece una manera nativa de marcar funciones “privadas”. Pero nos permite simular ese comportamiento mediante [clausuras](#)

Una clausura es una función que permite acceder al ámbito de una función exterior desde una función interior. En JavaScript, las clausuras se crean cada vez que una función es creada.

Veamos este ejemplo


```
const saludar = (() => {
  const componerNombre = (n, a) => `${n} ${a}`;

  return (nombre, apellido) => {
    const nombreApellido = componerNombre(nombre, apellido);
    console.log(`Hola ${nombreApellido}`)
  };
})();

saludar("Ricardo", "Sanchez");
```

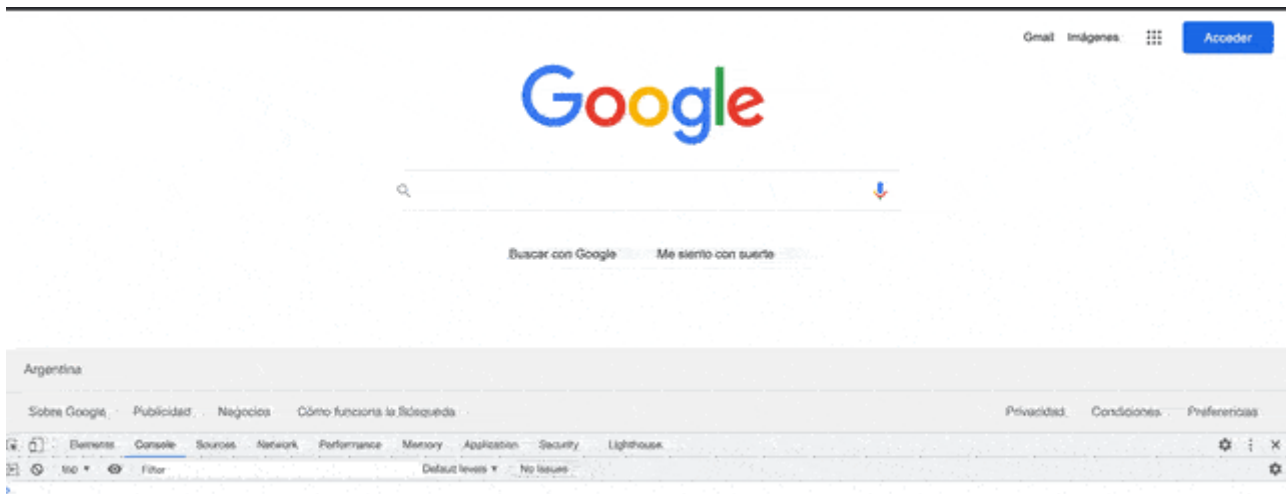
Estamos definiendo una función y luego ejecutándola inmediatamente, esta retorna otra función que toma `nombre` y `apellido` de parámetro y se asigna a la variable `saludar`. Ahora, desde la función `saludar` estamos haciendo referencia a la función interna `componerNombre` pero esta no puede ser accedida de forma externa, logrando un resultado similar al de las funciones privadas.

Asincronía en Javascript

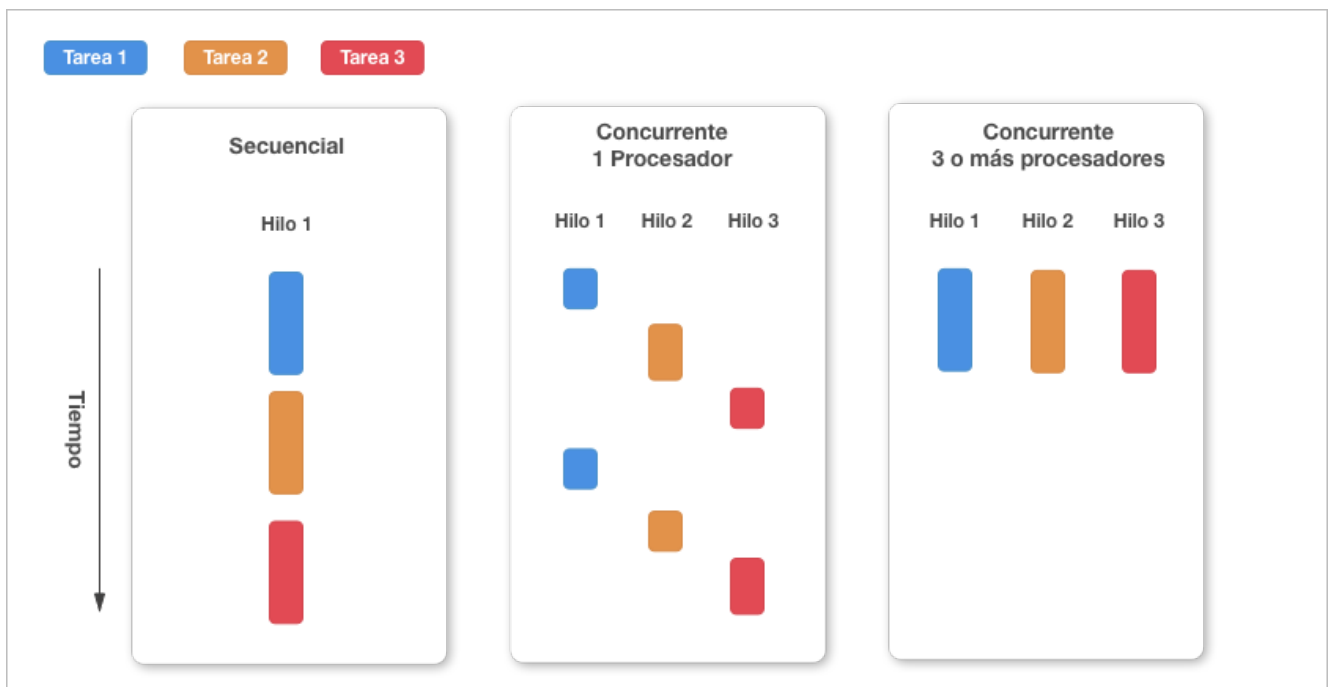
El runtime de Javascript solo puede ejecutar una sola cosa a la vez, es un lenguaje de programación single-threaded (un hilo) y por lo tanto solo puede procesarse de a una instrucción.

Esto nos impone una limitante al tratar de realizar operaciones que se extienden en el tiempo como consultar apis. Si nuestra función demora estaremos bloqueando el hilo principal y por lo tanto nada más podrá ejecutarse.

Por ejemplo, desde la consola de desarrollador de chrome podemos ejecutar `while(true){}` y de esta manera tildar completamente la tab, ya que no desocupamos el hilo para que la página siga funcionando.



Ahora, acabamos de decir que nuestro javascript ejecuta en un único hilo, por lo que no podemos recurrir a la programación **paralela** para solucionar nuestros problemas pero sí a la programación **concurrente**



Pero nosotros no ejecutamos solo javascript, ejecutamos javascript en un navegador o sobre nodejs y estos poseen un modelo de concurrencia basado en un "loop de eventos". Si bien este modelo es diferente al que estamos acostumbrados en lenguajes como Java y existen varios recursos dedicados a [explicar su funcionamiento](#) de momento nos alcanza con saber que es la estrategia para resolver tareas concurrentes.

Modelando la asincronía: Promesas con then/catch

En Javascript, una `Promise` es un objeto que *PUEDE* producir un valor en algún momento futuro. Se utilizan para representar la terminación o el fracaso de una operación asíncrona. Este objeto admite 3 estados: fulfilled (completada), rejected (rechazada), o pending (sin completar) y expone el método `then` que se llama cuando la promesa resuelve y `catch` cuando se rechaza, ambos métodos reciben callbacks como parámetros para manejar los datos o el error recibido.

Dado que la mayoría de las personas consumen `promises` ya creadas, empezaremos primero por cómo consumirlas.

Supongamos que tenemos un método `buscarDatosDeUsuario` que utiliza una api de internet para recuperar los datos de usuario, esta tarea no es instantánea, ya que una consulta de red puede demorar varios segundos. Si nuestra función devuelve un objeto promise entonces podemos modelarlo de la siguiente manera

```
buscarDatosDeUsuario()  
  .then(usuario => console.log(`Hola ${usuario.nombre}!`))  
  .catch(error => console.log(`No pudimos recuperar los datos. Razón: ${error}`))
```

El método `then` aplicado a una función devuelve otra promesa, esto nos permite encadenarlas. Siguiendo con nuestro ejemplo, supongamos que luego de recuperar los datos de un usuario tenemos que consultar en otra api las deudas que este posee, usando promesas encadenadas podríamos modelarlo de la siguiente manera

```
buscarDatosDeUsuario()  
  .then(usuario => buscarDeudasPendientes(usuario.cuil))  
  .then(deuda => console.log(deuda.monto == 0 ? "No posee deudas" : `Debe ${deuda.monto}!`))  
  .catch(error => console.log(`No pudimos recuperar los datos. Razón: ${error}`))
```

En un mundo sin promesas este tipo de operaciones encadenadas tendría que utilizar callbacks, por ejemplo

```
const falloCallback = error => console.log(`No pudimos recuperar los datos. Razón: ${error}`);
```

```
// En cada una pasamos los valores necesarios y
// luego un callback de éxito y uno de fallo
buscarDatosDeUsuario(usuario => {
  buscarDeudasPendientes(usuario.cuil, deuda => {
    if (deuda.monto == 0) {
      console.log("No posee deudas");
    } else {
      console.log(`Debe ${deuda.monto}!`);
    }
  }, falloCallback);
}, falloCallback);
```

Esta estrategia tiene pobre legibilidad y empeora con cada tarea extra, ya que cada callback incrementa el nivel de anidamiento del código, pero cuenta con una ventaja. Debido a que los callbacks son funciones dentro de otras funciones podríamos usar clausuras para acceder a los resultados de las operaciones previas, cosa que utilizando then/catch no sería tan sencillo.

```
const falloCallback = error => console.log(`No pudimos recuperar los datos.
Razon: ${error}`);

// En cada una pasamos los valores necesarios y
// luego un callback de éxito y uno de fallo
buscarDatosDeUsuario(usuario => {
  buscarDeudasPendientes(usuario.cuil, deuda => {
    if (deuda.monto == 0) {
      console.log(`No posee deudas. Felicitaciones
${usuario.nombre}`);
    } else {
      console.log(`Debe ${deuda.monto}! Muy mal ${usuario.nombre}`);
    }
  }, falloCallback);
}, falloCallback);
```

Esto no es mucho una justificación para usarlos, ya que desde ES6 existe una estrategia que soluciona este problema y además nos facilita trabajar con promesas en general.

Promesas con async await

Como mencionamos anteriormente las promesas son una manera de modelar el código asíncrono, desde es6 se incluyen las keywords `async` y `await` para poder escribir código asíncrono que utiliza promesas como si fuera código síncrono.

```
const tarea = async () => {
  try {
    const usuario = await buscarDatosDeUsuario();
    const deuda = await buscarDeudasPendientes(usuario.cuil);

    if (deuda.monto == 0) {
      console.log(`No posee deudas. Felicitaciones
${usuario.nombre}`);
    } else {
      console.log(`Debe ${deuda.monto}! Muy mal ${usuario.nombre}`);
    }
  } catch (error) {
    console.log(`No pudimos recuperar los datos. Razon: ${error}`);
  }
};
tarea();
```

Si no fuera por las palabras reservadas `async` y `await` parecería que `buscarDatosDeUsuario` y `buscarDeudasPendientes` son funciones síncronas cuando en realidad entre líneas pueden pasar varios segundos hasta que se ejecuten. También el manejo de errores se movió a un bloque `try/catch` como si fuera un manejo de excepciones.

La keyword `await` funciona haciendo que, de alguna manera, la ejecución del código “espere” hasta que la promesa se resuelva y luego continúe retornando el resultado de la promesa.

Sin entrar mucho en los detalles de cómo funciona podemos decir que `await` permite que javascript continúe ejecutando otros trabajos mientras la ejecución de nuestro código está suspendido esperando a que se complete la promesa.

TIP: No usar el bloque `finally` de un try/catch cuando se utiliza await en promesas. En caso de hacerlo veremos como el bloque finally se ejecutará varias veces, lo cual probablemente no sea el resultado esperado.

Si bien esta sintaxis es mucho más elegante y sencilla de leer que `.then(...)` tiene una restricción muy importante de recordar

No se puede utilizar `await` en funciones regulares. Solo en funciones asíncronas.

Definir una función como asíncrona es tan sencillo como agregar la keyword `async` delante de nuestra declaración de función. Es por esto que en nuestro ejemplo definimos la función `tarea`.

Si estamos utilizando NodeJS a partir de la v14.8 podemos “saltarnos” esta restricción definiendo nuestros scripts como **módulos**, la forma más sencilla para archivos individuales es cambiando la extensión de `.js` a `.mjs`

Importante: Si en tu código estabas usando importaciones con la sintaxis

```
const cosa = require(cosa)
```

Al cambiar la extensión a `.mjs` vas a tener que reemplazarlo por

```
import cosa from 'cosa'
```

Utilizando el código del ejemplo que veníamos siguiendo, nuestro resultado final sería

```
try {
  const usuario = await buscarDatosDeUsuario();
  const deuda = await buscarDeudasPendientes(usuario.cuil);

  if (deuda.monto == 0) {
    console.log(`No posee deudas. Felicitaciones ${usuario.nombre}`);
  } else {
    console.log(`Debe ${deuda.monto}! Muy mal ${usuario.nombre}`);
  }
} catch (error) {
  console.log(`No pudimos recuperar los datos. Razón: ${error}`);
}
```

2 - Manipulando objetos y arrays

Objetos

Los objetos en javascript son similares a los mapas de otros lenguajes de programación, no son más que una collection de propiedades que asocian un nombre (o clave) y un valor. El valor de una propiedad puede ser una función, en cuyo caso la propiedad es conocida como un método.

```
// Creando objeto con keyword new (no se suele utilizar)
const persona1 = new Object()

// Creando objeto con notación de literal
const persona2 = {}

// Agregando propiedades con notación '.'
persona2.nombre = "Ricardo"

// Agregando propiedades con notación de map
// Se suele utilizar solo cuando sabemos el nombre de la key durante la ejecución
persona2["nombre"] = "Ricardo"

// Accediendo a los valores de un objeto
console.log(persona2.nombre) // Imprime: "Ricardo"
console.log(persona2["nombre"]) // Imprime: "Ricardo"

// Si tratamos de acceder a los valores de un objeto que no existe
console.log(personaQueNoExiste.nombre) // Imprime: "Uncaught TypeError: Cannot read property 'nombre' of undefined"

//Accediendo a un valores indefinido de un objeto definido
console.log(persona2.apellido) // Imprime: "undefined"

// Usando la notación de literal podemos asignar valores iniciales
const persona = {
  nombre: "Ricardo",
  apellido: "Sanchez",
  cuil: 2038888885,
  intereses: ["programar en javascript", "quejarse de javascript"],
  prepararCafe: async () => {
    // TODO: Solucionar coffee leaks...
  }
}
```

```
}
```

Operaciones sobre arrays

Los arrays de javascript proporcionan varios métodos para efectuar operaciones de recorrido y de mutación. Tanto la longitud como el tipo de los elementos de un *array* son variables.

```
// Declarando un array
const frutas = ["Manzana", "Ananá", "Banana", "Pera"];

// Podemos tener cualquier tipo de dato dentro de un array
const caos = ["Hola", { nombre: "Ricardo" }, 123123, false, ["Ananá", "Banana"]];

// Accediendo elementos
console.log(frutas[0]); // Imprime: "Manzana"

// Logitud de un array - propiedad '.length'
console.log(frutas.length); // Imprime: 4

// Iterando sobre elementos de un array
frutas.forEach((elemento, indice, array) => {
    console.log(`Fruta: ${elemento} - Indice: ${indice}`);
});

// Copiar un array
const copiaDeFruta = frutas.slice()

// Buscar un elemento
const numeros = [5, 12, 8, 130, 44];
numeros.find(n => n > 10) // Retorna: 12
```

Los arrays de javascript incluyen una api muy potente para operar sobre ellos permitiendo filtrar, ordenar, mapear a otros valores o preguntar sobre los contenidos. Estas operaciones tienen la característica de no modificar el array original, un patrón bastante prevalente en javascript.

Por ejemplo, en los métodos como `filter` o `map` que retornan un array, este resultado es nuevo array, el original sigue con todos sus valores intactos. En un principio puede parecer una forma poco

performante de operar, desde lo teórico claramente lo es por el overhead extra de crear nuevos objetos, pero desde lo práctico es [debatible](#) el impacto de performance, difícilmente lo notaremos en nuestras aplicaciones y los beneficios de eliminar efectos secundarios suelen ser mayores.

Map

Devuelve un nuevo array que contiene el resultado de llamar a la función pasada como parámetro a todos los elementos del array sobre el que se invoca.

```
[1, 2, 3].map(x => x * 2) // Retorna: [2, 4, 6]
```

Reduce

Aplica la función pasada como parámetro a un *acumulador* y a cada valor del *array*, que se recorre de izquierda a derecha, para reducirlo a un único valor.

```
const miArray = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

miArray.reduce(reducer) // Retorna: 10

// También admite el valor inicial del accumulator como segundo parámetro
miArray.reduce(reducer, 1) // Retorna: 11
```

Filter

Devuelve un nuevo *array* que contiene todos los elementos de aquel que cumplan el predicado que se le pasa como parámetro.

```
const palabras = ["hola", "supercalifragilístico", "oso", "javascript"];

palabras.filter(p => p.length < 4) // Retorna: ["oso"]
```

Some

Devuelve `true` si al menos un elemento del *array* cumple con el predicado que se pasa como parámetro.

```
const numeros = [1, 2, 3, 4, 5];

const esPar = n => n % 2 === 0;

numeros.some(esPar); // Retorna: true
```

Every

Devuelve `true` si todos los elementos del *array* cumplen el predicado que recibe como parámetro.

```
const numeros = [1, 2, 3, 4, 5];
const pares = [4, 2, 6, 8, 10];

const esPar = n => n % 2 === 0;

numeros.every(esPar); // Retorna: false
pares.every(esPar); // Retorna: true
```

Operaciones que modifican el array original

En javascript tratamos de no modificar variables, ya que esta suele ser la fuente de muchos errores, por eso se trata de utilizar `const` donde sea posible y transformar los objetos en nuevos en lugar de mutar los existentes.

Sin embargo, Array también incluye algunas apis que modifican el array y las comentamos en esta sección.

```
// Añadir un elemento al final de un Array
frutas.push('Naranja'); // retorna la nueva longitud del array

// Eliminar el último elemento del Array
frutas.pop(); // retorna la nueva longitud del array

// Añadir un elemento al principio de un Array
frutas.unshift('Sandía'); // retorna la nueva longitud del array

// Eliminar el primer elemento de un Array
```

```
frutas.shift(); // retorna la nueva longitud del array

// Ordenar un array
const nums = [1, 20, 3];
nums.sort()
console.log(nums) // Imprime: [20, 3, 1]

// Invertir un array
const nombres = ["Juan", "Carlos", "Ruben"];
nombres.reverse();
console.log(nombres) // Imprime: ["Ruben", "Carlos", "Juan"]
```

Desestructuración y Spreads

La sintaxis de **desestructuración** y el operador **...** (spread) fueron agregadas en ES6 para facilitar operar con arrays y objetos.

La **desestructuración** es una expresión de JavaScript que permite “desempacar” valores de arreglos o propiedades de objetos en distintas variables.

La **sintaxis spread** permite a un elemento iterable tal como un arreglo o cadena sea “expandido” en lugares donde cero o más argumentos (para llamadas de función) o elementos (para Array literales) son esperados, o a un objeto ser expandido en lugares donde cero o más pares de valores clave (para literales Tipo Objeto) son esperados

La definición es bastante rebuscada y más que tratar de dar una más clara es mejor ver la utilidad que traen consigo

Operaciones con Arrays

```
// Asignando a: 10 y b: 20
const numeros = [10, 20];
let [a, b] = numeros;

console.log(a) // Imprime: 10
console.log(b) // Imprime: 20

// Intercambiando los valores de a y b
[a, b] = [b, a]
```

```
console.log(a) // Imprime: 20
console.log(b) // Imprime: 10

// Seleccionando los primeros 2 valores de un array usando spread para
copiar el resto de valores
const muchosNumeros = [10, 20, 30, 40, 50, 60];
const [primero, segundo, ...resto] = muchosNumeros

console.log(primero) // Imprime: 10
console.log(segundo) // Imprime: 20
console.log(resto) // Imprime: [30, 40, 50, 60]

// Copiando un array
const nuevoArray = [...muchosNumeros]

console.log(nuevoArray) // Imprime: [10, 20, 30, 40, 50, 60]
console.log(muchosNumeros) // Imprime: [10, 20, 30, 40, 50, 60]

nuevoArray.pop() // Removemos el último elemento de la copia

console.log(nuevoArray) // Imprime: [10, 20, 30, 40, 50]
console.log(muchosNumeros) // Imprime: [10, 20, 30, 40, 50, 60]

// Agregando elementos al principio un array
const numeros = [10, 20];
const masNumeros = [1, ...numeros] // Resultado: [1, 10, 20]

// Agregando elementos al final un array
const numeros = [10, 20];
const masNumeros = [...numeros, 1] // Resultado: [10, 20, 1]

// Concatenando arrays
const array1 = [1, 2];
const array2 = [5, 4];
const array3 = [8, 9, 10];

const resultado = [...array1, ...array3, ...array2]; // Resultado: [1, 2, 8,
9, 10, 5, 4]
```

Operaciones con Objetos

```
const persona = {
  nombre: "Ricardo",
```

```

    apellido: "Sanchez",
    cuil: 20388888885,
    intereses: ["programar en javascript", "quejarse de javascript"],
  };

  // Copiando un objeto
  const personaCopia = { ...persona };

  // Extrayendo keys del objeto
  const { nombre, intereses } = persona;

  console.log(nombre); // Imprime: "Ricardo"
  console.log(intereses); // Imprime: ["programar en javascript", "quejarse de javascript"]

  // Crear un nuevo objeto modificando keys pero copiando los demas valores
  const personaNueva = { ...persona, nombre: "Juan" };

  console.log(persona.nombre); // Imprime: "Ricardo"
  console.log(persona.apellido); // Imprime: "Sanchez"
  console.log(personaNueva.nombre); // Imprime: "Juan"
  console.log(personaNueva.apellido); // Imprime: "Sanchez"

  // Crear un nuevo objeto agregando keys pero copiando los demás valores
  const personaNueva = { ...persona, copado: true };

  console.log(!personaNueva.copado) // Imprime: true
  console.log(!persona.copado) // Imprime: false

```

3 - Mi primer proyecto en NodeJS

Durante esta guía venimos utilizando node para lanzar nuestros scripts de javascript, ahora veremos cómo escalar esta solución a un proyecto que cuenta con múltiples archivos y necesita utilizar librerías de terceros.

NPM

Hasta ahora solamente hemos usado las herramientas provistas por NodeJS, pero nos estamos perdiendo de una de las ventajas más grandes de javascript, su extensa comunidad y la gran cantidad de librerías de terceros a nuestra disposición.

Para ayudarnos a encontrar y administrar estas dependencias NodeJS, por defecto, incluye un sistema de gestión de paquetes llamado NPM (Node Package Manager)

Solo teniendo node instalado, desde una terminal, podemos acceder a él

```
$ npm -v
```

Todos nuestros ejercicios implicaban crear un archivo con extensión `.js` y ejecutarlo mediante el comando `node archivo.js`. Si bien puede ser suficiente para scripts sencillos, para desarrollar aplicaciones complejas que necesitan de librerías externas en versiones específicas empezamos a tener problemas.

Utilizando npm podemos generar una estructura de proyecto para nuestra aplicación mediante el comando `npm init`

```
$ mkdir mi-proyecto && cd mi-proyecto # Creamos una carpeta y navegamos dentro
$ npm init
```

Al ejecutar el comando `init` la utilidad de npm nos guiará paso a paso en la creación de un archivo `package.json`.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  1: bash

AR0FVFCH1NHL415:resoluciones leandamarill$ mkdir mi-proyecto
AR0FVFCH1NHL415:resoluciones leandamarill$ cd mi-proyecto/
AR0FVFCH1NHL415:mi-proyecto leandamarill$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (mi-proyecto)
version: (1.0.0)
description: Mi primer proyecto usando npm
entry point: (index.js)
test command:
git repository:
keywords:
author: Leandro Amarillo
license: (ISC)
About to write to /Users/leandamarill/Documents/javascript/resoluciones/mi-proyecto/package.json:

{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "description": "Mi primer proyecto usando npm",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Leandro Amarillo",
  "license": "ISC"
}

Is this OK? (yes)
AR0FVFCH1NHL415:mi-proyecto leandamarill$
```

En él queda descrito el nombre del proyecto, la versión y otras características de las cuales, sin duda la más importante, es el registro de las dependencias necesarias para ejecutar nuestro código.

Creando un script de inicialización de nuestro proyecto

Si revisamos nuestro package.json vemos que se autogeneró un tag `main` con el siguiente valor que simboliza el punto de entrada de nuestra aplicación

```
{
  // ...
  "main": "index.js"
}
```

Importante:

`npm init` no generó ningún archivo index.js así que vas a tener que generarlo o cambiar el nombre al que tengas

Antes para ejecutar nuestra aplicación ejecutábamos el comando `node index.js`, al tener `main` definido podemos solo escribir `node .` y dejar que el punto de entrada se levante del `package.json`. Ahora desde npm también podemos definir `scripts` de ejecución.

Para crear script solo hay que asignarle una nombre, seguido del comando de consola a ejecutar, dentro de la key `scripts` del `package.json`. Luego para ejecutarlo desde la terminal lanzamos

```
$ npm run nombre-de-script
```

Generalmente en el package json se agrega un script con el nombre `start` que ejecuta nuestra aplicación, por ejemplo

```
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "description": "Mi primer proyecto usando npm",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Leandro Amarillo",
  "license": "ISC"
}
```

De esta manera ejecutar `npm run start` lanza nuestra aplicación.

Si bien este es un caso bastante trivial, utilizar scripts empieza a volverse muy util para proyectos que cuenten con tests (podríamos crear un script `test` que los ejecute), cuando queremos modificar argumentos (ejemplo: variables de entorno) o si durante el desarrollo queremos utilizar alguna herramienta como [nodemon](#) (que veremos más adelante).

Agregando librerías de terceros

Como ya mencionamos, existe una gran cantidad de librerías de terceros listas para ser utilizadas en nuestro código, por ejemplo, podemos ver algunas de ellas en el

repositorio [awesome-javascript](#) que se dedica a recolectar y catalogarlas.

Para agregar una librería, desde una terminal en el directorio donde se encuentra el `package.json`, debemos ejecutar

```
$ npm i nombre-de-libreria
```

Luego de agregar una librería al proyecto el `package.json` habrá agregado una entrada dentro de la key `dependencies`.

Algunas librerías no son necesarias para ejecutar la aplicación, pero si para facilitar el desarrollo (ej.: framework de testing unitario) y para esto npm también nos deja instalarlas utilizando

```
$ npm i nombre-de-libreria -D
```

Estas dependencias también se agregan al `package.json` pero dentro de la entrada `devDependencies`

El directorio node-modules

Si nos descargamos un proyecto que utiliza npm, por ejemplo desde github, gracias a este `package.json` simplemente tenemos que ejecutar `npm i` para descargar las dependencias requeridas. Estas dependencias se guardan dentro de un directorio llamado `node-modules`, si es la primera vez que se descargan también se genera un archivo denominado `package-lock.json` el cual indica las versiones de los paquetes que se descargaron y si este está versionado permite que todos los que descarguen el proyecto utilicen las mismas versiones de las dependencias.

Importando módulos en nuestro código

NodeJS utiliza la sintaxis `require('nombre-del-modulo')` para cargar nuestras dependencias, esta función devuelve un objeto que contiene las funcionalidades que el módulo exporta es por esto que si, por ejemplo, estamos importando el módulo `fs` (file system) incluido en nodejs se suele hacer de la siguiente manera

```
const fs = require("fs");
```

La misma sintaxis aplica a las librerías de terceros, por ejemplo para utilizar la librería `dayjs` que nos facilita el manejo de fechas primero ejecutamos `npm i dayjs` para agregar y descargar la dependencia y luego para utilizarla solo hacemos

```
const dayjs = require("dayjs");
```

require (commonJS) vs import (ES6)

Por default NodeJS utiliza la sintaxis de commonJS, para utilizar la sintaxis de ES6 que utilizaremos cuando veamos

react a partir de NodeJS v12 podemos modificar el `package.json` agregando la key `"type": "module"` luego podemos reemplazar nuestros imports a la forma

```
import dayjs from 'dayjs'
```

Existen otras diferencias a la hora de generar nuestros propios módulos y detalles de implementación de cómo se cargan

que no vienen al caso en una guía introductoria. A partir de este momento se utilizará la sintaxis de ES6 para mantener la familiaridad para cuando veamos react a futuro.

4 - API usando NodeJS y Express

Preparación de proyecto

Esta sección explica cómo inicializar un proyecto de node con vscode. Algunos puntos ya los vimos anteriormente, pero se va a repetir para que sirva de punto de referencia.

Creamos una carpeta para almacenar nuestro proyecto, se recomienda utilizar `kebab-case` para este nombre.

```
$ mkdir node-express && cd node-express
```

Inicializamos un proyecto de npm con las opciones por default, usando la opción `-y`

```
$ npm init -y
```

Abrimos nuestro `package.json` que acabamos de generar y le agregamos la propiedad `"type": "module"` resultando en algo similar a

```
{
  "name": "node-express",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Recordemos que al setear el tipo de nuestro proyecto como `module` deberemos utilizar la sintaxis de **import** en lugar de **require**

Generamos el archivo `index.js` que será el punto de entrada en nuestra aplicación

```
$ touch index.js
```

Generamos una configuración para ejecutar el debugger de vscode y poder utilizar breakpoints en nuestro código desde el editor. Para esto vscode utiliza un archivo llamado `launch.json` que debe existir en una carpeta `.vscode` que por default no existen.

```
$ mkdir .vscode && cd .vscode
$ touch launch.json && cd ..
```

Desde vscode ahora abrimos el archivo `launch.json` y pegamos la siguiente config

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
```

```
// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "pwa-node",
      "request": "launch",
      "name": "Launch Node",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}/index.js"
    }
  ]
}
```

Ahora podemos agregar breakpoints en nuestro ide y lanzar el modo debug presionando **F5**.

Reiniciando nuestra aplicación cuando se modifican archivos

Cuando estemos utilizando express resultara bastante incómodo tener que finalizar nuestra app para probar los nuevos cambios. Nodemon es una herramienta que realiza este trabajo por nosotros, al detectar archivos modificados, nodemon finaliza el proceso actual y lo vuelve a lanzar. Esto nos permite iterar mucho más rápido.

Nodemon es una dependencia necesaria solo en tiempo de desarrollo, es por esto que la instalamos de la siguiente manera

```
$ npm i nodemon -D
```

Esto nos agregará una entrada al package json similar al siguiente

```
// ...
{
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
// ...
```

Con nodemon instalado podríamos ejecutar nuestra app utilizando `nodemon index.js` en lugar de `node index.js` pero una mejor solución sería modificar el archivo `launch.json` de `.vscode` para que lo haga por nosotros al lanzar el debugger. Para esto tenemos que agregar la siguiente línea dentro del array de `"configurations"`

```
{
  //...
  "runtimeExecutable":
    "${workspaceFolder}/node_modules/nodemon/bin/nodemon.js",
  //...
}
```

El archivo `launch.json` completo quedaría de la siguiente manera

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "pwa-node",
      "request": "launch",
      "name": "Launch Node",
      "runtimeExecutable":
        "${workspaceFolder}/node_modules/nodemon/bin/nodemon.js",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}/index.js"
    }
  ]
}
```

Y ahora al ejecutar nuestra aplicación con `F5` y realicemos cambios no será necesario detenerla, tan solo hay que esperar un par de segundos a que se reflejen.

Agregando Express

Express es el framework web más popular para NodeJS. Proporciona una arquitectura web flexible, minimalista y rápida de desarrollar. Además, es la base de un gran número de otros frameworks (ej.: [nest](#), [loopback](#), [sails](#), etc.).

Con él podemos construir nuestras API web, permitiendo escribir handlers para los distintos verbos http en diferentes endpoints, configurar los puertos en los que responde y agregar “middlewares” a nuestro pipeline para manejar aspectos cross como logging, autorización y autenticación de nuestras rutas.

Lo primero a realizar es agregar la dependencia de express

```
$ npm i express
```

Esto nos agregará a nuestro package json la versión más reciente de express al momento de ejecución, por ejemplo

```
//...resto de package.json...
{
  "dependencies": {
    "express": "^4.17.1"
  },
}
//...resto de package.json...
```

Desde nuestro index.js necesitamos importarlo e inicializarlo llamando a `express()`, esta llamada retorna un objeto que es el que usaremos para definir nuestras rutas y arrancar nuestra aplicación.

Veamos como ejemplo un hola mundo utilizando express

```
import express from "express";

const app = express();

// Agregamos un 'middleware' para parsear json en todas las rutas.
// Sin él no podemos acceder al body de una request.
app.use(express.json());

app.post("/saludame", (req, res) => {
```

```

    const nombre = req.body.nombre ?? "Desconocido";
    res.status(200).send(`Hola ${nombre}!`);
  });

app.get("/", (req, res) => {
  res.status(200).send("Hola Mundo!");
});

const port = 3000;
app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});

```

Luego de inicializar express lo asignamos a la variable `app` a la cual podemos asignarle nuestros distintos handlers. Inmediatamente después, y antes de definir cualquier handler, le pedimos a nuestra app que utilice el middleware de parseo de json, ya que sin él no podemos leer el objeto body. Es importante mencionar que el orden es importante, es por esto que lo declaramos primero para que se aplique en todas las rutas.

Para asignar un handler es necesario utilizar una función que tiene el nombre del verbo http a manejar (ej.: `get`, `post`, `patch`, `put`, `delete`) y recibe como primer parámetro la ruta (en nuestro caso `"/"`) y como segundo parámetro la función que manejará la request.

Opcionalmente entre la ruta y el handler también le podemos pasar un array de "middlewares" que ejecutarán para esa ruta en orden, antes del handler, pudiendo incluso finalizar la request sin dejarla continuar. Estos suelen utilizarse para manejar autenticación, autorización, logging o sanitizado de inputs.

Finalmente, para que nuestro servidor express pueda quedarse esperando peticiones, es necesario ejecutar el método `.listen(port)` que recibe de parámetro el puerto en donde escuchará peticiones y opcionalmente un callback que ejecutará cuando esté listo.

```

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});

```

Definiendo rutas

Cuando recibimos una request a nuestra api, luego de la dirección y el puerto, comienza la ruta del recurso que se está tratando de acceder. Para decirle a express quién tiene que responder a esta llamada es que tenemos que definir una ruta.

Al momento de definir una ruta para representar el string exacto (ej.: `/api/saludo`) o definir una variable que utilice parámetros de ruta (ej.: `/api/:nombre`), en este caso el valor de `:nombre` va a corresponder a cualquier string que se envíe luego de `/api/`, incluso `saludo`. Para evitar problemas tenemos que tener en cuenta lo siguiente:

El orden en que agregamos las rutas es importante, dado un request para un verbo http particular, express le entregara la respuesta al primer handler de ese verbo que matchee con la ruta.

Es por esto que con nuestro ejemplo, suponiendo que ambas rutas son para el mismo verbo, deberíamos definir primero `/api/saludo` y luego `/api/:nombre` para evitar que esta última maneje todos los request.

Definiendo funciones handlers

Un handler de express es una función con la siguiente estructura

```
const handler = (req, res) => {  
  // Mi lógica  
  // ...  
}
```

Por convención estos dos parámetros se llaman req y res, pero no hay nada que imponga que efectivamente se llamen así. Lo que sí es importante es no confundirlos de lugar.

Leyendo datos de la request con req

`req` es el objeto que utiliza nuestro servidor express cuando recibe datos de la llamada del cliente. Estos pueden llegar de tres maneras `req.params`, `req.body` y `req.query`.

El objeto `req.params` captura los datos basado en los parámetros de la ruta, si por ejemplo definimos una ruta como `/api/:nombre` y llega una request cuyo path es `/api/ricardo` podemos recuperar ese valor de la siguiente manera

```
req.params.nombre // Retorna: ricardo
```

El objeto `req.query` captura los datos basado en querystrings de la request, si por ejemplo definimos una ruta como `/api/saludar` y llega una request cuyo path es `/api/saludar?nombre=ricardo&apellido=sanchez` podemos recuperar estos valores de la siguiente manera

```
req.params.nombre // Retorna: ricardo  
req.params.apellido // Retorna: sanchez
```

El objeto `req.body` captura los datos basado en el body de la request, si por ejemplo definimos una ruta como `/api/saludar` y llega una request que contiene un body de la siguiente forma

```
{  
  "nombre": "ricardo",  
  "apellido": "sanchez"  
}
```

Podemos recuperar estos valores de la siguiente manera

```
req.body.nombre // Retorna: ricardo  
req.body.apellido // Retorna: sanchez
```

[req](#) contiene bastante más información sobre la request actual, por ejemplo utilizando `req.header(...)` podemos consultar el valor de algún header particular, conveniente para cuando utilizamos autorización por jwt, podemos recuperar el token simplemente llamando a `req.header('Authorization')`.

Devolviendo respuestas con res

`res` es el objeto que prepara la respuesta http que nuestra api en express enviará al cliente luego de recibir una request.

Podemos utilizar este objeto para setear los headers a retornar, por ejemplo:

```
res.set('Content-Type', 'application/json; charset=utf-8');
```

Además, podemos definir el [código de estado de la respuesta](#) que en caso de no hacerlo, por default, será `200`. Ejemplo:

```
res.status(500)
```

Finalmente podemos indicarle a express que queremos retornar mediante el método `send(...)` indicándole como argumento el valor de la respuesta, por ejemplo

```
// ... código de mi handler

if (error) {
  res.status(500).send({ mensaje: "Algo salió mal :c" });
} else {
  res.send({ nombre: "ricardo", apellido: "sanchez" });
}
```

Sin importar el resultado de la request hay algo que tenemos que tener en cuenta al escribir nuestros handlers y es lo siguiente:

Siempre tenemos que devolver una respuesta, aunque sea de error, si no el cliente quedará esperando hasta finalizar la llamada por timeout en el mejor de los casos.

Definiendo middlewares

Los middlewares son bastante similares a los handles, pero agregan un parámetro extra convencionalmente llamado `next`.

Son funciones de la siguiente forma

```
const middleware = (req, res, next) => {
  // Mi lógica
  // ...
}
```

Las funciones de middleware pueden realizar las siguientes tareas:

- Ejecutar cualquier código.
- Realizar cambios en la solicitud y los objetos de respuesta.
- Finalizar el ciclo de solicitud/respuestas.
- Invocar la siguiente función de middleware en la pila.

Ejecutar una llamada a `next()` continúa la ejecución del siguiente middleware, o si ya no quedan middlewares, el handler.

Un middleware sencillo puede ser uno que loguee en consola las llamadas que recibe, por ejemplo

```
const logger = (req, res, next) => {
  console.log('Request Path:', req.path);
  console.log('Request Type:', req.method);
  console.log('Request Time:', Date.now());
  console.log('-----');
  next();
}
```

Uno un poco más complejo podría bloquear llamadas no autenticadas, por ejemplo

```
const auth = (req, res, next) => {
  const authorization = req.header('Authorization');
  if (!!authorization && isValidToken(authorization)) {
    next();
  } else {
    res.status(401).send({ mensaje: "No se encuentra logueado o su sesión expiro, vuelva a intentarlo." });
  }
}
```

Una vez que tenemos definido nuestro middleware es necesario decidir **en qué momento de la request debe ejecutarse**.

Por ejemplo, anteriormente definimos:

```
app.use(express.json());
```

Es mediante el método `.use(...)` es que agregamos el middleware de `express.json()` para todas las request. Pero solo afecta a todas las request porque antes de él no hay ningún handler definido, todos

están definidos luego de agregar

este middleware. Si hubiéramos definido un handler para `POST` antes del middleware no hubiéramos podido decodificar el

body de la request y al tratar de obtener `req.body` obtendríamos siempre `undefined`

Además, como ya comentamos, es posible agregar middlewares al nivel de handler, veamos este ejemplo

```
import express from "express";

const app = express();

app.use(express.json());

const logger = (req, res, next) => {
  console.log('Request Path:', req.path);
  console.log('Request Type:', req.method);
  console.log('Request Time:', Date.now());
  console.log('-----');
  next();
}

const auth = (req, res, next) => {
  const authorization = req.header('Authorization');
  if (!!authorization && isValidToken(authorization)) {
    next();
  } else {
    res.status(401).send({ mensaje: "No se encuentra logueado o su sesión expiro, vuelva a intentarlo." });
  }
}

// Agregamos ambos middlewares al handler
app.post("/saludame", [logger, auth], (req, res) => {
  const nombre = req.body.nombre ?? "Desconocido";
  res.status(200).send(`Hola ${nombre}!`);
});

// No agregamos ningún middleware
app.get("/saludame", (req, res) => {
  res.status(200).send(`Hola!`);
});
```

```
const port = 3000;
app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

En este caso un `post` a `/saludame` ejecutará primero el middleware de parseo de json, luego el de logger, luego el de auth y finalmente, si la request llego con un token válido en los headers, el handler.

Pero para las request `get` a `/saludame` solo estamos pasando por el middleware de jsons, el cual no debería hacer nada, ya que los get normalmente no tienen `body` y llegamos al handler.