Coding Bootcamp

Sprint 2

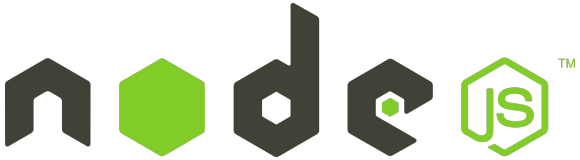
Temario

Temario

- ¿Qué es Node.js?
- Instalación.
- Hola Mundo.
- ¿Por qué Node.js?
- Módulos de Node.
- Ejemplos de uso.
- npm.
- ES+ y JavaScript específico de Node.

¿Qué es Node.js?

¿Qué es Node.js? (1/3)



"Es un runtime que permite ejecutar JavaScript fuera de los navegadores".

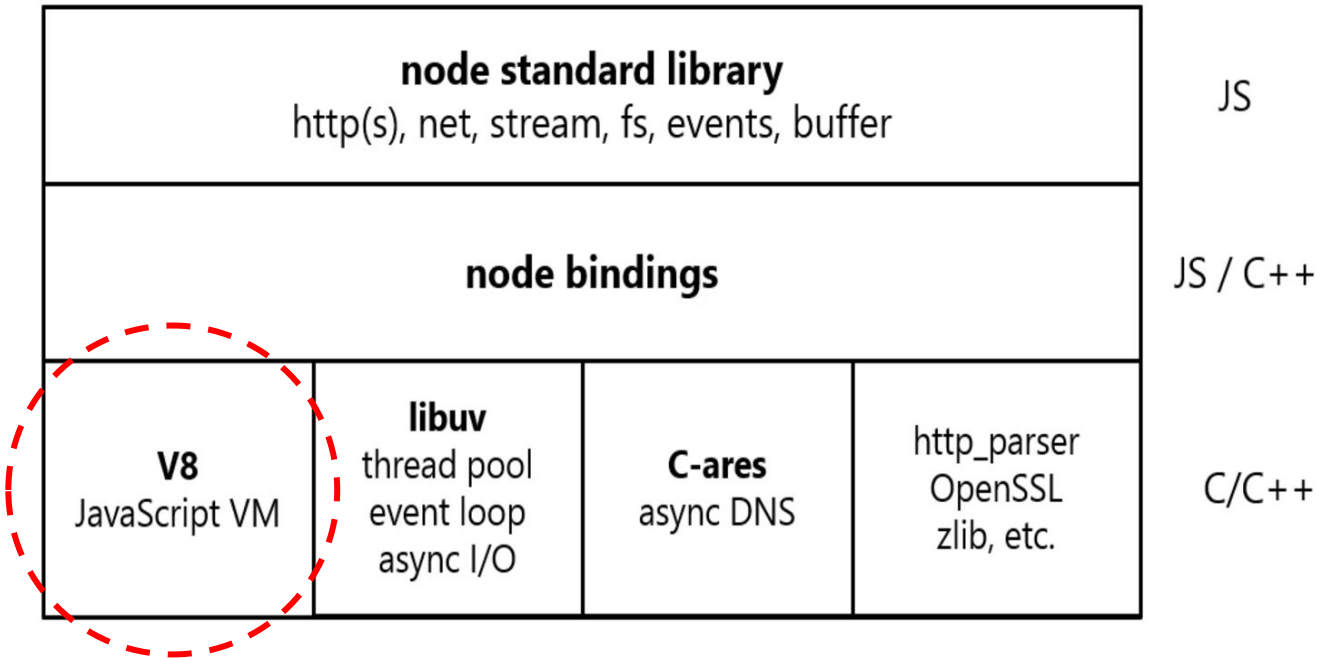
¿Qué es Node.js? (2/3)

Node.js es básicamente un runtime (construído sobre el motor de JavaScript V8 de Google Chrome) que permite ejecutar JavaScript fuera de los navegadores.

Es una herramienta que permite construir servidores web y aplicaciones de red escalables y asincrónicas.

Está escrito en C++ (mayormente) y JavaScript.

¿Qué es Node.js? (3/3) – Arquitectura



Fuente: Amazing Features of Node.js that makes it in top 5 Server Side Scripts
Disponible es: http://www.justtotaltech.co.uk/blog/amazing-features-node-js-top-5-server-side-scripts.

Instalación de Node

Instalación

Deberán instalar Node.js de https://nodejs.org.

La instalación incluye el gestor de dependencias npm – https://www.npmjs.com, aunque luego, si lo desean, podrán usar otro gestor de dependencias como Yarn.
👉 Ya hablaremos en más en detalle sobre esto.

Hola Mundo (básico)

Hola Mundo (básico)

En un archivo `index.js` escribir:

```
console.log("Hola Mundo");
```

Luego, en una terminal ejecutar:

```
node index.js
```

En la terminal aparecerá el texto "Hola Mundo".

Aclaración

Vale la pena aclarar que el código JavaScript que escribimos en Node.js corre en un servidor, no en un navegador.

Por lo tanto, no están disponibles los objetos `document` ni `window`, y por lo tanto no se pueden escribir código como:

```
const parrafo = document.querySelector("p");
```

```
window.alert("¡Hola Mundo!");
```

Por otro lado, con Node.js se podrá escribir código que no puede correr un navegador, como el que permite acceder al File System de una computadora.

Ejecutar JavaScript directamente en la terminal

Ejecutar JavaScript directamente en la terminal

Si en una terminal escriben solamente “node” y le dan Enter, les aparecerá una línea de comandos donde podrán ingresar y ejecutar código JavaScript desde ahí mismo, muy similar a la consola de un navegador.

Para salir deberán escribir ".exit" o sino CTRL + C.

¿Por qué Node.js?

¿Por qué Node.js? (1/4)

“I/O debe hacerse diferente. Lo hemos estado haciendo mal durante años”.

Ryan Dahl, 2009.

¿Por qué Node.js? (2/4)

Antiguamente, los servidores web funcionaban haciendo consultas similares al siguiente ejemplo:

1. Consultar una base de datos.
   Esperar por el resultado; la ejecución del programa queda congelada.
2. Recibir un resultado de la base de datos.
3. Usar el resultado obtenido.

```
var result = db.query("SELECT * FROM..."); // Zzzzz.
// Se usa el resultado.
```

¿Por qué Node.js? (3/4) – Event Driven

Volviendo al ejemplo de la consulta a la base de datos, Node.js propone resolverlo utilizando un sólo *thread* (hilo) y un *event loop*. Los pasos a seguir serían:

1. Consultar la base de datos.

2. No esperar por el resultado, pero “estar atento” a que el mismo esté pronto. Mientras tanto, realizar otras tareas.

3. Cuando la base datos tenga pronto el resultado y lo retorne (en el momento que sea), hacer algo con el mismo.

¿Por qué Node.js? (4/4) – Event Driven

El ejemplo anterior en Node.js *sería* algo así:

```
db.query("SELECT * FROM...", function() {
    // Se usa el resultado.
});
```

👉 También pueden ver este video que explica muy bien porqué usar Node.js.

Módulos de Node

Módulos de Node

Luego usaremos **npm** para instalar otros módulos (externos).

Node cuenta con varios “módulos” (especie de librerías JavaScript) integrados que permiten realizar distintas tareas como, por ejemplo:

- Acceder al filesystem (disco duro). Esto permite crear, modificar y eliminar archivos y carpetas.
- Crear un servidor http.
- Gestionar eventos.
- Parsear urls.
- Utilizar funciones criptográficas.
- Etc. La lista completa se puede acceder aquí: https://nodejs.org/api.

Hola Mundo (vía http)

Hola Mundo (vía http) (1/2)

A continuación se hará uso del módulo `http`, el cual permite crear un pequeño servidor que estará escuchando peticiones en el puerto 3000.

En un archivo `index.js`:

`require` es una función de Node que se utiliza para importar módulos.

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

const server = http.createServer(function(req, res) {
    res.end("Hola Mundo");
});

server.listen(3000);
```

Hola Mundo (vía http) (2/2)

Luego, en una terminal ejecutar:

```
node index.js
```

Notar que la terminal quedará como “paralizada”. Esto no es un error, es simplemente el servidor que está corriendo, esperando por peticiones. Si se cierra la terminal, se cerrará el servidor.

👉 Finalmente, entrar a través de un navegador a http://localhost:3000.

Event Loop

Event Loop

El Event Loop es lo que permite que Node.js utilice un único *thread* (hilo). Para entenderlo, conviene primero entender el paradigma de programación dirigida por eventos (*Event-Driven Programming*), que hace años viene siendo utilizado en programación de Front-End.

El *event-driven programming* es paradigma de programación orientado a eventos. Ejemplos de este tipo de programación: Realizar una tarea cuando se obtuvo un registro de una base de datos, se completó una petición (*request*), o se leyó un archivo del *file-system*.

Un entorno como el de Node permite mediante un mecanismo central que se encuentra permanentemente escuchando eventos, llamar funciones (llamadas *"callback"*) cuando el evento es detectado.

Entonces, ¿qué es Node.js?

Entonces, ¿qué es Node.js?

"Node.js es una infraestructura no-bloqueante y orientada a eventos para desarrollar programas con alta concurrencia."

Los programadores que utilicen Node.js pueden despreocuparse de la ocurrencia de deadlocks.

Ejemplo 1

Ejemplo 1 (1/4)

A continuación se hará el ejemplo del “Hola Mundo!” (que se vio anteriormente) y se lo analizará por partes.

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

const server = http.createServer((req, res) => {
    res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
    console.log("Servidor escuchando en http://localhost:8080.");
});
```

Esta es una nueva forma de definir funciones, llamada: **Arrow Functions**. Ver más aquí.

Ejemplo 1 (2/4)

Primero se importa el módulo integrado `http`, y se lo asigna a una variable con el mismo nombre (aunque podría ser otro).

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

const server = http.createServer((req, res) => {
    res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
    console.log("Servidor escuchando en http://localhost:8080.");
});
```

Ejemplo 1 (3/4)

Luego se “instancia” un nuevo servidor HTTP pasándole una función *callback* que contestará a cada *request* el mensaje “Hola Mundo”.

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

const server = http.createServer((req, res) => {
    res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
    console.log("Servidor escuchando en http://localhost:8080.");
});
```

Ejemplo 1 (4/4)

Finalmente se invoca la función `listen()` pasándole un puerto y una función *callback* que se ejecutará una vez que el servidor esté listo para recibir y contestar *requests*.

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

const server = http.createServer((req, res) => {
    res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
    console.log("Servidor escuchando en http://localhost:8080.");
});
```

Ejemplo 2

Ejemplo 2 (1/2)

Se reutilizará el siguiente código (del ejemplo anterior):

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

const server = http.createServer((req, res) => {
    // Código que se ejecutará cuando el servidor reciba un request.
});

server.listen(8081, () => {
    console.log("Servidor escuchando en http://localhost:8081.");
});
```

Ejemplo 2 (2/2)

Importar el módulo File System `fs` y utilizar la función `fs.writeFile` para crear o recrear un archivo de texto toda vez que el servidor recibe una requisición HTTP.

```
var http = require("http");
var fs = require("fs");

var server = http.createServer((req, res) => {
    fs.writeFile("archivo.txt", "Hola Mundo!\n", (err) => {
        if (err) return res.end("Ha ocurrido un error.");
        res.end("Se ha creado un archivo!\n");
    });
});

server.listen(8081, () => {
    console.log("Servidor escuchando en http://localhost:8081.");
});
```

npm

npm (1/5)

Es equivalente a **Composer** en PHP o **NuGet** en .NET.

- npm es el gestor (o manejador) de paquetes que viene con Node.
- Tiene un registro público donde desarrolladores pueden publicar paquetes, para que el resto de la comunidad pueda usarlos.
- Además, incluye una cli (command-line interface) que permite usar npm desde la consola para instalar y publicar dichos paquetes.
- npm Inc. es una empresa privada.
- La palabra npm de por sí, no es una sigla y no obedece a nada. Aunque, intuitivamente, podemos pensar en *node package manager.*
- Link: https://www.npmjs.com.

npm (2/5)

- npm también sirve para dar comienzo a un proyecto de Node.
- Esto se hace mediante el comando: `npm init`.
- Al hacerlo, el cli nos hará algunas preguntas para facilitar la creación del proyecto, aquellas que no queremos responder o que no sabemos cómo, simplemente las ignoramos.
- Al terminar, tendremos un nuevo archivo, llamado `package.json`.

npm (3/5) – package.json

El archivo `package.json` contendrá toda la información necesaria sobre nuestro proyecto como ser nombre, versión, descripción, autores, contribuidores, licencia, *punto de entrada*, y principalmente: dependencias y scripts.

Aquí un ejemplo de un archivo `package.json` creado con el comando `npm init`:

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

npm (4/5)

- Se utilizará npm para instalar dependencias.
- Una dependencia es un módulo externo a nuestra aplicación la cual sólo funcionará en tanto dicho módulo este instalado. En nuestro caso, éstos módulos externos provienen del registro público de npm.
- Las dependencias en los proyectos Node se instalan en un directorio, ubicado en la raíz del proyecto, llamado `node_modules`.
- El directorio `node_modules` no debe ser editado jamás por nosotros. La estructura de carpetas y los archivos que hay allí dentro sólo deben ser manejados por npm.

npm (5/5)

Para instalar dependencias, tenemos que estar ubicados en un proyecto de Node ya iniciado (es decir, que fue creado previamente usando el comando `npm init`) y luego ejecutar el comando:

```
npm install <nombre_del_modulo_externo>
```

Este comando consiste de:

- Invocar `npm`.
- Comando `install` (o simplemente `i`).
- Nombre del módulo externo (dependencia) que se desea instalar, tal como aparece en npmjs.com.

👉 Este comando instalará la dependencia en el directorio `node_modules` y agregará una entrada en el archivo `package.json`.

ES+ y JavaScript específico de Node

ES+ y JavaScript específico de Node (1/8)

Anteriormente se usaron los *módulos* `http` y `fs`. Se accedió a los mismos usando la función `require`, que es específica de Node.

En un entorno Node, la función `require` siempre estará disponible globalmente. Sirve para importar objetos, funciones, variables y otros elementos desde:

- Módulos de Node (cómo `http`, `fs` y otros que iremos viendo a lo largo del curso).
- Otros archivos nuestros.
- Dependencias externas, que vamos a manejar con npm.

Nota: Técnicamente hablando, la función `require` es local al módulo que está haciendo la importación, aunque la experiencia para el desarrollador es como si se tratase de una función global. Ver más información aquí.

ES+ y JavaScript específico de Node (2/8)

A diferencia de `require`, que es *específico de Node*, recientemente (2015) fue incorporada la sintaxis `import`. La sintaxis de `import` es propia del lenguaje JavaScript, es decir, *no específica* de Node.

En términos generales, cumple el mismo objetivo de `require`: incorporar en donde se usa, funciones, variables y objetos de otros archivos, módulos del contexto (Node en nuestro caso) y/o dependencias externas.

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

o

```
import modulo from 'modulo';
```

En todos los casos la extensión de los archivos se da por entendida (`.js`) y se omite.

Aquí se está definiendo una variable con la keyword `const`. Ver más aquí.

ES+ y JavaScript específico de Node (3/8)

Para importar módulos de Node, simplemente se elige el módulo que se quiere utilizar y se lo *requiere* o *importa*.

Ejemplo con `http`.

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

o

```
import http from 'http';
```

👉Esta sintaxis de ES6 puede no funcionar en Node.js por defecto: https://nodejs.org/api/esm.html

ES+ y JavaScript específico de Node (4/8)

Para importar módulos que están disponibles en otros archivos nuestros, debemos indicar tanto a `require` como a `import` la ruta (*path* en inglés) en donde se encuentra el mismo, relativa a la posición del archivo actual.

```
// Para indicar que el archivo está en el mismo directorio, se usa "./".
const nuestroModulo = require('./nuestro-modulo');
// o
import nuestroModulo from './nuestro-modulo';
```

👉Esta sintaxis de ES6 puede no funcionar en Node.js por defecto: https://nodejs.org/api/esm.html

ES+ y JavaScript específico de Node (5/8)

Para indicar que el archivo está un directorio “antes” (o “más afuera”) que éste archivo, se usa `"../"`.

```
const nuestroModulo = require('../nuestro-modulo');

// o

import nuestroModulo from '../nuestro-modulo';
```

👉Esta sintaxis de ES6 puede no funcionar en Node.js por defecto: https://nodejs.org/api/esm.html

ES+ y JavaScript específico de Node (6/8)

Para indicar que el archivo está un directorio “después” (o “más adentro”) que éste archivo, se usa `"./directorio/"`.

```
const nuestroModulo = require('./directorio/nuestro-modulo');

// o

import nuestroModulo from './directorio/nuestro-modulo';
```

👉Esta sintaxis de ES6 puede no funcionar en Node.js por defecto: https://nodejs.org/api/esm.html

ES+ y JavaScript específico de Node (7/8)

Siempre que se importen/requieran módulos *nuestros*, es necesario que dichos módulos estén *exportados* previamente. Esto se hace así:

```
// Dado un módulo, que bien puede ser una función, un objeto, una variable, o cualquier cosa...
const nuestroModulo = function() {};


// Usando la sintaxis propietaria a Node, lo exportamos así...
module.exports = nuestroModulo;


// O, usando la sintaxis standard de JavaScript...
export default nuestroModulo;
```

👉Esta sintaxis de ES6 puede no funcionar en Node.js por defecto: https://nodejs.org/api/esm.html

ES+ y JavaScript específico de Node (8/8)

Para importar módulos que se obtuvieron con npm, al igual que con los módulos de Node (como `http` y `fs`), no es necesario un prefijo de ruta, simplemente se escribe el nombre del paquete.

Ej: el framework *Express* no viene instalado por defecto con Node. Se debe instalar vía npm. Luego se importa en nuestro proyecto de la siguiente manera:

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

o

```
import express from 'express';
```

👉Esta sintaxis de ES6 puede no funcionar en Node.js por defecto: https://nodejs.org/api/esm.html

En resumen...

En resumen...

"`npm` permite instalar dependencias, las cuales se persisten como tales en `package.json` y en `node_modules`. Luego se utilizan en nuestro código importándolas con la función `require`"