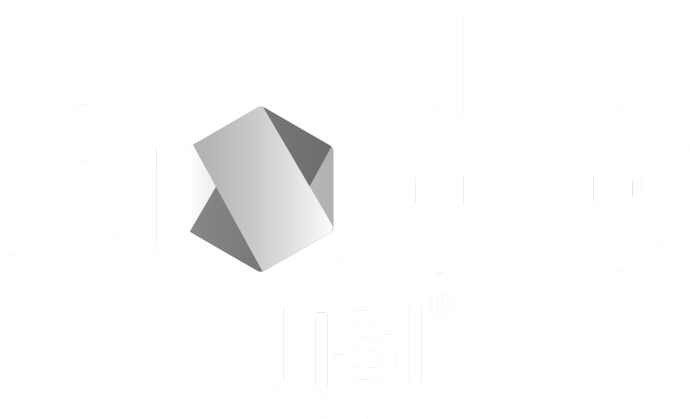


class: center, middle, inverse



name: contenido class: center, middle, inverse

[Node.js](#Qué es Node.js?)

name: default layout: true task:

.task[{{task}}]

name:contenido

###Contenido

- [Introducción](#)
- [Instalación](#)
- [npm](#)
- [módulos](#)
- [callbacks](#)
- [promesas](#)
- [async-await](#)
- [EventEmitter](#)
- [Buffers](#)
- [Streams](#)
- [FileSystem](#)
- [TCP](#)
- [http](#)
- [producción](#)
- [socket.io](#)

name:introduccion task: << [índice de contenidos](#) >>

Qué es Node.js?

Node.js (o abreviadamente *Node*) es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome desarrollado por Ryan Dahl, cuando estaba en Joyent. Dahl creó Node debido a las [limitaciones síncronas de los frameworks web actuales](#). Node nos permite correr JavaScript en cualquier sitio, incluidos servidores web y robots

En el pasado, el único entorno de ejecución para JavaScript era el navegador. Chrome, Safari y Internet Explorer tenían su propio entorno, pero no podíamos correr JavaScript fuera de ellos.

Dahl cambió esto extrayendo **el entorno de ejecución V8** de Chrome, y creando una implementación para servidores.

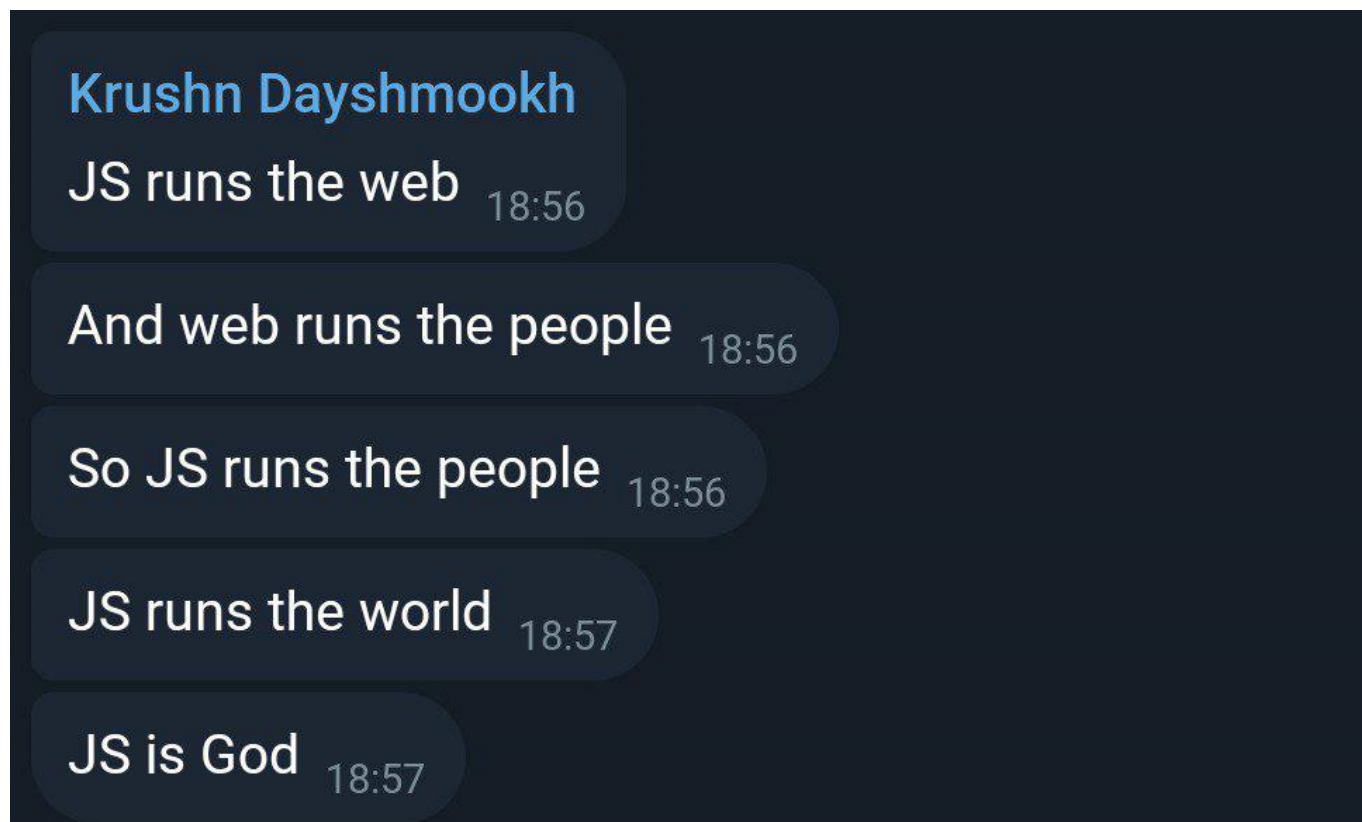
Qué no es

- **No** es un *framework* (Hay *frameworks* contruidos sobre Node.js)
- **No** es un lenguaje de programación (el lenguaje es JavaScript)

task: << [índice de contenidos](#) >>

Por qué Node.js?

- Porque utiliza JavaScript. Lenguaje universal de programación. Su curva de aprendizaje es menor para los que están familiarizados con el JavaScript de lado de cliente:



- Utiliza un modelo orientado a eventos, sin bloqueos de E/S que hace que sea ligero y eficiente, ideal para aplicaciones que requieren datos en tiempo real y que se ejecutan mediante dispositivos distribuidos

- Proporciona una amplia biblioteca de varios módulos de JavaScript (> 1.000.000) que simplifica el desarrollo de aplicaciones web usando Node.js

??? Because Node uses JavaScript's *event driven* and *asynchronous* model, it excels with certain web applications such as chat applications, and apps that need real time, live updating features.

Another (subjective and debateable) benefit is the language of JavaScript being universal. Theoretically, the learning curve may be lower for those who are already familiar with front-end JavaScript.

Related to that, is the concept of sharing code on your front end and back end because they're written in the same language, although many agree that back end code is very different, even if it is the same language.

In addition, Node has a strong and growing ecosystem. This includes packages created by the community, and continuing open source development on the runtime itself.

Because Node uses [Google's V8](#) behind the scenes, Node also benefits from competition between browsers. As Google improves V8 to compete with [Mozilla's SpiderMonkey](#), [Microsoft's Chakra](#) and others, Node reaps the benefits of those improvements as well.

task: << [índice de contenidos](#) >>

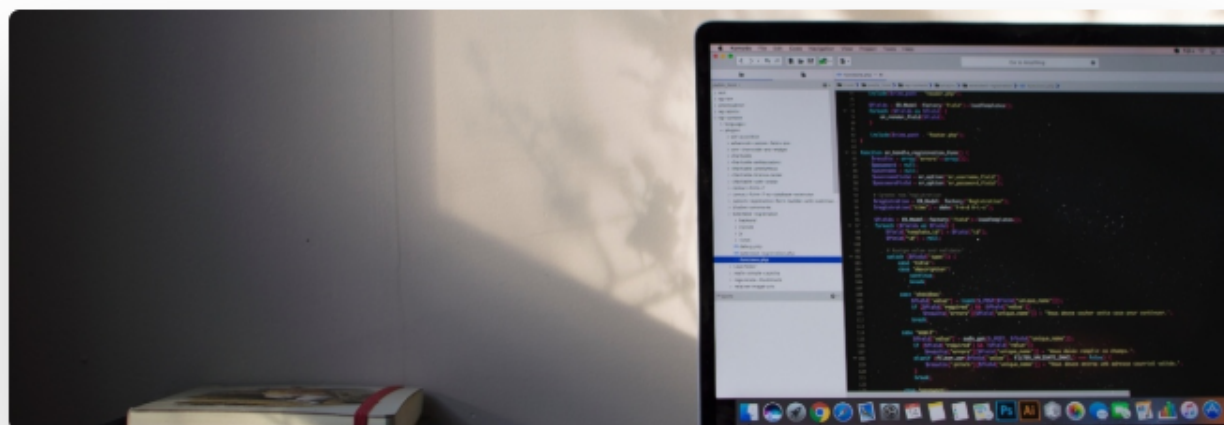
Casos de uso

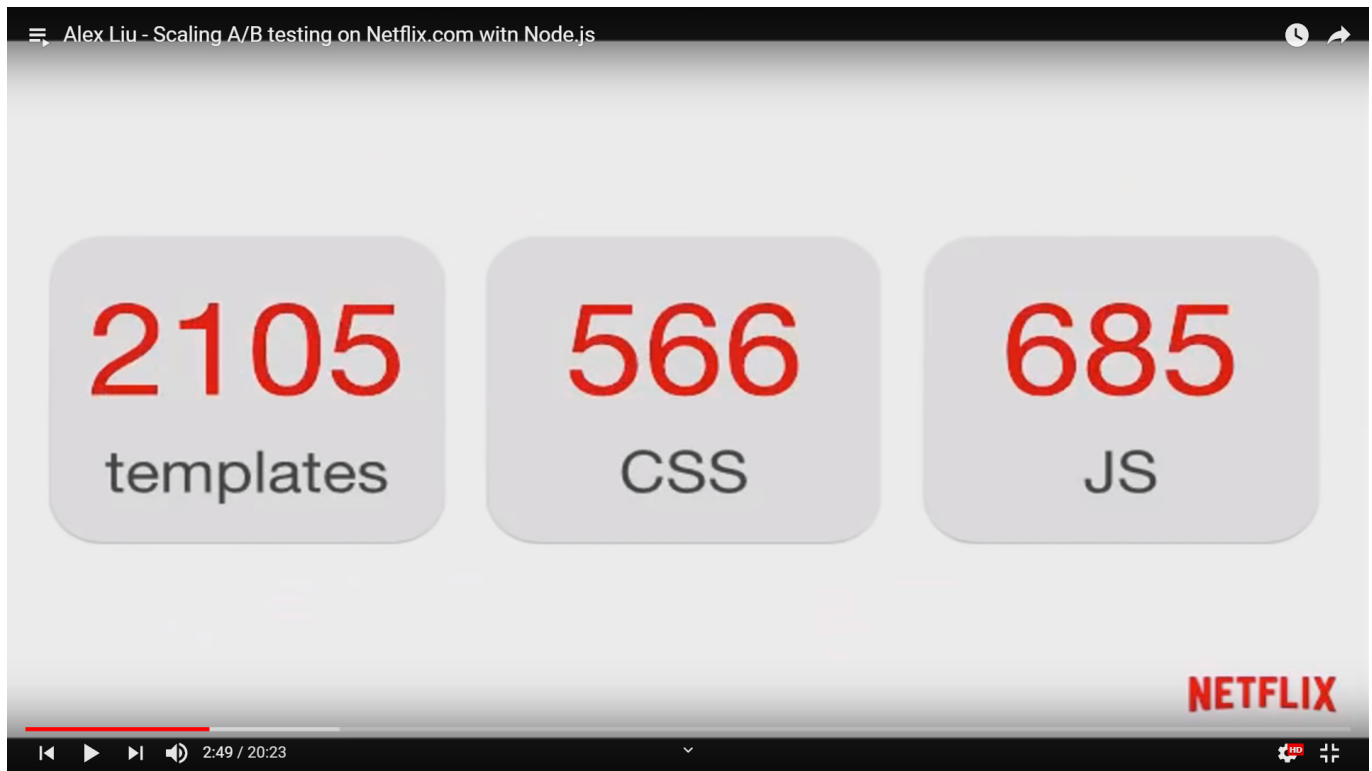
12 Top Applications Written in Node.js - Examples from Big Companies



Natalia Chrzanowska

Mar 17, 2017 | 12 min read [Node.js](#)





name:instalacion task: << [índice de contenidos](#) >>

Instalando Node.js en Windows

- Conectarse a la página oficial: <https://nodejs.org/es/>
 - **LTS versión** de Node.js con Long Term Support (LTS). Esta versión puede no tener disponibles las últimas tecnologías que todavía no se consideran estables.
 - **Current:** esta es la versión más reciente de Node.js e incluye todas las funcionalidades, incluso aquellas más novedosas y que no se consideran estables. Las versiones pares pasan a versiones lts (el actual Node 12 2019 pasará en Octubre a ser LTS, con soporte hasta 2022)
- Descargar el **archivo .msi** y seguir las instrucciones para instalar Node.js. De forma predeterminada, el instalador establece el directorio

`C:\Archivos de programa\nodejs\bin` en las variable de entorno `PATH` de windows.

- Para comprobar si Node se ha instalado, desde un terminal tecleamos:

.inverse[

```
node --version
```

]

task: << [índice de contenidos](#) >>

Node 12

Pasará a LTS en Octubre 2019, incorpora las siguientes mejoras:

1. **Soporte a módulos ECMAScript.** Se podrán utilizar comandos `import` / `export` sin necesidad de utilizar paquetes específicos. Soporte para 3 tipos de `import`:

```
// default exports
import module from 'module'
// named exports
import { namedExport } from 'module'
// namespace exports
import * as module from 'module'
```

Si se importa de un package `CommonJS`, sólo se puede utilizar el `import` de la sintaxis de `default export`: `import module from 'cjs-library'`.

También se puede utilizar `import()` dinámico para cargar ficheros en tiempo de ejecución.

`import()` devuelve una *promesa* que puede ser utilizada en módulos ES y CommonJS, indistintamente.

2. V8 Engine

- Node 12 utilizará la versión 7.4 del motor V8 de google que aporta:
 - Parseo más rápido de JavaScript
 - Llamadas más rápidas a funciones con disparidad de argumentos
 - `await` más rápido
-

3. Uso de campos privados en clases

se utilizará el simbolo `#` symbol.

```
class Greet {
  #name = 'World';
  get name() {
    return this.#name;
  }
  set name(name) {
    this.#name = name;
  }
  sayHello() {
    console.log(`Hello, ${this.#name}`);
  }
}
```

Si se intenta acceder a `#name` desde fuera de la clase se obtendrá un *syntax error*.

```
const greet = new Greet()  
greet.#name = 'NewName';  
// -> SyntaxError  
console.log(greet.#name)  
// -> SyntaxError
```

4. Rendimiento mejorado de arranque

Node 12 presenta una caché de código binario para sus propias librerías que mejora el tiempo de arrancada en un 30%.

5. TLS y Seguridad

Soporte a TLS 1.3, que mejora la seguridad y reduce la latencia. Node reduce también el tiempo de "saludo" requerido por el protocolo HTTPS (HTTPS handshake).

6. Tamaño del Heap de memoria mejorado

Node determina el tamaño de memoria que se asigna a memoria dinámica sin reservarla previamente

7. Funcionalidad de volcado Heap (Heap dump)

Node 12 provides the ability to generate a heap dump making it easier to investigate memory issues.

8. Informes de Diagnóstico

Node offers improved ability to diagnose issues (performance, CPU usage, memory, crashes, etc) within applications by providing an experimental diagnostic report feature.

9. Mejora del uso de módulos nativos de N-API

N-API was released to provide a more stable and native Node module system that would prevent libraries from breaking on every release by providing an ABI-stable abstraction over native JavaScript APIs. Node 12 offers improved support for N-API in combination with worker threads.

10. Otras mejoras

- Worker Threads no longer require a flag
- http has updated its default parser to be llhttp
- assert validates required arguments and adjusts loose assertions
- buffer improvements to make it more stable and secure
- async_hooks removes deprecated functionality
- Make global.process, global.Buffer getters to improve process
- A new welcome message for repl

task: << [índice de contenidos](#) >>

Usos de Node

Utilizaremos Node para escribir aplicaciones de JavaScript puras y, para crear un servidor web con Express.

Primer programa en Node

Crear un archivo en el directorio de trabajo llamado `test.js`.

En este fichero añadiremos el siguiente código:

```
console.log("Hello Node");
```

En el **terminal**:

```
.inverse[
```

```
node test
```

```
Hello Node]
```

Se puede escribir cualquier código JavaScript teniendo en cuenta que **no hay DOM**

task: << [índice de contenidos](#) >>

Segundo programa: Instalando un servidor web

1. Creamos un directorio para la aplicación y entramos:

```
.inverse[
```

```
md prueba-node
```

```
cd prueba-node
```

```
]
```

2. Creamos un archivo **index.js** dentro de la carpeta anterior, con el siguiente contenido:

```
const http = require("http"); const handler = (request, response) => { console.log('Recibimos petición');  
response.end(")
```

Hola Mundo web

```
"); } const server = http.createServer(handler); server.listen(8080); console.log('Server running at http://127.0.0.1:8080/');
```

3. Ejecutamos Node desde el terminal: `.inverse[`

```
node index
```

] ??? Crearemos una aplicación web con "Hello, World!" con los siguientes Componentes: Import de los módulos requeridos – Mediante la directiva `require` que carga módulos Node.js. Create server – Genera un servidor que escucha peticiones http Escucha peticiones y devuelve respuestas – El servidor recibe una petición http de un cliente, navegador o consola y devuelve una respuesta

name:npm task: << [índice de contenidos](#) >>

npm & módulos

Intro

npm es el gestor de paquetes de Node. Significa que todos los programadores pueden compartir código fácilmente usando un terminal de línea de comando

npm tiene el ecosistema de paquetes externos más potente de todas las comunidades de programación con más de un 1.000.000 de módulos para descargar!

Módulos son fragmentos de código reutilizable empaquetados con todo el código del que dependen (dependencias)

task: << [índice de contenidos](#) >>

Utilizando packages npm: Instalación de un módulo

`.inverse[`

```
npm install --save chalk
```

`]`

`npm install` se dirige a los servidores de npm y selecciona un package llamado "chalk". `--save` añade la dependencia al archivo `package.json`.

Además, se ha creado un directorio `node_modules`, donde se almacena el módulo `chalk`.

Para utilizar el módulo dentro de un fichero `index.js`:


```
const chalk = require('chalk');
console.log(chalk.blue('Hello, npm!'));
```

Ejecutamos desde el terminal: .inverse[

```
node index
Hello npm
```

] Toda la funcionalidad se encuentra en la página de [documentación](#) del package.

!!!!IMPORTANTE:: **NUNCA COMMIT DE node_modules a GIT.**

Para recrear el directorio **node_modules** utilizamos el comando **npm install**.

task: << [índice de contenidos](#) >>

Utilizando packages npm: Otros comandos

Por defecto NPM instala las dependencias en modo local. Es decir, crea un directorio node_modules dentro del directorio donde estamos ejecutando el comando. Se puede usar **npm ls** para listar los módulos instalados en local.

En la instalación global los paquetes se almacenan en el directorio system. Estas dependencias se pueden utilizar con un CLI (Command Line Interface) pero no se pueden importar en la aplicación directamente con require()

.inverse[

```
npm install nodemon -g
```

]

Para desinstalar un package:.inverse[

```
npm uninstall chalk
```

] Para actualizar un package:.inverse[

```
npm update chalk
```

] Para buscar un package:.inverse[

```
npm search chalk
```

```
]
```

task: << [índice de contenidos](#) >>

Ejercicio 1

Instalar un package de la siguiente [lista](https://github.com/sindresorhus/awesome-nodejs#weird), en especial de la sección "raritos" (_weird_).

Usar el package seleccionado en el archivo `index.js`.

Ejercicio 2

Importar la librería "os"

Obtener datos sobre cpu, sistema y servidor

Imprimirlos en el navegador

task: << [índice de contenidos](#) >>

Mi primer package npm: Registro

1. Regístrate en [npm](#) para tener una cuenta
2. Desde el [terminal](#) tecleamos:

```
.inverse[
```

```
npm adduser
```

```
]
```

También podemos utilizar: `.inverse[`

```
npm login
```

```
]
```

El terminal nos solicita ahora `username`, `password` y `email`.

En este momento obtenemos un mensaje similar a éste:

```
.inverse[
```

```
  Logged in as rglepe on https://registry.npmjs.org/
```

```
]
```

Importante!!! Hay que verificar el mail desde la cuenta de correo.

task: << [índice de contenidos](#) >>

Mi primer package npm: Preparación

Desde un directorio de trabajo, llamaremos a nuestro package **minimalista** e introduciremos una serie de instrucciones:

```
.inverse[
```

```
  md minimalista
```

```
]
```

Entramos al directorio creado: .inverse[

```
  cd minimalista
```

```
]
```

Para trabajar con npm es necesario definir un archivo de tipo *manifest*, que es un fichero donde se listan todas las dependencias de terceros que tiene nuestro programa. Este archivo es [package.json](#) :

```
.inverse[
```

```
  fsutil file createNew package.json 0
```

```
]
```

Crearemos un package npm con la información imprescindible en el fichero package.json: **nombre** y **versión** .inverse[

```
{  
  "name": "@rglepe/minimalista",
```

```
} "version": "1.0.0"
```

```
]
```

task: << [índice de contenidos](#) >>

Mi primer package npm: Publicación

Para publicar el package npm, corremos la instrucción: `.inverse[`

```
npm publish --access=public
```

] *Por defecto los packages se publican con visibilidad private. Para evitar esto usaremos el modificador `--access=public`*

Una vez ejecutado, aparecerá un signo "+", el nombre del package y la versión. `.inverse[`

```
+ @rglepe/minimalista@1.0.0
```

] !!!Ahora ya pertenecemos al **club npm**!!!

Recibiremos también un mail:

Successfully published @rglepe/minimalista@1.0.0

Recibidos x Notificaciones x



npm Inc support <support@npmjs.com>

para mí ▾

Hi rglepe!

A new version of the package @rglepe/minimalista (1.0.0) was published at 2019-08-20T10:38:54.211Z from 139.47.21.210. The shasum of this package was caeaf23a6dfe44b756cf5b7c128fc5437bffe66e.

If you have questions or security concerns, you can reply to this message or email support@npmjs.com.

npm loves you.

↩ Responder

➡ Reenviar

Ya tenemos la versión 1 del package:

<https://www.npmjs.com/package/@rglepe/minimalista/v/1.0.0>

task: << [índice de contenidos](#) >>

Package npm completo

Usaremos un repositorio [GitHub](#) al que añadiremos un fichero *readme* con las instrucciones necesarias para utilizar el package y...el código del package.

fichero README!

Vamos añadir una serie de etiquetas desde shields.io de forma que parezca que somos muy profesionales:

Vamos añadir la versión actual del package (npm scoped):

A badge showing 'npm' in a dark grey box and 'v2.0.3' in a blue box.

Y el tamaño del package (npm bundled size minified).

**Si todavía no hemos introducido el código, nos dará un error de inaccesible:*

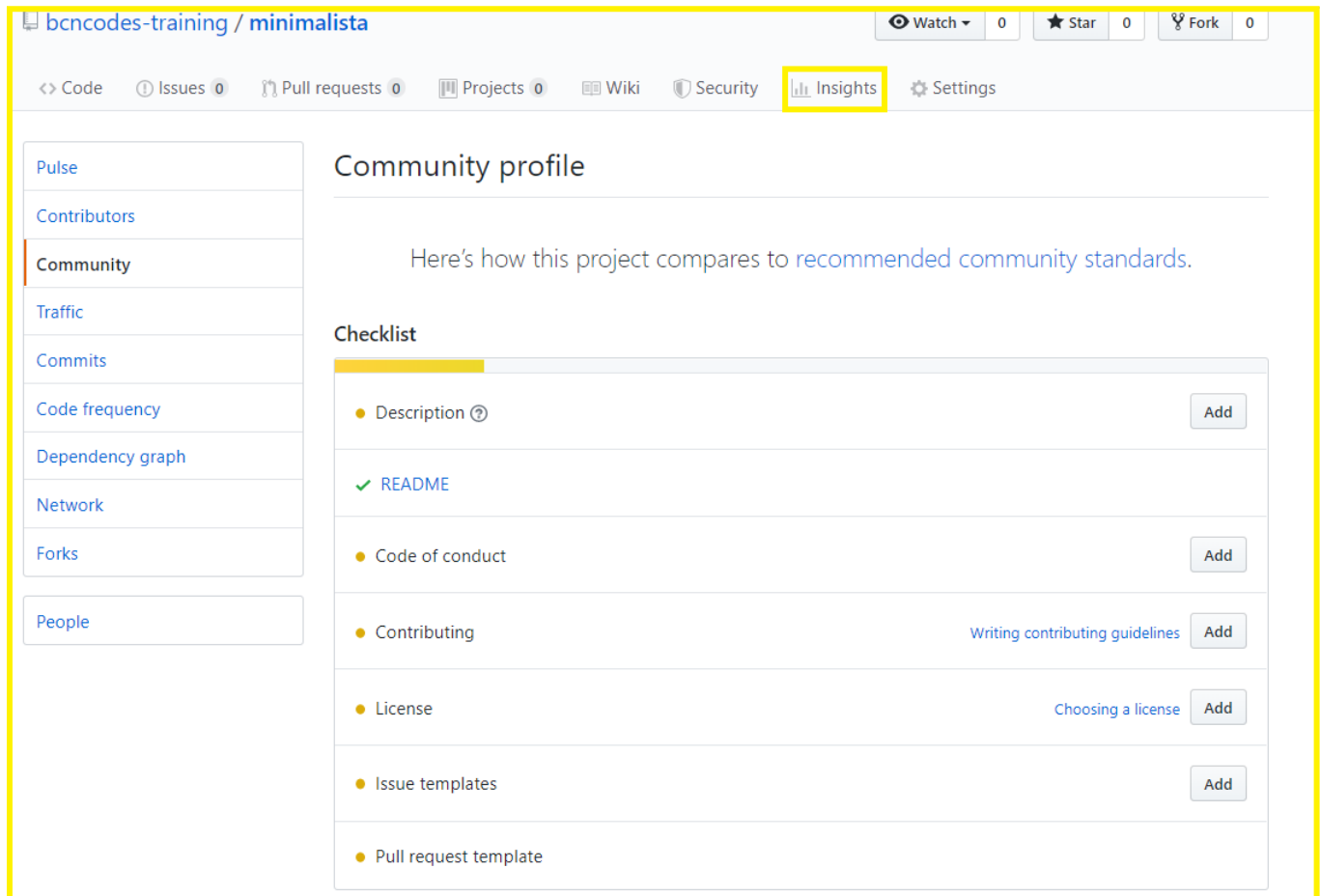
A badge showing 'minified size' in a dark grey box and '268 B' in a blue box.

task: << [índice de contenidos](#) >>

Package npm completo

Licencia

Una licencia permite conocer las condiciones de uso de nuestro package existen [muchas diferentes](#). Dentro del apartado **insights** de cada repositorio GitHub podemos seleccionar distintos estados — incluyendo los recomendados por la comunidad:



bcncodes-training / minimalista

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Security Insights Settings

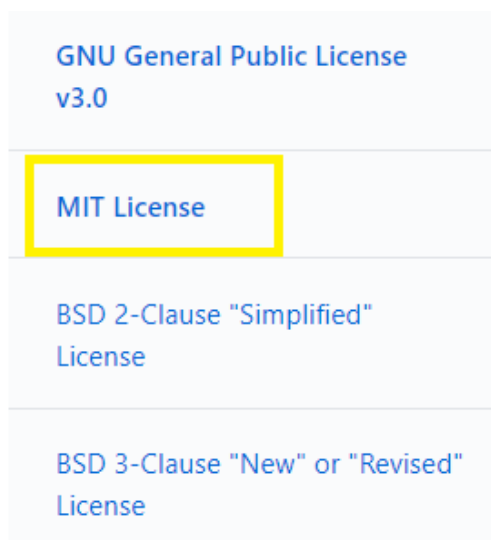
Community profile

Here's how this project compares to recommended community standards.

Checklist

- Description ⓘ Add
- ✓ README
- Code of conduct Add
- Contributing Writing contributing guidelines Add
- License Choosing a license Add
- Issue templates Add
- Pull request template

Aquí seleccionamos "Choosing a License", y después:



GNU General Public License v3.0

MIT License

BSD 2-Clause "Simplified" License

BSD 3-Clause "New" or "Revised" License



Choose a license to add

Select a template on the left
Learn more about [which license](#)

task: << índice de contenidos >>

Package npm completo

Código

Una función para eliminar espacios de un string:

```
module.exports = function minimalista(string) {  
  if (typeof string !== "string") throw new TypeError  
    ("Minimalista requiere un string!");  
  return string.replace(/\s/g, "");  
};
```

Añadimos el código dentro de un archivo **index.js**. Este será el *entry point* a nuestro package.

Completamos nuestro **package.json** básico y añadimos instrucciones al fichero **readme.md**, para explicar el uso de nuestro código.

task: << [índice de contenidos](#) >>

Package npm completo

package.json

Añadimos:

```
{  
  "name": "@rglepe/minimalista",  
  "version": "1.0.0",  
  "description": "Removes all spaces from a string",  
  "license": "MIT",  
  "repository": "bcncodes-training/minimalista",  
  "main": "index.js",  
  "keywords": [  
    "minimalista",  
    "npm",  
    "package",  
    "bcncodes-training"  
  ]  
}
```

task: << [índice de contenidos](#) >>

Package npm completo

readme.md

Añadimos instrucciones de cómo utilizar el package. Si necesitas alguna plantilla, échale un vistazo a la comunidad y usa alguno de sus formatos para empezar.

28 lines (18 sloc) | 592 Bytes

Raw Blame History

@rglepe/minimalista

npm v2.0.3 minified size 268 B

(<https://www.npmjs.com/package/@rglepe/minimalista>)

Removes all spaces from a string.

Install

```
$ npm install @rglepe/minimalista
```

Usage

```
const minimalista = require("@rglepe/minimalista");

minimalista("Me sobra espacio!");
//=> "Mesobraespacio!"

minimalista(1337);
//=> Uncaught TypeError: Minimalista necesita un string!
//   at minimalista (<anonymous>:2:41)
//   at <anonymous>:1:1
```

task: << índice de contenidos >>

Package npm completo

1. Actualizamos la versión npm mediante la instrucción: `.inverse[`

```
npm version major
```

```
]
```

Que genera: `.inverse[`

```
v2.0.0
```

```
]
```

2. Publicamos: `.inverse[`

```
npm publish --access=public
```



```
]
```

Obtenemos: `.inverse[`

```
+ @rglepe/minimalista@2.0.0
```

```
]
```

task: << índice de contenidos >>

Utilidades

- [Package Phobia](#) ofrece un resumen del package npm.
- También podemos revisar los ficheros en [Unpkg](#)

task: << índice de contenidos >> ####Ejercicio a entregar: Introducción a Node

Clona la rama del repositorio github.com/bcncodes-training/intro-node.git que lleva tu nombre. Una vez completado el ejercicio actualiza tu rama.

task: << índice de contenidos >>

Recursos extras

- [Getting the best out of NPM](#) - More advanced NPM concepts

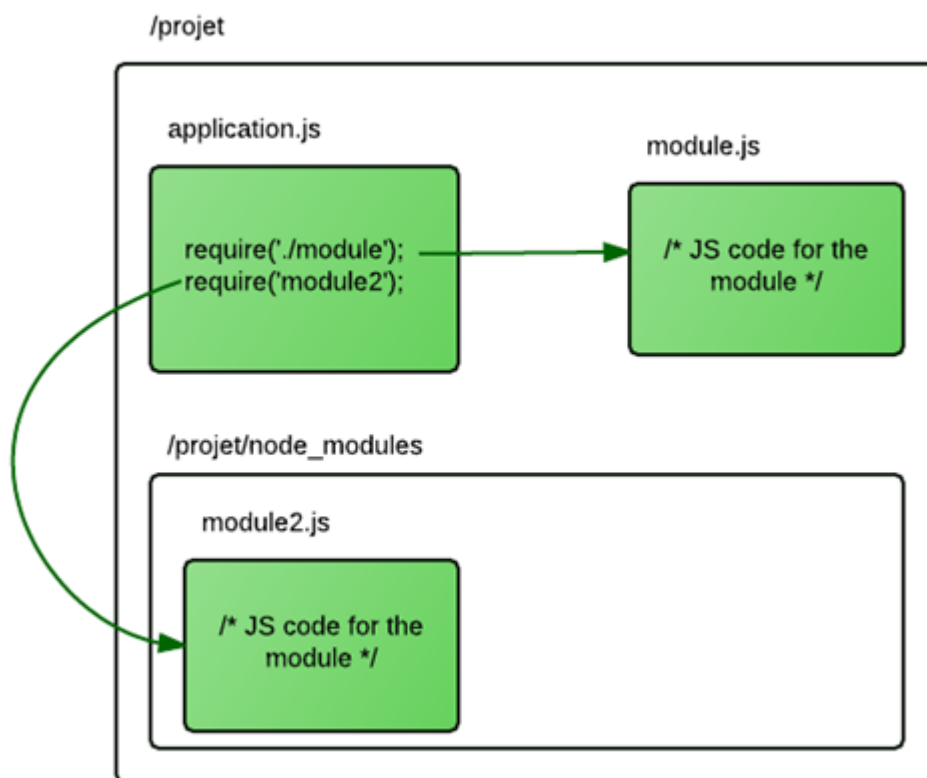
name: modulos class: center, middle, inverse

[Módulos en Node.js](#Qué es Node.js?)

name:modulosintro task: << índice de contenidos >>

Módulos Node: Intro

Los módulos en Node.js son funcionalidades simples o complejas organizadas en uno o varios ficheros JavaScript, completamente independientes entre si, y que pueden ser mezclados, añadidos y eliminados, sin alterar el sistema en su conjunto.



task: << [índice de contenidos](#) >>

Módulos Node: Ventajas

Sobre todo cuando el sistema crece, y queremos independizar bloques de código

Mantenimiento, un modulo bien diseñado, disminuye al máximo las dependencias, por tanto su crecimiento o rediseño no afecta a los demás

Espacios de nombre, cada módulo, es capaz de crear sus variables en un espacio privado (tiene contexto propio), sin contaminar al resto de módulos

Reutilización, un módulo es susceptible de ser usado por diversas aplicaciones en distintos contextos, ya que expone una interficie

task: << [índice de contenidos](#) >>

Origen de los módulos de Node.js: Revealing Module Pattern

- El principio central de este patrón de diseño es que todas las funcionalidades y variables deberían estar encapsuladas a menos que se expongan de forma deliberada
- Para implementar el patrón en JavaScript se utilizan los [closures](#) y las [IIFEs](#)

```

var modulo = (function() {
    var privMetodo = function() {...};
  
```

```
var privVariable = [];  
var export = {  
  publicMetodo: function() {...},  
  publicVar: function() {...}  
}  
  
return export;  
})();
```

task: << [índice de contenidos](#) >>

Módulos Node: CommonJS

- Es un grupo de trabajo que surge con el objetivo de estandarizar el ecosistema JavaScript y una de sus propuestas fueron los módulos CommonJS
- La librería CommonJS es, por tanto, la implementación del patrón revealing module en javascript
- CommonJS ofrece:
 - Una sintaxis compacta
 - Diseño para carga síncrona
 - Principal uso en servidores

task: << [índice de contenidos](#) >>

Módulos Node: Definición de un módulo CommonJS

La definición de un módulo CommonJS se realiza de la siguiente manera:

```
const myModule {  
  
  hello: () => 'hello!';  
  goodbye: () => 'goodbye!';  
  
}  
  
module.exports = myModule;
```

task: << [índice de contenidos](#) >>

Módulos Node: Utilización de un módulo CommonJS

Para utilizar el módulo desarrollado con CommonJS usamos el método `require()`:

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

myModule.hello();
myModule.goodbye();
```

De este modo, se evita la contaminación de namespaces globales, y las dependencias se vuelven mas explícitas.

task: << [índice de contenidos](#) >>

Módulos Node: CommonJS en Node

- Node lleva incorporado en el core a CommonJS como sistema de módulos
 - En NodeJS cada fichero .js es tratado como un módulo separado
 - Los módulos **se cargan en memoria** una vez y luego **son reutilizados**
-

task: << [índice de contenidos](#) >>

Diseño modular: Carga de módulos NodeJS

Node.js puede cargar dependencias utilizando la palabra clave “require” y asignando el módulo cargado a una variable, como se puede ver en el ejemplo:

```
const http = require('http');
const dns = require('dns');
```

También se pueden cargar archivos en rutas relativas:

```
const myFile = require('./myFile');
```

Si se instalan módulos desde npm, se manejan como los nativos, sin necesidad de especificar la ruta absoluta o relativa:

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

task: << [índice de contenidos](#) >>

Diseño modular: Exportar desde módulos NodeJS

Los módulos en Node.js no se inyectan automáticamente en el ámbito global, sino que simplemente se asignan a una variable.

Para exportar funciones o variables de un módulo se utiliza el objeto `module.exports` o `exports`.

- **Exports** es un objeto especial que se incluye en todos los ficheros JS de Node, por defecto.
- **module** es una variable que representa el módulo actual y
- **exports** es un objeto que se expondrá como módulo

```
// book.js
module.exports.name = 'pepe'; //asigna un valor al módulo
module.exports.read = function() {
  console.log('Hola ' + exports.name);
}
//asigna una función a la propiedad read del objeto exports
//devuelve el objeto {read: function(){...}}
```

task: << [índice de contenidos](#) >>

Diseño modular: Exportar desde módulos NodeJS

Con `module.exports` se puede exportar cualquier tipo de objetos, por ejemplo clases:

```
// book.js
module.exports = class book {
  public name;
  constructor(name){this.name = name};
  read = function() {
    console.log('Hola ' + this.name);
  }}

```

task: << [índice de contenidos](#) >>

Diseño modular: Exportar desde módulos NodeJS

Ejemplos de buenos y malos usos de `exports`:

```
// calculator-exports-examples.js

// good
module.exports = {
  add(a,b) { return a+b }
}

// good
module.exports.subtract = (a,b) => a-b
```

```
// valid
exports = module.exports

// good and simply a shorter version of the code above
exports.multiply = (a,b) => a*b

// bad, exports is never exported
exports = {
  divide(a,b) { return a/b }
}
```

task: << [índice de contenidos](#) >>

Diseño modular: ES Modules

Un módulo ES6 es un archivo que contiene código JS. No existe una palabra clave `module`; un módulo se lee casi como cualquier script. Existen dos diferencias.

Se puede usar `import` y `export` en los módulos.

Se puede hacer `export` de cualquier `function`, `class`, `var`, `let`, o `const` declarado al nivel más alto del script.

```
//kittydar.js - Encontrar todos los gatos en una imagen.

export function detectCats(canvas, options) {
  var kittydar = new Kittydar(options);
  return kittydar.detectCats(canvas);
}

export class Kittydar {
  //... varios métodos de procesamiento de imágenes ...
}

// esta función no será exportada.
function resizeCanvas(){
  ... }

```

Si queremos exportar todo el módulo podemos asignar: `export default`.

task: << [índice de contenidos](#) >>

Diseño modular: ES Modules

En un archivo separado, podemos importar y usar la función `detectCats()`:

```
//demo.js - Programa demo Kittydar

import {detectCats} from "kittydar.js";

function go(){
  var canvas = document.getElementById("catpix");
  var cats = detectCats(canvas);
  drawRectangles(canvas, cats);
}
```

Para importar múltiples nombre de un módulo, se escribiría:

```
import {detectCats, Kittydar} from "kittydar.js"
```

task: << [índice de contenidos](#) >>

Diseño modular: ES Modules en NodeJS

En NodeJS los módulos ES se encuentran en fase experimental hasta que entre en lts la versión 12. Para trabajar con ellos, de momento se requiere crear un fichero con la extensión `.mjs` y activarlo con el flag `--experimental-modules`:

`.inverse[`

```
node --experimental-modules my-app.mjs
```

`]`

Para exportar un módulo se escribiría:

```
//01-kettle.mjs
export const spout = 'the spout'
export const handle = 'the handle'
export const tea = 'hot tea'
```

Para usarlo en otro módulo 01-main.mjs

```
import {handle, spout, tea} from './01-kettle.mjs'

console.log(handle) // ==> the handle
console.log(spout) // ==> the spout
console.log(tea) // ==> hot tea
```

task: << [índice de contenidos](#) >>

Diferencias CJS Modules vs MJS Modules

- Los módulos ES6 se cargan de forma asíncrona, mientras que los CJS se cargan de forma síncrona.
- Los módulos ES6 son un estándar multiplataforma, son compatibles con Node.js y navegadores.
- Los Imports y Exports en ES6 son estáticos. Permite utilizar sólo la parte útil cuando se usan librerías de terceros.
- En CJS los imports son dinámicos, requieren asignarlos a una variable. Esto ralentiza la carga de los módulos.

task: << [índice de contenidos](#) >>

Notas sobre el diseño modular

1. Al empezar a diseñar módulos, dos conceptos clave del desarrollo software: **cohesión** y **desacoplamiento**. Hay que crear módulos que se encuentren muy cohesionados.
 - Los métodos dentro de un módulo deben tener una relación interna
 - Cuanto más **cohesionado** se encuentran los miembros de un módulo, mayor **reutilización**, **mantenimiento** y **evolución** podría llegar a tener, en términos generales.
 - El concepto de **acoplamiento**, tiene que ver con el **número de dependencias** que un módulo tiene con otros módulos.
2. Todo módulo, para que esté bien diseñado, debería cumplir con el principio de **única responsabilidad**.
 - Puede ser complicado y depende de muchos parámetros del contexto.
 - Cuantas más dependencias tiene un módulo más difícil de testear y más expuesto a cambios en el futuro podría estar.
 - Como, no depender de nada, no es posible, tendremos que crear formas en las que incluir esas dependencias nos generen la menor fricción posible.

task: << [índice de contenidos](#) >>

Notas sobre el diseño modular

- Tenemos que intentar que dentro de nuestro árbol de dependencias, los módulos que se encuentren en la parte más profundas (en la raíz) sean lo más genéricos posibles y que según nos acerquemos a las hojas, los módulos sean más específicos.

- En capas superiores, el código se encuentra ya muy ligado a la lógica de nuestro negocio y esto nos obliga a que tenga que ser así.
 - Para conseguir desacoplar el código hay que utilizar la inyección de dependencias:
<https://www.npmjs.com/search?q=dependency%20injection>
-

task: << índice de contenidos >>

Ejercicio 3

- Crear un módulo para importar la librería “os” y obtener datos sobre cpu, sistema y servidor
- Crear otro módulo para Imprimir los datos obtenidos en el paso anterior, en el navegador.

TIPS: Utilizar *backticks* (tildes francesas) para incluir variables: `${}`

BONUS: Mismo ejercicio en ES Modules

task: << índice de contenidos >>

Ejercicio 4

Guardar los datos anteriores en un fichero.

TIPS: Utilizar el método `appendFile()` del módulo `fs`. Requiere 3 parámetros:

- *Nombre fichero*,
- *Contenido*
- Función `callback`:

```
function(error) {if(error){console.log('se ha producido un error');}}
```

task: << índice de contenidos >>

Ejercicio 5

- Crea un módulo que encapsule el procesado de strings para generar procesos:
 - Primera mayúscula
 - Tipo Oración
 - Minúsculas
-

name: callbacks class: center, middle, inverse

[Callbacks](#Qué es Node.js?)

task: << [índice de contenidos](#) >>

callbacks: conceptos

- En JavaScript se usa la notación **CPS** (abreviatura de *continuation passing style*) para escribir el código de un programa en el que las Continuaciones se escriben y se pasan de forma explícita.
- Cuando se escribe un programa en notación CPS, cada función recibe un parámetro adicional, que representa la Continuación de la función. En lugar de retornar, la función invocará la continuación recibida pasando el valor de retorno. De esta forma, las funciones nunca regresan al código que las llamó, sino que la ejecución del programa transcurre “hacia adelante” sin retornar hasta que el programa finalice.
- CPS es un concepto general y no siempre está asociado a programación asíncrona.

task: << [índice de contenidos](#) >>

callbacks: conceptos

- Las funciones callbacks, por tanto, sustituyen a la instrucción **return** que requiere la ejecución síncrona.
- En Node todas las APIs soportan el uso de callbacks:

```
//Callbacks en programación síncrona function add(a, b, callback) { callback(a + b); }  
console.log('before'); add(1, 2, function(result) { console.log('Result: ' + result); }); console.log('after');  
//Resultado before Result: 3 after
```

```
//Callbacks en programación asíncrona function addAsync(a, b, callback) { setTimeout(function() {  
callback(a + b); }, 100); } console.log('before'); addAsync(1, 2, function(result) { console.log('Result: ' +  
result); }); console.log('after'); //Resultado before after Result: 3
```

task: << [índice de contenidos](#) >>

callbacks: conceptos

Ejemplo: Lectura de un fichero en modo síncrono:

Crear un fichero input.txt con un texto de prueba, en el directorio de un proyecto: ejemploCallbacks Crear un fichero main.js que importe el módulo fs Volcar el contenido del fichero a una variable con el método:

```
let data = fs.readFileSync('input.txt');
```

Imprimir el contenido del fichero por consola. Imprimir un mensaje por consola ('Fin del programa').

task: << [índice de contenidos](#) >>

callbacks: conceptos

Ejemplo: Lectura de un fichero en modo asíncrono:

Crear un fichero input.txt con un texto de prueba, en el directorio de un proyecto: ejemploCallbacks Crear un fichero main.js que importe el módulo fs Leer el fichero con el método:

```
let fs.readFile('input.txt', (err, data) => {
  if (err) return console.error(err); console.log(data.toString());
});
;
```

Imprimir el contenido del fichero por consola. Imprimir un mensaje por consola ('Fin del programa').

task: << [índice de contenidos](#) >>

Convenciones de Callbacks en NodeJS

- Las funciones *callback* siempre son el último argumento de la función. Ej:

```
fs.readFile(filename, [options], callback)
```

- Los errores siempre se sitúan al principio de la función *callback* y deben ser del tipo **Error**. Ej:

```
fs.readFile('foo.txt', 'utf8', function(err, data) {
  if(err){
    handleError(err);
  } else {
    processData(data);
  }
});
```

task: << [índice de contenidos](#) >>

Convenciones de Callbacks en NodeJS

- En CPS asíncrono los errores se tienen que propagar a la siguiente función de callback. Ej:

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

```
const readJSON = (filename, callback) => { fs.readFile(filename, 'utf8', (err,data)=>{
```

```
  let parsed;
```

```
    if(err)
      //propagate the error and exit the current function
      return callback(err);

    try {
      //parse the file contents
      parsed = JSON.parse(data);
    } catch(err) {
      // catch parsing errors
      return callback(err);
    }
    // no errors, propagate just the data

    callback(null, parsed);
  });
```

```
};
```

Ejercicio5

- Crear una función que pase el contenido de un array a un callback y éste genere un nuevo array que multiplique cada valor del array anterior por 2.

TIPS: imitar a la función `Array.map()`

name: promesas class: center, middle, inverse

[Promesas](#Qué es Node.js?)

name:promesas task: << [índice de contenidos](#) >>

Promesas: Intro

Una **promesa** es un objeto que representa la **finalización con éxito o con fallo** de una **operación asíncrona**. En esencia, una promesa es un objeto retornado al cual se le pueden asignar *callbacks*, en lugar de pasar un *callback* a una función

Es un recurso para manejar un proceso asíncrono de manera síncrona. Representa un valor que podemos manejar en algún momento, en el futuro de forma más sencilla que los *callbacks*

Una promesa puede ser creada en el código o puede ser devuelta por un package node externo

???

task: << [índice de contenidos](#) >>

Cómo crear Promesas

Se crean utilizando el operador `new` sobre el objeto **Promise** pasando como único parametro una función ejecutora. Esta función a su vez recibe dos parametros: un *callback* que será ejecutado cuando la promesa se ha cumplido (**resolve**) y otro que será ejecutado cuando no (**reject**).

La lógica de programación debe controlar cuándo y dónde llamar a esas funciones. Si la operación tiene éxito se pasan los datos al código que usa la promesa si no, se pasa el error.

```
const miPromesa = new Promise(function(resolve, reject) {  
  
  if (/* condition */) {  
    resolve(/* value */); // fulfilled successfully  
  }  
  else {  
    reject(/* reason */); // error, rejected  
  }  
});
```

task: << [índice de contenidos](#) >>

Estados de las promesas

De este modo, la **promesa** tiene tres estados:

- **Pendiente.** Hasta que se resuelve, permanece en estado pendiente
- **Completada.** Cuando se invoca la función **resolve** con el valor devuelto.
- **Rechazada.** Si se invoca la función **reject** con un valor de rechazo.

Una **Promesa** solo puede ser completada o rechazada una vez.

Una **Promesa** también se puede resolver de forma inmediata utilizando el método **Promise.resolve()**

```
let myMeth = Promise.resolve(42);
```

task: << [índice de contenidos](#) >>

Consumir Promesas: **.then()**

Una vez tengamos la promesa, se puede pasar como **valor**. La promesa es una representación del valor futuro, y puede ser devuelta a partir de una **función**, pasada como **parámetro** y usada como cualquier otro valor

Para consumir la promesa - es decir, utilizar el valor una vez que se ha completado - asociamos una **función** que gestione la promesa usando el método **.then()**.

```
let p = new Promise((resolve, reject) => resolve("BCNcodes"));
p.then((val) => console.log(val)); // BCNcodes
```

El método **.then()** recibe **dos parámetros posibles**.

- El primero es la función que se invocará si la promesa se ha completado.
- El segundo es la función que se invocará si la promesa se ha rechazado.
-

```
p.then((val) => console.log("fulfilled:", val),
      (err) => console.log("rejected: ", err));
```

task: << [índice de contenidos](#) >>

Consumir Promesas: **.catch()**

👁️: Si se omite el primer parámetro del método **.then()**, enviando un **null**, sería equivalente a encadenar el método **Promise.catch()**, que toma como único parámetro la llamada del rechazo de la promesa.

Son equivalentes:

```
p.then((val) => console.log("fulfilled:", val))
  .catch((err) => console.log("rejected:", err));

p.then((val) => console.log("fulfilled:", val))
  .then(null, (err) => console.log("rejected:", err));
```

Como se ve, la función **.then()** devuelve a su vez una promesa permitiendo que sean encadenadas fácilmente. Además el valor que se devuelve en **.then()** se pasa como parámetro al siguiente.

Mejor utilizar **.catch()** para gestionar los errores que la expresión **.then(null, function).catch()** es más explícito y se puede encadenar al final de una cadena de promesas, devolviendo cualquier excepción o rechazo de la promesa original o de cualquiera otra.

???

Dealing with errors

You should use **.catch()** for *handling errors*, rather than **.then(null, function)**. Using **.catch()** is more explicit and idiomatic, and when chaining you can have a single **.catch()** at the end of the chain to handle **any rejection or thrown exceptions** from either the original promise or any of it's handlers.

Throwing an exception in a **Promise** will automatically **reject** that **Promise** as well. This is the same for **.then()** handlers and their

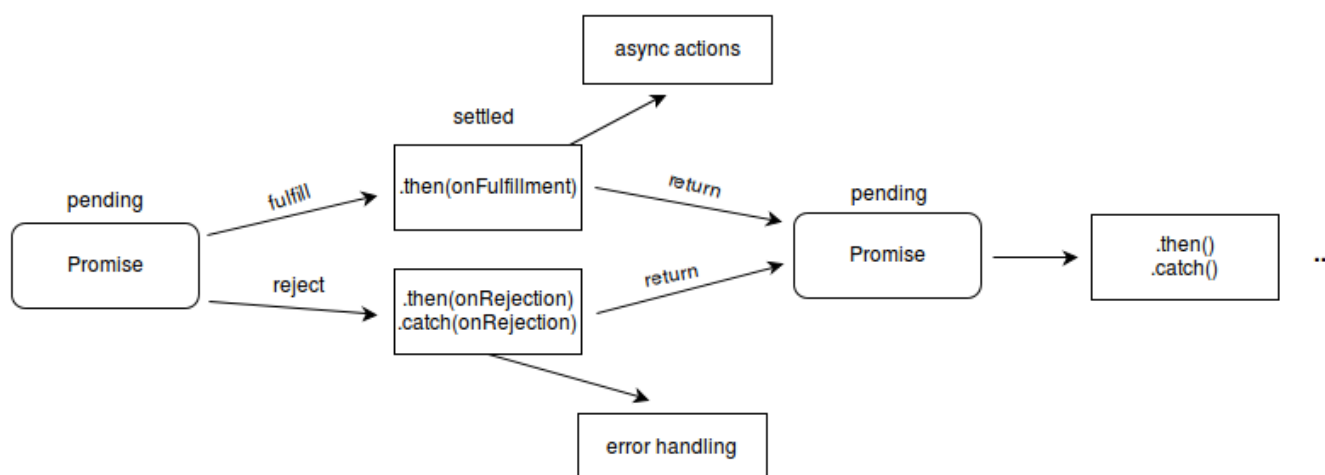
results and **return** values as well. A thrown error is wrapped in a **Promise** and treated as a rejection.

task: << índice de contenidos >>

Encadenamiento de promesas

```
var p1 = new Promise((resolve, reject) => {
  if (true)
    throw new Error("rejected!");
  else
    resolve(4);
});

p1.then((val) => val + 2)
  .then((val) => console.log("got", val))
  .catch((err) => console.log("error: ", err.message));
// => error: rejected!
```



task: << índice de contenidos >>

Ejemplo Promesa

(<https://scotch.io/tutorials/javascript-promises-for-dummies>)

Imagina que eres pequeñ@ y tu madre te promete que te regalará un móvil nuevo si te portas bien.

Creamos la promesa:

```
const isMomHappy = true;
const willIGetNewPhone = new Promise((resolve, reject) => {
  if (isMomHappy) {
    const phone = {
      brand: 'Samsung',
      color: 'black'
    };
    resolve(phone);
  } else {
    reject(new Error('Mom is not happy'));
  }
});
```

```
        resolve(phone);
    } else {
        const reason = new Error('mom is not happy');
        reject(reason);
    }
});
```

task: << [índice de contenidos](#) >>

Ejemplo Promesa

(<https://scotch.io/tutorials/javascript-promises-for-dummies>)

Llamamos a la promesa:

```
const askMom = function() {
    willIGetNewPhone
        .then(console.log)
        .catch(error => console.log(error.message));
}

askMom();
```

task: << [índice de contenidos](#) >>

Ejemplo Promesa

(<https://scotch.io/tutorials/javascript-promises-for-dummies>)

Ahora prometes que si tienes un móvil nuevo se lo enseñarás a tus amigos. Añades otra promesa

```
const showOff = function (phone) {
    const message =
        `Hey friend, I have a new ${phone.color} ${phone.brand} phone` ;

    return Promise.resolve(message);
};

// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
};

askMom();
```

task: << [índice de contenidos](#) >>

###Ejercicio 6

Crear una operación que devuelva el resultado de una suma asíncrona en una promesa. Después elevar al cuadrado el resultado.

- Si algún sumando es 0 devolverá error.
 - Si el resultado de la potencia es mayor que 100 devolverá error.
-

###Ejercicio node-meQuiere-noMeQuiere

Deshoja la margarita y descubre si tu amad@ te quiere o no te quiere.

Descripción

- Partís de una margarita de número variable de pétalos.
- Deshojar la margarita significa arrancarle un pétalo cada segundo (tip: pensad en setInterval())
- Cada pétalo **"arrancado"** significa imprimir en pantalla "me quiere!" o "no me quiere!!!".
- El programa resolverá con el valor último pintado en *AZUL* si me quiere o en *ROJO* si no me quiere (tip: chalk)

Detalles

Devuelve una promesa con el mensaje que toque y encadena la impresión por la consola.

task: << [índice de contenidos](#) >>

Promesas: Métodos estáticos

- **Promise.resolve**: Crea una promesa de cumplimiento

```
let promise = Promise.resolve(valor);
```

- **Promise.reject**: Crea una promesa de rechazo

```
let promise = Promise.reject(error);
```

- **Promise.all**: Ejecuta muchas promesas en paralelo y espera hasta que todas ellas han terminado. El iterador normalmente es un array que devuelve una promesa. Si se rechazan se devuelve el argumento de la primera promesa que se rechaza. Es un método útil para agregar el resultado de múltiples promesas.

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 3000)), // 1
  new Promise((resolve, reject) => setTimeout(() => resolve(2), 2000)), // 2
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 1000)) // 3
]).then(console.log);
//1,2,3 when promises are ready: each promise contributes an array member
```

???

Promises All

There will be sometime we will need to wait more than one **Promise** to complete to continue our program, no worries, you can use **Promise.all()** 😊

The **Promise.all()** method returns a single **Promise** that **resolves when all of the promises in the iterable argument have resolved** or when the iterable argument contains no promises. It **rejects with the reason of the first promise that rejects**. This method can be useful for aggregating the results of multiple promises.

Fulfillment

- If an empty iterable is passed, then this method returns (synchronously) an already resolved promise.
- If all of the passed-in promises fulfill, or are not promises, the promise returned by **Promise.all** is fulfilled asynchronously.
- In all cases, the **returned promise is fulfilled with an array containing all the values of the iterable passed as argument (also non-promise values)**.

Rejection If any of the passed-in promises **reject**, **Promise.all** asynchronously rejects with the value of the **promise that rejected**, whether or not the other promises have resolved.

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

task: << [índice de contenidos](#) >>

Promesas estáticas: Promise.all

Una estrategia utilizada es mapear un array de datos a un array de promesas y después envolverlo con **Promise.all**. Ej:

```

let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];
// map every url to the promise fetch(github url)

let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved

Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));

```

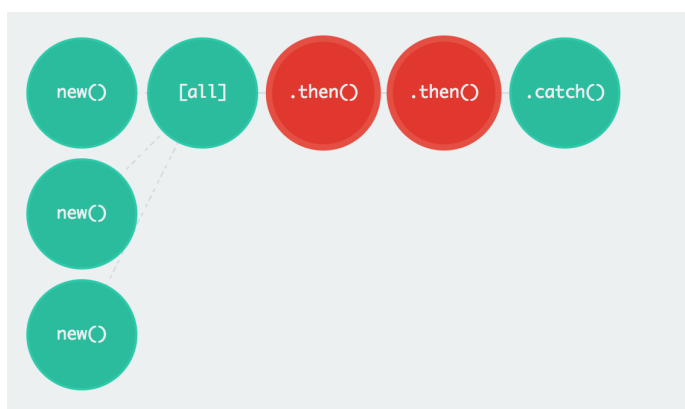
task: << [índice de contenidos](#) >>

Visualización gráfica de las promesas

```

Promise.all([
  new Promise(resolve => setTimeout(resolve, 1500)),
  new Promise(resolve => setTimeout(resolve, 900)),
  new Promise(resolve => setTimeout(resolve, 2200))
])
  .then(results => results.length.b.c)
  .then(c => console.info(c))
  .catch(err => console.error(err))

```



La web [Promises Visualization](#) permite comprobar de forma gráfica como se resuelven las promesas después de cada `setTimeout`. ??? Go ahead and play around with the different methods of `Promise`! 💪

Summary

We just learn a super powerfull tool for dealing with asynchrnous code. `Promises` give us the ability to write asynchronous code in a synchronous fashion, with flat indentation and a single exception channel.

Promises give us guarantees of no race conditions and immutability of the future value represented by the **Promise** (unlike callbacks and events).

task: << índice de contenidos >>

Ejercicio 7

Recorrer un directorio manejando la respuesta de si es un fichero o un directorio con promesas.

- Si es un fichero concatenar el nombre del fichero al directorio
 - Si es un directorio volver a invocar a la función.
 - Obtener el resultado por consola.
-

name: asyncawait class: center, middle, inverse

[async|await](#Qué es Node.js?)

name: introasync task: << índice de contenidos >>

async|await

Desde ES7 y nodejs > 6 se introduce la expresión **async/await** se utiliza para manejar la asincronía:

- **async** delante de una función significa que esa función devolverá una promesa.

```
async function f(){return 1;}
f().then(console.log);
//es igual a async function f() {return Promise.resolve(1)}
```

- **await** trabaja sólo con funciones **async** y lo que hace es parar la ejecución del programa hasta que se completa la función **async**.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait till the promise resolves (*)
  console.log(result); // "done!"
}

f();
```

task: << índice de contenidos >>

async|await: Manejo de errores

Manejar los errores con **try-catch**

```
async function f() {
  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}

f();
```

Ejercicio 8

Rehacer el ejemplo del "móvil nuevo" con async/await

[Volver al Menú Principal](#)

name: eventemitter class: center, middle, inverse

[Event-Driven Programming](#Qué es Node.js?)

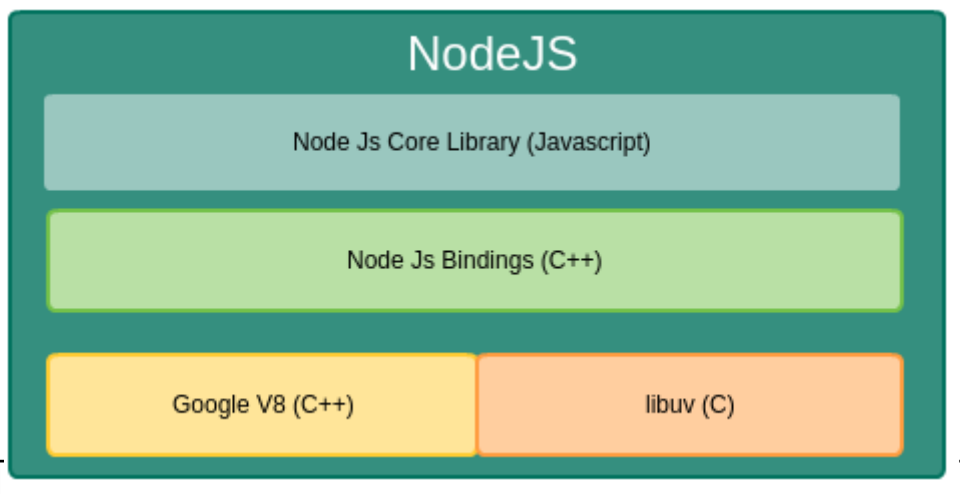
name:eventemitterintro task: << [índice de contenidos](#) >>

EDP: Intro

La **programación orientada a eventos** es un paradigma de programación en el cual el flujo del programa viene determinado por eventos tales como acciones de usuario (clicks del *mouse*, presión de teclas), salidas de sensores o mensajes de otros programas o hilos de ejecución.

Node usa un model orientado a eventos manejado por una librería llamada **libuv** que proporciona un mecanismo llamado **event loop**.

El comportamiento es muy parecido al del JavaScript del navegador



name:eventemitterintro task: << [índice de contenidos](#) >>

Event Loop

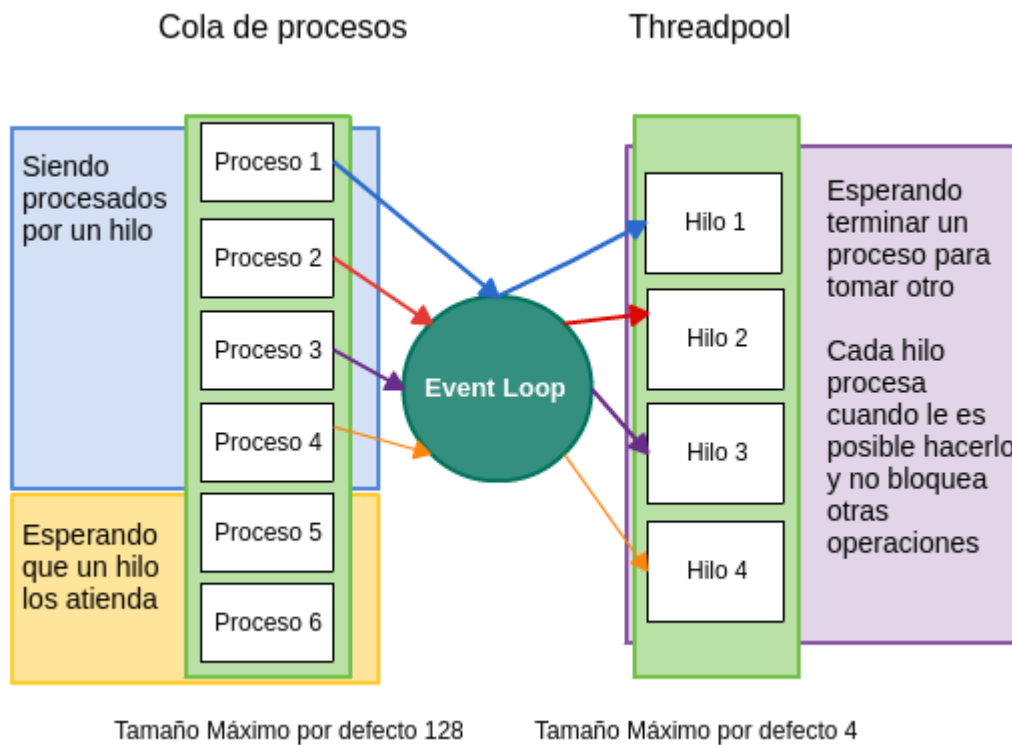
- Es un único subproceso que realiza todas las operaciones de entrada y salida (I/O) de forma asíncrona
- Es una cola de funciones. Cuando se ejecuta una función asíncrona, la función devuelve el código interno de la función, lo envuelve y se inserta en una cola
- El motor de JavaScript manda las operaciones a la cola y hace que se procesen en segundo plano para no bloquear las demás operaciones
- La librería **libuv**, proporciona la manera de añadir las operaciones necesarias a la cola de forma asíncrona (conocida como threadpool) y estas operaciones corren de forma nativa en el SO.

name:eventemitterintro task: << [índice de contenidos](#) >>

Event Loop: Proceso

El funcionamiento se da de la siguiente manera:

1. Existe una cola de tareas
2. Se define el número de hilos
3. Cada hilo toma una tarea y la ejecuta
4. Una vez que la tarea esté completa, se disparará otra
5. Si no hay tareas, el subproceso permanece inactivo
6. Si encuentra una nueva tarea en cola se inicia el procesamiento



Referencias:

[Entendiendo la magia detrás de node.js y su event loop](#)

Patrón Observer

- Define un objeto llamado `subject` que puede notificar a un grupo de observers (o listeners) cuando ocurre un cambio de estado
- La diferencia principal con el patron de callback es que el sujeto puede notificar a muchos observadores mientras que un callback es el único listener de la finalización de una tarea asíncrona.
- Las funciones que escuchan los eventos actúan como observadores. Cada vez que un evento se dispara, su función de escucha comienza a ejecutarse.
- Node.js tiene varios eventos incorporados disponibles a través del módulo de `events` y la clase `EventEmitter` que se utilizan para vincular eventos y `events listeners`.

Event Emitter

Crear un objeto `eventEmitter`:

```
//Módulo de eventos de importación

import {EventEmitter} from 'events';

const eventEmitter = new EventEmitter();
```

Enlazar un controlador de eventos con un evento:

```
//Vincular evento y controlador de eventos  
eventEmitter.on ('eventName', eventHandler);
```

Lanzar un evento programáticamente de la siguiente manera:

```
eventEmitter.emit('eventName');
```

Event Emitter: Ejemplo

```
import {EventEmitter} from 'events';  
  
const eventEmitter = new EventEmitter();  
  
//Create an event handler as follows  
  
const connectHandler = () => {  
    console.log('connection succesful.');
```

```
    // Fire the data_received event  
    eventEmitter.emit('data_received');
```

```
}  
  
// Bind the connection event with the handler  
  
eventEmitter.on ('connection', connectHandler);  
  
//Bind the data_received event with the anonymous function  
  
eventEmitter.on('data_received', ()=>{  
    console.log('data received succesfully .');
```

```
});  
  
// Fire the connection event  
  
eventEmitter.emit ('connection');
```

```
console.log("Program Ended");
```

Ejercicio 8

- Crear un objeto `__EventEmitter__` que emita la fecha actual (`Date.now()`) dentro de una función `setInterval()` cada medio segundo.

- Asociar un listener a este evento que imprima la fecha por consola

Realizar el ejercicio anterior con *callbacks*

Bonus track

Crear una función que acepte un **callback** y devuelva un **EventEmitter**

Ejercicio 9

Crear un objeto **event emitter** que emita un evento al que suscribir dos *listeners*:

- El primer listener imprimirá del 1 al 100 cada medio segundo
 - El segundo listener imprimirá del 100 al 1 cada segundo El programa terminará cuando los dos contadores impriman el mismo número.
-

name: buffers class: center, middle, inverse

[Buffers](#Qué es Node.js?)

name: buffersintro task: << índice de contenidos >>

Buffers

- Conjuntos de datos en crudo (*raw data*), datos binarios, que podemos tratar en NodeJS para realizar diversos tipos de acciones
 - Son instancias para almacenar datos sin procesar similares a una matriz de números enteros
 - En Node.js se implementan mediante una clase específica llamada **Buffer**
 - Se utilizan en la comunicación bidireccional que tenemos cuando manejamos sockets, pero también al manipular imágenes o streams de datos.
-

name: buffersintro task: << índice de contenidos >>

Crear un Buffer

- La clase **Buffer** es global en NodeJS, por lo que no necesita ser importada de ningún módulo para poder usarla.
- El tamaño de un buffer se establece en el momento de su creación y luego ya no es posible cambiarlo.
- Para crear un buffer:

```
let b1 = Buffer.alloc(20);
```

crea un buffer con tamaño de 20 bytes sin contenido. Cada uno de sus bytes estará inicializado a cero

- Para crear un buffer con contenido:

```
var b2 = Buffer.from('Ejemplo de buffer');
```

- En este caso la codificación por defecto es 'utf8'

name:bufferintro task: << [índice de contenidos](#) >>

Buffer: Escribir en una cadena

```
buffer.write(string[, offset][, length][, encoding])
```

Descripción de los parámetros utilizados:

- **String**: Esta es la cadena de datos que se escribirá en el buffer.
- **Offset**: Este es el índice del buffer para comenzar a escribir en un punto establecido, el valor predeterminado es 0.
- **Length**: Este es el número de bytes a escribir.
- **Encoding**: Codificación a utilizar. 'Utf8' es la codificación predeterminada.

Valor del retorno: Este método devuelve el número de octetos escritos. Si no hay suficiente espacio en el buffer para ajustar la cadena entera, escribirá solo una parte de la cadena (hasta donde alcance el espacio).

name:bufferintro task: << [índice de contenidos](#) >>

Buffer: Lectura

```
buf.toString([encoding][, start][, end])
```

Descripción de los parámetros utilizados:

- **Encoding**: Codificación a utilizar. 'Utf8' es la codificación predeterminada.
- **Start**: Índice inicial para empezar a leer, el valor predeterminado es 0.
- **End**: El índice final de la lectura, por defecto es el buffer completo.

Valor del retorno: Este método decodifica y devuelve una cadena de datos lo que esta escrito en el buffer.

name:bufferintro task: << [índice de contenidos](#) >>

Buffer: Concatenar buffers

```
Buffer.concat(list[, totalLength])
```

Descripción de los parametros utilizados:

- **list**: Lista array de objetos de los buffers a concatenar.
- **TotalLength**: Ésta es la longitud total de los buffers cuando están concatenados.

Ejercicio 10

Crea un buffer de '**Hola Mundo**' conviértelo a minúsculas y extrae el espacio. Deberá quedar:

```
//ho1amundo
```

name: streams class: center, middle, inverse

[Streams](#Qué es Node.js?)

name: streamsintro task: << [índice de contenidos](#) >>

Streams

Objetos que permiten leer datos de una Fuente o escribir datos en un destino en modo continuo:

Cuatro tipos de streams:

Readable – Streams de lectura

Writable – Streams de escritura

Duplex – Stream de lectura/escritura

Transform – A type of duplex stream where the output is computed based on input.

Cada tipo de **Stream** es una instancia de **EventEmitter** y emiten distintos *events* en diferentes instantes de tiempo. Los eventos más comunes son:

data – Evento de datos disponibles para lectura

end – Evento de no hay más datos de lectura

error – Error recibiendo o escribiendo datos

finish – Finalización de la tarea

name: streamsintro task: << [índice de contenidos](#) >>

Streams: Ejemplo de Lectura

- Crear un **stream** a partir de la lectura de un fichero utilizando el método **createReadStream()** del objeto **fs**:

```
import fs from 'fs'; const streamLectura = fs.createReadStream('./archivo-texto.txt');
```

- Asociar un evento "**data**" a un *listener* que se ejecutará cuando el dato se haya leído y se encuentre disponible, recibiendo como parámetro un buffer de datos

```
streamLectura.on('data', chunk => //chunk es un buffer de datos console.log(chunk instanceof Buffer));  
//escribe "true" por pantalla
```

- Leemos su contenido:

```
streamLectura.on('data', chunk =>{ console.log('He recibido ' + chunk.length + ' bytes de datos.');
```

```
console.log(chunk.toString()); });
```

name:streamsintro task: << [índice de contenidos](#) >>

Ejercicio 11

Leer el fichero **movieDetails.json** y mostrar el contenido por consola. Asociar listeners a cada uno de los eventos: data, end, error

name:streamsintro task: << [índice de contenidos](#) >>

process

El módulo **process** permite obtener información y modificar el proceso en curso de Node. Al contrario que la mayoría de los módulos, **process** es global y siempre está disponible.

process events

process es una instancia de EventEmitter.

- El evento **exit** proporciona un espacio de ejecución final antes de salir de Node. El *event loop* ya no corre después de un evento **exit**

```
process.on('exit', function () { setTimeout(function () { console.log('This will not run'); }, 100);  
console.log('Bye.');
```

Otros eventos interesantes: **uncaughtException**

name:streamsintro task: << [índice de contenidos](#) >>

process: Operating system input/output

process proporciona una serie de **streams** para interaccionar con el sistema operativo:

process.stdin Es un **stream** de lectura. Todo lo que escribimos por el terminal, se lee con **stdin**. Está siempre accesible aunque en modo pausa (Node puede escribir, pero nosotros no podemos leerlo). Para

poder utilizarlo es necesario invocar el método `resume()`

```
process.stdin.resume();
process.stdin.setEncoding('utf8');

process.stdin.on('data', function (chunk) {
  process.stdout.write('data: ' + chunk);
});

process.stdin.on('end', function () {
  process.stdout.write('end');
});
```

name:streamsintro task: << [índice de contenidos](#) >>

process: Operating system I/O

`process.stdout`

Es un stream de escritura con el que se puede generar salida del programa. En este caso la consola.

```
process.stdout.write(objetoBuffer);
```

- Mediante el método `write()` se escribe en un `stream` de escritura. Hay que enviarle un buffer y otra serie de parámetros opcionales como el tipo de codificación y una función *callback* a ejecutar cuando termine la operación de escritura.

```
process.stdin.setEncoding('utf8'); process.stdout.write('¿Cómo estás hoy? '); process.stdin.once('data',
function(res) { process.stdout.write('Has respondido: '); process.stdout.write(res);
process.stdin.pause(); })
```

name:streamsintro task: << [índice de contenidos](#) >>

process: Operating system I/O

`process.argv`

`argv` es un array que contiene argumentos de la línea de comandos. Comienza con el propio comando `node`:

```
node argv.js -t 3 -c "abc def" -erf      foo.js
[ 'node',
  '/Users/croucher/argv.js',
  '-t',
  '3',
  '-c',
```

```
'abc def',  
'-erf',  
'foo.js' ]
```

Ejercicio 12

- Reescribir el ejercicio anterior utilizando la propiedad `stdout` en lugar de `console.log`
 - Incorporar una petición mediante `stdin` para bajar el fichero.
 - Combinar la salida por consola con los métodos de la propiedad `process.stdin` para entrar información por consola.
 - Usar la interface `readline` para los objetos `stdin/stdout`
-

name:streamsintro task: << [índice de contenidos](#) >>

Streams: Ejemplo de Escritura

- Ejemplo con el método `createWriteStream()`

```
import fs from "fs"; const data = 'Simply Easy Learning'; // Create a writable stream const writerStream  
= fs.createWriteStream('output.txt'); // Write the data to stream with encoding to be utf8  
writerStream.write(data,'UTF8'); // Mark the end of file writerStream.end(); // Handle stream events --  
> finish, and error writerStream.on('finish', function() { console.log("Write completed."); });  
writerStream.on('error', function(err){ console.log(err.stack); }); console.log("Program Ended");
```

Ejercicio 13

- Leer de la consola un texto y guardarlo en un fichero
-

name:streamsintro task: << [índice de contenidos](#) >>

Pipes

Conexión de un stream de lectura con un stream de escritura:

```
const streamLectura = fs.createReadStream('./archivo-texto.txt');  
const streamEscritura = fs.createWriteStream('./otro-archivo.txt');
```

Para conectar los dos streams se usa:

```
streamLectura.pipe(streamEscritura);
```

Ejercicio 14

- Copiar de un fichero a otro (ejercicio de pipes)
-

name:streamsintro task: << [índice de contenidos](#) >>

Streams: Encadenando streams

- Mecanismo para conectar la salida de un stream a múltiples streams mediante piping.
- Ejemplo generar archivo comprimido:

```
import fs from 'fs'; import zlib from 'zlib'; // Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt') .pipe(zlib.createGzip()) .pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");
```

Ejercicio 14

Descomprimir un fichero .gz creado con el ejemplo anterior. Buscar la documentación de la api '[zlib](#)'

name: fs task: << [índice de contenidos](#) >>

Módulo FileSystem

En NodeJS todas las operaciones de acceso al sistema de archivos están englobadas dentro del módulo "fs" (File System):

```
import fs from 'fs';
```

Los métodos del módulo fs existen en las dos alternativas: síncrona y asíncrona. Ej:

```
import fs from 'fs';
```

```
// Asynchronous read fs.readFile('input.txt', (err, data) => { if (err) return console.error(err);
```

```
  console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
const data = fs.readFileSync('input.txt');
```

```
console.log("Synchronous read: " + data.toString());

console.log("Program Ended");
```

name: filesystem task: << [índice de contenidos](#) >>

FileSystem: Abrir un fichero

fs.open(path, flags[, mode], callback)

```
// Asynchronous - Opening File
console.log("Going to open file!");
fs.open('input.txt', 'r+', (err, fd) => {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
});
```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Sacar información de un fichero

fs.stat(path, callback)

```
console.log("Going to get file info!");
fs.stat('input.txt', (err, stats) => {
  if (err) {
    return console.error(err);
  }
  console.log(stats);
  console.log("Got file info successfully!");

  // Check file type
  console.log("isFile ? " + stats.isFile());
  console.log("isDirectory ? " + stats.isDirectory());
});
```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Escribir en un fichero

fs.writeFile(filename, data[, options], callback)


```

console.log("Going to write into existing file");
fs.writeFile('input.txt', 'Simply Easy Learning!', err => {
  if (err) {
    return console.error(err);
  }

  console.log("Data written successfully!");
  console.log("Let's read newly written data");
  fs.readFile('input.txt', (err, data) => {
    if (err) {
      return console.error(err);
    }
    console.log("Asynchronous read: " + data.toString());
  });
});

```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Leer de un fichero

fs.read(fd, buffer, offset, length, position, callback)

```

const buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', (err, fd) => {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, (err, bytes) => {
    if (err){
      console.log(err);
    }
    console.log(bytes + " bytes read");

    // Print only read bytes to avoid junk.
    if(bytes > 0){
      console.log(buf.slice(0, bytes).toString());
    }
  });
});

```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Cerrar un fichero

fs.close(fd, callback)

```
const buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', (err, fd) => {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");

  fs.read(fd, buf, 0, buf.length, 0, (err, bytes) => {
    if (err){
      console.log(err);
    }
    // Print only read bytes to avoid junk.
    if(bytes > 0){
      console.log(buf.slice(0, bytes).toString());
    }
    // Close the opened file.
    fs.close (fd, err => {
      if (err){
        console.log(err);
      }
      console.log("File closed successfully.");
    });
  });
});
```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Truncar un fichero**fs.ftruncate(fd, len, callback)**

```
const buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', (err, fd) => {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to truncate the file after 10 bytes");

  // Truncate the opened file.
  fs.ftruncate(fd, 10, err => {
    if (err){
```

SIGUE>>>>>>>>>>

FileSystem: Truncar un fichero

FileSystem: Borrar un fichero

51 / 80

```
    console.log("File deleted successfully!");  
  });
```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Crear un directorio

fs.mkdir(path[, mode], callback)

```
console.log("Going to create directory /tmp/test");  
  
fs.mkdir('/tmp/test', err => {  
  if (err) {  
    return console.error(err);  
  }  
  console.log("Directory created successfully!");  
});
```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Leer un directorio

fs.readdir(path, callback)

```
console.log("Going to read directory /tmp");  
fs.readdir("/tmp/", (err, files) => {  
  if (err) {  
    return console.error(err);  
  }  
  files.forEach( file => {  
    console.log( file );  
  });  
});
```

name:filesystem task: << [índice de contenidos](#) >>

FileSystem: Eliminar un directorio

fs.rmdir(path, callback)

```
console.log("Going to delete directory /tmp/test");  
  
fs.rmdir("/tmp/test", err =>{  
  if (err) {
```

```
        return console.error(err);
    }
    console.log("Going to read directory /tmp");

    fs.readdir("/tmp/", (err, files) => {
        if (err) {
            return console.error(err);
        }
        files.forEach( file => {
            console.log( file );
        });
    });
});
```

Ejercicio a entregar

Crear un módulo para empaquetar los ejercicios de node sin incluir los directorios node_modules.

name: tcp class: center, middle, inverse

[TCP y SOCKET](#Qué es Node.js?)

name:tcpintro task: << [índice de contenidos](#) >>

TCP y SOCKET

[Transmission Control Protocol](#), es el protocolo de comunicación más importante de los Internet protocol (IP). Proporciona un mecanismo de transporte a la capa de aplicaciones. En TCP se basan otros protocolos como: HTTP, iRC, SMTP e IMAP.

Node implementa un **servidor HTTP** en forma de pseudo-class en [http.Server](#), que proviene de la pseudo-clase [net.Server](#) del servidor TCP.

task: << [índice de contenidos](#) >>

TCP: Crear un servidor

Importamos el módulo [Net](#):

```
import net from 'net'
const server = net.createServer(socket => {
    console.log('new connection established');
    socket.on('data', data => {
        console.log('got data');
    });
});
```

```
socket.on('end', function(data) {
  console.log('connection ended');
});

socket.write('Bienvenido. Escribe algo');
})
server.listen(4001);
```

- El método `createServer()` del package `Net` se vincula al puerto TCP 4001.
- Al método `createServer()` se pasa una función de *callback* que se invoca cada vez que se produce un evento **'connection'**.
- Dentro del *callback* se maneja el objeto **Socket**, que se puede utilizar para enviar y recibir datos a y desde un cliente.

name:tcpintro task: << [índice de contenidos](#) >>

TCP: Ciclo de vida de un servidor

- El objeto `server` es un `EventEmitter`, que emite los siguientes eventos:
 - **"listening"**: Cuando el servidor escucha en el Puerto y dirección especificados
 - **"connection"**: Cuando se establece una nueva conexión. El *callback* recibe el objeto `socket`
 - **"close"**: Cuando se cierra el servidor, termina el enlace al puerto.
 - **"error"** — Cuando ocurre un error a nivel de servidor. Ej: Puerto ocupado o no hay permisos para hacer el binding al puerto

name:tcpintro task: << [índice de contenidos](#) >>

TCP: Ciclo de vida de un servidor

```
import net from 'net'

const server = net.createServer();
const port = 4001;

server.on('listening', () => {
  console.log('Server is listening on port', port);
});

server.on('connection', socket => {
  console.log('Server has a new connection');
  socket.end();
  server.close();
});

server.on('close', () => {
  console.log('Server is now closed');
});
```

```
server.on('error', err => {
  console.log('Error occurred:', err.message);
});

server.listen(port);
```

El **Ciente** para conectarse al servidor: `telnet localhost 4001` ó `nc localhost 4001`

Para salir del cliente: `ctrl++`

task: << [índice de contenidos](#) >>

SOCKET

Es el primer argumento de la función *callback* que devuelve el **evento** “**connection**”

El objeto `socket` es un *stream duplex* (lectura y escritura). Emite el evento “**data**” cuando devuelve un paquete de datos y el evento “**end**” cuando se cierra la conexión.

También es un stream de escritura, por lo que puede escribir buffers o strings mediante el método `socket.write()`.

Otros métodos accesibles, por ser stream son: `socket.pause()`, `socket.resume()`, o incluso se puede hacer pipe a cualquier stream de escritura.

Para otras características mirar la documentación de [node.js](#)

task: << [índice de contenidos](#) >>

Ejemplo de server tcp

```
import net from 'net'

const server = net.createServer();
const port = 4001;

const listener = socket =>{
  socket.setEncoding('utf8');
  socket.write("Hello! You can start typing. Type 'quit' to exit.\n");
  socket.on('data', function(data) {
    console.log('got:', data.toString())
    if (data.trim().toLowerCase() === 'quit') {
      socket.write('Bye bye!');
      return socket.end();
    }
    socket.write(data);
  });
  socket.on('end', function() {
    console.log('Client connection ended');
  })
}
```

```
server.on('connection', listener);  
server.listen(port);
```

task: << [índice de contenidos](#) >>

CHAT SERVER

Instanciar el servidor, bindear eventos importantes (error, close) y unirlo al puerto 4001:

```
import net from 'net';  
const server = net.createServer();  
server.on('error', err => {  
  console.log('Server error:', err.message);  
});  
server.on('close', () => console.log('Server closed'));  
  
server.listen(4001);
```

Aceptar peticiones de clientes (evento connection):

```
server.on('connection', socket => {  
  console.log('got a new connection');  
});
```

task: << [índice de contenidos](#) >>

CHAT SERVER

Leer datos de la conexión

```
server.on('connection', socket => {  
  console.log('got a new connection');  
  socket.on('data', function(data) {  
    console.log('got data:', data);  
  });  
});
```

Recopilar todos los clientes para transmitirles los datos de un cliente. Para ello hay que almacenar todas las conexiones en un repositorio

```
let sockets = [];  
server.on('connection', socket => {
```



```
console.log('got a new connection');  
sockets.push(socket);
```

task: << [índice de contenidos](#) >>

CHAT SERVER

Transmitir los datos a todos los clientes

```
socket.on('data', data => {  
  console.log('got data:', data);  
  sockets.forEach(otherSocket => {  
    if (otherSocket !== socket) {  
      otherSocket.write(data);  
    }  
  });  
});
```

Ejercicio 14

Sobre el servidor de chat realizar las siguientes funciones:

- Registrar la eliminación de una conexión cerrada
- Pedir un nombre a cada cliente y registrarlo asociado a la conexión. Si el nombre ya existe pedir otro nuevo.
- Cuando se conecte un cliente nuevo comunicarlo al resto de clientes, así como el número de clientes registrados.
- Imprimir el prompt de otro cliente, delante del texto.

name: http class: center, middle, inverse

[HTTP](#Qué es Node.js?)

name:httpintro task: << [índice de contenidos](#) >>

HTTP

Hypertext Transfer Protocol, o **HTTP**, es un protocolo a nivel de aplicación que permite la transmisión de contenido. Es el fundamento de la World Wide Web.

HTTP es un protocolo basado en texto que trabaja sobre la capa de TCP.

La versión encriptada de HTTP se denomina HTTP Secure, o **HTTPS**

HTTP es un protocolo de petición-respuesta desarrollado sobre las bases de la programación cliente-servidor. Normalmente, un navegador se utiliza como cliente en una transacción HTTP

Cuando se introduce una URL en un navegador, se realiza una **petición HTTP** al servidor que hospeda la URL, suele hacerse a través del puerto 80 (o 443 si es HTTPS). El servidor procesa la petición y responde al cliente

name:httpintro task: << [índice de contenidos](#) >>

Servidor HTTP en Node.JS

```
import http from 'http';

const server = http.createServer();

server.on('request', function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
});

server.listen(4000);
```

- Cuando un cliente hace una petición, el servidor HTTP emite un evento de **request** pasando un objeto **request HTTP** y uno **response HTTP**. El objeto **request** HTTP permite consultar propiedades de la petición, mientras que el objeto **response** HTTP permite construir la respuesta que se enviará al cliente

name:httpintro task: << [índice de contenidos](#) >>

HTTP: El objeto HTTPServerRequest

Contiene las siguientes propiedades:

- **req.url**: Contiene el string de la URL demandada desde el cliente
- **req.method**: Contiene el método HTTP usado en la petición: GET, POST, DELETE o HEAD
- **req.headers**: Contiene las propiedades de la cabecera de la petición
- **res.end(util.inspect(req.headers))**;

Cuando se realiza una petición a un servidor el **body** de la petición no se recibe inmediatamente. Pero se puede "escuchar" un evento **data** que maneje la información cuando llegue. El objeto request es un stream de lectura

```
const writeStream = ...
require('http').createServer(function(req, res) {
  req.on('data', function(data) {
```

```
writeStream.write(data);
});
}).listen(4001);
```

name:httpintro task: << [índice de contenidos](#) >>

HTTP: El objeto HTTPServerResponse

En la respuesta se puede escribir: **Head** y **Body**. Para escribir la cabecera usar: `res.writeHead(status, headers)`

```
require('http').createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/plain',
    'Cache-Control': 'max-age=3600' });
  res.end('Hello World!');
}).listen(4000);
```

Las cabeceras se pueden modificar mientras no se haya enviado el body de la respuesta (`res.write` o `res.end`):

```
res.setHeader(status, headers)
```

```
res.removeHeader('Cache-Control');
```

El body se escribe directamente: `res.write('Hello');` o utilizando un buffer:

```
let buffer = new Buffer('Hello World');
res.write(buffer);
```

name:httpintro task: << [índice de contenidos](#) >>

Respuesta troceada STREAMING HTTP

HTTP chunked encoding permite al servidor mantener el envío de datos al cliente sin tener que enviar el tamaño del cuerpo de la respuesta (body size). Si no se ha especificado la propiedad Content-Length del header, por defecto, El servidor Node HTTP envía el siguiente header al cliente:

```
.center[Transfer-Encoding: chunked]
```

Esta cabecera permite al cliente recibir los datos de forma troceada, enviando un ultimo trozo de longitud 0 antes de que el cliente termine de procesar la respuesta. Esto tiene utilidad para el envío de **streaming de texto, audio, o vídeo**

Algunos ejemplos de streaming usando estas características son el piping de un fichero al stream

Ejercicio 15

Crear un servidor http Crear un stream sobre un vídeo '.mp4' Fijar la propiedad Content-Type de la cabecera de la respuesta Hacer un pipe a la respuesta HTTP Abrir el navegador y comprobar que el vídeo empieza a visualizarse sin haberse cargado completamente.

Ejercicio 16

Crear un servidor HTTP que devuelva texto plano con 100 líneas separadas de timestamp cada segundo, durante 10 segundos.

name:httpintro task: << [índice de contenidos](#) >>

CLIENTE HTTP

HTTP ocupa el centro de muchas infraestructuras no sólo como servidor de contenido estático sino también como servidor de información a través de llamadas a una API pública

Una de las fortalezas de Node es el manejo de las E/S, esto lo convierten en una buena herramienta para proveer información y manejar los servicios HTTP.

El protocolo HTTP presenta dos propiedades: la URL y el método.

El método GET es el método principal más usado, otros métodos como POST (principal método para enviar formularios web), PUT, DELETE, and HEAD.

name:httpintro task: << [índice de contenidos](#) >>

Métodos o Verbos en una petición

Request method	Use	Response
POST	Create new data in the database	New data object as seen in the database
GET	Read data from the database	Data object answering the request
PUT	Update a document in the database	Updated data object as seen in the database
DELETE	Delete an object from the database	Null

El método es importante, porque una API REST bien diseñada debería utilizar la misma URL para diferentes acciones. En estos casos el método le dice al servidor el tipo de operación a realizar.

name:httpintro task: << [índice de contenidos](#) >>

Códigos de respuesta (STATUS_CODE)

Una API REST bien construida debe devolver el código de respuesta correcto

Status code	Name	Use case
200	OK	A successful GET or PUT request
201	Created	A successful POST request
204	No content	A successful DELETE request
400	Bad request	An unsuccessful GET, POST, or PUT request due to invalid content
401	Unauthorized	Requesting a restricted URL with incorrect credentials
403	Forbidden	Making a request that isn't allowed
404	Not found	Unsuccessful request due to an incorrect parameter in the URL
405	Method not allowed	Request method not allowed for the given URL
409	Conflict	Unsuccessful POST request when another object with the same data already exists
500	Internal server error	Problem with your server or the database server

name:httpintro task: << [índice de contenidos](#) >>

Crear peticiones HTTP

Además de crear servidores, el modulo http permite crear peticiones mediante el método **request()** El método request() toma dos argumentos: **options** y **callback**.

- **options** se usa para parametrizar la petición HTTP
- **callback** función que es invocada cuando se recibe una respuesta a la petición
- **IncomingMessage** es el único argumento que se le pasa al callback

request() También devuelve una instancia de **http.ClientRequest**, que es un stream de escritura.

name:httpintro task: << [índice de contenidos](#) >>

Ejemplo de Petición GET HTTP (Cliente HTTP)

```
import http from "http";

const request = http.request({
  hostname: "localhost",
  port: 8000,
  path: "/",
  method: "GET",
```

```
    headers: {"Host": "localhost:8000"    }
  },
  response => {
    let statusCode = response.statusCode;
    let headers = response.headers;
    let statusLine = "HTTP/" + response.httpVersion + " " + statusCode + " "
+ http.STATUS_CODES[statusCode];

    console.log(statusLine);
    for (header in headers) {
      console.log(header + ": " + headers[header]);
    }
    console.log();
    response.setEncoding("utf8");
    response.on("data", data => {
      process.stdout.write(data);
    });
    response.on("end", function() {
      console.log();
    });});

request.end();
```

name:httpintro task: << [índice de contenidos](#) >>

Ejemplo de Petición HTTP

Versión reducida, no se puede fijar la cabecera:

```
import...
const request = http.request("http://localhost:8000/", response => {
  response.setEncoding("utf8");
  response.on("data", function(data) {
    process.stdout.write(data);
  });
  response.on("end", function() {
    console.log();
  });
});

request.end();
```

name:httpintro task: << [índice de contenidos](#) >>

Ejemplo de Petición POST HTTP (Cliente HTTP)

```
import http from "http";
import qs from "querystring";

const body = qs.stringify({
  foo: "bar",
  baz: [1, 2]
});

const request = http.request({
  hostname: "localhost",
  port: 8000,
  path: "/",
  method: "POST",
  headers: {
    "Host": "localhost:8000",
    "Content-Type": "application/x-www-form-urlencoded",
    "Content-Length": Buffer.byteLength(body)
  }
},
response => {
  response.setEncoding("utf8");
  response.on("data", data => {
    process.stdout.write(data);
  });

  response.on("end", () => console.log());
});
request.end(body);
```

name:httpintro task: << [índice de contenidos](#) >>

Ejemplo de Servidor que procesa POST HTTP (Cliente HTTP)

```
import http from "http";
import qs from "querystring";

const server = http.createServer((request, response) => {

  let bodyString = "";
  request.setEncoding("utf8");

  request.on("data", data => {

    bodyString += data;
  });

  request.on("end", () => {
```

```
    let body = qs.parse(bodyString);

    for (let b in body) {

        response.write(b + ' = ' + body[b] + "\n");
    }
    response.end();
  });
});

server.listen(8000);
```

name:httpintro task: << [índice de contenidos](#) >>

Middlewares

Bloque de código que se ejecuta entre la petición que hace el usuario (**request**) hasta que la petición llega al servidor.

Una parte del middleware recibe una petición entrante y la puede procesar completamente o pasarla a otra parte del middleware para un procesamiento adicional antes de enviar la respuesta al cliente.

Las funciones de middleware, por tanto, tienen **acceso al objeto de solicitud (req)**, **al objeto de respuesta (res)** y a la **siguiente función de middleware** en el ciclo de solicitud/respuestas de la aplicación. La siguiente función de middleware se denota normalmente con una variable denominada **next**. Ej:

```
function middleware(request, response, next) {
  return next();
}
```

name:httpintro task: << [índice de contenidos](#) >>

Middlewares

Las funciones de middleware pueden realizar las siguientes tareas:

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

Si la función de middleware actual no finaliza el ciclo de solicitud/respuestas, debe invocar **next()** para pasar el control a la siguiente función de middleware. De lo contrario, la solicitud quedará colgada.

name:httpintro task: << [índice de contenidos](#) >>

EJEMPLO SERVIDOR CON MIDDLEWARE

```
import http from 'http';
// Instalamos e importamos el módulo connect
import connect from 'connect';

const app = connect();
let isLogin = false;

app.use((req,res,next) => {
  if(req.url.indexOf("favicon.ico")>0){
    return;
  }
  next()
})
app.use((req,res,next)=>{//Función middleware que evalúa si estamos
conectados
  if(isLogin) return next();
  else{
    console.log('No estás logado');
    res.end('No estas logado')}
  })
app.use((request,response,next)=>{//Función middleware que genere la
respuesta

  response.setHeader('Content-Type','text/html' );
  response.end('Hello <strong>HTTP</strong>!');
  })

app.listen(4000);

http.createServer(app);
```

Ejercicio 17

- Rehacer el servidor de la petición post con la capa de middleware:
 - Utilizar el package:

```
import bodyParser from 'body-parser'; app.use(bodyParser.urlencoded({extended: false}));
```

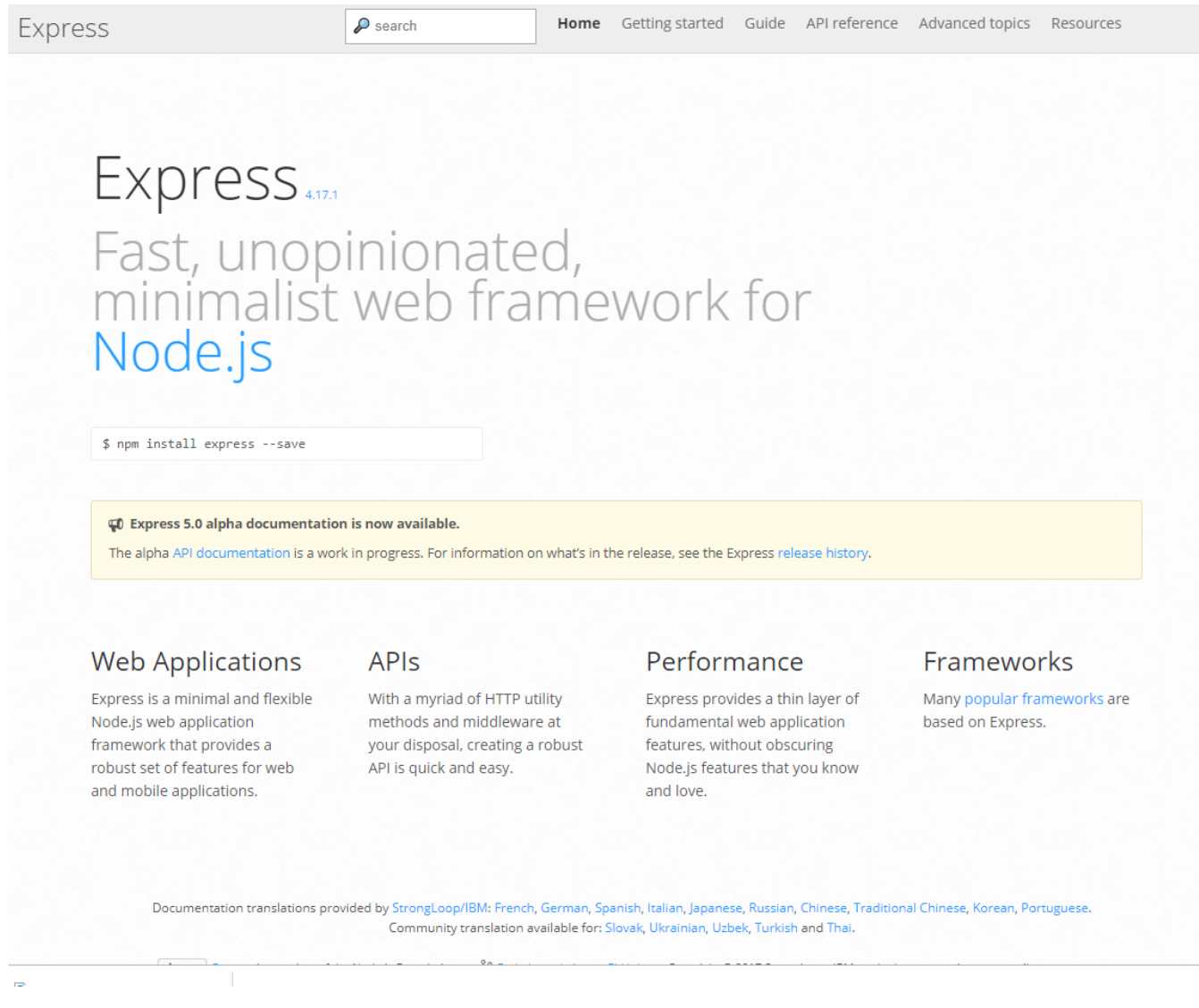
name: express class: center, middle, inverse

[EXPRESS](#Qué es Node.js?)

name:expressintro task: << índice de contenidos >>

MIDDLEWARES CON EXPRESS

.center[

The screenshot shows the Express.js website homepage. At the top, there's a navigation bar with the 'Express' logo on the left and links for 'Home', 'Getting started', 'Guide', 'API reference', 'Advanced topics', and 'Resources' on the right. A search bar is also present. The main heading is 'Express' followed by the version '4.17.1'. Below this, it says 'Fast, unopinionated, minimalist web framework for Node.js'. A code block shows the command '\$ npm install express --save'. A yellow banner announces 'Express 5.0 alpha documentation is now available.' with a link to the 'release history'. The page is divided into four columns: 'Web Applications', 'APIs', 'Performance', and 'Frameworks', each with a brief description of Express's capabilities. At the bottom, it mentions documentation translations provided by StrongLoop/IBM in various languages and community translations available for other languages. The page is enclosed in a container with a right square bracket ']'.

name:expressintro task: << índice de contenidos >>

MIDDLEWARES CON EXPRESS

- Creamos un proyecto con archivo `app.js` y generamos `package.json`
- Instalamos express:

.inverse[

```
npm install express --save
```

]

- De la página oficial bajamos el ejemplo ["hola mundo"]([https 🙄/expressjs.com/es/starter/hello world.html](https://expressjs.com/es/starter/hello world.html)):

```
import express from 'express'
const app = express();

app.get('/', (req, res) => res.send('Hello World!'));

app.listen(3000, ()=>
  console.log('Example app listening on port 3000!'));
```

Ejercicio 18

- Insertamos el resto de operaciones http: **post**, **put**, **delete**
- Probamos con Postman
- Incorporamos dos métodos de middleware que evalúen:
 - Si estamos logados
 - Se guarde una traza de la url desde donde se conecta el cliente y el método http

Ejercicio 19

Incorporar la fecha actual a la respuesta a la petición mediante un middleware.

name:expressintro task: << [índice de contenidos](#) >>

Middlewares de terceros

Middleware	Descripción
body-parser	Parsea el cuerpo de la petición entrante: req.body property.
compression	Comprime el cuerpo de la respuesta HTTP
cookie-parser	Parsea la cabecera "Cookie" y genera la propiedad req.cookies. Permite manejar cookies firmadas
cookie-session	Almacena todos los datos de sesión de cliente en una cookie
express-debug	Añade a la aplicación un separador con información sobre las variables de plantilla(locals), la sesión actual, datos de solicitud útiles, etc.

Middleware	Descripción
express-session	Almacena el id de session en la cookie y el resto de información en el servidor
express-simple-cdn	Utiliza una CDN (Red de entrega de contenido) para los activos estáticos, con soporte de varios hosts (por ejemplo, cdn1.host.com, cdn2.host.com)
helmet	Protege las aplicaciones estableciendo varias cabeceras HTTP.
method-override	Anula las cabeceras que no permiten ciertos métodos HTTP (PUT, DELETE)
morgan: anteriormente logger	Genera un log de la aplicación
Passport	Autentica la aplicación mediante diferentes estrategias
serve-static:	Sirve contenido estático

name: apirest class: center, middle, inverse

[API RESTful](#Qué es Node.js?)

name: apirest task: << [índice de contenidos](#) >>

API RESTful

REST no es un protocolo, sino un **conjunto de reglas y principios** que permiten desarrollar servicios web utilizando **HTTP como protocolo** de comunicaciones entre el cliente y el servicio web.

Se basa en definir acciones sobre recursos mediante el uso de los métodos **GET, POST, PUT y DELETE** inherentes a HTTP.

Una API REST es un interfaz sin estado (*stateless*) de una aplicación. En el caso de MEAN stack, la API REST se usa para crear un interfaz sin estado de acceso a la base de datos

name: apirest2 task: << [índice de contenidos](#) >>

API RESTful

REST se basa en el uso de estándares abiertos:

- **URI** para la localización de recursos,
- **HTTP** como protocolo de transporte,
- Los **verbos HTTP** para especificar las acciones sobre los recursos y

- Los **tipos MIME** (Multipurpose Internet Mail Extensions) para la representación de los recursos (XML, JSON, XHTML, HTML, PDF, GIF, JPG, PNG, etc.).

name:apirest3 task: << [índice de contenidos](#) >>

API RESTful: URLs

Las URLs de las peticiones a una API REST siguen un formato estándar. Este formato permite seleccionar, utilizar y mantener de forma fácil nuestra API.

La mejor manera de representar las URLs de la API es pensando en las colecciones de la base de datos. Normalmente, se dispondrá de un conjunto de URLs por cada colección. Se pueden tener, además, un conjunto de URLs para cada conjunto de subdocumentos. Cada URL deberá tener, dentro del conjunto, la misma ruta básica (*path*), y parámetros adicionales.

Por cada conjunto de URLs, se requieren cubrir distintas acciones, basadas en las operaciones CRUD:

1. Alguien o algo realiza una petición a la API.
2. La API procesa la petición, requiriendo a la base de datos si fuese necesario.
3. The API **siempre** envía una respuesta al peticionario.

name:apirest4 task: << [índice de contenidos](#) >>

API RESTful: VERBOS (Métodos)

A REST API recibe una petición HTTP, realiza algún procesamiento y devuelve una respuesta HTTP:

- Crea un nuevo ítem
- Lee una lista de varios ítems
- Lee un ítem específico
- Actualiza un ítem específico
- Elimina un ítem específico

Los cuatro verbos más comunes son: POST, GET, PUT, and DELETE. Una API REST bien diseñada tendrá la misma URL para distintas acciones. En estos casos, el método indica al servidor la operación que tiene que realizar

Un plan para una API, quedaría:

Acción	Método	URL ruta	Ejemplo
Create new location	POST	/locations	http://loc8r.com/api/locations
Read list of locations	GET	/locations	http://loc8r.com/api/locations
Read a specific location	GET	/locations/:locationid	http://loc8r.com/api/locations/123
Update a specific location	PUT	/locations/:locationid	http://loc8r.com/api/locations/123
Delete a specific location	DELETE	/locations/:locationid	http://loc8r.com/api/locations/123

name:apirest5 task: << índice de contenidos >>

API RESTful: Formato de la respuesta

La API debería devolver un formato consistente de datos. Los formatos típicos de una API REST son XML y/o JSON. En el caso de MEAN stack utilizaremos JSON, que encaja de forma natural con el ecosistema JavaScript. JSON es también más compacto que XML lo que ayuda a acelerar los tiempos de respuesta y la eficiencia al disminuir el ancho de banda requerido.

La API devolverá una de las siguientes respuestas por cada petición:

- Un objeto JSON con los datos de respuesta a la consulta
- Un objeto JSON con datos de error
- Una respuesta nula

El tipo de respuesta debería especificarse con el parámetro **content-type** dentro del **header**:

```
{Content-type: application/json}
```

name: produccion class: center, middle, inverse

[Producción](#Qué es Node.js?)

name:prodcontenido task: << índice de contenidos >>

A tener en cuenta

- Mejorar el rendimiento
 - Configurar la aplicación
 - Subir a la nube: PaaS heroku
-

name:rendimiento task: << índice de contenidos >>

Mejora del rendimiento

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/deployment

- Utilizar la compresión de gzip
 - La compresión de gzip puede disminuir significativamente el tamaño del cuerpo de respuesta y, por lo tanto, aumentar la velocidad de una aplicación web. Utilizar el middleware de **compresión** para utilizar la compresión de gzip en Express.js
 - Si se usa un proxy inverso (Nginx) no es necesario el middleware
- No utilizar funciones síncronas
 - Ralentizan las llamadas. Utilizar `--trace-sync` en desarrollo para detectar las funciones síncronas

- Utilizar un middleware para el servicio de archivos estáticos (`serve-static`) en lugar de `res.sendFile()`
 - Realizar un registro correcto
 - `console.log` y `console.err` son funciones síncronas. Cambiar por un módulo de depuración (debug)
 - Manejar todas las excepciones correctamente. Utilizar:
 - try-catch
 - promesas
-

name:confiprod task: << [índice de contenidos](#) >>

Configuración de la aplicación

- Configurar variables de entorno
- Establecer `NODE_ENV` en "production"
- Asegurarse de que la aplicación se reinicia automáticamente

Configuración del servidor

- Ejecutar la aplicación en un clúster
 - Almacenar en la caché los resultados de la solicitud
 - Utilizar un equilibrador de carga
 - Utilizar un proxy inverso
-

name:confiprod task: << [índice de contenidos](#) >>

Variables de entorno

- Antes del despliegue a producción, hay que preparar el código de la aplicación
 - Proteger la información privada dentro de variables de entorno:
 - Claves de API,
 - Passwords Admin
 - Rutas de db
 - Puerto HTTP
 - Indicar la base de datos de Desarrollo, Pruebas, Integración o Producción
 - URLs de recursos del servidor
 - CDNs para testing vs. production...
-

name:confiprod task: << [índice de contenidos](#) >>

Variables de entorno: `process.env`

Variable global inyectada por Node en tiempo de ejecución Se accede a través del objeto process, la propiedad env y la variable que queremos invocar. Ej.:

`process.env.PORT`

Cuando no están definidos los valores de las variables se pueden definir como:

```
const port = process.env.PORT || 3000; process.env.PORT = process.env.PORT || 3000;
```

Definición de las variables en tiempo de ejecución:

```
$ PORT = 3000 node server
```

name:confiproduct task: << índice de contenidos >>

Variables de entorno: Almacenamiento

En fichero `config.js`:

```
//config.js
module.export = {
  port: process.env.PORT || 3001,
  db: process.env.MONGODB_URI || "mongodb:http://localhost:27017/test",
  SECRET_TOKEN: "mi clave de token"
}
```

En fichero `.env`

```
NODE_ENV=development
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=myspassword DB_NAME=desarrolloactivo_db
```

- Incluir en fichero `.gitignore`:

```
.inverse[npm i dotenv]
```

```
require('dotenv').config();
```

name:confiproduct task: << índice de contenidos >>

```
NODE_ENV = 'production'
```

- Almacena en la caché las plantillas de vistas
- Almacena en la caché los archivos CSS generados en las extensiones CSS
- Genera menos mensajes de error detallados

name:confiprod task: << [índice de contenidos](#) >>

Reinicio Automático

- Utilizar un **gestor de procesos**: contenedor de aplicaciones que facilita el despliegue, proporciona una alta disponibilidad y permite gestionar la aplicación en el tiempo de ejecución.
- Además de reiniciar la aplicación cuando se bloquea, un gestor de procesos permite:
 - Obtener información útil sobre el rendimiento en tiempo de ejecución y el consumo de recursos.
 - Modificar dinámicamente los valores para mejorar el rendimiento.
 - Controlar la agrupación en clúster (StrongLoop PM y pm2).
- Los gestores de procesos más conocidos para Node son los siguientes:
 - [StrongLoop Process Manager](#)
 - [PM2](#)
 - [Forever](#)

name:confiprod task: << [índice de contenidos](#) >>

Gestor de procesos: `forever.js`

- Herramienta de interfaz de línea de mandatos simple que permite garantizar la ejecución continua (forever/siempre) de un determinado script
- Ideal para ejecutar los despliegues más pequeños de scripts y aplicaciones Node.js:

```
.inverse[npm install forever -g]
```

- Para iniciar un script:

```
.inverse[forever start script.js]
```

name:heroku task: << [índice de contenidos](#) >>

Heroku: Instalación

- Heroku es un servicio en la nube tipo PaaS (plataforma como servicio) donde se pueden **alojar y desplegar aplicaciones**
- Pasos para subir una aplicación a Heroku:
 1. Crear una cuenta en <https://signup.heroku.com/>
 2. Crear una nueva aplicación

Install the Heroku CLI

Download and install the Heroku CLI.

3. Clicar en **CLI:**

para instalar el **cliente Heroku**

4. Comprobar la versión instalada en consola: `heroku -v`

name:herokusubir task: << índice de contenidos >>

Heroku: Subir aplicación

5. Logarse desde el terminal: `.inverse[heroku login]`

- Crear un repositorio .git local (si no estaba creado): `.inverse[git init]`
- Crear un repositorio remoto git desde la ruta de la aplicación apuntando a la aplicación creada en heroku:
`.inverse[heroku git:remote -a api-rest-rgl]`
- Subimos la aplicación al repositorio:

8.1. Añadir código al repositorio

`.inverse[git add .]`

8.2. Hacer commit de todo

`.inverse[git commit -m 'version ok']`

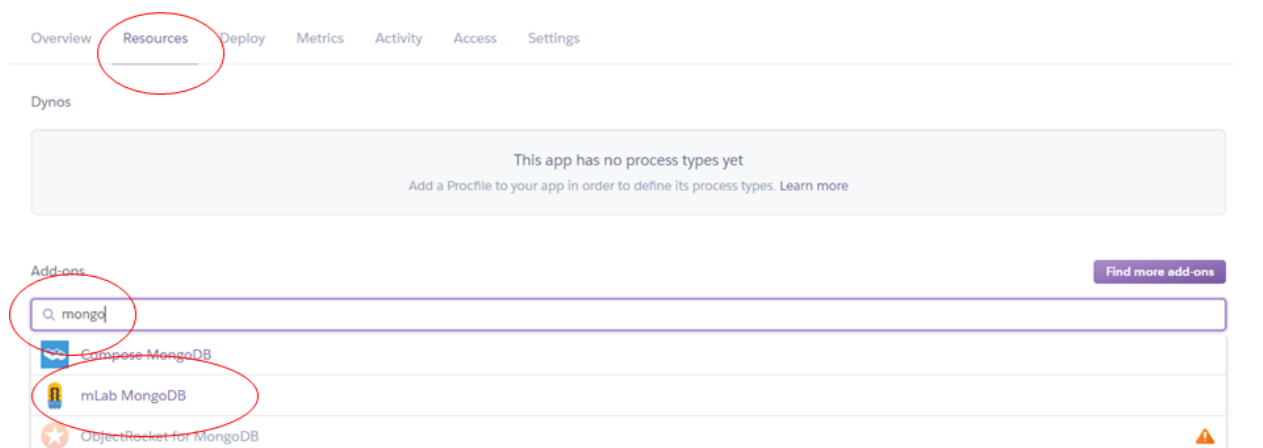
8.3. Subir el código a heroku

`.inverse[git push heroku master]`

name:herokusubir task: << índice de contenidos >>

Heroku: Configurar aplicación

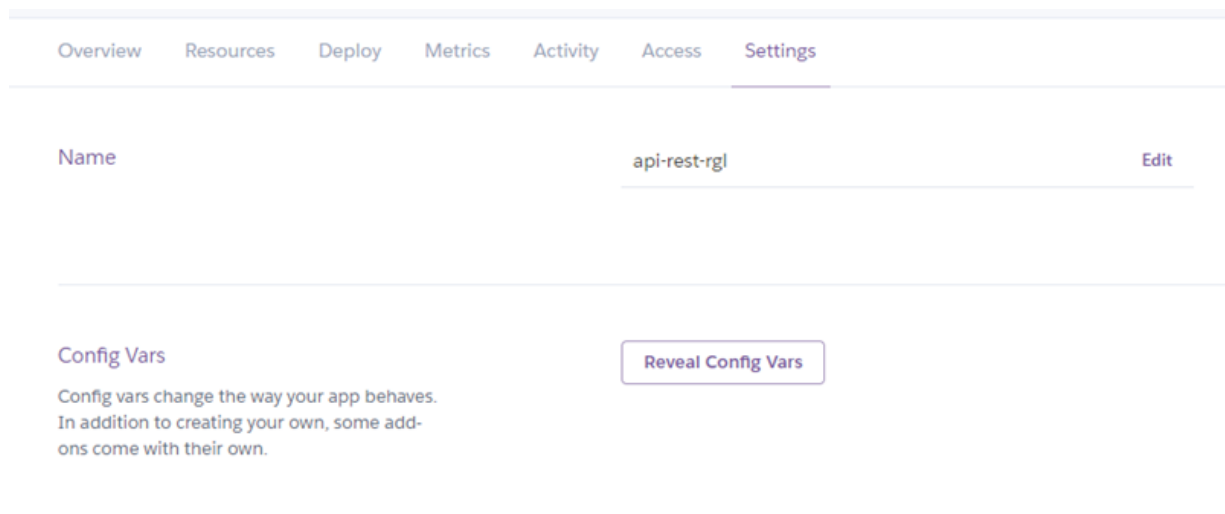
9. Añadir MongoDB:



name:herokusubir task: << [índice de contenidos](#) >>

Heroku: Subir aplicación

10. Configurar variables de entorno en Heroku:



name: socket class: center, middle



name: sockettitle class: center, middle, inverse

[socket.io](#Qué es Node.js?)

name:push task: << [índice de contenidos](#) >>

La tecnología PUSH

- Forma de comunicación a través de internet en la que la petición de envío tiene origen en el servidor
- Se busca la manera de asegurar la actualización en tiempo real de las aplicaciones clientes
- Alternativas:
 - **AJAX** - request → response. Crea una conexión al servidor, envía una cabecera de petición y recibe una respuesta del servidor. Después Cierra la conexión. En Node.js se puede implementar con [axios](#)

- **Long poll** - request → wait → response. Crea una conexión al servidor como AJAX, pero la mantiene viva durante un tiempo. Durante la conexión el cliente recibe datos del servidor. El cliente se va reconectando periódicamente. En la parte del servidor la petición es HTTP, aunque la respuesta puede ser en el momento o postpuesta según la lógica de la aplicación. En Node.js se puede implementar con [pollymer](#)
-

name:websocket task: << [índice de contenidos](#) >>

WebSocket

Tecnología que proporciona un canal de comunicación **persistente**, **bidireccional** y **full-duplex** sobre un único socket TCP

Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse en cualquier tipo de aplicación cliente/servidor

Proporciona una solución al bloqueo de puertos diferentes del 80, proveyendo una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP (a costa de una pequeña sobrecarga del protocolo)

[WebSocket](#) del lado del cliente se puede implementar en HTML5 en Mozilla Firefox > 8, Google Chrome > 4 y Safari > 5, así como la versión móvil de Safari en el iOS 4.2.1.

En Node lo podemos implementar con [WebSocket-Node](#)

name:socketiointro task: << [índice de contenidos](#) >>

[socket.io](#)

- Facilita la comunicación en tiempo real, bidireccional y basada en eventos entre el navegador y el servidor. Consiste en:
 - Un servidor Node.js
 - Un cliente Javascript para el navegador

"Socket.io no es una implementación de WebSocket"

- socket.io utiliza WebSocket como transporte pero añade metadatos a cada paquete: tipo de paquete, namespace y el id de reconocimiento del mensaje
 - Clientes y servidores WebSocket no son compatibles con socket.io
-

name:socketcaracteristicas task: << [índice de contenidos](#) >>

[socket.io](#): Características

Confiabilidad => La conexión se establece incluso en presencia de proxies o firewalls.

Soporte de auto-reconocimiento => Un cliente desconectado intenta reconectarse de manera continua hasta que el servidor está disponible nuevamente

Detección de desconexión => Un mecanismo de pings (*heartbeat*) permite al servidor y al cliente saber cuando uno de los dos no está respondiendo

Soporte binario => Se puede emitir ArrayBuffer, Blob (navegador) o Buffer (Node.js)

Soporte multiplexado => Se pueden crear varios Namespaces que actúan como canales separados, sobre una misma conexión

Soporte room => Sobre cada namespace se pueden definir canales arbitrarios (Rooms) a los cuales se pueden unir los sockets. Los mensajes se emiten a un room determinado llegando a los sockets adheridos al room

name:socketinstall task: << [índice de contenidos](#) >>

socket.io: instalación

Instalar socket.io en el servidor:

```
.inverse[ npm install --save socket.io ]
```

Se puede utilizar un cliente desde socket.io:

```
.inverse[ npm install --save socket.io-client ]
```

name:socketinstall task: << [índice de contenidos](#) >>

socket.io: configuración del servidor

```
import http from 'http';
import express from 'express';
import socketio from 'socket.io';
import { dirname } from 'path';
import { fileURLToPath } from 'url';

const __dirname = dirname(fileURLToPath(import.meta.url));

const app = express();
const server = http.createServer(app);

const io = socketio(server);

server.listen(80);
// WARNING: app.listen(80) no funciona

app.get('/', (req, res) => {
  res.sendFile(`${__dirname}/index.html`);
});
```

```
// Eventos en servidor:
'connection','anything','disconnect','message')
io.on('connection', socket => {

    socket.emit('news', { hello: 'world' });

    socket.on('my other event', data => console.log(data));

});

...

<ul style="font:10px italic">
<li> Para instalar socket.io es necesario tener instanciado un servidor
http </li>
<li> Se carga el módulo de <code>socket.io</code> y se pasa el servidor
como parámetro </li>
<li> Para gestionar si se está produciendo una nueva conexión de websockets
al servidor se añade un _listener_ al evento <code>connection</code> sobre
<code>sockets.io</code> </li>
<li> La conexión devuelve un <code>stream socket</code> (igual que la
conexión sobre TCP) que se utiliza para gestionar la conexión con el
cliente </li>
<li> Se utiliza express para gestionar las rutas de las peticiones http al
servidor. </li>
<li> En este caso se carga el cliente en el navegador con
<code>res.sendFile(`${__dirname}/index.html`);</code> </li>

</ul>

---

name:socketinstall
task: [<< índice de contenidos >>](#contenido)

### [socket.io](https://socket.io/): configuración del cliente

- El cliente se puede configurar en el mismo servidor a partir del cliente
de socket.io (<"/socket.io/socket.io.js">)

- También puede configurarse mediante cdn:

```html
<script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js">
</script>
```

- O también instalando el package `socket.io-client` para otros servidores nodeJS

---

name:socketinstall task: << índice de contenidos >>

[socket.io](#): Namespaces

- Los namespaces permiten asignar diferentes *endpoints* o rutas a un *socket*

- Utilidad:

- Minimiza el número de recursos (conexiones TCP)
- Separa conceptos entre canales de comunicación
- Namespace por DEFECTO: "/"
- Se pueden personalizar:

```
const nsp = io.of('/my-namespace');
```

```
nsp.on('connection', socket => console.log('someone connected')); nsp.emit('hi', 'everyone!');
```

- El cliente se conecta a la ruta expuesta:

```
const socket = io('/my-namespace');
```

---

name:socketinstall task: << índice de contenidos >>

## socket.io: Rooms

Son **canales de emisión** definidos dentro de cada namespace. Los sockets pueden unirse o salir.

### Joining

Se utiliza `socket.join` para subscribirse al canal:

```
io.on('connection', socket => socket.join('some room'));
```

Y se utiliza `to` o `in` para emitir un mensaje:

```
io.to('some room').emit('some event');
```

---

name:socketinstall task: << índice de contenidos >>

## socket.io: Rooms

### Default Room

Cada socket está identificado por un id único: **Socket#id**.

Y cada socket se subscribe automáticamente al canal (room) identificado por su id

```
io.on('connection', socket => {

 socket.on('say to someone', (id, msg) =>
 socket.broadcast.to(id).emit('my message', msg))

});
```

---

name:socketinstall task: << [índice de contenidos](#) >>

## socket.io: Real-time API

En el marco de Desarrollo de una API RESTful la emission de eventos debería recaer en el servidor y no en el cliente.

El servidor difunde los mensajes a los clientes

Ej.:

```
import http from 'http';
import express from 'express';
import socketio from 'socket.io';

const app = express();
const server = http.createServer(app);

const io = socketio(server);

server.listen(80);

app.post('/webhook/orders/updated', (req, res, next) => {

 io.sockets.emit('order', "Order Id " + req.body.data.id + " :
Updated");

});
```

---

## Ejercicio 28

Implementar comunicación vía socket.io en el ejercicio de student-api. Cada vez que se inserte un nuevo documento en la colección student debe verse en el navegador de un cliente.

BONUS: configurar un segundo cliente en un servidor aparte.