Node.js

Parte IV

Javier Miguel

- CTO & Freelance Developer
- Email: jamg44@gmail.com
- Twitter: @javiermiguelg

Consumir APIs de terceros

Consumir APIs

Uno de los módulos más usados para esto es *request*.

https://github.com/request/request

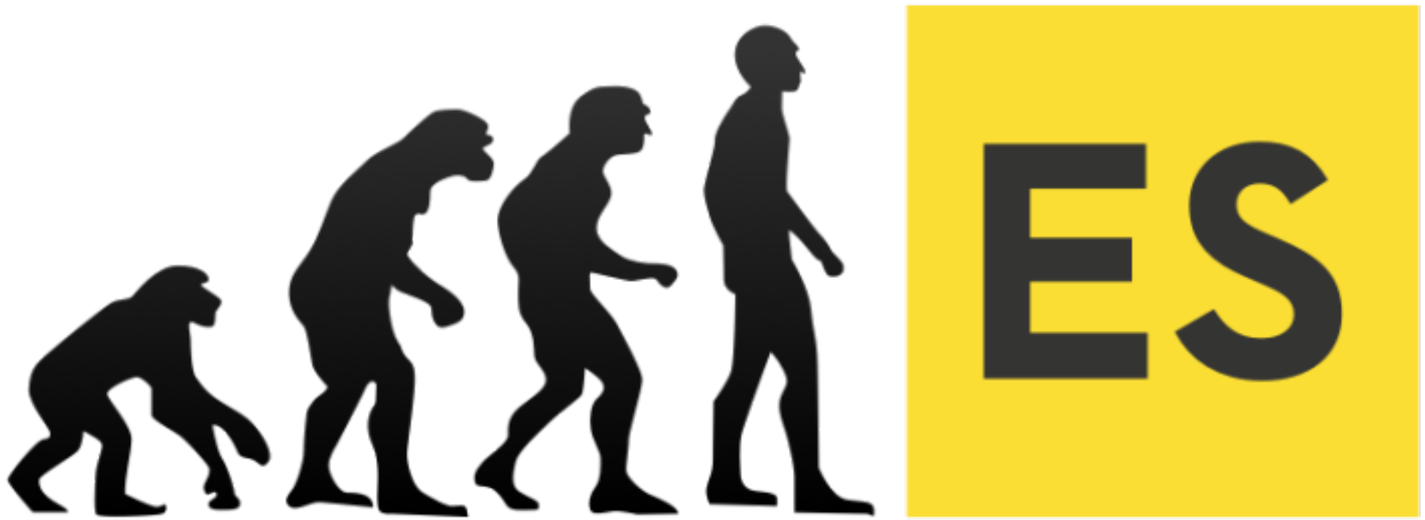
npm install request --save

Consumir APIs

```
var options = {
  method: 'GET',
  url: 'http://api.example.com/emails',
  //headers: {'User-Agent': '...'},
  json: true
};

request(options, function (err, response, body) {
  if (err || response.statusCode >= 400) {
    console.error(err, response.statusCode);
    return;
  }
 // body tendrá nuestro contenido
});
```

diccionario/diccionario-backend



ECMAScript 2015 (ES6) standard

ECMAScript 2015 (ES6)

Desde la versión 4.0 de Node.js

Una de sus principales mejoras es que incluye la librería V8 en la versión v4.5, incluyendo de forma estable muchas características de ES2015 que harán nuestra vida más fácil.

Block scoping, classes, typed arrays, generators, Promises, Symbols, template strings, collections (Map, Set, etc.) and arrow functions.

Callback Hell

```
// esto no es vida
carga(function(err, valor0) {
  valida(valor0, function(err, valor1) {
    procesa(valor1, function(err, valor2) {
      guarda(valor2, function(err, valor3) {
        comprueba(valor3, function(err, valor4) {
          junta(valor4, function(err, valor5) {
            limpia(valor5, function(err, valor6) {
              avisa(valor6, function(err, valor6) {
                res.json({piramide: true});
              });
            });
          });
        });
      });
    });
  });
});
```

```
174
175         async.series([
176             function(callback) {
177                 if (todos == "true") {
178                     models.PromocionesBase.contarTodos(todos, functi
179                         flag = 1;
180                         resultado = result;
181                         procesarSegmento(segmento_id, function(segme
182                             var id = parseInt(segmento_id);
183                             models.segmentacion.find(id).success(fun
184                                 if (segmento) {
185                                     var obj = {
186                                         cp: segmento.cp,
187                                         edad: segmento.edad,
188                                         sexo: segmento.sexo,
189                                         hijo: segmento.hijo,
190                                         favoritos: segmento.favorito
191                                     };
192                                     models.PromocionesBase.contarSim
193                                         if (err) return callback(err
194                                         return callback(null, result
195                                     }
196                                 }
197                             });
198                         }
199                     });
200                 }
201             },
```

Promesas

Promesas

Una promesa es un objeto que representa una operación que aún no se ha completado, pero que se completará más adelante.

Antes de ES2015 podíamos usarlas con librerías, pero estas librerías tienen ligeras (o no tan ligeras) diferencias entre ellas. Ahora ya forman parte del estándar y el lenguaje y no necesitamos estas librerías.

Promesas

Tiene tres estados posibles (https://promisesaplus.com/)

1. Pending
2. Fullfilled(value)
3. Rejected(reason)

2.1. Promise States

A promise must be in one of three states: pending, fulfilled, or rejected.

2.1.1. When pending, a promise:

- 2.1.1.1. may transition to either the fulfilled or rejected state.

2.1.2. When fulfilled, a promise:

- 2.1.2.1. must not transition to any other state.
- 2.1.2.2. must have a value, which must not change.

2.1.3. When rejected, a promise:

- 2.1.3.1. must not transition to any other state.
- 2.1.3.2. must have a reason, which must not change.

Promesas

Cuando una promesa está en uno de los dos estados fulfilled o rejected se le llama settled.

Si la promesa se hubiera cumplimentado (fulfilled) o rechazado (rejected) antes de asignarle un then o catch, cuando se le asignen serán llamados con el resultado o el error.

Promesas

Como se hace

```
var promesa = new Promise(function(resolve, reject) {
  // llamo a resolve con el resultado
  // o llamo a reject con el error
});


promesa.then( function(resultado) {

}).catch( function(error) {

});
```

ejemplos/promise

Promesas

```
promesa.then( function(resultado) {

}).catch( function(error) {

});
```

Es simplemente azúcar sintáctico para la forma:

```
promesa.then(
  function(resultado) { },
  function(error) { }
);
```

ejemplos/promise

Ejercicio

Hacer el comando sleep(milisegundos)

Promesas

Podemos encadenar promesas.

```
promesa1()
.then( promesa2)
.then( promesa3)
.then( function(data) { // final
    console.log(data); })
.catch( function(err) {
    console.log('ERROR', err);
});
```

ejemplos/promise3

Promesas

```
var ingredientes = ['sal','pimienta','conejo','gambas'];

// echar() recibe un string y retorna una promesa
var promisedTexts = ingredientes.map(echar);

Promise.all(promisedTexts)
    .then(function (texts) {
       console.log(texts); // han acabado todas
    })
    .catch(function (reason) {
        // llegaremos aqui con el primero que falle
    });
```

ejemplos/promise4

Promesas

Si Promise.all esperaba a que estuvieran todas cumplidas, Promise.race lo hace cuando cumpla la primera, devolviendo su resultado.

```
Promise.race([p1, p2, p3])
    .then(function (textoDelMasRapido) {
        // llegaremos aqui con el primero que acabe
    })
    .catch(function (reason) {
        // llegaremos aqui con el primero que falle
    });
```

Promesas

El objeto Promise tiene también un par de métodos estáticos que pueden ser útiles:

```
Promise.resolve(valor);
```
Devuelve una promesa resuelta con el valor proporcionado.

```
Promise.reject(razon);
```
Devuelve una promesa resuelta con la razón suministrada. La razón debería ser un error (generalmente una instancia de objeto Error).

Promesas

```
Promise.resolve("bien!").then(function(value) {
  console.log(value); // "Prueba resolve"
}, function(reason) {
  // not called
});

Promise.reject(new Error("chungo...")).then(function(value) {
  // not called
}, function(error) {
  console.log(error); // Stacktrace
});
```

ECMAScript 2015 (ES6) - Block scoping

```
const maxHeight = 200;
let var1 = 5;
```

Solo son visibles en el scope de su **bloque**, no de una función.

No son hoisted!

`const` no puede ser ni re-asignado ni re-declarado dentro de su bloque. Si apunta a un objeto, no puede re-asignarse a otro pero el objeto sigue siendo mutable.

ECMAScript 2015 (ES6) - Clases

```
class Pet {
    constructor(name) {
        this._name = name;
    }
    sayHello() {
        console.log(this._name + ' say hello!');
    }
}

class Cat extends Pet {
    constructor(name) {
        super(name);
    }
    sayHello() {
        super.sayHello();
    }
}
```

```
let pet = new Pet('Toby');
pet.sayHello();

let cat = new Cat('Fritz');
cat.sayHello();
```

ECMAScript 2015 (ES6) - Arrow functions

```
// función normal
const doble = function(a) { return a * 2; }


// función de flecha
const doble = (a) => { return a * 2; }
```

ECMAScript 2015 (ES6) - Arrow functions

```
// si recibimos un solo parámetro podemos omitir paréntesis
const doble = a => { return a * 2; }


// si el cuerpo empieza por return podemos omitir las llaves
const doble = a => a * 2;
```

ECMAScript 2015 (ES6) - Arrow functions

```
// antes
enviarInvitaciones() {
    this._invitados.forEach(function (invitado) {
        this.enviaInvitacion(invitado);
    }.bind(this));
};

// ahora mantiene el this
enviarInvitaciones() {
    this._invitados.forEach((invitado) => {
        this.enviaInvitacion(invitado);
    });
};
```

ECMAScript 2015 (ES6) - Template strings

```
var multilineAnt = 'Loren ipsum\n' +
    'dolor\n' +
    '...';

var multiline =
`Loren ipsum
dolor
...`;

console.log(multilineAnt);
console.log(multiline);
```

ECMAScript 2015 (ES6) - Template strings

```
var nombre = 'Neo';

var template = `Wake up, ${nombre}…`;

console.log(template); // Wake up, Neo...
```

Generadores

ECMAScript 2015 (ES6) - Generators

Crea un iterador que puede ser detenido en un punto con yield, y al volver a llamarlo continuará desde ese punto en el mismo estado.

```
function* rangeCreator (start, end) { // generador
    while (start < end) {
        yield start;
        start += 1;
    }
}

var range = rangeCreator(0, 10); // iterador

console.log(range.next().value); // 0
console.log(range.next().value); // 1
```

ejemplos/generador

ECMAScript 2015 (ES6) - Generators

Al llamar a un generador, este no se ejecuta sino que devuelve un iterador.

Cuando llamamos al método next() del iterador, la función se ejecuta hasta el yield devolviendo su valor en next().value.

ECMAScript 2015 (ES6) - Generators

range.next() devuelve un objeto con dos propiedades: value y done.

```
console.log(range.next()); // { value: 0, done: false }
```

`Value` tendrá lo que ha devuelto yield.
`Done` es un booleano que indica si la función generadora ha llegado a su fin.

ECMAScript 2015 (ES6) - Generators

```
function* rangeCreator(start, end) {
    while (start <= end) {
        yield start;
        start += 10;
    }
    return 'fin';
}
var range = rangeCreator(0, 10);

console.log(range.next()); // { value: 0, done: false }
console.log(range.next()); // { value: 10, done: false }
console.log(range.next()); // { value: 'fin', done: true }
console.log(range.next()); // { value: undefined, done: true }
```

ejemplos/generador

ECMAScript 2015 (ES6) - Generators

Una buena forma de usarlos es con `for of`

```
// usando el iterador
for (let valor of range){
    console.log(valor);
}

// directamente con el generador
for (let valor of rangeCreator(0, 10)){
    console.log(valor);
}
```

* cuidado, `for of` solo iterará lo retornado por `yield`! (`done===false`), por tanto **no** nos dará lo que retorne el generador con `return`.

ejemplos/generador

ECMAScript 2015 (ES6) - Generators

Un generador nos permite manejar de forma síncrona una función asíncrona que retorna callbacks.

```
function run(generatorFn) {
  function sigue(cbValue) {
    iterator.next(cbValue);
  }
  var iterator = generatorFn(sigue);
  iterator.next();
}

run( function*(sigue) {
  var res = yield espera(1000, sigue);
  console.log(res);
});
```

ejemplos/generador2

async / await

ECMAScript 2015 (ES6) - Generators

await consume una promesa

async hace que la función devuelva una promesa.

ECMAScript 2015 (ES6) - async / await

```
async function guarda(registro) {
     let datos = await registro.save(); // <— consume promesa
    return 'fin';
}

// guarda() devuelve una promesa

guarda
  .then(res => console.log(res))
  .catch(err => console.log(err));
```

ejemplos/async-await

ECMAScript 2015 (ES6) --harmony_*

El modificador de linea de comando --harmony nos permite habilitar en Node.js las características de V8 que no están aún en estado estable.

```
$ node --v8-options | grep "in progress"
  --harmony_modules (enable "harmony modules" (in progress))
  --harmony_array_includes (enable "harmony Array.prototype.includes" (in progress))
  --harmony_regexps (enable "harmony regular expression extensions" (in progress))
  --harmony_proxies (enable "harmony proxies" (in progress))
  --harmony_sloppy (enable "harmony features in sloppy mode" (in progress))
  --harmony_unicode_regexps (enable "harmony unicode regexps" (in progress))
  --harmony_reflect (enable "harmony Reflect API" (in progress))
  --harmony_destructuring (enable "harmony destructuring" (in progress))
  --harmony_sharedarraybuffer (enable "harmony sharedarraybuffer" (in progress))
  --harmony_atomics (enable "harmony atomics" (in progress))
  --harmony_new_target (enable "harmony new.target" (in progress))
```

Debugging

Debugging - consola

Debugging - consola

```
//mensaje con marcador para un string, salida en stdout stream
console.log("Hello %s", "World");
//mensaje con marcador para un int, salida en stdout stream
console.log("Number of items: %d", 5);
//mensaje con salida en stdout stream
console.info("Hello Info");
//mensaje con salida en stderr stream
console.error("Hello on Stdout");
//mensaje con salida en stderr stream
console.warn("Hello Warn");
```

Debugging - consola

```
//empieza a contar tiempo
console.time("100mil_elementos");

for(var i=0;i<100000000;i++)
{
    let a = 1;
    a = a * i;
}

//para de contar
console.timeEnd("100mil_elementos");

// 100mil_elementos: 462ms
```

ejemplos/debug

Debugging - consola

```
//Escribe a stderr 'Trace :', seguido de nuestro mensaje
console.trace("Traza");
```

```
Trace: Traza
    at test (/Users/javi/traza.js:21:13)
    at Object.<anonymous> (/Users/javi/traza.js:25:1)
    at Module._compile (module.js:434:26)
    at Object.Module._extensions..js (module.js:452:10)
    at Module.load (module.js:355:32)
    at Function.Module._load (module.js:310:12)
    at Function.Module.runMain (module.js:475:10)
    at startup (node.js:117:18)
    at node.js:951:3
```

ejemplos/debug

Debugging - consola

Si ponemos la instrucción `debugger;` en una linea podremos parar la ejecución ahí. Arrancamos con argumento `debug`:

```
$ node debug prueba.js

< Debugger listening on port 5858
debug> . ok
break in prueba.js:1
> 1 "use strict";
  2
  3 let neo = { name: 'Thomas', age: 33, surname: 'Andreson'};
debug>
```

ejemplos/debug

Debugging - consola

Algunos comandos en debugger:

cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
pause - Pause running code

repl - Entra en modo evaluación (Ctrl+c para salir)
help - Muestra comandos
restart - Re-inicia aplicación
kill - Mata la aplicación
scripts - Muestra scripts cargados

Debugging - WebStorm

Debugging

Ejercicio con WebStorm

```
1  "use strict";
2
3  let neo = { name: 'Thomas', surname: 'Andreson'};
4
5  let test = function(character) {  character: Object {name: "Thomas"}
6      let name = character.name;  character.name: "Thomas"
7      console.log(name);
8  };
9
10 test(neo);
```

Debugging - node inspector

Debugging - node inspector

```
$ npm install -g node-inspector
```

```
$ node-debug app.js
```

Más información en https://github.com/node-inspector/node-inspector

Cluster

Cluster

Desde hace algún tiempo `cluster` es un módulo core de Node.js.

Podemos requerirlo y pedirle que cree nuevos procesos (fork) compartiendo el mismo puerto de escucha.

Esto debemos hacerlo solo al inicio cuando sabemos que somos el master (cluster.isMaster).

Cluster

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

if (cluster.isMaster) {

  // hacer forks creando workers, uno por core lógico
  var numCPUs = require('os').cpus().length;
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

} else {

  // arranque normal de nuestro server

}
```

ejemplos/cluster

Cluster

Esto arrancará n procesos adicionales, a los que el master les ira dando peticiones según lógica round robin.

Cluster emite eventos a los que podemos suscribirnos, como por ejemplo:

- online
- exit
- listening
- disconnect
- message

ejemplos/cluster

Cluster

Los eventos que emite los escucharemos así:

```
// al arrancar un worker...
cluster.on('online', function(worker) {
  console.log('Worker ' + worker.id +
    ' is online with pid ' + worker.process.pid);
});


// al terminar un worker...
cluster.on('exit', function(worker, code, signal) {
  console.log('worker ' + worker.process.pid + ' died');
});
```

ejemplos/cluster

Cluster

Cuando un worker muere emite el evento exit. En este momento podemos reaccionar y crear otro que le sustituya:

```
  // al terminar un worker...
cluster.on('exit', function(worker, code, signal) {
  console.log('Worker ' + worker.process.pid +
    ' died with code: ' + code +
    ', and signal: ' + signal);
  console.log('Starting a new worker');
  cluster.fork();
});
```

ejemplos/cluster

Cluster

El proceso master puede mandar mensajes a los workers.

```
// master
// cluster.workers nos da una lista de los que tenemos
worker.send('hello from the master');

// worker
process.on('message', function(message) {
    console.log(message);
});
```

ejemplos/cluster

Cluster

Los workers pueden mandar mensajes al master.

```
// worker
process.send('hello from worker with id: ' + process.pid);

// master
worker.on('message', function(message) {
    console.log(message);
});
```

ejemplos/cluster

Cluster

Con cluster aumentaremos notablemente el rendimiento de nuestro servidor, usando más recursos del sistema.

Concurrent Connections	1	2	4	8	16
Single Process	654	711	783	776	754
8 Workers	594	1198	2110	3010	3024

* Ejemplo de medición con peticiones por segundo.

Cluster

Es recomendable leer la documentación!

https://nodejs.org/api/cluster.html

Gestores de procesos

En la web de Express.js hay una página con información sobre algunos gestores que usan con Express.

http://expressjs.com/advanced/pm.html