BackEnd - Node.JS

1. Páginas Estáticas vs Páginas Dinámicas
2. Por qué Node.
3. ¿JavaScript del lado del servidor? Porque usar Node?
4. Instalación Node.
5. Comprobación de la instalación
6. Primeros pasos.
7. Comentarios en Node.
8. Configuración del entorno de desarrollo.
9. Variables
10. Concepto de Bloqueo.
11. Bibliotecas de uso habitual.
12. Utilizando require ,
13. Request y Response

Páginas web estáticas vs páginas web dinámicas

Cuando se trata de desarrollar un sitio web lo primero que tenes que considerar es cómo lo querés construir, como un sitio web estático o como un sitio web dinámico. Pero, ¿qué hace que un sitio web sea categorizado "*estático*" o "*dinámico*"?

Te invito que me acompañes en este comparativo para que conozcas cómo distinguir las diferencias entre estos dos tipos de páginas web y que puedas decidir cuál es el que te conviene según tu necesidad. Pero antes de entrar en detalles de cada uno de estos tipos de sitios web, primero debes entender cómo funciona la comunicación en Internet cuando queremos ver una página web.

Comunicación entre servidores y navegadores web

Para entender la comunicación más básica que ocurre en Internet, debemos saber que en ella están involucrados un servidor web, como IIS, Apache o NGINX, que contiene los archivos HTML, CSS y JavaScript de las páginas web, y un cliente, el navegador (*Chrome, Firefox, Edge*).

El servidor web y el cliente se comunican a través de los protocolos **HTTP** (*Hypertext Transfer Protocol*), el protocolo de Transferencia de Hipertexto y la versión segura **HTTPS** (*HyperText Transfer Protocol Secure*), una serie de reglas que permiten la transferencia de información a través de archivos en la Internet. La diferencia principal entre HTTP y HTTPS, es que, con este último, la comunicación entre el servidor y el cliente es cifrada permitiendo que la transmisión de los datos sea segura, brindando mayor integridad y confidencialidad a los mismos.

La comunicación entre ellos inicia cuando el usuario, a través del navegador, ingresa la dirección de un sitio web (*conocida como la URL del inglés Uniform Resource Locator ó LRU que se refiere al Localizador de Recursos Uniforme en español*), generando una petición al servidor web para encontrar los archivos de dicha página en dónde están alojados (*HTTP request*). Luego, el servidor web responde a la petición y devuelve los archivos del sitio web (*HTTP response*). La respuesta llega al navegador como una copia en formato HTML de la página web, y es en este momento cuando el usuario puede verla en su pantalla.

Qué es una página web estática

Lo primero que debemos entender es ¿a qué nos referimos con la palabra estática en el contexto de una página web?, y no es más que aquello que en el ámbito del código fuente del sitio web se encuentra fijo, no se mueve ni cambia de ninguna manera. Cuando hablamos de "estático" también podemos referirnos a que la página web tiene un número fijo de página, es decir, que tal como fue

diseñada y almacenada en el servidor web, así mismo la recibe el navegador y la ve el usuario, como un número fijo de páginas HTML.
Una página web estática está compuesta por archivos HTML individuales por cada página que son pre-generados y presentados al usuario a través del navegador de la misma forma.
Como una página web estática básica está compuesta por elementos como títulos, cuadros de textos, etiquetas, imágenes y otros elementos multimedia, un usuario solo puede interactuar con una página web estática a través de lo que permiten los elementos HTML, por ejemplo haciendo clic en enlaces, botones o rellenando formularios como el clásico formulario de subscripción.
No son tan complejos técnicamente como un sitio web dinámico, pero tampoco son tan versátiles y efectivos cuando se trata de entregar funcionalidad. En pocas palabras, en una página web estática, verás la misma información, diseño y contenido cada vez que la visites, a menos que alguien aplique cambios al código fuente de forma manual.
Si quisieras crear una página web estática solo necesitas un editor de texto como el Bloc de notas y saber de HTML y CSS, no es necesario utilizar entornos de desarrollo complejos.

Ventajas de una página web estática

Entre las ventajas de una página web estática podemos mencionar:

- **El costo inicial de una página web estática puede ser mucho menor que al de una dinámica**.

Por su naturaleza estática, la complejidad y tiempo de desarrollo es menor porque no requiere del uso de lenguajes de programación o bases de datos, y por ende su costo monetario es más bajo.

- **Son muy flexibles cuando se trata del diseño**.

Dado a su naturaleza independiente, cada página puede tener un diseño diferente. No es necesario un solo diseño para múltiples tipos de contenido, lo que en los sitios web dinámicos se le conoce como plantillas (templates).

- **Los tiempos de carga son muy rápidos**.

Ya que los sitios web estáticos son construidos previamente. No implica ejecución de scripts o secuencias de comandos complejas, bases de datos ni análisis de contenido a través de lenguajes de plantillas, etc.

Sin embargo, con la revolución del Jamstack, los generadores de sitios web estáticos como Jekyll, GatsbyJS o Eleventy, y los Headless CMS como Netlify CMS, Siteleaf o Forestry, y además la incorporación de CDN (*Content Delivery Network en inglés*) para gestionar los recursos multimedia,

se puede generar un aumento en el tiempo de carga de una página web estática dependiendo de sus características.

Desventajas de una página web estática

Algunas desventajas de elegir una página web estática son:

- **Una página web estática puede ser más difícil de actualizar**.

Para usuarios no técnicos, una vez que la página es creada, hacer pequeños ajustes en el contenido puede representar un desafío a menos que estén familiarizados con HTML, CSS y el código del sitio web en general. Si no es así, es posible que deban pedirle al desarrollador que la creó originalmente, que realice los cambios que necesitan.

- **Agregar contenido a la página web o realizar actualizaciones puede incurrir en costos adicionales**.

Esto puede verse como una consecuencia de la desventaja anterior. Es decir que, con el tiempo, el mantenimiento de un sitio estático puede generar costos de mantenimiento continuo que podrían evitarse si tuvieras una página web dinámica.

- **Agregar nuevas páginas o funcionalidades a una web estática puede ser más difícil que hacerlo para una web dinámica**.

Por ejemplo, si creas una página web para promocionar productos de tecnología, cada vez que querés agregar un producto, como un nuevo televisor o un nuevo celular, tendrías que crear una nueva página específicamente para ese producto, lo que puede llevar mucho tiempo además del costo que puede llevar este proceso.

Ejemplos de páginas web estáticas

Un ejemplo sencillo de cómo es una página web estática, es el siguiente:

```
<head>
<title>Ejemplo página web estática</title>
</head>
<body>
La fecha de hoy es Enero 1, 2022
</body>
</html>
```

Aquí, la fecha está escrita directamente en el código de la página (*estática*) y cada vez que se recargue la página, dirá lo mismo, **Enero 1, 2022**… la única forma de que cambie es si alguien actualiza el

código y escribe otra fecha o aplica alguna instrucción que la haga dinámica para que la fecha sea diferente cada vez que carga.

Qué es una página web dinámica

La palabra dinámica se refiere a elementos que cambian continuamente, son interactivos y funcionales, en lugar de ser simplemente informativos. Por supuesto, eso requiere utilizar más que solo código HTML y CSS.

En comparación con las páginas web estáticas, que son mayoritariamente informativas, una página web dinámica incluye aspectos que se caracterizan por la interactividad y la funcionalidad, por ejemplo, los usuarios pueden interactuar con la información que se presenta en la página gracias a las instrucciones creadas a través de los lenguajes de programación y la base de datos sobre la que está construida.

Los sitios web dinámicos basan su comportamiento y funcionalidad en dos tipos de programación, **front-end** (del lado del cliente) y **back-end** (*del lado del servidor*). Las instrucciones del lado del cliente es código JavaScript que se ejecuta en el navegador. Mientras que las instrucciones que se ejecutan del lado del servidor son instrucciones escritas en lenguajes de scripting o programación, como ASP.Net, PHP, Python, etc. y que son ejecutadas para crear lo que el usuario ha solicitado en su interacción con la página. En nuestro caso utilizaremos Node.Js como lenguaje de back-end.

Una vez ejecutadas las instrucciones en el servidor, un nuevo HTTP response se envía al navegador del usuario para mostrarle lo que ha solicitado. El resultado final es el mismo que en un sitio web estático: una página HTML que el usuario ve desde el navegador.

Por resumir, una página web dinámica puede ser más compleja cuando hablamos de su diseño y desarrollo, pero también es más versátil cuando se trata de la funcionalidad que ofrece.

Ventajas de una página web dinámica

Entre las ventajas de una página web dinámica están:

- **Puede gestionar información a través de bases de datos**.

Esto permite que el usuario pueda solicitar información fácilmente de una manera organizada y estructurada dentro de un catálogo, además de crear y mostrar contenido según el tipo de usuario que acceda a la página.

- **El contenido se puede gestionar a través de un CMS**.

El contenido almacenado en el CMS puede incluir una variedad de archivos, desde el texto hasta las imágenes que se muestran, diseños de página, configuraciones del sitio y más. Esto permite una flexibilidad extrema a la hora de crear el sitio y también permite que varios usuarios puedan manipular el contenido según sea necesario.

- **El coste de mantenimiento es menor**.

Si la página no necesita cambios en el diseño básico o en la funcionalidad definida al inicio de su desarrollo. Ya que se puede gestionar la información a través de un CMS (por ej: WordPress) , existe poco o nada de costes cuando se trata de su mantenimiento.

Desventajas de una página web dinámica

Algunas desventajas de una página web dinámica son:

- **Pueden existir limitaciones en el diseño.**

Ya que el contenido está principalmente basado en la información contenida en la base de datos y la presentación al usuario se basa en la estructura de la misma. Esto puede hacer que el diseño sea complicado, ya que lo más sencillo es optar por un enfoque único para todas las páginas. Dependiendo del CMS, puede resultar difícil crear varios diseños o plantillas que permitan mostrar diferentes tipos de contenido de diferentes formas.

- **Puede involucrar altos costos de construcción iniciales.**

Al coste del desarrollo de la página web se le suma el coste del desarrollo de las bases de datos donde se guardará el contenido a mostrar, etc. El desarrollo también puede costar más a medida que se agregan nuevas funcionalidades. Si bien los costos de mantenimiento pueden ser más bajos como fue mencionado en las ventajas, también puede involucrar costos de desarrollo iniciales mucho más altos que al desarrollar una página web estática.

Ejemplos de páginas web dinámicas

Como ya hemos visto, es muy sencillo determinar si una página web es dinámica: por ejemplo, cuando puedes interactuar con ella, o si cada vez que la recargas, puedes ver contenido distinto.
Por lo tanto, la mayoría de las páginas que regularmente visitas es probable que sean dinámicas porque son interactivas. Por ejemplo, una página web dinámica te permite crear un perfil de usuario **Facebook.com**, comentar una publicación **LinkedIn.com**, o hacer una reserva .
Siguiendo el ejemplo de la página que muestra una fecha, si queremos convertirla en una página web dinámica, podemos cambiar la fecha escrita textualmente por una función que retorne la fecha actual, de esta forma:

```
<head>
<title>Página web dinámica</title>
</head>
<body>
La fecha de hoy es <%=Datetime.Now()%>
</body>
</html>
```

Aquí, cada vez que se recarga la página, se mostrará la fecha y hora actual, es decir será diferente en cada recarga de la página, ya que la instrucción **<%=Datetime.Now()%>** le indica al servidor que retorne la fecha del momento en que recibe la petición.

Página web dinámica vs estática: Conclusión

En conclusión, si tenés que crear una página web y vas a tomar la decisión entre crear una página web estática o una dinámica, tu decisión debe ser principalmente en los objetivos que querés cumplir con tu página web y los recursos de tiempo y conocimientos que tengas disponibles.

La mayoría de las personas que no poseen conocimientos técnicos de diseño y desarrollo de páginas web, prefieren los sitios web dinámicos porque a través de plataformas CMS como WordPress, Joomla, Drupal o Ghost pueden crear sitios web dinámicos de una forma muy fácil y rápida, a la vez que son más fáciles de mantener a largo plazo.

Si bien es cierto que las páginas web dinámicas ofrecen más posibilidades, pueden ser mucho más complejas de construir y mantener para los usuarios que no tengan conocimientos técnicos y deseen incorporar integraciones que no ofrezcan los CMS; mientras que las páginas web estáticas son algo más limitadas, pero en principio son mucho más simples de crear y mantener si tenés conocimientos en HTML y CSS.

Introducción

La creciente popularidad de JavaScript ha traído consigo varios cambios, incluyendo la superficie del desarrollo web, ya que hoy en día es radicalmente diferente. Las cosas que podemos hacer en la web hoy, con JavaScript ejecutando en el servidor, como también en el navegador, eran difíciles de imaginar hace varios años, o se encapsulan dentro de entornos “sandbox” como Flash y Java.

Antes de indagar en Node.js, tenés que leer acerca de los beneficios de utilizar JavaScript a través del stack, que unifica el idioma y el formato de datos (JSON), lo que permite reutilizar de manera óptima los recursos del desarrollador. Como esto es más un beneficio de JavaScript que de Node.js específicamente, no hablaremos mucho de ello aquí. Sin embargo, es una ventaja clave para la incorporación de Node en su pila.
La web de Node indica: “Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.” Más allá de eso, vale la pena señalar que el creador de Node.js, Ryan Dahl fue encargado a crear sitios web en tiempo real con función de inserción, “inspirado por aplicaciones como Gmail”. En Node.js, dió a los desarrolladores una herramienta para trabajar en el paradigma no-bloqueante, event-driven I/O.

En resumen: Node.js se destaca en aplicaciones web de tiempo real empleando la tecnología push a través de Websockets. ¿Qué es tan revolucionario acerca de eso? Bueno, después de más de 20 años de webs basadas en el paradigma de petición-respuesta, finalmente tenemos aplicaciones web en tiempo real, las conexiones bidireccionales, donde tanto el cliente como el servidor pueden iniciar la comunicación, lo que les permite intercambiar datos libremente. Esto está en contraste con el paradigma de respuesta web típica, donde el cliente siempre inicia la comunicación. Además, todo se basa en el Open Web Stack (HTML, CSS y JS) que se ejecuta en el puerto estándar 80.

Podríamos argumentar que hemos tenido este formato durante años en forma de Flash y Applets de Java, pero en realidad, eran simplemente un entorno de Sandbox usando la web como un protocolo de transporte para ser entregado al cliente. Además, se ejecutan en aislamiento y a menudo operan a través de un puerto no estándar, el cual podía tener requisitos adicionales para su uso.

Con todas sus ventajas, Node.js ahora juega un papel crítico en la pila de tecnología de muchas empresas de alto perfil que dependen de sus exclusivas ventajas.

Vamos a analizar no sólo cómo estas ventajas son obtenidas, sino también por qué es posible que desees utilizar Node.js y por qué no usar algunos de los clásicos modelos de aplicaciones web como ejemplos.

¿Cómo funciona?

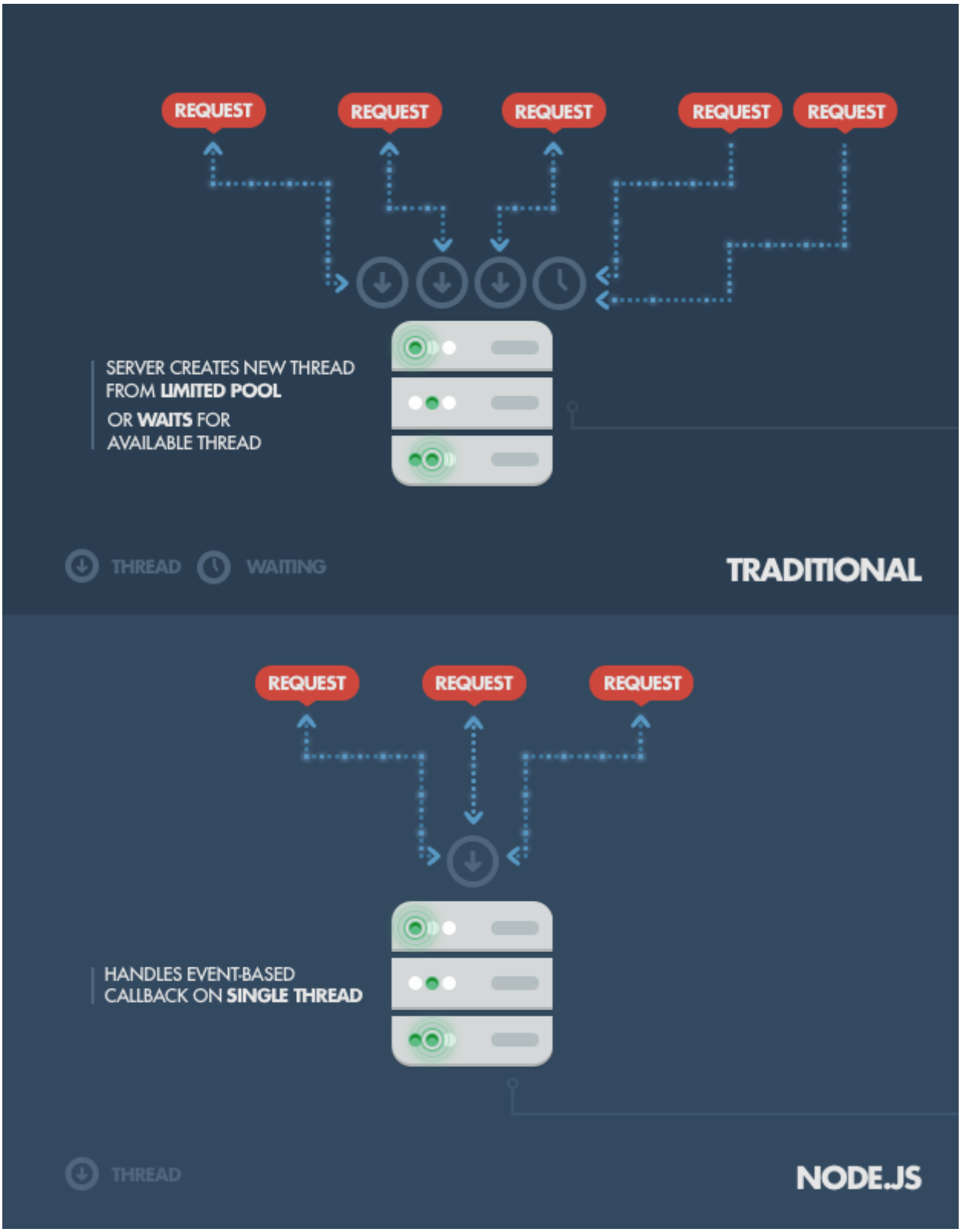
La idea principal de Node.js: uso no-bloqueante, event-driven I/O, permanecer ligero y eficiente en la superficie del uso intensivo de datos en tiempo real de las aplicaciones que se ejecutan en dispositivos distribuidos.

Y este entendimiento es absolutamente esencial. Definitivamente no tenés que usar Node.js para operaciones intensivas de CPU; de hecho, utilizándolo para el cálculo pesado anulará casi todas sus ventajas. Donde Node REALMENTE destaca es en la construcción rápida y escalable de aplicaciones de red, debido a que es capaz de manejar un gran número de conexiones simultáneas con alto rendimiento, lo que equivale a una alta escalabilidad.

Cómo funciona internamente es bastante interesante. Frente a las tradicionales técnicas de servicio web donde cada conexión (solicitud) genera un nuevo subproceso, retomando la RAM del sistema y finalmente a tope a la cantidad de RAM disponible, Node.js opera en un solo subproceso, no utiliza el bloqueo de

llamadas de E/S, lo que le permite admitir decenas de miles de conexiones simultáneas (celebrada en el caso de loop).

Un cálculo rápido: suponiendo que cada subproceso tiene un potencial acompañado de 2 MB de memoria, el cual se ejecutará dentro de un sistema con 8 GB de RAM nos pone a un máximo teórico de 4.000 conexiones simultáneas, además del costo de cambio de contexto entre subprocesos. Ese es el escenario que se suelen tratar con técnicas de servicio web tradicional. Evitando todo eso, Node.js alcanza niveles de escalabilidad de más de 1M de conexiones simultáneas.



Existe por supuesto, la posibilidad de compartir un único subproceso entre todas las solicitudes de clientes, convirtiéndola en una falla potencial de escribir aplicaciones Node.js. En primer lugar, el cómputo pesado podría estancarse y provocar problemas para todos los clientes (más sobre esto más adelante) como las peticiones entrantes, las cuales serían bloqueadas hasta que dicho cálculo se haya completado. En segundo

lugar, los desarrolladores necesitan ser muy cuidadosos en no permitir una excepción burbujeante hacia el núcleo (la superior), lo que provocaría que la instancia de Node.js se terminase.
La técnica utilizada para evitar excepciones transfiere los errores a la llamada como llamada de parámetros (en lugar de tirar de ellos, al igual que en otros entornos).

NPM: El Node Package Manager

Cuando hablamos de Node.js, una cosa que definitivamente no debe omitirse es integrarlo en el apoyo de la gestión de paquetes utilizando la herramienta NPM que viene por defecto con cada instalación de Node.js. La idea de los módulos NPM es muy similar a la de Ruby Gemas: un conjunto de componentes reutilizables disponibles públicamente a través de una fácil instalación a través de un repositorio en línea, con la versión y la dependencia de gestión.

Una lista completa de los paquetes de módulos puede encontrarse en el sitio web de NPM Https://npmjs.org/ o acceder utilizando la herramienta de la CLI de NPM que automáticamente se instala con Node.js. El módulo es un ecosistema abierto a todos, y cualquiera puede publicar su propio módulo que será incluido en el repositorio de NPM. Una breve introducción a la NPM (un poco viejo, pero sigue siendo válido) se puede encontrar en http://howtonode.org/introduction-to-npm.
Algunos de los más populares hoy en día son módulos de NPM:

- express - Express.js, inspirado en el framework de desarrollo web para Node.js, y el estándar de facto para la mayoría de aplicaciones Node.js de hoy en día.
- connect - Connect es un servidor HTTP extensible framework para Node.js, que proporciona una colección de alto rendimiento de plugins conocidos como middleware; sirve como fundamento para expresar.
- socket.io y sockjs - Componente del servidor de los dos componentes de websockets más comúnes en la actualidad.
- Jade - Uno de los más populares motores de plantillas, inspirados por HAML, un defecto en Express.js.

- mongo y mongojs - mongoDB wrappers para proporcionar la API para bases de datos de objetos MongoDB en Node.js.
- redis - Redis biblioteca cliente.
- coffee-script - CoffeeScript compilador que permite a los desarrolladores escribir sus programas Node.js con café.
- Underscore (lodash, lazy) - La biblioteca de utilidades más popular de JavaScript, empaquetados para ser utilizado con Node.js, así como sus dos contrapartes, que prometen mejorar el rendimiento mediante la adopción de un enfoque de aplicación ligeramente diferente.
- forever - Probablemente la utilidad más común para asegurar que un determinado Node script se ejecuta continuamente. Mantiene su proceso de Node.js en la producción y en el rostro de cualquier fallo inesperado.

La lista es interminable. Hay toneladas de paquetes realmente útiles y disponible para todos (sin ofender a los que he omitido aquí).

Ejemplos en donde Node.js debe utilizarse:

Chat

Es la forma más típica en tiempo real y una multi-aplicación de usuario. Desde IRC , a través de muchos propietarios y protocolos abiertos girando en puertos no estándar, con la capacidad de instrumentar todo en Node.js con websockets corriendo sobre el puerto estándar 80.

La aplicación de chat es realmente perfecta para Node.js: es ligera, tiene un alto tráfico de datos intensivos (pero baja/procesamiento de cómputo) y es una aplicación que funciona en dispositivos distribuidos. También es un gran caso de uso para el aprendizaje, ya que es demasiado simple, pero al mismo tiempo que cubre la mayoría de herramientas que podés utilizar en una típica aplicación Node.js.

Vamos a tratar de describir cómo funciona.

En el ejemplo más sencillo, tenemos una sola sala de chat en nuestro sitio web donde la gente puede venir e intercambiar mensajes ya sea con una persona o con varias. Por ejemplo, supongamos que tenemos tres personas en el sitio todos los conectados a nuestro tablero de mensajes.

En el lado del servidor, tenemos un simple Express.js que implementa dos cosas: 1) Obtener un controlador de solicitudes '/' que sirve la página web que contiene un tablero de mensajes y un botón 'Enviar' para inicializar el nuevo mensaje de entrada, y 2) un servidor websockets que escucha los mensajes emitidos por los clientes de websocket.

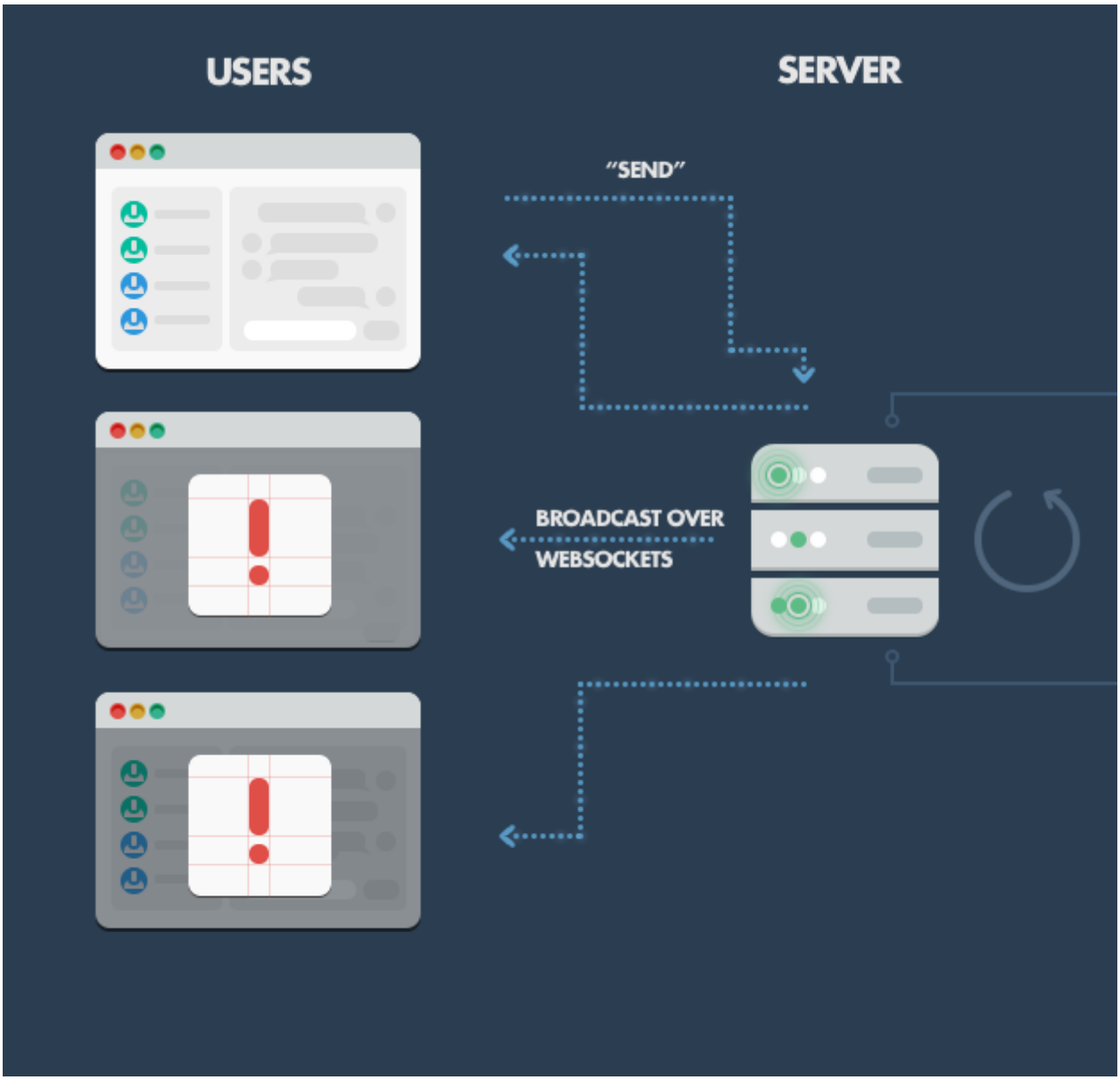
En el cliente, tenemos una página HTML con un par de controladores, uno para el Send' evento de clic de botón, que recoge el mensaje de entrada y lo envía hacia abajo el websocket, y otro que escucha los mensajes entrantes del nuevo cliente de websockets (es decir, los mensajes enviados por otros usuarios, que el servidor ahora quiere que el cliente muestre).

Cuando uno de los clientes envía un mensaje, lo que sucede es lo siguiente:

1. El explorador atrapa el clic con el botón 'Send' a través de un controlador de JavaScript que recoge el valor del campo de entrada (es decir, el texto del mensaje), y emite un mensaje al websocket utilizando el cliente conectado a nuestro servidor (inicializado con la página web).

2. El componente del servidor de la conexión websocket recibe el mensaje y lo reenvía a todos los demás clientes conectados mediante el método de difusión.

3. Todos los clientes reciben el mensaje como un mensaje de inserción a través de un componente de cliente websockets que se ejecuta dentro de la página web. Ellos entonces recogen el contenido del mensaje y actualizan la página web en lugar de anexar el nuevo mensaje a la junta.



Este es el ejemplo más sencillo. Para una solución más robusta, podrías utilizar un caché simple basado en la Redis store. O incluso en una solución más avanzada, una cola de mensajes para gestionar el enrutamiento de mensajes a los clientes y un mecanismo de entrega más robusto que pueda cubrir pérdidas de conexión temporal o almacenar mensajes para clientes registrados mientras está desconectado. Pero independientemente de las mejoras que realices, Node.js todavía operará bajo los mismos principios básicos: reaccionar a eventos, manejo de muchas conexiones simultáneas, y mantenimiento en la fluidez de la experiencia del usuario.

API en la parte superior de un OBJETO DB

Aunque Node.js realmente destaca entre aplicaciones de tiempo real, es una adaptación natural para exponer los datos de objeto DBs (p. ej. MongoDB). El almacenamiento de datos JSON permite que Node.js funcione sin la desigualdad de impedancia y la conversión de datos.

Por ejemplo, si estás utilizando Rails, tendrías que convertir los datos de JSON para modelos binarios y después exponer nuevamente como JSON sobre HTTP cuando el dato es consumido por el backbone.js, angulares, etc., o incluso llamadas AJAX jQuery normal. Con Node.js, simplemente podés exponer tus objetos JSON con una API REST para que el cliente consuma. Además, no necesitas preocuparte por la conversión entre JSON y cualquier otra cosa al leer o escribir desde su base de datos (si estás usando MongoDB). En conclusión, podés evitar la necesidad de realizar varias conversiones mediante un formato de la serialización de datos uniformes a través del cliente, servidor y base de datos.

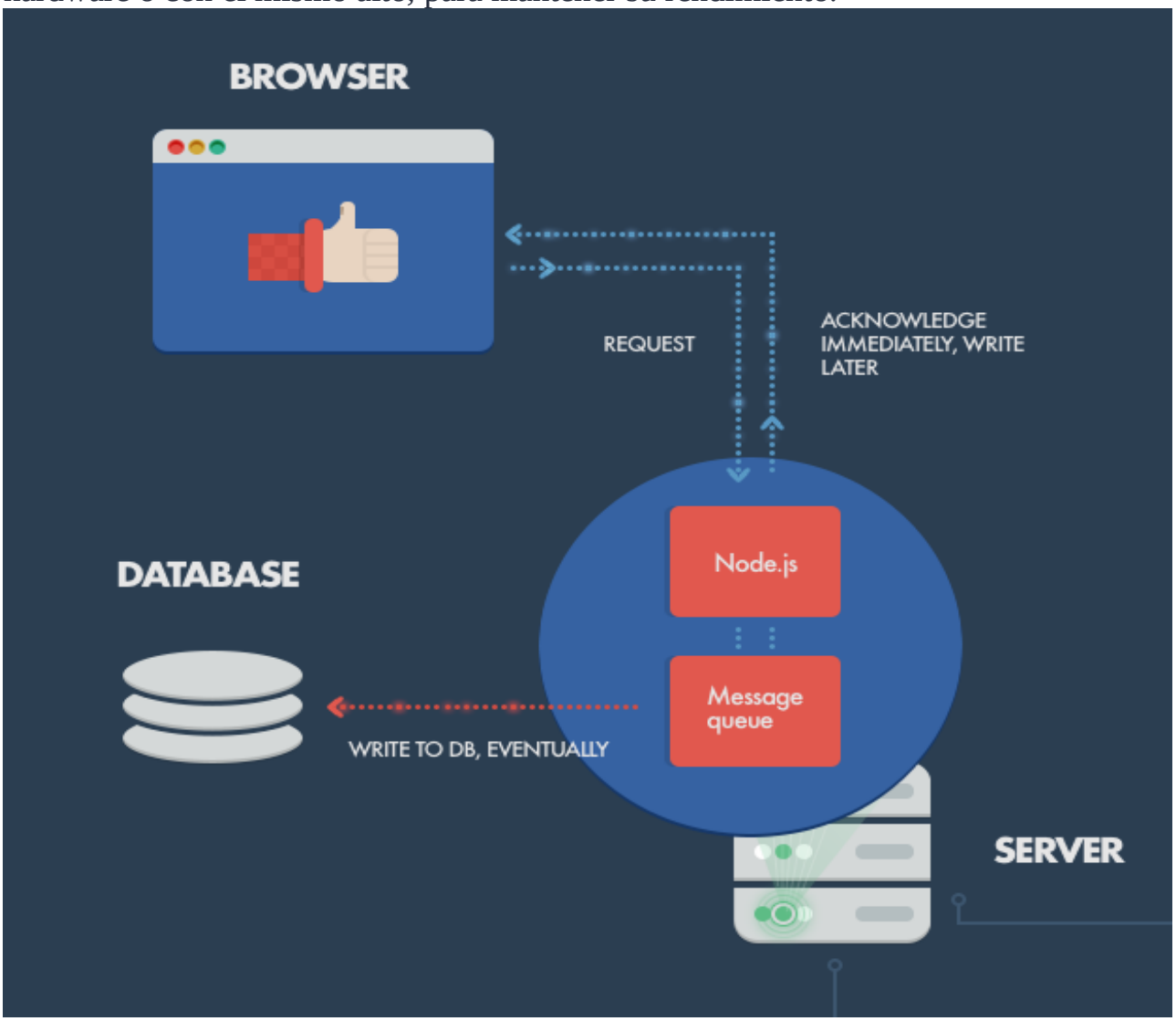
Entradas en espera

Si estás recibiendo una gran cantidad de datos concurrentes, tu base de datos puede ahogarse. Como se ha descrito más arriba, Node.js puede manejar fácilmente las conexiones simultáneas al mismo tiempo. Pero debido a que el acceso a la base de datos es una operación de bloqueo (en este caso), nos topamos con problemas. La solución es reconocer el comportamiento del cliente antes de que los datos se escriban en la verdadera base de datos.

Con ese enfoque, el sistema mantiene su sensibilidad bajo una carga pesada, lo que es particularmente útil cuando el cliente no necesita una firme confirmación de la correcta escritura de datos. Ejemplos típicos incluyen: el registro o la escritura de datos de seguimiento de usuario, procesamiento en lotes que no se utilizan hasta un momento posterior, así como las operaciones que no necesitan ser reflejadas al

instante (como actualizar el recuento de Likes en Facebook) donde la coherencia final (tan a menudo utilizadas en el mundo NoSQL) es aceptable.

Los datos se ponen en cola a través de algún tipo de caché o de Message Queue Server (por ejemplo, infraestructura, RabbitMQ, ZeroMQ) y resumido por un proceso separado escrito en lote, cálculo o procesamiento intensivo servicios backend, escrito en un mejor desempeño de plataforma para tales tareas. Un comportamiento similar puede implementarse con otros lenguajes/frameworks, pero no con el mismo hardware o con el mismo alto, para mantener su rendimiento.



En resumen: con Node, podés empujar la base de datos escrita a un lado y tratar con ella más tarde, para proceder como que si esta hubiera sido exitosa.

Transmisión de Datos

En plataformas web más tradicional, las peticiones y respuestas HTTP son tratadas como eventos aislados; de hecho, son realmente corrientes. Esta observación puede ser utilizada en Node.js para construir algunas características interesantes. Por ejemplo, es posible procesar archivos mientras están siendo cargados, ya que los datos entran a través de un arroyo, y pueden ser procesados en una línea de moda. Esto podría hacerse en tiempo real para la codificación de audio o vídeo, como proxy entre diferentes fuentes de datos (véase la sección siguiente).

PROXY

Node PROXY.js es empleado como un servidor proxy el cual puede manejar una gran cantidad de conexiones simultáneas en un modo de no-bloqueo. Es especialmente útil para proxy de diferentes servicios con distintos tiempos de respuesta, o para la recopilación de datos desde varios puntos de origen.

Un ejemplo: considere una aplicación de servidor que se comunica con recursos de terceros, extrayendo datos de diferentes fuentes, o almacenando los activos como imágenes y vídeos a servicios terceros de Cloud.

Aunque existen servidores de proxy dedicados, utilizando en su lugar Node podría ser útil si su infraestructura de servidores proxy es inexistente o si necesita una solución para el desarrollo local. Con esto, quiero decir que se podría construir una aplicación del lado del cliente con un servidor de desarrollo Node.js para activos como proxy/stubbing solicitudes de API, mientras que en la producción manejarías tales interacciones con un dedicado servicio de proxy (nginx, HAProxy, etc.).

Brokerage-Dashboard del Stock Trader

Volvamos al nivel de aplicación. Otro ejemplo donde domina el software de escritorio, sin embargo podría ser fácilmente reemplazado con una web en tiempo real es la solución comercial de los agentes de software; se utiliza para realizar el seguimiento de los precios de las existencias, realizar cálculos y análisis técnico y crear los gráficos y diagramas.

Cambiar a tiempo real es una solución basada en la web que permitiría a los corredores cambiar fácilmente de estaciones de trabajo o lugares de trabajo. Pronto podríamos comenzar a verlos en la playa de Florida, Ibiza…o Bali.

Panel de Supervisión de Aplicaciones

Otro caso de uso común en qué el Node-con-web-sockets encaja perfectamente es el siguiente: el seguimiento de los visitantes del sitio web y la visualización de sus interacciones a tiempo real. (Si estás interesado, esta idea ya se produjo por Colibrí).

Podrías recopilar estadísticas a tiempo real desde tu usuario, o inclusive subir al siguiente nivel mediante la introducción de interacciones selectivas con tus visitantes abriendo un canal de comunicación cuando llegan a un punto específico en el embudo. (Si estás interesado, esta idea ya se produjo por CANDDi).

Imagina cómo podría mejorar tu negocio si supieras lo que estuvieran haciendo tus visitantes en tiempo real; si pudieras visualizar sus interacciones. Con el tiempo real, ahora podés tomar dos vías de Node.js.

Ahora el panel de monitorización del sistema, vamos a conocer la perspectiva de la infraestructura de las cosas. Imagínate, por ejemplo, un proveedor de SaaS que le quiere ofrecer a sus usuarios un servicio de supervisión (por ejemplo, la página de GitHub). Con el evento Node.js-loop, podemos crear un poderoso tablero basado en

la web que comprueba los servicios de los estados de manera asíncrona y envía datos a los clientes usando Websockets.

Tanto internos (intra-empresa) como también los de los servicios públicos de los Estados, pueden ser reportados en vivo y a tiempo real utilizando esta tecnología. Empuja esta idea un poco más lejos y trata de imaginar un centro de operaciones de red (NOC) en aplicaciones de supervisión de un operador de telecomunicaciones, cloud/red/proveedor de servicios de hosting, o alguna institución financiera, todos se ejecutan en el open web stack respaldado por Node.js y Websockets en lugar de Java y/o applets de Java.

Nota: No intentes construir sistemas a tiempo real duros en Node (es decir, sistemas que requieran tiempos de respuesta coherentes). Erlang es probablemente una mejor elección para esta clase de aplicación.

Donde node.js se puede utilizar

Aplicaciones Web del lado del Servidor

Node.js con Express.js también pueden ser utilizados para crear aplicaciones web clásicas en el servidor. Sin embargo, mientras sea posible, este paradigma en petición-respuesta de Node.js sería llevar alrededor de HTML, no es el más típico de los casos de uso. Hay argumentos para estar a favor y en contra de este enfoque. Aquí están algunos hechos a considerar:

PROS:

- Si tu aplicación no tiene ningún cálculo intensivo del CPU, podés construir en Javascript de arriba a abajo, inclusive a nivel de base de datos si utilizas el objeto de almacenamiento JSON como MongoDB DB. Esto facilita el desarrollo (incluyendo la contratación) significativamente.

- Los Crawlers reciben una respuesta totalmente HTML, que es mucho más SEO-friendly, digamos, una sola página o en una aplicación de Websockets app se ejecuta sobre Node.js.

CONS:

- Un CPU de cálculo intensivo bloqueará la receptividad del Node.js, por lo que una plataforma de roscado es un mejor enfoque. Alternativamente, podrías intentar escalar el cómputo [*].

- Utilizando Node.js con una base de datos relacional es aún bastante doloroso (leer más abajo para ver más detalles). Hazte un favor y escoge cualquier otro entorno como Rails, Django, o ASP.NET MVC si estás intentando realizar operaciones relacionales.

Donde Node.js no debe usarse

En el lado del Servidor de Aplicaciones Web con una DB Relaciónal detrás

Comparando Node.js con Express.js en contra de Ruby on Rails, por ejemplo, hay una decisión clara en favor de esta última cuando se trata de acceso a datos relacionales. El DB relacional con herramientas para Node.js está aún en sus primeras etapas; es bastante prematuro y por ende no tan agradable trabajar con ello. Por otro lado, Rails automáticamente proporciona datos de configuración del acceso a la derecha de la caja junto con el esquema de base de datos y herramientas de soporte de migraciones de otras Gemas (con doble sentido). Rails y su homólogo marco han madurado y probado que Active Record Data Mapper recopila implementaciones del acceso a datos y que echarás de menos si intentas replicarlo con JavaScript puro.[*]

Aún, si estás muy inclinado a permanecer en JS todo el camino, mantén un ojo sobre Sequelize ORM y Nodo2 ya que ambos son todavía inmaduros, pero eventualmente pueden alcanzar a los demás lenguajes de programación.

Cuando se trata de cómputo pesado, Node.js no es la mejor plataforma. Definitivamente no querés construir un servidor de cálculo Fibonacci en Node.js. En general, cualquier operación de uso intensivo de CPU anula todas las ventajas de rendimiento y bloquearía cualquier petición entrante de un subproceso.

Como se dijo anteriormente, Node.js es Single-threaded y utiliza un único núcleo del CPU. Cuando se trata de la adición de la concurrencia en un servidor multi-core, hay algunos trabajos realizados por el Node básico en la forma de un módulo cluster [ref: http://nodejs.org/api/cluster.html]. También podés ejecutar varias instancias del servidor Node.js bastante fácil detrás de un proxy inverso a través de nginx.
Con la agrupación, debes descargar todo el cómputo pesado para procesar un fondo escrito dentro de un entorno más apropiado, y que ellos se comuniquen a través de Message Queue Server como RabbitMQ.

Aunque tu procesamiento en segundo plano puede ejecutarse en el mismo servidor inicialmente, este enfoque tiene el potencial de una muy alta escalabilidad. Los servicios de procesamiento de fondo podrían ser fácilmente distribuidos al trabajador independientemente de de servidores sin la necesidad de configurar las cargas de los distintos servidores web.

Por supuesto utilizarías el mismo enfoque en otras plataformas también, pero con Node.js podés conseguir un alto reqs/s del que hemos hablado, ya que cada petición es una tarea pequeña y un manejo muy rápido y eficientemente.

Conclusión

Hemos hablado de Node.js desde una teoría práctica, comenzando con sus objetivos y ambiciones, y terminando con algunos de los escollos que tiene programar en ese entorno. Cuando las personas tienen problemas con Node, casi siempre deducen al hecho de que el bloqueo de operaciones son la raíz de todo mal. El 99% del abuso de procesos en Node viene como consecuencia directa.

Para recordar: Node.js nunca fue creado para resolver el problema de escalado de computación. Fue creado para resolver el problema de escalado de E/S, lo que lo hace muy bien.

¿Por qué usar Node.js? Si el caso de uso no contiene operaciones intensivas del CPU ni el acceso a los recursos de bloqueo, podés aprovechar los beneficios de Node.js y disfrutar de aplicaciones de red rápidas y escalables. Bienvenido a la web en tiempo real.

¿Porque usar Node.Js?

Inicialmente uno de los puntos que se deben tener en claro es que Node.JS por definición es un **entorno de ejecución para JavaScript.** Así mismo, sus características son aquellas que hacen que sea tan interesante a la hora de utilizarlo ya bien sea para un desarrollo batch, un servicio web, una API Rest o cualquier herramienta a nivel de batch.

Anteriormente, los desarrolladores de JavaScript sólo podían utilizar este lenguaje con la obligación de utilizar un navegador web ya sea Firefox, Chrome, entre otros. Lo que ocasionaba que se tuviera una limitación a la hora de realizar cierto tipo de aplicaciones, ya que no se podían generar o programar aplicaciones que se renderizaran en el servidor.

Con la llegada de **Node.JS**, se abrió un nuevo mundo y empezaron a surgir los servidores web hechos con **Express** o con otras librerías basadas en Node, las de API Rest, incluso se abrió un nuevo mundo a la hora de desarrollar para **IOT**. Por ejemplo: las placas arduino, ya que éstas se pueden desarrollar con Node en una aplicación y utilizarlas en ese tipo de placas. Con lo cual, se puede decir que Node tiene una cantidad considerable de características entre las cuales destacan las siguientes:

- **Desarrollo en JavaScript:** Para desarrollar en Node se realiza a través del lenguaje de programación JavaScript, que actualmente está teniendo popularidad y mejoras permitiendo el desarrollo tanto para frontend como para backend, abriendo el camino a los profesionales fullstack.

- **Basado en el motor V8 de Chrome:** Es uno de los motores más avanzados a nivel de JavaScript ya que se mantiene actualizado con las nuevas funcionalidades del estándar **ECMAScript** 6, 7 y 8. También existe una versión de Node.JS que utiliza el motor de JavaScript Chakra propio de Microsoft, aunque en su mayoría las versiones y los proyectos de Node se encuentran basadas en V8.

- **Operaciones de E/S sin bloqueos:** Node está pensado para que las operaciones de entrada y salida sean sin bloqueos, por ejemplo: un servidor web realiza una petición única y espera una respuesta.

- **Orientado a eventos (POE):** Para comprender esta característica, pensemos en un bus de datos cuando un trozo de código realiza una operación, pública un evento ese evento en otra instancia (en otro momento del tiempo) lo recibe otro trozo de código y hace otra acción con él; en este punto de hecho, se habla mucho del término **asincronía** del tema de ajax. Por ejemplo, a la hora de hacer peticiones a un servicio web externo una API Rest la sincronía es una consecuencia de la orientación a eventos, Node.JS funciona perfectamente con temas de

asincronía y es una muy buena opción si queremos hablar de códigos asíncronos que queramos hacer para nuestra aplicación.

- **Liviano y Eficiente:** En resumen por todo lo anteriormente mencionado (entradas y salidas sin bloqueo, desarrollado bajo JavaScript, basado en V8, orientado a eventos y el tema de la asincronía) hace que Node.JS sea liviano (pese muy poco) y a su vez sea muy eficiente en los casos de gestión de eventos, orientación a eventos y los casos de entrada y salida.

Node.JS es una buena opción para desarrollar cierto tipo de cosas como lo son:

- **Servidores Web:** Con el uso de librerías que se encuentran en los paquetes propios de Node.JS o de terceros como Express, Koa y Hapi.
- **Sockets:** Son eventos que para realizar chats y aplicaciones en tiempo real es una excelente opción, sobretodo gracias a su gran velocidad.
- **IOT:** Programar placas pequeñas con poco hardware como un Arduino, permite desarrollar una aplicación y desplegarla.

Instalación de NodeJS en Windows

Primero tenés que obtener el instalador desde https://www.nodejs.org/es/

Si estás en Windows, al pulsar sobre el botón verde se descargará el instalador para este sistema, un archivo con extensión "msi" que como ya sabes, te mostrará el típico asistente de instalación de software.

Una vez descargado, ejecutas el instalador y ¡ya lo tienes!

A partir de ahora, para ejecutar "Node" tienes que irte a la línea de comandos de Windows e introducir el comando "node".

Nota: Acceder a la línea de comandos de Windows desde el menú de inicio, buscas "cmd" y encuentras el cmd.exe, que abre la línea de comandos. En algunos sistemas Windows anteriores a 7 accedes también desde el menú de inicio, utilizas la opción "Ejecutar" del menú de inicio y escribes "cmd".

Entonces entrarás en la línea de comandos del propio NodeJS donde podés ya escribir tus comandos Node, que luego veremos.

Instalar NodeJS en Linux

En la página de instalación de NodeJS te ofrecen los comandos para instalar Node en Linux. Son un par de comandos sencillos, pero depende de tu distro, así que es recomendable que te documentes allí. En mi caso quería instalar NodeJS en Ubuntu 20.04. Para ello he usado los siguientes comandos, a ejecutar uno después del otro.

```
curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Con el primer comando obtienes el instalador y con el segundo comando haces la instalación propiamente dicha. Luego podés verificar la instalación con el comando "node -v" y te debería salir la versión que se ha instalado en tu máquina.

También si lo deseas, desde la página de descargas de Node accederás a los binarios de Linux o al código fuente para complilarlo.

Instalar NodeJS en Mac

Para instalar NodeJS sobre Mac es tan sencillo como acceder a la página de Node y descargar el instalador. Se instala como cualquier otra aplicación, tanto el propio NodeJS como npm.

Alternativa con Homebrew: La instalación en Mac es muy sencilla si cuentas con el gestor de paquetes "homebrew". Es tan fácil como lanzar el comando:

```
brew install nodejs
```

Durante la instalación es posible que te solicite incluir en tu sistema un paquete de utilidades por línea de comandos de xcode, si es que no lo tienes ya instalado en tu OS X. Si se produce un error durante la instalación prueba a hacer un update de homebrew.

`brew update`

Probando los primeros comandos NodeJS

En NodeJS la consola de Node podés escribir instrucciones Javascript. Si lo deseas, podés mandar mensajes a la consola con console.log() por lo que ésta podría ser una bonita instrucción para comenzar con node:

```
$ node

console.log("hola mundo");
```

Te mostrará el mensaje "hola mundo" en la consola.

¿Cómo empiezo con Node.js después de instalarlo?

Una vez que hayamos instalado Node.js, construyamos nuestro primer servidor web. Creá un archivo llamado `app.js` que contenga el siguiente contenido:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Ahora, ejecutalo en tu servidor web usando `node app.js`. Visitá `http://localhost:3000` en tu navegador web y verás un mensaje que dice "Hola mundo".

Bloqueo frente a no bloqueo

Bloqueo

El bloqueo es cuando la ejecución de JavaScript adicional en el proceso de Node.js debe esperar hasta que se complete una operación que no es de JavaScript. Esto sucede porque el bucle de eventos no puede seguir ejecutando JavaScript mientras se produce una operación de **bloqueo** .

En Node.js, JavaScript que presenta un rendimiento deficiente debido a que consume mucha CPU en lugar de esperar una operación que no es JavaScript, como E / S, no se suele denominar **bloqueo** . Los métodos síncronos en la biblioteca estándar de Node.js que usan libuv son las operaciones de **bloqueo** más comúnmente utilizadas . Los módulos nativos también pueden tener métodos de **bloqueo** .

Todos los métodos de E / S en la biblioteca estándar de Node.js proporcionan versiones asincrónicas, que **no** son **bloqueantes** y aceptan funciones de devolución de llamada. Algunos métodos también tienen contrapartes de **bloqueo** , que tienen nombres que terminan en `Sync`.

Comparación de código

Los métodos de **bloqueo se** ejecutan de forma **sincrónica** y **los** métodos **sin bloqueo se** ejecutan de **forma asincrónica** .

Usando el módulo Sistema de archivos como ejemplo, este es un archivo **síncrono** leído:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

Y aquí hay un ejemplo **asincrónico** equivalente :

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
    if (err) throw err;
});
```

El primer ejemplo parece más simple que el segundo, pero tiene la desventaja de que la segunda línea **bloquea** la ejecución de cualquier JavaScript adicional hasta que se lea todo el archivo. Tenga en cuenta que en la versión síncrona, si se produce un error, será necesario detectarlo o el proceso se bloqueará. En la

versión asincrónica, depende del autor decidir si se debe producir un error como se muestra.

Ampliemos un poco nuestro ejemplo:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

Y aquí hay un ejemplo asincrónico similar, pero no equivalente:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
    if (err) throw err;
    console.log(data);
});
moreWork(); // will run before console.log
```

En el primer ejemplo anterior, `console.log` se llamará antes que `moreWork()`. En el segundo ejemplo `fs.readFile()` es **sin bloqueo,** por lo que la ejecución de JavaScript puede continuar y `moreWork()` se llamará primero. La capacidad de ejecutarse `moreWork()` sin esperar a que se complete la lectura del archivo es una opción de diseño clave que permite un mayor rendimiento.

Simultaneidad y rendimiento

La ejecución de JavaScript en Node.js es de un solo subproceso, por lo que la concurrencia se refiere a la capacidad del bucle de eventos para ejecutar funciones de devolución de llamada de JavaScript después de completar otro trabajo. Cualquier código que se espere que se ejecute de manera concurrente debe permitir que el bucle de eventos continúe ejecutándose mientras se están produciendo operaciones que no son de JavaScript, como E / S.

Como ejemplo, consideremos un caso en el que cada solicitud a un servidor web tarda 50 ms en completarse y 45 ms de esos 50 ms son E / S de base de datos que se pueden realizar de forma asincrónica. La elección de operaciones asincrónicas **sin bloqueo** libera esos 45 ms por solicitud para manejar otras solicitudes. Esta es una diferencia significativa en la capacidad con solo elegir usar métodos **sin bloqueo en** lugar de métodos de **bloqueo** .

El bucle de eventos es diferente a los modelos en muchos otros lenguajes donde se pueden crear hilos adicionales para manejar el trabajo concurrente.

Peligros de mezclar código de bloqueo y no bloqueo

Hay algunos patrones que deben evitarse cuando se trabaja con E / S.

Veamos un ejemplo:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
    if (err) throw err;
    console.log(data);
});
fs.unlinkSync('/file.md');
```

En el ejemplo anterior, `fs.unlinkSync()` es probable que se ejecute antes `fs.readFile()`, lo que se eliminaría `file.md` antes de que realmente se lea. Una mejor manera de escribir esto, que es completamente **sin bloqueo** y se garantiza que se ejecutará en el orden correcto es:

```
const fs = require('fs');
fs.readFile('/file.md', (readFileErr, data) => {
    if (readFileErr) throw readFileErr;
    console.log(data);
    fs.unlink('/file.md', (unlinkErr) => {
        if (unlinkErr) throw unlinkErr;
    });
});
```

Lo anterior coloca una llamada **sin bloqueo** `fs.unlink()` dentro de la devolución de llamada `fs.readFile()` que garantiza el orden correcto de las operaciones.

Comentarios en Node.Js

No tenemos que olvidarnos que Node es JavaScript, por ende, permite insertar comentarios en el código, al igual que la mayoría de los lenguajes de programación y puntualmente que JS. En concreto hay dos tipos de comentarios permitidos, los comentarios en línea que comienzan con una doble barra: //, y los comentarios multilínea, que comienzan con /* y terminan con */.

Importando y creando módulos

Un modulo es una librería o archivo JavaScript que puede ser importado dentro de otro código utilizando la función `require()` de Node. Por sí mismo, Express es un modulo, como lo son el middleware y las librerías de bases de datos que se utilizan en las aplicaciones Express.

El código mostrado abajo, muestra como puede importarse un modulo con base a su nombre, como ejemplo se utiliza el framework Express . Primero se invoca la función `require()`, indicando como parámetro el nombre del módulo o librería como una cadena (`'express'`), posteriormente se invoca el objeto obtenido para crear una aplicación Express.

Posteriormente, se puede acceder a las propiedades y funciones del objeto Aplicación.

```
var express = require('express');


var app = express();
```