**M1S9: Algunos fundamentos de NodeJS (Estructura, NPM, Versionamiento, Export/Import)**

**¡Hola, nos encontramos nuevamente! Espero que estés tan emocionad@ como yo🥳. En esta semana, entregas tu segundo proyecto y ya espero verlo, recuerda que obtendrás una insignia que podrás mostrar en tus redes sociales y agregar a tu hoja de vida en el caso de que apruebes tus proyectos, demostrando así los objetivos que cumpliste para poderla alcanzar🏅.**

**Sigue adelante con los fundamentos de Node.JS y prepara tu proyecto🤓. Recuerda, cualquier duda o pregunta se la puedes hacer a tus coaches, están en Teams para ayudarte🙌🏻. ¡Seguimos...!**

**ÍNDICE**

* [Node.JS](https://github.com/U-Camp/BOOT-M1-SEM9/blob/main/README.md#nodejs)
* [Estructura](https://github.com/U-Camp/BOOT-M1-SEM9/blob/main/README.md#estructura)
* [NPM](https://github.com/U-Camp/BOOT-M1-SEM9/blob/main/README.md#npm)
* [Export / Import](https://github.com/U-Camp/BOOT-M1-SEM9/blob/main/README.md#export--import)

**NodeJS**

Si se habla de JavaScript del lado del servidor, entonces se habla de Node.JS, esta implementación del lenguaje surgió por la popularidad de JavaScript y debido a que en un principio JavaScript se ejecutaba únicamente del lado del cliente (navegadores web); Node.JS se concibió como un entorno de ejecución del lado del servidor (Runtime Environment) orientado a eventos asíncronos y diseñado para construir aplicaciones en red altamente escalables.

Node.JS ejecuta JavaScript del lado del servidor implementando el JavaScript Engine V8, el cual fue desarrollado por Google para usarlo en el navegador Chrome; los navegadores proveen acceso a Web APIs, sin embargo, Node.JS no requiere de su uso e implementación, ya que no hace uso de un entorno ejecución web como tal, sino que Node.JS provee un Runtime Environment construido en C++ mediante el cual es capaz de comunicarse con el sistema operativo, lo que impacta en su ejecución ya que permite el desarrollo de aplicaciones altamente escalables, soportando procesos intensos de entrada / salida, favoreciendo las aplicaciones en tiempo real, debido a la naturaleza No Bloqueante o Asíncrona de Node.

El hecho de que el motor V8 compila el código JavaScript a código máquina nativo en lugar de interpretarlo o ejecutarlo como *bytecode* permite velocidades de compilación y ejecución bastante rápidas, con *benchmarks* que son difíciles de igualar con otros lenguajes de programación. Estas ventajas competitivas le dan a Node.JS el valor necesario para elegirlo como tecnología del lado del servidor que se usará en este bootcamp.

**Estructura**

{

"name": "ejemplo",

"version": "0.0.1",

"description": "Ejemplo de archivo package.json",

"main": "ejemplos.js",

"keywords": [

"bootcamp",

"programación",

"desarrollo",

"software"

],

"author": "Armando <el\_silfo@hotmail.com> (https://github.com/curicaveri)",

"license": "MIT",

"dependencies": {

"bootstrap": "4.5.2"

}

}

*Ejemplo de archivo package.json*

Dentro del archivo package.json se puede especificar una sección de *scripts*, los cuales son pequeños comandos que se pueden interpretar por npm y permiten la ejecución de validaciones, pruebas unitarias o la ejecución misma del proyecto, que son tareas repetitivas que con el uso de scripts se automatizan. De igual manera hasta se pueden realizar tareas más complejas que incluyan el uso de bash scripts. Se escriben en formato JSON con la llave siendo el nombre del comando y el valor es el script que se va a ejecutar cuando ese comando se escriba con un npm run <comando>, lo que ocasiona que estos scripts se ejecuten en la misma terminal en la que se ejecutó el comando. Así mismo, es posible crear scripts personalizados, o scripts que llamen otros scripts, y finalmente *hooks*, los cuales son scripts personalizados que se ejecutan cuando determinada acción va a pasar, por ejemplo, cuando se va a hacer push sobre una rama de git específica o ejecutar pruebas unitarias antes de que se haga un commit, para validar que los cambios introducidos no rompan la aplicación.

**NPM**

Dentro de la especificación de Node.JS, se encuentra NPM, el cual es una especificación para el manejo de paquetes, es una herramienta que permite el correcto manejo de las dependencias de cada proyecto, el cual descarga las librerías que el proyecto vaya a usar, las instala y resuelve conflictos de versiones si los hay. Al momento de instalar Node.JS, automáticamente se instala NPM.

Dentro de Node.JS, el código se organiza por módulos o paquetes compilados, por lo que NPM permite la administración de éstos por medio de varios comandos de la terminal. Cuando se quiere instalar un paquete, simplemente se ejecuta el comando $ npm install <nombre\_paquete> y NPM se encarga de ver dentro de los repositorios de software e instala automáticamente el paquete y sus dependencias si es que tiene alguna; normalmente los paquetes y sus dependencias ya se encuentran deployados en el servidor NPM , por lo que nada más es buscar el paquete que se necesita, instalarlo con el comando anteriormente descrito y eso es todo, se puede comenzar a usar el paquete sin problemas.

“NPM” (Node Package Manager, Sistema de Gestión de Paquetes) maneja las dependencias de cualquier aplicación JavaScript. Esto lo hace gracias a un archivo especial llamado package.json en el cual se definen las dependencias específicas de la aplicación.

El archivo package.json es un objeto JSON que contiene distintas propiedades que definen el nombre de la aplicación y su versión, las cuales son las más importantes y son los únicos campos requeridos, ya que con base en éstas dos propiedades se puede publicar el proyecto en el registro público de NPM o en algún registro de un servidor NPM privado, lo cual permite que otros puedan descargarlo e instalarlo como dependencia de sus propios proyectos; también dentro del archivo se definen las dependencias del proyecto, la ubicación del repositorio del proyecto, los autores, la licencia, scripts y otros más, pero todos estos campos son opcionales aunque es muy recomendable usarlos. Como se puede ver package.json es como una clase de manifiesto del proyecto que asegura la integridad y la versión del mismo, con lo que permite manejar el proyecto de manera adecuada.

**EXPORT / IMPORT**

El manejo de las exportaciones e importaciones dentro de un proyecto de Node.JS se establece a partir de separar la funcionalidad, reutilizar código y tener una mejor estructura de consumo.

Todo esto ayuda a que, conforme el proyecto/aplicación avanza, darle mantenimiento puede ser mucho más fácil, incluso cuando estamos hablando de temas de debugging.

Por ello, hablaremos sobre la construcción de funciones y módulos.

Las funciones en un módulo específico pueden ser importados y exportados. Es decir, pueden ser utilizados por otros archivos que van fluyendo las definiciones entre ellas mismas.

Podemos considerar que, para realizar esta práctica, necesitamos:

* **Creación y manejo de módulos**

Para esto, necesitamos que los archivos estén creados con JavaScript, a partir del ambiente de desarrollo de Node.JS.

* **Exportación**

Para exportar, utilizaremos module.exports al final del archivo para especificar qué funciones van a permitir su acceso al público externo.

suma.js

const suma = (a,b) => {

return a + b

}

module.exports = { suma }

* **Importación**

Para importar, podremos situarnos en otro archivo de JavaScript (el cual será considerado otro módulo) y utilizaremos require, el cuál estará establecido al inicio del archivo.

app.js

const operacion = require("./suma.js")

operacion.suma(4.4) // => 8

La idea es ir separando los archivos por funcionalidad y darle contexto a cada archivo a través de su nombre. En este caso, suma.js entiende que dentro estará toda la funcionalidad que manipulará datos para adicionarse. Y, app.js se refiere al archivo principal del proyecto el cual utiliza este módulo para ejecutarse.

**¡Muy bien! Esta es la última semana del módulo 1 por lo tanto has llegado al final de los contenidos, por favor ve diseñando y desarrollando tu proyecto, espero verlo. Saludos, nos vemos en nuestra próxima U Class. 💻**