# Node.js + Express

# Javascript del lado del Servidor

## ¿Qué es Node?

Es una librería de Javascript que nos permite ejecutar código desde el lado del servidor (backend). Es dirigido por eventos de manera asíncrona y se ejecuta sobre el intérprete de JavaScript creado por Google, V8.

Nos permite, entre otras cosas:

* Acceder al sistema operativo del equipo
* Comunicar al servidor con el cliente a través de Sockets
* Manejar servicios Rest
* Sistema de Archivos
* Manejar Procesos

La fuerza de node reside en su arquitectura dirigida a eventos Non Blocking (No bloqueantes), lo que permite optimizar el performance de nuestra aplicación de manera significativa.

Antes de comenzar con Node, es necesario reforzar algunos conceptos

## ¿Qué es Javascript?

Si nos remitimos a Wiki, podemos ver su definición como:

*Un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.*

El hecho de ser un lenguaje basado en prototipos es lo que les dá las características subsiguientes, debido a que podemos acceder y modificar todo el core y funcionalidades del lenguaje de una manera sencilla, convirtiéndose en un lenguaje sumamente flexible a la hora de desarrollar.

Se utiliza principalmente en su forma del lado del cliente (client-side), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas4​ aunque existe una forma de JavaScript del lado del servidor(Server-side JavaScript o SSJS). Su uso en aplicaciones externas a la web, por ejemplo en documentos PDF, aplicaciones de escritorio (mayoritariamente widgets) es también significativo.

Es un lenguaje de guiones (scripts) interpretado (no compilado).El código de un programa fuente JavaScript se inserta directamente en el documento HTML y no necesita ser compilado, es el propio navegador el que se encarga de ir interpretando dicho código.

### Programando en Javascript

#### Clonar Repositorio

Para comenzar a internalizarse en los conceptos principales de Javascript, lo invito a clonar el siguiente repositorio:

<https://github.com/PailletJuanPablo/mmCurso>

Para clonarlo, es necesario tener instalado Git en su sistema (Ver guía de unidad 1), abrir su terminal en el directorio donde desea guardar los archivos y ejecutar el siguiente comando:

***git clone*** [***https://github.com/PailletJuanPablo/mmCurso.git***](https://github.com/PailletJuanPablo/mmCurso.git)

Una vez que el repositorio haya sido clonado exitosamente, dispondrá de carpetas con materiales de la clase 1 y 2. La que nos servirá actualmente es la número **3: Node y Express.**

Una vez dentro de la carpeta, ingrese a la carpeta Fundamentos, donde encontrará diferentes archivos referidos al lenguaje.

### Variables en Javascript

*(Abrir el archivo variables.js)*

Una variable es un espacio reservado de memoria que contiene información determinada.

En javascript, una variable se declara con la siguiente sintáxis:

var nombreDeLaVariable = ValorDeLaVariable

La palabra *var* es una palabra reservada (propia del lenguaje y que no puede suplantarse. Con la misma se inicializa la creación de una variable. Luego de la palabra reservada *var* escribimos el nombre de la variable, para introducir su valor (con la palabra igual precediendo).

Las variables pueden tener diferentes tipos, siendo los principales (Como verán en el archivo):

**Números**: Variables para almacenar números enteros o decimales, indistintamente.

// Números

var numero = 2;

// Para decimales no utilizamos coma, sino punto

var numeroDecimal = 1122.32;

**Cadenas**: Variables para almacenar datos alfanuméricos (texto o números). Es importante remarcar que si almacenamos un número en una string, el mismo no podrá ser tomado como un elemento matemático, por lo que no podremos realizar cálculos con el mismo.

// Strings

var cadena = "Hola";

**Booleanos**: En el mismo almacenamos un valor que puede ser true (verdadero) o false (falso). Estos valores nos permiten realizar diferentes validaciones.

// Booleanos

var booleano = true || false;

**Arrays**: Grupos de Variables. Nos permiten agrupar en una variable múltiples variables, útiles para optimizar el consumo de memoria de nuestra aplicación como así también agrupar elementos similares.

// Arrays - Grupo de Variables

var colores = new Array("rojo","azul","verde");

**Objetos**: Toda variable en javascript es un objeto. La diferencia de utilizar la variable objeto es que podemos especificar las características de los diferentes elementos que compondrán al mismo. Es similar al Array, ya que estamos almacenando un grupo de variables, con la diferencia que en el mismo podemos almacenar también funciones y elementos con diferentes atributos.

// Objects - Elementos que tienen diferentes atributos

var objeto2 = {

propiedad1: "Rojo",

propiedad2: "ff1123"

}

### Funciones en Javascript

(Ver archivo funciones.js)

Las funciones son un bloque de código que nos permite almacenar instrucciones de una manera determinada.

Para declarar una función, utilizamos también una palabra reservada:

// Funciones

function nombreFuncion(parametros){

instrucciones de la función

}

Utilizando function, podemos comenzar a crear una función. Posteriormente introducimos el nombre de la función y los parámetros de la misma entre paréntesis (si la función no recibe parámetros los paréntesis se dejan en blanco). Posteriormente, abrimos corchetes y especificamos las diferentes instrucciones que serán ejecutadas con la función:

function saludar() {

console.log("Hola!")

}

// Para llamar a la función:

saludar();

En este ejemplo, creamos una función saludar, que ejecutará como instrucciones realizar un console.log(impresión en consola) de la cadena hola.

Para ser llamada, una vez que declaramos la función, escribimos su nombre y los parámetros de la misma (dejando en blanco si esta no posee parámetros).

// Pueden recibir parametros

function saludarConParametro(nombre ) {

console.log("Hola "+ nombre + "!")

}

// Podemos llamar con o sin parametros

saludarConParametro("Juan");

Condiciones

(ver archivo condiciones.js)

Las condiciones son un tipo de función reservado que obligatoriamente recibirá como parámetro un boleano (true or false) y ejecutará instrucciones de acuerdo al resultado obtenido.

Se declaran de la siguiente manera:

if (condición){

código a ejecutar si la condición se cumple

} else {

código a ejecutar si la condición no se cumple

}

Con la palabra reservada if, seguida de la evaluación que se hará entre parentésis, especificamos las instrucciones a seguir en el caso de cada validación.

En un mismo bloque if, podemos concatenar múltiples validaciones, por ejemplo:

// Pueden concatenarse muchas condiciones

if (numero1 > numero2) {

console.log("Numero 1 es mayor!");

} else if (numero1 == numero2) {

console.log("Numero 1 y Numero 2 son iguales!");

} else {

console.log("Numero 2 es mayor!");

}

En este caso, primero se verificará si número 1 es mayor que número 2. Si esta condición no se cumple se verificará nuevamente si estos números son iguales, y si esta segunda condición no se cumple, se ejecutará una tercera (podríamos continuar realizando tantas validaciones como queramos).

Sin embargo, para poder escribir múltiples validaciones de una forma más estructurada, podemos utilizar el Switch:

**Switch**

Con la función reservada Switch podemos realizar múltiples validaciones de una manera más estructurada. Su sintáxis es la siguiente:

switch (variableEvaluar){

  case valor:

  // Lo que deseemos

  return;

  case otrValor:

  //Lo que deseeemos

  return

}

Utilizando la palabra switch, abrimos paréntesis y especificamos la variable a ser evaluada. Posteriormente, dentro de los corchetes, podemos especificar los diferentes cases (valores) que podría tener la variable en particular. Nótese que, a diferencia del if, en el mismo podemos introducir cualquier valor para los diferentes cases, no solamente true or false.

var dia = "L";

switch (dia) {

case "L":

console.log("Es lunes!");

break;

case "M":

console.log("Es martes!");

break;

}

En este ejemplo declaramos una variable alfanumérica día. Posteriormente, comenzamos un switch que irá variando de acuerdo al valor que la misma tenga. Para el caso del ejemplo, el resultado sería el console.log de “Es Lunes”. Si la variable dia fuera igual a “M”, el comportamiento sería otro, de acuerdo a los pasos especificados en la función.

### Es6

EcmaScript6 (ES6) es el estándar actual de desarrollo en Javascript. Tiene como finalidad corregir diferentes limitaciones en la lógica, como así también volver más sólido el lenguaje:

Principales novedades:

#### Variables Let y Const

(Abrir archivo letvar.js)

Con let definimos una variable que solo pertenecerá a un scope determinado. Si es asignada fuera del ámbito de una función, pertenecerá al scope global. El Scope, se refiere al conjunto de variables y funciones al que tenemos acceso en un contexto determinado de nuestro código.

function saludar() {

// Si es asignada dentro de una función, pertenecerá solamente al scope de la misma

let nombre = "Jose";

console.log(nombre);

}

En este ejemplo, al crear una variable con let dentro del scope de la función saludar (lo que esta entre corchetes) la variable solo existirá dentro de la función. Si, por el contrario, utilizáramos la palabra var, la misma existiría en todo nuestro código.

También podemos utilizar las variables const.

Este tipo de variables se comporta de manera igual a let, pero con la diferencia de que las mismas no pueden ser modificadas.

const nombre = "Juan"

Esta variable, tendrá como diferencia que su valor no podrá ser alterado en nuestro código.

#### Funciones de Flecha (arrow functions)

Abrir archivo arrow.js

La expresión de función flecha tiene una sintaxis más corta que una expresión de función convencional y no vincula sus propios this, arguments, super, o new.target. Las funciones flecha siempre son anónimas. Estas funciones son funciones no relacionadas con métodos y no pueden ser usadas como constructores.

*This: Estado que almacena el elemento superior donde se ejecuta nuestro bloque de código*

*La sintáxis de una función de flecha es la siguiente:*

nombreFuncion = (parametros) => {

instrucciones

}

Luego de escribir el nombre que tendrá la función, abrimos el signo igual y escribimos los parámetros entre paréntesis, luego abrimos una especie de flecha con el igual y el signo mayor a (por eso se las llama funciones de flecha) para especificar, entre corchetes, las diferentes instrucciones.

Por ejemplo:

getNombre = () => {

console.log("Juan")

}

En este caso, creamos la función getNombre, con la sintáxis de flecha.

getNombre = (nombre) => {

console.log("Hola " + nombre)

}

Las funciones de flecha también pueden recibir parámetros para ser ejecutados en la misma.

#### Callbacks

(ver callback.js)

Un callback es un objeto que es devuelto cuando una función finaliza. Nos permiten manipular datos de una manera estructurada y asíncrona, para optimizar nuestro código y mantener una lógica fácil de entender.

Para crear una función que devuelva un callback, podemos utilizar la sintáxis de flecha, agregando la palabra reservada callback entre sus parámetros.

// Definimos la funcion GetPost, que recibe un parámetro id y retornará un callback con un post

let getPost = (id, callback) => {

**let post = { titulo: "Titulo del post",** fecha: new Date().toLocaleString()}

callback(post)

}

En este caso, al crear la función, utilizamos la función callback para especificar que se retornará una respuesta al ejecutar la función (en este caso un post).

Posteriormente, podremos ejecutar la función y llamar al objeto recibido de la misma para continuar con la lógica que deseemos.

getPost(12, (respuestaCallback)=>{

console.log(respuestaCallback)

})

Recibimos el callback (le damos el nombre que deseeemos, en este caso respuestaCabllack) y posteriormente abrimos una nueva función de flecha con las instrucciones a ejecutar con el callback.

#### Promises

(ver promises.js)

Las promesas son similares a los Callbacks, ya que tienen como principal característica el retorno de un objeto. Tienen como diferencia que es un tipo de valor que será retornado de forma dinámico según una condición determinada se cumpla o no.

Con las promises, podemos crear funciones que retornarán datos y manejar los errores que puedan ejecutarse en las mismas.

Una promise se declara de la siguiente manera:

New promise (estas una palabra reservada), y posteriormente entre paréntesis establecemos los parámetros resolve y reject, que serán enviados cuando la promesa sea finalizada correcta o incorrectamente. Luego, con una función de flecha comenzamos a desarrollar la lógica de nuestra promise, y llamamos a los parámetros resolve y reject según corresponda.

//Creamos la promise, definiendo una variable y ejecutando new Promise

let promesa = new Promise((resolve, reject)=>{

// EL comando new Promise recibe 2 parámetros, resolve y reject.

// Resolve se aplica cuando la función se ejecuta correctamente

// Reject cuando falla

if (verification == true){

// Si la variable verificación es true, ejecutamos el resolve y devolvemos una cadena

resolve ("Todo ok!")

}else{

// Si la variable es false, ejecutamos el reject y devolvemos un error

reject("Error!")

}

});

Una vez que declaramos nuestra promesa, podemos llamar a la misma y ejecutar los métodos reservados then y catch, donde recibiremos los datos enviados en el caso de resolve y reject, respectivamente.

promesa.then((res)=>{

console.log(res);

}, (err)=>{

console.log(err)

})

A continuación, dejo una lista de links con información al respecto:

Promesas: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar_promesas>

Callbacks: <https://developer.mozilla.org/es/docs/Glossary/Callback_function>

Let y Var: <https://cybmeta.com/var-let-y-const-en-javascript>

Novedades Es6: <http://www.enrique7mc.com/2015/12/novedades-de-es6/>

### NPM

Npm es el gestor de paquetes de Node. El mismo nos permitirá importar de una manera sencilla diferentes librerías para potenciar nuestros desarrollos, como así también permitir la exportación de nuestro código y su utilización en diferentes dispositivos de manera sencilla.

Cuando instalamos nuevos paquetes lo que hace npm es instalarlo de manera local en nuestro proyecto dentro de la carpeta node\_modules, pero nosotros podemos decirle que lo instale de manera global de ser necesario.

Además, existen módulos que ya vienen instalados por defecto en Node.js por lo que no será necesario usar npm para traerlas, a estos módulos se les conoce como “módulos nativos”.

Para inicializar un proyecto Node con NPM, simplemente ejecutar el comando:

*Npm init*

en la carpeta donde deseemos inicializar el proyecto. A continuación, se nos otorgarán diferentes instrucciones donde podremos configurar el paquete de Node a crear (por el momento, pueden dejar todo en blanco).

Una vez que el proceso de creación finaliza, en la carpeta se creará un un archivo llamado package.json. El mismo dispone de información relacionada con nuestro proyecto de NPM y contiene los nombres y versiones de los diferentes módulos de node que utilizaremos en nuestro desarrollo.

{

"name": "Nombre del paquete”,

"version": “Versión del paquete”,

"description": “Descripción”,

"main": “Archivo donde se comenzará a ejecutar el código”,

"scripts": {

“Scripts personalizados que podremos ejecutar”

},

"author": “Nombre del creador del código”,

"license": “Tipo de licencia”

"dependencies": {

“Diferentes dependencias y librerías utilizadas, con sus correspondientes versiones, por ejemplo:”,

"express": "^4.16.3”

}

}

#### Instalar Paquetes y Librerias a nuestro proyecto node.

Para importar módulos y librerías con NPM, es necesario abrir la terminal en la carpeta de nuestro proyecto, y posteriormente ejecutar el comando:

***Npm install nombre-del-paquete –save***

* NPM es para llamar al gestor de paquetes de node
* Con install aclaramos que realizaremos una instalación de un paquete
* Posteriormente, dejamos un espacio y escribimos el nombre del paquete (por ejemplo, express)
* Y por último, añadimos el flag --save para especificar que el mismo se guardará de manera local y en nuestro package.json

### Express

Express es un framework de desarrollo en Node que nos permitirá desarrollar con una interfaz minimalista, pensada para webs y de una manera mucho más estructurada. En el archivo descargado, podrán observar un ejemplo de node funcionando.

#### Como instalar express en nuestro proyecto de node.

Una vez ejecutado el comando npm init, nos movemos al directorio de nuestro proyecto y ejecutamos la instrucción

***Npm install express –save***

Posteriormente, se agregará a nuestro proyecto, dentro de la carpeta node\_modules, las diferentes extensiones requeridas por Express.

Una vez que lo hayamos instalado, podremos comenzar a utilizarlo en nuestro proyecto de Node.

### Require en Node

A través del comando Require, podemos acceder a las diferentes librerías y paquetes instalados, como así también importar nuestros archivos locales para ejecutar las diferentes funciones que hayamos armado.

// Creamos una variable del tipo const e importamos en la misma el módulo http de node

const http = require('http');

// Cuando require comienza con ./ estamos aclarando que vamos a importar un archivo de nuestro codigo

const momento = require('./momento');

Podemos realizar 2 tipos de require.

* El primero, que comienza directamente con el nombre del paquete, nos permite importar módulos externos a la carpeta de nuestro proyecto.
* El segundo, que comienza con un ./, nos permitirá importar archivos de nuestro código y acceder a sus funciones.

## Creación de un Servidor con Node y Epxress desde 0

Para crear un servidor con Node, el primer paso es iniciar nuestro proyecto con NPM. Para ello, nos dirigimos a la carpeta donde deseemos comenzar, abrimos la terminal y ejecutamos:

Npm init

Una vez completados los pasos (con dar enter a todo es suficiente, por el momento), veremos que en la carpeta se creó el archivo package.json.

Posteriormente, instalamos express en nuestro proyecto, ejecutando:

***Npm install express –save***

Una vez ejecutado el comando, veremos que aparece una carpeta llamada node\_modules, que contiene todas las librerías necesarias para que Express funcione correctamente. También, veremos que se crea un archivo package-lock.json. El mismo contiene la referencia a todos los paquetes que Express necesitará para funcionar.

Una vez ejecutadas las instrucciones, creamos un archivo index.js en la carpeta principal del proyecto. En el mismo, se realizará toda la lógica de nuestra app.

Archivo: Index.js

En primer lugar, es necesario hacer el require, para importar Express, e inicializar su constructor dentro de una variable.

var express = require('express')

var app = express()

Una vez que hayamos hecho esto, podremos comenzar a establecer la lógica de nuestra aplicación utilizando la sintáxis de Express.

#### Requests y Responses

La mayoría de las funciones en node manejan 2 parámetros principales, que son req (request) y res (response). Sus nombres son autodescriptivos, el primero de ellos es un objeto que contiene todos los datos relacionados con la petición y el segundo objeto es utilizado para responder al navegador.

La request se refiere a la petición realizada por el cliente (que puede ser tanto Get o Post) y res se refiere a la respuesta (los datos que recibirá el cliente).

#### Rutas en Express

Las rutas nos permiten establecer las respuestas que serán ejecutadas frente a las diferentes peticiones del cliente a nuestro servidor, especificando la URL a la que deberá ingresar y las instrucciones a llevar a cabo en ese caso.

Para definir una ruta, simplemente llamamos a la variable donde tenemos inicializado express y posteriormente, con un punto, especificamos el método que activará las instrucciones, este puede ser:

Get (app.get): Todas las peticiones que se realizan sin enviar un cuerpo con información (Estas son las peticiones por defecto cuando ingresamos a una página web).

Post (app.post): Las peticiones con las cuales se envía un cuerpo con datos (Por ejemplo, al rellenar un formulario).

**En el archivo index.js, añadimos las siguientes líneas:**

// Definimos nuestras primeras rutas

app.get('/, function (req, res) {

res.send('Hola desde Rutas')

})

En este ejemplo especificamos que cuando se ingrese a la ruta “/” (sin contenido), se le enviará una respuesta con el texto “Hola desde rutas”.

Las rutas, también pueden recibir parámetros en su estructura, convirtiéndose, dentro de la ejecución de instrucciones, en variables.

**En el archivo index.js, añadimos las siguientes líneas:**

// En las rutas podemos enviar parametros

// Agregando un signo de pregunta al final del parametro lo convertimos en opcional

app.get('/saludar/:nombre/:apellido',(req,res)=>{

let nombre = req.params.nombre;

let apellido = req.params.apellido;

res.send("Hola "+nombre+ " "+ apellido)

});

En este ejemplo, estamos estableciendo que para la ruta saludar/, podremos agregar 2 parámetros, uno para nombre y otro para apellido, para posteriormente retornar una respuesta con los parámetros recibidos.

Entonces, cuando el usuario ingrese a la ruta y envíe los parámetros, por ejemplo:

[http://www. nombredelservidor.com/saludar/juan/paillet](http://www.miserver.com/saludar/juan/paillet)

Le será enviada la respuesta con “Hola Juan Paillet”.

#### Sirviendo contenido estático con Node.

En el supuesto de querer servir contenido estático (que no se modificará de manera dinámica) podemos establecer un directorio público, que será como una ruta global, y al acceder a la misma, se mostrarán todos los archivos correspondientes a ese directorio:

**En el archivo index.js, añadimos las siguientes líneas:**

//Podemos servir archivos estáticos y asignarles una ruta

app.use('/publico', express.static(\_\_dirname +'/publico'));

De esta forma, en lugar de usar get (para establecer rutas basadas en peticiones) llamamos al método use, que recibe una ruta y una carpeta con archivos (que en este caso será nuestra página estática). Al ingresar a la ruta [www.nombredelservidor.com/publico](http://www.nombredelservidor.com/publico), se abrirá la carpeta con los archivos HTML, CSS Y JS que deseemos mostrar.

A continuación, en nuestro directorio, creen una carpeta llamada **publico** y añadan un archivo llamado **index.html** (archivo que se abrirá por defecto).

Dentro del mismo, escriban el html que deseen mostrar, por ejemplo:

(Archivo publico/index.html)

<html>

<h1 class="display-4">Hola desde Node Estático!</h1>

</html>

Una vez guardado el archivo, cuando levantemos nuestro servidor de node, si un usuario ingresar a la ruta público, automáticamente se mostrarán los archivos contenidos en la carpeta, por defecto, el index.html.

Una vez establecido nuestro contenido estático, volvemos a nuestro fichero principal de node (index.js) para especificar el puerto en el que se levantará nuestro servidor:

**Por último, en el archivo index.js, añadimos las siguientes líneas:**

// Especificamos el puerto donde escuchará nuestro servidor

app.listen(1234,()=>{console.log("Server listo en puerto 1234")})

Con la directriz listen, especificamos el puerto en el que se levantará nuestro servidor. Esto nos devolverá un callback que podemos ejecutar de manera opcional para ejecutar las instrucciones que deseemos, en este caso, un console.log (impresión en consola) de que nuestro servidor se encuentra funcionando correctamente.

Siguiendo los pasos, su archivo index.js debería haber quedado similar a esto:

var express = require('express')

var app = express()

// Definimos nuestras primeras rutas

app.get('/, function (req, res) {

res.send('Hola desde Rutas')

})

// En las rutas podemos enviar parametros

// Agregando un signo de pregunta al final del parametro lo convertimos en opcional

app.get('/saludar/:nombre/:apellido',(req,res)=>{

let nombre = req.params.nombre;

let apellido = req.params.apellido;

res.send("Hola "+nombre+ " "+ apellido)

});

//Podemos servir archivos estáticos y asignarles una ruta

app.use('/publico', express.static(\_\_dirname +'/publico'));

// Especificamos el puerto donde escuchará nuestro servidor

app.listen(1234,()=>{console.log("Server listo en puerto 1234")})

Y una carpeta llamada “publico” con un archivo “index.html” dentro de la misma, que contenga el código HTML que deseen.

## Levantando el servidor

Una vez configurado nuestros archivos, podemos levantar nuestro servidor abriendo la consola y ejecutando el comando

**Node index.js**

Automáticamente, se imprimirá en la terminal el mensaje con el callback de conexión (“Server listo en puerto 1234”).

Para ingresar al servidor, abrir en el navegador localhost:puerto, (en este caso localhost:1234) y a la ruta deseada, las cuales pueden ser, hasta el momento

**http://localhost:1234/** (para ingresar a ruta por defecto)

**http://localhost:1234/saludar/nombre/apellido** (para ingresar a ruta con parámetros)

**http://localhost:1234/publico** (para ingresar a ruta que devuelve contenido estático)

Nodemon

Una de las principales herramientas para trabajar con Node es Nodemon. La misma nos permite que el servidor se reinicie cada vez que realicemos cambios (en lugar de tener que hacerlo de manera manual).

Para instalar y utilizar nodemon, cerramos nuestro servidor (si es que lo levantaron) presionando Control + C.

Posteriormente, ejecutamos el comando

**Npm install nodemon –save**

Para que el módulo nodemon se descargue en nuestro proyecto.

Una vez instalado, levantaremos nuestro servidor con un nuevo comando:

**Nodemon .**

Ahora, cuando realicemos modificaciones en nuestros archivos, automáticamente se reiniciará nuestro servidor.