NodeJS

Taller de Nuevas Tecnologías (IF017)

Nahuel Defossé

NodeJS

NodeJS es un **runtime** de ECMAScript/JavaScript orientado a eventos. A diferencia de bibliotecas como EventMachine de Ruby, o Twisted para Python, el bucle de recepción de eventos está incorporado en el núcleo del lenguaje.

[1] https://github.com/v8/v8
[2] https://github.com/libuv/libuv

NodeJS

NodeJS es un **runtime** de ECMAScript/JavaScript orientado a eventos. A diferencia de bibliotecas como EventMachine de Ruby, o Twisted para Python, el bucle de recepción de eventos está incorporado en el núcleo del lenguaje.

Está basado actualmente en el motor de ejecución de JavaScript/ECMAScript de Google denominado **V8** [1] y la biblioteca de programación asincrónica para C llamada **libuv** [2].

[1] https://github.com/v8/v8
[2] https://github.com/libuv/libuv

NodeJS

NodeJS es un **runtime** de ECMAScript/JavaScript orientado a eventos. A diferencia de bibliotecas como EventMachine de Ruby, o Twisted para Python, el bucle de recepción de eventos está incorporado en el núcleo del lenguaje.

Está basado actualmente en el motor de ejecución de JavaScript/ECMAScript de Google denominado **V8** [1] y la biblioteca de programación asincrónica para C llamada **libuv** [2].

Mayoritariamente utilizado en programación *server side*, aunque también utilizado como *runtime* embebido en teléfonos (Cordova, PhoneGap, ReactNative) y aplicaciones de escritorio (Electron), incluso en algunos micro-controladores grandes (ARM Cortex M3+, Intel Galileo).

---

[1] https://github.com/v8/v8

[2] https://github.com/libuv/libuv

Figura1: LibUV, V8, NodeJS

ECMAScript/JavaScript

ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el lenguaje JavaScript propuesto como estándar por Netscape Communications, antecesora del actual navegador Mozilla Firefox.

ECMAScript/JavaScript

ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el lenguaje JavaScript propuesto como estándar por Netscape Communications, antecesora del actual navegador Mozilla Firefox.

Cada versión de NodeJS implementa una versión de ECMAScript (ES) que puede ser distinta a la de los navegadores de los clientes. Esto genera un problema que se soluciona con la **transpilación**, generalmente los transpiladores son paquetes creados con NodeJS.

Versiones de ECMAScript/JavaScript

- 1.0 a 1.4 Desde 1996 a 1999, implementaciones propias de cada navegador. Varias denominaciones JScript en Microsoft, JavaScript en Netscape.
- 1.5 Estándar ECMA-262
- 1.6 En Diciembre de 2005, agrega métodos a `array`, `for each`, métodos de `String` en cualquier objeto, mejor acceso al DOM, mediante `E4X`.
- 1.7 Octubre de 2006, agrega generadores Pythónicos, iteradores y `let`

Versiones recientes de ES (ECMAScript)

1.8 En Junio de 2008, agrega expresiones generadoras y expresiones de clausura.

1.8.1 Agrega soporte de `JSON` nativo

1.8.5 En 2010, edición ratificada como la ES 5ta edición. Agrega modo estricto, propiedades o *accesors* (`get` y `set`). Las comas de más no molestan.
Meta-programación agregando métodos a `Object` (introspección, reflexión, etc.), métodos funcionales a `Array`.

Ver detalle en sitio de *Speaking JS*

ES6

Versión más reciente:

- Constantes `const`
- Funciones con sintaxis flecha `arr.map(v => v + 1)`.
- Valores por defecto para argumentos y agrupamiento `function x(a=1, b=2, ...resto)`.
- Operador *repartir* `var params = [ "hello", true, 7 ]; var other = [ 1, 2, ...params ])`.
- Interpolación de cadenas con `${}` y cadenas largas

```
let x = `
largo
`
```

Ver más en *ECMAScript 6 — New Features: Overview & Comparison*

Gestión de proyectos con NodeJS

En las instalaciones actuales de NodeJS, se instala la utilidad de línea de comandos `npm`. Esta se encarga de instalar paquetes de software de terceros, gestionar proyectos y proveer un lanzador de scripts sencillo, entre otras funcionalidades.

docker run --rm -ti node npm

```
Usage: npm <command>

where <command> is one of:
    access, adduser, bin, bugs, c, cache, completion, confi
    ddp, dedupe, deprecate, dist-tag, docs, doctor, edit,
    explore, get, help, help-search, i, init, install,
    install-test, it, link, list, ln, login, logout, ls,
    outdated, owner, pack, ping, prefix, profile, prune,
    publish, rb, rebuild, repo, restart, root, run, run-sc
    ...

npm <command> -h     quick help on <command>
npm -l               display full usage info
Specify configs in the ini-formatted file:
    /root/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config
```

Comandos de npm

`init`	Crea el proyecto, generando un archivo `package.json`
`install`	Instala paquete (si se usa `--save`) se agrega el `package.json` si existiese. Si se usa `-g` se hace global al sistema, esto debe hacerse con cuidado.
`search`	Busca paquetes en el repositorio de https://www.npmjs.org
`ls`	Lista los paquetes instalados (también acepta `-g`)
`update`	Actualiza dependencias

Ver otros comandos en *documentación oficial de npm*

Estructura de un proyecto

```
package.json
package-lock.json # Generado por npm
                  # si hay cambios peligrosos
index.js
node_modules/
  accepts
  array-flatten
  body-parser
  bytes
  ...
```

Estructura de un proyecto - `package.json`

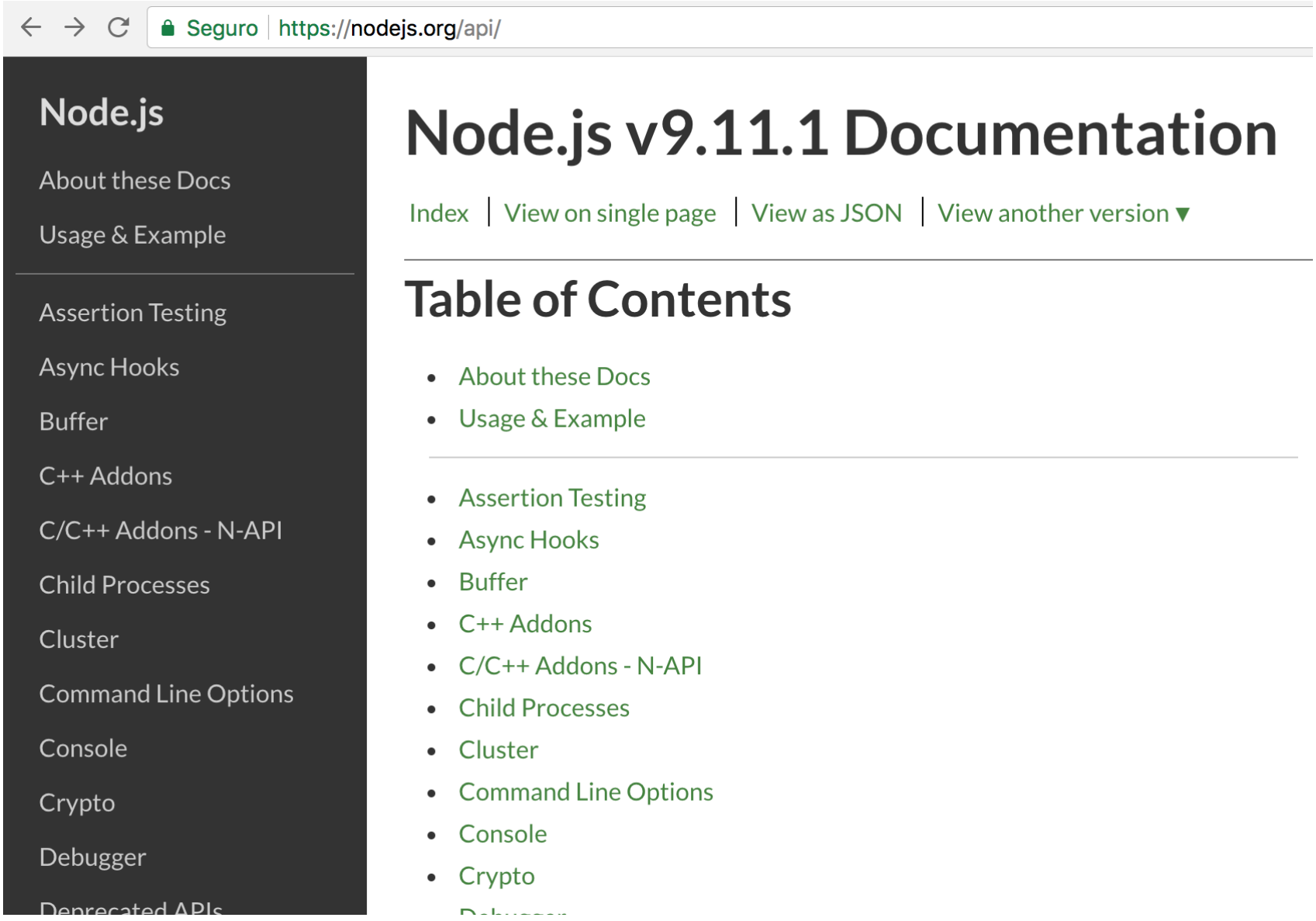
Este es el archivo que define:

- nombre
- versión
- autores y licencia de distribución
- puntos de entrada (o scripts)
- dependencias

Ejemplo:

```
{
  "name": "mi-super-paquete",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
  }
}
```

Biblioteca estándar o API



Ejemplos de la API - Archivos

Utilización del módulo `fs` para lectura de un archivo desde disco:

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

fs.open('/open/some/file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

Ejemplos de la API - Servidor HTTP

Creación de un servidor con el paquete `http` de NodeJS.

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hola mundo!\n');
});

server.listen(port, hostname, () => {
  console.log(`Acceda a http://${hostname}:${port}/`);
});
```

Depuración

Node.js incluye una utilidad de depuración externa accesible via el Inspector de V8

```
$ node inspect myscript.js
< Debugger listening on ws://127.0.0.1:9229/80e7a814-7cd3-4
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in myscript.js:1
> 1 (function (exports, require, module, __filename, __dirn
  2 setTimeout(() => {
  3   console.log('world');
debug>
```

Al igual que en un *browser*, `console.log()` puede ser escribir a la salida standard.

Definición de Módulos

NodeJS respeta la especificación de ES6, sin embargo soporta varios mecanismos:

Ej:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.a
```

Con una definición de `./circle.js` como:

```
const { PI } = Math;

exports.area = (r) => PI * r ** 2;

exports.circumference = (r) => 2 * PI * r;
```

Nota Para podes importar una carpeta como módulo se debe definir un `package.json` en esta.

Ver más en *Documentación sobre módulos*.

Programación Asincrónica con `async` y `await`

A partir de NodeJS 8 (LTS), se incorporan los *keywords* `async` y `await`, sumándose (que ya habían tenido éxito en C# y fueron implementados en Python también):

Simplifican el código dónde se debían utilizar promesas por la naturaleza asincrónica del lenguaje:

```
const makeRequest = () =>
  getJSON()
    .then(data => {
      console.log(data)
      return "done"
    })

makeRequest()
```

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

makeRequest()
```

Mejora en el manejo de errores

Con promesas:

```
const makeRequest = () => {
  try {
    getJSON()
      .then(result => {
        // this parse may fail
        const data = JSON.parse(result)
        console.log(data)
      })
      // captura explícita de errores...
      // .catch((err) => {
      //   console.log(err)
      // })
  } catch (err) {
    console.log(err)
  }
}
```

Mejora en el manejo de errores (cont.)

Con async:

```
const makeRequest = async () => {
  try {
    // this parse may fail
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
```

Mejora en la lógica condicional

Con promesas:

```
const makeRequest = () => {
  return getJSON()
    .then(data => {
      if (data.needsAnotherRequest) {
        return makeAnotherRequest(data)
          .then(moreData => {
            console.log(moreData)
            return moreData
          })
      } else {
        console.log(data)
        return data
      }
    })
}
```

Mejora en la lógica condicional (cont.)

Con `await`:

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
    return data
  }
}
```

Mejoras en la obtención de valore intermedios

Ejemplo con promesas anidadas:

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      // hacer algo
      return promise2(value1)
        .then(value2 => {
          // hacer algo
          return promise3(value1, value2)
        })
    })
}
```

Mejoras en la obtención de valore intermedios (cont.)

Ejemplo con `await`

```
const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
}
```

Además simplifica la depuración (permite insertar puntos de ruptura en `await`) y los *tracebacks* son más concisos.