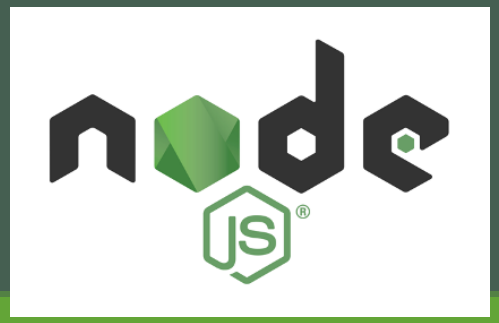


Módulos

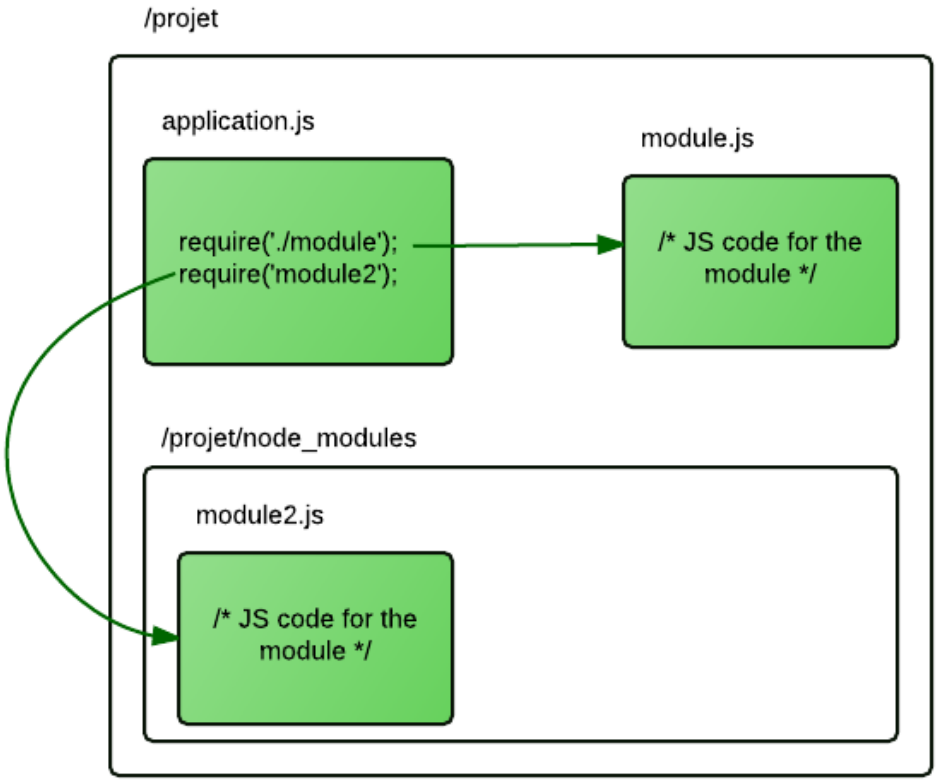
Contenido

- Introducción
- Trabajar con módulos internos de Node
- Módulos externos
- Módulos propios

Diseño Modular

- El sistema modular de JavaScript se vuelve cada vez mas importante para los desarrolladores web

- Los módulos son grupos de código, los cuales son completamente independientes entre sí, con funcionalidades distintas, y que pueden ser mezclados, añadidos y eliminados, sin alterar el sistema en su conjunto.



Diseño modular: 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, sin contaminar al resto de módulos.
- **Reutilización,** un módulo es susceptible de ser usado por diversas aplicaciones en distintos contextos, ya que expone una interface.

Diseño modular: Patrón Módulo Revelado (Revealing Module Pattern)

- Se usa para “imitar” el concepto de clase y, así, almacenar métodos y poder trabajar con variables públicas y privadas dentro de un objeto –de manera similar a Java o Phyton.
- Esto permite crear una API para los métodos que se quieren exponer al mundo, a la vez que se encapsulan variables y métodos privados en un contexto cerrado.
- En javascript () el patrón se implementa como:

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

Diseño modular: 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

Diseño modular: CommonJS

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

```
function myModule() {
  this.hello = function() {
    return 'hello!';
  }

  this.goodbye = function() {
    return 'goodbye!';
  }
}

module.exports = myModule;
```

- Usamos el objeto especial del módulo y colocamos una referencia de la función en **module.exports**.

Diseño modular: CommonJS

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

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

var myModuleInstance = new myModule();
myModuleInstance.hello();
myModuleInstance.goodbye();
```

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

Diseño modular: CommonJS en NodeJS

- NodeJS 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

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');
```

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.
- No hay que preocuparse por si dos o más módulos tienen funciones con el mismo nombre.
- Para exportar funciones o variables de un módulo se utiliza “exports”:

```
// book.js
exports.name = 'pepe';
exports.read = function() {
console.log('Hola ' + exports.name);
}
```

- O también “module.exports”:

```
// book.js
module.exports = function() {
name = 'pepe';
console.log('Hola ' + exports.name);
}
```

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);
}}
```

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 }
}
```

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.
  - Los módulos ES6 son automáticamente código en modo estricto, incluso si no se escribe "use strict"; en ellos.
  - 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.
// (Heather Arthur realmente escribió esta librería)
// (pero no usó módulos, porque era el 2013)

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() {
  ...
}
...
```

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";
```

Diseño modular: ES Modules en NodeJS

- En NodeJS los módulos ES se encuentran en fase experimental.
- Para trabajar con ellos se requiere crear un fichero con la extensión .mjs y activarlo con el flag *--experimental-modules:*

```
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
```

Diferencias CJS Modules vs MJS Modules

- Lós módulos ES6 se cargan de forma asíncrona, mientras que los CJS se cargan de forma síncrona.
- Los modulos 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.

Notas sobre el diseño modular

- Al empezar a diseñar módulos, dos conceptos clave del desarrollo software: **cohesión y desacoplamiento**.
- Todo módulo, para que esté bien diseñado, debería **cumplir con el principio de única responsabilidad**.
  - Definir cual es la única responsabilidad de un módulo suele ser complicado y depende de muchos parámetros del contexto, pero tenerlo en mente puede ayudar a crear mejor software.
- 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.**
  - 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.

- 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

Ejercicio 2

- 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 tildes francesas para incluir variables: `${}`

- BONUS: Mismo ejercicio en ES Modules

Ejercicio 3

- 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');}}

Ejercicio 4

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