

# Integración de aplicaciones

## Tema 1. Introducción

© 2020 Javier Esparza Peidro - [jesparza@dsic.upv.es](mailto:jesparza@dsic.upv.es)

# Contenido

- Introducción
- Principios de diseño
- Node.js

# Introducción

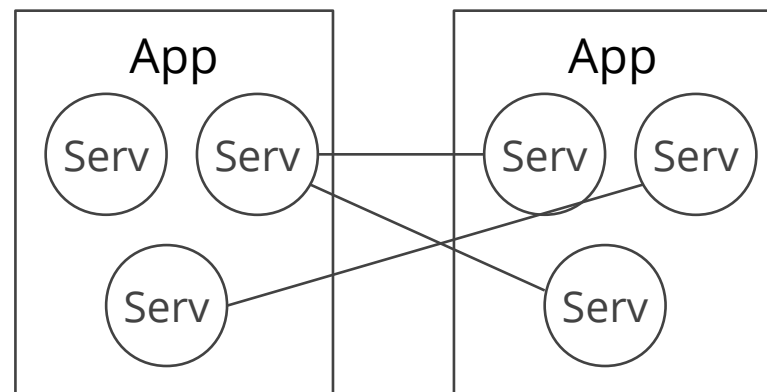
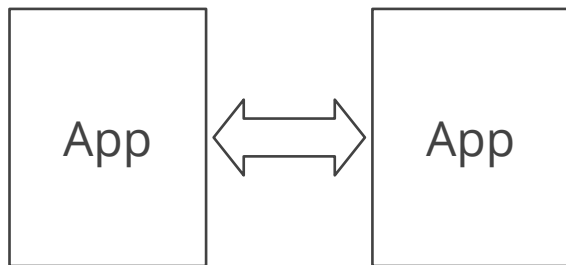
## ¿Por qué integrar aplicaciones?

- La complejidad del software va en aumento
- Es raro (y nada deseable) encontrar “monolitos” que contengan TODA la información e integren TODA la funcionalidad de la compañía
- La mayoría de compañías dispone de múltiples sistemas independientes
- Las aplicaciones se deben conectar para alcanzar sus objetivos

# Introducción

## ¿Qué es la integración de aplicaciones?

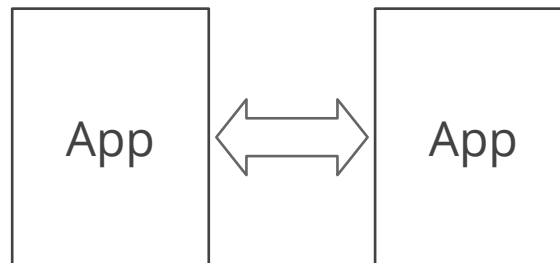
- Es el proceso que permite a aplicaciones diseñadas de manera independiente trabajar de manera conjunta
- Generalmente consiste en **compartir información**
- Integración **inter-aplicación** vs **intra-aplicación**
- Actualmente, con SOA, la frontera es muy difusa



# Introducción

## Problemas

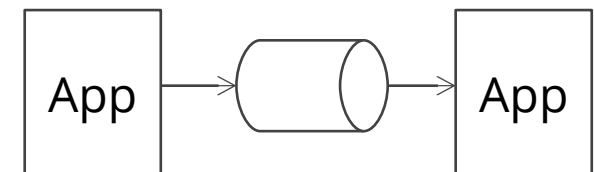
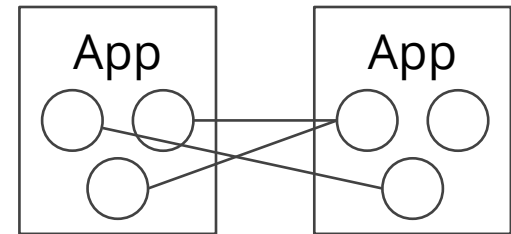
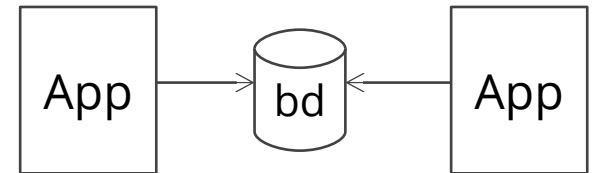
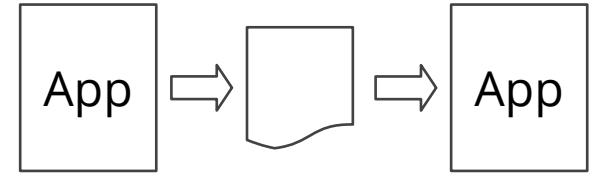
- Cada aplicación se ejecuta en su entorno
- Problemas de comunicaciones: fiabilidad, velocidad
- Distintas tecnologías, modelos y formatos de datos
- Evolución independiente: compatibilidad



# Introducción

## Soluciones

- Transferencia de fichero
- Bases de datos compartidas
- Arquitecturas SOA
- Sistemas basados en mensajería



# Principios de diseño

- Conceptualmente, en un sistema compuesto de aplicaciones, cada aplicación juega el papel de un módulo/componente
- Es habitual aplicar los principios fundamentales de cualquier diseño modular: **acoplamiento** y **cohesión**
- Estos dos principios están relacionados entre sí: una cohesión alta suele implicar un acoplamiento bajo y viceversa

# Principios de diseño

## Acoplamiento

- Mide el grado de **interdependencia** entre componentes
- Se persigue obtener componentes con **acoplamiento bajo**
- Favorece la reutilización de componentes
- Mejora la mantenibilidad de componentes
- Contribuye a contener fallos



# Principios de diseño

## Cohesión


- Mide el nivel de **interrelación** de los elementos incluidos en un mismo componente
- Se persigue obtener componentes con **cohesión fuerte**
- Los componentes son más sencillos
- Favorece la reutilización de componentes
- Favorece la mantenibilidad de componentes

# Node.js

- En la actualidad es posible desarrollar aplicaciones con múltiples tecnologías y lenguajes de programación
- En esta asignatura trabajaremos con **Node.js**:
  - El alumno posee un conocimiento básico de JavaScript
  - Es una tecnología muy eficiente para construir aplicaciones en el servidor pero también GUIs
  - Librerías de terceros para implementar cualquier estrategia de integración

# Node.js

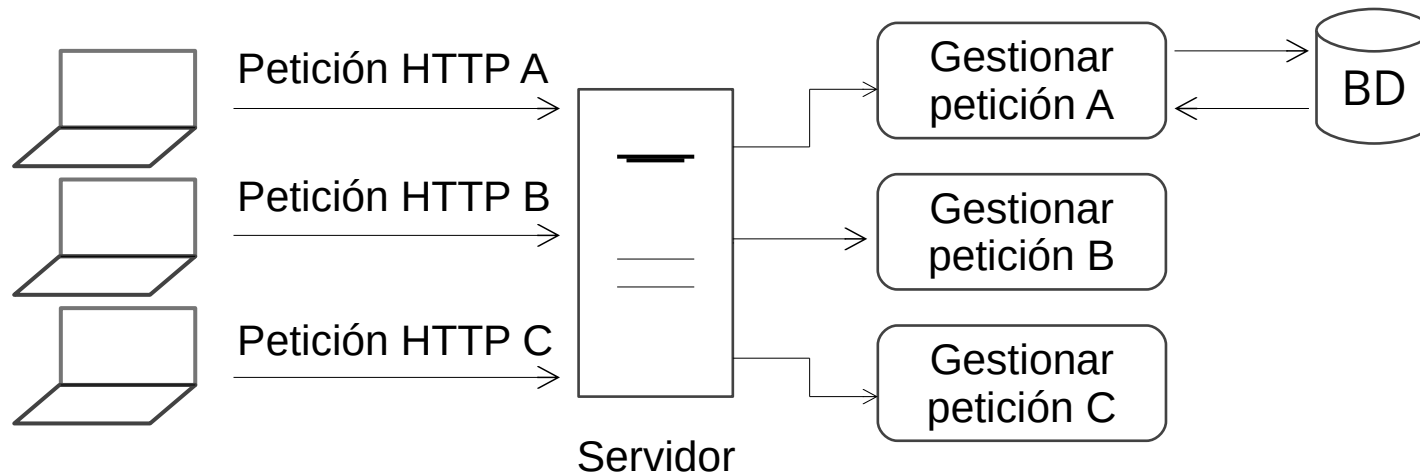
## Introducción

- <https://nodejs.org> 
- Runtime JavaScript basado en el motor V8 (corazón del intérprete JS de Chrome) de Google
- Tecnología especializada en crear aplicaciones muy eficientes, poniendo énfasis en la escalabilidad
- Dos secretos principales:
  1. Sólo existe un thread de ejecución: aplicaciones dirigidas por eventos
  2. Operaciones I/O asíncronas (no bloquean)

# Node.js

## Un sólo hilo de ejecución

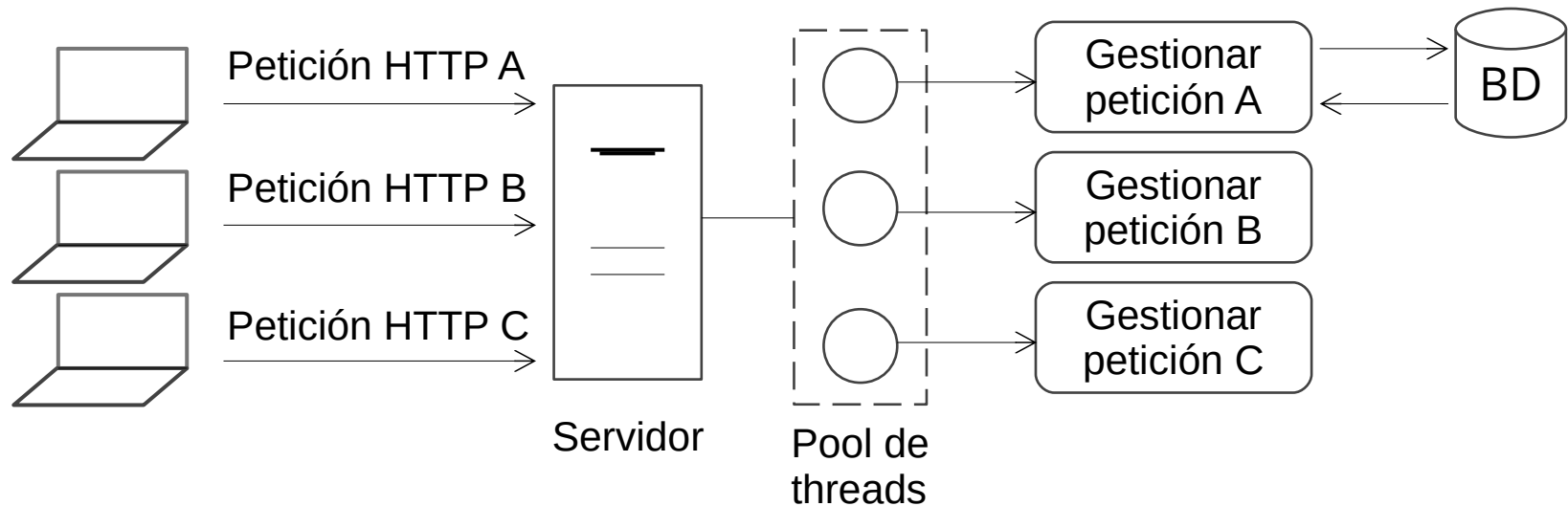
- Servidor web multi-proceso tradicional



# Node.js

## Un sólo hilo de ejecución

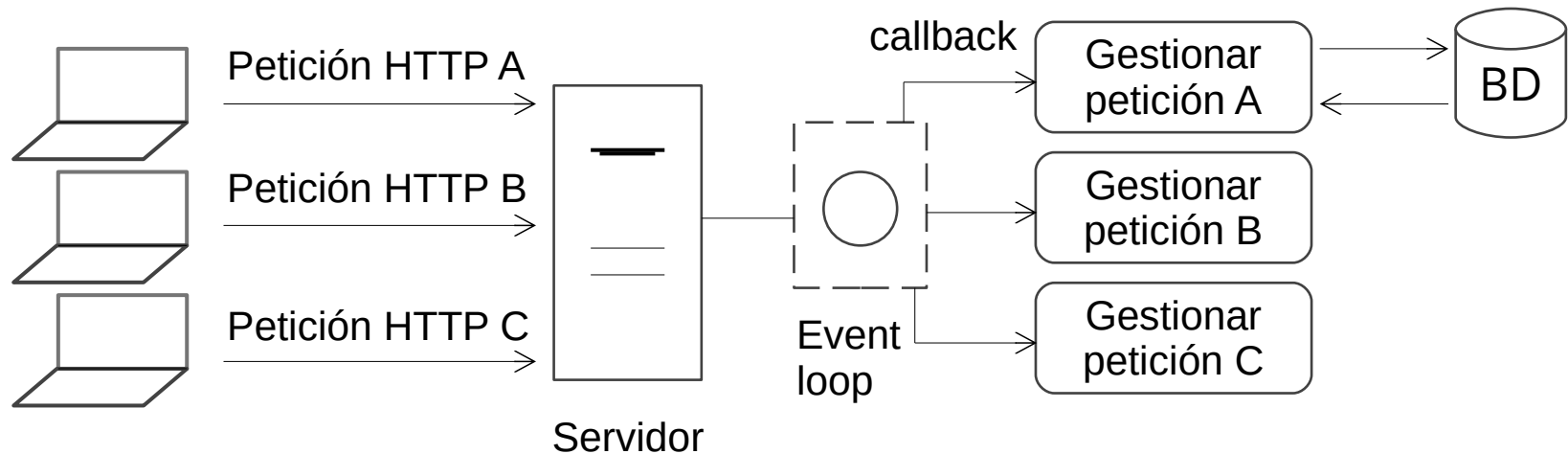
- Servidor web con pool de threads



# Node.js

## Un sólo hilo de ejecución

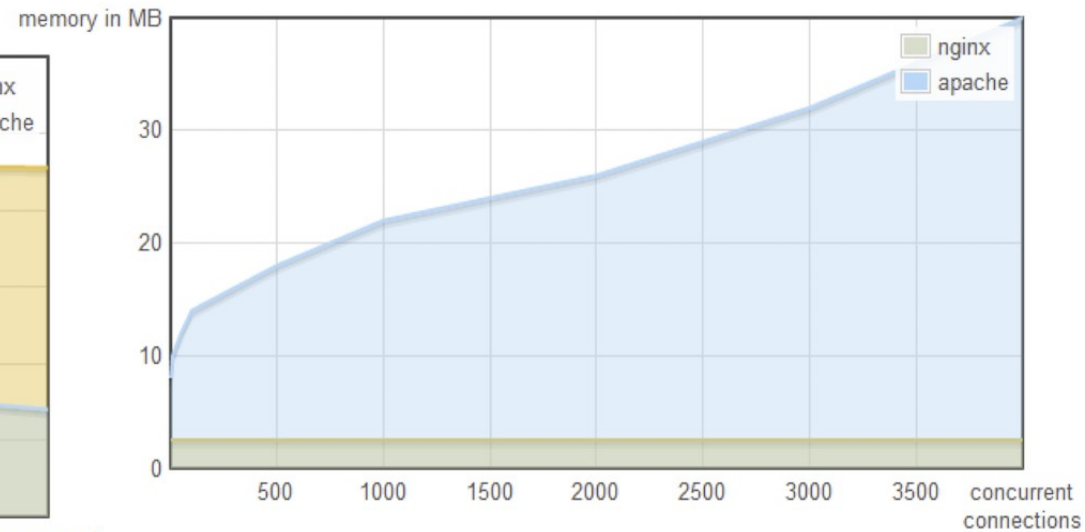
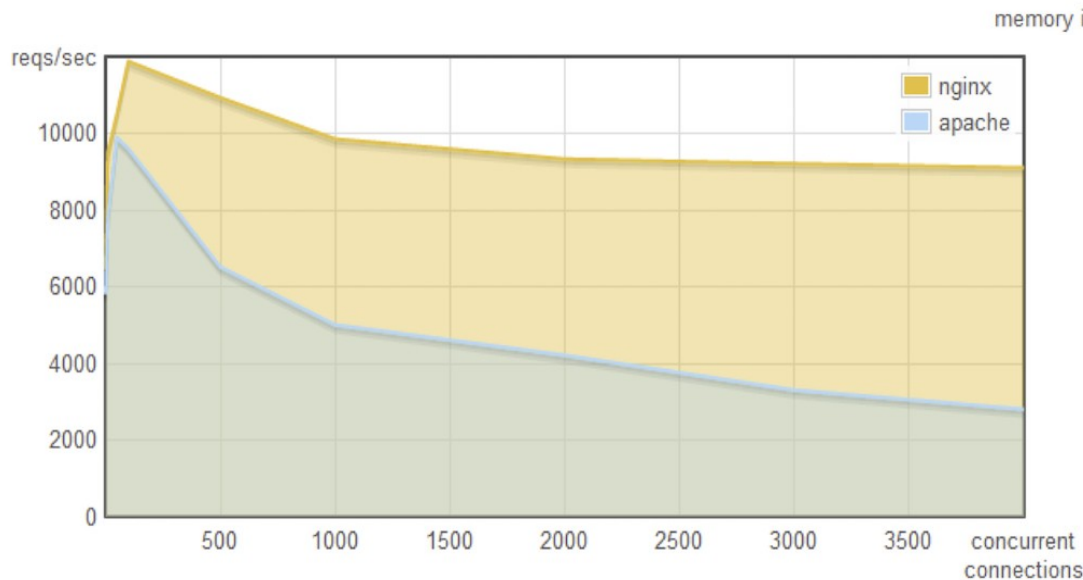
- Servidor web con un solo thread



# Node.js

## Un sólo hilo de ejecución

- Apache (pool threads) vs nginx (un solo thread)

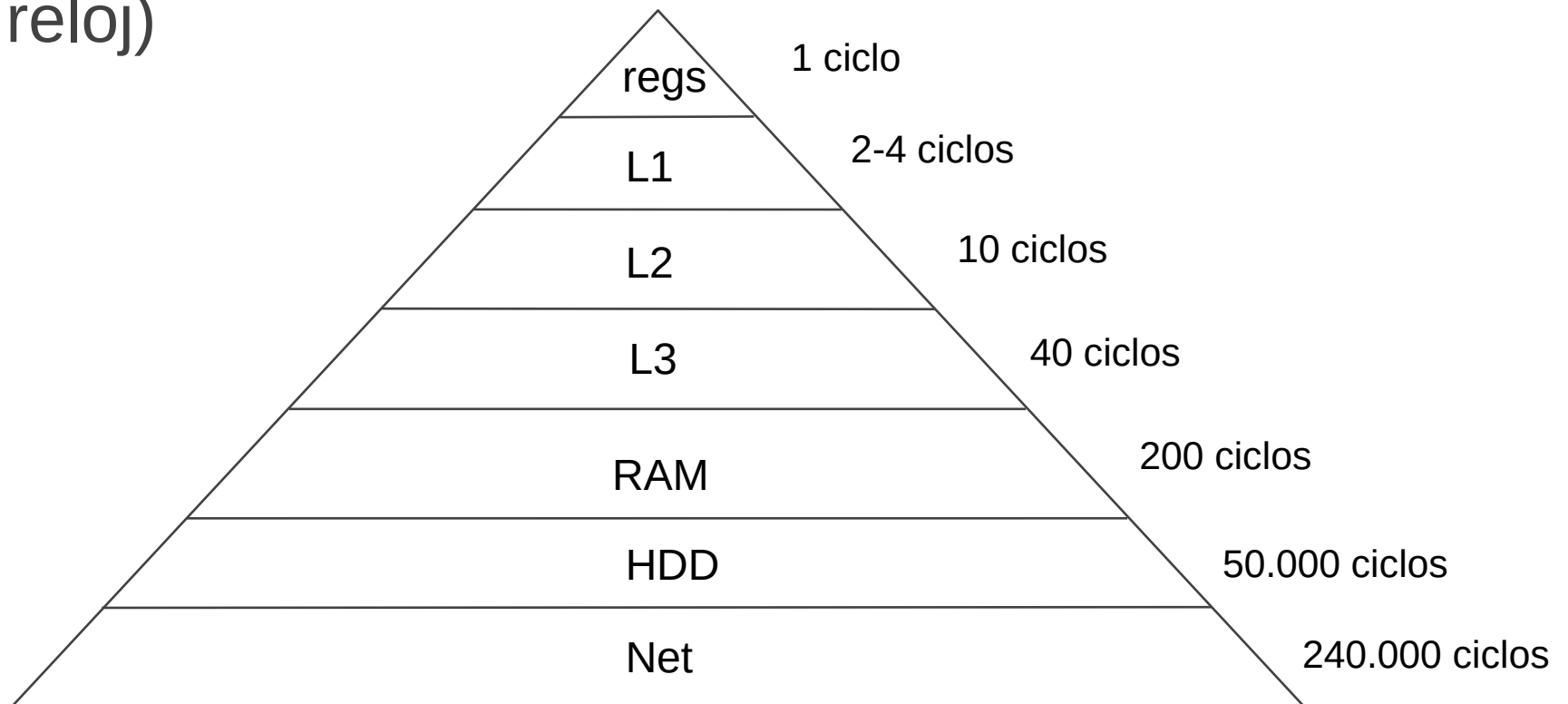


<https://www.nginx.com/blog/using-nginx-plus-web-server/>

# Node.js

## Asíncrono

- El problema de escalado I/O
- Las operaciones I/O bloquean el thread (muchos ciclos de reloj)





# Node.js

## Asíncrono

- Seamos asíncronos ...

```
// nueva petición HTTP
function handleClientRequest(request, response) {

    // hacer llamada asíncrona
    makeDBCall("select * from users", function(result) {

        // cuando finalizar, el resultado está accesible
        response.write(result);

    });
}
```

- No más operaciones bloqueantes: ficheros, red, bd, etc.
- Claves de Node.js: asincronía + eventos + streams

# Node.js

## Hoja de ruta

- Instalación
- Módulos y paquetes
- Eventos
- Streams

# Node.js

## Instalación

- <http://nodejs.org>
- REPL (Read-Evaluate-Print Loop)

```
$ node
> .help
break    Sometimes you get stuck, this gets you out
clear    Alias for .break
exit     Exit the repl
help     Show repl options
load     Load JS from a file into the REPL session
save     Save all evaluated commands in this REPL session to a file

> console.log('Hola mundo!');
Hola mundo!
undefined
```

# Node.js

## Módulos y paquetes

- Una aplicación Node.js se compone de varios módulos (ficheros .js)
- Uno de ellos es el módulo principal
- Ejecutamos el módulo principal usando el comando **node**

```
// módulo app.js  
console.log('Hola mundo!!!');
```

```
> node app  
Hola mundo!!!
```

# Node.js

## Módulos y paquetes

- El resto de módulos son típicamente ficheros .js que exportan alguna funcionalidad (son librerías)
- Para exportar cierta funcionalidad, un módulo utiliza la variable implícita `module.exports` o `exports` (alias)
- En un módulo podemos importar otro con `require('mod')`
- Al importar un módulo recibimos un objeto que posee la funcionalidad exportada (es el objeto exports!!)

```
// módulo foo.js (librería)  
exports.print = function(str) { console.log(str); }
```

```
// módulo app.js (principal)  
var foo = require('./foo');  
foo.print('Hola mundo!!');
```

Importar módulo

# Node.js

## Módulos y paquetes

### Crear un módulo

- Fichero .js que tiene acceso a algunas **variables implícitas**
  - **module**: representa el módulo actual (.id, .exports, .filename, .loaded, .parent, .children, etc.)
  - **exports** (alias de module.exports): objeto que contiene las características exportadas del módulo (como propiedades)
  - **global**: objeto global de Node.js (como window en browser)
  - **\_\_filename**, **\_\_dirname**: path del módulo actual
  - **console**, **process**: consola salida y utilidades proceso
  - set/clearTimeout(), set/clearInterval(), ...

# Node.js

## Módulos y paquetes

### Crear un módulo

- Ejemplo

```
// módulo info.js
console.log('__filename:' + __filename);
console.log('__dirname:' + __dirname);
console.log('module.id:' + module.id);
console.log('module.filename:' + module.filename);
console.log('process.pid:' + process.pid);
console.log('process.title:' + process.title);
console.log('process.arch:' + process.arch);
console.log('process.platform:' + process.platform);
for (var att in process.argv)
    console.log(att + ':' + process.argv[att]);
```

# Node.js

## Módulos y paquetes

### Importar un módulo

- Con `require('<módulo>')`
- Cuando un módulo se importa, sus instrucciones se ejecutan de manera **síncrona**
- Dependiendo de cómo se especifica <módulo>, Node.js soporta tres tipos de módulos:
  1. Módulo de tipo fichero
  2. Módulo core
  3. Módulo de librería (paquete)



# Node.js

## Módulos y paquetes

### 1. Importar un módulo de tipo fichero

- Con `require('./xxx')`, `require('../xxx')`, `require('/xxx')`
- 'xxx' es cualquier path (ej. './padre/hijo/modulo')
- La extensión no es necesaria: se buscan 'xxx.js', 'xxx.json', 'xxx.node'

```
// módulo ./foo/bar.js  
console.log('en bar');  
module.exports = {msg: 'Hola mundo!'};
```

```
// módulo ./app.js  
console.log('en app');  
var bar = require('./foo/bar');  
console.log(bar.msg);
```

```
./  
 \_ app.js  
 \_ foo/  
     \_ bar.js
```

# Node.js

## Módulos y paquetes

### 2. Importar un módulo core

- Con `require('xxx')`, donde xxx es un identificador conocido
- Node.js publica varios módulos core: path, fs, os, util, http, net, child\_process, etc.

```
var path = require('path');
console.log(path.normalize('/foo/bar/..'));
console.log(path.join('foo', '/bar', 'bas'));
var fs = require('fs');
fs.writeFileSync('test.txt', 'Hello fs!');
console.log(fs.readFileSync('test.txt').toString());
var os = require('os');
console.log('Total Memory', os.totalmem());
console.log('Available Memory', os.freemem());
var util = require('util');
util.log('sample message');
console.log(util.format('%s has %d dollars', 'michael', 10));
console.log(util.isDate(new Date()));
```

# Node.js

## Módulos y paquetes

### 3. Importar un módulo de librería (paquete)

- En Node.js las librerías de terceros se denominan “paquetes”
- Antes de usarlas, es necesario instalarlas (usando NPM)

```
> npm install <paquete>
```

- Se instalan típicamente bajo una subcarpeta `/node_modules`
- Una vez instaladas, pueden ser importadas usando `require('xxx')`, donde xxx es un identificador (el identificador del paquete) pero no es un módulo core

# Node.js

## Módulos y paquetes

### 3. Importar un módulo de librería (paquete)

- Entonces Node.js busca dentro del directorio `./node_modules` en el directorio raíz del módulo actual
- Si no lo encuentra, lo busca en el padre (y en el padre del padre ...)
- Hasta que llega al directorio raíz `/`. Si no lo encuentra, error

```
// módulo /home/ry/projects/foo.js  
require('bar');
```

path búsqueda

```
/home/ry/projects/node_modules/bar.js  
/home/ry/node_modules/bar.js  
/home/node_modules/bar.js  
/node_modules/bar.js
```

# Node.js

## Módulos y paquetes

- Node.js no viene con una lista masiva de librerías
- Filosofía de Node.js: proporcionar librerías base y que la comunidad haga el resto
- Las extensiones son desarrolladas por la Comunidad Open Source Community: **paquetes**
- Node.js proporciona su propio gestor de paquetes: **npm** (Node Package Manager)

# Node.js

## Módulos y paquetes

### Paquete

- Es un artefacto Node.js autocontenido
- Contiene el fichero package.json en el directorio raíz

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "A test app",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "@angular/core": "5.2.9",
    "ionic-angular": "3.9.2",
    ...
  }
}
```

# Node.js

## Módulos y paquetes

### Paquete

- En su interior se pueden definir módulos, binarios, recursos, etc.

```
underscore/  
├── LICENSE  
├── package.json  
├── README.md  
├── underscore.js  
├── underscore-min.js  
└── underscore-min.map
```

- Los paquetes se instalan/actualizan/desinstalan usando **npm**

# Node.js

## Módulos y paquetes

### NPM

- Herramienta que automatiza gestión (instalación, actualización, eliminación, ...) de dependencias en nuestro proyecto
- Es el gestor de paquetes más popular: +1.200.000
- Se integra con el registro online: <https://www.npmjs.com>



# Node.js

## Módulos y paquetes

### NPM. Inicializar un proyecto

```
> npm init
```

- Se crea package.json en el directorio actual
- package.json registra todos los paquetes instalados en el proyecto, y otras cosas ...

```
{  
  "name": "test",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": { ... }  
}
```

# Node.js

## Módulos y paquetes

### NPM. Instalar un paquete

```
> npm install <package> [-g] [--save-dev]
```

- La versión más actual del paquete (y dependencias) se instala en `./node_modules` y se registra en `package.json`
- Con `-g` el paquete se instala en el registro global

```
> npm install underscore
```

```
{  
  "name": "xxx",  
  ...  
  "dependencies": {  
    "underscore": "^1.9.0",  
    ...  
  }  
  "devDependencies": { ... }  
}
```

# Node.js

## Módulos y paquetes

NPM. Reconstruir dependencias

```
> npm install
```

- Instala todas las dependencias registradas en package.json

Otros comandos

```
> npm ls|list [-g]
```

```
> npm rm|uninstall <package> [--save]
```

```
> npm search <package>
```

# Node.js

## Módulos y paquetes

### NPM. Automatizar tareas

> npm run <command>

- En package.json > scripts se definen comandos que permiten automatizar tareas

```
{  
  "name": "xxx",  
  ...  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "build": "ts *.ts",  
    "serve": "http-server -p 8100 dist/"  
  },  
  "dependencies": { ... }  
}
```

> npm run build

> npm run serve

# Node.js

## Módulos y paquetes

### NPM. Ejecutar binarios

> npx <command>

- Algunos paquetes instalan binarios-utilidades (bajo node\_modules/.bin generalmente)
- Podemos ejecutarlos con npx

# Node.js

## Eventos

- Esenciales para habilitar programación asíncrona
- Un objeto difunde/emite eventos
- Un cliente/suscriptor registra/suscribe callbacks/manejadores en dicho objeto para recibir los eventos y reaccionar convenientemente
- En Node.js hay muchos objetos que emiten eventos; todos ellos son instancias de la clase EventEmitter

# Node.js

## EventEmitter

- Incluido en el modulo 'events':  
`require('events').EventEmitter`
- Para suscribirnos/cancelar suscripción de un evento:  
`emitter.on/removeListener('event', listener)`
- Para emitir un evento:  
`emitter.emit('event', arg1, arg2, ...)`

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
emitter.on('hello', function() { console.log('hello'); });
emitter.emit('hello');
```

# Node.js

## EventEmitter

- En Node.js los emisores de eventos típicamente emiten el evento **'error'** cuando se encuentra un error.
- Si este evento no es gestionado por ningún manejador, la acción por defecto consiste en imprimir la pila de ejecución y salir del programa.

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
emitter.emit('error'); // finishes program execution

emitter.on('error', function() { console.log('error'); });
emitter.emit('error'); // program execution does not finish
```

- En Node.js existen muchos objetos que implementan EventEmitter: process, net.Server, stream.Readable, etc.



# Node.js

## process

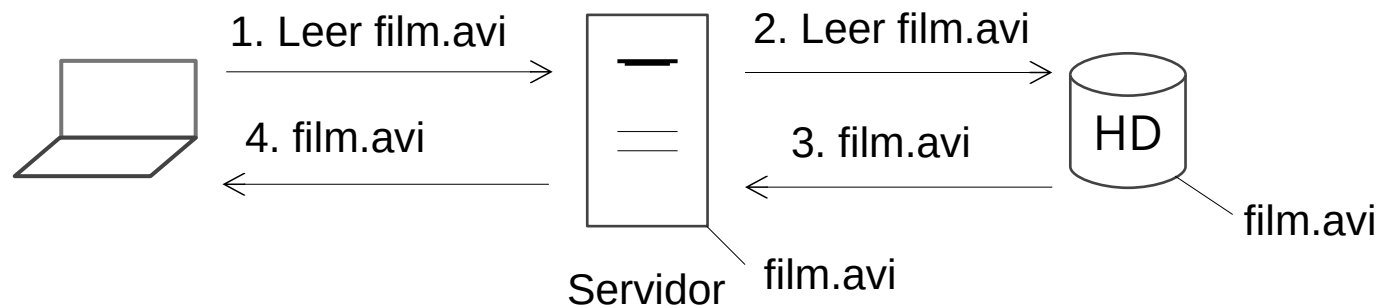
- Variable global instancia de **EventEmitter**
- Emite eventos: **'exit'** (la aplicación va a parar), **'beforeExit'** (no hay más eventos que procesar), **'uncaughtException'** (excepción inesperada que se puede controlar), **'<signal>'** (SIGHUP, SIGINT, SIGKILL, ...)

```
process.on('exit', function() { console.log('adios'); });
process.on('uncaughtException', function() {
  console.log('Evitando que la aplicación finalice');
});
process.on('beforeExit', function() {
  console.log('plan more work to avoid exiting');
});
```

# Node.js

## Streams vs Buffering

- Servir un fichero de 1Gb desde un servidor web usando **buffering**

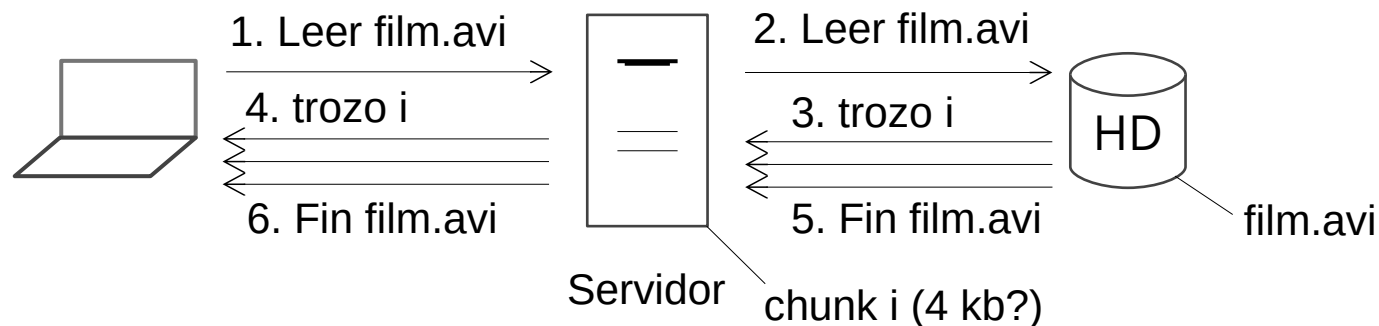


- Es necesario reservar muchos recursos por el camino
- ¿Qué pasa si suceden n peticiones concurrentes?

# Node.js

## Streams vs Buffering

- Servir un fichero de 1Gb desde un servidor web usando **streaming**



- Se reservan muy pocos recursos por el camino
- No hay problema con n peticiones concurrentes

# Node.js

## Streams

- Lo básico en el módulo 'stream': `require('stream')`
- Los streams son **EventEmitters**:
  - **Readable**: streams de sólo lectura (ej. petición HTTP, etc.)
    - Métodos: `.read()`, `.pipe()`, `.pause()`, `.resume()`; Eventos: `'readable'`, `'data'`, `'end'`, `'close'`, `'error'`
  - **Writable**: streams de sólo escritura (ej. respuesta HTTP, etc.)
    - Métodos: `.write()`, `.end()`; Eventos: `'finish'`, `'pipe'`, `'error'`
  - **Duplex**: implementa Readable y Writable (ej. socket)
  - **Transform**: Duplex que cambia la entrada y genera una salida (ej. stream zlib, stream encriptación, etc.)

# Node.js

## Ficheros

- Módulo 'fs': `var fs = require('fs')`
- `var inp = fs.createReadStream(path, options)`
- `var outp = fs.createWriteStream(path, options)`

```
var fs = require('fs');
var inp = fs.createReadStream('test.txt');
inp.on('data', function(data) {
  console.log('Recupero ' + data.length + ' bytes de datos');
});

// conectar entrada a salida
var inp = fs.createReadStream('test.txt');
var outp = fs.createWriteStream('output.txt');
inp.pipe(outp);
```

# Node.js

## Ficheros

- Además de con streams, el sistema de ficheros puede ser accedido utilizando otras técnicas (sync o async):
  - `fs.open[Sync](path, flags, mode[, callback])`
  - `fs.close[Sync](fd [, callback])`
  - `fs.read[Sync](fd, buf, offset, length, position [, callback])`
  - `fs.write[Sync](fd, buf, offset, length [, callback])`

```
var fs = require('fs');
fs.open('test.txt', 'r', function(err, fd) {
  if (!err) {
    var buf = new Buffer(1000);
    fs.read(fd, buf, 0, 1000, 0, function(err, bytesRead, buf) {
      console.log('Bytes leídos: ' + bytesRead + ' -> ' + buf);
    });
  }
});
```

# Node.js



## Ejercicio 1

Diseñar una aplicación que permita efectuar operaciones básicas de gestión de contactos a través de la consola:

- REPL (Read-Evaluate-Print-Loop) vs CLI (Command line interface)
- Listar
- Añadir
- Actualizar
- Eliminar

