# INTRODUCCION

Contenido

[VARIABLES 6](#_Toc178771938)

[STRINGS 10](#_Toc178771939)

[NUMEROS 17](#_Toc178771940)

[OPERADORES 23](#_Toc178771941)

[BOOLEAN 27](#_Toc178771942)

[OBJETOS 30](#_Toc178771943)

[ARRAYS 39](#_Toc178771944)

[FUNCIONES 49](#_Toc178771945)

[ESTRUCTURAS 61](#_Toc178771946)

[ITERADORES 68](#_Toc178771947)

[ARRAY METHODS 76](#_Toc178771948)

[DOM 85](#_Toc178771949)

[EVENTOS 98](#_Toc178771950)

[LOCAL STORAGE 109](#_Toc178771951)

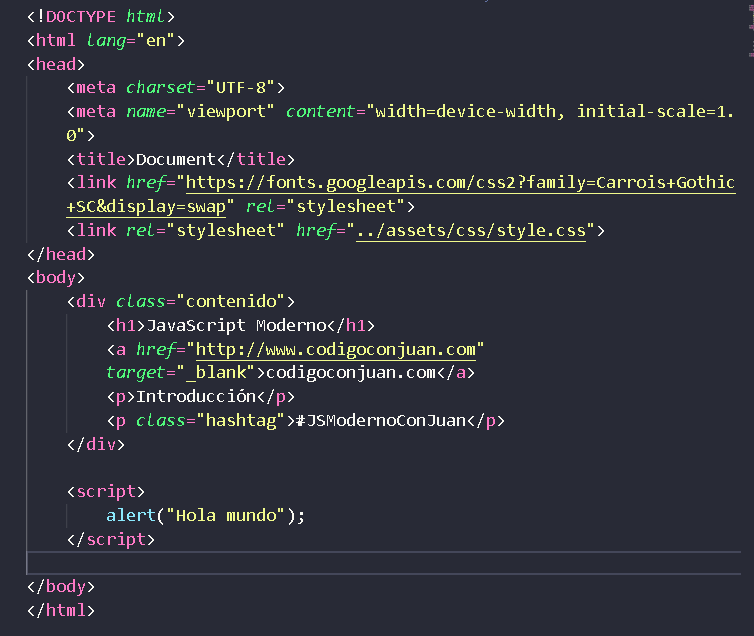
[Fechas 113](#_Toc178771952)

[Prototypes 116](#_Toc178771953)

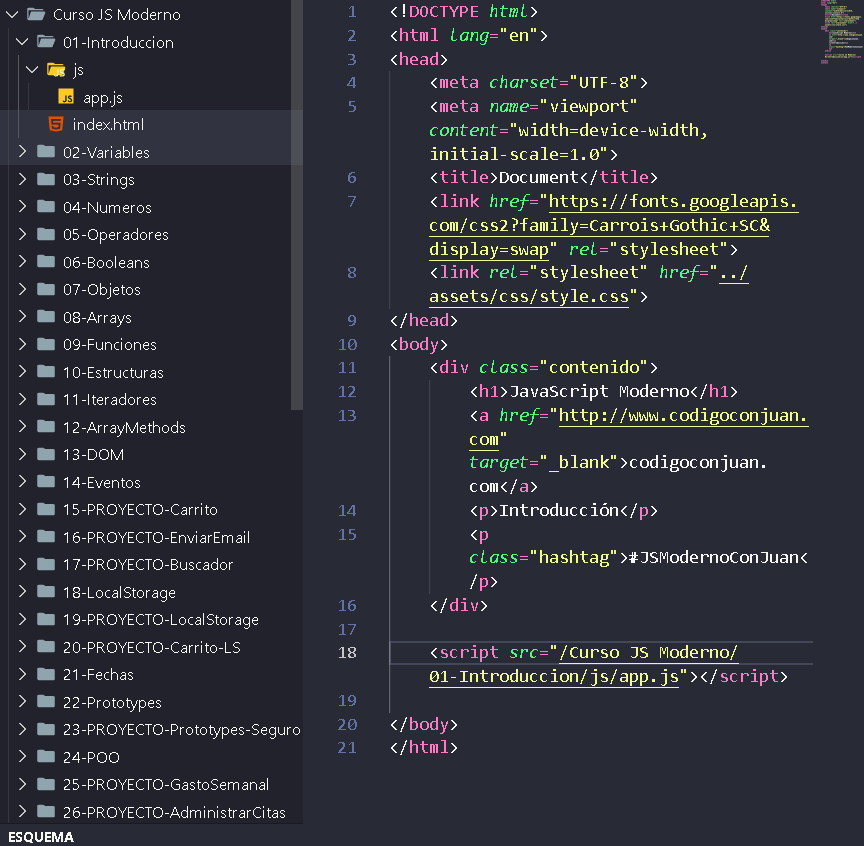
[POO 121](#_Toc178771954)

El codigo de javascript puede estar en cualquier lugar, puede estar en el Head, en el Body o después del Body.

Usualmente se coloca antes del cierre del Body.

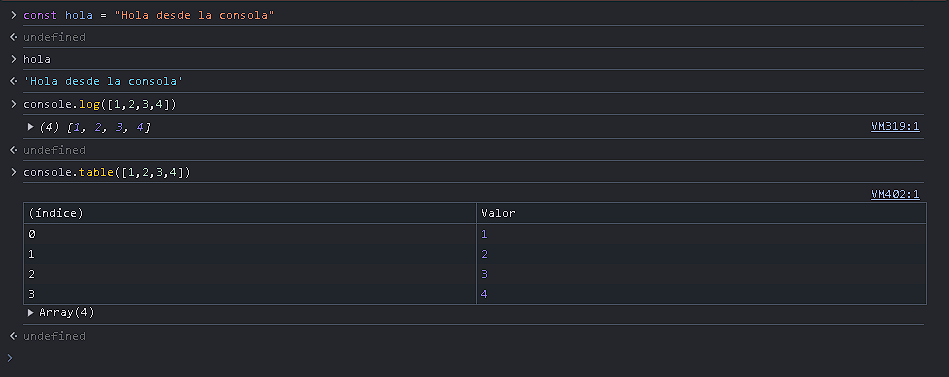


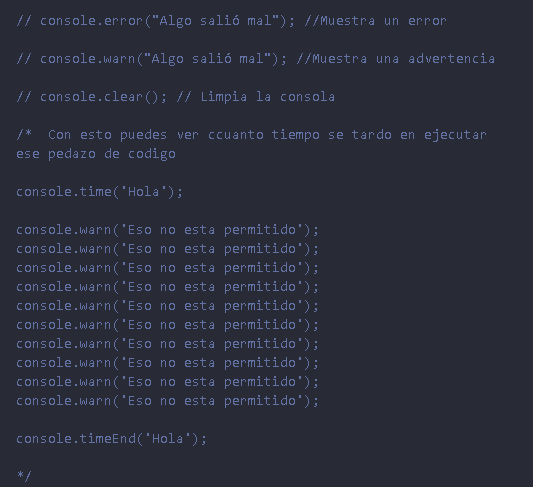
Se recomienda crear una carpeta y crear un archivo .js para allí meter el código que se desee compilar.

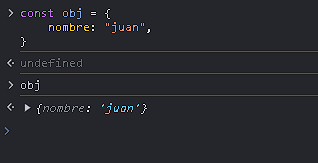


Diferentes cosas que puedes hacer desde la consola.

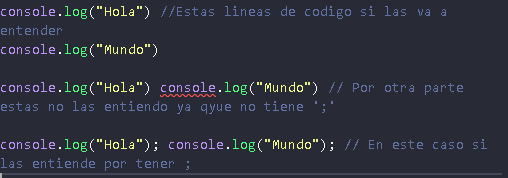
* Puedes crear variables y mandarlas a llamar
* Puedes crear arreglos e imprimirlos
* Puedes crear una tabla con arreglos e imprimirla (útil cuando tienes multiples registros)



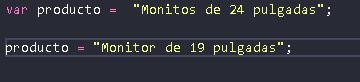
También puedes mostrar varios mensajes desde la consola, estos son unos de los cuales puedes mostrar:  


También puedes crear objetos y mostrarlos:  


# VARIABLES

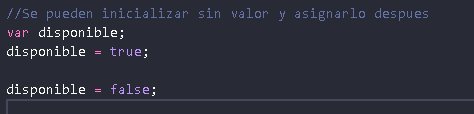
En Js, el ‘:’ no es obligatorio, mas sin embargo es necesario cuando se necesita dejar en claro cuando es que finaliza una línea de código:  


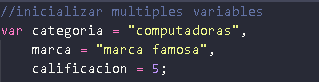
Las variables en js se les puede re-asignar un valor:



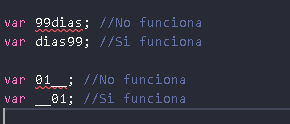
Javascript no es un lenguaje tipado, es un lenguaje de tipo dinámico, no necesitas agregar un int, boolean, etc.

Las variables se pueden inicializar sin valor y después se les puede asignar:

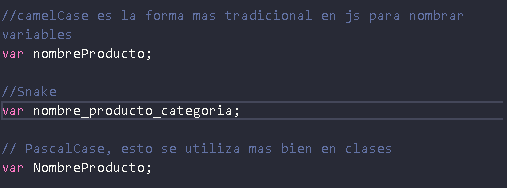


Puedes inicializar multiples variables:  


Una variable puede contener letras, guines bajos o numeros, pero NO pueden iniciar con un numero:



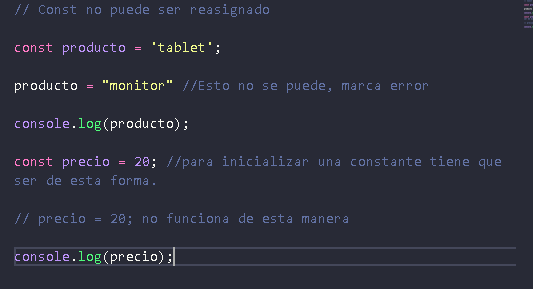
Formas de escribir una variable:



Para la variable let aplica casi lo mismo:  

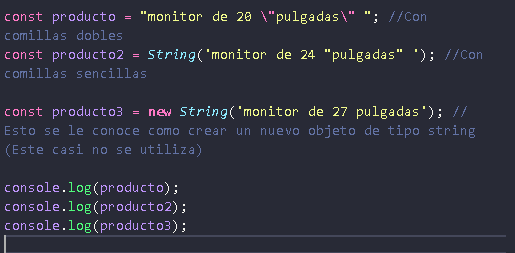

La diferencia entre estas 2 es que es lo que se conoce como el scoop de las variables.

La diferencia entre const y las otras dos, es que const no se puede reasignar y que tampoco se puede inicializar después, tienes que inicializarla ahí mismo cuando la creas.

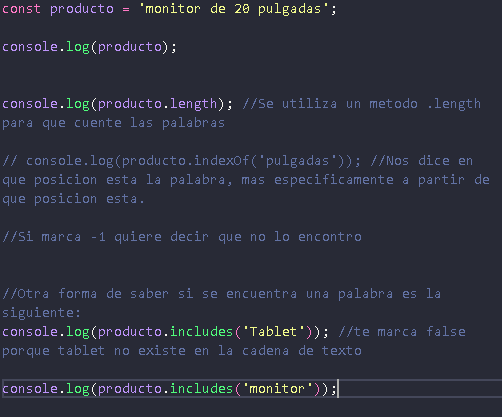


# STRINGS

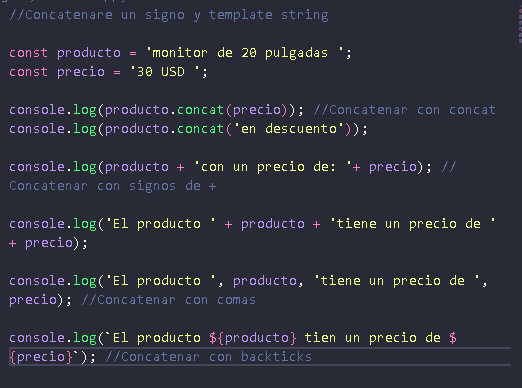
Los string o cadenas de texto, representan un texto, que se puede asignar a una varible, cosas como el nombre de un cliente, el nombre de un producto, la categoria, todo eso es un texto y se representa por medio de un string.



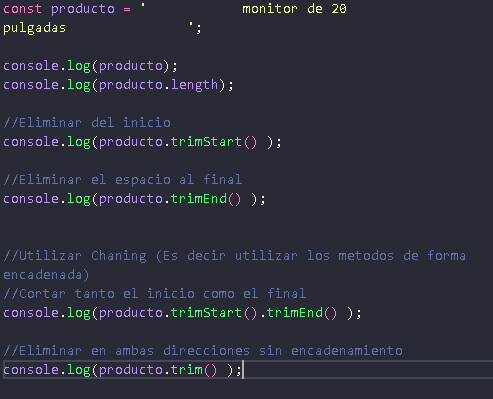
A esto se le conoce como metodos de los strings, son propiedades que tienen los strings.



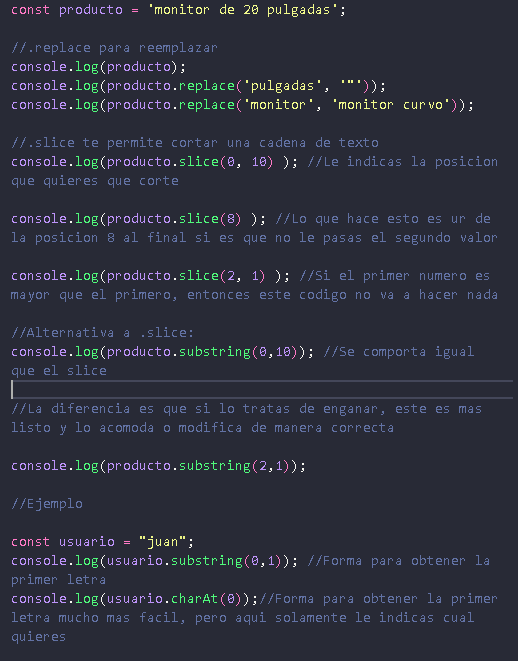
Hay diferentes formas de concatenar una cadena de texto, en especial se utilizan 4, en donde la más recomendada es la última:



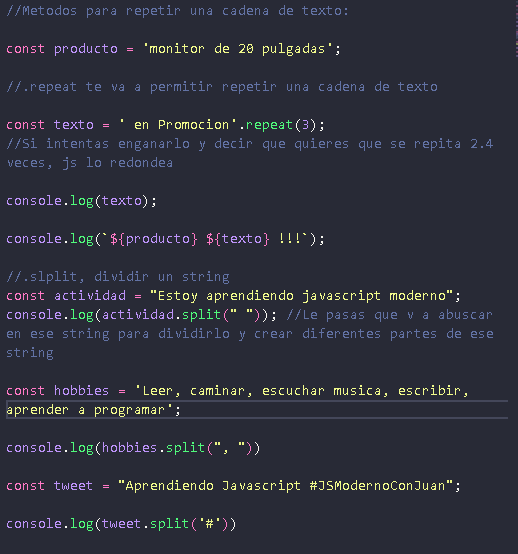
Métodos para eliminar el espacio en blanco al inicio y al final de una cadena de texto



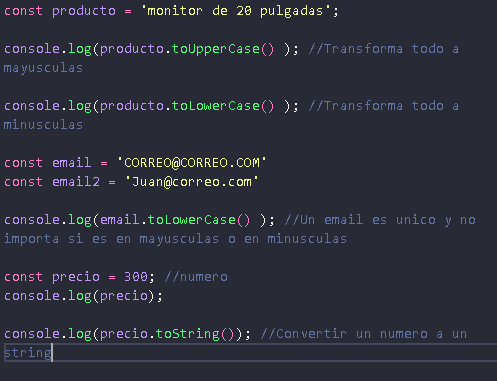
Método para reemplazar el texto de una cadena de texto



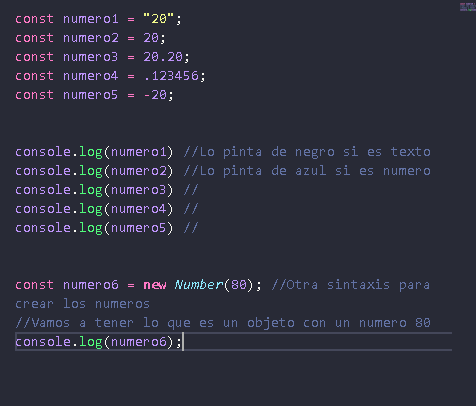
Métodos para repetir y dividir una cadena de texto:



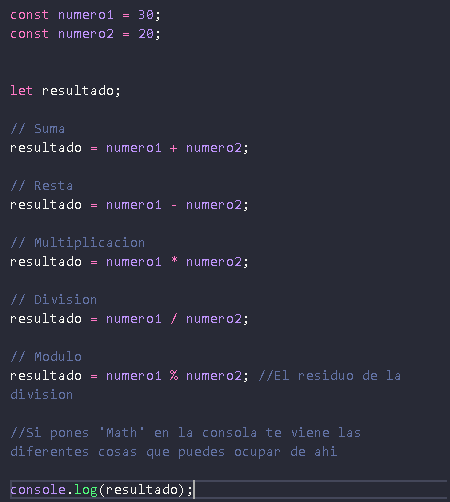
Metodos string para convertir a mayusculas o minúsculas y convertir un numero a string:



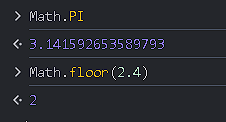
# NUMEROS

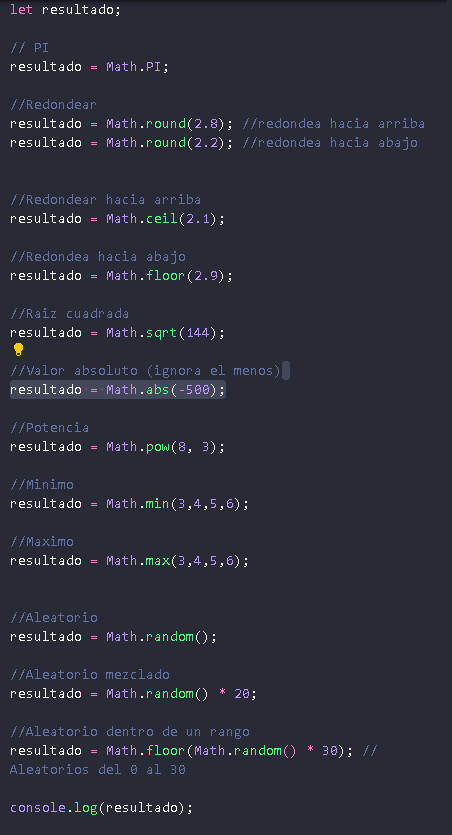
Los números pueden ser positivos, pueden ser negativos, etc, pero siempre se crean de la misma forma en js. Lo único que NO lo haría un numero seria que esten entre comillas.

Operaciones básicas en js, como suma, resta, multiplicación, división y modulo.

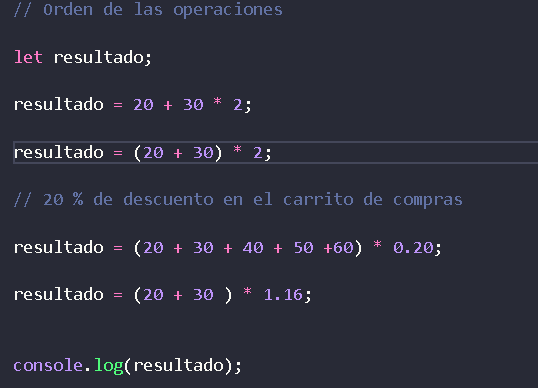


Operaciones con Math:

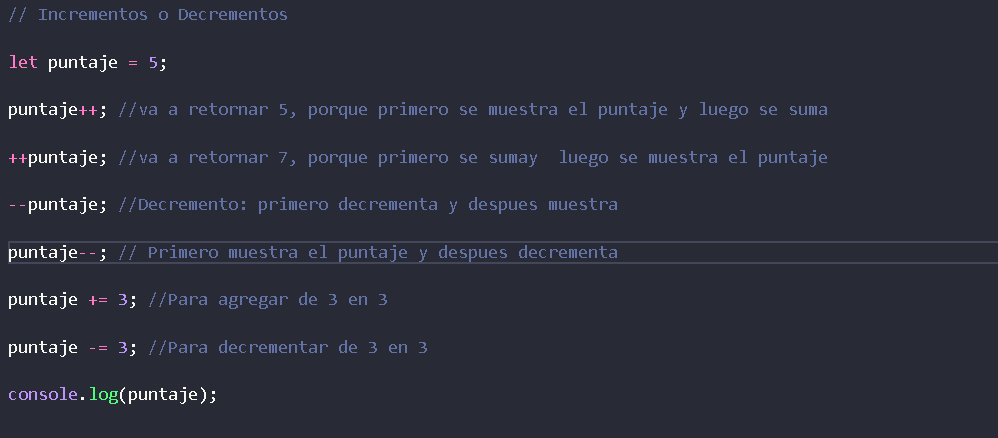




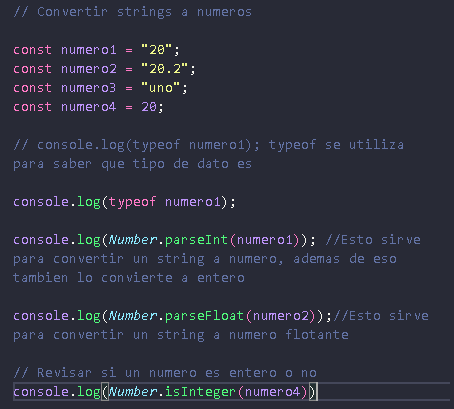
Orden de las operaciones:



Incrementos o Decrementos:

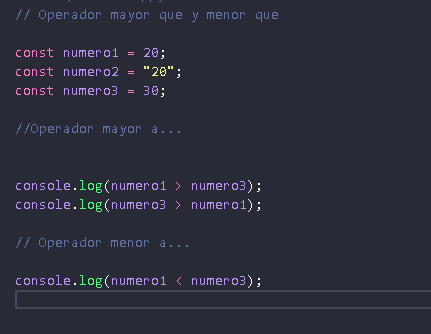


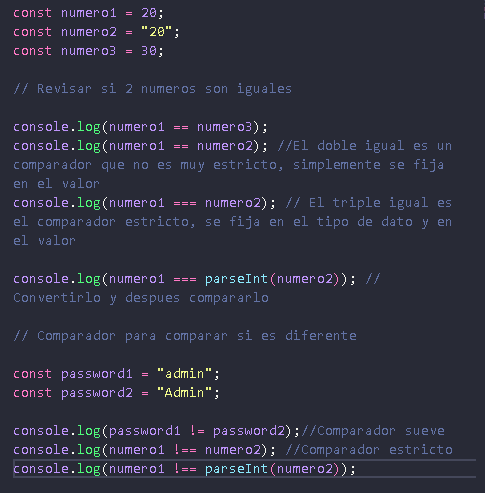
Convertir strings a números:



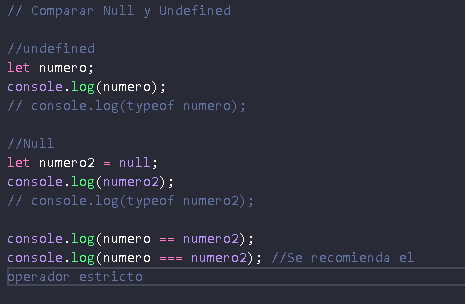
# OPERADORES

Operador mayor que y menor que:



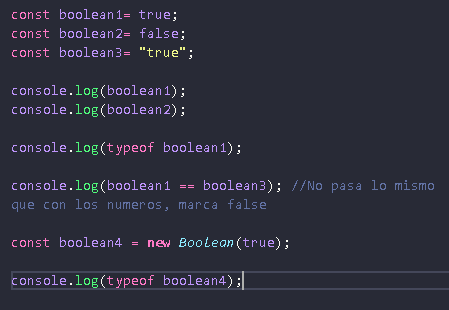
Comprar valores:  


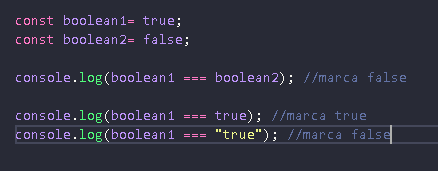
Comparar Null y Undefined

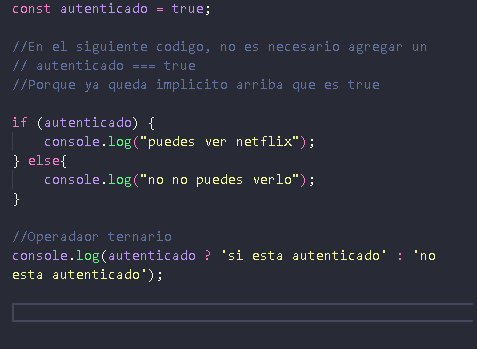


# BOOLEAN

Crear y comparar boolean

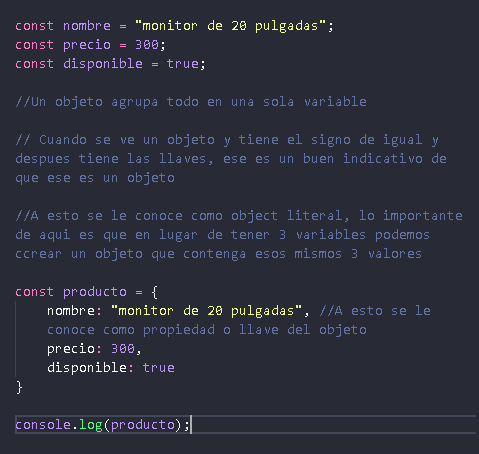




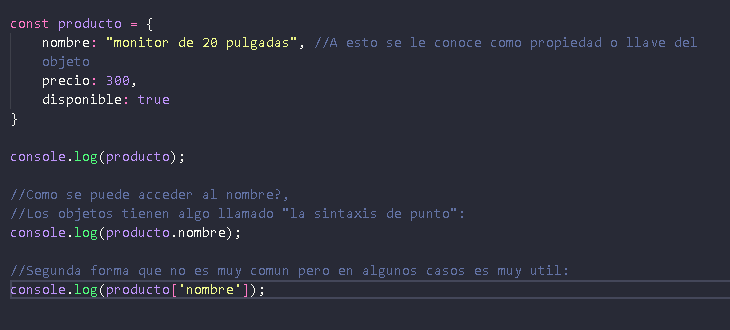


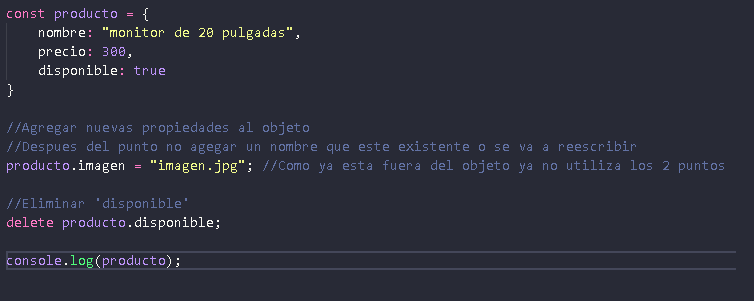
# OBJETOS

Crear objetos en JS

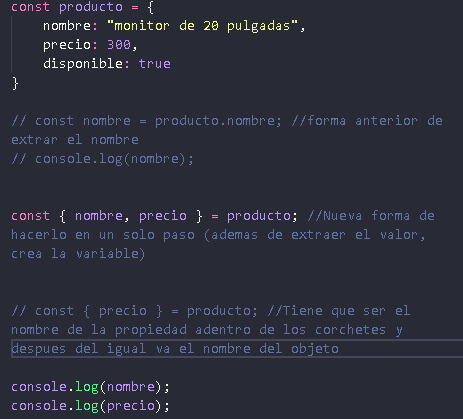


Como acceder a los valores de un objeto

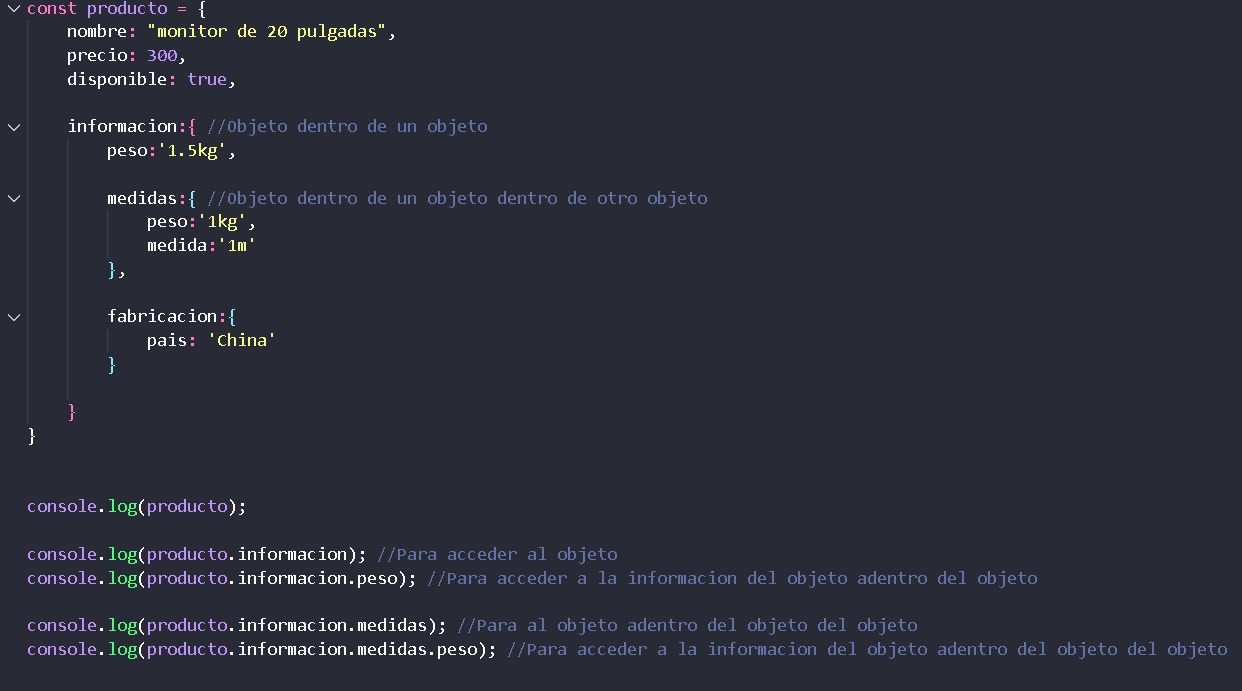


Agregar o Eliminar propiedades de un objeto

Desestructuración de objetos



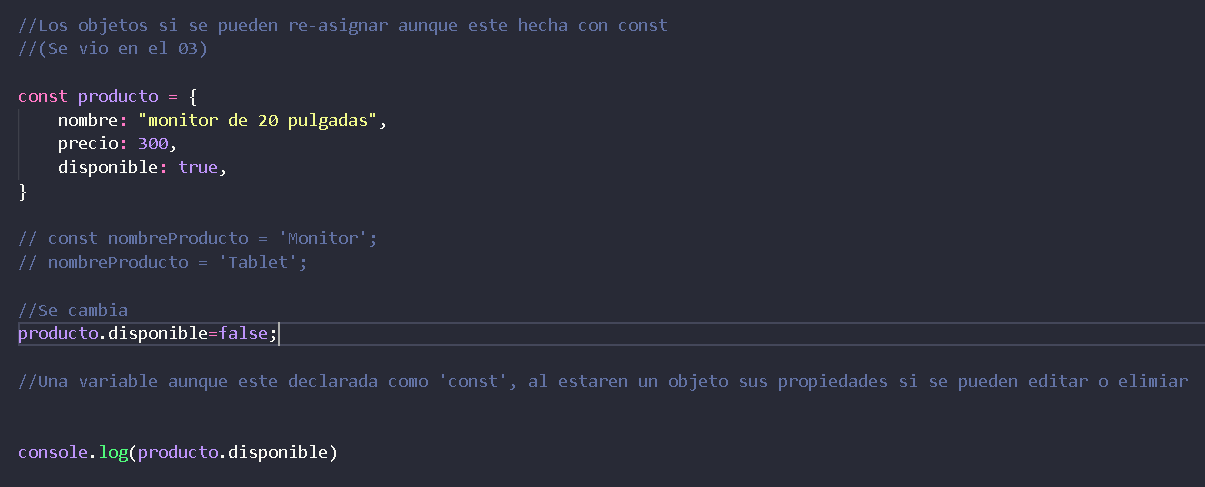
Objetos dentro de un objeto



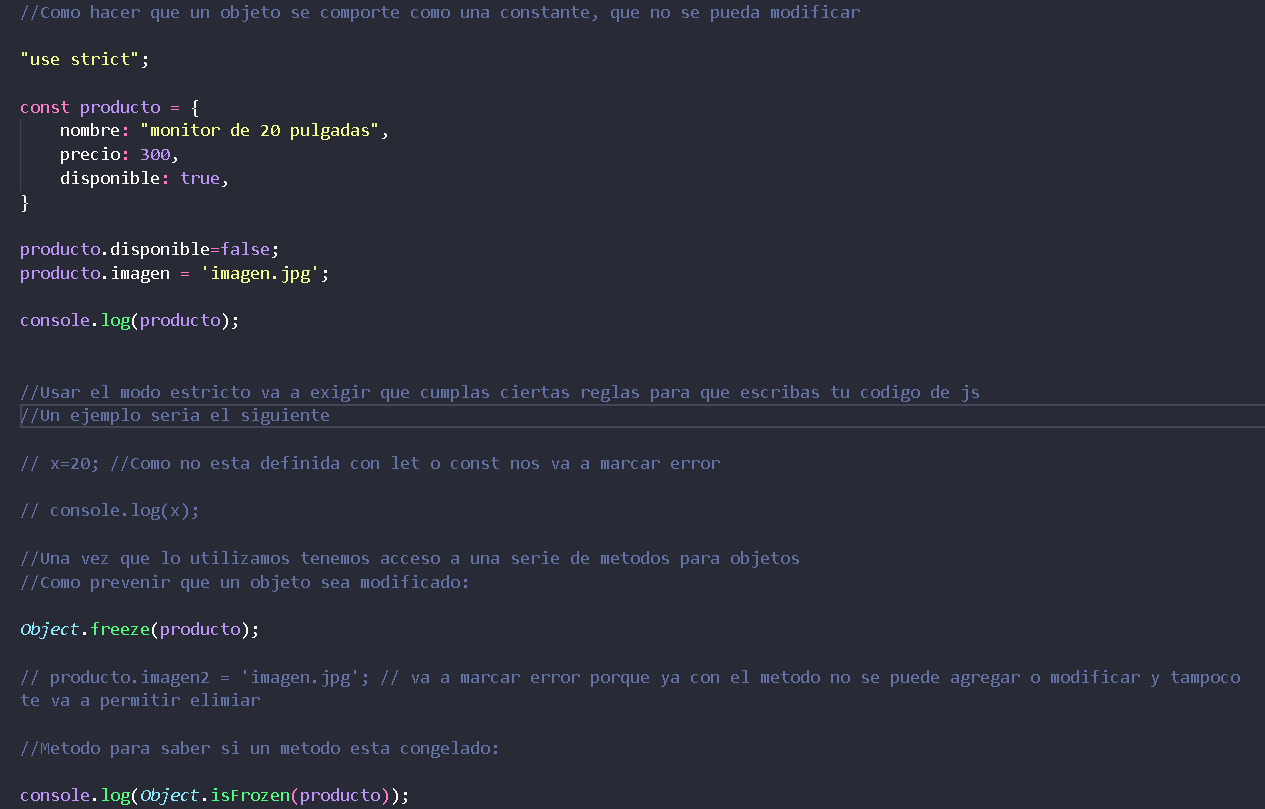
Destructuring de Objetos Anidados



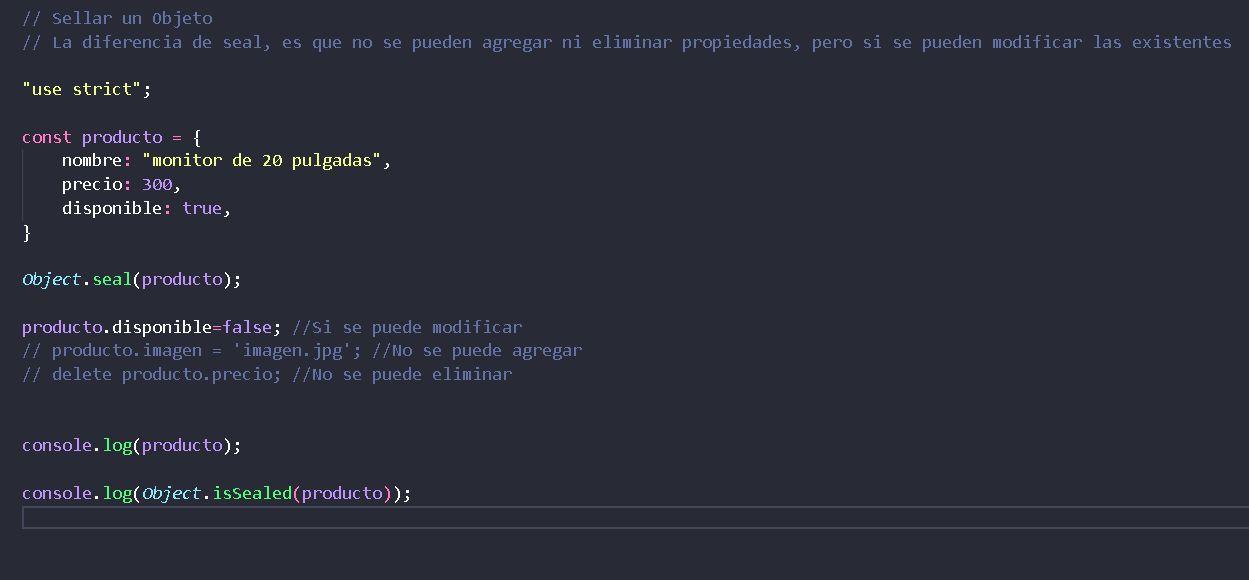
El Problema con los objetos



Congelar un Objeto para no poderlo modificar



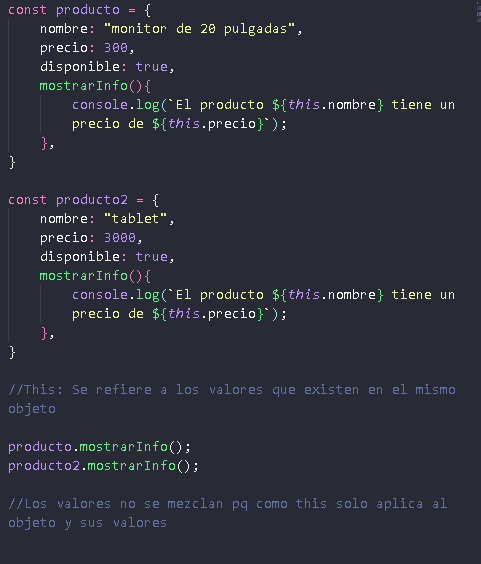
Sellar un Objeto



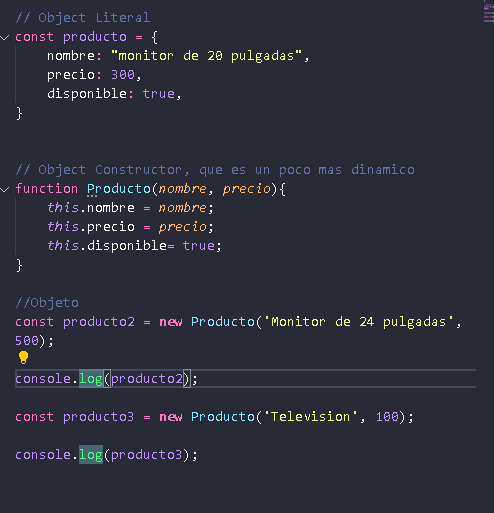
Unir 2 objetos



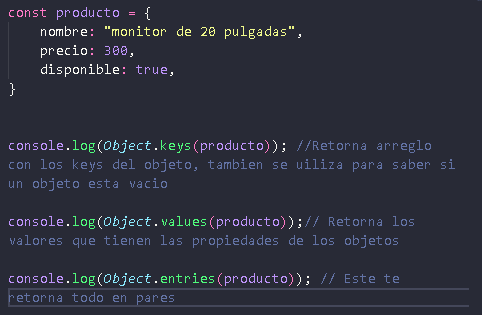
Funciones en objetos y acceder a sus valores



El Object Constructor

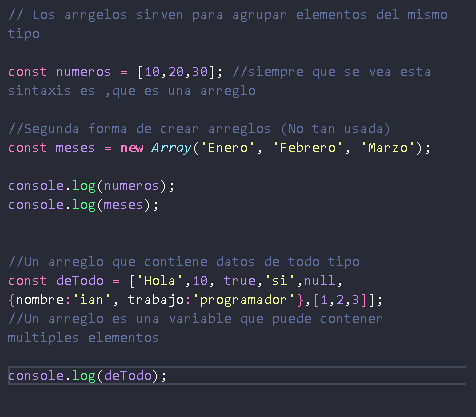


Object .keys, .values y .entries

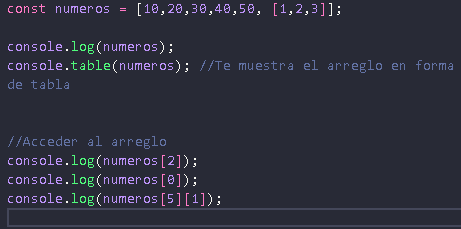


# ARRAYS

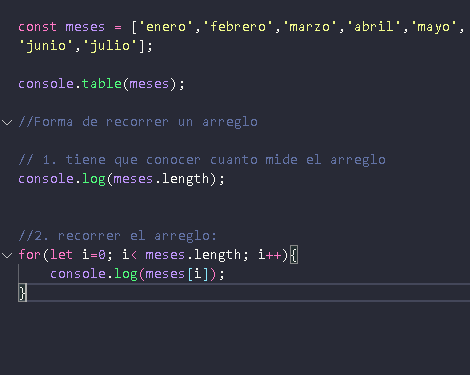
Crear Arrays en JavaScript



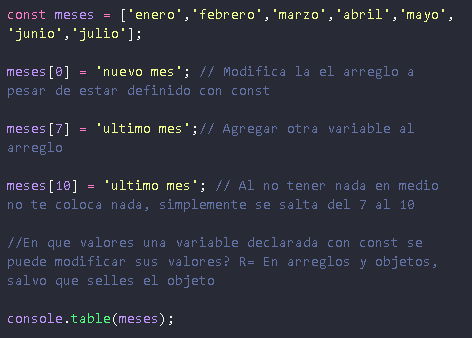
Acceder a los valores de un Array



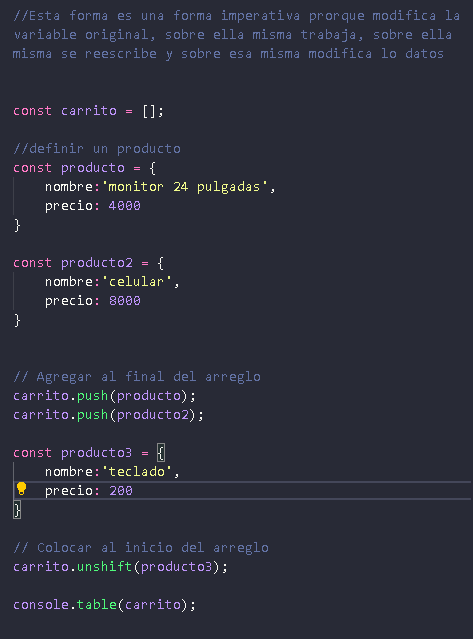
Recorrer un Array



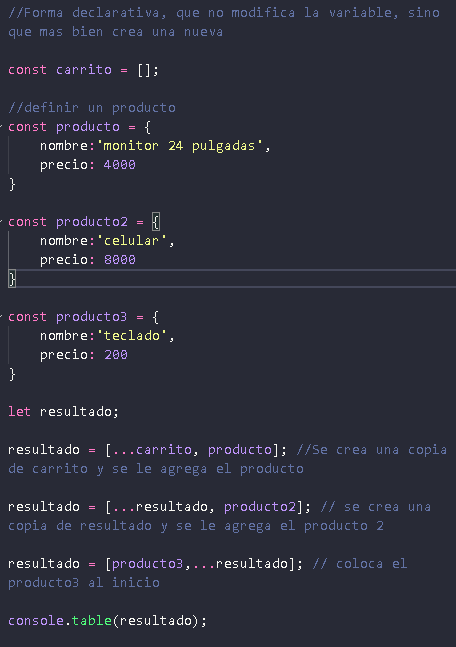
Agregar nuevos valores a un array



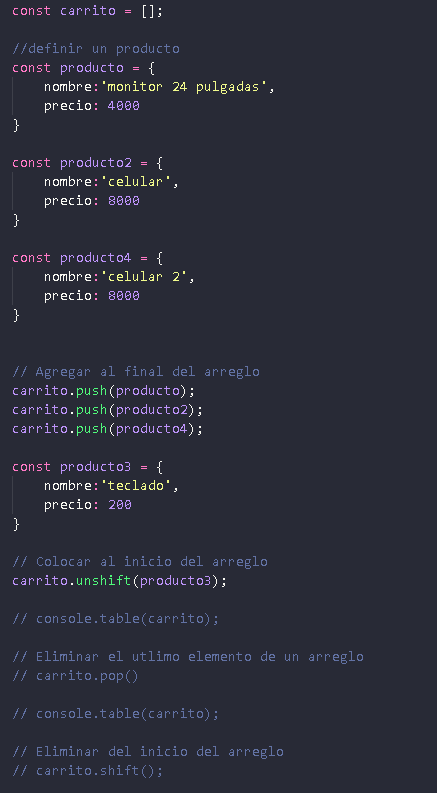
Añadir nuevos elementos al fin o Inicio de un array

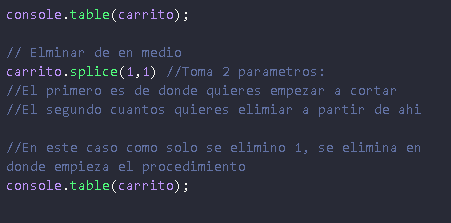


Crear un nuevo arreglo con el spread operator

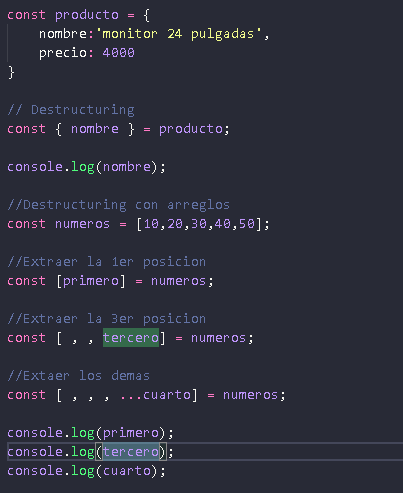


Eliminar elementos con Splice

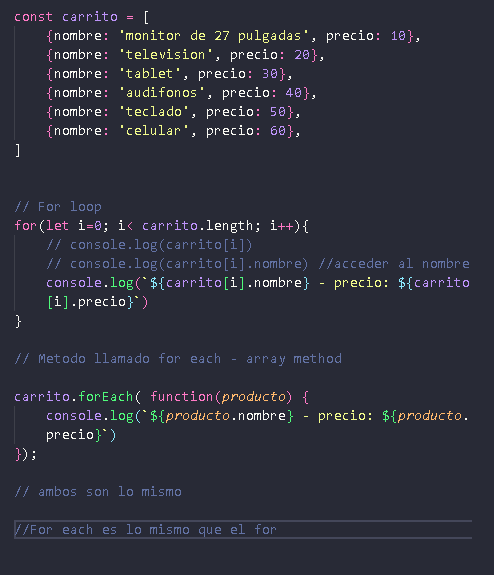




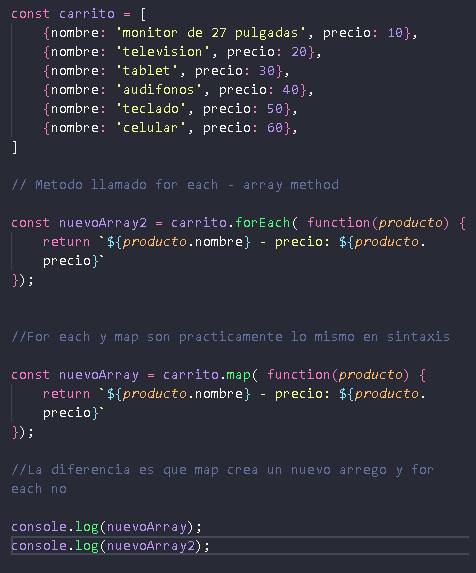
Destructuring de Arrays



forEach para iterar un array

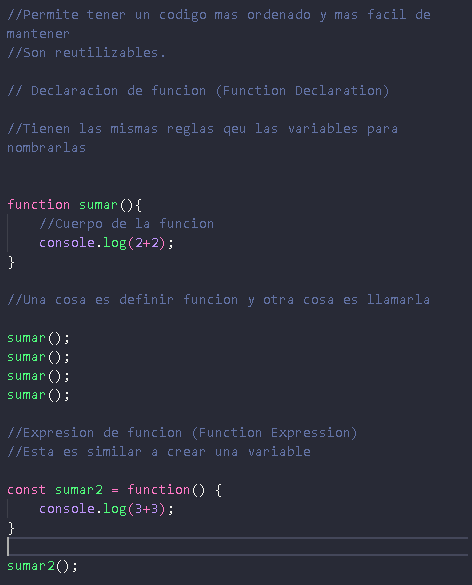


.map para iterar un array, y sus diferencias con forEach

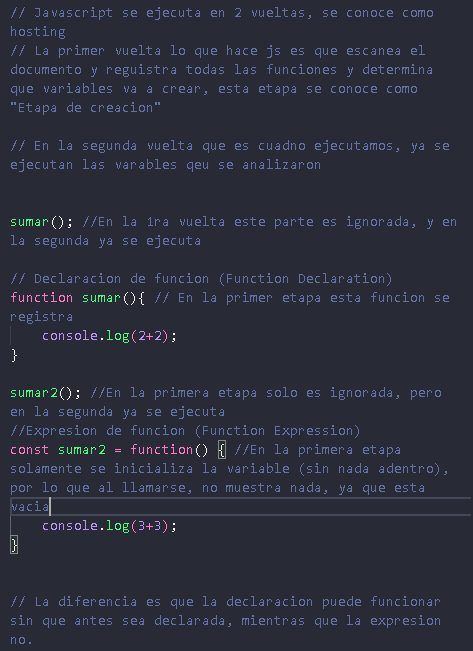


# FUNCIONES

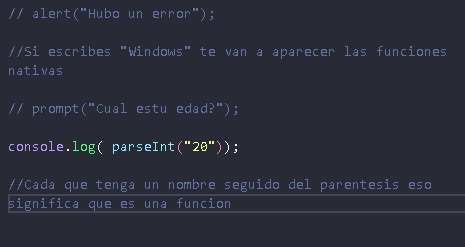
Crear Funciones en JS - Function expression y Function Declaration



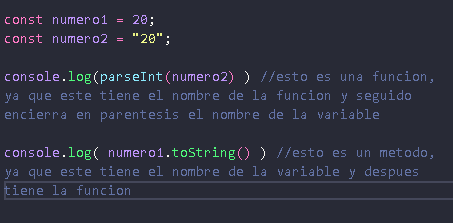
La Diferencia entre Function expression y Declaration



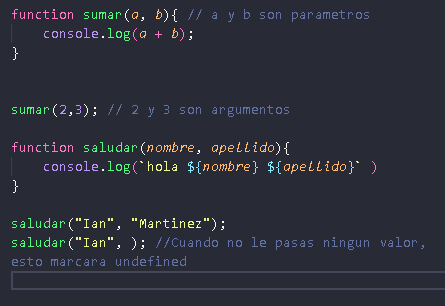
Algunas funciones nativas en JavaScript



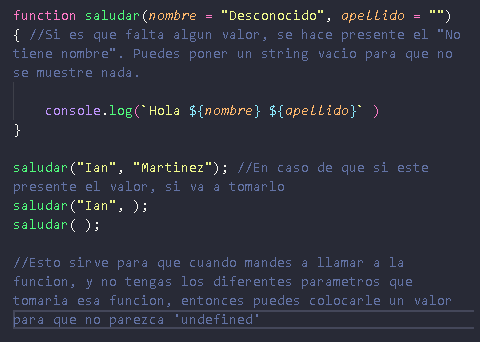
La Diferencia entre Función y Método



Parametros y Argumentos en Funciones



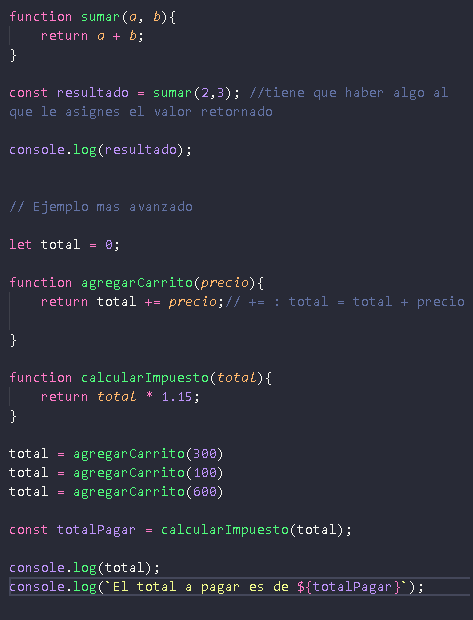
Parametros por default



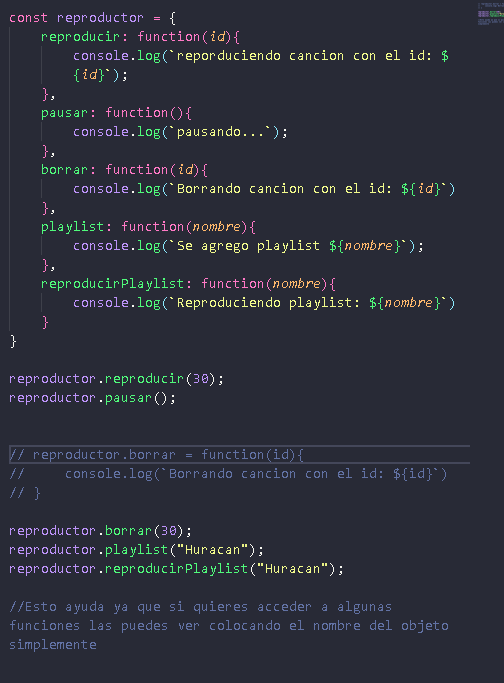
Como se Comunican las funciones



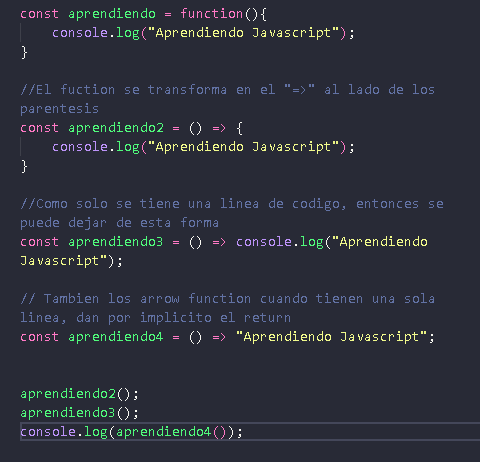
Ejemplo con múltiples funciones que se pasan valores



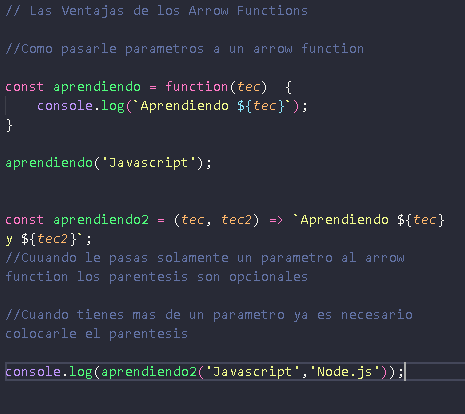
Añadir Funciones en un Objeto



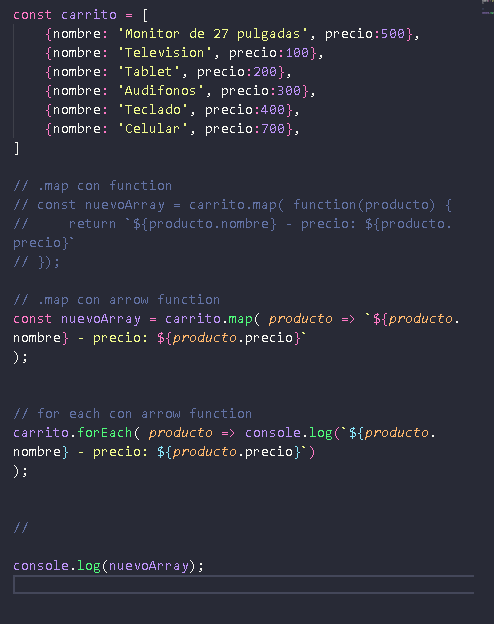
Arrow Functions



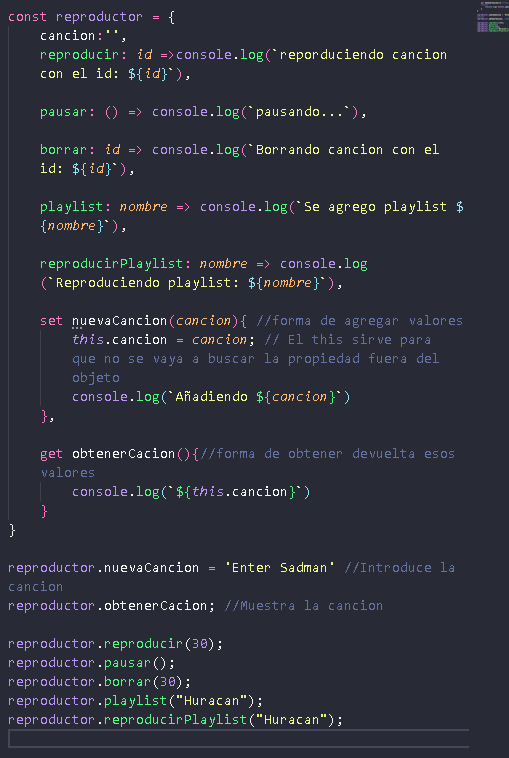
Las Ventajas de los Arrow Functions



Arrow Functions en un forEach y un map

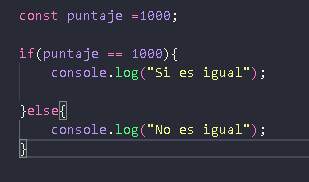


Arrow Functions en el Reproductor de Música

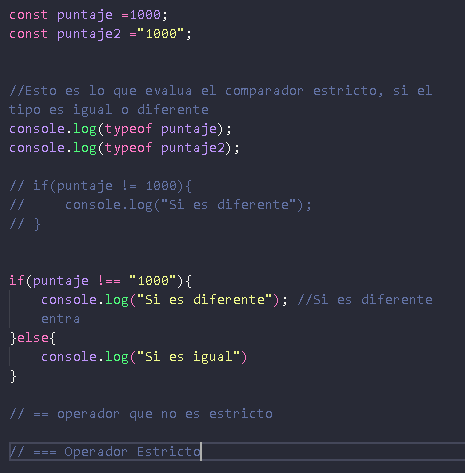


# ESTRUCTURAS

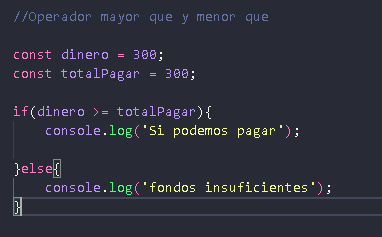
Creando un If



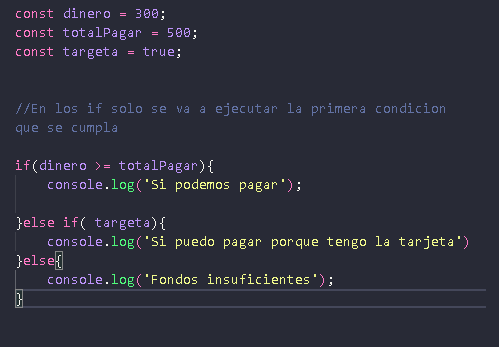
Comparador Estricto



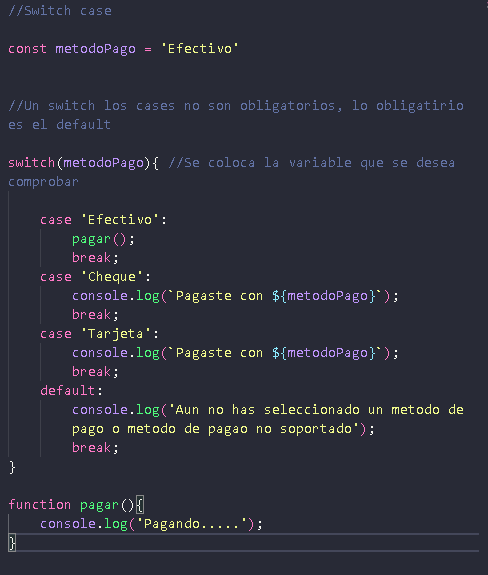
Comparando Mayor o Menor que



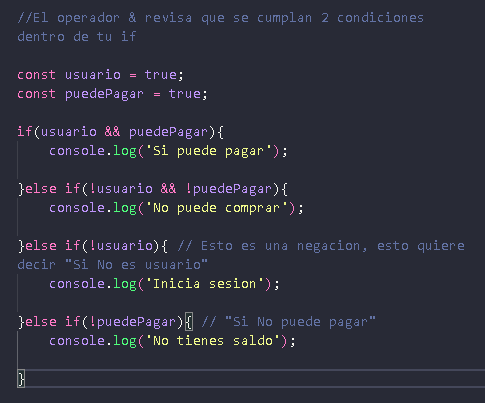
Mayor o Igual y else if



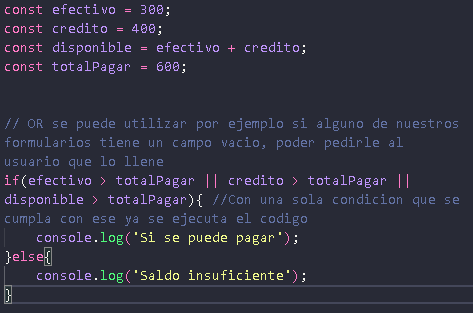
Switch para evaluar múltiples Condiciones



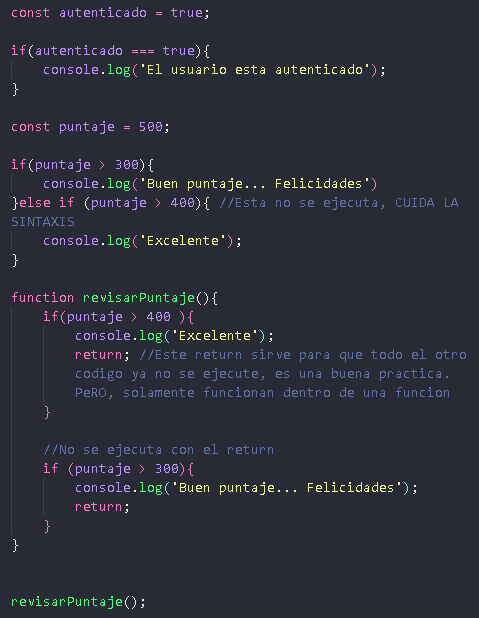
El Operador && para revisar que se cumplan 2 o más condiciones



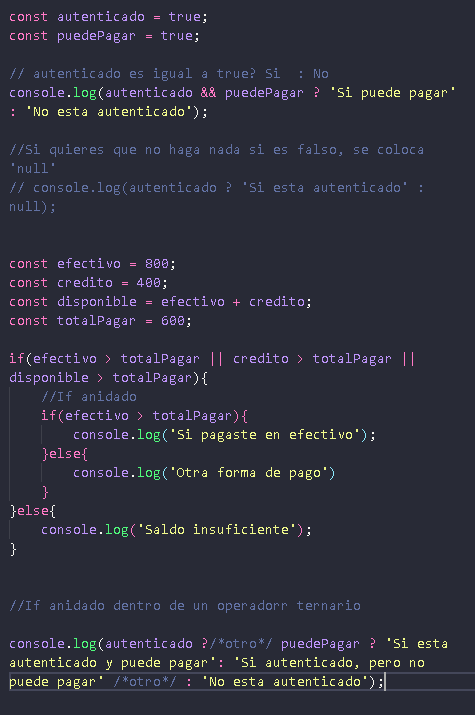
El Operador OR para que se cumpla al menos una condición



Detener la ejecución de un if con una función

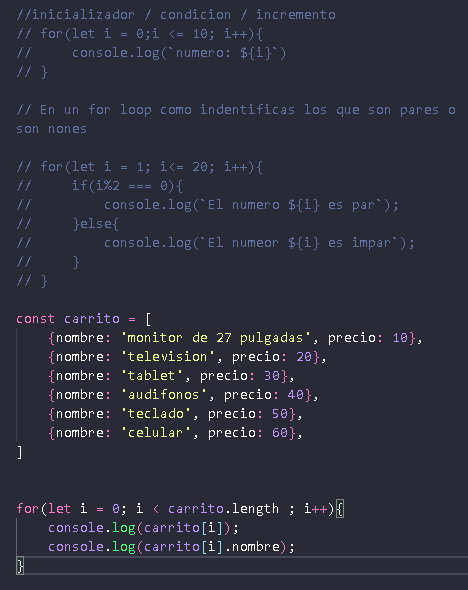


El Operador Ternario

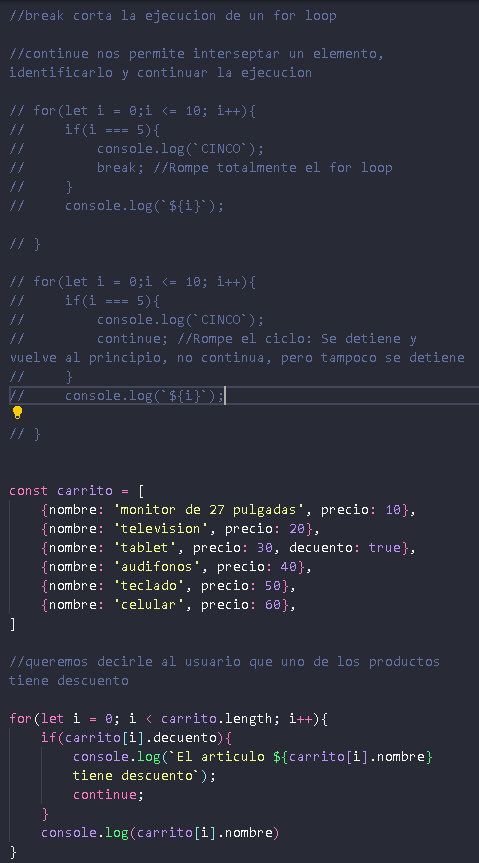


# ITERADORES

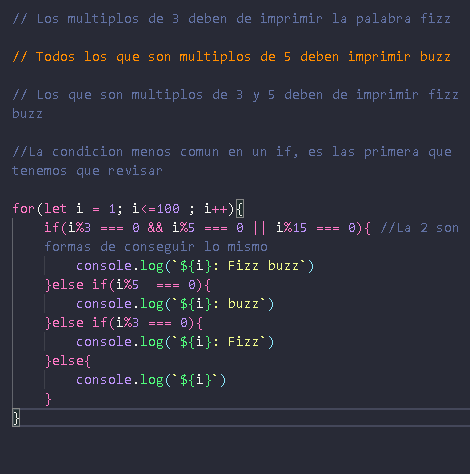
For Loop



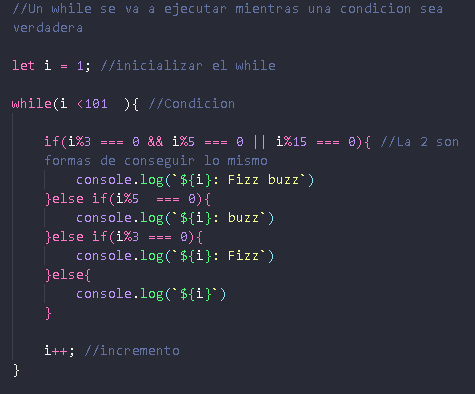
break y continue; en un for loop



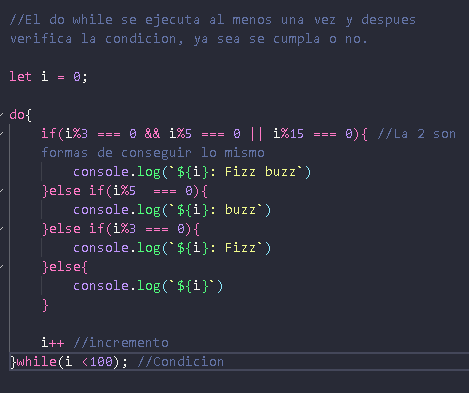
El Ejercicio Fizz Buzz



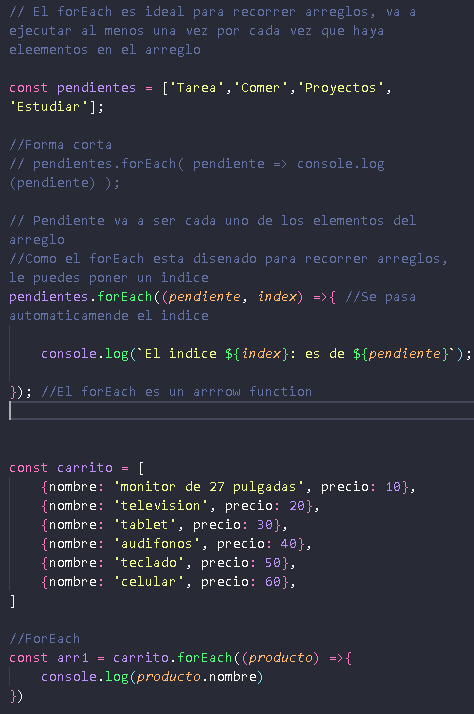
While Loop

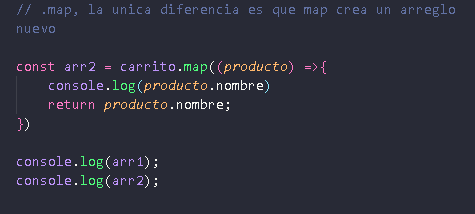


Do While Loop

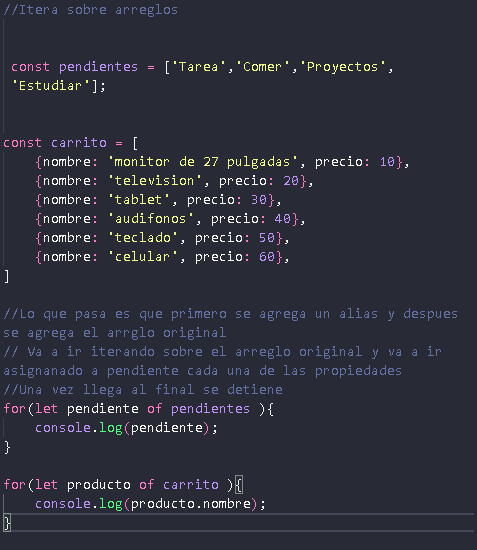


.forEach y .map

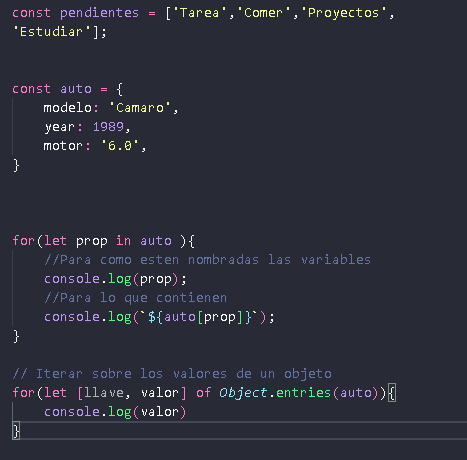




for ....of

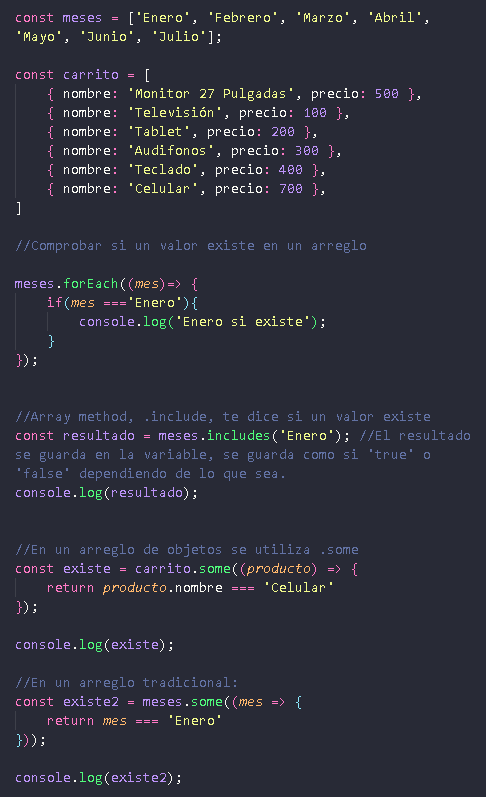


For in itera sobre objetos

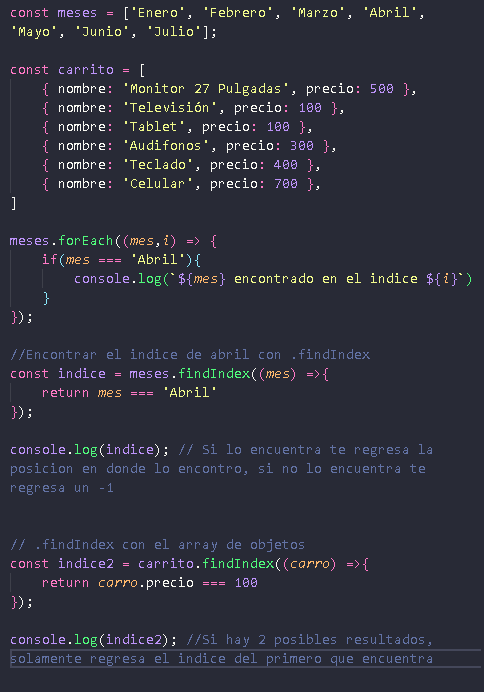


# ARRAY METHODS

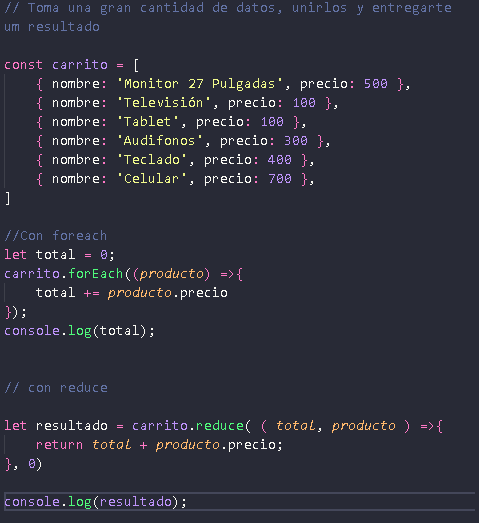
Array method .some



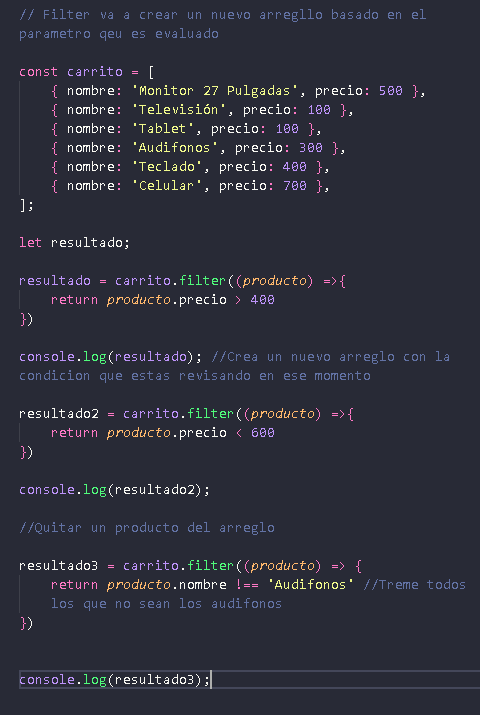
.findIndex para encontrar la posición en un array



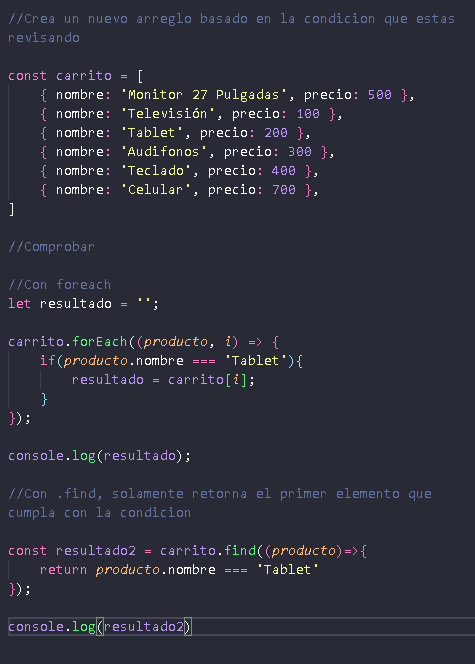
.reduce



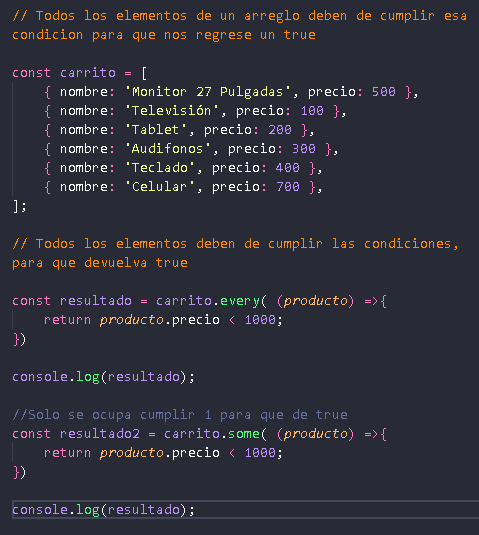
.filter



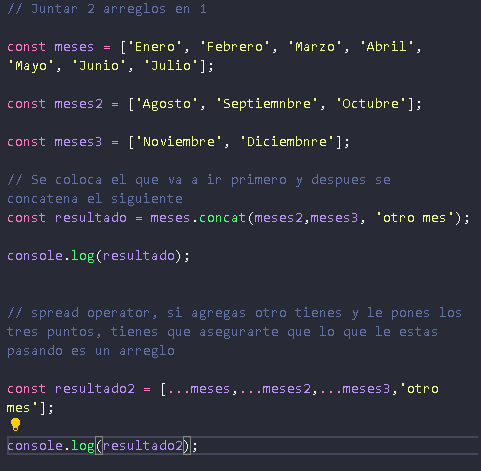
.find



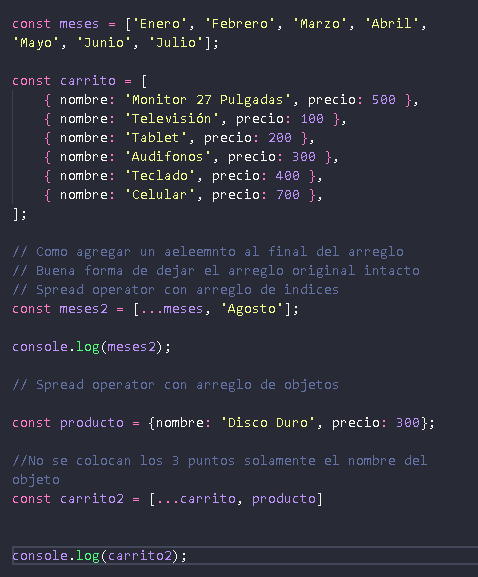
.every



.concat

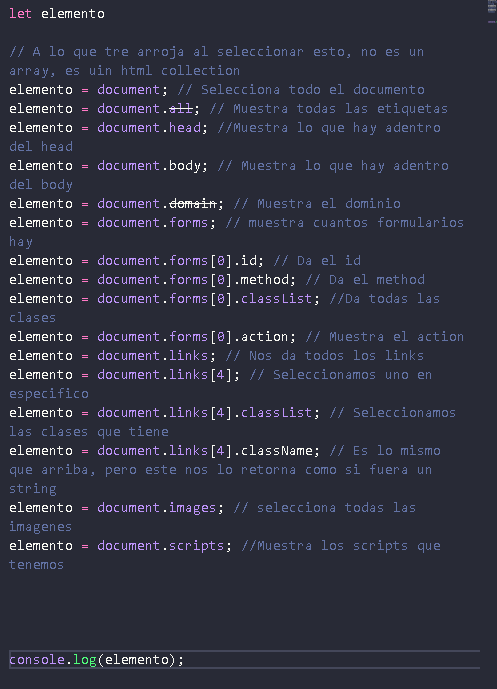


Spread Operator

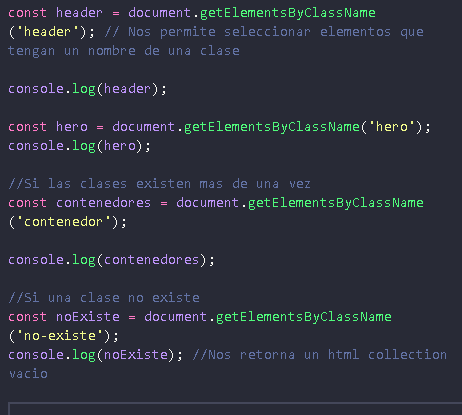


# DOM

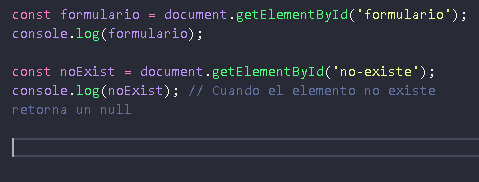
Acceder a elementos del HTML con document



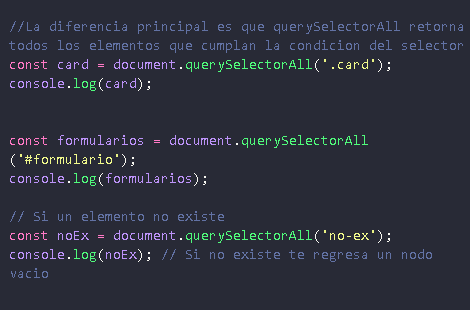
getElementsByClassName



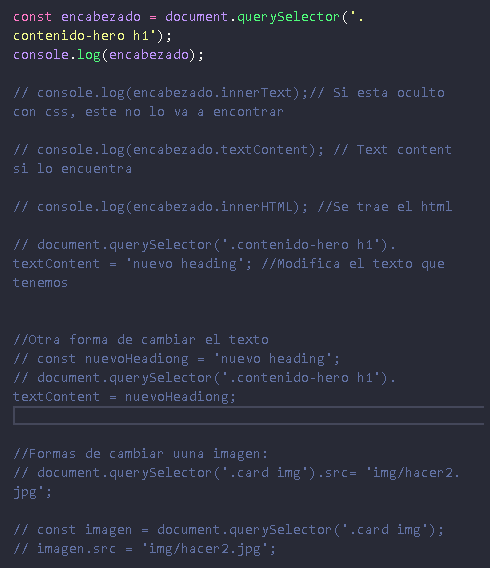
getElementById



querySelectorAll



Modificar Textos o Imagenes con JS



// innerText: Devuelve o establece el texto visible del elemento, considerando el estilo CSS.

// innerText te permite establecer o modificar el texto visible dentro de una etiqueta, respetando el formato visual del navegador, como saltos de línea y estilos CSS aplicados (incluso estilos que hacen que el texto sea invisible).

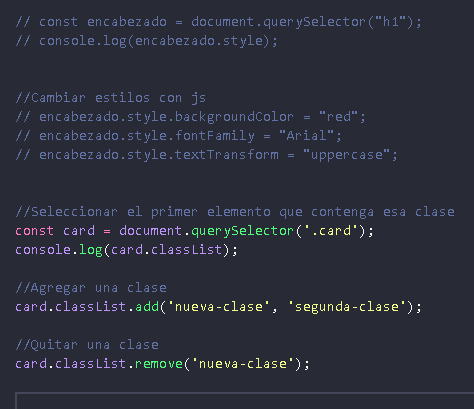
// textContent: Devuelve o establece el texto completo del elemento, ignorando el estilo CSS.

// textContent permite establecer o modificar todo el contenido de texto dentro de una etiqueta, sin importar su visibilidad o formato CSS. Es más rápido que innerText porque no desencadena el reflujo del DOM.

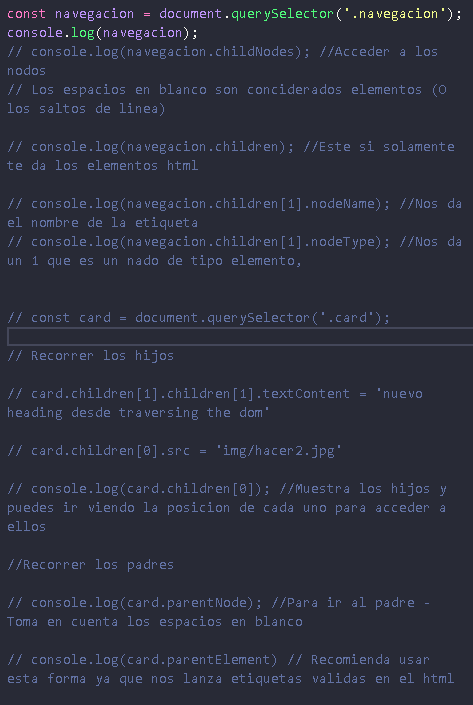
// innerHTML: Devuelve o establece el contenido HTML del elemento, incluyendo las etiquetas HTML.

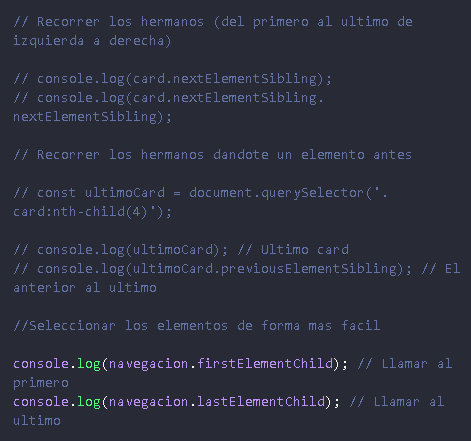
// innerHTML permite modificar el contenido HTML de una etiqueta, lo que incluye tanto el texto como las etiquetas HTML. Cuando se usa para establecer texto sin etiquetas HTML, funciona similar a textContent y innerText.

Cambiando el CSS con JavaScript

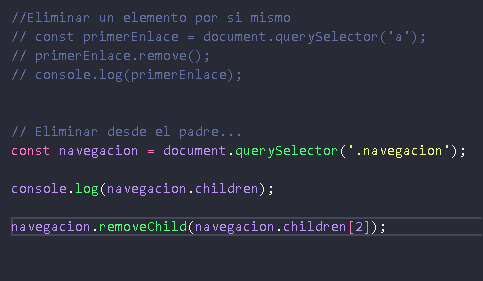


Traversing the DOM

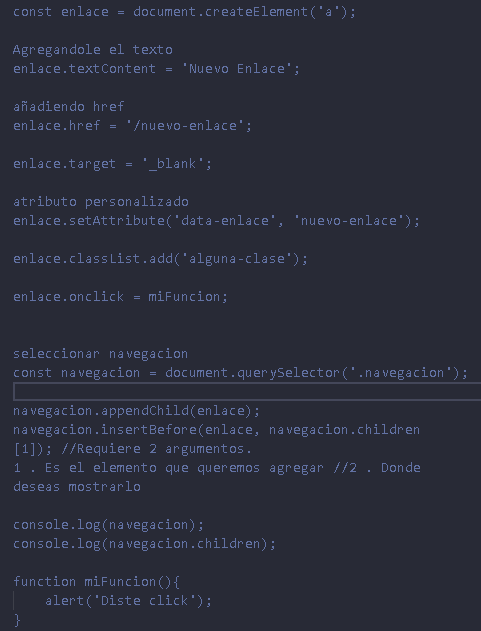


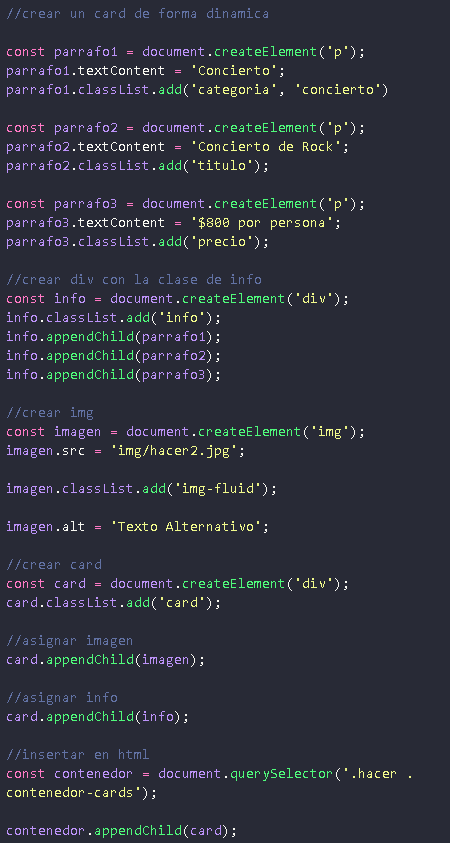


Eliminar elementos en el DOM

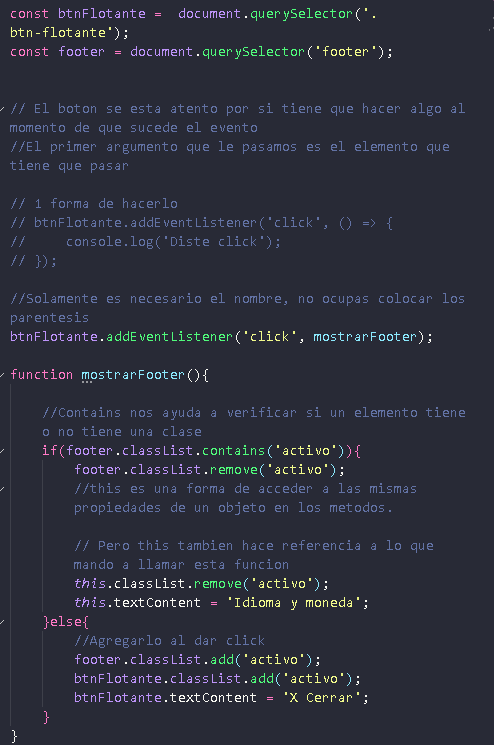


Generar HTML con JavaScript



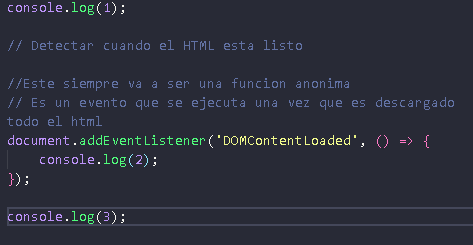


Ejemplo más avanzado de lo que podemos hacer con JS

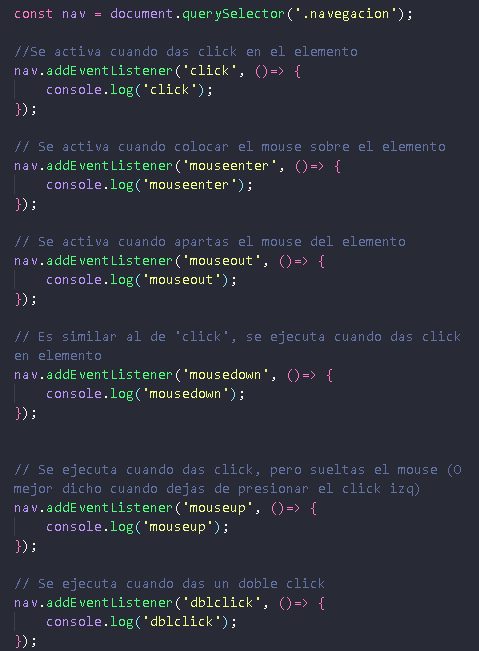


# EVENTOS

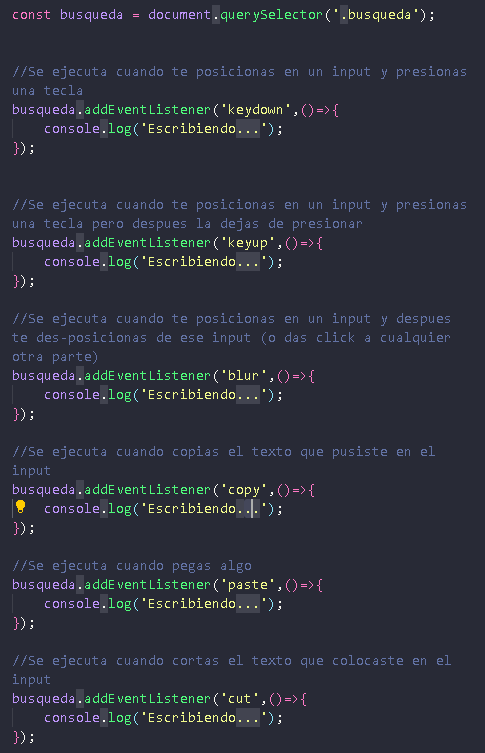
Detectar cuando el HTML esta listo

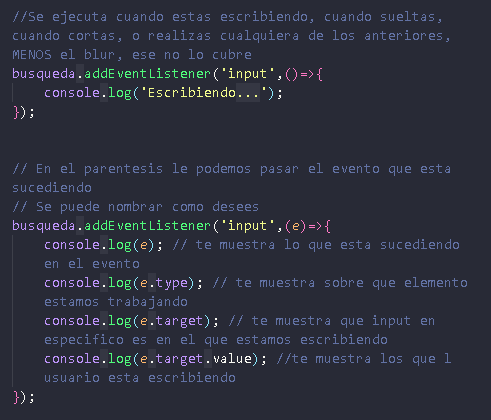


Eventos con el Mouse

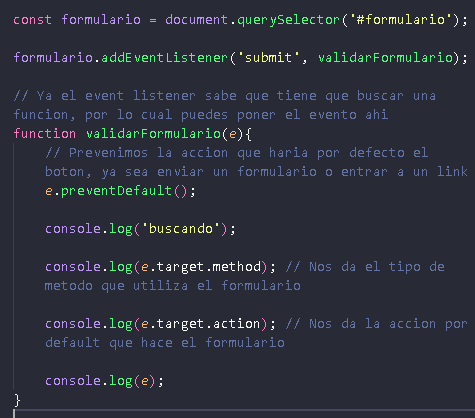


Eventos sobre los Inputs

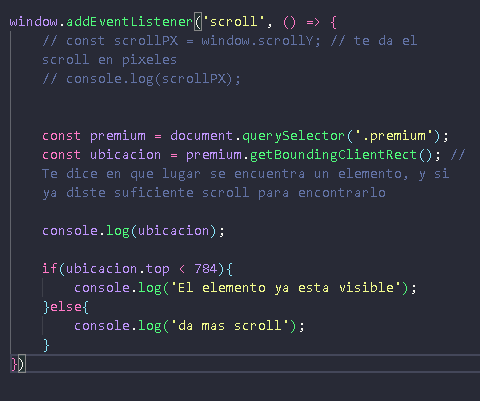




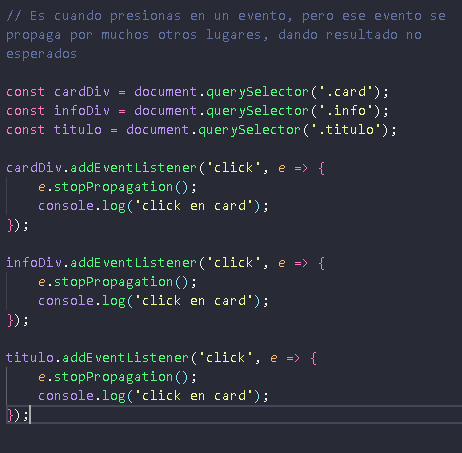
Evento Submit a un formulario



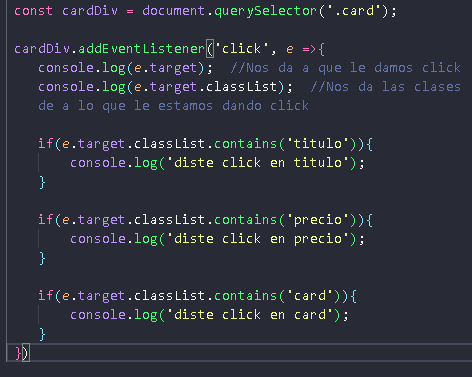
Eventos al dar scroll con el mouse



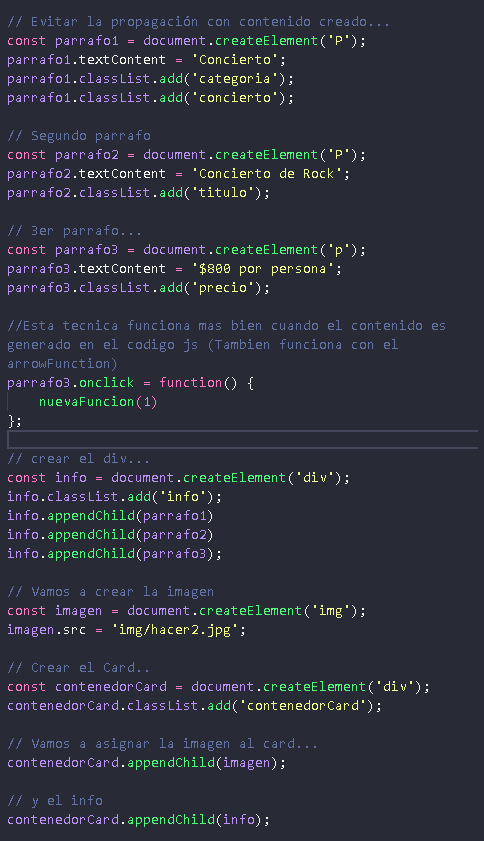
Qué es el Event Bubbling?

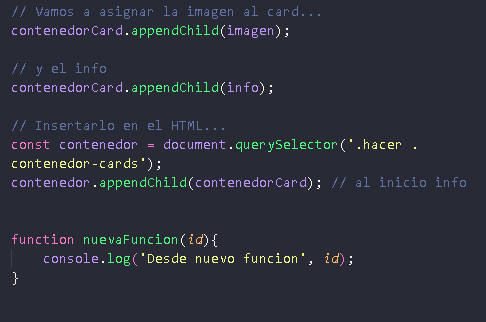


Prevenir Event Bubbling con Delegation



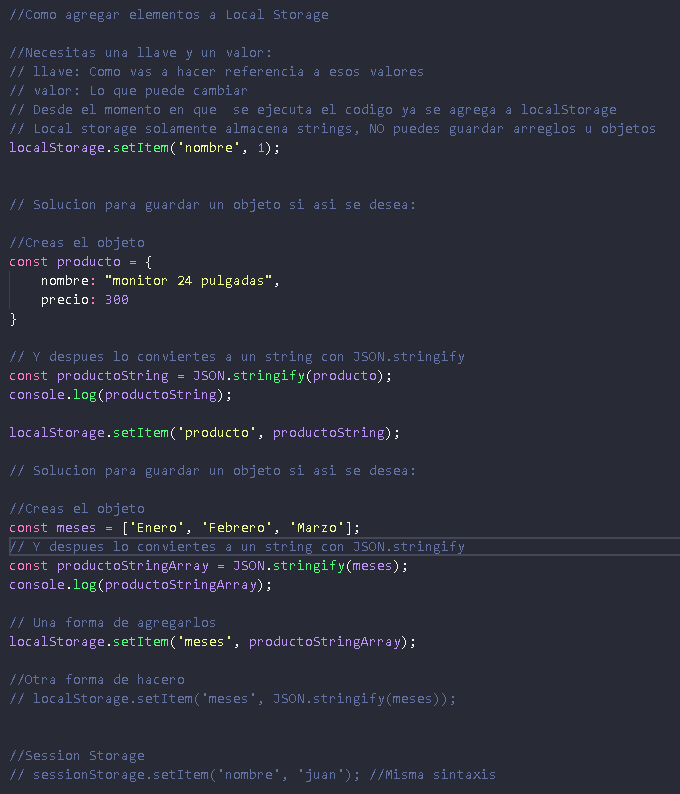
Prevenir Event Bubbling con un método

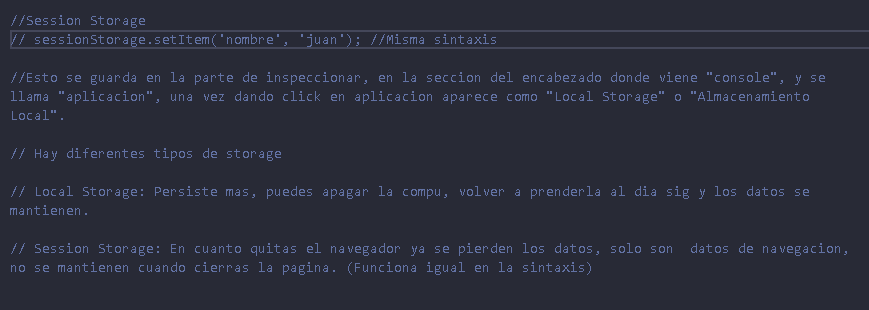




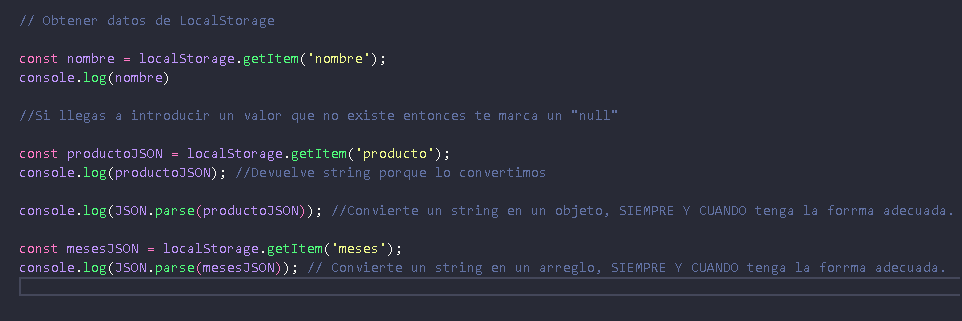
# LOCAL STORAGE

Primeros pasos con Local Storage

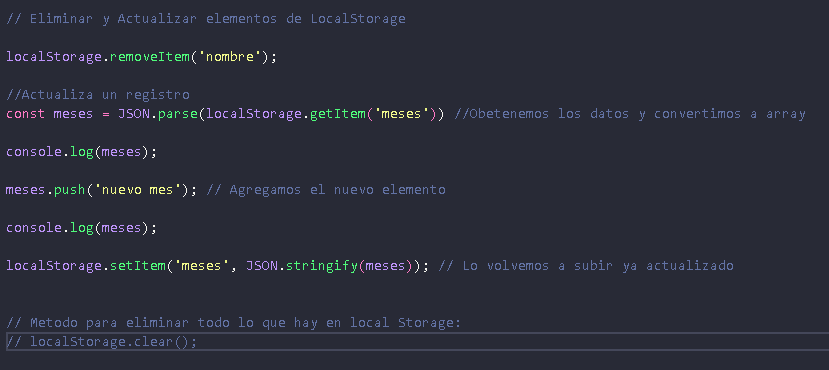




Obtener datos de LocalStorage

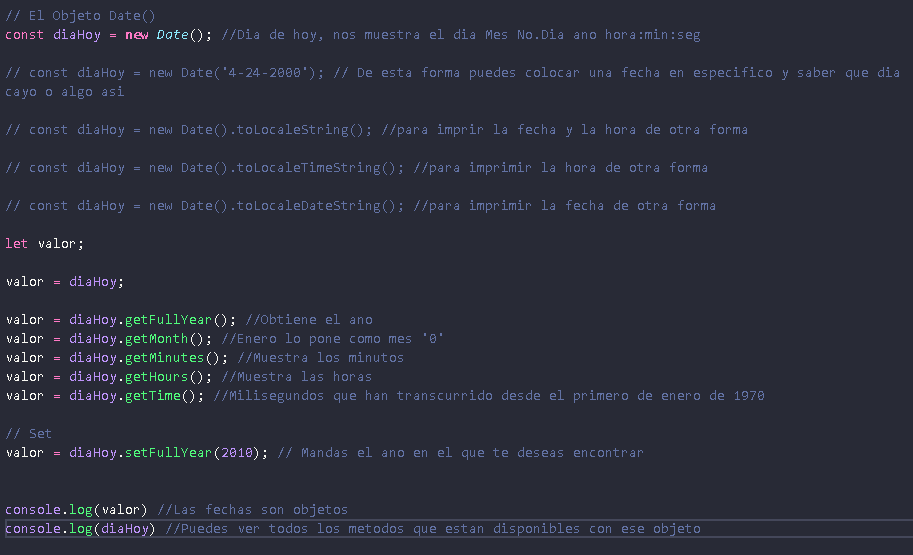


Eliminar y Actualizar elementos de LocalStorage

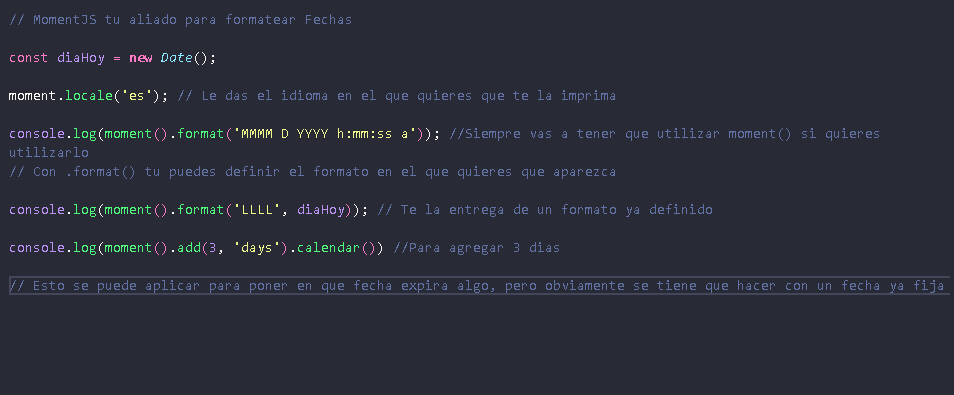


# Fechas

El Objeto Date()

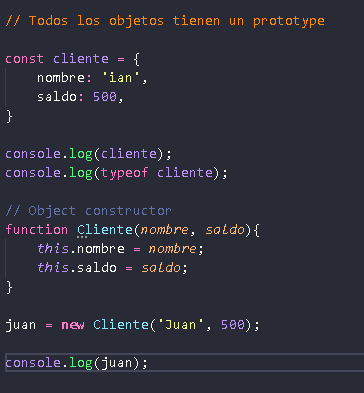


MomentJS tu aliado para formatear Fechas



# Prototypes

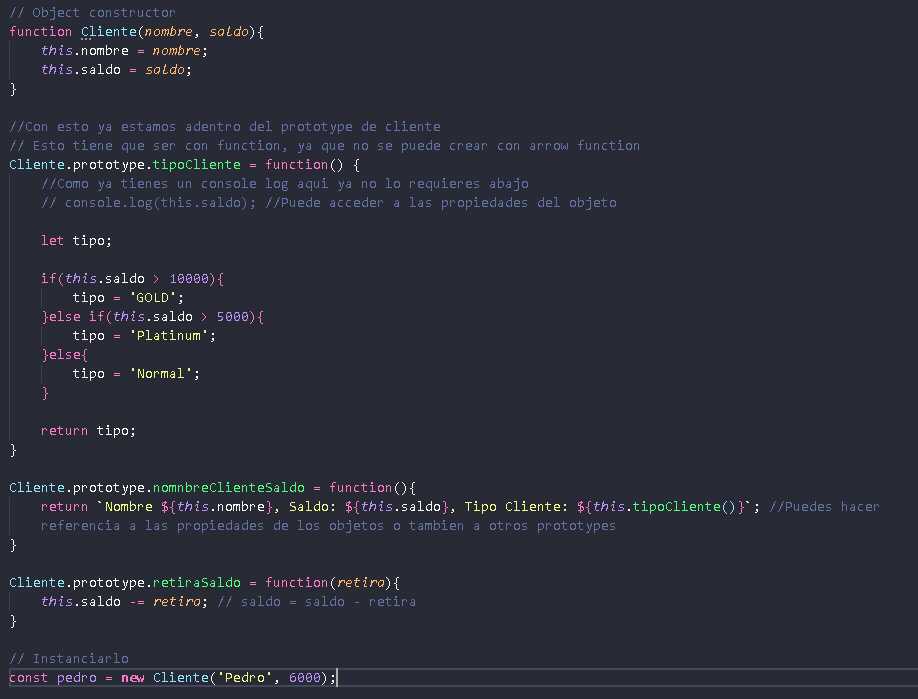
Veamos que es el Proto y crear un tipo de objeto Nuevo

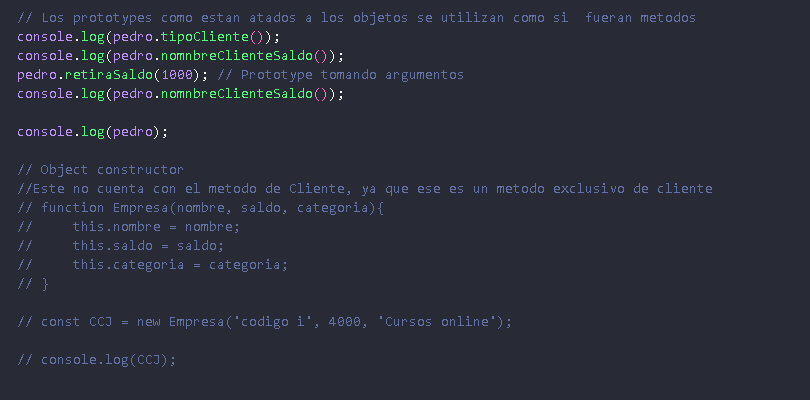


El Problema de no usar Prototypes

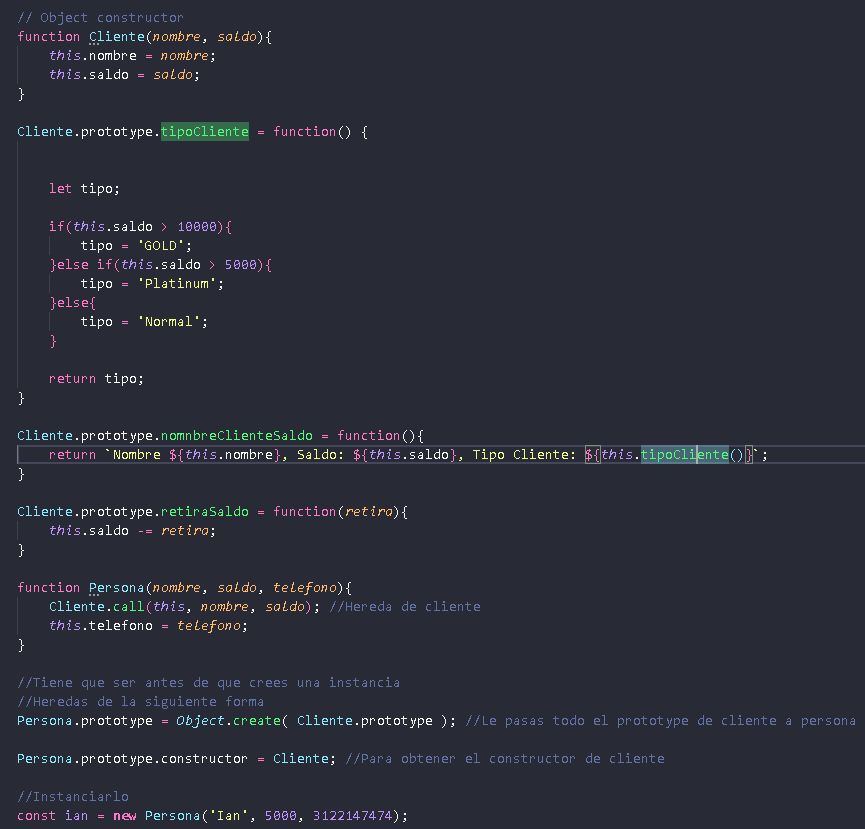


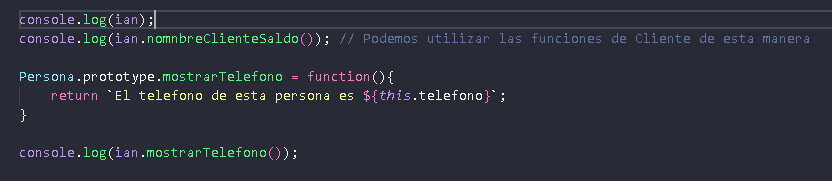
Creando un Prototype





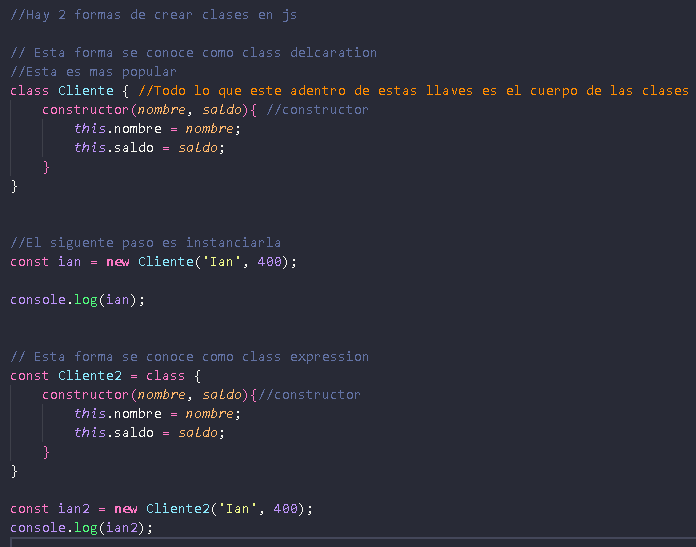
Heredar un Prototype



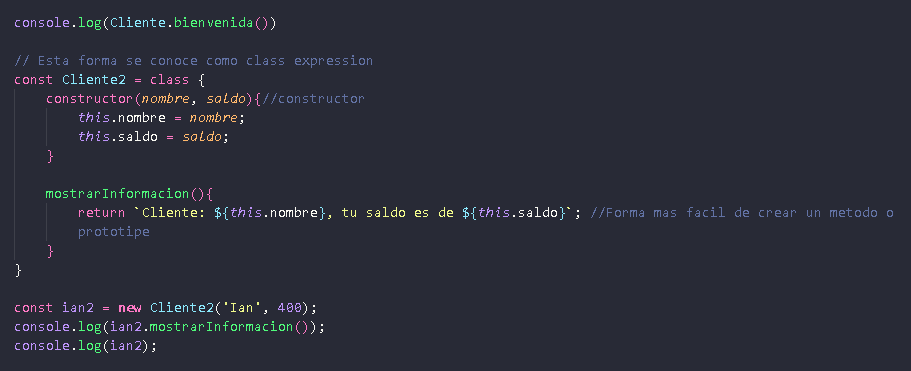
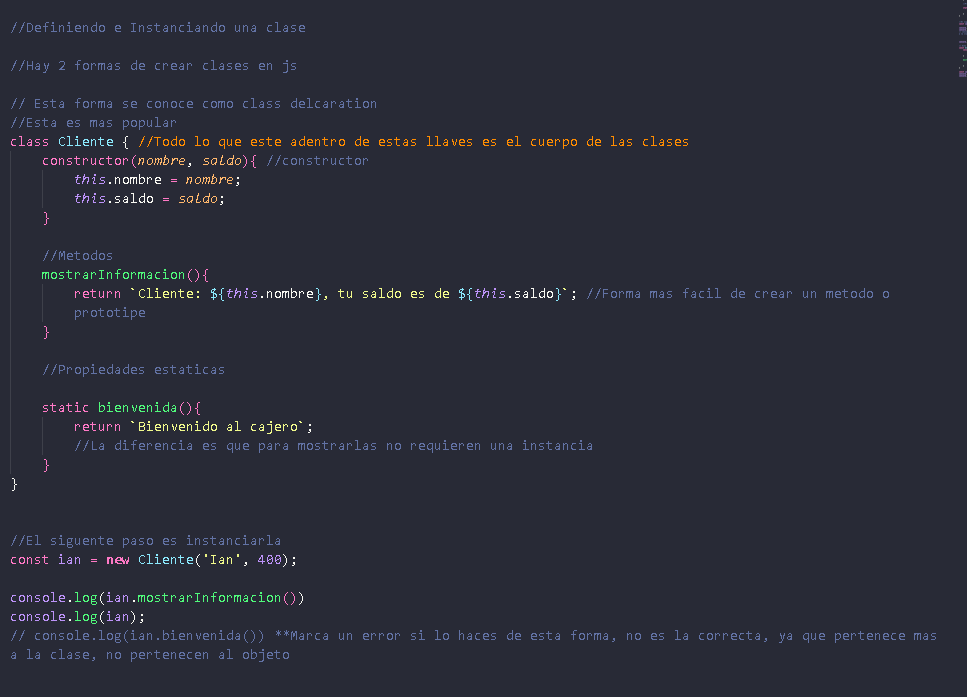


# POO

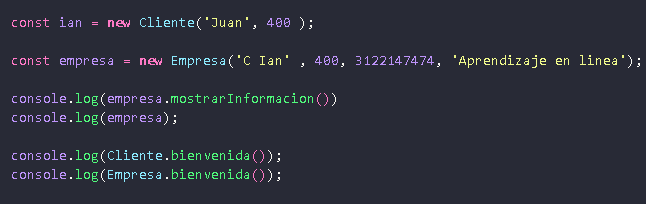
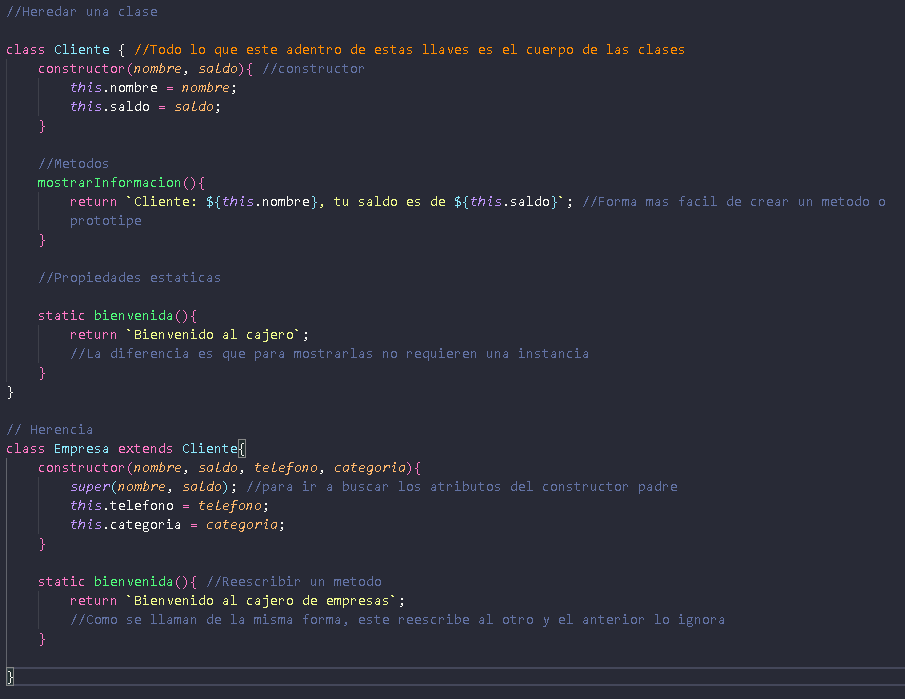
Definiendo e Instanciando una clase



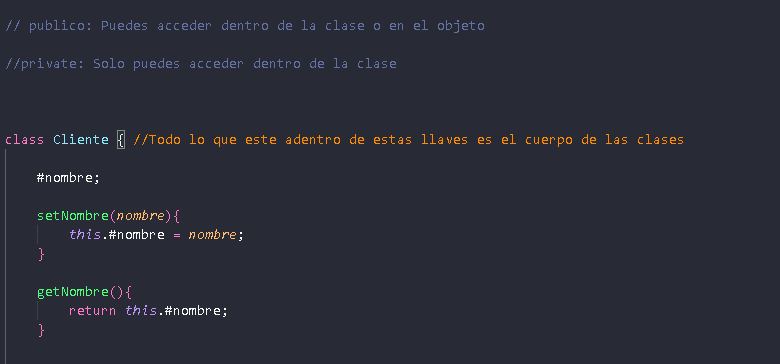
Métodos y Métodos estaticos en las classes

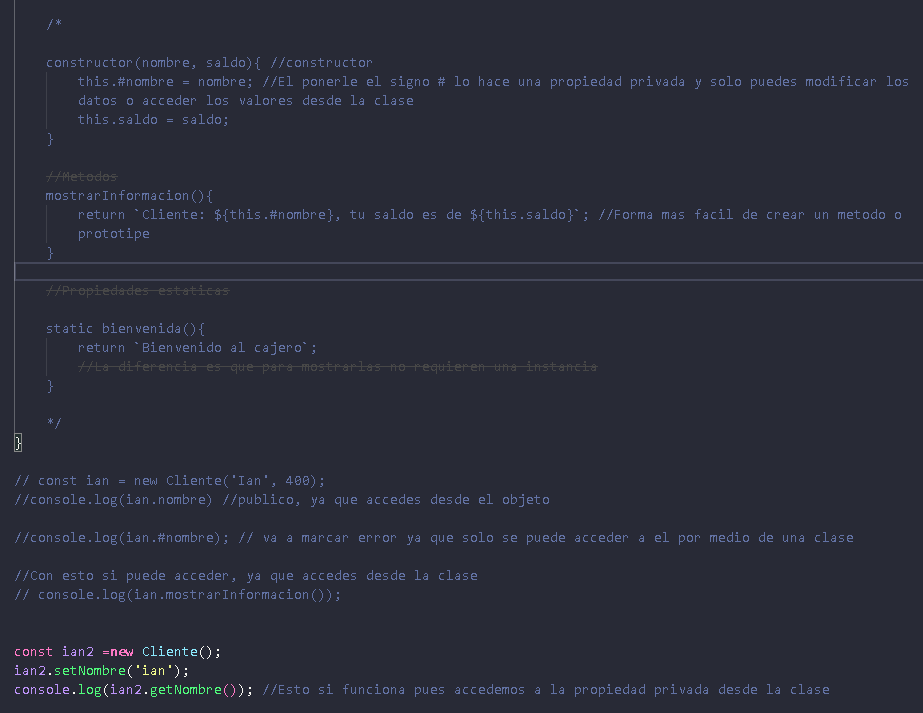


Heredar una clase



Propiedades Privadas en JavaScript





# Modulos

# 

Code:

// Para que no se mezclen las variables si es que tenemos varios codigos, es con un IIFE o una funcion que se ejecuta inmediatamente

//ifi:

//Esto mantiene estas variables o estas fucniones localmente en este archivo y no se van a mezclar con otros

//El problema es que si queremos tener un codigo mas ordenado y separar funciones no se puede hacer, pues van a estar encerradas en esta parte

(function(){

    console.log('desde IIFE');

    //Lo que hace window es que va a asignar el nombre del cliente a la ventana global, por lo que lo vas a poder utilizar

    window.nombreCliente = 'juan';

})()//Este parentesis de al final es lo que manda a llamar a la funcion inmediatamente.

**Caracteristicas de una PWA:**

Rápidas: Cargan toda la información en menos de 5 segundos.

Instalable: Se pueden navegar o instalar en tu navegador o teléfono móvil como una aplicación nativa

Soporte offline: pueden funcionar incluso sin conexión a internet

**Service workers:**

Es la base de una PWA. Son scripts que están corriendo todo el tiempo detrás de escenas.

Funciona Offline

No tienen acceso al DOM (Este no funciona con el DOM, para eso tendrías que tener tu script que funcione con el DOM, esto solo funciona en otra parte de la aplicacion)

Cargan de forma instantánea

Pueden sincronizar datos detrás de escena o sin interferir en la navegación

**Funciones no disponibles en Service Workers**

Window (utiliza self)

Document (Se utiliza caches)

localStorage (se utiliza fetch)

Estructura de manifest.json

{

    "name":"APV", (Nombre de la aplicacion)

    "short\_name":"APV", (Nombre corto para mas informacion)

    "start\_url": "/index.html", (Cual va a ser la pantalla de inicio una vez cargue el proyecto)

    "display":"standalone", (Como quieres que se vea la aplicación, si quieres que se vea la barra de navegación o no) (en este caso no la va a tener)

    "background\_color":"#D41872", (color de la aplicación)

    "theme\_color":"#D41872", (Tema de la aplicacion)

    "orientation":"portrait", (Orientacion)(esta aplicación solo va a funcionar de pie)

    "icons": [ (Diferentes iconos para la diferente resolusion)

    {

      "src": "img/icons/Icon-72",

      "type": "image/png",

      "sizes": "72x72"

    },

    {

      "src": "img/icons/Icon-120.png",

      "type": "image/png",

      "sizes": "120x120"

    },

    {

      "src": "img/icons/Icon-128.png",

      "type": "image/png",

      "sizes": "128x128"

    },

    {

      "src": "img/icons/Icon-144.png",

      "type": "image/png",

      "sizes": "144x144"

    },

    {

      "src": "img/icons/Icon-152.png",

      "type": "image/png",

      "sizes": "152x152"

    },

    {

      "src": "img/icons/Icon-196.png",

      "type": "image/png",

      "sizes": "196x196"

    },

    {

      "src": "img/icons/Icon-256.png",

      "type": "image/png",

      "sizes": "256x256"

    },

    {

      "src": "img/icons/Icon-512.png",

      "type": "image/png",

      "sizes": "512x512"

    }

  ]

}

**Que son los Desing Patterns**

Son soluciones típicas a problemas comunes en Desarrollo de software, cada patrón es como un plano que se puede personalizar para resolver un problema de diseño en el código.

**Beneficios**

Son soluciones a problemas de diseño de código

Son soluciones probadas

Son soluciones conocidas por todos, y evitan la forma de escribir código “como cada quien entiende”

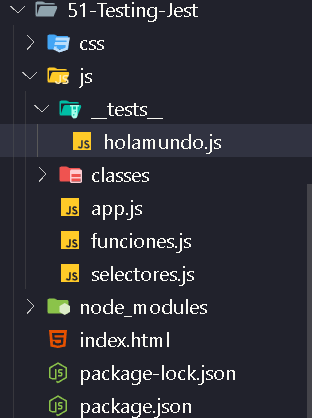
**Categorías de Patrones**

De creación – Permite crear objetos y permiten la re-utilizacion del código

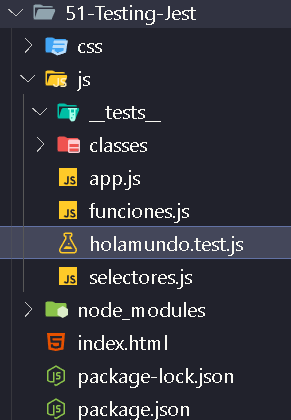
Estructura – Explican como deben comunicarse los objetos y classes en grandes proyectos

Comportamiento – Se encargan de como se comportan y comunican los objetos

Jest:

Puedes generar una carpeta para que jest entre automáticamente a ella y ejecute los códigos que tengas, ya que lo reconoce:  


O también puedes hacer que los ejecute nombrando los archivos a los que les vas a hacer un test con .test:



//Cada prueba utiliza la funcion llamada test

test('Hola mundo en Jest', () =>{

});

//Tambien puedes utilizar 'it' en vez de 'test' (hacen lo mismo)

it('Hola mundo en Jest', () =>{

});

//No vas a crear un archivo para cada prueba, puedes agruparlos dentro de una funcion que se llama 'describe'

describe('Grupo de pruebas', () => { //Y este archivo estaria realizando 2 pruebas

    //Es buena practica que pruebas que esten relacionadas esten en un solo archivo

    test('Hola mundo en Jest', () =>{

    });

    it('Otro Hola mundo en Jest', () =>{

    });

});

**Cypress:**

Es una herramienta para hacer testing de la “Siguiente generación” ofrece pruebas que antes no se podían hacer.

Cuando escribes pruebas son de código, pero no sabes que es lo que el usuario experimenta (con cypress es posible saber lo que está viendo el usuario en pantalla).

El sitio web donde puedes encontrar más información sobre cypress es: cypress.io

Tipos de pruebas en Cypress:

Es considerado end-to-end pero también ofrece pruebas unitarias y de integración.

Funciona en el navegador, por lo tanto puedes probar con Cypress proyectos hechos en Java, C#, Python, PHP, Node, React o VueJS.

Como iniciarlo?:

En lugar de escribir “npm start” escribimos “npx cypress open”

Cypress tiene otra funcionalidad que se ejecuta con “npx cypress run”, pero este se ejecuta desde la consola.

Con esto puedes descargar videos de como se realizan las pruebas y también puedes tomar screenshots para poder mostrarlas a los de producción.

Node o Node.js

**¿Que es?**  
Es un entorno de código abierto, multi-plataforma que permite crear aplicaciones del lado del servidor en javascript.

Permite crear api’s, aplicaciones web e incluso acceder a archivos y leer sus contenidos

**Ventajas de node:**

* Velocidad y rendimiento, node fue pensado para solucionar estos problemas, además es excelente para aplicaciones en tiempo real.
* Javascript, si ya tienes experiencia en JS, ahora podras crear aplicaciones fullstack con node sin necesidad de compiladores ni lenguajes extras.
* NPM (node package manager):  
  Es una gran cantidad de paquetes y librerías de código abierto ya listos, entre ellos encontraras subida de archivos, autenticación, enviar, emails, template engines, seguridad y mucho más.
* Intalar paquetes con npm:  
  Solo necesitas colocar: ‘npm install (nombre del paquete)’ o ‘npm i (nombre del paquete)’
* Funciona con windows, mac y Linux. Asi como con una gran cantidad de librerías, soporte, documentación, ejemplos y una comunidad.

**¿Que es express?**

Es el framework web mas popular de node, y es la herramienta sobre la que están desarrollados otros frameworks como Mean, Sails, Loopback, Graphql, Yoga y otros.

Caracteristicas de express:

Soporta los diferentes verbos de http como son: post, get, put, patch, delete en las rutas (routes).

Soporta ‘vistas’ que son los datos mostrados en la pantalla mediante determinadas respuestas.

Permite la creación de aplicaciones MVC (Model View Controller) que permiten tener una separación de código.

Soporta middleware, mediante peticiones que se ejecutan en la tubería de la petición.

Middleware en express:  
Se utiliza bastante en express, ya que permite acceder a archivos, revisar si un usuario esta autenticado o no, entre otras.

Ejemplo:  
router.get(‘/administracion’,

authController.usuarioAutenticado, (que al entrar a la pagina, verifique si esta autenticado)

authController.panelAdministracion); (si esta autenticado, que muestre el panel de administracion)

El middleware se ejecuta en pila, por lo tanto, se ejecuta uno y después otro.

En algunos casos el middleware lo creas tu (similar a crear tus propias funciones) p también esta disponible en express una gran cantidad de middlewares.

Desventajas de express:

A diferencia de otros frameworks como Django o Laravel; express no viene con “baterías incluidas” como puede ser autenticación, roles, ORM o generador de modelos, estas piezas usualmente se instalan vía NPM.

De esta forma solo instalas lo que necesitas en tu aplicación (Lo cual puede ser una ventaja).

Lo primero que vas a hacer siempre en todos los proyectos de node es crear un archivo que se llama “package.json”

Este archivo tiene 2 fines:

Primero > tener todas tus dependencias del proyecto (las dependencias son como los ‘instalables’ que se necesitan si quieres mostrar algunas cosas en pantalla, mandar algo al servidor, etc. Esas dependencias tienen una versión que gracias al package.json todos van a poder trabajar con esa versión.

Para instalarlo se usa lo siguiente:  
npm init

Despues te aparecerán una serie de preguntas:  
El nombre del package

Version

Descripcion

Punto de entrada

Comando de text

Repositorio de git

Autor

Licencia

Todo esta bien? (yes)

**Dependencias**

Hay 2 tipos de dependencias:

Las de desarrollo: que son las que requieres cuando solamente estas creando el proyecto.

Las de producción: que son las que se van a requerir ya cuando el proyecto este publicado

**Dependencia Nodemon (dependencia de desarrollo): (naturalmente las dependencias de desarrollo se instalan colocando --save-dev )**

Como vamos a estar haciendo muchos cambios en desarrollo, es decir guardando cambios y agregando mas funciones, nodemon lo que va a hacer es detectar los cambios en esos archivos y va a reiniciar el servidor (de otra forma tendríamos que ir a la terminal, detener el servidor y volverlo a arrancar).

Ejemplo:  
{

  "name": "agenciaviajesnode",

  "version": "1.0.0",

  "description": "Proyecto con Node.js",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "author": "",

  "license": "ISC",

  "dependencies": { (estas se suben a produccion)

    "express": "^4.21.1"

  },

  "devDependencies": { (estas no se suben en produccion, pero se pueden usar localmente)

    "nodemon": "^3.1.7"

  }

}

Esto porque no vas a realizar cambios al servidor al momento de estar modificando el código.

Para arrancar el servidor, utilizamos el package.json, y en la parte de script, colocamos: dev (porque es como “arrancar el desarrollo”)

(Si te preguntas de donde sale nodemon, viene en: node\_modules > bin > nodemon)

{

  "name": "agenciaviajesnode",

  "version": "1.0.0",

  "description": "Proyecto con Node.js",

  "main": "index.js",

  "scripts": {

    "dev":"nodemon index.js"

  },

  "author": "",

  "license": "ISC",

  "dependencies": {

    "express": "^4.21.1"

  },

  "devDependencies": {

    "nodemon": "^3.1.7"

  }

}

Para mandar a llamar ese script lo tienes que hacer desde la terminal:

npm run dev <- (que ‘dev’ es el nombre del script)

(eso va a ejecutar cualquier script que tengas en package.json)

Despues lo puedes ver con:  
localhost:4000

Sintaxis de common.js

//Este va a ser nuestro archivo sobre el cual vamos a configurar express

//Se importa express:

const express = require('express');

//Este contiene una funcion para ejecutar express que se le asigna a app

const app = express();

//Definir puerto

//Esta variable de entorno con el puerto nos la va a asignar usualmente donde hagamos el deployment, esto porque no sabemos que puerto va a haber disponible y lo va a asignar automaticamente.

//En local process.env.PORT no existe, por lo tanto se ejecuta en el puerto 4000

//Una vez en deployment ya no se usa 4000, se usa un puerto disponible (no se sabe cual te va a tocar)

const port = process.env.PORT || 4000;

//Arrancamos el servidor con el metodo .listen

app.listen(port, () =>{

    console.log(`El servidor esta funcionando en el puerto ${port}`);

})

Sintaxis con imports y exports

import express from 'express';

const app = express();

const port = process.env.PORT || 4000;

app.listen(port, () =>{

    console.log(`El servidor esta funcionando en el puerto ${port}`);

})

Y para que te lo respete tienes que colocar en el package.json el “type”:

{

  "name": "agenciaviajesnode",

  "version": "1.0.0",

  "description": "Proyecto con Node.js",

  "main": "index.js",

  "scripts": {

    "dev":"nodemon index.js"

  },

  "type": "module",

  "author": "",

  "license": "ISC",

  "dependencies": {

    "express": "^4.21.1"

  },

  "devDependencies": {

    "nodemon": "^3.1.7"

  }

}

Templete Engine:

**Que son?**

Son la V (vista) del MVC.

Permiten mostrar la parte visual (html) en una aplicación express, debido a que el modelo retorna un objeto (o arreglo) de datos, un template engine permitirá acceder a los resultados de una consulta y mostrarlos en pantalla.

Características de un template engina

Hay una gran variedad y cada uno tiene su propia sintaxis.

Usualmente puedes escribir código JS dentro del HTML.

Si tienes experiencia en Angula, react o vue, usualmente reemplazan estos template engines en una aplicación.

Los template engines mas comunes en express:  
PUG – (antes Jade)

EJS – Embedded Javascript

HBS – Handlebars.js (mustache.js)

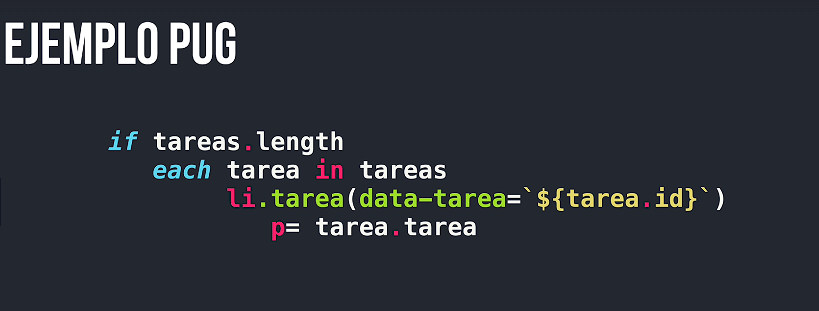
React

Como se instala un template engine?

Un template engine se instala via NPM, colocando:  
npm install pug

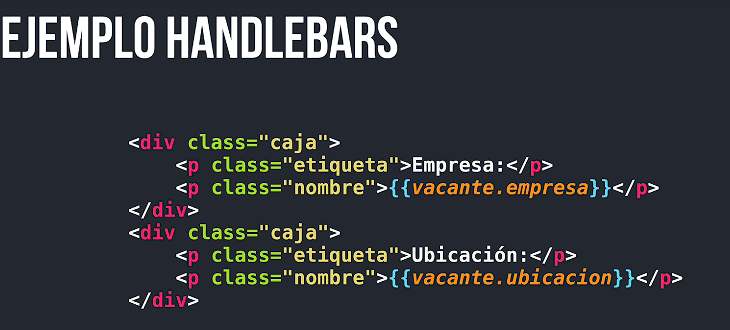
Y se habilita en el archivo principal colocando:

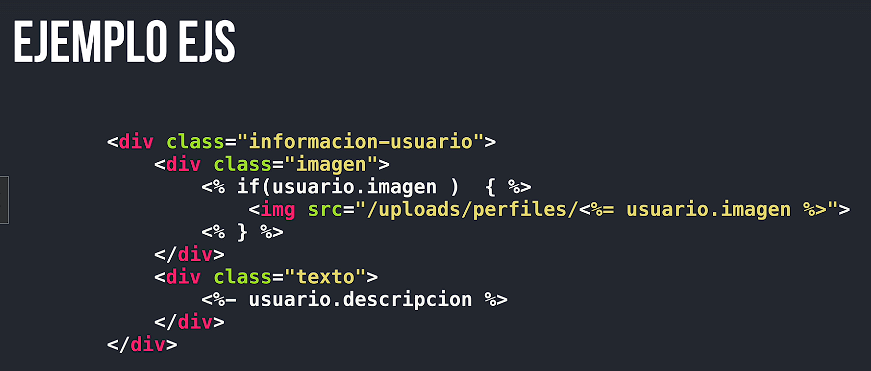
app.set(‘view engine’, ‘pug’);

Ejemplo:  


La identacion es muy importante en pug y se te va a permitir ir creando bloques de código.

Sintaxis de handlebars:



Sintaxis de EJS:  


Express utiliza request y response para enviar y recibir datos:

//express en el callback utiliza lo que es el request y el response:

//request : lo que tu envias (como enviar un formulario, etc)

//response : lo que express te responde (como las alertas)

//app.get('/',(req,res) =>{ //Envias una peticion de tipo get, es decir cuando visitamos esa pagina

//    res.send('Inicio'); //tambien puedes crear tu propia respuesta

    //.send es un metodo utilizado para mostrar algo en pantalla

    // res.json({

    //     id:1

    // }); //tambien puedes obtener una respuesta json

    //res.render(); // se utiliza para mostar una vista

//})

// app.get('/nosotros',(req,res) =>{

//     res.send('Nosotros');

// })

// app.get('/contacto',(req,res) =>{

//     res.send('Contacto');

// })

Routing en express:  
Se crea una carpeta de preferencia y solamente puede haber una instancia de nuestra aplicación.

Por lo que hacemos lo siguiente:  
import express from 'express';

//Aqui estaremos colocando todo lo relacionado a las rutas

const router = express.Router(); // De esta forma estaremos utilizando la misma instancia de express (que se encuentra en el index.js afuera de la carpeta), pero estamos utilizando su router

Y de esa forma abajo ya podemos declarar las rutas:

router.get('/',(*req*,*res*) =>{ //Envias una peticion de tipo get, es decir cuando visitamos esa pagina

    //res.send('Inicio'); //tambien puedes crear tu propia respuesta

    //.send es un metodo utilizado para mostrar algo en pantalla

    // res.json({

    //     id:1

    // }); //tambien puedes obtener una respuesta json

    //res.render(); // se utiliza para mostar una vista

*res*.render('inicio');

});

router.get('/nosotros',(*req*,*res*) =>{

    //res.send('Nosotros');

    const viajes = 'Viaje a Alemania'; //variable

*res*.render('nosotros', {

        //Aqui puedes poner toda la informacion que requieras mandar hacia la vista

        textoViajes:viajes

    }) //render espera el nombre de una vista

});

router.get('/contacto',(*req*,*res*) =>{

*res*.send('Contacto');

});

Y para poder utilizarlo lo exportamos a la pagina principal:

export default router;

Lo que daría como resultado lo siguiente:

import express from 'express';

//Aqui estaremos colocando todo lo relacionado a las rutas

const router = express.Router(); // De esta forma estaremos utilizando la misma instancia de express (que se encuentra en el index.js afuera de la carpeta), pero estamos utilizando su router

(si quieres enviar datos, se utiliza set y para eliminar se utiliza delete)

router.get('/',(*req*,*res*) =>{ //Envias una peticion de tipo get, es decir cuando visitamos esa pagina

    //res.send('Inicio'); //tambien puedes crear tu propia respuesta

    //.send es un metodo utilizado para mostrar algo en pantalla

    // res.json({

    //     id:1

    // }); //tambien puedes obtener una respuesta json

    //res.render(); // se utiliza para mostar una vista

*res*.render('inicio');

});

router.get('/nosotros',(*req*,*res*) =>{

    //res.send('Nosotros');

    const viajes = 'Viaje a Alemania'; //variable

*res*.render('nosotros', {

        //Aqui puedes poner toda la informacion que requieras mandar hacia la vista

        textoViajes:viajes

    }) //render espera el nombre de una vista

});

router.get('/contacto',(*req*,*res*) =>{

*res*.send('Contacto');

});

export default router;

Despues desde la pagina principal, agregamos router:

//Agregar router

app.use('/', router); //Use soporta get, post, put, patch y delete y todos los diferentes verbos que hay

//Y esto lo que hace es que desde la pagina principal, agrega inicio, contacto y nosotros, y todas las diferentes rutas que vayamos definiendo

Habilitar pug:

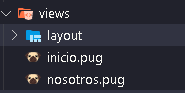
Pug se instala por el lado de producción, es decir no ocupas el –dev-save:  
npm install pug

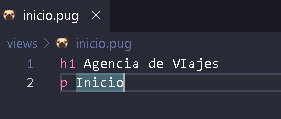
Se habilita en index:

//Habilitar pug

app.set('view engine', 'pug');

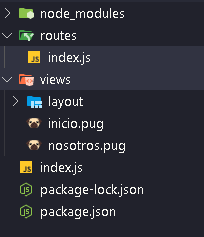
Despues puedes crear una carpeta en donde guardar las vistas, y todas las vistas de ese archivo que quieras que sean con pug, tienen que tener la extencion ‘.pug’.



La sintaxis de pug es la siguiente:  


No espera etiquetas de cierre, solamente que tienes que seguir la identacion para que pug entienda de cual es el componente padre y cual es el hijo

Despues, asociado con el routing, necesitamos que busque y encuentre el archivo de pug, por lo tanto:



router.get('/nosotros',(*req*,*res*) =>{

    //res.send('Nosotros');

(puedes definir variables que después se las puedes enviar a pug)

    const viajes = 'Viaje a Alemania'; //variable

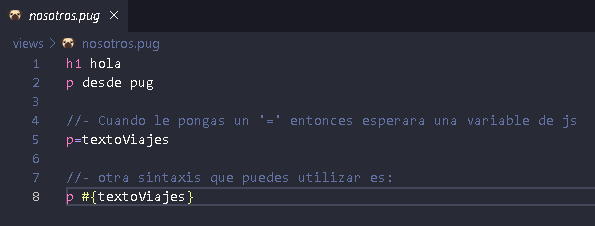
*res*.render('nosotros', { (escanea y busca el archivo para mostrarlo)

        //Aqui puedes poner toda la informacion que requieras mandar hacia la vista

        textoViajes:viajes

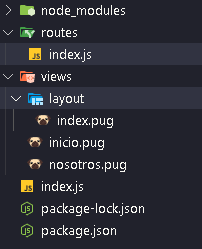
    }) //render espera el nombre de una vista

});

Como puedes ver le pasamos una variable, en este caso “textoViajes” que contiene lo que hay en “viajes” definida arriba. Y la podemos llamar de 2 formas:  


Creando un Layout Principal (o master page) en pug:

Creamos una carpeta layout, que Incluye estructuras como el encabezado, pie de página, menús de navegación y barras laterales, que se muestran en todas las páginas de la aplicación.



Y la pagina principal del layout, en donde vamos a mostrar el encabezado etc, la llamaremos como index.

Ahí tendremos que poner una estructura doctype:

doctype html

html(*lang*="en")

    head

        meta(*charset*="UTF-8")

        meta(*name*="viewport", *content*="width=device-width, initial-scale=1.0")

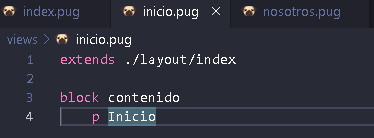
        title Document

    body

        h1 Agencia de Viajes (esto será lo que se muestre en todas las paginas)

        block contenido (esto será lo que cada pagina contenga)

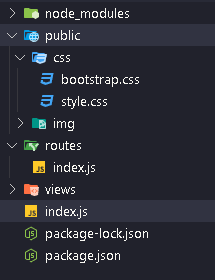
Pug utiliza esto que se llaman ‘bloques’ y tu puedes definir tus propios bloques

Y para que en inicio.pug se muestre lo que hay en el layout, tenemos que colocar la siguiente sintaxis:  


No necesitas la extencion de index, ya que los 2 son un archivo .pug

Creando hojas de estilo:

Las imágenes y hojas de estilo naturalmente se van a colocar en una carpeta llamada ‘public’.



Despues definimos la carpeta publica en nuestro archivo principal:

//Definir la carpeta publica:

app.use(express.static('public'));

Como colocar imágenes en pug:

img(*src*="img/cupon.jpg")

Como agregarle clases a los div en pug:  
Forma 1:

div(*class*="navegacion")

Forma 2:

div*.navegacion*

Forma 3: (te crea el div por default)

*.navegacion*

Clases anidadas:

*.row.justify-content-center.justify-content-md-between.aling-items-center.py-4*

Como mandar una variable a distintos archivos utilizando middlewares:

app.use((*req*,*res*, *next*)=>{ //Ese next lo que hace es ir al siguiente middleware

*res*.locals.unaVariable = "una nueva variable"; //Locals son como variables internas de express, al crear una variable aqui, como es el archivo principal, se puede utilizar en todas las vistas (Y ya simplemente escribes el nombre de la variable en cualquiera de los archivos que la quieras utilizar)

    console.log(*res*.locals); //Te va a mostrar varias salidas pq use se ejecuta en diferentes metodos

    //console.log(res); //te muestra en consola todo lo que abarca la respuesta, si colocas req, tambien te muestra las funcionalidades de este

    return next(); //Pasa al siguiente middleware (return es como un !important, lo obliga a hacer la accion)

})

**Ahora para instalamos unas dependencias que nos van a permitir conectar a la base de datos, pero también interactuar:**

npm install mysql2 sequelize (sequelize seria un orm)

**Después creamos una carpeta llamada config:**

Dentro de esta carpeta creamos un archivo llamado db.js en donde va ir la configuración de nuestra base de datos.

En donde ira la siguiente configuración:

import { Sequelize } from "sequelize";

const db = **new** Sequelize('agenciaviajes', 'root','root', {

    host: '127.0.0.1',

    port: '3306',

    dialect: 'mysql',

    define:{

        timestamps: false //Porque tiende a agregar un par de columnas cuando fue creado y cuando fue actuaklizado un registro

    },

    pool:{ //Configuracion de sequeliza

        max:5,

        min: 0,

        acquire: 30000,

        idle: 10000

    },

    operatorsAliases: false

}); //Y tienes que pasarle los siguientes valores: El primero es el nombre de la base de datos a la cual te quieres conectar, El segundo es el nombre del usuario, El tercero es el password, El cuarto es una serie de configuraciones.

export default db;

Despues lo importamos a nuestro index principal:

import db from './config/db.js'

Y lo conectamos:

//Conectar a la base de datos

db.authenticate()

    .then( ()=> console.log('Base de datos Conectada'))

    .catch( *error* => console.log(*error*));

Model – View – Controller

**Que** **es?**

- Es un patron de diseño de software que permite la separación de obligaciones de cada pieza de código en tu aplicación web.

- Enfatiza la separación de la lógica de programación y lo que se muestra en pantalla

**Model**:

Encargado de los datos (desde una base de datos) y de la lógica para mostrar esos datos.

**View:**

Se encarga de lo que se ve en pantalla (HTML)

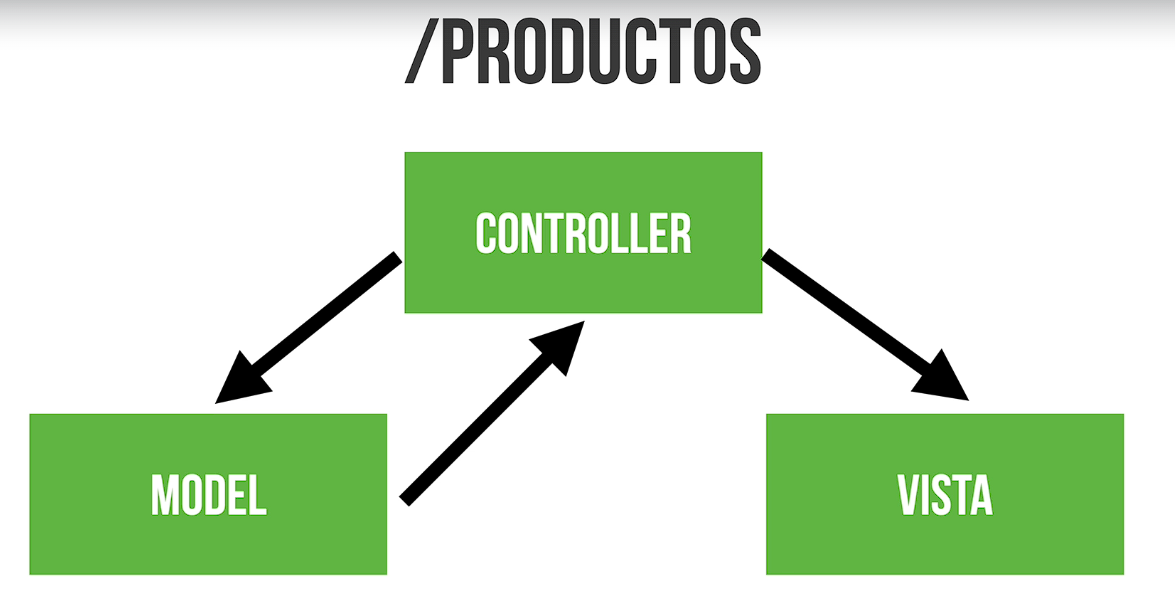
**Controller:**

Es el que se comunica entre el modelo y la vista, antes de que el modelo consulte a la base de datos es el encargado de mandarlo llamar, y también una vez que el modelo tiene los resultados de la consulta, es el que se encarga de pasarlos a la vista.

**Router:**

Encargado de registrar todas las url’s o endpoints que la aplicación soporta.

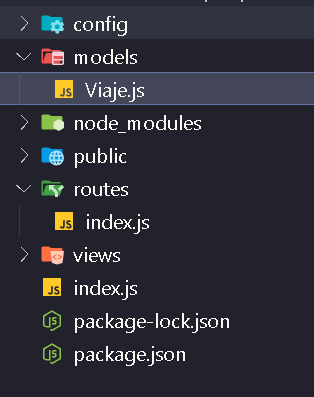
Si el usuario accede a /productos, el router llama un controlador, que se comunica con el modelo para obtener los datos que son pasados hacia la vista para ser mostrados.



1. Al entrar a la pagina se accede al controlador
2. El controlador se comunica con el modelo y le pide los datos
3. El modelo se los regresa al controlador
4. Y el controlador se lo pasa hacia la vista

**Definir el modelo:**  
**Creamos una carpeta llamada models:**

Dentro de esta carpeta creamos un archivo js (que no es obligatorio, pero se recomienda que la primera letra sea mayúscula).



**Definir Controlador:**Creamos una carpeta llamada “controllers”:

En donde adentro colocamos el archivo js en donde se van a manejar los controladores (las paginas que queremos que se muestren)

**Así es como definimos un controlador:**

const paginaTestimoniales = (*req*,*res*) =>{

*res*.render('testimoniales', {

        pagina: 'Testimoniales'

    });

};

**Y después lo exportamos:**

**export {**

    paginaInicio,

    paginaNosotros,

    paginaViajes,

    paginaTestimoniales

}

**Nos vamos a nuestro archivo principal en donde se definen las rutas, importamos y pasamos los valores:**

import { paginaInicio,paginaNosotros,paginaViajes,paginaTestimoniales } from '../controllers/paginasControllers.js';

//Aqui estaremos colocando todo lo relacionado a las rutas

const router = express.Router(); // De esta forma estaremos utilizando la misma instancia de express (que se encuentra en el index.js afuera de la carpeta), pero estamos utilizando su router

router.get('/',paginaInicio);

router.get('/nosotros',paginaNosotros);

**Para consultar la base de datos se hace lo siguiente:**

const paginaViajes = async (*req*,*res*) =>{

    //Consultar base de datos

    const viajes = await Viaje.findAll();

    console.log(viajes);

*res*.render('viajes', {

        pagina: 'Viajes',

        resultado: viajes

    });

};

Leer datos en un formulario:

Primero que nada creas tu form, este es un ejemplo de como hacerlo:

block contenido

    main*.container.mt-5*

        h1*.text-center.mt-5* #{pagina}

*.row*

            h2*.d-block.w-100.text-center* Agrega un Testimonial

*.row.justify-content-center.col*

*.col-md-8*

                    //- El metodo POST, es la forma en la que envias datos a un servidor

                    //- En el action tenemos que definir hacia donde queremos enviar los datos

                    form(*action*="/testimoniales" *method*="POST")

                        //- El for sirve para que cuando presiones en el label se active el input, por lo cual el id y el for tienen que ser iguales

                        //- la etiqueta 'name', al momento de imprimir en consola con req, te va a paracer el campo con ese nombre

*.form-group*

                            label(*for*="nombre") Nombre

                            input*#nombre*(*type*="text", *placeholder*="Tu Nombre", *name*="nombre", *class*="form-control")

*.form-group*

                            label(*for*="correo") Correo Electronico

                            input*#correo*(*type*="text", *placeholder*="Tu Correo", *name*="correo", *class*="form-control")

*.form-group*

                            label(*for*="mensaje") Mensaje

                            textarea*#mensaje*(*name*="mensaje", *class*="form-control",*placeholder*="Tu Mensaje",*rows*=3)

                        input*.btn.btn-success.btn-block*(*type*="submit")

Despues defines en ‘routes’ que vas a enviar datos al servidor (no te olvides de importar):

router.post('/testimoniales',guardarTestimonial); //Post sirve para que se ejecute la funcion cuando se envien los datos

Y creas un controlador para el momento en el que envíes los datos (en donde también pues validar los datos):

const guardarTestimonial = (*req*, *res*) =>{

    //Validar

    const {nombre,correo,mensaje} = *req*.body; //req.body tiene toda la información cuando la envias, es por eso que obtienes nombre,correo y msj

    const errores = []

    if(nombre.trim() === ''){

        errores.push({mensaje: 'El nombre esta vacio'})

    }

    if(correo.trim() === ''){

        errores.push({mensaje: 'El correo esta vacio'})

    }

    if(mensaje.trim() === ''){

        errores.push({mensaje: 'El mensaje esta vacio'})

    }

    console.log(errores)

    //console.log(req.body); //req.body va a ser lo que el usuario coloque en el formlario

}

export {

    guardarTestimonial

}

También, para que el ‘req.body’ funcione y pueda leer los datos, tenemos que agregar lo siguiente al index principal (de esta forma ya aparecerá en consola):

//Agregar body parser para leer los datos del forulario

app.use(express.urlencoded({extended:true}));

Validación de Formularios y mostrar alertas:

const guardarTestimonial = (*req*, *res*) =>{

    //Validar

    const {nombre,correo,mensaje} = *req*.body;

    const errores = [] //Creas un array donde guardar los errores

    if(nombre.trim() === ''){

        errores.push({mensaje: 'El nombre esta vacio'})

    }

    if(correo.trim() === ''){

        errores.push({mensaje: 'El correo esta vacio'})

    }

    if(mensaje.trim() === ''){

        errores.push({mensaje: 'El mensaje esta vacio'})

    }

    if(errores.length > 0){

        //Mostrar la vista con errores

*res*.render('testimoniales', { // Los mandas a la pagina

            pagina: 'Testimoniales',

            errores: errores,

            nombre,

            correo,

            mensaje

        });

    }else{

        //Almacenarlo en la base de datos

    }

    //console.log(req.body); //req.body va a ser lo que el usuario coloque en el formlario

}

export {

    guardarTestimonial

}

Y si hay errores, por lo tanto existe el arreglo, eso quiere decir que lo puedes mandar a imprimir a página, también, para que se te guarden los valores, tienes que asignar la etiqueta value (excepto en los textarea):

extends ./layout/index

block contenido

    main*.container.mt-5*

        h1*.text-center.mt-5* #{pagina}

*.row*

*.col-md-12*

                h2*.d-block.w-100.text-center* Agrega un Testimonial

                //- Se muestran mensajes de error si existe ese arreglo

                if(*errores*)

                    //- Iteras sobre el array de mensajes

                    each error in errores

                        //- Cuando iteras, es un array, pero se manda a llamar las variables como objeto

*.alert.col.alert-danger.text-center*=error.mensaje

*.row.justify-content-center.col*

*.col-md-8*

                    //- El metodo POST, es la forma en la que envias datos a un servidor

                    //- En el action tenemos que definir hacia donde queremos enviar los datos

                    form(*action*="/testimoniales" *method*="POST")

                        //- El for sirve para que cuando presiones en el label se active el input, por lo cual el id y el for tienen que ser iguales

                        //- la etiqueta 'name', al momento de imprimir en consola con req, te va a paracer el campo con ese nombre

*.form-group*

                            label(*for*="nombre") Nombre

                            input*#nombre*(*type*="text", *placeholder*="Tu Nombre", *name*="nombre", *class*="form-control", *value*= nombre)

*.form-group*

                            label(*for*="correo") Correo Electronico

                            input*#correo*(*type*="text", *placeholder*="Tu Correo", *name*="correo", *class*="form-control", *value*= correo)

*.form-group*

                            label(*for*="mensaje") Mensaje

                            textarea*#mensaje*(*name*="mensaje", *class*="form-control",*placeholder*="Tu Mensaje",*rows*=3)= mensaje

                        input*.btn.btn-success.btn-block*(*type*="submit")

Almacenar los datos en la base de datos

Para agregar datos, tienes que tener la tabla a la que quieres agregar los datos del formulario, en este caso la creeamos porque no la teníamos, y solamente la tienes que configurar como lo hiciste con los viajes:  
1. Crea un modelo como lo hiciste con viajes:  
import { Sequelize } from "sequelize";

import db from '../config/db.js';

//Le pasamos 2 valores, el primero es el nombre de tu tabla en sql y el segundo es el obj de config

export const Testimonial = db.define('testimoniales', { //En este objetode configuracion vas a tener que definir cada una de las tablas, Ya sea que empiezes desde 0 o ya tengas tablas agregadas.

    //El id no es necesario, ese se da por sentado que existe

    nombre: { //Le tenemos que indicar que tipo de dato va tener y cuantos caracteres va a utilizar

        type: Sequelize.STRING //Sequelize utiliza string en vez de varchart y text

    },

    correo: {

        type: Sequelize.STRING

    },

    mensaje: {

        type: Sequelize.STRING

    }

});

2. Despues creas un testimonial para almacenar los datos:

import { Testimonial } from "../models/Testimoniales.js";

const guardarTestimonial = async (*req*, *res*) =>{

    //Validar

    const {nombre,correo,mensaje} = *req*.body;

    const errores = []

    if(nombre.trim() === ''){

        errores.push({mensaje: 'El nombre esta vacio'})

    }

    if(correo.trim() === ''){

        errores.push({mensaje: 'El correo esta vacio'})

    }

    if(mensaje.trim() === ''){

        errores.push({mensaje: 'El mensaje esta vacio'})

    }

    if(errores.length > 0){

        //Consultar testimoniales existentes

        const testimoniales = await Testimonial.findAll();

        //Mostrar la vista con errores

*res*.render('testimoniales', {

            pagina: 'Testimoniales',

            errores: errores,

            nombre,

            correo,

            mensaje,

            testimoniales

        });

    }else{

        try{

            await Testimonial.create({ //Se crea testimonial

                nombre,

                correo,

                mensaje

            });

*res*.redirect('/testimoniales');

        }catch(error){

            console.log(error);

        }

    }

    //console.log(req.body); //req.body va a ser lo que el usuario coloque en el formlario

}

export {

    guardarTestimonial

}

3. Para mostrar los datos te vas al controlador y le pasas los datos:

import { Testimonial } from '../models/Testimoniales.js';

const paginaTestimoniales = async (*req*,*res*) =>{

    try{

        const testimoniales = await Testimonial.findAll();

*res*.render('testimoniales', {

            pagina: 'Testimoniales',

            testimoniales:testimoniales

        });

    }catch(error){

        console.log(error);

    }

};

Consideraciones del Deployment

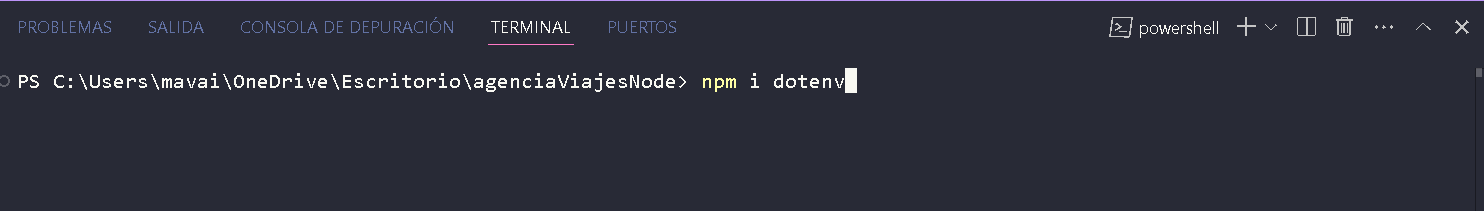
* Los proyectos de Node / Express deberán ser publicados en un servido que soporte esta plataforma
* La base de datos también deberá publicarse en un servidor ya que para que los usuarios puedan acceder a ella, se necesita que su acceso sea publico
* Es necesario realizar algunos ajustes a nuestro proyecto antes de realizar el deployment

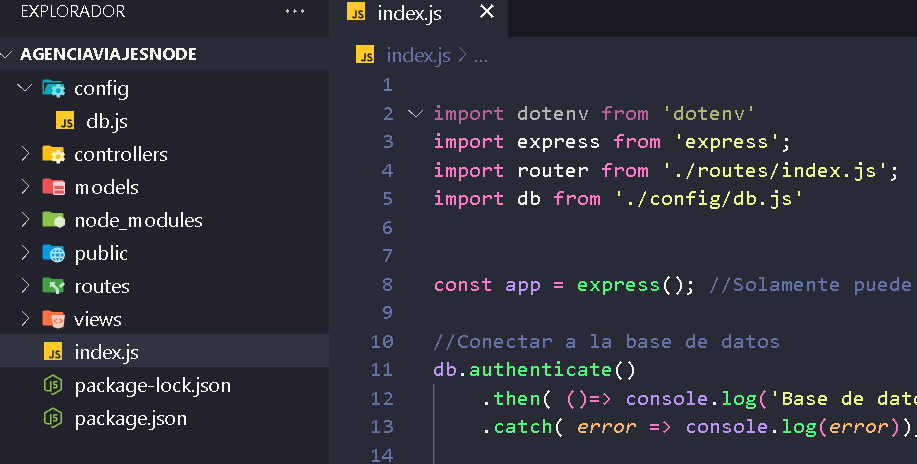
Ajustes:  
1er ajuste:

-Variables de entorno: Las variables de entorno te permiten añadir valores para diferentes entornos – Desarrollo o producción

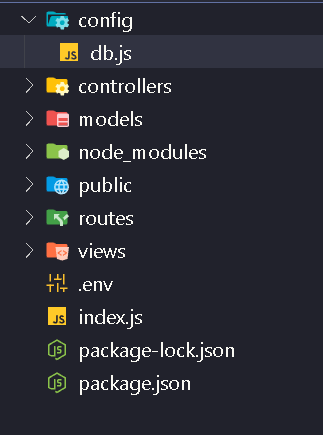
-Además de que las variables de entorno evitan que valores sensibles sean almacenados en Github o que se puedan acceder de forma muy sencilla

Agregar variables de entorno:

Abrimos la terminal y usamos un paquete llamado dotenv  


La importamos en el index principal:  


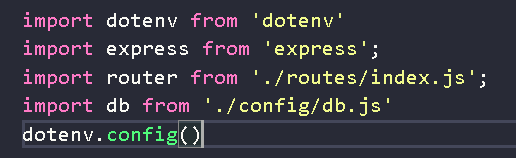
En el caso de express, vas a crear un archivo llamado .env:



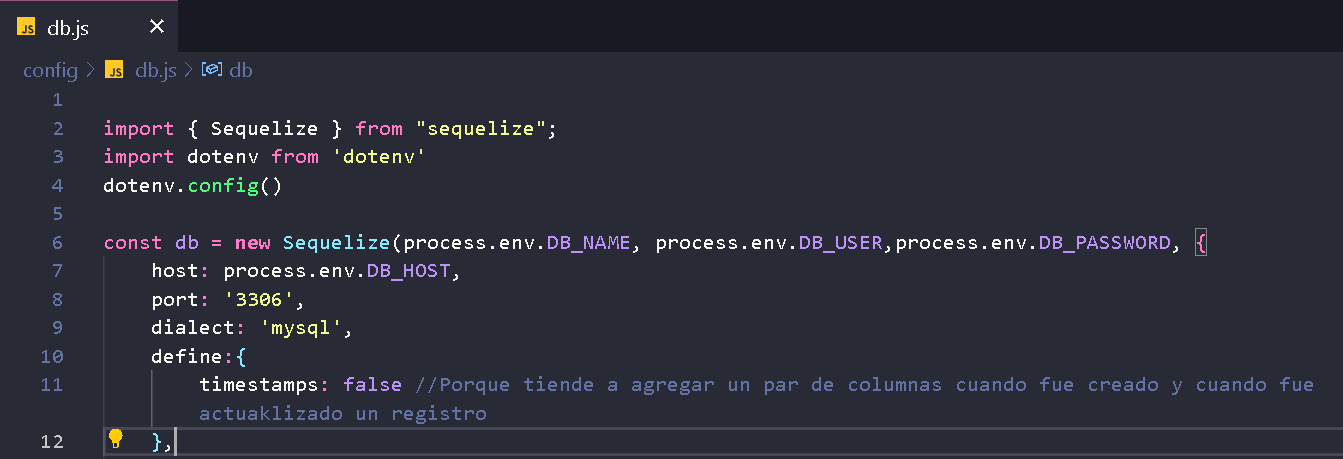
Recomendación para nombrar variables de entorno:  
Se recomienda que estén en mayúsculas.

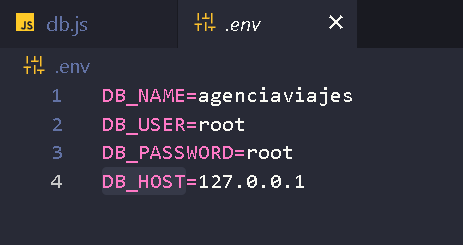
Se recomienda que si tienen espacios le pongas un “\_” en la parte inferior.

Que no tenga números ni nada de eso, sin embargo, no hay problema si finaliza con números como si fuera una variable.

Para tener acceso a los valores de esa variable de entorno se usar ,config:  


Lo anterior de importarlo en el index principal era para un ejemplo, en verdad vas a importarlo donde lo ocupes, que naturalmente es donde esta la configuración de tu base de datos:

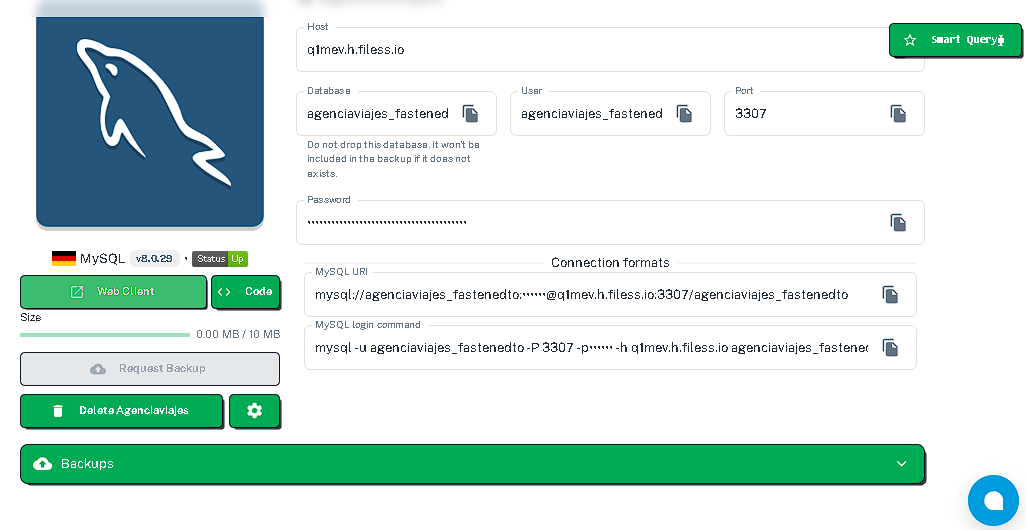




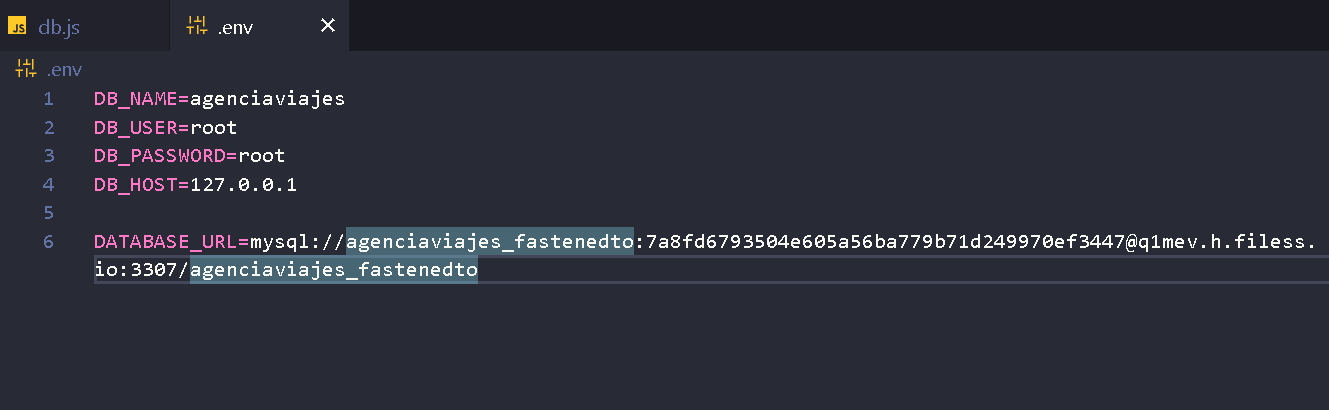
El puerto y el dialecto no es necesario ponerle una variable de entorno.

Como migrar base de datos de local hacia producción:

Creamos una base de datos en filess.io

Una vez creada nos da estos datos:  


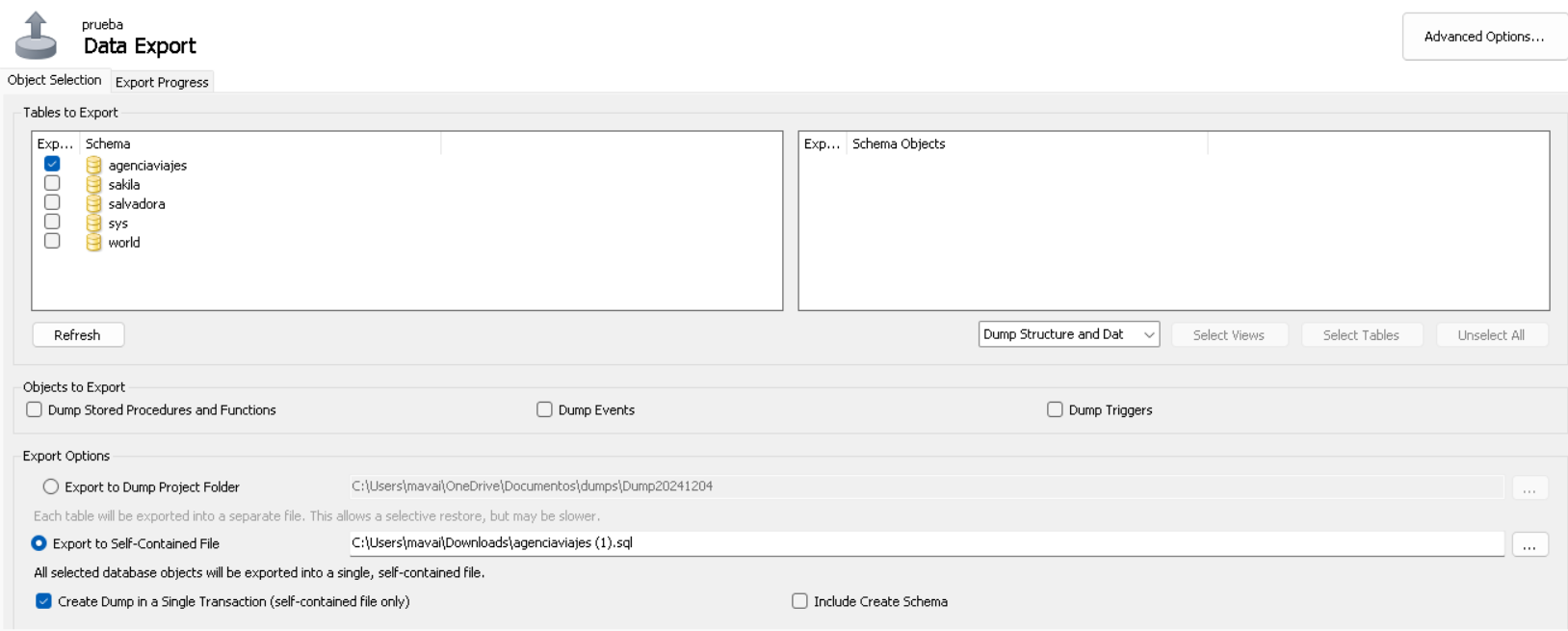
En donde vamos a utilizar MySQL URL para realizar la conexión.

Lo agregamos a env:  


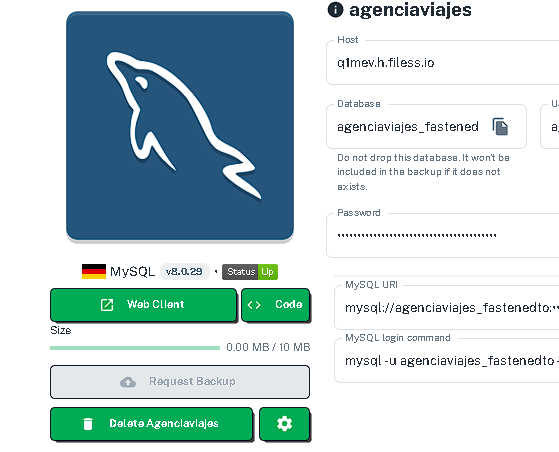
Seguido de eso lo agregamos a la configuración de esta forma:  

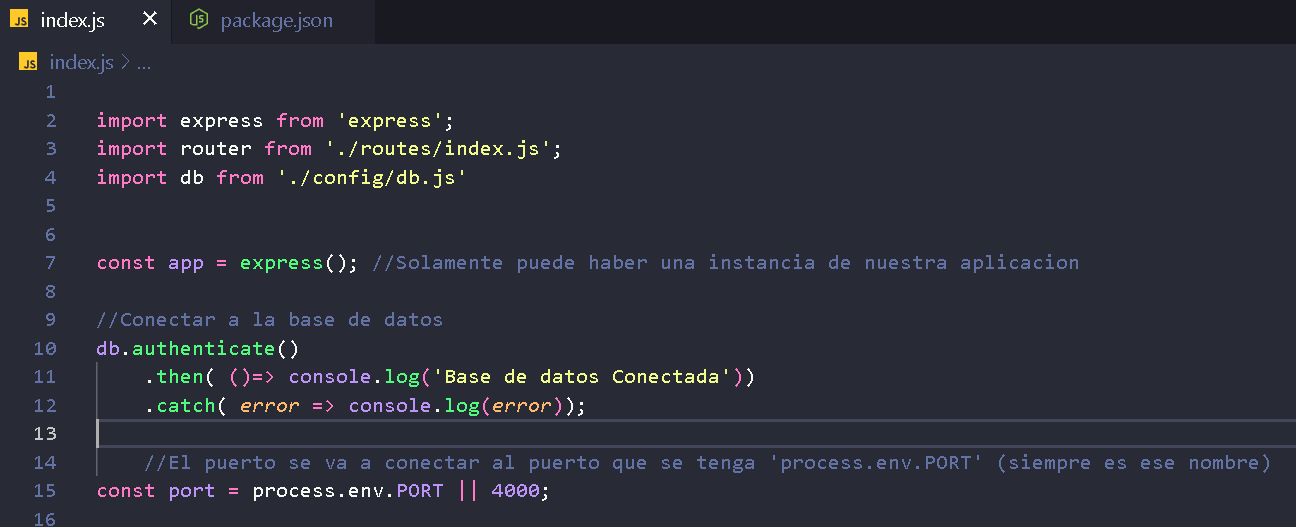

Migrar la información de la base de datos local al servidor:

Exportas tu base de datos:

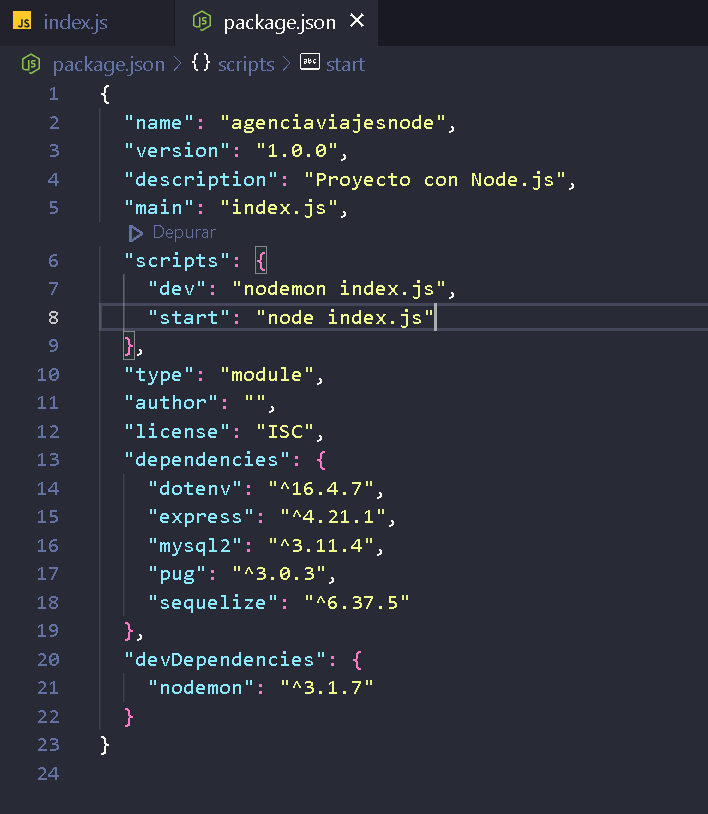


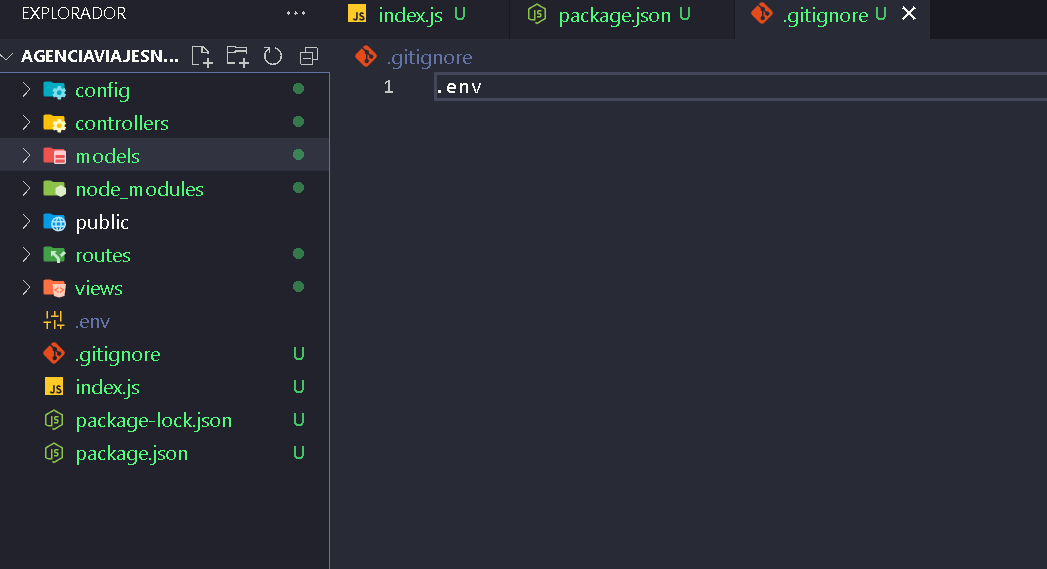
Utilizas “create Dump in a Single Transaction (self-contained file only)” para asi poder crear un solo archive, ya que si no se hace de esa forma se crea un archivo por cada tabla que hay.

Despues le damos a “web client”:  


**Que cambiar cuando vayamos a subir nuestro proyecto:**En nuestro archivo principal, ocupamos definir que se lea el puerto de process.env.PORT:  


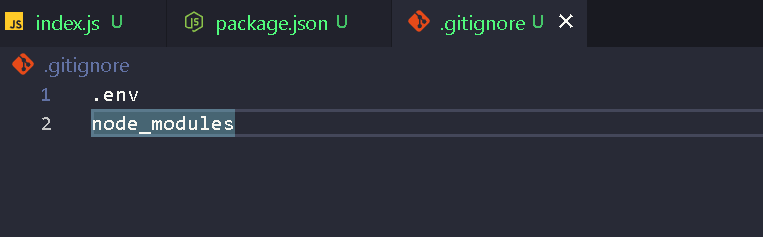
Y en package.json, ocupamos definir que el se ejecute con “node” y que lea nuestro archivo “index.js”:



Agregamos un archivo .gitignore, para que no agregue el archivo .env:  


Esto porque .env contiene las credenciales de nuestra base de datos

Tambien hacemos lo mismo con node modules, porque cuando subimos el proyecto hacia un hosting, este tiene que descargar sus propios node modules y va a descargar las dependencias que sean del proyecto, las que sean de desarrollo no.



**REACT:**

¿Que es?

React o React.js es una librería de JS para la creación de interfaces de usuario desarrollada por Meta.

Utiliza una sintaxis basada en componentes que te permite desarrollar de forma eficiente interfaces de usuario ya sean simples o complejas.

Instalar React:  
Existen diferentes formas de instalar y crear proyectos en React; por muchos años se utilizaba Create-react-app pero una herramienta que ha ganado mucha popularidad en es Vite.

Vite es una herramienta que permite crear un proyecto de React en un par de minutos e instalar todas sus dependencias.

Para instalar React con Vite puedes hacerlo con npm o yarn.

Para instalarlo tienes que escribir en consola lo siguiente:

1. Npm create vite@latest
2. Colocar el nombre del proyecto
3. Seleccionar el framework (react en este caso)
4. Seleccionar la variante (js en este caso)
5. Entrar al proyecto y darle “npm install” para instalar las dependencias
6. Y para arrancar el proyecto le das “npm run dev”

Carpetas:

Node\_modules:

Es donde se guardan las dependencias.

Public:

Es una carpeta donde puedes colocar imágenes, videos, hoja de estilos que quieras utilizar en el proyecto.

Src:

Es donde vas a escribir el código como: funciones, componentes, etc.

Nota:

Hay 2 hojas de estilos:

Apps.css:

Se puede eliminar si así se quiere.

Index.css:

Usualmente aquí vas a tener los estilos globales que aplican a todo tu proyecto, Pero también puedes crear hojas de estilos que afecten a otros componentes

**App.jsx:**

Todo lo que este adentro del return es lo que se va a mostrar en pantalla.

Este archivo, ya es un componente de react.

Los componentes de nombran en mayúsculas, ejemplo:

function App() { //Esto es un componente, se necesita nombrar en mayusculas.

  return (

    <h1>Hola Mundo</h1>

  )

}

export default App

Lo que esta previo al return es código de js, puedes crear funciones, variables, aplicar toda la lógica.

La parte de return es reservada únicamente para los templates o para la parte visual que se va a mostrar en la pantalla, el HTML.

**Main.jsx**

Este es el archivo principal, este es el archivo que se encarga de importar react, de montarlo en el HTML y de agregar todos los componentes, Este es el archivo MAS IMPORTANTE en React.

Este es el código de main.jsx en un principio:

import { StrictMode } from 'react'

import { createRoot } from 'react-dom/client'

import './index.css'

import App from './App.jsx'

createRoot(document.getElementById('root')).render(

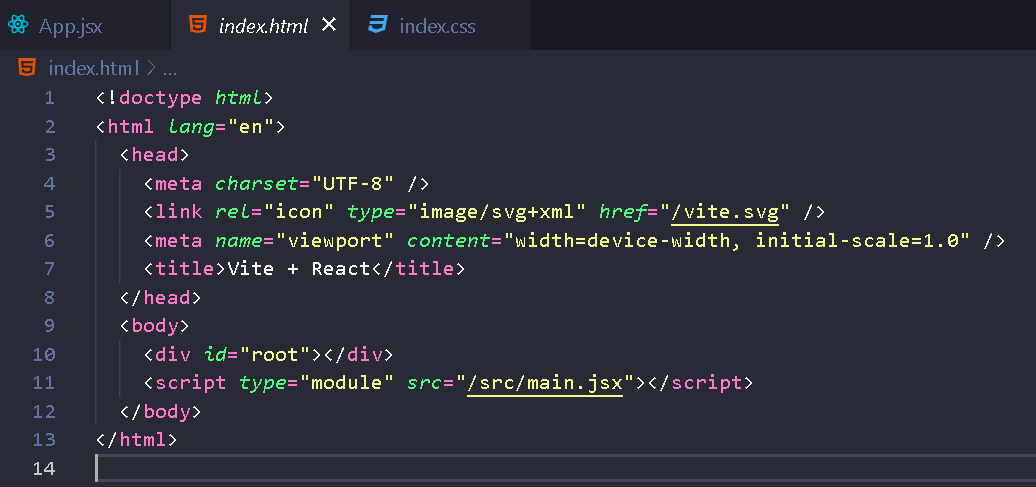
  <*StrictMode*>

    <*App* />

  </*StrictMode*>,

)

En donde getElementById busca un id llamado ‘root’, que se encuentra en index.html:



Y en el div con el id de ‘root’ es donde se inyecta todo el código de react.

**Como instalar Tailwind en React o vite:**

Colocamos este comando (se instalará en dependencias de desarrollo):

npm install -D tailwindcss postcss autoprefixer

Despues colocamos el siguiente comando para que nos cree postcss.config.js y tailwind.config.js:  
npx tailwindcss init -p

En donde para configurar tailwind se accede a tailwind.config.js:

En donde en content se agrega en donde vamos a utilizar tailwind

/\*\* @type *{import('tailwindcss').Config}* \*/

export default {

  content: [

    // Indicamos que index.html va a utilizar tailwind

    "./index.html",

    // Indicamos que entre a todas las carpetas de src de forma recursiva y busque en todas las carpetas que tengan las extenciones que colocamos.

    "./src/\*\*/\*.{js,ts,jsx,tsx}"

  ],

  theme: {

    extend: {},

  },

  plugins: [],

}

Y en el archivo style.css vamos a poner las directivas de este:

@tailwind base;

@tailwind components;

@tailwind utilities;

Regla en react sobre la etiqueta “class”:

function App() { //Esto es un componente, se necesita nombrar en mayusculas.

  return (

    <h1 *className*="text-6xl font-extrabold">Hola Mundo</h1>

  )

}

export default App

En react vas a utilizar tener que utilizar “className” ya que ellos dicen que “class” es una palabra reservada en JS (si lo dejas como class funciona correctamente, pero te va a marcar warnings).

**Que es JSX:**  
Es una sintaxis que permite combinar js y html

Todos los elementos HTML se pueden utilizar, pero no todas las funciones de JS se pueden utilizar

JSX tiene una sintaxis especial para asi poder combinar html y js, para si quieres insertar código de js y que te lo imprima como html, por ejemplo, en este caso si quieres imprimir el contenido de una variable en de JS, lo tienes que encerrar entre ‘{ }’, un ejemplo:

//El App.jsx va a ser componente principal de tu proyecto

function App() { //Esto es un componente, se necesita nombrar en mayusculas.

  const hola = 'Hola mundo';

  return (

    <div *className*="my-20 max-w-lg mx-auto bg-white shadow p-10">

      {hola}

    </div>

  )

}

export default App

Los componentes de react tienen 2 partes:

1. Es la declaración del componente (que es de donde empiezan las llaves hasta el final):
2. { //Esto es un componente, se necesita nombrar en mayusculas.
3. const hola = 'Hola mundo';
4. return (
5. <div *className*="my-20 max-w-lg mx-auto bg-white shadow p-10">
6. {hola}
7. </div>
8. )
9. }

Y otro es el “return”, antes del ‘return’ puedes escribir cualquier código de JS, un for, mas funciones, crear variables. Pero dentro del return ya está un poco limitada esta parte.

Por ejemplo, no puedes crear una variable dentro del return.

Lo que si puedes utilizar adentro de react es un ternario:

//El App.jsx va a ser componente principal de tu proyecto

function App() { //Esto es un componente, se necesita nombrar en mayusculas.

  const hola = 'Hola mundo ';

  const auth = true;

  return (

    <div *className*="my-20 max-w-lg mx-auto bg-white shadow p-10">

      {hola}

      {auth ? 'Autenticado' : 'No autenticado'}

    </div>

  )

}

export default App

**Componentes en React:**

Los componentes te van a permitir dividir tu código en partes reutilizables.

Los componentes utilizan la extensión .js o .jsx y se importan con un import de JavaScript.

Se puede pasar información de un componente a otro por Props.

Naturalmente se crea una carpeta que se denomina “components”, en donde adentro van a ir los componentes de react.

El archivo componente se crea con mayúsculas y con la extensión. jsx

**Un componente es una función**

Para exportar se necesita seguir la siguiente sintaxis:

function Header(){

    return(

        <h1 *className*="text-4xl font-extrabold text-gray-500 text-center">

        Cuanto <span *className*="text-indigo-600">dinero</span> necesitas?

        </h1>

    )

}

export default Header;

Después importas el componente al sitio en donde lo vas a utilizar:

import Header from './components/Header'

function App() {

  return (

    <div *className*="my-20 max-w-lg mx-auto bg-white shadow p-10">

      <*Header* /> //No se necesita etiqueta de apertura o cierre

    </div>

  )

}

export default App

**Reglas de react conforme a lo las etiquetas:**  
Si una etiqueta HTML solo es de apertura como los son los inputs o las imágenes, tienes que añadirle una etiqueta de cierre al final.

**State en React:**  
El state es la fuente de la verdad (source of truth) de tu app.

Un listado de clientes, un carrito de compras lleno o vacio, un usuario autenticado es lo que determina cual es el estado o state de tu app.

Para definir el state en React, se importa useState y la función al momento de utilizarse nos retorna 2 valores: state y la función que modifica el state.

**Administrar State:**

useState es suficiente para crear aplicaciones en react.

El state no debe modificarse directamente; para eso se debe utiliza la función que lo modifica.

En un equipo de trabajo o un proyecto grande se recomienda administrar el state con un state global como Context, Zustand, Recoil o Redux.

**Hooks:**

Existen algunas reglas de los hooks, las dos mas importantes es que tienes que definir el uso de tus hooks en la parte superior (no puedes colocar el hook dentro del return).

Tampoco lo puedes colocar de forma condicional, no puedes colocar la definición de un hook dentro de un if.

Ejemplo:

import {useState} from 'react'

import Header from './components/Header'

function App() { //Esto es un componente, se necesita nombrar en mayusculas.

  //Para el state, le puedes colocar el nombre que desees.

  //El segundo valor va a ser la funcion que va a modificar el state (Como recomendacion el nombre de la funcion que lo va a modificar tiene que ser el mismo que el nombre del state)

  //Dentro de los parentesis se coloca el valor incial

  const [cantidad, setCantidad] = useState(10000); //destructuring de arreglos

  console.log(cantidad);

  return (

    <div *className*="my-20 max-w-lg mx-auto bg-white shadow p-10">

      <*Header* />

      <input

*type*='range'

*className*='w-full h-6 bg-gray-200 accent-lime-500 hover:accent-lime-600'

      />

    </div>

  )

}

export default App

Si quieres modificar el valor del state tendrías que llamar a la función, no lo puedes modificar asignándole otro valor a la variable.

**Eventos en react:**

Los eventos en React inician con la palabra “on”, seguido del evento en mayúscula; onClick, onSubmit, onChange, estos solamente son algunos ejemplos.

Estos eventos se agregan vía atributo en el HTML.

Aquí hay un ejemplo de como se tiene que modificar el state y como llamar a un evento en react:

import {useState} from 'react'

import Header from './components/Header'

function App() {

  const [cantidad, setCantidad] = useState(10000); //destructuring de arreglos

  function handleChange(*e*){

    setCantidad(+*e*.target.value)

  }

  return (

    <div *className*="my-20 max-w-lg mx-auto bg-white shadow p-10">

      <*Header* />

      <input

*type*='range'

*className*='w-full h-6 bg-gray-200 accent-lime-500 hover:accent-lime-600'

*onChange*={handleChange}

      />

      {cantidad}

    </div>

  )

}

export default App

**UseEffect:**

useEffect es un hook que se ejecuta una vez que el componente esta listo, y toma un array de dependencias.

En su listado de dependencias se le puede pasar un state para “escuchar” los cambios que ocurren en ese state; en caso de que el state se actualice, useEffecct se va a ejecutar nuevamente.

La sintaxis es la siguiente:

useEffect(() =>{

  },[])

* Siempre va a tener un callback adentro
* Después le agregas una coma y le puedes agregar un arreglo de dependencia, entonces siempre va a ser un arreglo (si se lo pasas vacío solamente se ejecuta una vez).
* Es mejor tener useEffect’s que escuchen por una dependencia o 2 máximo, porque si no se va a estar ejecutando todo el tiempo.

**Vue.js**

Es un framework de Js para la creación de interfaces de usuario, para sitios y aplicaciones visual (es decir únicamente la parte visual, no consulta una base de datos, tampoco agrega seguridad, ni agrega clases como tailwind o Bootstrap, únicamente se encarga de toda la parte de js para una interfaz de usuario.

Construido con y sobre HTML, CSS y JS; tiene una sintaxis basada en componentes que te permite desarrollar de forma eficiente interfaces de usuario ya sean simples o complejas.

**Instalar Vue.js**

Existen diferentes formas de instalar Vue.js; por muchos años se utilizaba VueCLI pero esta en desarrollo; otras opciones son create-vue y vite.

Vite es una herramienta que permite crear un proyecto de Vue en un par de minutos e instalar todas sus dependencias.

Para instalar Vue con vite puede hacerlo con NPM o YARN.

**Instalacion**:

1. npm create vite@latest
2. Nombre del proyecto
3. Vue
4. Javascrip (o como lo quieras desarrollar)
5. Entra al proyecto
6. Npm install
7. Npm run dev

Archivos:  
Vamos a encontrar una estructura muy similar a lo que encontraríamos en react.

También encontraras una carpeta src en donde tendrás la mayoría de tu codigol.

Vue también cuenta con un archivo App.vue.

Tenemos un achivo Main.js que lo que hace es montar nuestra aplicación. Aquí esta:

import { createApp } from 'vue'

import './style.css'

import App from './App.vue' //Importa App que es nuestro archivo App.vue

createApp(App).mount('#app') //Crea la aplicacion con el App que importamos anteriormente y lo monta en un div con el id de "app" (que este id se encuentra en el index.html)

Single File Components:

Vue.js tiene su propio estilo para crear proyectos y estructurarlos.

Single file components (SFC) es una convención en la cual cada componente tiene 3 partes:

<script><style> y <template> ( no es obligatorio tenerlos los 3).

En <script> se coloca toda la lógica de JS de ese componente (ahí vas a importar otros componentes, vas a declarar tu state y también algunas funciones).

En <style> se coloca todo el código CSS de ese componente.

En <template> se coloca todo el código HTML de ese componente.

Lo que coloques en ‘template’ se va a mostrar en pantalla:

<template>

  <div>

    <h1>Hola Mundo</h1>

  </div>

</template>

Lo que coloque en ‘script’ lo voy a mostrar en consola:

<script setup>

</script>

Lo que coloques en ‘style’ le va a dar estilos a lo que tengas en ‘template’.

Lo que hace es ‘scoped’ es que el style se lo va a aplicar únicamente al template anterior, sino se lo va a aplicar a componentes diferentes.

<style scoped>

*.logo* {

  height: 6em;

  padding: 1.5em;

  will-change: filter;

  transition: filter 300ms;

}

*.logo:hover* {

  filter: drop-shadow(0 0 2em #646cffaa);

}

*.logo.vue:hover* {

  filter: drop-shadow(0 0 2em #42b883aa);

}

</style>

**Componentes en Vue:**

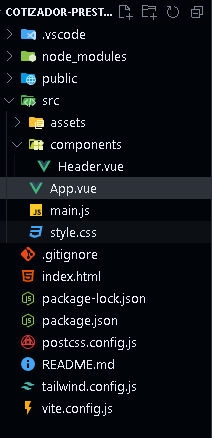
Los componentes te van a permitir dividir tu código en partes reutilizables

Los componentes utilizan la extensión. vue y se importan con un import de JS.

Con <script setup> cualquier componente que es importado se hacen disponibles en el template.

Se puede pasar información de un componente a otro por Props.

Para crear un componente se necesita crear un archivo en la carpeta de ‘components’ (en este caso llamado Header):



En este archivo agregamos el componente adentro del template:

<template>

    <h1 class="text-4xl font-extrabold text-gray-500 text-center">

        Cuanto <span class="text-indigo-500">dinero </span> necesitas?

      </h1>

</template>

Y lo importamos a nuestro archivo que necesitemos (‘setup’ hace disponible ese componente una vez que lo estamos importando ya en el template)

<script setup>

  import Header from './components/Header.vue';

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

  </div>

</template>

**API Styles en Vue.js - Options API y Composition API**

Los componentes de Vue se pueden escribir en 2 API’s diferentes:

Options API y Composition API

Options API tu aplicación utiliza una sintaxis de tipo Objeto.

Con composition API se definen los componentes utilizando los imports y escribiendo directamente las funciones directamente en el Componente.

Con composition (sabemos que utilizamos composition porque utilizamos ‘setup’):

<script setup>

  import Header from './components/Header.vue';

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

  </div>

</template>

Con options (sabemos que es con ‘options’ porque declaras tu componente como un objeto:

<script>

  import Header from './components/Header.vue';

  export default {

    components:{

      Header

    }

  }

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

  </div>

</template>

**Cuando utilizar Options API y Composition:**

Options API es un poco mas amigable para personas que van comenzando, también es la opción recomendada para personas que tienen mas experiencia con lenguajes orientados a objetos.

Options API es recomendado si tu proyecto utilizara pequeñas piezas de Vue o escenarios no tan complejos.

Composition API es recomendado si todo tu proyecto será hecho con Vue.js.

**Imprimir una variable en Vue:**

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          @input="handleChange"

        />

        {{cantidad}} //Para imprimr una cantidad utilizas dobles corchetes

    </div>

  </div>

</template>

**Eventos en Vue:**

Todas las aplicaciones requieren eventos: cuando un usuario escribe en un campo, realiza un submit a un formulario, click en un elemento.

Los eventos en Vue.js se agregan con la directiva: “v-on:evento” aunque usualmente se recomienda se la sintaxis corta de @evento.

Ejemplo de evento:

<script setup>

  import Header from './components/Header.vue';

  function handleChange(e) {

    console.log(e.target.value);

  }

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          v-on:input="handleChange"//Colocas el nombre de la funcion

        /> //v-on es una directiva, que una directiva basicamente es una funcion especial de vue

    </div>

  </div>

</template>

Colocar esa función especial de vue es lo mismo que poner en js:

//const formulario = document.querySelect...

//formulario.addEventListener('input');

Otra sintaxis que se recomienda para agregar eventos es (ya que es mas corta y para eventos es la recomendada):

<script setup>

  import Header from './components/Header.vue';

  function handleChange(e) {

    console.log(e.target.value);

  }

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          @input="handleChange"

*<*/div>

  </div>

</template>

**State en Vue.js:**

El state es la fuente de la verdad (source of truth) de tu app.

Un listado de clientes, un carrito de compras lleno o vacio, un usuario autenticado es lo que determina cual es el estado o state de tu app.

Vue.js maneja y actualiza este state en base a ciertas funciones o condiciones en tu código.

**Administrar State:**

Las 2 formas mas sencillas de manejar un state en Vue son con ref y reactive, ambos se importan desde vue.

Ref toma valores primitivos. Mientras que reactive toma objetos.

En un equipo de trabajo o un proyecto grande se recomienda administrar el state con una herramienta llamada ‘Pinia’ (sustituto de VueX).

Ejemplo de reactive y ref:

const cantidad = ref(0); //Toma valores primitivos

const state = reactive({

    canitdad: 0

  }); //siempre le tienes que pasar un obj

  function handleChange(e) {

    console.log(e.target.value);

  }

  console.log(cantidad.value); //Para que 'ref' te retorne un valor, vas a tener que colocar '.value despues de la variable'

  console.log(state.canitdad);// En este caso muestra directamente la cantidad.

**Modificar el valor de ref:**

Como mencionamos anteriormente, para poder mostrar el valor de ref, es necesario colocar ‘.value’ por lo que para cambiar el valor, asignamos el valor que queramos pasar a ‘cantidad.value’.

function handleChange(e) {

    cantidad.value = +e.target.value;

  }

Y para mostrarlo, lo podemos mostrar en el div, con dobles corchetes que es como se muestra una variable en Vue:

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          @input="handleChange"

        />

        {{cantidad}}

    </div>

  </div>

</template>

**Modificar** **un valor con reactive:**Es parecido a como modificamos el ‘ref’ solo que aquí como ‘state’ es mas parecido a un objeto, entonces asignamos el valor como si fuera uno, en este caso con ‘state.cantidad’ :

function handleChange(e) {

    state.cantidad = +e.target.value;

  }

Y lo mandamos a llamar de la misma manera:

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          @input="handleChange"

        />

        {{state.cantidad}}

    </div>

  </div>

</template>

Mostrar el State en Pantalla:

Lo puede hacer de la manera tradicional:

<p>{{cantidad}}</p>

O lo puedes hacer con una directiva:

<p v-text="cantidad"></p>

Para colocar texto antes o después de la variable:

<p>$ {{cantidad}}</p>

<p v-text="`$ ${cantidad}`"></p>

**Atributos Dinamicos en Vue.js:**

Para que el input busque valores dinámicos y los detecte, se necesita colocar ‘:’ al principio de las etiquetas

<script setup>

  import Header from './components/Header.vue';

  import {ref} from 'vue';

  const cantidad = ref(10000); //Toma valores primitivos

  const MIN =0;

  const MAX =20000;

  const STEP =100;

  function handleChange(e) {

    cantidad.value = +e.target.value;

  }

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          :min="MIN"

          :max="MAX"

          :step="STEP"

          value="10000"

          @input="handleChange"

        />

        <p>$ {{cantidad}}</p>

    </div>

  </div>

</template>

**La Directiva v-model:**

Con v-model, podemos pasar de esto:

const cantidad = ref(10000); //Toma valores primitivos

  const MIN =0;

  const MAX =20000;

  const STEP =100;

  function handleChange(e) {

    cantidad.value = +e.target.value;

  }

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          :min="MIN"

          :max="MAX"

          :step="STEP"

          :value="cantidad"

          @input="handleChange"

        />

        <p>$ {{cantidad}}</p>

    </div>

  </div>

</template>

A esto:

  const cantidad = ref(10000); //Toma valores primitivos

  const MIN =0;

  const MAX =20000;

  const STEP =100;

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          :min="MIN"

          :max="MAX"

          :step="STEP"

          v-model="cantidad"

        />

        <p>$ {{cantidad}}</p>

    </div>

  </div>

</template>

v-model lo que tiene es que reemplaza el ‘@input’ y el ‘value’ y hace la función de los 2, por lo que ya no es necesario llamar a una función para indicar lo que se va a modificar el valor. Tambien tiene modificadores, como el siguiente que te convierte de un string a un número:

v-model.number="cantidad"

**Compute properties:**

Para tener un codigo mas ordenado y seguir teniendo actualizaciones en tu codigo; Vue.js recomienda el uso de Computed Properties.

Una Computed Property es una function; que esta al pendiente de los cambios de tu state y realiza los cambios necesarios cuando este cambia.

Los computed properties no pueden tomar argumentos.

Ejemplo de compute:

const formatearDinero = computed( () => {

    const formatter = new Intl.NumberFormat('en-US',{

      style:'currency',

      currency:'USD'

    });

    return formatter.format(cantidad.value)

  }); //Utilizamos compute cuando se va a mostrar algo en pantalla

<template>

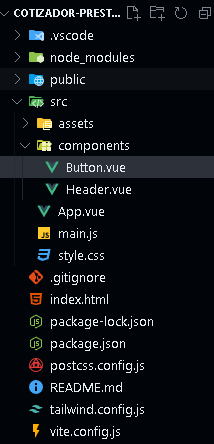
    <div class="my-5">

        <p class="text-center my-10 text-5xl font-extrabold text-indigo-600">{{formatearDinero}}</p>

    </div>

</template>

Forma de pasar crear un componente y pasar props:  
Crear en la carpeta ‘components’ y crear un componente (en este caso llamado ‘Button.vue’):



Despues crear la logica que llevara ese componente:

<script setup>

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    >-</button>

</template>

Y despues lo puedes mandar a llamar al template importandolo:

<script setup>

  import {ref, computed} from 'vue';

  import Button from './components/Button.vue';

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="flex justify-between mt-10 ">

      <Button

        :operador="'-'"

      />

      <Button

        :operador="'+'"

      />

  </div>

</template>

Hay 2 formas de pasar proprs:  
El primer paso es que la sintaxis para mandar un prop es la siguiente:

<script setup>

  import {ref, computed} from 'vue';

  import Button from './components/Button.vue';

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="flex justify-between mt-10 ">

      <Button

        :operador="'-'" //De esta forma mandas el prop

      />

      <Button

        :operador="'+'"

      />

  </div>

</template>

Y lo recibes aca:

<script setup>

    const props = defineProps(['operador']); // Colocas el nombre del prop

    console.log(props); //Lo imprimes a consola

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    >-</button>

</template>

La Segunda forma es como objeto:

<script setup>

    const props = defineProps({

        operador: String

    })

    console.log(props);

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    >-</button>

</template>

Para mostrarlo simplemente utilizas llaves dobles:

<script setup>

    const props = defineProps({

        operador: String

    })

    console.log(props);

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    >{{operador}}</button>

</template>

Para pasar una function se hace lo siguiente:

<script setup>

  import {ref, computed} from 'vue';

  import Header from './components/Header.vue';

  import Button from './components/Button.vue';

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="flex justify-between mt-10 ">

      <Button

        :operador="'-'"

        :fn="handleChangeDecremento" //creas un prop y le asignas la funcion

      />

      <Button

        :operador="'+'"

        :fn="handleChangeIncremento"

      />

    </div>

  </div>

</template>

Despues te vas al componente y creas el evento y le asignas el prop:

<script setup>

    const props = defineProps({

        operador: String,

        fn:Function

    })

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    @click="fn"

    >{{operador}}</button>

</template>

**Eventos personalizados:**

Para crear un evento personalizado se necesita colocar la ‘@’ al principio de este.

<script setup>

  import {ref, computed} from 'vue';

  import Button from './components/Button.vue';

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="flex justify-between mt-10 ">

      <Button

        :operador="'-'"

        @fn="handleChangeDecremento"

      />

      <Button

        :operador="'+'"

        @fn="handleChangeIncremento"

      />

    </div>

    <div class="

  </div>

</template>

Y despues en el componente se lo pasas:

<script setup>

    const props = defineProps({

        operador: String

    })

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    @click="$emit('fn')"

    >{{operador}}</button>

</template>

Si tambien pasar algun tipo de dato al momento de presionar o dar click, entonces lo puedes hacer colocando el valor en cuestion a un lado, por ejemplo:

<script setup>

    const props = defineProps({

        operador: String

    });

    const hola = "hola";

</script>

<template>

    <button

    class="h-10 w-10 flex items-center justify-center font-bold bg-lime-500 rounded-full hover:outline-none hover:ring-2 hover:ring-offset-2 hover:ring-lime-500 text-white text-2xl"

    @click="$emit('fn',hola)"

    >{{operador}}</button>

</template>

Esto lo que va a hacer es enviar el valor de un componente hijo a un componente padre, en este caso, la variable ‘hola’ se va a enviar a la function y va a imprimir ‘hola’ en consola:

<script setup>

  import {ref, computed} from 'vue';

  import Button from './components/Button.vue';

  const handleChangeDecremento = (texto) =>{

    const valor = cantidad.value - STEP;

    if(valor < MIN){

      alert('Canitidad no valida');

      return;

    }

    cantidad.value = valor;

    console.log(texto)

  }

  const handleChangeIncremento = (texto) =>{

    const valor = cantidad.value + STEP;

    if( valor > MAX){

      alert('Canitidad no valida');

      return;

    }

    cantidad.value = valor;

    console.log(texto)

  }

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <Header />

    <div class="flex justify-between mt-10 ">

      <Button

        :operador="'-'"

        @fn="handleChangeDecremento"

      />

      <Button

        :operador="'+'"

        @fn="handleChangeIncremento"

      />

    </div>

  </div>

</template>

**Watchers en Vue.js:**

Con los computedProperties podemos revisar los valores derivados de una función; pero algunas veces queremos modificar algo extra cuando algún state cambia, con composition API puedes utilizar la función watch para lograrlo.

Un watch tomara una o multiples dependencias; y cuando estas cambien entonces podremos llamar otra función o realizar una validación.

Ejemplo:

<script setup>

  import {ref, computed, watch} from 'vue';

  import Button from './components/Button.vue';

  import {calcularTotalPagar} from './helpers'

  const cantidad = ref(10000); //Toma valores primitivos

  const meses = ref(6);

  const total = ref(calcularTotalPagar(cantidad.value, meses.value));

  watch([cantidad, meses], () =>{

    total.value = calcularTotalPagar(cantidad.value, meses.value)

  })

</script>

<template>

  <div class="my-20 max-w-lg mx-auto bg-white shadow p-10">

    <div class="my-5">

        <input

          type="range"

          class="w-full bg-gray-200 accent-lime-500 hover:accent-lime-600"

          :min="MIN"

          :max="MAX"

          :step="STEP"

          v-model.number="cantidad"

        />

        <p class="text-center my-10 text-5xl font-extrabold text-indigo-600">{{formatearDinero(cantidad)}}</p>

        <h2 class="text-2xl font-extrabold text-gray-500 text-center">Elige un <span class="text-indigo-600">Plazo</span> a pagar</h2>

        <select name="" id="" class="w-full p-2 bg-white border border-gray-300 rounded-lg text-center text-xl font-bold text-gray-500 mt-5"

        :value="meses"

        v-model.number="meses" >

          <option value="6">6 Meses</option>

          <option value="12">12 Meses</option>

          <option value="24">24 Meses</option>

        </select>

    </div>

    <div class="my-5 space-y-3 bg-gray-50 p-5">

      <h2 class="text-2xl font-extrabold text-gray-500 text-center">

        Resumen <span class="text-indigo-600">de pagos</span>

      </h2>

      <p class="text-xl text-gray-500 text-center font-bold">{{meses}} Meses </p>

      <p class="text-xl text-gray-500 text-center font-bold">Total a pagar: {{formatearDinero(total)}}</p>

      <p class="text-xl text-gray-500 text-center font-bold">Mensuales </p>

    </div>

  </div>

</template>

Si quieres que lo actualice en cuanto cargue el objeto, entonces tienes que agregarle lo siguiente al watch:

watch([cantidad, meses], () =>{

    total.value = calcularTotalPagar(cantidad.value, meses.value);

  }, {immediate:true} )

**¿Qué es typescript?**

Es un lenguaje de programación fuertemente tipado que se convierte a JS.

Es un lenguaje fuertemente tipado.

“Fuertemente tipado” es un termino utilizado en el contexto de la programación para describir un lenguaje que aplica reglas estrictas sobre el manejo de tipos de datos.

En JS una varibla puede contener cualquier tipo de dato e incluso reasignarse, typescript previene ese tipo de comportamiento.

**Ventajas:**

Su principal ventaja es encontrar errores en el editor.

Su segunda ventaja mas importante es la velocidad de escribir código, en especial si se utiliza con VSCode.

Si todo en tu código de Typescript esta bien en el editor puedes “compilar” ese código hacia JS y tener la garantía de que ese código va a funcionar correctamente.

**Mitos de Typescript:**

Corrige código de JS.

No deberías aprender JS.

Un mal código de JS se soluciona migrando a Typescript.

**Inferir en Typescript:**

// Selectores

export const pacienteInput = document.querySelector('#paciente')

export const propietarioInput = document.querySelector('#propietario')

export const emailInput = document.querySelector('#email')

export const fechaInput = document.querySelector('#fecha')

export const sintomasInput = document.querySelector('#sintomas')

export const formulario = document.querySelector('#formulario-cita')

export const formularioInput = document.querySelector('#formulario-cita input[type="submit"]')

export const contenedorCitas = document.querySelector('#citas')

Al tu no señalar que tipo de dato es en typescript, lo que hace typescript es inferir o deducir que tipo de dato es, por ejemplo al tu hacer esto de colocar selectores, lo que va a inferir typescript es que es un tipo de dato ‘elemento’ o que es ‘null’ es decir que si existe es un tipo de dato ‘elemento’, pero si no existe por si acaso, seria ‘null’.

Pero esto al intentar agregarle un evento podría marcarte un error, ya que los eventos no pueden ser nulos:

// Eventos

pacienteInput.addEventListener('change', datosCita)

propietarioInput.addEventListener('change', datosCita)

emailInput.addEventListener('change', datosCita)

fechaInput.addEventListener('change', datosCita)

sintomasInput.addEventListener('change', datosCita)

formulario.addEventListener('submit', submitCita)

Esto se soluciona de una manera muy fácil, con el ‘optional chaning’:

// Eventos

pacienteInput?.addEventListener('change', datosCita)

propietarioInput?.addEventListener('change', datosCita)

emailInput?.addEventListener('change', datosCita)

fechaInput?.addEventListener('change', datosCita)

sintomasInput?.addEventListener('change', datosCita)

formulario?.addEventListener('submit', submitCita)

Que básicamente lo que dice es que: en caso de que este elemento exista regístrale este evento.

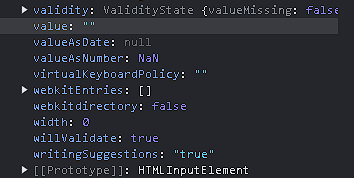
Otra forma de hacerlo es que en lugar de que Typescript infiera que tipo de dato es, nosotros indicarle que tipo de dato es.

Para saber que tipo de dato es y colocarlo, entonces puedes colocar en consola lo siguiente:  
console.dir(el selector)

Ejemplo:

console.dir(document.querySelector('#paciente') )

Le das enter y hasta abajo te viene que tipo de elemento es.



Entonces para definirlo como un tipo de elemento en especial se coloca:

const pacienteInput = document.querySelector('#paciente') as HTMLInputElement

O tambien se puede utilizar la sintaxis de generics:

const pacienteInput = document.querySelector<HTMLInputElement>('#paciente')

Pero con esto va a ser necesario agregar el optional chaning al momento de utilizar este selector.

Porque básicamente con generics lo que le estas indicando es que ‘pacienteInput’ tiene que tener este tipo de dato.

Esta es una forma en la cual puedes reutilizar el mismo código y aplicarlo a diferente tipos de datos.

En este caso este es un ejemplo en el que estamos utilizando querySelector, y en lugar de que solo infiera que es un element, indicarle que es un HTMLInputElement.

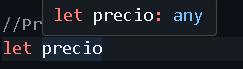
**Primitive Types en TypeScript**

Estos son los tipos de datos que soporta TypeScript de forma nativa.

Estos son number, string, boolean, null y undefined.

En el caso de los arreglos hay una sintaxis especial para crearlos.

TypeScrip si tu no le asignas el tipo de dato, este por defecto le va a asignar “any” (lo cual está mal que quede definido de esta manera).



Se lo puedes definir colocando la sintaxis de “:” :

let precio:number

Si quieres que acepte otro tipo de dato podrías hacer lo siguiente (puedes añadir todos los que desees, pero es recomendado evitar esta práctica):

let precio:number | boolean

Ya bien definidos quedaría de la siguiente manera:

//Primitive types

let precio:number

let producto : string

let disponible : boolean

precio = 50;

producto = "Monitos 34 pulgadas"

disponible = false;

**Types e Interfaces en TypeScript:**

2 de las características que mas vas a utilizar en TypeScript son Types e Interfaces.

Ambas pueden ser utilizadas incuso de forma intercambiable, hay muy pocas diferencias entre ambos.

Definir “interface” para un objeto:

//Definir Interface para asignarle los tipos de datos a mi objeto

interface Producto {

    precio: number;

    producto: string;

    disponible: boolean;

}

const producto : Producto = {

    precio: 50,

    producto: "Monitos 34 pulgadas",

    disponible: false

}

Definir “type” para un objeto:

//Definir Type para asignarle los tipos de datos a mi objeto

type Producto = {

    precio: number;

    producto: string;

    disponible: boolean;

}

const producto : Producto = {

    precio: 50,

    producto: "Monitos 34 pulgadas",

    disponible: false

}

Para definir el tipo de valor en una clase se hace lo siguiente:

type NotificacionType = {

    texto : string,

    tipo:'error' | 'exito' | ''

}

export default class Notificacion {

    notificacion: NotificacionType = {

        texto: '',

        tipo:''

    }

    constructor({texto, tipo} : NotificacionType) {

        this.notificacion.texto = texto

        this.notificacion.tipo = tipo

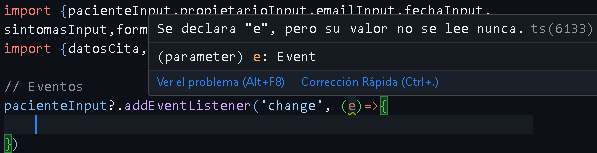
        this.mostrar()

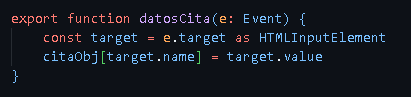
    }

}

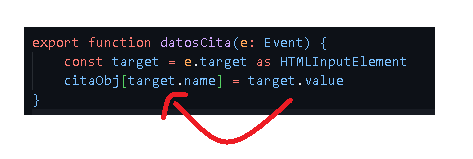
**Eventos en TypeScript**

Como identificar los tipos de eventos en typescript:

Puedes hacer que typescript lo infiera, en este caso te lo da por default (en este caso lo marca como que es un evento):  


Tambien es necesario definirle el inteface que va a tener el ‘e.target’, por lo que en este caso que el e.target va a ser el input, se lo definimos de la siguiente manera:  


Tambien, si vas a agregar un tipo de dato dinámicamente es necesario que le indiques con un ‘index signature’ que tipo de dato es, de la siguiente forma:



Como ahí estas agregando ‘target.value’ a citasObj, tienes que indicar que tipo de dato le estas introduciendo, en este caso es un string:

interface Cita {

    [key: string] : string //index signature

    id: string;

    paciente: string;

    propietario: string;

    email: string;

    fecha: string;

    sintomas: string;

}

// Objeto de Cita

const citaObj : Cita = {

    id: generarId(),

    paciente: '',

    propietario: '',

    email: '',

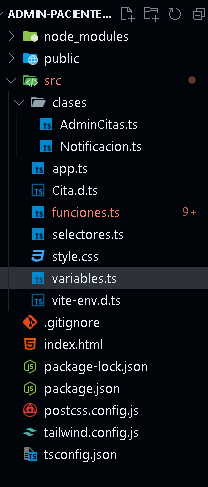
    fecha: '',

    sintomas: ''

}

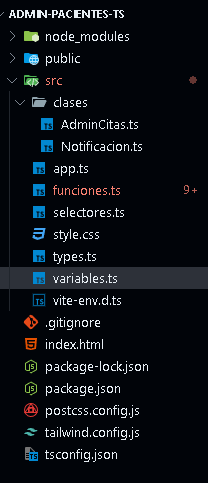
**Creando archivos para los Types:**

Una forma no recomendable es que crees un archivo en donde metas las interfaces. El archivo tiene que tener la extencion ‘.d.ts’:



Lo conveniente de esto es que se importa automáticamente y tu no tienes que ir importando uno por uno, pero el equipo de typescript no lo recomienda.

Otra forma de hacerlo es creando un archivo comúnmente llamado ‘type.ts’ y ahí colocando la sintaxis del type:



Sintaxis:

export interface Cita {

    [key: string] : string //index signature

    id: string;

    paciente: string;

    propietario: string;

    email: string;

    fecha: string;

    sintomas: string;

}

Despues de eso lo tienes que importar de la siguiente manera en los archivos que necesites:

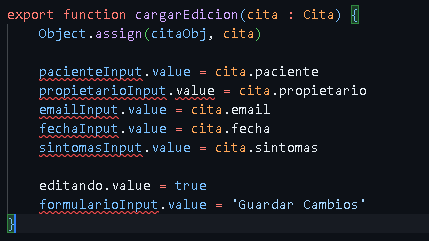
import type {Cita} from './types'

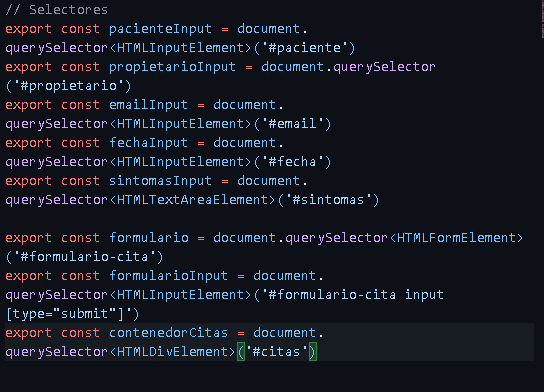
Usas el ‘type’ para aclarar aun mas que lo que estas importando en es un type y no una variable, objeto o función (en si no es necesario, pero es recomendado hacerlo).

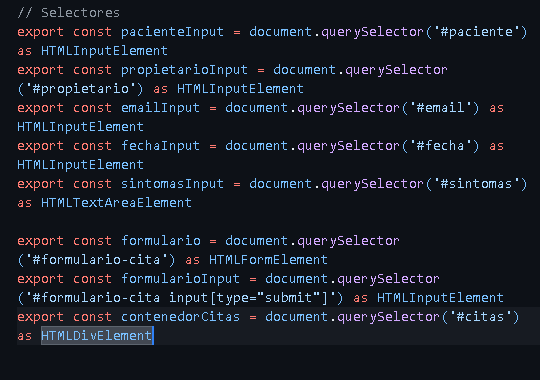
RECOMENDACIÓN:

Si solo lo vas a utilizar en un lugar, crea el type en donde lo vas a necesitar, pero si lo ocupas en diferentes lugares, entonces colócalo en el archivo de types.

Otra recomendación, es que si tienes un montón de errores que es porque typescript te indica de que si un elemento es probable que no exista y no quieres agregar muchos if, como lo seria el siguiente caso:



Es que en lugar de usar generics:  


Utilices ‘as’:  


Que prácticamente lo que le dice a typescript es: confía en mi, ese elemento va a existir.

Tipos de datos que soporta Mongoose:

