Contenido

[Bibliografía 3](#_Toc507755910)

[1 Introducción a Node.js 4](#_Toc507755911)

[¿Qué es Node.js? 4](#_Toc507755912)

[Ventajas y desventajas de Node.js 4](#_Toc507755913)

[Arquitectura node js 4](#_Toc507755914)

[Instalación de NodeJs 8](#_Toc507755915)

[Instalación de Atom 8](#_Toc507755916)

[Hola Mundo con Node.js 8](#_Toc507755917)

[Instalando body-parser y nodemon 9](#_Toc507755918)

[2 Express y Swig o Handlebars 9](#_Toc507755919)

[¿Qué es Express? 9](#_Toc507755920)

[Instalación de Express 9](#_Toc507755921)

[Instalación de Swig 10](#_Toc507755922)

[Instalación de Handlebars 10](#_Toc507755923)

[Compilando un simple Typescript a Javascript 10](#_Toc507755924)

[Módulos de tipografía y palabras clave de importación / exportación 11](#_Toc507755925)

[Module formats in JavaScript CommonJS , AMD , ES6 12](#_Toc507755926)

[Calling Javascript module loaders in HTML UI 15](#_Toc507755927)

[Understanding Module Bundlers using WebPack 16](#_Toc507755928)

[Instalar WebPack 17](#_Toc507755929)

[Llamar a la clase javascript en la página web. 18](#_Toc507755930)

[**3** Introduccion a ES6 Promesas 18](#_Toc507755931)

[Callbacks 19](#_Toc507755932)

[El temido Callback Hell o Pyramid of Doom 20](#_Toc507755933)

[Promesas 21](#_Toc507755934)

[Async/Await 22](#_Toc507755935)

[4 MongoDB y Mongoose 23](#_Toc507755936)

[¿Qué es MongoDB? 23](#_Toc507755937)

[Mongoose (definir un esquema y trabajo con modelos) 23](#_Toc507755938)

Bibliografía

<https://www.codeproject.com/Articles/1207118/Learn-Angular-Tutorial#How_does_this_article_series_teach_you_Angular_>

<https://nodejs.org/docs/latest-v7.x/api/>

<https://www.udemy.com/the-complete-nodejs-developer-course-2/>

<https://code.i-harness.com/es/q/2488309>

<http://mongoosejs.com/docs/guide.html>

<http://fernando-gaitan.com.ar/introduccion-a-node-js-parte-13-crud-con-mongoose/>

# 1 Introducción a Node.js

## ¿Qué es Node.js?

NodeJS es un framework de código abierto de JavaScript que hace dos cosas:

    Ayuda a ejecutar JavaScript fuera del navegador. NodeJS utiliza el motor de JavaScript de Chrome para ejecutar JavaScript fuera del navegador para que podamos crear aplicaciones basadas en el escritorio y el servidor mediante JavaScript y actúa como un repositorio central desde donde podemos obtener cualquier marco de JavaScript usando NPM (Node Package Manager).

## Ventajas y desventajas de Node.js

Ventajas NodeJS:

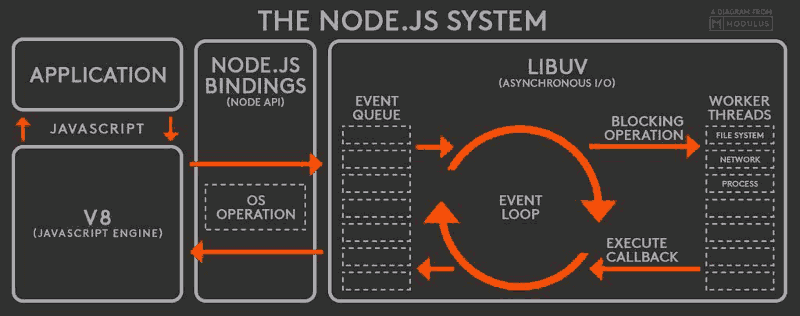
* Se ejecuta Javascript, por lo que puede utilizar el **mismo idioma** en el servidor y el cliente, e incluso compartir algún código entre ellos.
* El sistema basado en eventos de un solo un hilo es rapido incluso cuando se manejan muchas peticiones a la vez, y también es simple, en comparación con los marcos de Java ROR tradicionales multi-threaded.
* El siempre creciente grupo de **paquetes accesibles a través de NPM** , incluyendo bibliotecas / módulos del cliente y del servidor, así como herramientas de línea de comandos para el desarrollo web.
* Se ha convertido en el entorno estándar para ejecutar **herramientas relacionadas con Javascript** y otras **herramientas relacionadas con la web,** incluyendo corredores de tareas, minifiers, embellecedores, linters, preprocesadores, bundlers y procesadores de análisis.
* Parece bastante adecuado para la creación de prototipos, el desarrollo ágil.
* La E / S impulsada por eventos asíncronos ayuda al manejo concurrente de solicitudes
* Fácil de escalar
* Multiplataforma

Desventajas NodeJS:

* Ejecuta Javascript, que no tiene comprobación de tipo de tiempo de compilación.
* Npmjs.org no tiene ningún mecanismo para clasificar los paquetes, lo que ha llevado a una proliferación de paquetes que hacen más o menos lo mismo, de los cuales un gran porcentaje ya no se mantienen.
* El grupo cada vez mayor de paquetes puede hacer que un proyecto NodeJS aparezca **radicalmente diferente** del siguiente.
* Tratar con archivos puede ser un poco un dolor.
* Se ejecuta en JavaScript, que es fácil de aprender, pero muy difícil de escribir, menos código propenso a errores y no tiene comprobación de tipo en tiempo de compilación
* Defectos anidados
* La falta de bibliotecas robustas

## Arquitectura node js

El código de la aplicación es compilado por V8 (motor de JavaScript desarrollado por google para usar en Chrome). El código se comunica con los componentes de bajo nivel Node.js mediante enlaces. Todos los eventos escritos en el código están registrados con Node.js. Una vez que se desencadenan los eventos, se ponen en cola en la cola de eventos de acuerdo con el orden en que se desencadenan. Mientras que aún haya eventos restantes en la cola de eventos, el bucle de eventos los sigue captando, llamando a sus funciones de devolución de llamada y enviándolos a los hilos de trabajo para su procesamiento. Una vez que se ejecuta una función de devolución de llamada, su devolución de llamada nuevamente se envía a la cola de eventos, esperando a ser recuperada por el bucle de eventos nuevamente.



**Libev** es el bucle de eventos que realmente se ejecuta internamente en node.js para realizar operaciones de bucle de eventos. Está escrito originalmente para sistemas \* nix. **Libev** proporciona un bucle de evento simple pero optimizado para que el proceso se ejecute. (http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod)

**LibEio** es una biblioteca para realizar salidas de entrada de forma asincrónica. Maneja descriptores de archivos, manejadores de datos, tomas de corriente, etc.( http://pod.tst.eu/http://cvs.schmorp.de/libeio/eio.pod)

**LibUv** es una capa de abstracción en la parte superior de libeio, libev, c-are (para DNS) y iocp (para windows asyncronous-io). LibUv realiza, mantiene y administra todos los io y eventos en el grupo de eventos. (en caso de threadpool libeio). ()

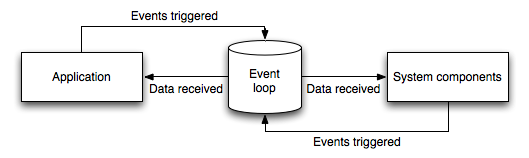
Básicamente, lo que sucede dentro de node.js es que el bucle v8 se ejecuta y maneja todas las partes de JavaScript, así como los módulos de C ++ [cuando se ejecutan en un hilo principal (según la documentación oficial node.js tiene un solo hilo)]. Cuando están fuera del hilo principal, libev y libeio lo manejan en el grupo de hilos y liberan la interacción con el ciclo principal. Por lo tanto, node.js tiene 1 ciclo de eventos permanentes: ese es el ciclo de eventos v8. Para manejar las tareas asíncronas de C ++, está utilizando un subproceso de subprocesos [a través de libeio & libev].

El node js es una aplicación de un solo hilo pero admite simultaneidad a través del concepto de evento y de devolución de llamada. Como todas las API de Node js son asincrónicas y tienen un único hilo, utilizan llamadas de función asíncrona para mantener la concurrencia. El nodo usa un patrón de observador. El hilo del nodo mantiene un bucle de evento y cada vez que se completa una tarea, activa el evento correspondiente que indica que la función del detector de eventos se ejecuta.

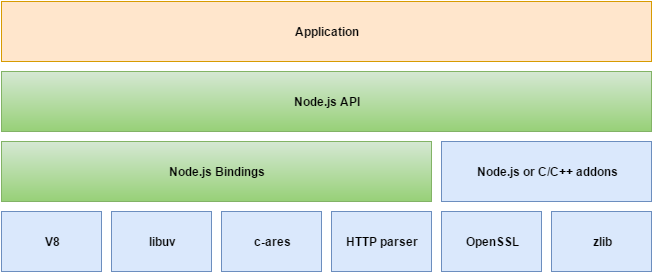


Node.js usa los eventos en gran medida y es también una de las razones por las que Node.js es bastante rápido en comparación con otras tecnologías similares. Tan pronto como el nodo inicia su servidor, simplemente inicia sus variables, libera funciones y simplemente espera a que ocurra el evento.

En una aplicación impulsada por eventos, generalmente hay un bucle principal que escucha eventos y luego activa una función de devolución de llamada cuando se detecta uno de esos eventos.



Mientras que los Eventos se parecen a lo que son las devoluciones de llamada. La diferencia radica en el hecho de que las funciones de devolución de llamada se invocan cuando una función asíncrona devuelve su resultado cuando el manejo de eventos funciona en el patrón del observador. Las funciones que escuchan eventos actúan como observadores. Cada vez que se dispara un evento, su función de escucha comienza a ejecutarse.



**V8:** V8 es el motor de código abierto JavaScript de alto rendimiento de Google, escrito en C ++ y utilizado en Chromium, Node.js y otras aplicaciones de inserción múltiples

**libuv**: libuv fue desarrollado originalmente para proporcionar E / S asíncronas que incluyen zócalos TCP y UDP asincrónicos, bucle de evento (famoso), resolución DNS asíncrona, lectura / escritura del sistema de archivos, etc. libuv está escrito en C.

**c-ares:** c-ares es una biblioteca C para solicitudes de DNS asíncronas (incluido el nombre resuelve)

**Analizador HTTP:** esta biblioteca se usa para analizar mensajes HTTP (solicitud y respuesta) escritos en C

**OpenSSL:** OpenSSL es un proyecto de código abierto que proporciona un conjunto de herramientas robusto, de calidad comercial y con todas las funciones para los protocolos Transport Layer Security (TLS) y Secure Sockets Layer (SSL).

**zlib** y muchos otros.

**Enlace Node.js**: un enlace básicamente es un contenedor alrededor de una biblioteca escrita en un idioma y expone la biblioteca a códigos escritos en otro idioma para que los códigos escritos en diferentes idiomas puedan comunicarse.

**Complementos C / C ++:** los complementos C / C ++ son objetos compartidos vinculados dinámicamente, escritos en C o C ++, que pueden cargarse en Node.js utilizando la función requerida y utilizarse como si fueran un Node.js ordinario módulo. Se utilizan principalmente para proporcionar una interfaz entre JavaScript que se ejecuta en las bibliotecas Node.js y C / C ++.

**API Node.js:** la API Node.js es una abstracción de otros módulos de bajo nivel y es la interfaz con bibliotecas integradas.

**Aplicación:** código de aplicación Node.js

## Instalación de NodeJs

Para instalar NodeJS nos dirigimos a https://nodejs.org/ y descargamos la última versión e instalamos. Una vez que instalado el nodo, deberías ver el símbolo del sistema de NodeJs en tus archivos de programa.

## Instalación de Atom

El comando "npm install" le ayuda a obtener la última versión de cualquier marco de código abierto de JavaScript.

Por ejemplo, si desea instalar jquery, abrirá el símbolo del sistema del nodo y tecleará "npm install jquery". ¿Te estás preguntando dónde se ha instalado Jquery? Se ha instalado en la misma carpeta donde ejecutó el comando NPM.

En esa carpeta, ha creado una carpeta "node\_modules" y en ella ha creado la carpeta "jquery" donde todo Jquery había sido cargado por "npm".

## Hola Mundo con Node.js

* Cree un directorio llamado myapp
* Con una consola colóquese en la carpeta
* Ejecute npm init
* Instale Express
* Cree un archivo llamado app.js
* Añade el código siguiente:

var express = require('express');

var app = express();

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

res.send('Hello World!');

});

app.listen(3000, function () {

console.log('Example app listening on port 3000!');

});

* Ejecute la aplicación con el siguiente mandato:

$ node app.js

* A continuación, cargue <http://localhost:3000/> en un navegador para ver la salida.
* La aplicación inicia un servidor y escucha las conexiones en el puerto 3000. La aplicación responde con “Hello World!” para las solicitudes al URL raíz (/) o a la ruta raíz. Para cada vía de acceso diferente, responderá con un error **404 Not Found**.

## Instalando body-parser y nodemon

Para manejar HTTP POST en **Express.js** versión 4(+), necesita instalar un módulo de middleware llamado body-parser .

**body-parser extrae toda la parte del cuerpo de una secuencia de solicitud entrante y la expone en req.body .**

Este módulo body-parser analiza el JSON, el búfer, la cadena y los datos codificados de URL enviados mediante HTTP POST .

Instale body-parser usando NPM como se muestra a continuación.

npm install body-parser --save

# 2 Express y Swig o Handlebars

## ¿Qué es Express?

Es una extensión de [connect](http://senchalabs.org/connect) y esta inspirado en **sinatra**, además es robusto, rápido, flexible, simple. Sus características son las siguientes:

* Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).
* Integración con motores de renderización de "vistas" para generar respuestas mediante la introducción de datos en plantillas.
* Establecer ajustes de aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
* Añadir procesamiento de peticiones "middleware" adicional en cualquier punto dentro de la tubería de manejo de la petición.

A pesar de que Express es en sí mismo bastante minimalista, los desarrolladores han creado paquetes de middleware para abordar casi cualquier problema de desarrollo web. Hay librerías para trabajar con cookies, sesiones, inicios de sesión de usuario, parámetros URL, datos POST, cabeceras de seguridad y muchos más. ([Express Middleware](https://expressjs.com/es/resources/middleware.html))

## Instalación de Express

Abrir una consola y teclear dentro de la carpeta donde existe el proyecto.

$ npm install express –save.

## Instalación de Swig

Swig es un motor de plantillas de JavaScript simple, potente y extensible

Abrir una consola y teclear dentro de la carpeta donde existe el proyecto.

$ npm install swig --save

## Instalación de Handlebars

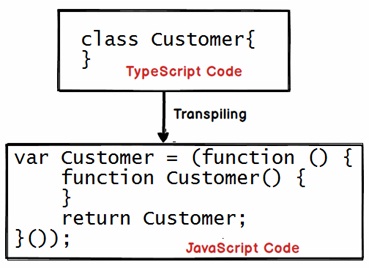
Handlebars permite crear plantillas semánticas de manera efectiva. Handlebars es ampliamente compatible con las plantillas de Moustache.

Abrir una consola y teclear dentro de la carpeta donde existe el proyecto

$ npm install --save handlebars

Entonces, en JavaScript, si desea heredar, necesita usar un prototipo, no es un lenguaje fuertemente tipado, no hay polimorfismo, etc. Entonces, cuando los desarrolladores provienen de C # y Java, es muy difícil para ellos familiarizarse con este lenguaje.

Entonces, para llenar esta respuesta GAP es "TypeScript".



Typescript es un marco de código abierto de JavaScript, por lo que la mejor manera de instalarlo es mediante el uso de "npm". Así que abra el indicador de comando del nodo y escriba

"npm install typescript -g".

El comando "-g" dice que puede ejecutar el comando de mecanografía desde cualquier carpeta.

# Compilando un simple Typescript a Javascript

Creamos un archivo simple "Customer.ts" con el siguiente código.

class Customer{

}

Ahora abra el prompt del comando nodeJS y escriba el comando “tsc Customer.ts ''. Una vez que presione enter, creará "Customer.js" en la misma carpeta.

A continuación se muestra la salida de javascript

var Customer = (function () {

function Customer() {

}

return Customer;

}());

Mucha gente llama a esta conversión "compilación". Aunque otras personas proponen llamarlo "transpiling".

Compilando conversiones desde lenguajes de nivel a lenguaje de máquina o algún lenguaje intermedio que no puede ser leído por humanos. Mientras transfiere conversiones de un lenguaje de nivel superior a otro lenguaje de nivel superior.

Pero ahora pensemos de forma práctica, si quiero transpilar con la especificación ES5, a un directorio específico sin comentarios, la línea de comando se convertiría en algo como se muestra a continuación.

tsc Customer.ts --outdir "c:\users\shiv" --target ES5 --removecomments

Ahí es donde el archivo tsconfig.json viene a rescatar. Se puede poner todas estas configuraciones en el archivo "tsconfig.json" y luego ejecutar "tsc".

{

"compilerOptions": {

"target": "es5",

"removeComments": false,

"outDir": "/Shiv"

}

}

## Módulos de tipografía y palabras clave de importación / exportación

El desarrollo modular es uno de los pilares importantes del desarrollo. Un buen software siempre tendrá módulos autónomos.

Por lo tanto para crear archivos físicos separados de texto o JavaScript que tengan un código independiente se debe tener algún tipo de mecanismo de referencia por el cual los módulos se pueden referir entre esos archivos físicos y hacemos esto mediante el uso de palabras clave "importar" y "exportar".

Por lo tanto, los módulos que deben exponerse deben tener la palabra clave "export", mientras que los módulos que desean importar los módulos exportados deben tener la palabra clave "import".

Por ejemplo, digamos que tenemos dos archivos de texto "Customer.ts" y "Dal.ts". Supongamos que "Customer.ts" está usando "Dal.ts".

Entonces "Dal.ts" usará exportar para exponer sus módulos mientras que "Customer.ts" usará para importar para obtener el módulo exportado.

Por lo tanto, en "Dal.ts", las clases que desee exportar se deben marcar como "exportadas", como se muestra en el siguiente código. Si no lo marca exportado, no se puede importar

export class Dal{

Add(){

alert("Dal add called");

}

}

Ahora en "Customer.ts" usamos "import" para llamar a la clase exportada desde "Dal.ts".

import {Dal} from "./Dal "

export class Customer{

Add(){

var dal = new Dal();

dal.Add();

}

}

En resumen, usa exportar e importar para hacer un desarrollo modular en texto. Pero ahora, ¿cómo funciona este "TRANSPILE" al código de JavaScript que veremos en la siguiente sección? Al final del día, todos estos módulos se transponen a JavaScript, así que vamos a entender cómo funciona bajo esa cubierta.

### Formato de módulos en JavaScript CommonJS , AMD , ES6

Los formatos de módulo definen las sintaxis de JavaScript sobre cómo debe exportarse e importarse el módulo.

En JavaScript hay dos formas de definir formatos de módulos: - Forma no oficial y Forma oficial. Por lo tanto, antes de ES6 no había una forma oficial, por lo que algunas de las formas virales no oficiales de definir formatos de módulos son CommonJs, AMD, UMD, etc. La forma oficial es única y solo una ES6.

En el formato de módulo "ES6" para exponer la clase, necesitamos "export" palabras clave y para consumir necesitamos usar "import".

Puede ver a continuación la salida JS donde el dal se exporta usando la variable "exports".

Object.defineProperty(exports, "\_\_esModule", { value: true });

var Dal = (function () {

function Dal() {

}

Dal.prototype.Add = function () {

alert("Dal add called");

};

return Dal;

}());

exports.Dal = Dal;

El código para "Customer.js" que usa "require" para cargar "Dal.js".

Object.defineProperty(exports, "\_\_esModule", { value: true });

var Dal\_js\_1 = require("./Dal.js");

var Customer = (function () {

function Customer() {

}

Customer.prototype.Add = function () {

var dal = new Dal\_js\_1.Dal();

dal.Add();

};

return Customer;

}());

**exports.Customer = Customer;**

Así que ahora este es un formato JS común de la misma manera que tenemos otros formatos también. Por ejemplo, debajo está el formato del módulo "amd".

En esto, exportamos las clases en la variable "export" y usamos "define" para importar. Debajo está el código de "definir". No estamos pegando de "exportar" lo mismo que commonJS.

define(["require", "exports", "./Dal.js", "./Validation.js"], function (require, exports, Dal\_js\_1, Validation\_js\_1) {

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

var Customer = (function () {

function Customer() {

}

Customer.prototype.Add = function () {

var val = new Validation\_js\_1.Validation();

var dal = new Dal\_js\_1.Dal();

dal.Add();

};

return Customer;

}());

exports.Customer = Customer;

});

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

import { Dal } from "./Dal.js";

import { Validation } from "./Validation.js";

var Customer = (function () {

function Customer() {

}

Customer.prototype.Add = function () {

var val = new Validation();

var dal = new Dal();

dal.Add();

};

return Customer;

}());

export { Customer };

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

var Dal = (function () {

function Dal() {

}

Dal.prototype.Add = function () {

alert("Dal add called");

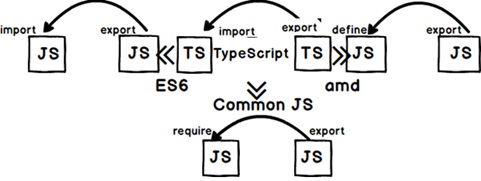
};

return Dal;

}());

export { Dal };

Entonces, en palabras simples "amd", "commonJS" y "ES6" definen cómo los módulos se comunicarán entre sí. Concluyendo ES6 usa "importación / exportación", amd usa "define / export" y commonJs usa "require / export".



Entonces en "tsconfig.json" podemos establecer en "módulo" qué formato de módulo queremos.

### Llamando a los módulos de Javascript para cargarlos en HTML UI

Ahora, cuando intentamos cargar funciones de JavaScript que usan formatos de módulos como AMD, CommonJS o ES6, no es tan fácil. Por ejemplo, en el código siguiente en la interfaz de usuario HTML, hemos cargado "Dal.js" y "Cliente" .js ". También hemos puesto la secuencia correctamente, primero hemos agregado la referencia de "Dal.js" y luego "Customer.js" porque "Customer.js" depende de "Dal.js".

Pero cuando tratamos de crear el objeto "Cliente" e intentamos llamar a "Agregar", no funciona.

<script src="Dal.js"></script>

<script src="Customer.js"></script>

<script>

var cust = new Customer();

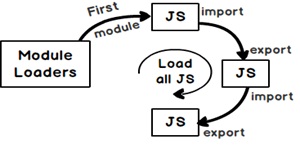
cust.Add();

</script>

El segundo problema es que incluso si este código hubiera funcionado, aún tendría problemas de pedido para un gran número de referencias. Digamos que tenemos 15 módulos que hacen referencia a los formatos de módulos que terminaríamos con la vida media de gastos arreglando esas secuencias en un archivo HTML. Sería genial si podemos simplemente señalar a "Customer.js" y usar automáticamente "exports" e "imports" se identifican las referencias y se carga "Address.js".

Ahí es donde necesitamos los cargadores de módulos Javascript. Algunos ejemplos de cargadores de módulos son SystemJS, WebPack, etc.

Por lo tanto, si utilizamos cargadores de módulos, solo debemos señalar el primer archivo JS y usar automáticamente "importar / exigir / definir" y "exportar", obtendrá referencias de todos los archivos JS dependientes y los cargará en consecuencia.



Permite mostrar un módulo usando "SystemJS". Por lo tanto, primero ingrese el indicador de comando Node instale "systemjs".

npm install systemjs

Por lo tanto, en la interfaz de usuario HTML, debemos indicar "system.js", que es el primer archivo JS que se cargará. Puede ver en el código siguiente que estamos diciendo "SystemJS.import" cargar "Customer.js" como primer archivo.

<script src="system.js"></script>

<script>

SystemJS.import('./Customer.js')

.then(function(module){

var cust = new module.Customer();

cust.Add();

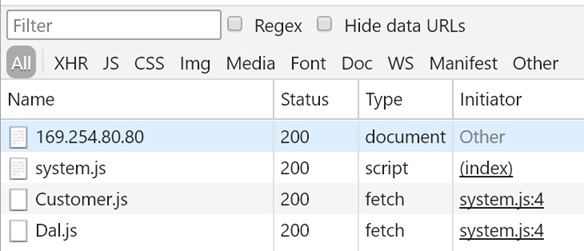
}).catch(function (err)

{ console.error(err); });;

</script>

Una vez que el archivo ha sido cargado en la función entonces obtenemos los módulos. Entonces podemos referir la variable del módulo y crear el objeto de la función "Cliente".

Si miras la pestaña de red del navegador Chrome, puedes ver que primero "system.js" carga "Customer.js" y luego carga su referencia que es "Dal.js".

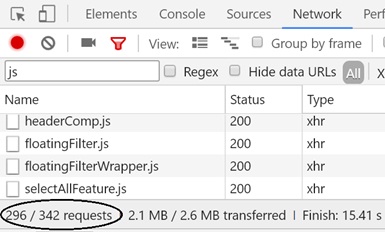


## Comprendiendo los límites de los módulos WebPack

Lab 4 anteriorJS estaba haciendo todo en el tiempo de ejecución. Entonces, en el navegador primero carga "Customer.js", luego "Address.js", y así sucesivamente. Si tienes muchas dependencias, terminarías con muchas solicitudes.

En el lado hay una imagen simple de un proyecto empresarial donde tenemos 342 solicitudes de archivos JavaScript para cargar el sitio.

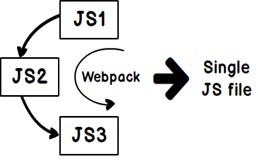
Ahora que es una gran cantidad y con tantas solicitudes, el rendimiento de su aplicación bajará mucho.



Si podemos crear un SINGLE BUNDLE durante el tiempo de compilación, sería un gran refuerzo de rendimiento.

Ahí es donde viene a usarse el paquete web (https://webpack.js.org/). Webpack es un paquete de módulos.

Toma el primer archivo JS utiliza definiciones de módulos como commonjs / AMD / UMD, etc. y calcula las referencias y genera un solo archivo JS DURANTE EL TIEMPO DE COMPILACIÓN. Puede tomar este paquete único JS y ponerlo en su página web.



Entonces, intentemos comprender los conceptos básicos de cómo funciona Webpack.

## Instalar WebPack

Entonces, el primer paso es instalar el paquete web. Abra el indicador de comando del nodo y escriba el siguiente comando de NPM. Por favor, mira que usamos "-g" la bandera global.

npm install -g webpack

Genera un solo paquete Tomamos el mismo código que hemos utilizado en "Lab 4". En el laboratorio 4, si ve que tenemos "Customer.js" llamando a "Address.js". Entonces, el primer archivo Javascript es "Customer.js". Solo debemos dar el nombre del archivo JS en el comando webpack y el nombre del archivo del paquete final. También puede ver que hay un parámetro de indicador "-output-library". Webpack expondrá el componente "CustomerLibrary" desde donde podemos acceder a la clase "Customer" en la interfaz de usuario.

webpack "Customer.js" "Bundle.js" --output-library='CustomerLibrary'

## Llamar a la clase javascript en la página web.

Entonces, ahora que tenemos un paquete único, solo podemos cargar el archivo JS en la página web y acceder a las clases a través de "CustomerLibrary". Recuerde que esta "CustomerLibrary" proviene de la línea de comandos, vuelva a consultar el paso 2 para obtener más información.

<script src="Bundle.js"></script>

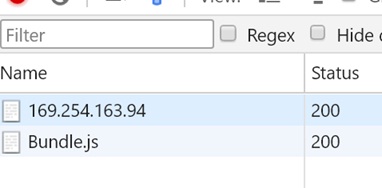
<script>

var x = new CustomerLibrary.Customer();

x.Add();

</script>

Si ahora ves la pestaña de red, verás ahora solo un único archivo "Bundle.js" en comparación con varios archivos.



# **3** Introduccion a ES6 Promesas

#### ¿Que es una promesa?

Una promesa es algo que pasará en el futuro pero no inmediatamente. Por ejemplo una imagen que todavía no está cargada, pero cumple todos los requisitos necesarios así sea, y que en un futuro lo estará. Si queremos que una parte del código se ejecute solo cuando la imagen esté cargada, podemos utilizar promesas.

Las promesas se originaron en el ámbito de la programación funcional, aunque diversos paradigmas las han incorporado, generalmente para gestionar la programación asíncrona. En resumen, nos permiten definir cómo se tratará un dato que sólo estará disponible en un futuro, especificando qué se realizará con ese dato más adelante.

Si hay algo que caracteriza a JavaScript, es la asincronía que presentan algunas funciones. Concretamente las que realizan operaciones de entrada/salida como escritura o lectura del disco o una petición AJAX.

En JavaScript existen varias formas de manejar estos procesos en nuestros desarrollos.  
Veamos cuales son.

## Callbacks

Es la primera y la forma más común de controlar la asincronía en JavaScript.

En el siguiente ejemplo tenemos una función que recibe como parámetros un dato de entrada: data, un array con datos array y una función de callback: callback.

El funcionamiento de la función es muy simple, al array se le añade el data que viene por parámetro y cuando termine, llama a la función de callback que recibe por parámetro, en ese caso la llama con el array modificado.

He añadido un pequeño bloque para comprobar si el array existe y si no lanzar un error que pasaremos al callback.

function addToArray (data, array, callback) {

if (!array) {

callback(new Error('No existe el array), null)

} else {

array.push(data)

callback(null, array)

}

}

En el siguiente código vemos como llamara a este función y tratar el callback:

var array = [1,2,3];

addToArray(4, array, function (err) {

if (err) return console.log(err.message)

console.log(array)

})

// [1, 2, 3, 4]

¿Esto que va a devolver? Cuando se termine de ejecutar la función addToArray se ejecutará el callback y nos mostrará el array con el nuevo dato.

Parece una tontería porque podríamos añadir el dato al array y después imprimirlo con console.log, pero imaginemos que esa operación de añadir un item al array fuera asincrónica, como podría ser una llamada vía AJAX.

Para simular esto vamos a utilizar la función setTimeout para añadir un retardo de 1 segundo:

function addToArray (data, array, callback) {

if (!array) {

return callback(new Error('No existe el array', null)

}

setTimeout(function() {

array.push(data)

callback(null, array)

}, 1000)

}

var array = [1,2,3];

addToArray(4, array, function (err) {

if (err) return console.log(err.message)

console.log(array)

})

// (1 seg de delay)-> [1, 2, 3, 4]

Si no tuviéramos una función de callback, y la función addToArray fuera:

function addToArray (data, array) {

setTimeout(function() {

array.push(data)

}, 1000)

}

y ejecutáramos la función, nos devolvería lo siguiente:

var array = [1, 2, 3]

addToArray(4, array)

console.log(array)

// [1, 2, 3]

Cuando imprimimos el array aún no se ha añadido el nuevo item, por lo tanto el comportamiento que sucede no es el buscado. De esta forma los callbacks nos ayudan a lidiar con esto.

Pero si tenemos varias funciones así... puede ocurrir lo siguiente:

var array = [1,2,3];

addToArray(4, array, function (err) {

if (err) ...

addToArray(5, array, function (err) {

if (err) ...

addToArray(6, array, function (err) {

if (err) ...

addToArray(7, array, function () {

// Hasta el infinito y más allá...

})

})

})

});

### El temido **Callback Hell** o **Pyramid of Doom**

Por suerte esto se ha podido resolver utilizando librerías como async, o empleando promesas con librerías como Q.

## Promesas

Por suerte, en la nueva especificación de JavaScript (ES6 o ES2015) **las Promesas ya son nativas** y no necesitamos requerir librerías de terceros.

Veamos el mismo ejemplo que antes pero utilizando Promesas nativas de ES2015

function addToArray (data, array) {

const promise = new Promise(function (resolve, reject) {

setTimeout(function() {

array.push(data)

resolve(array)

}, 1000);

if (!array) {

reject(new Error('No existe un array'))

}

})

return promise

}

const array = [1, 2, 3]

addToArray(4, array).then(function () {

console.log(array)

})

Ahora la función *addToArray* crea un objeto *Promise* que recibe como parámetros una función con las funciones *resolve* y *reject*.

resolve la llamaremos cuando nuestra ejecución finalice correctamente.

De esta manera, podemos escribir código de manera más elegante, y el Callback Hell anterior puede ser resuelto así:

const array = [1, 2, 3]

addToArray(4, array)

.then(function() { return addToArray(5, array) })

.then(function() { return addToArray(6, array) })

.then(function() { return addToArray(7, array) })

.then(function () {

console.log(array)

})

// (4 seg. de delay)-> [1,2,3,4,5,6,7]

Esto se conoce como anidar promesas.

La forma de tratar errores en una promesa, es por medio de la función catch que recoge lo que enviamos en la función reject dentro de la Promesa. Y esta función solo hay que invocarla una vez, no necesitamos comprobar en cada llamada si existe error o no. Lo cual reduce mucho la cantidad de código

const array = ''

addToArray(4, array)

.then(...)

.then(...)

.then(...)

.catch(err => console.log(err.message))

// No existe el array

## Async/Await

El siguiente paso en el tratamiento de los procesos asíncronos es usar los métodos async/await. Ésta forma no está disponible en el estándar de ES6 o ES2015, si no que **forma parte de la próxima versión ES7 o ES2016**, pero que podemos utilizar hoy día [con Babel y el preset de plugins stage-3](http://babeljs.io/docs/plugins/preset-stage-3/)

La sintaxis para una función que utilice async/await es la siguiente

async function myFuncion () {

try {

var result = await functionAsincrona()

} catch (err) {

...

}

}

La función irá precedida por la palabra reservada async y dentro de ella tendremos un bloque try-catch. Dentro del try llamararemos a la función asíncrona con la palabra reservada await delante, con esto hacemos que la función espere a que se ejecute y el resultado de la misma está disponible en este caso en la variable result.

Si ocurre algún error durante la ejecución, se ejecutará el bloque catch donde trataremos el error.

Combinando *async/await* con una función basada en Promesas, podemos hacer lo siguiente con el ejemplo que estábamos viendo:

function addToArray (data, array) {

const promise = new Promise(function (resolve, reject) {

setTimeout(function() {

array.push(data);

resolve(array);

}, 1000);

if (!array) {

reject(new Error('No existe un array'));

}

})

return promise;

}

const array = [1, 2, 3];

async function processData (data, array) {

try {

const result = await addToArray(data, array);

console.log(result)

} catch (err) {

return console.log(err.message);

}

}

processData(4, array)

// [1,2,3,4]

processData(5, array)

// [1,2,3,4,5]

processData(6, array)

// [1,2,3,4,5,6]

De esta manera estamos escribiendo código de manera secuencial pero JavaScript está por debajo ejecutando código asincrónico.

Esto permite que, desarrolladores que estén menos familiarizados con JavaScript y su comportamiento, no tengan tantas barreras de entrada para empezar a trabajar con el lenguaje.

4 MongoDB y Mongoose

¿Qué es MongoDB?

**MongoDB** es una base de datos orientada a documentos. Esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos. Estos documentos son almacenados en BSON, que es una representación binaria de JSON.

Cuándo estamos trabajando con node.js y MongoDB nos encontramos con el tema de cómo conectar node.js con nuestra base de datos en MongoDB aquí es donde usamos la librería llamada mongoose.

Mongoose (definir un esquema y trabajo con modelos)

Ahora bien, para hacer la instalación de la librería mongoose hacemos uso del siguiente comando en terminal:

$ npm install mongoose

Para llamar el módulo y construir la conexión a la base de datos hacemos lo siguiente:

var mongoose = require('mongoose');

var db = mongoose.createConnection( 'mongodb://localhost:3000/test');

Ahora, veremos con se hace la construcción de la base de datos:

Hacemos uso de esquemas porque todo en mongoose inicia con un esquema, aquí es donde definimos la forma de los documentos y la fundición de las propiedades dentro de esa colección.

Todo en Mongoose comienza con un esquema. Cada esquema se asigna a una colección MongoDB y define la forma de los documentos dentro de esa colección. Recordemos también que una colección es un conjunto de documentos.

var Schema = mongoose.Schema;

var bookSchema = new Schema({

tittle: string,

content : string,

author : string

});

Cómo información general los tipos de esquemas permitidos son: Number, string, Date, Boolean, Mixed, Buffer, ObjectId, Array.

Para poder utilizar nuestro esquema definido anteriormente, lo que tenemos que hacer es convertirlo a un modelo donde lo podamos trabajar.

El primer parámetro es el nombre de nuestro modelo, y el segundo parámetro es el nombre del esquema que declaramos anteriormente.

var Book = mongoose.model('Book', bookSchema);

y a partir del modelo es cómo empezamos a crear las acciones para nuestros documentos.

<http://mongoosejs.com/docs/guide.html>

### **5 Sesiones en Node.js**