Programación con JavaScript II

Manejadores de paquetes y empaquetadores de módulos

Introducción

En un proyecto web complejo o de alto nivel, muchas veces se utilizan herramientas como *frameworks*, plugins, librerías y otros recursos; la cantidad de estos puede crecer según la complejidad con que se esté desarrollando. Estar pendiente de todo lo que necesita tu aplicación se va complicando en medida que el número de requerimientos crece, entonces, para aligerar un poco la estresante exigencia de mantener estas dependencias organizadas adecuadamente y sobre todo actualizadas, se sugiere el uso de un gestor de dependencias que las pueda gestionar de manera eficiente para los desarrolladores y administradores del proyecto.

A continuación, aprenderás qué es un gestor de dependencias y cuáles son los más populares para el lenguaje JavaScript; a su vez, conocerás la estructura moderna de una aplicación web, cómo se empaqueta y es interpretada por los navegadores de internet.

Explicación

Manejadores de paquetes

Una de las grandes ventajas de desarrollar una aplicación, a partir de piezas separadas, es que se pueden utilizar una o varias en diferentes programas. Para hacerlo, solamente será necesario copiar el código requerido en la nueva aplicación y utilizarlo; sin embargo, ¿qué pasa si se encuentra un error en él o se desea optimizar para un mejor funcionamiento?, lo más probable es que estos cambios se realicen solamente en el programa que se está trabajando actualmente y se olvide modificarlo en todos aquellos en donde también se haya utilizado.

Si se duplica código, pronto llegará el momento en que se pierda tiempo y recursos moviendo copias entre las aplicaciones que lo usan, intentando mantener actualizado el código en todas ellas. Es por esto que nace el concepto de paquetes, el cual, como lo define Haverbeke (2018), es un pedazo de código que puede ser distribuido, es decir, copiado e instalado. Estos paquetes pueden contener en su interior uno o más módulos y guardan la información referente a las dependencias con otros, así como la documentación de su uso. A diferencia del manejo manual, cuando se encuentra un problema en un paquete o se agrega nueva funcionalidad, este se actualiza y se distribuyen en una nueva versión.

En la medida que se van generando más paquetes y nuevas versiones de ellos, surge la necesidad de tener un mecanismo que nos permita almacenarlos y encontrarlos de una forma conveniente, al igual que instalarlos, actualizarlos, y administrarlos; estas características son las que **definen** a los manejadores de paquetes.

Los administradores de paquetes permiten cargar en el orden adecuado todas y cada una de las dependencias (paquetes, módulos, bibliotecas), en las que se basa el módulo o aplicación. Además de esto, como ya se mencionó, permiten administrar, actualizar, modificar y eliminar paquetes a medida que cambian las necesidades del proyecto. La mayoría de este tipo de herramientas, realizan un seguimiento de las dependencias con la ayuda de un árbol jerárquico

Aunque existen varias opciones en el mercado, y con diferente tipo de licenciamiento, son dos los que se han posicionado como los **principales manejadores de paquetes** y los definiremos enseguida.

NPM

Comenzaremos con Node (comúnmente conocido como NPM del inglés: Node Package Manager), es un gestor de paquetes en línea para módulos JavaScript. Según menciona Simões (2021) en el prestigiado portal ITDO, es el gestor de paquetes más popular hoy en día, es un producto de GitHub (Microsoft), predeterminado para el entorno de ejecución de Node o mejor dicho **Node.js**. Los desarrolladores de código abierto de todo el mundo utilizan NPM para la administración de paquetes, pero este proyecto no solamente está dedicado para el código abierto, sino que también muchas organizaciones a nivel mundial usan NPM para la gestión de su desarrollo privado. NPM consta de tres componentes distintos:

Explicación

- Sitio web: en el puedes descubrir paquetes de otros programadores, administrar tus paquetes y configurar otros aspectos, por ejemplo, permite configurar equipos para la administración de acceso a paquetes (públicos o privados).
- Interfaz de línea de comandos (CLI): se ejecuta desde una terminal de línea de comandos en cualquier computadora y es la forma en que NPM permite la interacción con los paquetes a la mayoría de los desarrolladores que utilizan el gestor.
- Registro: es una base de datos pública de paquetes JavaScript de código abierto y la metainformación necesaria para el uso de este código.

El NPM, como se comentó, esta incorporado a NodeJS, esto es, que al instalar NodeJS se instalará NPM de manera automática, y se puede obtener el software instalable desde el sitio https://nodejs.org/es/ sin costo.

Una vez realizada la descarga e instalación del software, se puede comprobar si Node y NPM se encuentran instalados correctamente abriendo una ventana de terminal de línea de comandos y ejecutar los siguientes comandos.

```
node  -v
npm   -v
```

Figura 1. Comandos para validación de la versión de Node y NPM.

Y se debería obtener un resultado similar al siguiente:

```
$ npm   -v
6.14.15
$ node  -v
v14.18.1
```

Figura 2. Validación de la versión de Node y NPM.

Algunos comandos básicos de npm que se deben aprender son:

Comando	Descripción
npm init	Crea el archivo **packaje.json**, el cual contiene información importante del proyecto que NPM usa para la identificación y gestión de las dependencias.
npm install	Permite instalar paquetes en el ambiente local. Los paquetes descargados son direccionados automáticamente a una carpeta llamada **node_modules**. Este comando lee el archivo **package.json** e intenta instalar todas las dependencias que encuentre. En caso de que un paquete ya se encuentre instalado, entonces intentará actualizarlo.

Explicación

npm install <nombre-paquete>	Puede instalar un paquete en particular, por ejemplo, **npm install jquery**. Ejemplo: $ npm i jquery + jquery@3.6.0 added 1 package from 1 contributor and audited 1315 packages in 57.594s 88 packages are looking for funding run `npm fund` for details found 13 vulnerabilities (6 moderate, 7 high) run `npm audit fix` to fix them, or `npm audit` for details
npm uninstall <nombre-paquete>	Desinstala un paquete del entorno local y lo elimina de la carpeta **node_module** así como de las dependencias del archivo **packaje.json**. Ejemplo: $npm uninstall jquery removed 2 packages and audited 1314 packages in 18.779s 88 packages are looking for funding run `npm fund` for details found 13 vulnerabilities (6 moderate, 7 high) run `npm audit fix` to fix them, or `npm audit` for details
npm update	Actualiza todos los paquetes y sus dependencias
npm update <nombre-paquete>	Actualiza un paquete en particular y sus dependencias, ejemplo: npm update parcel

Figura 3. Comandos básicos de NPM.

Yarn

Otro de los gestores de paquetes más utilizados en la industria es Yarn, que al igual que todos los de su tipo, permite utilizar y compartir código con otros desarrolladores, lo cual facilita el desarrollo de nuevos proyectos. Es un software en el que colaboran empresas como Facebook, Exponent, Google y Tilde, pero por ser de código abierto, si se tienen problemas, estos pueden ser informados, o bien, se puede contribuir en la solución de estos.

El código se comparte a través de paquetes (a veces también denominados módulos). Un paquete contiene aparte del código que se comparte, un archivo **package.json** que describe el paquete. Para comenzar a utilizarlo se requiere contar con NodeJS, debido a que depende de él, es importante mencionar que para cada sistema operativo existe una modalidad muy diferente de instalarlo, ten esto en cuenta, para que lo hagas de la mejor manera. Toda la documentación, se encuentra disponible desde el sitio oficial del producto.

Explicación

En el ecosistema de Node, las dependencias se colocan dentro de un directorio **node_modules** del proyecto. Sin embargo, esta estructura de archivos puede diferir del árbol de dependencias real, esto ocurre debido a que las dependencias que llegan a estar duplicadas se fusionan entre sí. El cliente NPM instala las dependencias en el directorio node_modules de forma no determinista, significa que, según el orden en que se instalen las dependencias, se va construyendo la estructura del directorio **node_modules**, lo que podría ocasionar que sea diferente de una instalación a otra. Estas diferencias causan errores de incompatibilidades entre uno y otro equipo, entre una y otra instalación, y toman mucho tiempo para ser detectados.

Como lo menciona Simões (2021), Yarn resuelve estos problemas que se relacionan con el control de versiones y el no determinismo mediante el uso de archivos de bloqueo y un algoritmo de instalación determinista y confiable. Como su nombre lo indica, estos archivos de bloqueo congelan las dependencias instaladas a una versión específica y garantizan que cada instalación genere como resultado una estructura de archivos en **node_modules** exactamente igual en todas las máquinas. El archivo de bloqueo escrito utiliza un formato conciso con claves ordenadas para garantizar que los cambios sean mínimos y la revisión sea simple.

El proceso de instalación de yarn se divide en tres pasos:

1. Resolución, comienza al resolver dependencias realizando solicitudes al registro y buscando de forma recursiva cada dependencia.
2. Búsqueda, posteriormente busca en un directorio de caché global para ver si el paquete necesario ya se ha descargado, si no es así, busca el paquete y lo coloca en la caché global para que pueda funcionar sin conexión y no necesite descargar dependencias más de una vez. Las dependencias también se pueden colocar en el control de código fuente como archivos comprimidos para instalaciones completas sin conexión.
3. Vinculación, finalmente, vincula copiando todos los archivos necesarios de la caché global en el directorio local **node_modules**.

La instalación de yarn se puede realizar con NPM. Simões (2021) menciona que un objetivo de Yarn es hacer más rápidas y confiables las instalaciones, además se simplifica el flujo de trabajo gracias a las siguientes características:

- Es compatible con los flujos de trabajo de NPM, permite la mezcla de registros.
- Tiene la habilidad de restringir las licencias de los módulos instalados.
- Es una API pública estable y confiable para el consumo a través de herramientas de complicación.
- Genera salidas del CLI entendibles y mínimas.

Explicación

Aunque existe una gran similitud entre ambas herramientas, Fernández (2019) acentúa que Yarn está muy orientada en seguridad y en velocidad. También hace mención de algunas de las principales diferencias que tiene respecto a NPM y que se deben considerar:

Yarn	NPM
Provee mayor seguridad.	No está orientado a la seguridad.
Algoritmos de determinismo en los archivos de bloqueo.	No utiliza archivos de bloqueo.
Decide solamente utilizar el **package.lock** para generar esta representación en el **directorio node-modules** cuando queremos bloquear versiones de las dependencias.	Se inclina por tener dos fuentes de verdad en el momento de actualizar las dependencias, tanto en el archivo **package.json** como en el **package.lock.**
Validador de licencias, el cual es útil para saber con exactitud los permisos de los paquetes que se están usando, además de otras cosas.	

Figura 4. Diferencias entre Yarn y NPM.

Simões (2021) también indica que Yarn utiliza el registro de NPM y si ya tienes instalado Node.JS lo único que tienes que hacer es abrir una ventana de línea de comando y ejecutar el comando de instalación.

```
npm install -g yarn
```

Figura 5. Comando de instalación de yarn.

```
npm install → yarn
```

Figura 6. Comando de configuración de yarn.

Explicación

Sin argumentos, el comando yarn leerá su package.json, buscará paquetes del registro NPM y completará su carpeta **node_modules**. Es equivalente a ejecutar **npm install**.

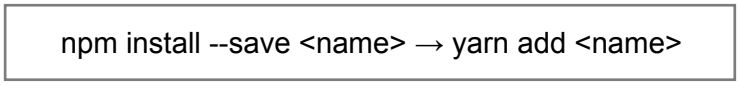

Figura 7. Comando de instalación de yarn.

Eliminamos el comportamiento de "dependencia invisible" de **npm install** <name> y dividimos el comando. Ejecutar **yarn add** <name> es equivalente a hacerlo con ***npm install --save*** <name>.

Si luego desea actualizar Yarn a la última versión, simplemente se debe ejecutar:

Figura 8. Comando para actualización de yarn.

Yarn descargará el binario más reciente de su sitio web y lo instalará en tus proyectos.

Algunos de los comandos de Yarn son:

Comando	**Descripción**
yarn help	Muestra la lista de comandos.
yarn init	Sirve para comenzar con un nuevo proyecto.
yarn install	Sirve para comenzar con un nuevo proyecto.
yarn add[paquete] *yarn add [paquete]@[versión]* *yarn add [paquete]@[etiqueta]*	Agrega una dependencia específica.
yarn add [paquete] --dev # dev dependencies *yarn add [paquete] --peer # peer dependencies*	Agrega una dependencia a diferentes categorías de dependencias.
yarn remove [paquete]	Elimina una dependencia.
yarn set version latest *yarn set version from sources*	Actualiza la versión de Yarn.

Figura 9. Comandos básicos de Yarn.

Explicación

Webpack

Kantor (2021) nos indica que, durante mucho tiempo, JavaScript existió sin una sintaxis de módulo a nivel de lenguaje sin embargo, esto no fue precisamente un problema, porque inicialmente los scripts que se desarrollaban eran pequeños y simples, por lo que no se necesitaba tener mucha estructura.

Pero, como hemos visto, con el tiempo los códigos se volvieron cada vez más complejos, por lo que la comunidad de desarrolladores tuvo que inventar diversas formas de organizar el código en módulos, o en bibliotecas especiales para cargar módulos a solicitud.

En JavaScript uno de los conceptos básicos es el sistema de módulos a nivel de lenguaje, el cual apareció en el estándar 2015 y ha evolucionado gradualmente desde entonces hasta llegar a la actualidad en que es compatible con todos los principales navegadores y en Node.js.

A medida que las aplicaciones desarrolladas van creciendo, será necesario dividirlas en múltiples archivos, a los cuales llamaremos “módulos”. Un módulo es un archivo, un script puede ser un módulo y este puede contener una clase o una biblioteca de funciones para un propósito específico.

Los módulos en ocasiones se cargan entre sí para lo cual deben usar directivas especiales **export** e **import** con el fin de hacer posible el intercambio de funcionalidad, llamar a funciones de un módulo y utilizarlas en otro. La palabra reservada **export** permite que las variables y funciones puedan ser accesibles desde fuera del módulo actual, de manera inversa, con la palabra reservada **import** se habilita la posibilidad de utilizar funcionalidades de otros módulos desde el actual.

Un ejemplo sería tener el archivo diHola.js que exporta una función:

```
export function diHola(usuario) {
 alert(`Hola, ${usuario}!´);
 }
```

Figura 10. Exportación de una función.

Posteriormente, para poder set utilizado, otro archivo podría importarlo y usarlo de la siguiente manera:

```
import {diHola} from './diHola.js';

alert(diHola); // function...
diHola('Juan'); // Hola, Juan!
```

Figura 11. Importación de una función.

Explicación

En este caso la directiva import realiza la carga del módulo por la ruta ./js/diHola.js, y asigna la función exportada diHola a la variable correspondiente.

La forma para poder visualizar el resultado en el navegador sería indicarle que un script deberá tratarse como módulo, y esto se puede realizar indicando en la etiqueta <script> de la siguiente manera:

```
<!doctype html>
 <script type="module">
 import {diHola} from './js/diHola.js';

 document.body.innerHTML = diHola('Juan');
 </script>
```

Figura 12. Exportación de una función.

De esta manera, el navegador buscará el módulo importado y lo evaluará automáticamente (y de ser necesario, sus importaciones), y posteriormente ejecuta el script que nos devolverá como resultado

Figura 13. Resultado de código modular.

Es importante mencionar que, para este momento, ya debes estar trabajando implementaciones en el protocolo http, es decir, utilizando un servidor web, si no es así, las directivas import y export no servirán si tratamos de usarlas con el protocolo file.

Pareciera ser que no hay mayor diferencia entre un módulo en comparación con un script como los que hemos trabajado hasta el momento, pero existen las siguientes características que los distinguen:

Kantor (2021) explica que los módulos siempre trabajan en modo estricto, un ejemplo muy común es asignarlo a una variable sin declarar, lo cual nos dará un error al emplear el siguiente código:

```
<script type="module">
 x = 5; // error
 </script>
```

Figura 14. Error generado por el modo estricto.

Explicación

El navegador nos arrojará el siguiente mensaje:

```
Uncaught ReferenceError: x is not defined
```

Figura 15. Error en consola.

Otra característica es que existe un alcance a nivel de módulo. Dicho de otra manera, las funciones y variables de nivel superior, no deben estar dentro de una función de un módulo, ya que no son visibles en otros scripts. Como puedes observar en el siguiente ejemplo, se importan desde index.html dos scripts **usuario.js** y **hola.js**, el segundo intenta usar la variable **usuario** declarada en **usuario.js.** Falla, porque es un módulo separado:

```
//usuario.js
let usuario = "Juan";
```

```
//hola.js
alert(usuario); // no existe la variable porque cada módulo tiene variables independientes
```

```
// index.html
<!doctype html>
<script type="module" src="usuario.js"></script>
<script type="module" src="hola.js"></script>
```

Figura 16. Alcance a nivel de módulo.

Los módulos deberán enlazarse a través de un export a lo que se quiera acceder desde afuera y hacer import de lo que necesiten de otros módulos. En otras palabras, con módulos usamos las expresiones import/export para no depender de variables globales. Como en este caso, el módulo usuario.js debe exportar la variable usuario y el módulo hola.js debe importarla desde el módulo usuario.js, en estos módulos se utiliza import o export en vez de depender de variables globales.

Esta es la variante correcta:

```
//usuario.js
export let usuario = "Juan";
```

```
//hola.js
import {usuario} from './usuario.js';
document.body.innerHTML = usuario; // Juan
```

```
//index.html
<!doctype html>
<script type="module" src="./js/hola.js"></script>
```

Figura 17. Módulos correctamente implementados.

Explicación

Ahora que ya tenemos un panorama general sobre lo que es un módulo y que una aplicación web puede formarse por un gran número de ellos es necesario conocer la manera de administrar estos módulos, además de que en la vida real rara vez se utilizan de forma pura, por lo general se agrupan con **empaquetadores de módulos**.

Para Valencia (2020) un empaquetador de módulos permite generar un archivo único con todos aquellos módulos que una aplicación necesita para funcionar. De manera general, nos explica que se pueden incluir todos tus archivos .js en un único archivo, también nos dice que incluso se pueden incorporar archivos de estilos .css hasta en el mismo archivo llamado ***.bundle.js**. Y por si fuera poco, se pueden realizar otras tareas para la optimización del código, tales como la minificación y la compresión.

Una implementación de estos empaquetadores de módulos es ***Webpack***, una herramienta que se puede configurar para poder realizar algunas de las tareas comunes del desarrollo de aplicaciones web en tareas automatizadas y preparar nuestra aplicación web para poder llevarla a ambientes productivos.

Es importante definir algunos conceptos básicos necesarios para la mejor comprensión de cómo funcionan este tipo de herramientas:

- Punto de Entrada: indican a la herramienta los archivos de entrada para generar los archivos ***.bundle.js.**
- Salida: indican al empaquetador el lugar donde se colocarán los paquetes ***.bundle.*** generados, ya sean de tipo JavaScript, CSS, HTML, etc.
- Cargadores: son las instrucciones que hacen posible que Webpack realice la carga, transforme y procese todas las entradas.
- Complementos: amplían las funcionalidades básicas que se incluyen por defecto en Webpack. Dotan a la herramienta de la capacidad para optimizar, minificar, y ofuscar el código de la aplicación entre otras actividades.

Para ejemplificar imagina que tienes el siguiente proyecto realizado en **Vue.js**, es importante que sepas que **Vue** es un framework de Javascript para la generación de aplicaciones de una sola página y Bootstrap (librería CCS).

La estructura inicial de este proyecto sería parecida a la siguiente:

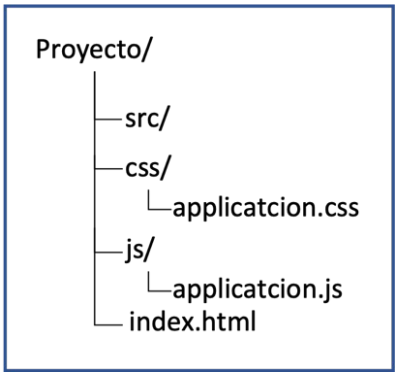

Figura 18. Estructura inicial de proyecto.

Explicación

En el archivo **src/css/aplicacion.css** se definen los estilos del proyecto. En el archivo src/js/aplicacion.js se define la lógica de negocio y los componentes que utiliza el framework **Vue.js** para hacer el despliegue de la aplicación. En el archivo **index.html** se define la vista y se hace la referencia a las librerías **Vue.js** y Bootstrap directamente desde el servidor CDN.

```
<!-- index.html -->
<!DOCTYPE html>
<html>
 <head>
<title>Webpack – Tutorial de inicio</title>
 <!—inicio de estilos CSS -->
 <link href="src/css/aplicacion.css" rel="stylesheet">
 <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
rel="stylesheet">
 <!—Termino de estilos -->
 </head>
 <body>
 <h1 class="text-center py-4">Mi Blog</h1>

 <!—Inician scripts -->
 <script src="src/js/applicacion.js"></script>
 <script src="https://cdn.jsdelivr.net/npm/vue"></script>
 <!—Terminan scripts -->
 </body>
</html>
```

Figura 19. Vista del archivo html.

Al incluir Webpack se genera un solo archivo **aplicacion.bundle.js**, como se muestra en la siguiente imagen:

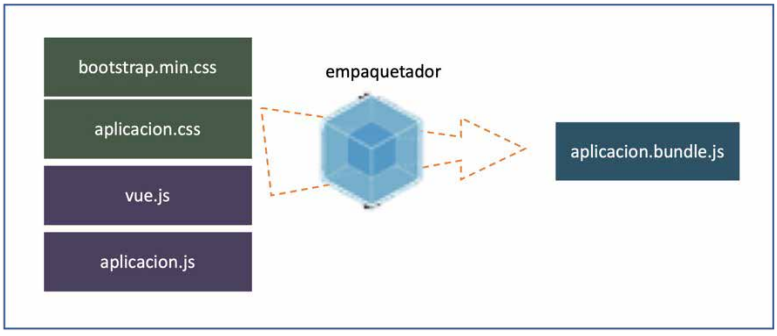

Figura 20. Empaquetado de archivos.

Explicación

En este archivo **aplicacion.bundle.js** resultante se empaquetarían los estilos de la librería de Bootstrap, el archivo de estilos del proyecto, la librería de **Vue** y el JavaScript del proyecto.

El Output o archivo de salida se generará sobre en la carpeta **dist** con el nombre seleccionado, para efectos de este ejemplo, seria **app.bundle.js.** Finalmente, es requerido modificar archivo **index.html** para hacer la referencia al archivo final **aplicacion.bundle.js.**

```
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Inicio</title>
</head>
<body>
<h1 class="text-center py-4">Mi Blog</h1>
<script src="dist/applicacion.bundle.js"></script>
</body>
</html>
```

Figura 21. Invocación de archivos empaquetados.

Al ingresar a tu proyecto se debería obtener el mismo resultado que la versión mostrada, solo que ahora en lugar de cargar dos archivos de los estilos y dos scripts de la aplicación, solo será necesario cargar uno en el cual se encuentran empaquetados los estilos y los scripts, esto ayuda porque el proyecto cargará de una manera más rápida.

Cierre

En este tema pudiste darte cuenta de que existen varias herramientas que ayudan a realizar proyectos web de gran escala, haciendo uso de las mejores prácticas de la industria del software, la importancia de este tema reside en que son conceptos que seguramente utilizarás más en el ambiente laboral. Si bien es cierto que no profundizamos demasiado en cada uno de ellos, pero lo que se abarcó te da una visión mucho más amplia de algunas de las herramientas que te permitirán destacar en el área profesional.

Ahora sabes la importancia de modularidad de tu código y los beneficios que esto trae, también aprendiste que los gestores de dependencias o de paquetes se han convertido en una herramienta indispensable al desarrollar aplicaciones, en especial si se trata de proyectos complejos a gran escala.

¿Consideras que conocer las mejores prácticas utilizadas en el mundo laboral te da una ventaja competitiva frente a otros desarrolladores?

Referencias bibliográficas

- Fernández, P. (2019). *Qué es Yarn.* Recuperado de https://openwebinars.net/blog/que-es-yarn/
- Haverbeke, M. (2018). *Eloquent JavaScript* (3rd ed.). Estados Unidos: No Starch Press.
- Kantor, I. (2021). *The JavaScript Languaje.* Recuperado de https://javascript.info/
- Simões, Ch. (2021). *Javascript ¿Qué es Yarn?* Recuperado de https://www.itdo.com/blog/javascript-que-es-yarn/
- Valencia, V. (2020). *¿Qué es Webpack?* Recuperado de https://medium.com/@victor.valencia.rico/qu%C3%A9-es-webpack-75cb56559759

Para saber más

Lecturas

- Alvarez, M. (2020). *Transpilado de JavaScript con Webpack y Babel.* Recuperado de https://desarrolloweb.com/articulos/transpilado-javascript-webpack.html
- npm Docs. (s.f.). *Packages and modules.* Recuperado de https://docs.npmjs.com/packages-and-modules
- Webpack. (s.f.). *Concepts.* Recuperado de https://webpack.js.org/concepts/
- Yarn. (s.f.). *Getting started.* Recuperado de https://yarnpkg.com/getting-started
- Simões, Ch. (2021). *Javascript ¿Qué es npm?* Recuperado de https://www.itdo.com/blog/javascript-que-es- NPM/

Videos

- jonmircha. (2020, 24 de marzo). *Curso JavaScript: 33. Módulos (import/export) - #jonmircha* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=0GEUyQXe3NI

Checkpoints

Asegúrate de:

- Desarrollar aplicaciones en módulos porque es la manera en que programarás en aplicaciones grandes en el campo laboral.
- Comprender la funcionalidad de NPM y Yarn y la forma de instalarlo en diferentes sistemas operativos.
- Identifica los comandos básicos de NPM para utilizarlos en el desarrollo de aplicaciones.

Requerimientos técnicos

- Computadora con acceso a internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

Lo primero que se tiene que hacer es instalar NPM:

1. Ve a la página de nodejs.org.
2. La página detectará tu sistema operativo, por tanto, solo tendrás que elegir que versión instalar. Elige la versión recomendada, la versión actual puede tener algunos módulos inestables o en desarrollo, por tanto, no se recomienda.
3. Una vez que haya descargado el paquete, instálalo en tu computadora siguiendo las instrucciones del instalador.
4. En una terminal de comando, ejecuta el comando **npm –v** para validar que se haya instalado correctamente la aplicación y puedas visualizar la versión.