DICCIONARIO

Índice

Microservicios……………..……………………………………………………………………………………………………………3

Docker………………………………………………………………………………………………………………………………………6

Npm vs Npx………………………………………………………………………………………………………………………………9

Que es un CMS……………………………………………………………………………………………………………………….11

Que es un Endpoint………………………………………………………………………………………………………………..12

Que es el destructuring en JavaScript….………………………………………………………………………………….13

Promesas JavaScript….………………………………………..………………………………………………………………….14

Microservicios

Un [microservicio](https://platzi.com/blog/microservicios-sin-servidor/-) es una **pequeña aplicación** que **se encarga de una parte de un software más complejo** de manera aislada, y se comunica con el resto del software mediante diferentes métodos: peticiones HTTP, o algún sistema de colas. [micro.js](https://github.com/zeit/micro) es una **librería muy pequeña** (alrededor de **100 líneas de código**) de [JavaScript](https://platzi.com/cursos/javascript/?utm_source=blog&utm_medium=tx&utm_campaign=20160818_JavaScript_EVERYONE&utm_content=textlink), que nos permite **usar**[**Node.js**](https://platzi.com/cursos/javascript/backend/?utm_source=blog&utm_medium=tx&utm_campaign=20160818_JavaScript_EVERYONE&utm_content=textlink)**para crear fácilmente microservicios** que funcionen **sobre el protocolo HTTP**, y **haciendo uso de Async/Await** y todas las características que se incluyeron en **ECMAScript 2015** para facilitarnos el programarlos.

**¿Qué son los microservicios?**

La arquitectura de microservicios o microservicios a secas, es un **distintivo sistema de desarrollo de software**que ha crecido en popularidad en los últimos años. De hecho, a pesar de que su extensión en uso no ha llegado tan allá donde sí lo ha hecho su teoría, muchos desarrolladores están descubriendo cómo esta forma de creación de software favorece el tiempo, rendimiento y estabilidad de sus proyectos.

Gracias a su **sencilla escalabilidad**, este método de arquitectura se considera especialmente adecuado cuando se tiene que procurar la compatibilidad con un amplio sector de diferentes plataformas (IoT, web, móvil, wearables…) o simplemente cuando no sabemos a ciencia cierta hacia qué tipo de dispositivos estamos orientando nuestro trabajo.

Si bien **no existe un estándar o definición formal para los microservicios**, hay ciertas características que nos ayudan a identificarlos. Básicamente el desarrollo de un proyecto que se base en este método conforma una aplicación o herramienta mediante la conjunción de diversos **servicios independientes que se despliegan según se vayan necesitando**. Por tanto, tendremos una aplicación modular a base de “pequeñas piezas”, que podremos ir ampliando o reduciendo a medida que se requiera.

Y la cuestión que muchos se harán es ¿Cómo comunico un servicio con otro? Y es como preguntar ¿Qué ordenador me compro? La respuesta será depende, tanto de nuestras habilidades, como de las preferencias o el sistema que mejor se adapte o nos soliciten… Muchos usan HTTP/REST con JSON o Protobuf, pero como los desarrolladores sois vosotros, usad aquello que mejor consideréis, aunque REST se posiciona en extensión de uso al ser uno de los métodos de integración cuya curva de aprendizaje y uso es increíblemente rápida respecto a otros protocolos.

**Comprendiendo la Arquitectura de Microservicios.**

Pero para comprender la arquitectura basada en microservicios, lo mejor es considerar su opuesto, su némesis, **la arquitectura monolítica.** En ésta, la aplicación se desarrolla como **una única unidad que no necesitará de ningún componente externo para funcionar**. Por ejemplo, en una aplicación como la que desarrollamos hace unos días en Meteor que cuenta con un lado del cliente y un lado servidor, el “monolito” cliente se encargará de las peticiones HTTP, ejecutar la lógica y recibir/actualizar la información desde la base de datos.

El problema de las aplicaciones monolíticas es que todos los “ciclos de cambio” **están vinculados unos a otros**, por lo que la más mínima modificación en una remota sección de la app conllevaría a la creación y despliegue de una versión completamente nueva, con el gasto de recursos correspondiente. Por no hablar de si queremos escalar un apartado específico del proyecto o directamente la aplicación al completo; y aquí es donde los desarrolladores han visto el potencial de los microservicios.

**Características del Software de microservicios.**

Como decía no hay un estándar en estas arquitecturas, pero sí que podemos destacar varias características comunes en mayor o menor medida:

1. El **software construido en base a microservicios se podrá descomponer en varias partes funcionales independientes.** Siendo así, cada uno de estos servicios podrá ser desplegado, modificado y re-desplegado sin comprometer los otros aspectos funcionales de la aplicación; y como resultado en caso de necesitarlo, sólo tendremos que modificar un par de servicios en lugar de redesplegar toda la aplicación al completo nuevamente.
2. La forma en la que **se organizan los microservicios suele ser en torno a las necesidades, capacidades y prioridades del cliente o negocio en el que se implantará.** A diferencia de un entorno monolítico donde cada equipo de trabajo tiene un enfoque específico sobre un apartado de la aplicación, en la arquitectura de microservicios se utilizan módulos multifuncionales, adaptando así un módulo común a todos para que ofrezca un servicio determinado. El ahorro en tiempo de desarrollo es inmenso, por no hablar de la comodidad a la hora de programar tareas de mantenimiento, donde podemos revisar un módulo mientras el resto del equipo de trabajo no ve interrumpida su jornada.
3. El funcionamiento del software de microservicios **puede parecerse al sistema de trabajo clásico de UNIX**(se recibe una petición, se procesa y se genera una respuesta consecuente. Al contrario que los entornos ESB (Enterprise Service Buses, o Bus de servicio empresarial) donde se utilizan equipos para enrutar mensajes, redireccionar tráfico, aplicar reglas de denegación de acceso etc… Se podría decir que la arquitectura de microservicios cuenta con puntos finales “inteligentes” que procesan la información en término y aplican la lógica establecida por el desarrollador.
4. La arquitectura de microservicios **mantiene un sistema similar a un gobierno descentralizado**, donde cada módulo contará por ejemplo con su propia base de datos, en lugar de acudir todos a la misma sobrecargándola así de solicitudes y arriesgándonos a que si falla ésta, todas la aplicación caiga.
5. Cuando varios servicios están comunicados entre sí, por lo general **contarán con un sistema de aviso y actuación si alguno de éstos servicios llega a fallar**(como mostrar una advertencia, enviar un mail a soporte, avisar a los usuarios de un fallo temporal, etc…), filtrando adecuadamente la información destinada a este módulo y favoreciendo la correcta gestión de los recursos entre los módulos funcionales restantes.

**Pros / Contras.**

Conociendo ya qué son los microservicios, sus características principales y algunos ejemplos en los que se han implementado con éxito, os dejo algunos pros y contras de esta tecnología:

**Pros:**

* Otorga a los desarrolladores libertad de desarrollar y desplegar servicios de forma independiente.
* Un microservicio se puede desarrollar con un equipo de trabajo mínimo.
* Se pueden usar diferentes lenguajes de programación en diferentes módulos.
* Fácil integración y despliegue automático (por ejemplo con Jenkins…).
* Fácil de entender y modificar, por lo que la integración de nuevos miembros al equipo de desarrollo será muy rápida.
* Los desarrolladores podrán hacer uso de las tecnologías más actuales.
* El uso de contenedores hará el desarrollo y despliegue de la app mucho más rápido.
* Funcionalidad modular, con lo que la modificación de un módulo no afectará al funcionamiento del resto.
* Fácil de escalar e integrar con aplicaciones de terceros.

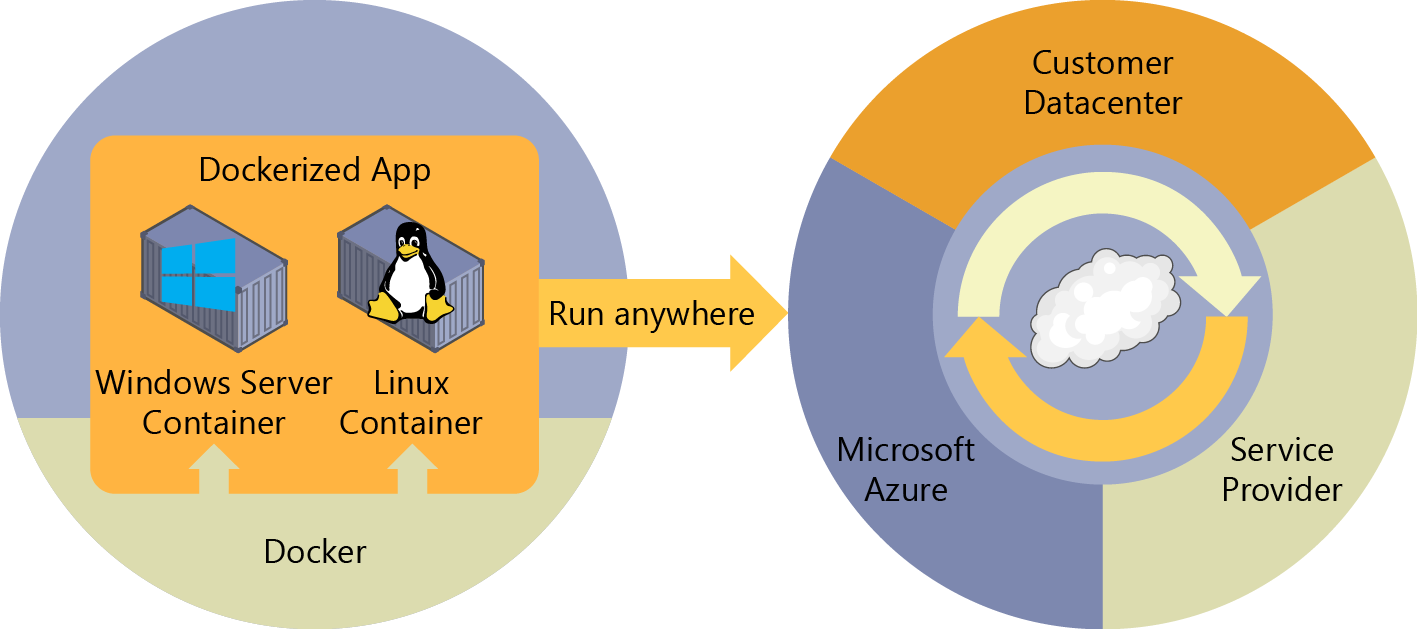
**Contras:**

* Las pruebas o testeos pueden resultar complicados debido al despliegue distribuido.
* Un gran número de servicios puede dar lugar a grandes bloques de información que gestionar.
* Será labor de los desarrolladores lidiar con aspectos como la latencia de la red, tolerancia a fallos, balanceo de carga, cantidad de formatos admitidos, etc…
* Sistema distribuido puede llegar a significar doble trabajo.
* Si se cuenta con un gran número de servicios, integrarlos y gestionarlos puede resultar muy complejo.
* Esta tecnología suele incurrir en un alto consumo de memoria.
* Fragmentar una aplicación en diferentes microservicios puede llevar muchas horas de planificación (y casi podría considerarse un arte).

DOCKER

**¿Qué es Docker?**

[Docker](https://www.docker.com/) es un [proyecto de código abierto](https://github.com/docker/docker) para automatizar la implementación de aplicaciones como contenedores portátiles y autosuficientes que se pueden ejecutar en la nube o localmente. Docker es también una [empresa](https://www.docker.com/) que promueve e impulsa esta tecnología, en colaboración con proveedores de la nube, Linux y Windows, incluido Microsoft.



**Figura 1-2**. Docker implementa contenedores en todas las capas de la nube híbrida

Tal y como se muestra en el diagrama anterior, los contenedores de Docker se pueden ejecutar en cualquier lugar, localmente en el centro de recursos de cliente, en un proveedor de servicios externo o en la nube, en Azure. Los contenedores de imágenes de Docker también se pueden ejecutar de forma nativa en Linux y Windows. Sin embargo, las imágenes de Windows solo pueden ejecutarse en hosts de Windows y las imágenes de Linux pueden ejecutarse en hosts de Linux y hosts de Windows (con una máquina virtual Linux de Hyper-V, hasta el momento), donde host significa un servidor o una máquina virtual.

Los desarrolladores pueden usar entornos de desarrollo en Windows, Linux o macOS. En el equipo de desarrollo, el desarrollador ejecuta un host de Docker en que se implementan imágenes de Docker, incluidas la aplicación y sus dependencias. Los desarrolladores que trabajan en Linux o en Mac usan un host de Docker basado en Linux y solo pueden crear imágenes para contenedores de Linux. (Los desarrolladores que trabajan en Mac pueden editar código o ejecutar la interfaz de la línea de comandos, o CLI, de Docker en macOS, pero en el momento de redactar este artículo, los contenedores no se ejecutan directamente en macOS). Los desarrolladores que trabajan en Windows pueden crear imágenes para contenedores de Windows o Linux.

Para hospedar contenedores en entornos de desarrollo y proporcionar herramientas de desarrollador adicionales, Docker entrega [Docker Community Edition (CE)](https://www.docker.com/community-edition) para Windows o macOS. Estos productos instalan la máquina virtual necesaria (el host de Docker) para hospedar los contenedores. Docker también pone a disposición [Docker Enterprise Edition (EE)](https://www.docker.com/enterprise-edition), que está diseñado para el desarrollo empresarial y se usa en los equipos de TI que crean, envían y ejecutan aplicaciones críticas para la empresa en producción.

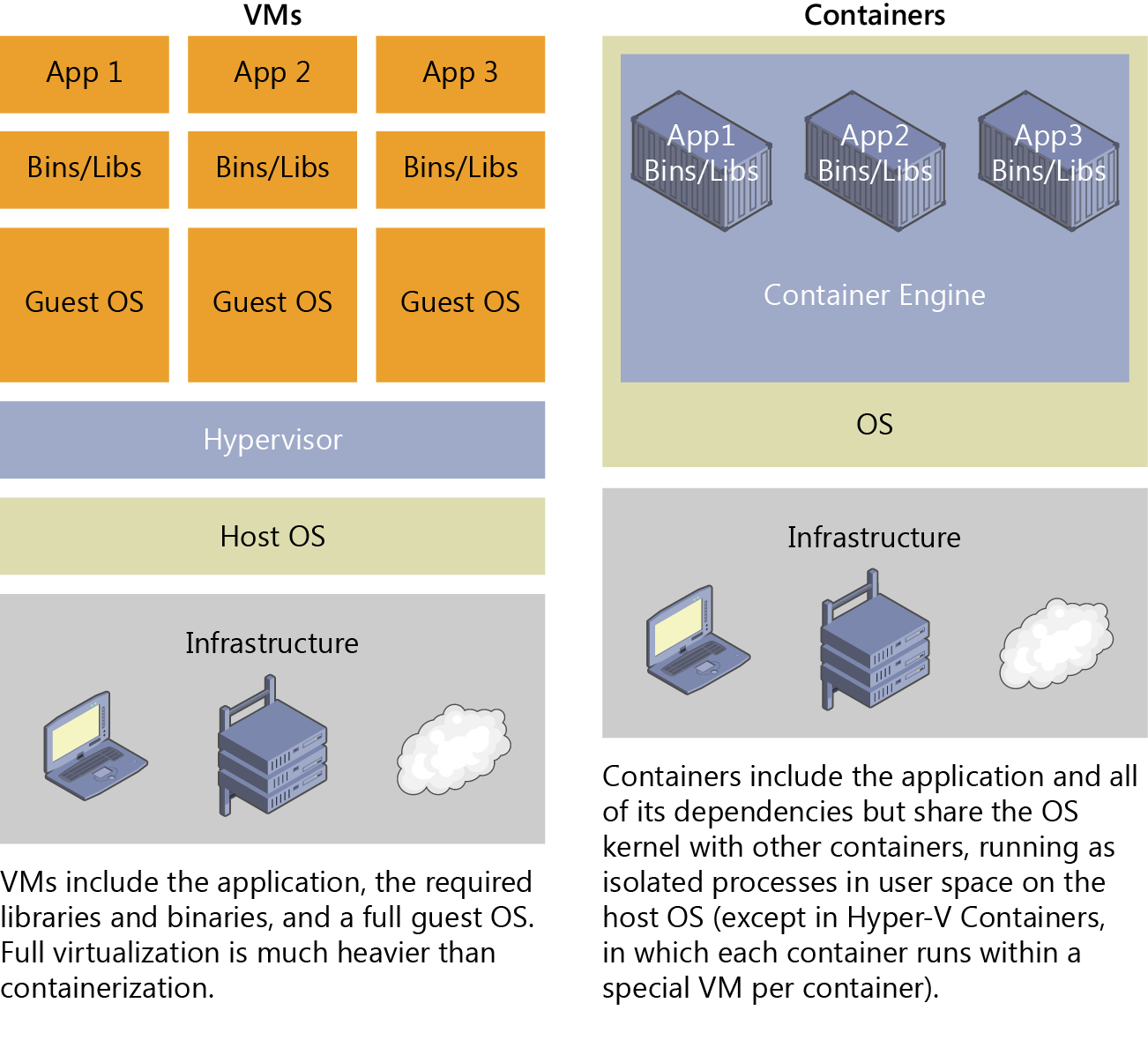
Para ejecutar [contenedores de Windows](https://docs.microsoft.com/es-es/virtualization/windowscontainers/about/), hay dos tipos de tiempos de ejecución:

* Los **contenedores de Windows Server** ofrecen aislamiento de aplicaciones a través de tecnología de aislamiento de proceso y de espacio de nombres. Un contenedor de Windows Server comparte el kernel con el host de contenedor y con todos los contenedores que se ejecutan en el host.
* Los **contenedores de Hyper-V** amplían el aislamiento que ofrecen los contenedores de Windows Server mediante la ejecución de cada contenedor en una máquina virtual altamente optimizada. En esta configuración, el kernel del host del contenedor no se comparte con los contenedores de Hyper-V, lo que proporciona un mejor aislamiento.

Las imágenes de estos contenedores se crean y funcionan de la misma manera. La diferencia radica en cómo se crea el contenedor desde la imagen ejecutando un contenedor de Hyper-V que requiere un parámetro adicional. Para más información, vea [Contenedores de Hyper-V](https://docs.microsoft.com/es-es/virtualization/windowscontainers/manage-containers/hyperv-container).

**Comparación de los contenedores de Docker con las máquinas virtuales**

En la figura 1-3 se muestra una comparación entre las máquinas virtuales y los contenedores de Docker.



**Figura 1-3**. Comparación de las máquinas virtuales tradicionales con los contenedores de Docker

Tal y como se muestra en el diagrama anterior, en el caso de las máquinas virtuales, hay tres capas base en el servidor host. De abajo hacia arriba: infraestructura, sistema operativo host y un hipervisor. Encima de todo eso, cada máquina virtual tiene su propio sistema operativo y todas las bibliotecas necesarias. Por otro lado, para Docker, el servidor host solo tiene la infraestructura y el sistema operativo. Además, el motor de contenedor mantiene los contenedores aislados, pero les permite compartir los servicios del sistema operativo de base única.

Dado que los contenedores requieren muchos menos recursos (por ejemplo, no necesitan un sistema operativo completo), se inician rápidamente y son fáciles de implementar. Esto permite tener una mayor densidad, lo que significa que se pueden ejecutar más servicios en la misma unidad de hardware, reduciendo así los costos.

Como efecto secundario de la ejecución en el mismo kernel, obtiene menos aislamiento que las máquinas virtuales.

El objetivo principal de una imagen consiste en garantizar el mismo entorno (dependencias) entre las distintas implementaciones. Esto significa que puede depurarlo en su equipo y, después, implementarlo en otra máquina con el mismo entorno garantizado.

Una imagen de contenedor es una manera de empaquetar una aplicación o un servicio e implementarlo de forma confiable y reproducible. Podría decir que Docker no solo es una tecnología, sino también una filosofía y un proceso.

Al usar Docker, no escuchará a los desarrolladores decir "Si funciona en mi máquina, ¿por qué no en producción?". Pueden decir simplemente "Se ejecuta en Docker", porque la aplicación de Docker empaquetada puede ejecutarse en cualquier entorno de Docker compatible, y se ejecuta de la forma prevista en todos los destinos de implementación (como desarrollo, control de calidad, almacenamiento provisional y producción).

**Una analogía simple**

Quizás una analogía simple puede ayudar a entender el concepto básico de Docker.

Vamos a remontarnos a la década de 1950 por un momento. No había ningún procesador de texto y las fotocopiadoras se usaban en todas partes (o casi).

Imagine que es responsable de enviar lotes de cartas, según proceda, para enviarlas por correo a los clientes en papel y sobres reales, que se entregarán físicamente en la dirección postal de cada cliente (entonces no existía el correo electrónico).

En algún momento, se da cuenta de que las cartas no son más que una composición de un conjunto grande de párrafos, que se eligen y ordenan según proceda, según el propósito de la carga, por lo que diseña un sistema para emitir cartas rápidamente, esperando conseguir una increíble mejora.

El sistema es simple:

1. Comience con un conjunto de hojas transparentes que contienen un párrafo.
2. Para emitir un conjunto de cartas, elija las hojas con los párrafos necesarios, apílelas y alinéelas para que queden y se lean bien.
3. Por último, colóquelas en la fotocopiadora y presione inicio para producir tantas cartas como sean necesarias.

Por tanto, para simplificar, esa es la idea principal de Docker.

En Docker, cada capa es el conjunto resultante de los cambios que se producen en el sistema de archivos después de ejecutar un comando, como instalar un programa.

Por lo tanto, al "Examinar" el sistema de archivos después de que se ha copiado la capa, verá todos los archivos, incluida la capa cuando se instaló el programa.

Puede pensar en una imagen como un disco duro de solo lectura auxiliar listo para instalarse en un "equipo" donde el sistema operativo ya está instalado.

De forma similar, puede pensar en un contenedor como el "equipo" con el disco duro de imagen instalado. El contenedor, como un equipo, se puede apagar o desactivar.

Npm vs Npx

NPM por sí solo no ejecuta ningún paquete, de hecho, no ejecuta ningún paquete. Si desea ejecutar un paquete utilizando NPM, debe especificar ese paquete en su archivo packages.json.

Cuando los ejecutables se instalan a través de paquetes NPM, NPM los vincula con ellos:

* local las instalaciones tienen "enlaces" creados en el directorio ./node\_modules/.bin/.
* global las instalaciones tienen "enlaces" creados desde el directorio global bin/ (por ejemplo, /usr/local/bin) en Linux o en %AppData%/npm en Windows.

Uno podría instalar un paquete localmente en un proyecto determinado:

npm install some-package

Ahora digamos que desea que NodeJS ejecute ese paquete desde la línea de comandos:

$ some-package

Lo anterior fallará. Solo instalado globalmente paquetes pueden ejecutarse escribiendo su nombre solo.

Para solucionar este problema y ejecutarlo, debe escribir la ruta local:

$ ./node\_modules/.bin/some-package

Técnicamente puede ejecutar un paquete instalado localmente editando su archivo packages.json y agregando ese paquete en la sección scripts:

{

"name": "whatever",

"version": "1.0.0",

"scripts": {

"some-package": "some-package"

}

}

Luego ejecute el script usando npm run-script (o npm run):

npm run some-package

NPX:

npx comprobará si <command> existe en $PATH, o en los binarios del proyecto local, y lo ejecutará. Entonces, para el ejemplo anterior, si desea ejecutar el paquete some-package instalado localmente, todo lo que necesita hacer es escribir:

npx some-package

Otra major ventaja de npx es la capacidad de ejecutar un paquete que no estaba instalado previamente:

$ npx create-react-app my-app

**En resumen:**

**NPM - Administra paquetes pero no hace la vida fácil ejecutando cualquiera.**

**NPX - Una herramienta para ejecutar paquetes de nodo.**

**Qué es un CMS**

## CMS son las siglas de Content Management System, o lo que viene a ser un ****sistema de gestión de contenidos para páginas web****.

Un CMS es un software desarrollado para que cualquier usuario pueda administrar y gestionar contenidos de una web con facilidad y sin conocimientos de programación Web. Previamente un programador deberá desarrollar la página web en base al CMS más apropiado para el tipo de web, dejando las tareas posteriores de gestión en manos del usuario final.

No todas las páginas Web son iguales, y para ello **tenemos varios tipos de CMS según el tipo de página que necesitemos**. Los hay para blogs, páginas corporativas, inmobiliarias, tiendas on-line, noticias o magazines, contenidos multimedia, y así un largo etc. Algunos CMS son desarrollos con proyectos open source con complementos opcionales «premium» y otros funcionan bajo licencia de pago..

### Los CMS más importantes se pueden dividir en dos grandes grupos:

**CMS para páginas web**:

* WordPress
* Drupal
* Joomla

**CMS para tiendas on-line**:

* Prestashop
* WordPress + WooCommerce
* Magento

### Las ventajas más importantes de los CMS son:

* Los programadores y la comunidad están organizados para desarrollar y lanzar nuevas versiones de estos CMS de forma periódica.
* Existen miles de módulos (o plugins) desarrollados por otros programadores independientes que nos sirven para complementar el CMS base con nuevas funcionalidades de forma gratuita o de pago.
* Están diseñados para que una vez implementado el proyecto web, pueda ser el cliente final quien haga la gestión de contenidos, dejando al desarrollador para tareas de modificaciones, actualizaciones y mantenimiento.

**Que es un EntryPoint - Endpoint**

**Entry point** es la URL que el visitante habrá ingresado en su navegador para ver su aplicación o sitio. Antiguamente, cada sección de un sitio web era un entrypoint

- home.html

- galeria.html

- contacto.html

- about us.html

Con el advenimiento de librerías que proveen ruteo (Backbone, Ract, Angular, Vue) se ha vuelto trivial servir aplicaciones de una sola página (SPA) en donde el visitante llega a un landing y desde ahí puede visitar el resto del sitio, *dando la impresión de que navega* entre páginas distintas cuando, en realidad, *sigue siendo el mismo landing* que le muestra distintas secciones según la url que vaya solicitando. El entrypoint contendrá la funcionalidad de actual como router y mostrar distinto contenido según la url solicitada, "interceptando" los links internos.

El visitante ve que la URL va cambiando y realmente siente que se está moviendo entre links, pero no ha dejado el mismo entrypoint.

- Home

- galería

- contacto

- about us

Hay un delicado equilibro entre tener muy pocos entrypoints con dependencias que no necesitan, o muchos con el manteniemiento que ello conlleva.

**Los endpoints** son las URLs de un API o un backend que responden a una petición. Los mismos entrypoints tienen que calzar con un endpoint para existir. Algo debe responder para que se renderice un sitio con sentido para el visitante. Por cada entrypoint esperando la visita de un usuario puede haber docenas de endpoints sirviendo los datos para llenar cada gráfico e infografía que se despliega en el entrypoint.

La diferencia entre entrypoint y endpoint es que los **endpoints no están pensados para interactuar con el usuario final**. Usualmente sólo devolverán json, o no devolverán nada. Y más que frecuentemente, un entrypoint hará varios llamados a distintos endpoints para mostrar estadísticas, galerías, últimos comentarios, etc.

Adicionalmente, se asume que cuando se habla de un endpoint estamos en un entorno **RESTful**, por lo cual (a diferencia del uso de un browser), el cliente *puede usar un mismo endpoint con distintos verbos*.

Un mismo endpoint, por ejemplo:

/users

Va a devolver una lista de usuarios si usas el verbo GET, y va a crear un usuario si usas el verbo POST. El endpoint por sí mismo no dice nada de las acciones que puedes hacer con él.

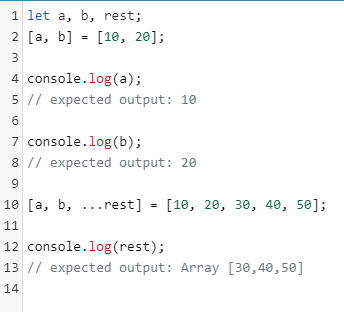
La existencia de endpoints usualmente es proporcional a la cantidad de entidades que quieres modelar en tu backend o en tu API. Digamos que por cada entidad debiera existir al menos un endpoint, y por cada uno de ellos, podrías realizar las acciones de crear, leer, actualizar y borrar datos. Y luego, si consideras que cada relación entre dos entidades da lugar a otro endpoint, puede que tengas muchos más. De nuevo, dado el modelo User, si cada usuario tiene N galerías, el **endpoint**

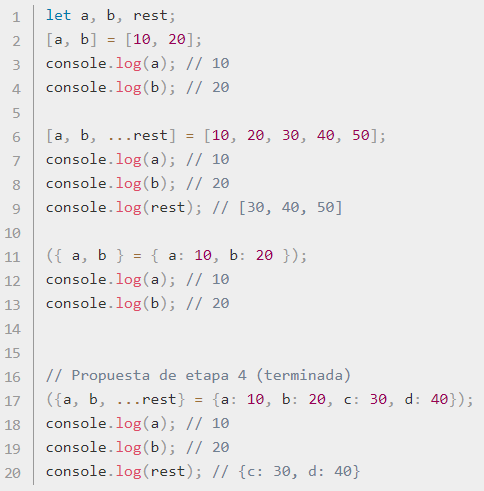
users/{id}/gallery

Te va a devolver las galerías del usuario que has pedido. De nuevo en un simple json, porque no está pensado para que el usuario final lo lea directamente.

**Destructuring JavaScript**

La sintaxis de **asignación desestructurante**(*destructuring assignment*) es una expresión que posibilita la extracción de datos de *arrays, o* de propiedades de objetos, en variables distintas.





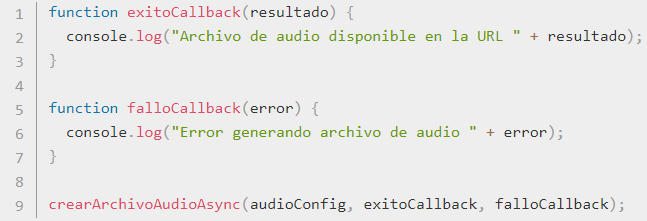
<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Destructuring_assignment>

**Promesas**

Una [Promise](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise) es un objeto que representa la terminación o el fracaso de una operación asíncrona. Dado que la mayoría de las personas consumen promises ya creadas, esta guía explicará primero cómo consumirlas, y luego cómo crearlas.

Esencialmente, una promesa es un objeto devuelto al cuál se adjuntan funciones callback, en lugar de pasar callbacks a una función.

Considera la función crearArchivoAudioAsync(), el cuál genera de manera asíncrona un archivo de sonido de acuerdo a un archivo de configuración, y dos funciones callback, una que es llamada si el archivo de audio es creado satisfactoriamente, y la otra que es llamada si ocurre un error. El código podría verse de la siguiente forma:



... las funciones modernas devuelven un objeto promise al que puedes adjuntar funciones de retorno (callbacks). Si crearArchivoAudioAsync fuera escrita de manera tal que devuelva un objeto promise, usarla sería tan simple como esto:



Lo cuál es la versión corta de:



Llamamos a esto una llamada a función asíncrona. Esta convención tiene varias ventajas. Exploraremos cada una de ellas.

**Garantías**

A diferencia de las funciones callback pasadas al "viejo estilo", una promesa viene con algunas garantías:

* Las funciones callback nunca serán llamadas antes de la [terminación de la ejecución actual](https://developer.mozilla.org/es/docs/Web/JavaScript/EventLoop#Ejecutar_hasta_completar) del bucle de eventos de JavaScript.
* Las funciones callback añadidas con then() incluso después del éxito o fracaso de la operación asíncrona serán llamadas como se mostró anteriormente.
* Múltiples funciones callback pueden ser añadidas llamando a then() varias veces. Cada una de ellas es ejecutada una seguida de la otra, en el orden en el que fueron insertadas.

Una de las grandes ventajas de usar promises es el encadenamiento, explicado a continuación.

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