

## NodeJS introduction

### Primera Sessió

1. Explicar com s'instala: o te'l baixes precompilat o simplement fas un apt-get install nodejs.
2. Explicar que fa la comanda "node" (és un interpret de javascript, en concret el V8 de google, + una non-blocking I/O api).
3. Fem uns quants exemples per familiaritzar-nos (obrir **node** a **class-term.sh**):

---

```
> 3 + 3;
6
> "hola" + 2;
'hola2'
> true || false;
true
> 0 == 0;
true
> 0 == false;
true
> 0 === false; // equal value and equal type
false
> (true && false) === false;
true
> true !== false;
true
> typeof 0;
'number'
> typeof false;
'boolean'
```

---

4. Variables

---

```
> foo = 3; // variable no declarada = scope global
> var bar = 4; // variable declarada = scope funció
// Fer servir var =
// (a) te scope funció i també:
// (b) Undeclared variables do not exist until the code assigning to them
//     is executed.
// (c) Undeclared variables are configurable (e.g. can be deleted).
```

---

5. Coses "rars":

---

```
> foo = "hola"
'hola'
// Hi ha el null...
> foo = null
null
> typeof null;
'object'
// I també el undefined...
> undefined;
undefined
> typeof xxx;
undefined;
```

```
// All numbers are doubles:
> 0.1 + 0.2
0.30000000000000004
> 0/0
NaN
> 1/0
Infinity // No hauria de ser +/- Infinity !? ...
> 1/-0
-Infinity // ...no pq el zero té signe!
```

---

## 6. Objectes i funcions:

```
> l = [1,2,3,4]; // (a) una llista
> t = { a : "hola", b : "adeu" } // (b) map o millor dir Object
> t.a
> t.b
> m = /3/ // (c) Una perl-like regex /3/gmi
> m.test(2)
> m.test(3)
> m = /3/g
> m.exec('33')
> m.exec('33')
> m.exec('33')
> function f1(a) { return a + 1 };
> var f1 = function(a) { return a + 1 }; // (d) una funció
> f1.toString();
> f1(1);
> f1p = function() { };
> f1p();
undefined;
```

---

## 7. Programació funcional

```
// Map
> l.map(f1);
// Pels side effects (en aquest cas res)
> l.forEach(f1);
// Funcions anònimes
> (function(a) { return a + 1 })(1);
```

---

## 8. Closures

```
// Podem definir una funció dintre d'una altra (això no és una closure!).
> f2 = function(a) {
var f3 = function(b) {
return b + 1;
}
return f3(a);
}
> f2(3);

// Aquest és la closure:
> f2 = function(a) {
return function (b) { return a + b; };
}
```

```

> f2
[function]
> f2(1)
[function]
> f2(1)(3)
4
// Un altre exemple:
> genRandom = function(seed) {
    var state = seed;
    return function() {
        // aquest es (era?) el congruential PSNR de glibc:
        state = (state * 1103515245 + 12345) % (2*(1<<30));
        return state; };
}
> genRandom(1)
// Comentari sobre els LFSR

```

---

9. 'objects', just some hints...

```

> t;
{ a: 'hola', b: 'adeu' }
> t.method = function() {};
> t.method();
> t.method = function() {return this.a; };
'hola'
// Javascript és força més complicat del que sembla a primera vista...
// Més sobre això més endavant.

```

---

10. Ara anem a veure que passa amb l'altra meitat de node (la non-blocking I/O api). Primer de tot tenim l'objecte Console que ve de serie (hi ha un munt més).

```

> console
{ log: [Function], info: [Function], warn: [Function], error: [Function],
  dir: [Function], time: [Function], timeEnd: [Function], trace: [Function],
  assert: [Function], Console: [Function: Console] }
> console.log("hola");
'hola'
undefined
> a = function () {console.trace("hola"); };
> a();

```

---

11. Despres tenim altres objectes (moduls) que els hem d'importar

```

> os = require('os');
{endianness: [Function], hostname: [Function], loadavg: [Function],
 uptime: [Function], freemem: [Function], totalmem: [Function],
 cpus: [Function], type: [Function], release: [Function],
 networkInterfaces: [Function], arch: [Function], platform: [Function],
 tmpdir: [Function], tmpDir: [Function],
 getNetworkInterfaces: [Function: deprecated], EOL: '\n' }
> os.type
[function]
> os.type();
'Linux'
> os.hostname();

```

```
'laptop'
> os.hostname.toString();
'function () { [native code] }'
> fs = require('fs');
// Per cert...
> require.toString();
```

---

## 12. I/O

```
> fs.writeFileSync('hola.txt', 'text\n');
undefined
> fs.readFileSync('hola.txt');
<Buffer 74 65 78 74>
> fs.readFileSync('hola.txt') + '';
> fs.unlinkSync('hola.txt');
// Fins a aquí tot ha estat síncron... demanem una cosa, esperem,
// i després ja està feta.
// Ara en asíncron...
> fs.readFile('hola.txt', f);
> f = function() { console.log('done'); }
// Aquí fer molt emfasi en que la crida a la funció f no ha bloquejat
// les coses que es podrien fer després...
> f = function(err, result) { console.log(result); }
> f = function(err, result) { fs.writeFile('hola2.txt', result); }
> f = function(err, result) { fs.writeFile('hola2.txt', result,
  function() {console.log('yes');}); console.log('task done?'); }
// D'una altra manera
> a = fs.createReadStream('hola.txt'); // Origen, Font
> b = fs.createWriteStream('hola3.txt'); // Destí, Clavagueró (Sumidero)
> a.pipe(b);
```

---

## 13. Passem a un arxiu... node file.js

```
http = require('http');
http.createServer(null).listen(8080);
--
http = require('http');

f = function(request, response) {
  response.writeHead(200, { "Content-Type" : "text/plain" });
  response.write("Hello World\n");
  response.end();
}

http.createServer(f).listen(8080);

console.log('Server running at http://127.0.0.1:8080/');
```

---

```
// Provar amb alguna cosa que gastí molta cpu...
for(i=0; i < 1E9; i++) { i + 1 };
// fer emfasi en que passa si la funció f triga 20 segons...
```

---

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

---

```
f = function(request, response) {
  response.writeHead(200, { 'Content-Type' : 'text/plain' });
  fs.readFile('Hola.txt',
    function(err, result) {
      response.write(result);
      response.end();
    }
  );
}
```

---

```
url = require('url');

url.parse(request.url).pathname;
```

---

## Segona sessió

### 1. Private Scope

```
// A JS un scope es una funció si poses var, i s'enten global si no poses res.
> f1 = function() { var a = 1; { var b = 2; } return b; }
// Ojo amb això que no els brackets que no són de scope
// (són un block que pot tindre una etiqueta, i sortir-ne amb
// un break. e.g, block: { break block; } )

// Per fer un scope privat fem una funció:
private = function () {
}
private();

// Per fer un scope privat ben fet: funció anonima + executar-la allà mateix

// Això:
> privat = function() { /* scope privat */ }
> privat();

// Ho podem canviar per això:
> privat = (function() { /* scope privat */ })
> privat();

// I per això:
> (function() { /* scope privat */ })();

// Exemple 1:

var a = 1;

(function() {
  var variablePrivada = 1;
  // Faig coses amb la variable privada..
})();

// No puc accedir ja a la variable privada...

// Exemple 2:
function test() {
var a = 1;
```

```

(function() {
var b = 23;
a *= b;
})();

return a;
}

// -> Aquí haig de fer un altre exemple perquè sinò no s'enten el concepte.

```

---

## 2. Module Pattern

```

var testModule = (function() {
    var a = 1;

    return {
        increase : function() { a++; },
        value: function() { return a; }
    };
})();

```

---

## 3. Revealing Module Pattern

```

var testModule = (function() {
    var _a = 1;

    var _increase = function () { _a++; },
    var _value = function() { return _a; }

    return {
        increase : _increase,
        value: _value
    };
})();

```

---

## 4. Classes

```

> f1 = function() { return this.a; }
> o1 = { a : 33, b : f1 }
> o2 = { a : 44, b : f1 }
// Podem fer servir el this per saber de quin objecte penja la funcio b
// que es crida.
> o1.b()
> o2.b()
// Amb això fem fields i methods...

// Constructor + keyword new
function Counter() {
    this.a = 1;
}
o3 = new Counter();
o3.a

// a Counter podem afegir funcions etc...

```

```
function Counter() {
    this.a = 1;
    this.increase = function () { this.a++; },
    this.value = function () { return this.a; }
}
o3 = new Counter();
o3.increase()
o3.value()
```

---

## 5. Static fields

```
// Una variable vinculada a una classe i no a una instància...
// Les penjem del constructor.
Counter.MAX_VALUE = 23;
```

---

## 6. Inheritance

```
// Principals motius per fer herència:
// (A) As a subtyping mechanism; Interfaces: E.g., a aquesta funció se li pot
//      passar tot allò que compleixi la interfície.
// (B) Code-reuse: reaprofitar codi, override parts.
// A javascript te poc sentit el primer perquè es un llenguatge weakly-typed
// (hi ha poques restriccions per posar-ho tot a tot arreu; i si ho fas
// malament ja reventarà...). Per tant, ens queda la (B) !?.

// Prototype-based programming
// Cada objecte te un punter a un altre objecte: el seu prototipus.
// Quan no s'hi troba un mètode o variable (és el mateix),
// es busca al prototipus. I així recursivament...
> a = [1]
> a.__proto__ [tab]
// Veiem que les funcions d'array de fet les hereda
> a.__proto__.__proto__ ...
// Fem un miniexemple:
function ClassA() {
    this.a = 1;
}
ClassA.prototype = { b : 2 }; // ojo que això és un objecte existent
// i no un constructor.
o = new ClassA();
// D'una altra manera...
function ClassA() {
    this.a = 1;
}
function ClassB() {
    this.b = 2;
}
ClassA.prototype = new ClassB();
o = new ClassA();
// o.a, o.b

// Entendre el prototipatge com a manera d'obtenir còpies d'objectes ràpid
// per fer canvis... (code reuse)
car = {tires: 25, wheel: true };
bus = {tires: 100, __proto__: car };
// Això es pot fer més be amb altres parts de l'api (Object.create()), però
// de cara a entendre el concepte, amb això ja fem.
```

## Tercera sessió

### Zip Archiver

Introduir exemple: volem fer un tros de codi que ens agafi uns quants arxius, ens els posi tots en un zip, i serveixi després el zip al client. Tot, a mesura que el client es descarrega el zip, clar!

1. Primer de tot ens cal alguna manera de fer zips amb nodejs...  
NPM (node package manager) +100k packages!  
**npm install archiver** (es un modul per fer zips), això ho instal·la a la carpeta local del teu projecte t'ho deixa a → **node\_modules**
2. Mirem l'api i l'exemple de com fer-ho anar...
3. Fem això:
  - (a) Fem un server
  - (b) Creem una llista amb els readStreams que vulguem posar al zip (**names.map** amb **fs.createReadStream**).
  - (c) Fem una funció que agafi els streams i els empaqueti en un zip (**zip.append**).
  - (d) Penjem un event '**finish**' del **zip** per tancar el response.

---

```
var http = require('http');
var archiver = require('archiver');

// needs to be switched to a database
var fs = require('fs');

function sendFilesInAZip(streams, response) {
  // Create new zip file
  var zip = archiver('zip');

  // Set up some callbacks
  zip.on('finish', function() {
    console.log('Zip file has been sent, with a total of '
      + zip.pointer() + ' bytes. ');
    response.end();
  });

  // Put the result in...
  zip.pipe(response);

  console.log('Setting up zip file contents. ');

  // Queue files to archive
  streams.forEach(function(s) {
    zip.append(s.stream, { name: s.name });
  });

  // Signal end of queue
  zip.finalize();

  console.log('Zip creation has been started. ');
}

function serverFunction(request, response) {
  response.writeHead(200, { 'Content-Type': 'application/zip',
```



```

    'Content-Disposition' : 'attachment; filename=files.zip' });

// pop ids from db using fields from request
var names = ['file1.txt', 'file2.txt'];

var streams = names.map(function(name) {
    var s = {
        name : name,
        stream : fs.createReadStream(name),
    };

    return s;
});

sendFilesInAZip(streams, response);
};
http.createServer(serverFunction).listen(8080);
console.log('Server running at http://127.0.0.1:8080/');

```

---

4. Probar via `wget` i `unzip -t`

## Express

---

```

// NOTA: fer en un arxiu .js i posar les coses en ordre.
// 1-
var express = require('express');

var app = express();

app.listen(8080);

// 2-
app.get('/link.txt', function(req, res) {
    console.log('here');
    res.end();
});

// 3- Subset of regex (no . or -)
app.get('/lin(k|e).txt', ...

// Capturar la regex amb un /:id(regex)/

// 4- perl-like regex
app.get('/[A-z][a-z]*[0-9]$/', ...

// -> fa un match del primer que coincideix...

// 5- Pels parametres d'una query (despres del ?)
app.get('/link', function(req, res) {
    res.end("El ID era: " + req.query.id);
});

// 6- (abans que l'altre get)
app.use('/link', function(req, res, next) {
    console.log(req.query.id);
    next();
});

```

```
// 7- (no explicar)
app.use('/public', express.static(path.join(__dirname, 'public')));

// 8 -
var cookieParser = require('cookie-parser')
app.use(cookieParser())

request.cookies = {name : key}

// 9 - (no explicar)
var bodyParser = require('body-parser')
app.use(bodyParser())

request.body = {name : key}
```

---

### Test de concurrencia

1. Recordem com crear un servidor:

---

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

f = function(request, response) {
    response.writeHead(200);
    response.write("hola!");
    response.end();
}

http.createServer(f).listen(8080);

console.log("Server running!");
```

---

2. Ho provem **class-ff.sh**
3. Ho probem amb un stress test (explicar ApacheBenchmark). Fem una prova:

```
ab -n 100 localhost:8080/
```

-n és el nombre de peticions  
(preguntar com entenen les dades que es mostren)

4. Fem que trigui una mica més:

```
for (var i = 0; i < 1E7; i++) { i * 2 };
```

```
ab -n 100 -c 2 localhost:8080/
```

-c es el nivell de concurrencia (es fan x totes juntes)  
Ojo al time per request: 10.259 ms

5. Ok, anem a remenar una mica més amb un exemple més complicat i de pas repasem nodejs una mica més.

---

```
// funcio que escriu la resposta
var r = function() {
    response.write("Time is " + new Date().toString());
    response.end();
};

// espera activa
var d = new Date();
```

```
while(new Date().getSeconds() == d.getSeconds()) { }  
r();
```

---

**ab -n 10 localhost:8080/** com interpretem els resultats (perquè marca 1000?)

**ab -n 10 -c 2 localhost:8080/** com interpretem els resultats (perquè marca 2000?)

---

*// espera per events*

```
setTimeout(r, 1000 - new Date().getMilliseconds());
```

---

**ab -n 10 localhost:8080/** com interpretem els resultats (perquè marca 1000?)

**ab -n 10 -c 2 localhost:8080/** com interpretem els resultats (perquè marca 1000?)

6. Suposem que tenim el amb espera activa (com si el servidor realment estigues fent alguna cosa de cpu). Aleshores a mesura que incrementa el paral·lelisme el servidor se satura i tothom comença a rebre time outs
7. Que passaria amb un servidor multithread?
8. Que hem de fer? Mantenir el temps de resposta sota control, i respondre 503 si estem sobrecarregats.