UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de un gestor de contenido basado en Electron

Autor: Jaime Elso de Blas

Tutor: Roberto Latorre Camino

JULIO 2020

Dedicado a Fernán García de Zúñiga…

“The idea is not to live forever, it is to create something that will.”

Andy Warhol

Agradecimientos

Gracias a mi familia, sobre todo a mis padres, por la inversión económica realizada en mi educación que me ha llevado a adquirir los conocimientos que tengo hoy en día, por la paciencia, el apoyo y cariño depositado. Quisiera hacer una mención especial a mi tío, Fernando Mateus Fernández, quien siempre a sido una inspiración para mí y gracias a él, me interesé por la ciencia de la computación.

A todos mis amigos que tanto me han apoyado durante todo el camino, animado y motivado en los momentos difíciles. Sin todos y cada uno de ellos no habría sido posible.

A Fernán García de Zúñiga, quien además de ser un buen amigo, ha sido, es y será siempre mi mentor.

A todos los profesores de la escuela que durante estos años me han formado como profesional, pero en especial a mi tutor, Roberto Latorre Camino, que, aún estando muy ocupado, no dudó en aceptar ser mi tutor cuando acudí a su despacho y me ha guiado de la mejor manera en el desarrollo de este trabajo fin de grado.

A todos mis compañeros de la escuela con los que he pasado muchas horas de trabajo y estudio, pero también de diversión.

Abstract

When managing the content of a website, developers usually take advantage of tools that make this task easier, such as WordPress. It has been the preferred solution and market's dominant, for both business and particulars. This project presents an alternative solution to WordPress. Although it has a smaller-scale functionality, it proposes a different structure and offers a new method to display the content.

This content manager has been designed and developed from scratch, using Electron as the platform to create desktop apps, NodeJS as the JavaScript interpreter on the *Back-End* and, finally, HTML, CSS and JavaScript for the interface. With a modular structure, this application uses a common panel for different modules. For more flexibility, each of these modules can be modified independently, without affecting the rest.

The objective of this application is to generate static pages with no need of execution, obtaining better performance and faster content loading regarding other systems. In addition, these pages comply with W3C standards and the HTML content has been optimized, getting better results in search engines.

As a conclusion, I have compared the response time of the generated static pages and their equivalent in WordPress. Additionally, I have run tests with Lighthouse to obtain a global rating of the pages and be able to validate performance, accessibility, good practices and SEO results.

Resumen

A la hora de gestionar el contenido de una página web, los desarrolladores han optado por la utilización de herramientas que les faciliten la tarea, como solución, WordPress ha sido el dominante del mercado tanto para empresas como particulares. Este proyecto presenta una solución alternativa a menor escala, en lo que a funcionalidad se refiere, pero proponiendo un paradigma distinto en cuanto a generación y la manera de servir el contenido.

Este gestor de contenido ha sido diseñado y desarrollado desde cero, haciendo uso de Electron, como plataforma para crear aplicaciones de escritorio, NodeJS, como intérprete de JavaScript del lado del *Back-End*, y HTML, CSS, JavaScript para la parte de interfaz, en el *Front-End*. El desarrollo se ha realizado de forma modular empleando un panel común para diferentes módulos y cada uno de estos módulos pueden ser modificados sin afectar al resto.

El objetivo de esta aplicación es generar páginas estáticas sin necesidad de ejecución, obteniendo así un rendimiento y velocidad de carga superior al de gestores populares. Un segundo objetivo es cumplir los estándares de W3C y necesidades para la optimización en motores de búsqueda, usando los elementos de HTML que serán interpretados por los motores de búsqueda a la hora de indexar el contenido.

Para llegar a unas conclusiones sobre el resultado de la aplicación, se comparan los tiempos de respuesta de las páginas generadas contra una página similar hecha con WordPress, así mismo se realizan pruebas con Lighthouse para obtener una valoración global de las páginas y validar los resultados en cuanto a rendimiento, accesibilidad, buenas prácticas y SEO.

Índice

1 INTRODUCCIÓN 1

1.1 MOTIVACIÓN 1
1.2 OBJETIVOS 1

2 ESTADO DEL ARTE 3

2.1 COMPOSICIÓN DE UNA PÁGINA WEB 3
2.2 SEO 3
2.3 LIGHTHOUSE 4

3 GENERACIÓN MANUAL DE HTML 5

4 GESTORES DE CONTENIDO 7

4.1 ¿QUÉ ES UN CMS? 7
4.2 HISTORIA DE LOS CMS 7
4.3 TIPOS DE CMS 7
4.4 WORDPRESS 8
4.4.1 Tecnología 8
4.4.2 Auditoría web 9
4.4.3 Seguridad 10
4.5 ¿POR QUÉ NO USO WORDPRESS? 10

5 DESARROLLO DE UN CMS PROPIO 13

5.1 PETICIÓN DE DOCUMENTOS HTML 13
5.1.1 Reglas del servidor 13
5.1.2 Devolución de documentos HTML 14
5.1.3 Ejecución del lado del cliente 14
5.2 MODELO DE MI APLICACIÓN 15

6 TECNOLOGÍA EMPLEADA 17

6.1 NODEJS 17
6.2 ELECTRON 17
6.2.1 Aplicaciones web 18
6.2.2 Arquitectura 18
6.2.3 Características 19

7. DESARROLLO DEL PROYECTO 21

7.1 ARQUITECTURA 21
7.2 ESTRUCTURA DE FICHEROS 21
7.2.1 Archivos en la raíz 22
7.2.2 Carpetas en la raíz 22
7.3 PANEL 23
7.4 PANEL MODULAR 24
7.4.1 Carga automática de JavaScript 24
7.4.2 Carga automática de módulos 25
7.5 EVENTOS EN EL BACK-END 26
7.6 CONFIGURACIÓN DE LA APLICACIÓN 27

7.7 LOGIN 29
7.8 TEMA OSCURO Y TEMA CLARO 30
7.9 MÓDULO EDITOR 31
7.9.1 Bloques 32
7.9.2 Interfaz del módulo 32
7.9.3 Template 34
7.9.4 Exportación del artículo 35
7.10 MÓDULO FINDER 36
7.10.1 Eventos sobre páginas 37
7.10.2 Sincronización 37
8 MEJORAS FUTURAS DE LA APLICACIÓN 39
8.1 MÁS TIPOS DE BLOQUES 39
8.2 MÓDULO PRODUCTOS 39
8.3 MODO OFFLINE Y SINCRONIZACIÓN ASÍNCRONA 39
9 PRUEBAS DE RENDIMIENTO 41
10. CONCLUSIÓN 47
BIBLIOGRAFÍA 48
ANEXO A: LIGHTHOUSE WORDPRESS VACÍO 53
ANEXO B: LIGHTHOUSE "TECHCRUNCH.COM" 59
ANEXO C: LIGHTHOUSE MI PÁGINA 65
ANEXO D: LIGHTHOUSE MISMA PÁGINA EN WP 71

Índice de ilustraciones

Ilustración 1: Resultado Lighthouse sobre WordPress recién instalado ... 9
Ilustración 2: Resultado Lighthouse sobre "techcrunch.com" ... 10
Ilustración 3: Arquitectura del sistema ... 21
Ilustración 4: Panel vacío ... 23
Ilustración 5: Esquema objeto proxy ... 24
Ilustración 6: Configuración inicial de la aplicación ... 27
Ilustración 7: Crear usuario de la aplicación ... 28
Ilustración 8: Proceso de Login ... 29
Ilustración 9: Interfaz con tema claro ... 30
Ilustración 10: Definición de colores tema claro y oscuro ... 31
Ilustración 11: Bloque de ejemplo ... 31
Ilustración 12: Interfaz del editor ... 33
Ilustración 13: Metadatos del artículo ... 34
Ilustración 14: SHTML entrada de blog ... 35
Ilustración 15: Ejemplo de una entrada ... 35
Ilustración 16: Módulo finder ... 36
Ilustración 17: Articulo en mi web ... 41
Ilustración 18: Articulo en WordPress ... 42
Ilustración 19: Tiempos de carga mi web ... 42
Ilustración 20: Tiempos de carga WordPress ... 43
Ilustración 21: Tiempo de respuesta humano [25] ... 44
Ilustración 22: Lighthouse mi página ... 45
Ilustración 23: Lighthouse misma página WordPress ... 45

1 Introducción

1.1 Motivación

A lo largo de los diferentes desarrollos de aplicaciones y sitios web que he llevado a cabo, me he encontrado una problemática recurrente a la hora de gestionar el contenido del que se compone debido a la necesidad de utilizar herramientas externas, cuando este varía con frecuencia o es necesario generar nuevo contenido diariamente, como sucede en un *blog,* por ejemplo. Un *blog* es una sección casi imprescindible hoy en día en cualquier proyecto y la primera vez que me enfrenté a la problemática de gestionarlo, fue en el desarrollo de la web para la empresa Alien Ventures, donde se optó por contenido estático generado a mano por un administrador. Esta opción no es la mejor ya que supone mucho esfuerzo de mantenimiento, pero es la que menos desarrollo en un principio requiere y por eso se optó por hacerlo de esa forma. La segunda, y la que me hizo reflexionar sobre una alternativa mejor, fue mi página web personal, en la que la necesidad de incluir un *blog* y poder publicar de manera rápida y eficiente era imprescindible para mí.

Desde mi experiencia, para gestionar el contenido de un blog hay tres opciones: cada vez que haya que publicar un nuevo contenido, generar manualmente los HTML (siglas en inglés de *HyperText Markup Language*) correspondientes, recurrir a alguna herramienta existente que te supla esta tarea o desarrollar tu propio sistema de publicación de contenido.

Dependiendo de la frecuencia con la que el contenido de una página web tenga que ser actualizado o requiera de añadir o eliminar contenidos, la opción de generar o editar esos archivos HTML puede llegar a ser viable o no, además debemos tener en cuenta que entrar a editar código no es algo accesible para cualquier persona.

Utilizar un gestor de contenido, o CMS (siglas en inglés de *Content Management System*) como los llamaremos a partir de ahora, supone una serie de ventajas, pero también una gran cantidad de desventajas que veremos más adelante y que han motivado este trabajo.

Desarrollar una herramienta de publicación propia es una tarea laboriosa y compleja, que no se encuentra al alcance de un usuario básico o medio que desea gestionar su *blog* personal o incluso el de una empresa.

1.2 Objetivos

Antes de comenzar a desarrollar la aplicación, había una serie de requisitos esenciales para mí. La herramienta se basaría en tecnología web, esto es porque es un área en la que no hemos profundizado mucho en el grado y quería ampliar mis conocimientos ya que es muy demandada en el mercado laboral, por lo que he podido experimentar buscando empleo.

La herramienta ha de estar pensada para que una persona sin conocimientos de programación web, sea capaz de poder escribir un artículo y publicarlo. El contenido generado por el gestor ha de ser contenido estático y optimizado para el SEO (siglas en inglés de *Search Engine Optimization*), concepto que veremos más adelante. La aplicación, aún basándose en tecnología web, no necesariamente significa que se acceda a ella desde un navegador, de hecho, la aplicación a desarrollar en este TFG será de escritorio y multiplataforma, de manera que usuarios tanto de Windows, Mac y Linux tengan la posibilidad de instalarla en sus ordenadores y empezar a funcionar.

Este trabajo de fin de grado pretende exponer y explicar la aplicación resultante cumpliendo con los objetivos marcados, así como dar una visión general sobre las soluciones existentes en la actualidad y ampliar información sobre ellas.

2 Estado del arte

Este trabajo se basa completamente en el desarrollo automatizado, gracias a mi aplicación, de software web. No solo el producto final que genera la aplicación es web, si no que la aplicación en sí, su interfaz está generada como si una web se tratase con un *Back-End* en JavaScript interpretado por NodeJS como veremos más adelante. Para un lector no familiarizado con este tipo de tecnología, a continuación, se explican distintos lenguajes y herramientas que serán utilizadas a lo largo del TFG.

2.1 Composición de una página web

Desde mi experiencia como desarrollador web, una página, en su mínima expresión, es un documento HTML que define los elementos que el navegador tiene que representar por pantalla. Un documento HTML se compone principalmente de dos bloques principales, una cabecera donde se especifica información para los navegadores y para los motores de búsqueda, como Google, entre otra información que puede aparecer también en esta cabecera. El otro bloque principal es el cuerpo de la página, donde mediante distintas etiquetas definimos los elementos, estos elementos pueden ser de distintos tipos y si que es importante mantener una estructura, para que nuestra página este bien diseñada y sea fácilmente interpretada.

A este documento se le pueden añadir otros lenguajes que aportan más características y nos permiten personalizar y dotar de funcionalidad a nuestra página. El lenguaje CSS (siglas en inglés de *Cascading Style Sheets*) nos da la posibilidad de especificar el estilo concreto con el que queremos representar un elemento del documento HTML, por ejemplo, que cierto texto tenga un tamaño de doce *píxeles* o que un bloque tenga un fondo de color rojo.

JavaScript es un lenguaje de programación, que puede ser ejecutado en el navegador, y dota a las páginas web de dinamismo, nos permite modificar los elementos del documento HTML, lanzar eventos en respuesta a iteraciones del usuario con la interfaz (pulsar en un botón, hacer *scroll*…) o incluso realizar peticiones extra desde el navegador al servidor.

2.2 SEO

La optimización de páginas web para buscadores es uno de los fines más perseguidos por las empresas en la actualidad a la hora de desarrollar una web. Esto lo aprendí trabajando en Arsys (proveedor de infraestructura *cloud*), donde posicionar una de tus páginas en los primeros resultados ante una determinada búsqueda, se traduce en un incremento de las ventas y, por lo tanto, en un incremento de los beneficios.

En diferentes formaciones por agencias especializadas en SEO, aprendí que el obtener una buena puntuación por parte de los diferentes motores de búsqueda existentes, depende de muchos factores y los algoritmos utilizados para puntuar

tu página varían cada poco tiempo, por lo que posicionar tu web es un trabajo de actualización continua. Aunque estos criterios puedan variar con el paso del tiempo, generar una página web con una estructura correcta, por ejemplo, que no haya un h2 (subtítulo) por encima de un h1 (título), o la velocidad con la que carga tu web, son atributos de la página que siempre favorecerán obtener una mejor puntuación y que serán buscados en las páginas generadas con mi aplicación.

2.3 Lighthouse

Para poder valorar la calidad de una página web, voy a hacer uso de la herramienta Lighthouse. Lighthouse es un software de código abierto desarrollado por la comunidad e impulsado por Google para realizar auditoría de sitios web [1].

A partir de una URL (siglas en inglés de *Uniform Resource Locator*), Lighthouse lanza una serie de cargas sobre tu página web y analiza los resultados obtenidos, estas cargas las simula tanto para una versión de escritorio como versión móvil. A partir de los resultados obtenidos, Lighthouse otorga una puntuación de cero a cien en cuatro categorías distintas: *performance*, *accessibility*, *best practices* y SEO [2].

La nota obtenida en cada una de las categorías es una media ponderada del resultado de cada una de las métricas que la componen. La categoría que más nos interesará para este trabajo es la de rendimiento, dentro de ella encontramos como métricas el índice de velocidad de la página, el tiempo total que la web permanece bloqueada al cargar, el tiempo en el que en navegador hace una primera representación gráfica del contenido, entre otras [3].

El resumen de las auditorías realizadas sobre las diferentes webs, serán puestas como ilustración, mientras que el informe completo estará adjuntado como anexo por si el lector quiere consultarlo en profundidad.

3 Generación manual de HTML

Un modelo de generación manual de contenido embebido en HTML es un enfoque clásico, que con el paso de los años a quedado atrás, aun que, en mi experiencia, siguen existiendo casos en los que este modelo es utilizado. El primer paso es la construcción de una estructura básica que va a seguir cada una de tus publicaciones en la página web. Una vez esta estructura es creada y acompañada de estilos y sus scripts correspondientes, el generar nuevas páginas similares, pero con diferente contenido, no supone un gran esfuerzo en cuanto a desarrollo. Los pasos que seguir serían, replicar el código en una nueva página y reemplazar el contenido existente por el nuevo.

Aunque no suponga un gran esfuerzo en cuanto a nuevo desarrollo, sí que supone una mayor pérdida de tiempo a la hora de mantenerlo e incorporar nuevo contenido. En un primer momento esta opción sería la más rápida de implantar, pero la menos útil a largo plazo.

También debemos tener en cuenta que, siguiendo este modelo, la persona que genere contenido ha de tener una noción básica de maquetación HTML, mientras que, con las otras dos alternativas podríamos ampliar el desarrollo de esta tarea a cualquier persona. En este modelo el error humano se incrementa, puede haber errores en las etiquetas HTML y eso generar problemas de visualización.

En base a mi experiencia previa, eligiendo este método como solución a la implantación de un *blog*, esto quedó totalmente descartado, buscando una solución más efectiva y cómoda para mí.

4 Gestores de contenido

4.1 ¿Qué es un CMS?

Un CMS no es más que un software pensado para ser utilizado por cualquier usuario, sin necesidad de conocimientos técnicos más allá del uso del programa, para la creación, edición y publicación de contenido, generalmente de sitios web.

Existen diferentes tipos de CMS, según el propósito para el que va a ser utilizado, por ejemplo, un blog personal, una página empresarial o una tienda de comercio electrónico, entre otros fines.

Estos CMS gestionan bases de datos a través de una interfaz, utilizadas por la página web para mostrar el contenido de manera independiente a los estilos, por lo que permite cambiar la manera de visualizarlo sin necesidad de que este sea modificado. Están pensados para que una vez implantados, estos puedan ser utilizados por los editores dejando las tareas de mantenimiento y actualizaciones a los desarrolladores [4].

4.2 Historia de los CMS

Los primeros CMS de la historia se basaban en simples comandos en el servidor los cuales, gracias a la aparición del DOM (siglas en inglés de *Document Object Model*) que es la representación de la estructura del documento y que nos otorga una API para poder manipularlo [5], eran capaces de a partir de una base de datos, modificar el contenido HTML que finalmente se le sirve al cliente.

Estos primeros CMS fueron creados por empresas con el fin de utilizarlos para ellos mismos, adaptándolos así a sus propias necesidades entre los años 1990 y 2000. A partir de entonces, estos CMS se fueron expandiendo y abarcando muchas más funciones para las empresas, las cuales con estos softwares veían como las tareas de gestión de contenidos se simplificaban. Simultáneamente aparecieron CMS de código abierto disponibles para todos los usuarios como WordPress (https://wordpress.org/) del que hablaremos más adelante, Joomla (https://www.joomla.org/) que no solo permite gestionar el contenido si no crear foros, encuestas y calendarios entre otras funcionalidades, Drupal (https://www.drupal.org/) desarrollado por la comunidad con características muy parecidas a las de Joomla, PrestaShop (https://www.prestashop.com/es) o Magento (https://magento.com/) estos dos últimos gestores están orientados a comercios electrónicos [6]. Todos estos CMS están desarrollados en PHP (siglas en inglés de *Hypertext Preprocessor*) y se apoyan en la utilización de bases de datos, aproximación muy distinta a la del gestor de contenido desarrollado en este TFG.

4.3 Tipos de CMS

Hoy en día encontramos distintos tipos de CMS en función de la utilidad que se le vaya a dar [6], estos son:

- Creadores de páginas estáticas, están pensados para generar archivos los cuales no van a ser modificados muy a menudo como por ejemplo una página web que muestra simple información sobre una empresa. Wix (https://es.wix.com/) es un ejemplo de este tipo de constructores de webs [6].
- *Headless*, se trata de los gestores de contenido que separan el contenido y la presentación que hacemos de este siguiendo el modelo vista-controlador donde el contenido no depende para nada de los estilos dados [6].
- Híbrido, son una mezcla entre los constructores de páginas web y los CMS *headless*, nos permiten crear y editar nuestras páginas, pero a su vez podemos enviar contenido a plataformas *headless* [6].
- Plataformas de experiencia digital, o DXP, son paquetes donde están incluidas todas las herramientas que una empresa necesita para la automatización de contenido en webs, aplicaciones y dispositivos de IoT (siglas en inglés de *Internet of Things*) así como un aprendizaje automático de los datos generados [7].

Si atendemos al modelo de comercialización de estos CMS podemos distinguir tres tipos [8]:

- CMS de código abierto, este tipo de software no tiene coste de entrada, es el más común en cuanto a los gestores utilizados por el público en general y es el modelo de comercialización que seguirá mi aplicación [8].
- CMS propietario, estos CMS pueden ser o bien privativos que pertenecen a una empresa para su uso interno y no los comparten con el resto de los usuarios o bien son vendidos a través de un proceso de licenciamiento a terceras personas [8]. Aquí se encontrarían por ejemplo los gestores que utilizan los periódicos para sus páginas web.
- CMS como SaaS (siglas en inglés de *Software as a Service)*, consiste en CMS alojados en la nube y se suele pagar a modo de suscripción [8].

4.4 WordPress

WordPress es un CMS de código abierto lanzado en 2003 por su fundador Matt Mullenweg pensado en un principio para la creación de *blogs* en sitios web. Se trata de una bifurcación del ya desaparecido CMS b2/cafeblog [9]. Con el paso de los años, WordPress se ha convertido en una herramienta capaz de crear cualquier tipo de sitio web gracias a su evolución y a la multitud de extensiones y componentes desarrollados en PHP por la comunidad o por ti mismo, esto permite que no sea algo que te restrinja a lo que te marcan, convirtiéndose así en el CMS más popular de la actualidad. Se dice que tiene una cuota del 30% de todas las páginas web del mundo [10] [11].

4.4.1 Tecnología

WordPress aporta una interfaz gráfica accesible desde el navegador y alojada en el mismo servidor web donde ha sido instalado, permite a desarrolladores o

usuarios finales, administrar o actualizar contenido. Este proceso consiste en ejecución de código PHP del lado del servidor, para realizar consultas SQL (siglas en inglés de *Structured Query Language*) sobre una base de datos relacionales, usando concretamente MySQL. Posteriormente a la hora de mostrar el resultado final de la página, será la ejecución de más código PHP, el que, tras consultar a esta base de datos, genere un documento HTML que enviar al navegador. La manera en que ese contenido es mostrado por el navegador a un usuario final depende del tema, que consiste en un conjunto de código PHP que marca la distribución de contenido, hojas de estilo que definen como mostrarlo y JavaScript para gestiones de la interfaz [12].

En mi opinión, aun que este haya sido pensado para un manejo fácil, tras el paso del tiempo, la realidad es que el proceso de implantación de un nuevo sitio WordPress, es una tarea tediosa y que requiere de conocimientos previos. Es por lo que generalmente son desarrolladores los que realizan el proceso de implantación y de mantenimiento y el cliente final gestiona solo la publicación y modificación de contenido.

4.4.2 Auditoría web

Me parece interesante, a modo de prueba, como se comporta un WordPress recién instalado cuando es auditado con Lighthouse. Para ello he instalado WordPress en mi servidor web y no he realizado ninguna configuración ni modificación del contenido que viene inicialmente, se encuentra con el tema por defecto y sin ningún *plugin* añadido. Este es el resultado de la auditoría.

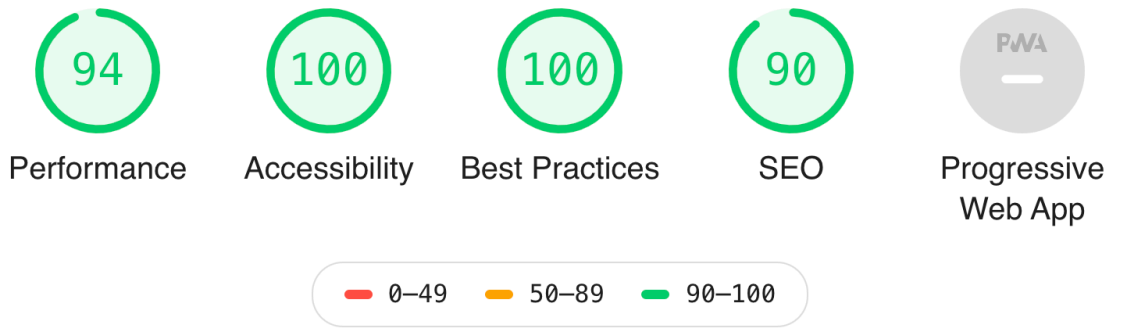


Ilustración 1: Resultado Lighthouse sobre WordPress recién instalado

En vista de los resultados obtenidos en el Anexo A, podemos decir que hemos logrado una buena puntuación, sin embargo, llama la atención que sin haber añadido absolutamente nada al WordPress, no obtengamos de serie una puntuación de cien, en cuanto a rendimiento. No obtener la máxima puntuación en SEO es algo que podemos comprender, ya que esta valoración depende de completar metadatos utilizados posteriormente por los buscadores que no ha sido completada.

Para poder añadir funcionalidad a nuestro sitio web, Worpress ofrece la posibilidad de instalar *plugins* de terceros, desarrolladas por la comunidad. Cada uno de ellos tiene un comportamiento diferente y puede llegar a ser complicado la utilización de un *plugin* para casi cualquier cosa que desees añadir. Según tu

sitio WordPress va creciendo, es más complejo y tiene más contenido, su rendimiento es prácticamente imposible mantenerlo alto, obteniendo pésimas valoraciones por parte de Lighthouse. Por ejemplo, TechCrunch, uno de los portales más famosos de tecnología del mundo, está hecho con WordPress y a continuación podemos ver el resultado de su auditoría en el Anexo B.

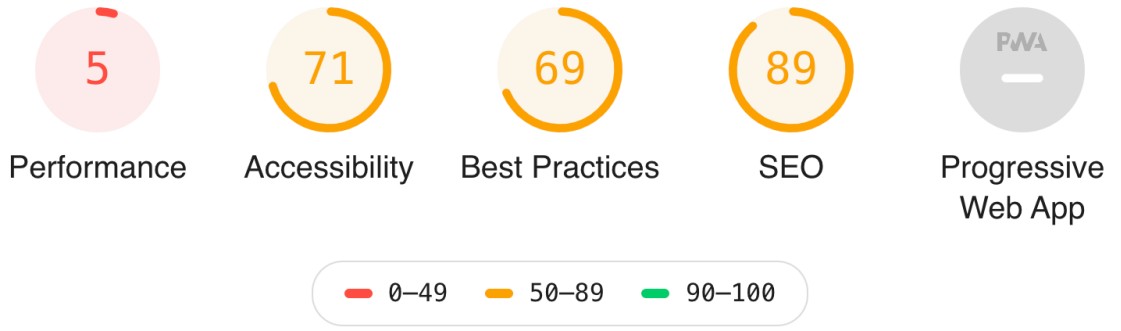


Ilustración 2: Resultado Lighthouse sobre "techcrunch.com"

Es lógico pensar que según aumenta la complejidad de una página web, es muy difícil mantener un buen rendimiento, y más difícil aún es mantenerlo con WordPress, ya que llegas a perder el control de toda la ejecución y peticiones que existen por detrás.

4.4.3 Seguridad

Si tenemos en cuenta el dato de que el 30% de todas las webs del planeta están hechas con WordPress [11], tiene sentido que ésta sea una de las plataformas más *hackeadas* del mundo [13]. El tener mucha popularidad no siempre es algo positivo, y es que si construyes tu página web con WordPress, estas expuesto a cualquier problema de seguridad que esta plataforma tenga, ya que se usa ejecución en el servidor para cualquier tarea.

El panel de control de WordPress se encuentra accesible a través de internet para cualquier persona, generalmente en la URL “dominio/wp-login.php”, esto deja expuesto el proceso de iniciar sesión ante cualquier atacante.

4.5 ¿Por qué no uso WordPress?

WordPress es una herramienta muy potente y útil precisamente para el cometido que yo necesitaba a la hora de implantar un *blog* en mi página web personal, sin embargo, esta nunca fue una opción válida para mí. Una de las problemáticas más grandes que me encontré, es que yo necesitaba una herramienta capaz de solo gestionar el contenido con independencia total de donde fuera a ser insertado ya que el resto del sitio web ya estaba construido con un desarrollo propio, por lo que insertar un blog de WordPress requeriría de la laboriosa tarea de crear un tema replicando exactamente la interfaz y comportamiento del resto del mi sitio web.

Que se acceda al panel de gestión a través del navegador y que éste resida en el servidor, es una manera de exponer innecesariamente a la plataforma de

posibles intrusiones no deseadas o ataques maliciosos, por este motivo decidí que mi aplicación fuese de escritorio, protegiéndola más del mundo exterior al encontrarse en local en mi ordenador.

WordPress añade demasiada funcionalidad y código que se vuelve innecesario cuando solo necesitas una tarea en específico, en mi caso una gestión simple de artículos, por lo que veo mejor otra herramienta enfocada en concreto a lo que necesito y que no añada más código del necesario. La creación de estas páginas de manera dinámica es otra de las razones por las que no estoy dispuesto a usar la herramienta, ya que el contenido deseado de mi web es estático y el objetivo es obtener el mayor rendimiento posible, evitando consultas a bases de datos o ejecuciones del lado del servidor.

5 Desarrollo de un CMS propio

Visto que las dos primeras alternativas comentadas no eran de mi agrado por no cumplir los requisitos esenciales para mí, tomé la decisión de embarcarme en el proyecto de diseñar una herramienta propia. Para entender las decisiones que tomé en cuanto a su diseño, es importante tener claros algunos conceptos sobre diferentes arquitecturas posibles.

5.1 Petición de documentos HTML

Un usuario solicita al navegador acceder a un recurso a través de una dirección URL, esta dirección no es más que un alias que los servidores DNS (siglas en inglés de *Domain Name System*) traducen a una dirección IP (siglas en inglés de *Internet Protocol*) perteneciente al servidor web donde se aloja la página. El navegador envía una petición HTTP/HTTPS (siglas en inglés de *Hypertext Transfer Protocol Secure*), con sus cabeceras y método correspondiente al servidor web. Utilizando las cabeceras, el servidor web responde con el recurso que posteriormente el navegador interpretará. A parte de los elementos que componen el recurso, se pueden especificar recursos extra, que son necesarios para la construcción final de lo que el usuario va a poder ver, estos pueden ser hojas de estilos, código JavaScript, fotografías, videos… Es según se va encontrando estas referencias, cuando el navegador solicita al servidor web estos recursos extra, que son enviados de vuelta.

Uno de estos recursos por los que pregunta el navegador puede ser un documento HTML y este no tienen porque estar completo en el momento previo a la petición, sino que puede ser generado o modificado a la hora de realizarla o a posteriori. Durante todo este proceso encontramos diferentes pasos hasta que obtenemos un resultado final.

5.1.1 Reglas del servidor

Bien estemos hablando de un servidor web Apache, Nginx o NodeJS por ejemplo, todos ellos cuentan con una serie de reglas a la hora de procesar una petición para saber cómo tienen que manejar cada una de ellas. En el caso de Apache se utiliza el fichero “.htaccess” en el cual se pueden especificar redirecciones de unas URL a otras, reescritura de la propia URL, métodos de compresión para recursos…

Este paso está siempre presente en cualquier servidor web e independientemente de como finalmente vayamos a generar el HTML resultante, tenemos que contar con este tiempo extra que añade pasar por este procesamiento de la petición.

5.1.2 Devolución de documentos HTML

Una vez la petición ha pasado el procesamiento de esta siguiendo las reglas especificadas, el servidor web procede a devolver el documento solicitado por el navegador. En este punto se pueden dar dos casuísticas.

5.1.2.1 Ejecución

El documento solicitado no esta listo para ser devuelto en un primer momento, ya que carece total o parcialmente de su contenido y requiere de un procesamiento previo. Este procesamiento lo realiza la ejecución de un código en el servidor, por lo que estaremos utilizando recursos de CPU (siglas en inglés de *Central Processing Unit*) y memoria RAM (siglas en inglés de *Random Access Memory*) de la máquina que, dependiendo de su dimensionamiento, tipo de ejecución y número de peticiones que recibamos, puede suponer un problema para el rendimiento de nuestro servidor web.

Son múltiples los lenguajes utilizados del lado del servidor, por ejemplo, PHP, Java, Ruby, Python o incluso JavaScript. Este modelo supone una gran ventaja a la hora de simplificar el contenido de una página web ya que de manera automatizada y dependiendo del recurso que solicite el usuario, el servidor web es capaz de generar el documento HTML correspondiente. Además, podemos ir un paso más allá y consultar datos, ya sea en una base de datos, en un archivo JSON (siglas en inglés de *JavaScript Object Notation*) o mismamente en un archivo de texto plano, para generar el contenido que tendrá el HTML resultante.

Independientemente de la optimización del código, de la potencia del servidor o los motores de bases de datos utilizados, esto supone añadir más pasos a la hora de devolver un documento HTML, que se traduce en más tiempo de espera para el usuario final. Además, si nuestro modelo se vuelve más complejo y consta de muchas dependencias externas cada una de estas llamadas incrementa el tiempo de respuesta.

5.1.2.2 Ficheros estáticos

El fichero solicitado está listo para ser devuelto por el servidor web, ya que éste cuenta con la totalidad de su contenido y no requiere de un procesamiento extra que lo complete ni consultas a bases de datos. En este modelo obviamos pasos extra consiguiendo un menor tiempo de repuesta y liberando al servidor de consumo de recursos.

Por contrapartida perdemos la flexibilidad y dinamismo que nos aporta la ejecución del lado del servidor, limitando ciertas funcionalidades.

5.1.3 Ejecución del lado del cliente

El navegador una vez ha obtenido una respuesta, procede a interpretar y realizar el *renderizado* en base a las etiquetas HTML, los estilos CSS y la ejecución del código JavaScript presente. Dentro de esta ejecución del lado del cliente

podemos encontrar otros lenguajes como C o C++ gracias a la aparición de WebAssembly que no es más que un formato de código binario portable [14].

Estos *scripts* pueden gestionar eventos sobre la propia interfaz de la web, pero también encontramos otros *scripts* que se encargan de modificar el propio contenido de la página realizando peticiones adicionales al servidor. Esto ultimo es lo que *frameworks* como Angular, React o Vue hacen, no son más que librerías JavaScript que aportan herramientas a los desarrolladores para la elaboración de interfaces y sobre todo de páginas SPA (siglas en inglés de *Single Page Aplication*). Este tipo de generación de páginas presenta al igual que la ejecución del lado del servidor una serie de desventajas, ya que estamos añadiendo más pasos antes de poder mostrar un contenido final al usuario y esto supone más tiempo de espera para el usuario.

5.2 Modelo de mi aplicación

Teniendo en cuenta que el objetivo de mi aplicación es gestionar el contenido de una página web, tiene sentido pensar que el modelo perfecto a seguir es la ejecución del lado del servidor consultando bases de datos donde obtener el contenido de cada uno de los artículos del *blog* y quizás utilizar algún tipo de *framework* JavaScript para la generación de la interfaz. Esta manera es la más utilizada y común de hacerlo, sin embargo, en mi experiencia como analista SEO, este modelo no es el óptimo por varios motivos.

Los motores de búsqueda como Google, en la actualidad, recorren internet consultando las páginas existentes e indexando su contenido en sus páginas de resultados. Concretamente Google *renderiza* tu página web dos veces, una primera y la más importante de las dos, simulando estar en un dispositivo móvil, y una segunda, simulando estar en escritorio. Durante el proceso de *renderización* llegan a tomar varias capturas en diferentes momentos del proceso de carga. Cuando tu tienes un HTML generado desde un principio en el cliente, te aseguras de que cuando Google haga una captura de tu web para comprobar el contenido visible, este esté presente. Esto lo conseguimos tanto con ficheros estáticos que posteriormente no requieren de peticiones extra desde el cliente, como con páginas generadas en ejecución en el servidor. Sin embargo, cuando utilizamos *frameworks* JavaScript del lado del cliente, nos arriesgamos a que en el momento en el que Google comprueba nuestra web, parte del contenido o el más relevante no este siendo todavía mostrado y por lo tanto no sea indexado en los resultados de búsqueda, esto pasa porque Google no se queda mirando infinitamente tu web, si no que espera unos segundos mientras toma las capturas y se marcha. Como uno de los objetivos de las páginas generadas con mi aplicación es el favorecer la optimización de las búsquedas, utilizo HTML con todo el contenido embebido en el.

Otra meta que me puse, es el intentar que el tiempo de respuesta al consultar la web sea el mínimo posible, y por otro lado, también quería que la aplicación y los ficheros de contenido que generara fuera independiente al servidor en el que la web va a ser alojada y las tecnologías que este pueda tener, de manera que

no sea un requisito que el servidor cuente con PHP o con NodeJS si no que pueda ser usado en cualquier tipo de servidor web. Es por esto por lo que la ejecución del lado del servidor quedó descartada, y por lo tanto el modelo de consultas a bases de datos también, reduciendo así tanto recursos en el servidor como llamadas y tiempos de espera.

6 Tecnología empleada

6.1 NodeJS

Los lenguajes de *scripting* están pensados para utilizarse junto a otros leguajes de programación o de marcado como HTML, este tipo de lenguajes son interpretados y no requieren de un proceso de compilación previo a la ejecución del código [15].

Uno de los lenguajes de *scripting* más populares del mundo es JavaScript, este lenguaje orientado a objetos fue incorporado por primera vez en el navegador Netscape y copiado por Microsoft en Internet Explorer bajo el nombre de JScript. Desde entonces JavaScript no ha hecho más que aumentar su popularidad y funcionalidad [16]. Junto con la aparición del DOM los programadores web son capaces de manipular elementos de la página y crear un dinamismo en determinados elementos o interacciones del usuario con la interfaz. JavaScript es interpretado por motores JavaScript integrados en los navegadores, por ejemplo, Google Chrome, utiliza un motor de código abierto escrito en C++ llamado V8.

Teniendo en cuenta la popularidad y utilidad de este lenguaje de *scripting*, cobra sentido la idea de poder ser utilizado fuera de los navegadores, ya sea en una computadora personal o en un servidor. Para solventar esto aparece NodeJS, se trata de un interprete de JavaScript, utilizando el motor V8 de Google, basado en la ejecución de eventos asíncronos, pensado sobre todo para aplicaciones de red que puedan ser escalables [17].

Para la gestión de las llamadas asíncronas, NodeJS incorpora la posibilidad de enviar funciones en las llamadas a modo de *callback*, que son ejecutadas una vez el proceso a terminado. Una de las ventajas es que los procesos nunca quedan bloqueados y está diseñado para trabajar sin hilos, aunque existe la opción si el desarrollador lo desea [18].

Personalmente, la utilización de NodeJS para desarrollar aplicaciones de escritorio, como para la ejecución en el servidor, supone una enorme ventaja ya que se unifica el desarrollo *Back-End* y *Front-End* en un único lenguaje, JavaScript. Esto siempre supone un mantenimiento más sencillo y una menor tarea en la especialización del desarrollador en diferentes lenguajes. Gracias a las llamadas asíncronas, podemos conseguir de una manera sencilla mejores rendimientos que en un lenguaje concurrente.

NodeJS, por mi experiencia, es una tecnología al alza cada vez más presente en todos los desarrollos informáticos y que no ha sido estudiada en el grado, es por esto también que aprovecho mi trabajo de fin de grado para obtener información y habilidades con esta tecnología.

6.2 Electron

Electron es un *framework* para desarrollar aplicaciones multiplataforma de escritorio con JavaScript, propiedad de Github, que a su vez es propiedad de Microsoft. Proporciona una serie de APIs (siglas en inglés de *Application Programming Interface*) a nivel de sistema operativo, y utiliza Chromium para la visualización de páginas web a modo de interfaz.

6.2.1 Aplicaciones web

Desde la creación de Electron por parte de Github y el desarrollo de sus dos aplicaciones originalmente más famosas, el editor de código Atom y el propio cliente de Github, es una tecnología que ha ido cobrando más y más fuerza. La adquisición de Github por parte de Microsoft dio paso a que aplicaciones tan populares como Skype o Teams pasaran a estar sobre esta tecnología. En la actualidad encontramos una gran cantidad de aplicaciones famosas creadas con Electron, como pueden ser Whatsapp, VScode, Discord… [19]

La fuerte presencia que está teniendo Electron, se debe sobre todo a la facilidad con la que se pueden portar aplicaciones de la web a escritorio o viceversa, ya que el código empleado es altamente reutilizable. De esta manera puedes mantener tanto la versión web como la versión de escritorio prácticamente como si fuera un mismo desarrollo, sin tener paralelamente dos completamente independientes. También es una gran ventaja el hecho de poder exportar tus aplicaciones a MacOS, Linux y Windows de manera automática sin tener que hacer modificaciones en el código, teniendo la posibilidad de incluso generar la aplicación en modo portable.

A mi modo de ver, las aplicaciones de escritorio bajo tecnología web es el futuro y el camino que está siguiendo el mercado, el buen rendimiento, seguridad y accesibilidad que ofrece este modelo de desarrollos suponen considerables ventajas para los desarrolladores.

Otras empresas como Google están apostando fuerte también por las aplicaciones web, sin embargo, ellos optan por su propia tecnología llamada *Progressive Web Apps*, la cual consiste *grosso* modo de una instancia de Chrome con una página en concreto y eliminando elementos de la interfaz del navegador.

6.2.2 Arquitectura

Electron consta de dos tipos distintos de procesos. El llamado proceso principal, es el encargado de generar páginas web a modo de interfaz gráfica, solo existe uno de este tipo por aplicación y es especificado en el JSON “package.json” el fichero que contiene el código de este proceso. Cada vez que se genera una nueva página web es creado un nuevo proceso llamado proceso visualizador, aislado de otros procesos de este mismo tipo, que pueden existir cuantos se requiera por parte de la aplicación. Una vez la instancia de la página es destruida por el proceso principal, este proceso visualizador también es destruido. El proceso principal es ejecutado con NodeJS, es por esto por lo que, podemos acceder a todas sus librerías y rutinas [20].

La principal diferencia a la hora de implementar paginas web como interfaz en este tipo de aplicaciones con respecto a la ejecución que podrían tener en un navegador, es que Electron proporciona al JavaScript la opción de comunicarse con rutinas del sistema operativo, cosa impensable en páginas web convencionales, ya que esto supondría serios problemas de seguridad y de fuga de recursos.

Para entender bien el funcionamiento de Electron, vamos a hacer un símil con la arquitectura de una página web convencional. Electron consta de un *Front-End* y de un *Back-End*. El *Front-End* es manejado por el proceso visualizador y sería equivalente a un navegador, al igual que cualquier página web consta de ficheros HTML, ficheros CSS y ficheros JavaScript que, a parte de su funcionalidad corriente, son capaces de comunicarse con el *Back-End* utilizando módulos ofrecidos por Electron. El *Back-End* es manejado por el proceso principal y sería el equivalente a un servidor web, utiliza JavaScript interpretado por NodeJS y es el encargado de responder a las peticiones de los procesos visualizadores y de la creación y destrucción de ellos.

La comunicación entre *Front-End* y *Back-End*, es decir, entre el proceso principal y los distintos procesos visualizadores, se lleva acabo por los métodos “ipcRender” e “ipcMain”. El módulo “ipcRender” proporciona una serie de métodos para el envío de eventos sincrónicos (el proceso visualizador se detiene hasta que obtiene una respuesta) o asíncrona (el proceso visualizador continua su ejecución) al proceso principal [21]. El módulo “ipcMain” proporciona métodos para el manejo de eventos recibidos en el proceso principal, teniendo la posibilidad de devolver una respuesta [22].

6.2.3 Características

Ya que desde Electron podemos acceder a rutinas del sistema operativo, nos aporta la opción de interactuar con características generales de los sistemas operativos como puede ser los atajos de teclado, las notificaciones, arrastrar y soltar archivos o soporte para modo oscuro si el usuario lo tiene configurado en el sistema operativo. También nos da la posibilidad de integrar características de plataformas concretas como puede ser la compatibilidad con la *TouchBar* de los MacBook.

7. Desarrollo del proyecto

La aplicación desarrollada recibe el nombre de Gestor de Bloques o GdB, como nos referiremos a ella a partir de ahora. En los siguientes apartados veremos su funcionalidad y como ésta ha sido implementada.

7.1 Arquitectura

Para el correcto funcionamiento de la aplicación solo es necesario disponer de un servidor web y un dominio que resuelva en la dirección IP y directorio donde GdB sincroniza los archivos HTML. Al ser contenido estático, este gestor de contenido es compatible con servidores Apache con PHP, servidores de NodeJS o cualquier otro tipo de servidor web. No es necesaria la existencia de ningún tipo de base de datos.

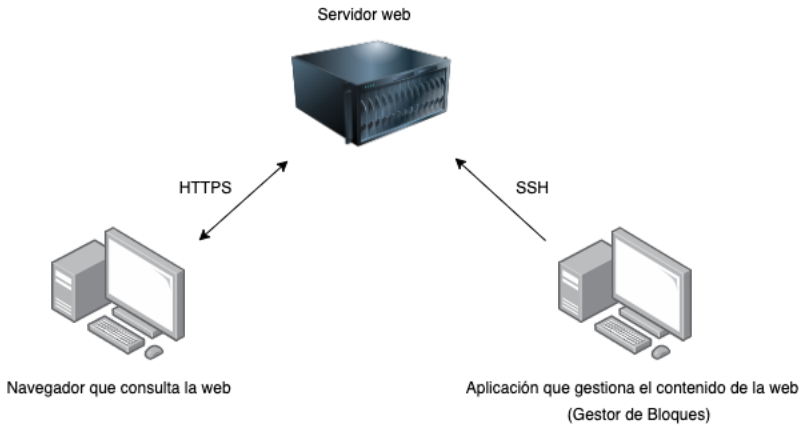


Ilustración 3: Arquitectura del sistema

GdB se instala en el ordenador del usuario que vaya a realizar la gestión del contenido de la página web, tras ser correctamente configurada, esta aplicación se comunica por SSH (siglas en inglés de *Secure Shell*) con el servidor web. A través de este túnel cifrado, GdB es capaz de subir o eliminar archivos HTML que posteriormente serán servidos a un navegador por HTTPS.

Una de las ventajas de que la aplicación vaya instalada en el ordenador, es la posibilidad de hacer compatible la aplicación con una modalidad de edición de contenido *offline* y que posteriormente con conexión, se realice la sincronización con el servidor web. Esta característica no está implementada en este TFG y se queda como una posible mejora para el futuro.

7.2 Estructura de ficheros

La aplicación se compone de unos pocos archivos principales en la raíz del directorio y de algunas carpetas donde se concentran distintos recursos y funcionalidades de la aplicación.

7.2.1 Archivos en la raíz

En la raíz del directorio encontramos los siguientes ficheros:

- “package.json”: cuando estamos creando una aplicación en Electron, lo que realmente estamos creando es una aplicación NodeJS en la que especificamos que estamos utilizando el *framework* de Electron. Esto lo especificamos con el metadato “scripts” en el que indicamos que el comando “start” de NPM (siglas en ingles de *Node Package Manager*), es el gestor de paquetes de NodeJS, ejecute Electon. En este archivo también son especificadas dependencias de la aplicación, es decir, paquetes de NodeJS, así como la versión de la aplicación, su nombre, autor, descripción…
- “main.js”: este archivo JavaScript es el *Back-End* de la aplicación ejecutado por el proceso principal. Es el encargado de responder a las llamadas de los JavaScript del *Front-End* y de crear la ventana de la interfaz web.
- “index.html”: es el archivo HTML utilizado para la interfaz gráfica de la aplicación, sobre este archivo se muestra toda la aplicación ya que se trata de una SPA, es decir, moviéndonos por diferentes apartados de la interfaz no generamos ni destruimos ventanas, cambiamos el contenido sobre el mismo HTML.
- “style.css”: define los estilos generales del panel de la aplicación, existen otros ficheros de estilos, pero para otras partes más específicas de la aplicación.

7.2.2 Carpetas en la raíz

En la raíz del directorio encontramos los siguientes directorios:

- “node_modules”: este directorio contiene cada uno de los módulos NPM de los que hace uso la aplicación, a parte de las dependencias que he necesitado, también se encuentran todos los ficheros pertenecientes al *framework* Electron.
- “modules”: cada una de las secciones del panel de la aplicación es un módulo distinto, esto será visto en profundidad más adelante. En este directorio se encuentran las carpetas de cada uno de los módulos de los que consta el panel.
- “font”: contiene la fuente utilizada por los textos de la aplicación (Roboto), así como la fuente de iconos (Material Icons).
- “auto”: la carpeta auto contiene todos los archivos JavaScript que son utilizados del lado del *Front-End*.
- “data”, almacena ficheros con información utilizada en local por la aplicación.
- “pages”, contiene los HTML y JSONS generados por la aplicación que posteriormente serán sincronizados con el servidor.

- “template”, contiene los ficheros necesarios para que la exportación del contenido sea en el formato deseado, dependiendo de como se tenga que construir ese HTML final.

7.3 Panel

Como he comentado previamente, el panel se compone de un único fichero HTML sobre el que cargamos y descargamos, mostramos y ocultamos contenido desde el JavaScript. El primer paso para la generación de la interfaz es la creación de la ventana desde el *Back-End*, para esto el *framework* nos da el objeto “BroserWindow” en el cual especificamos que la anchura y la altura de la ventana, por defecto será 1200 *pixeles*, siendo la altura y anchura mínimas permitidas de 800 *pixeles*. Además, eliminamos el *frame* por defecto del sistema operativo para posteriormente crear y gestionar los eventos de ventana yo mismo.

El panel se compone de tres elementos básicos: una cabecera, un menú lateral y la sección donde se muestra el contenido. A continuación, se puede ver el panel vacío sin ningún modulo cargado.

Ilustración 4: Panel vacío

Tanto la cabecera como el menú lateral tienen una posición fija, cuando se produce el evento de *scroll* estos no se ven afectados. La sección principal del panel, donde aparecerá el contenido de cada sección del menú, tiene márgenes interiores en la parte superior e izquierda para que los otros dos elementos no se superpongan encima del contenido. En la cabecera hemos incluido nuestros propios iconos de minimizar, maximizar y cerrar la aplicación.

7.4 Panel modular

Una de las intenciones de este panel, es que estuviera abierto a nuevos desarrolladores que quisieran utilizarlo para distintas aplicaciones con otros usos, es por esto por lo que la funcionalidad de moverte a través de los módulos del menú y la carga de estos mismos sobre el panel es independiente al gestor de contenido en sí. Tiene dos peculiaridades que lo hacen modular.

7.4.1 Carga automática de JavaScript

Los JavaScript utilizados en la aplicación desde el *Front-End*, al igual que en las páginas web, son referenciados desde el archivo HTML que especifica que los requiere. La aplicación requiere de muchos JavaScript diferentes, sin embargo, desde el único HTML que tenemos solo referenciamos uno, “part.js”.

Para contener nuestro código JavaScript, utilizamos un objeto que contiene todos los métodos que necesitamos. Para generar esta carga automática de JavaScript he creado un objeto *proxy* que hace de intermediario entre cualquier llamada a un método del objeto y el objeto en sí.

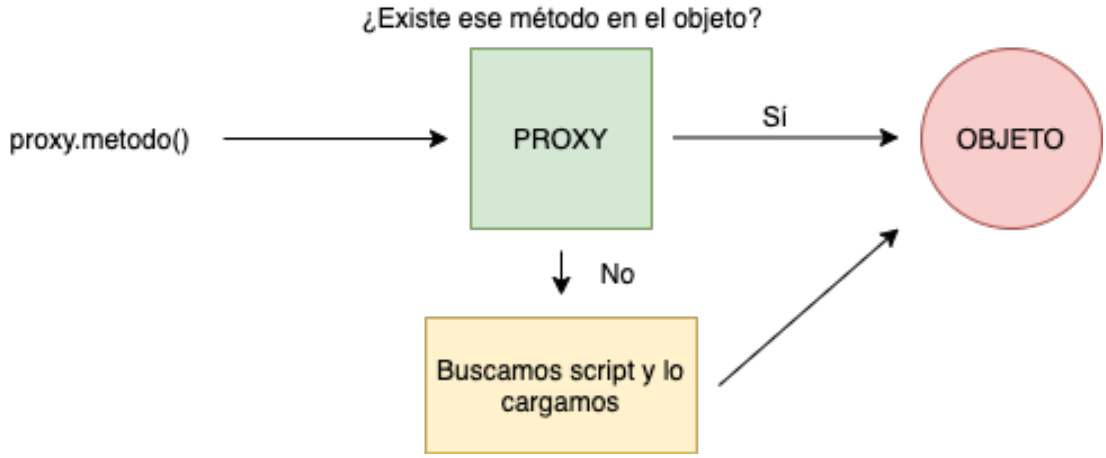


Ilustración 5: Esquema objeto proxy

El objeto proxy ha sido llamado “auto”, las únicas condiciones a la hora de crear un nuevo archivo JavaScript es que sea una extensión de este objeto y que el fichero se llame igual que el método que implementa. Cada vez que llamamos a un método de este *proxy*, comprueba si este método existe en el objeto final. Si este método existe, procede a llamar al método, en caso contrario, el *proxy* busca en el directorio “auto” un archivo JavaScript con el mismo nombre que el método solicitado, finalmente inserta una nueva etiqueta “script” en el HTML incluyendo este fichero.

Con todo esto lo que conseguimos es que la incorporación de nuevos *scripts* en este panel es tan fácil como crear un nuevo fichero en el directorio “auto” y sin tener que incluir su carga en el HTML, la primera vez que sea llamado este método será cargado automáticamente. Al no cargarse todos los *scripts* al

principio ahorramos tiempo en un primer momento, y solo realizamos la carga cuando es realmente necesario.

7.4.2 Carga automática de módulos

Llamamos módulo del panel a cada una de las secciones del menú y que una vez accedido a ellas, se carga su contenido en el espacio destinado para ello. Cada uno de estos módulos es un directorio, dentro del directorio “modules”, que contiene cuatro archivos distintos.

- “content.html”: en este fichero va todo el código HTML del módulo en sí, que posteriormente será embebido dentro del HTML principal del panel.
- “content.css”: en este fichero va todo el código CSS del módulo en sí, que posteriormente será añadido mediante la etiqueta “style” en el HTML principal para cargar esta hoja de estilos.
- “content.js”: en este fichero va todo el código JavaScript que hace efecto sobre las partes de interfaz de la parte del HTML de este módulo, así como de funcionalidad propia de este.
- “module.json”: define las características propias del módulo, utilizadas por el panel para su correcta carga. Es un fichero JSON con las propiedades:
    - “icon”: especifica el icono que ha de salir en el menú.
    - “name”: especifica el nombre del módulo, utilizado también por el menú.
    - “html”: especifica el nombre del fichero HTML del módulo.
    - “css”: especifica el nombre del fichero CSS del módulo.
    - “js”: especifica el nombre del fichero JavaScript del módulo.
    - “order”: especifica el orden en el que tiene que aparecer en el menú.

De esta manera la creación de apartados del panel es completamente independiente al panel en si mismo, dando la posibilidad de incluir lo que queramos, o incluso abriendo la puerta a que desarrolladores creen nuevos módulos que amplíen la funcionalidad del gestor de contenido.

7.4.2.1 Funcionamiento

Una vez la aplicación ha arrancado, el JavaScript hace un escáner del directorio “modules” viendo los módulos existentes, por cada uno de ellos, accede al fichero JSON para coger toda la información del módulo y poder poner el icono y el nombre correspondientes.

A la hora de añadir un nuevo módulo, hay que crear su apartado correspondiente en el menú, por lo que generamos un nuevo elemento HTML “li” que contiene el atributo “data-folder=nombreModulo”, con esto, a la hora de que el usuario pulse sobre un apartado del menú, sabe que módulo hay que cargar.

La primera vez que se entra desde el panel en un módulo, este no existe, por lo tanto, tenemos que cargarlo en el panel. Un fichero JavaScript es el encargado de comprobar si ya existe o no, si existe, ocultamos el módulo previo y

mostramos al que queremos acceder, si no existe, gracias al atributo anteriormente mencionado, sabe a que directorio hay que acceder para conseguir los ficheros HTML, CSS y JavaScript correspondientes que son insertados en el HTML principal del panel.

7.4.2.2 Conflicto entre estilos

Uno de los problemas encontrados al seguir esta lógica de los módulos, es que un desarrollador externo a mí, sus estilos aplicados sobre su módulo entren en conflicto con los estilos generales del panel. Esto ocurre porque no existe una manera en tecnología web de indicar que un fichero de estilos afecte solamente a un módulo del panel, por lo que se podría dar el caso de que reglas de estilos afecten a otros elementos HTML no deseados, provocando errores de visualización en la interfaz.

Para solventar esta problemática, todos los ficheros CSS de módulos, sus reglas de estilos han de comenzar con la condición “[data-module=nombreModulo]”, así nos aseguramos de que estas reglas afecten solo al HTML contenido dentro de la etiqueta que contiene al módulo en sí.

7.4.2.3 Conflicto entre JavaScript

El mismo problema que teníamos con los estilos, lo encontramos con el JavaScript. Al tener un fichero por cada módulo y estos poder haber sido desarrollados por diferentes personas, puede darse el caso de solapamiento de nombres de variables, sobre escritura de métodos o al hacer los “querySelect” para coger elementos del DOM que estemos seleccionando elementos no pertenecientes a dicho módulo.

Para solventarlo, creamos un objeto que contiene por cada uno de los módulos un atributo propio que contiene todo el HTML de ese módulo, a la hora de hacer un “querySelect” en vez de hacerlo sobre todo el DOM, lo hacemos solo sobre este atributo del objeto (objeto.nombreModulo.querySelector('.clase');).

7.4.2.4 Comunicación entre módulos

Al ser los módulos independientes entre sí, no existe una manera directa de compartir información entre ellos ya que sus JavaScript no pertenecen a un mismo objeto o dependen de otro JavaScript común que pueda hacer de intermediario. La solución a este problema ha sido implementar un atributo en el objeto proxy del panel, accesible por cualquier módulo para poder almacenar variables a modo de *cookie* local del ordenador.

7.5 Eventos en el Back-End

Hemos visto como desde el proceso principal hemos creado la ventana donde se muestra la interfaz de la aplicación, aparte de esta función, desde aquí manejamos todas las llamadas que nos llegan desde el *Front-End*.

Para esto ponemos eventos a la escucha para cada una de las llamadas posibles que necesitemos responder, por ejemplo, como hemos eliminado el *frame* del sistema operativo de la ventana, los iconos mostrados en HTML una vez son pulsados, el evento es recogido por el JavaScript del lado del *Front-End* pero este no tiene la capacidad de ejecutar las instrucciones de Electron para realizar la terea correspondiente. Lo que hacen esos eventos es enviar un mensaje al Back-End indicando que se quiere realizar dicha tarea y esta es ejecutada por el proceso principal.

Estos eventos pueden haber sido enviados de manera síncrona o asíncrona, en el caso de ser eventos síncronos, enviaremos una respuesta de nuevo al *Front-End* para que este pueda seguir con su ejecución.

7.6 Configuración de la aplicación

La primera vez que ejecutas la aplicación, una vez esta ha sido instalada en tu ordenador, aparece un proceso de configuración sencillo para otorgarla de la dirección y credenciales del servidor web que va a ser utilizado.

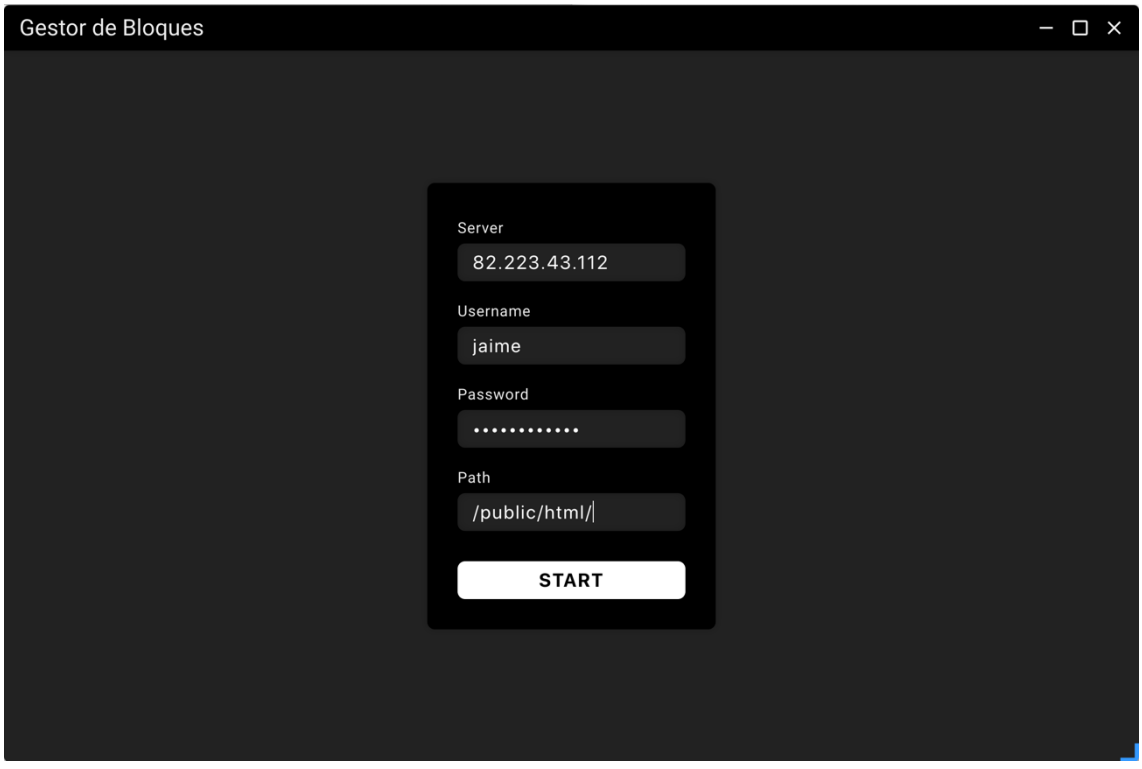


Ilustración 6: Configuración inicial de la aplicación

El campo “Server”, admite tanto la dirección IP de un servidor, como un dominio que resuelva a esa dirección IP. Los campos “Username” y “Password” corresponden a las credenciales SSH del servidor, preferiblemente y por evitar problemas de permisos, es recomendable que el usuario proporcionado sea *root*. El campo “Path” especifica el directorio en el que la aplicación ha de trabajar, este directorio tiene que estar dentro de los directorios del servidor web y que pueda resolver la petición a estos archivos por HTTPS.

Cuando el usuario pulsa el botón “START”, la aplicación procede a establecer una conexión SSH con el servidor, esta conexión la conseguimos haciendo uso de la librería “node-ssh” de NPM. Lo primero que hacemos es ejecutar el comando “ls” en el directorio especificado por el usuario y comprobar que no exista ya una carpeta llamada “blog”, que es el nombre del directorio utilizado por la aplicación. Como lo estamos configurando por primera vez en un servidor donde esta aplicación nunca ha sida conectada, este directorio no existirá, por lo que por SSH creamos el directorio con otro directorio dentro llamado “data”, que utilizaremos para almacenar las credenciales de inicio de sesión. Por último, antes de finalizar este proceso, generamos un archivo JSON que almacenaremos en local con los datos que a introducido el usuario, de esta manera la próxima vez que entremos en la aplicación, antes de pedirlos, comprobara si existen en este fichero, en caso de que sí, se saltará este paso y leerá el JSON para tener las credenciales y poder establecer la comunicación.

Ya tenemos la conexión establecida con el servidor, pero no tenemos un usuario creado, esto lo sabe por que al hacer SSH y hacer un “ls” en la carpeta “data” no existe un “users.json”, por lo que la aplicación nos pide que creemos uno nuevo.

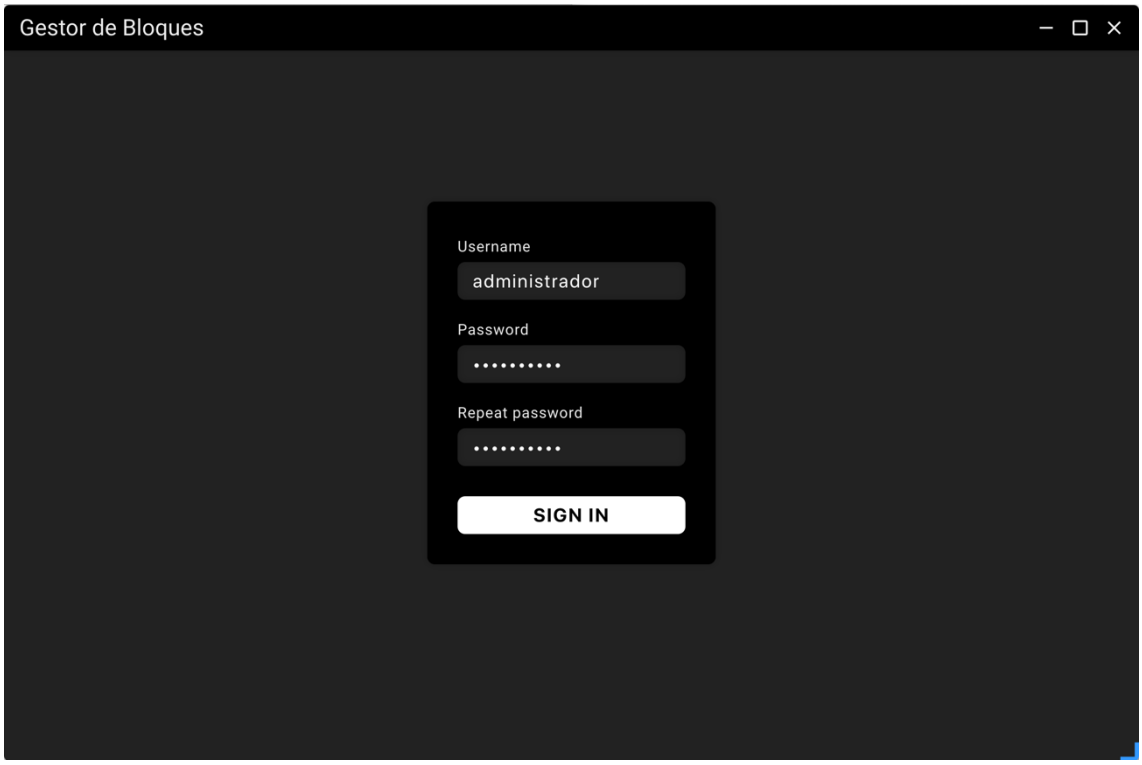


Ilustración 7: Crear usuario de la aplicación

Una vez el usuario pulsa en “SIGN IN” lo primero que se comprueba es que ambas contraseñas introducidas sean idénticas, en caso afirmativo procedemos a generar el usuario. Generar un usuario no es más que la creación de un fichero JSON con los datos, como este archivo va a residir en el servidor, no podemos almacenar la contraseña en claro, por lo que la pasamos por una función *hash* segura. Para esto utilizamos la librería “bcrypt”, en concreto el método “hash”, el cual utiliza una función hash de implementación propia, a la que se indica el

número de rondas por las que será pasada la contraseña en un argumento, cada una de ellas con un *salt* diferente. Una vez tenemos el *hash* de la contraseña, generamos un fichero “users.json” donde almacenamos todos estos datos y lo subimos al servidor al directorio “data”.

El propósito de este TFG no es crear una aplicación totalmente segura, y por esto no se ha profundizado más en la seguridad de las credenciales. Por la naturaleza de la aplicación, no es posible acceder al proceso de autenticación desde el exterior ya que la aplicación reside en local en tu ordenador, pero, como la carpeta “data” reside en el servidor, y ésta contiene información confidencial, se recomienda al usuario no dejar accesible esta carpeta vía petición HTTP/HTTPS. Este tipo de reglas en servidores web como Apache, se pueden especificar en el fichero “.htaccess”. Para futuras actualizaciones de la aplicación sería interesante reforzar la aplicación para evitar posibles ataques y manipulaciones no deseadas de las páginas web generadas.

7.7 Login

Para poder utilizar la aplicación, es necesario pasar por un proceso de autenticación y autorización, para evitar que otra persona que acceda a tu ordenador pueda modificar la web. También este proceso será importante para posteriores características de esta aplicación, como la diferenciación entre *roles* de usuarios y los permisos que tienen cada uno de ellos, mejoras no implementadas en este TFG.

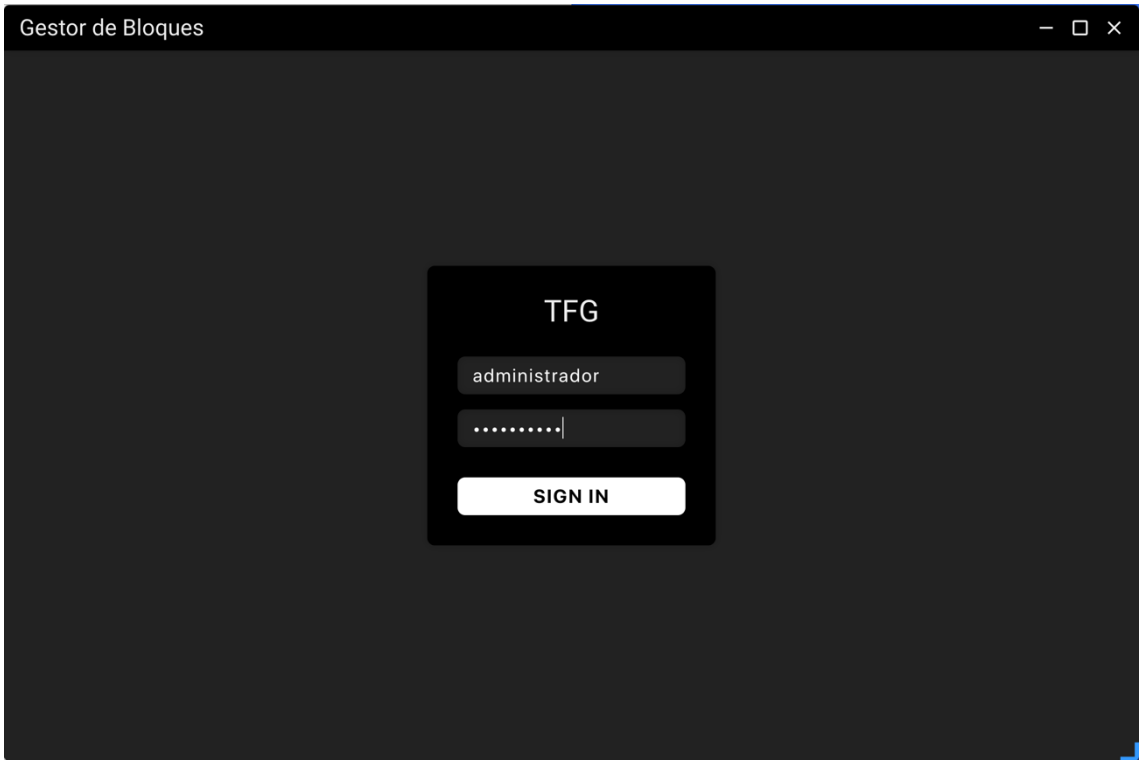


Ilustración 8: Proceso de Login

La aplicación descarga del servidor, cada vez que hacemos *login*, la última versión del fichero “users.json”. Una vez tenemos en local el fichero, lo pasamos a un objeto de JavaScript y comparamos los campos existentes con el que ha introducido el usuario, como la contraseña no la tenemos en claro, utilizamos el método “compare” de la librería “bcrypt”, el cual vuelve a pasar la contraseña en claro que ha puesto el usuario por la función *hash* y determina si es correcta.

Una vez el *Back-End* comunica al *Front-End* que el proceso a sido un éxito, este procede a ocultar la caja de *login* y cargar en el panel los módulos de la aplicación como hemos explicado anteriormente.

7.8 Tema oscuro y tema claro

Siguiendo la tendencia que siguen actualmente los frontales web, las aplicaciones móviles y las aplicaciones de escritorio, GdB cuenta con soporte para tema oscuro (el visto hasta ahora) y para tema claro.

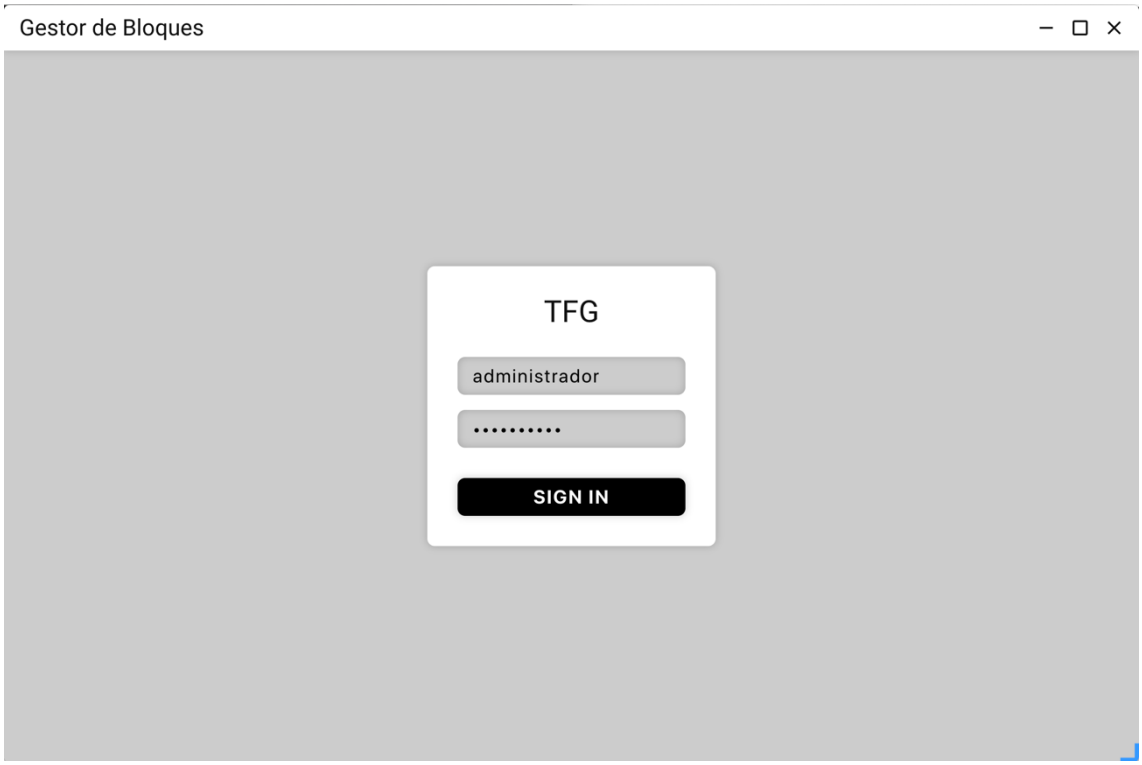


Ilustración 9: Interfaz con tema claro

Para poder desarrollar una sola interfaz independiente de los colores, he desarrollado todas las reglas de CSS referidas a colores, en vez de valores fijos, nombres de variables referidas a niveles de oscuridad. En concreto la interfaz se mueve en siete tonalidades, es decir, siete niveles de colores, que por detrás estos pueden ser cuales quiera siguiendo una escala en la misma gama.

Para decidir que gama es la que se aplica a la interfaz, las variables de CSS que contienen los colores son aplicadas para diferentes clases del elemento “body” del documento HTML. Si el “body” no tiene una clase definida, por defecto la

interfaz es clara, sin embargo, cuando le añadimos la clase “invert” pasa a ser negra, esto ocurre porque sobrescribimos las variables con esos colores y automáticamente todos los elementos de la interfaz cambian.

```
body {
    --level-0: #fff;
    --level-1: #ccc;
    --level-2: #aaa;
    --level-3: #888;
    --level-4: #666;
    --level-5: #444;
    --level-6: #222;
    --level-7: #000;
    --strong-color: #eee;
    --medium-color: #ddd;
    --smooth-color: #ccc;
    --contra-color: #aaa;
    --font-color: #111;
    --font-color-opposite: #eee;
    --logo-lighter: #333;
    --logo-darker: #111;
}
```

```
body.invert {
    --level-0: #000;
    --level-1: #222;
    --level-2: #444;
    --level-3: #666;
    --level-4: #888;
    --level-5: #aaa;
    --level-6: #ccc;
    --level-7: #fff;
    --strong-color: #111;
    --medium-color: #222;
    --smooth-color: #333;
    --contra-color: #555;
    --font-color: #eee;
    --font-color-opposite: #111;
    --logo-lighter: #eee;
    --logo-darker: #ccc;
}
```

Ilustración 10: Definición de colores tema claro y oscuro

7.9 Módulo editor

Un gestor de contenido ha de tener un editor de texto que brinde a sus usuarios una interfaz sencilla donde escribir. El módulo que se encarga de esto recibe el nombre de “editor”, ha sido diseñado siguiendo una estructura de bloques, de aquí el nombre de “gestor de bloques”, en el que cada elemento a insertar de contenido es un bloque. Cada uno de estos bloques consta de dos elementos, el texto del bloque y su tipo.

Ilustración 11: Bloque de ejemplo

Un post de un blog en este gestor de contenido no es más que una sucesión de bloques, en el que cada uno de ellos cuenta con su espacio donde escribir el texto y un selector en la parte inferior derecha que nos permite definir si ese texto es un párrafo, título, subtítulo o una cabecera. Los botones que encontramos en la parte inferior izquierda de cada uno de estos bloques nos permiten:

- El botón más, nos permite añadir un nuevo bloque que se insertará justo debajo del bloque desde donde estemos pulsando el botón.
- El botón menos, nos permite eliminar el bloque en el que nos encontramos en este momento.

- El botón flecha hacia arriba, permite mover el bloque una posición hacia arriba.
- El botón flecha hacia abajo, permite mover el bloque una posición hacia abajo.

7.9.1 Bloques

Cada uno de estos bloques en HTML es un “section” con la clase “elementEditor” que nos permite después, desde el JavaScript, ir recorriendo cada uno de estos para extraer el contenido. A su vez, el bloque en el que nos encontramos en un momento dado recibe la clase “active”, la cual remarca el bloque para que el usuario sepa donde está, despliega el menú inferior con los botones y permite al usuario editar el texto. Dentro de este “section” encontramos dos partes diferentes.

La parte del bloque que permite al usuario introducir texto, es un “div” el cual mediante el atributo “data-type” especifica que tipo de elemento contiene, en la ilustración 11 vemos como se trata de un bloque de tipo texto, por lo tanto, el “data-type” de este bloque será “text” y el elemento HTML que tenga dentro será un “p”. Para que el usuario pueda insertar texto, al elemento HTML se le especifica el atributo “contenteditable=true”, esto permite que un usuario desde la interfaz pueda insertar texto desde la interfaz, y este pase a estar dentro del elemento HTML que recibió esa propiedad.

La segunda parte del bloque es otro “div” el cual contiene los iconos de los botones con sus respectivos *id* para poder ser tratados los eventos desde el JavaScript y de un “select” para poder cambiar el tipo de bloque que queremos.

7.9.2 Interfaz del módulo

Cuando vamos a crear un nuevo artículo para el *blog*, entramos en el módulo editor del panel, por defecto aparece solo un bloque de tipo título, es decir, un “h1”. Este bloque es el único que la aplicación no te permite cambiar de tipo, ni de ser movido de posición con respeto a otros bloques, esto se debe a que, por buenas prácticas a la hora de generar una web, todas han de contener un único “h1” y este ha de estar posicionado al principio, ya que los motores de búsqueda le dan mucha relevancia a su contenido de cara a indexar resultados. Otros bloques posteriores tampoco permiten el cambiar el tipo a título, para evitar el que haya más de un “h1” en la página.

Para una mayor rapidez a la hora de escribir, además de los botones disponibles en cada uno de los bloques, creamos eventos que se lanzan con pulsaciones en determinadas teclas del teclado, es el caso por ejemplo del *enter*, el cual tiene el mismo comportamiento que el botón más, las flechas del teclado nos permiten de una manera rápida desplazarnos entre los diferentes bloques.

Cuando añadimos un nuevo bloque, lo que hacemos es generar el HTML del bloque desde el JavaScript e insertarlo a continuación del bloque desde el que esta siendo añadido. Lo mismo hacemos cuando lo eliminamos o lo

desplazamos entre bloques, todo son eventos de JavaScript que alteran el DOM para cumplir la funcionalidad de cada uno de los botones.

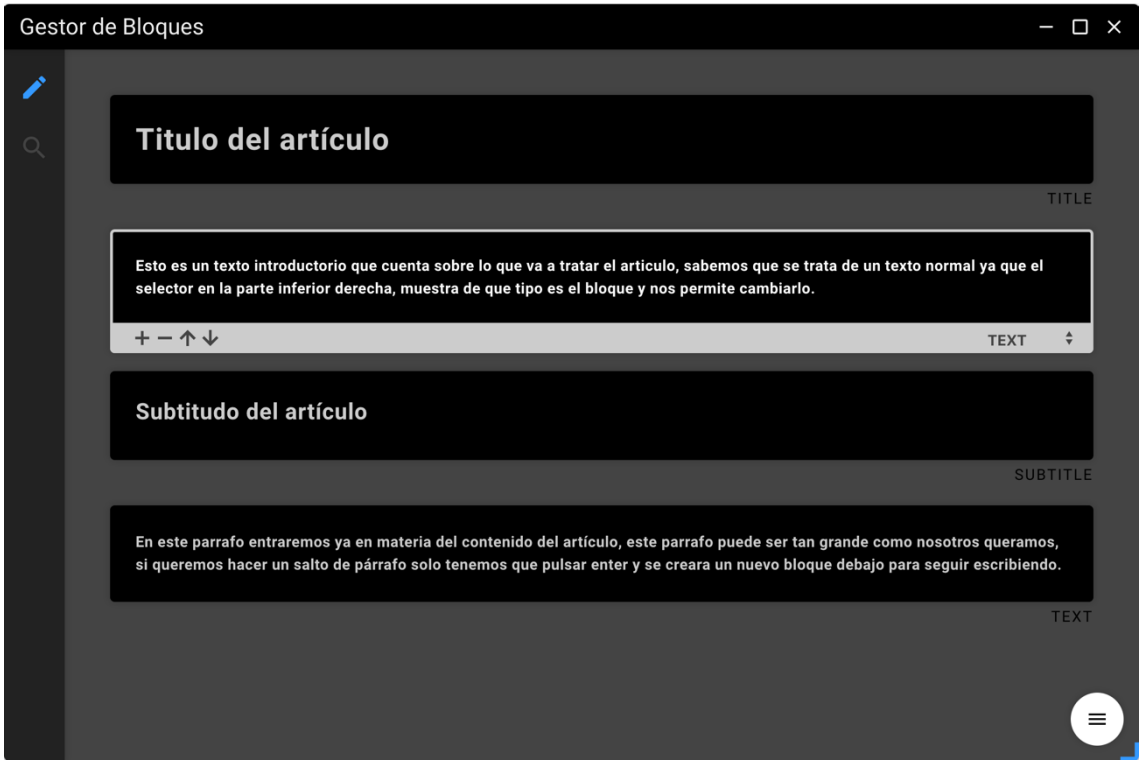


Ilustración 12: Interfaz del editor

En la parte inferior derecha, encontramos un botón el cual cuando pulsamos se despliegan otros dos, con uno de ellos podernos guardar el artículo y con el otro accederemos a los campos que tenemos que rellenar de metadatos del artículo.

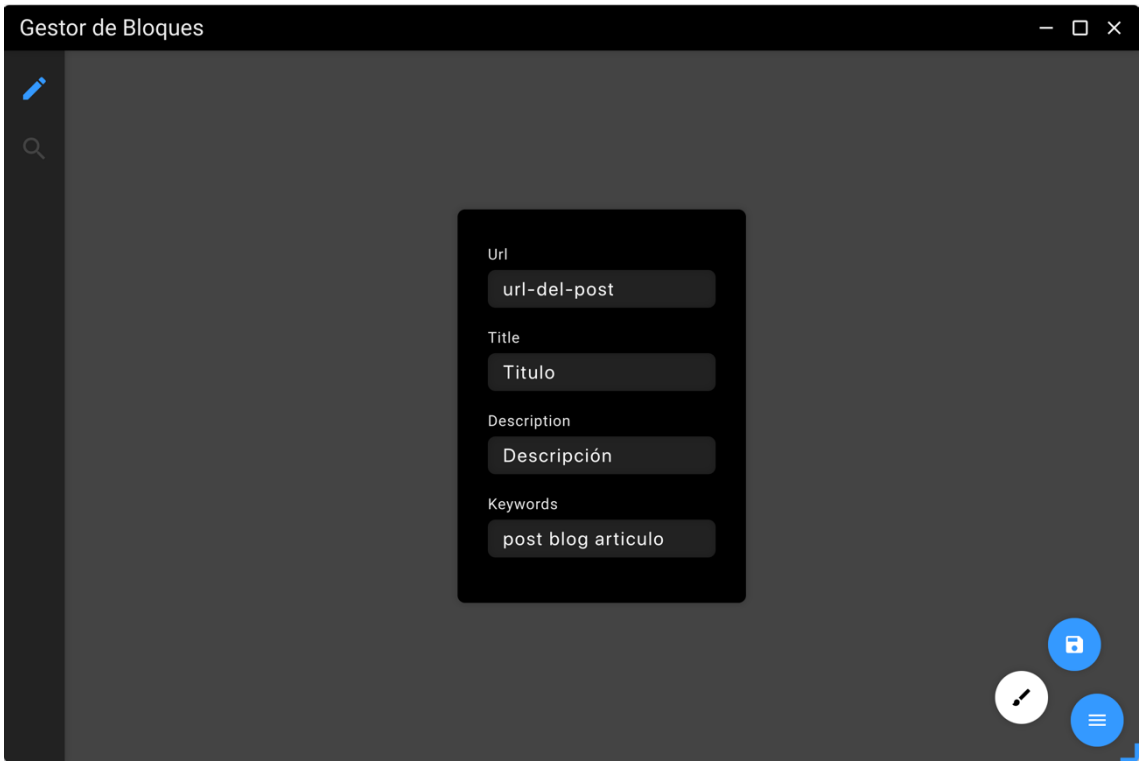


Ilustración 13: Metadatos del artículo

7.9.3 Template

Como ya hemos comentado, uno de los objetivos es poder añadir este *blog* sobre páginas ya desarrolladas. Para que la exportación de los bloques se haga en un HTML adaptado a la página web sobre la que este *blog* irá, es tan fácil como añadir un *template* en el directorio destinado para ello dentro de la aplicación.

Un *template* se compone de tres SHTML (trozo de HTML que se inserta en algún lugar) distintos, y utiliza un sistema de marcado para un posterior reemplazo mediante la búsqueda de palabras clave que son sustituidas por contenido. La idea es dividir la estructura HTML de tu página web actual en dos partes, por un lado, todo el HTML que debe ir por encima del contenido del articulo que vas a publicar (menú de la web, cabecera…) y, por otro lado, el código HTML que deba ir debajo del artículo (pie de página, menú inferior…). El nombre que recibe el SHTML superior es “up.shtml” y el inferior “down.shtml”. El tercer SHTML necesario es el que define la estructura HTML en la que se va a mostrar cada una de las entradas a los artículos en la página principal del blog, desde la cual puedes acceder a cada uno de los artículos.

```
<article>
    <h3><a href="/blog/#url#">#title#</a></h3>
    <p>
        #description#
    </p>
    <a href="/blog/#url#">READ MORE</a>
    <span>// #date#</span>
</article>
```

Ilustración 14: SHTML entrada de blog

En la ilustración 14 se puede observar la estructura definida para la exportación de cada una de las entradas para que se vea como deseo en mi página web, donde “#url#” es sustituido por la URL que te lleva al post en cuestión, “#title#” por el título del artículo, “#description#” por la descripción general del artículo y por último “#date#” por a fecha en la que se escribió. Esto una vez en el servidor y con el CSS de mi web se vería de la siguiente forma.

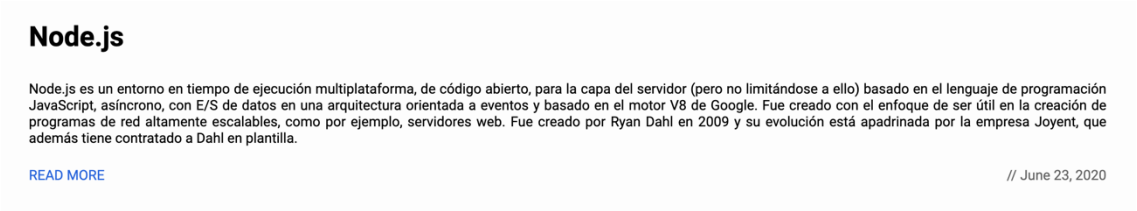


Ilustración 15: Ejemplo de una entrada

7.9.4 Exportación del artículo

Una vez el usuario ha terminado de editar el artículo, este solo tiene que darle al botón de guardar. Cuando esto pasa, por detrás en la aplicación genera dos archivos, un fichero JSON y otro HTML.

Como hemos comentado previamente, los bloques donde edita el usuario son un “div” que dentro contienen el elemento en cuestión (h1, h2, h3, p) dependiendo del tipo de bloque que sea, con la propiedad de que puedan ser editados, por lo que para generar el HTML final que se va a publicar en el servidor solo hemos de coger de la carpeta del *template* el fichero “up.shtml” y ese *string* concatenarlo con todos los bloques que haya hecho el usuario y por último esto concatenarlo con el fichero “down.shtml”. De esta manera tenemos el fichero completo listo para ser subido al servidor, aunque de momento este archivo solo existe en local en nuestro ordenador.

Al mismo tiempo, cuando guardamos, generamos un nuevo objeto que finalmente será guardado en un JSON, este objeto almacenará información relevante sobre el articulo que acabamos de crear, como por ejemplo la fecha en la que se creo, si se encuentra *online* u *offline* en el servidor, un *id* único para diferenciarlos entre cada uno de ellos y por último un objeto dentro de este y un *array* de objetos. El objeto “metas” contiene todos los metadatos que ha puesto el usuario sobre el artículo. El *array* de objetos “blocks”, es el que contiene todos

los bloques del editor, cada uno de estos bloques es un objeto con dos propiedades, el tipo del bloque y su contenido.

Este JSON que hemos generado, nos valdrá a posteriori para obtener información importante sobre cada uno de los HTML a la hora de hacer sincronizaciones con el servidor o de reconstruir el editor de artículos con el contenido del artículo, cuando un usuario desea hacer alguna modificación sobre uno generado en otro momento.

7.10 Módulo finder

Este módulo nos permite visualizar las páginas existentes de nuestro *blog*, con la posibilidad de modificarlas, eliminarlas o establecer si en la próxima sincronización han de estar *online* en el servidor web, o en caso contrario establecemos que no se sincronicen y existan solo en local en nuestro ordenador.

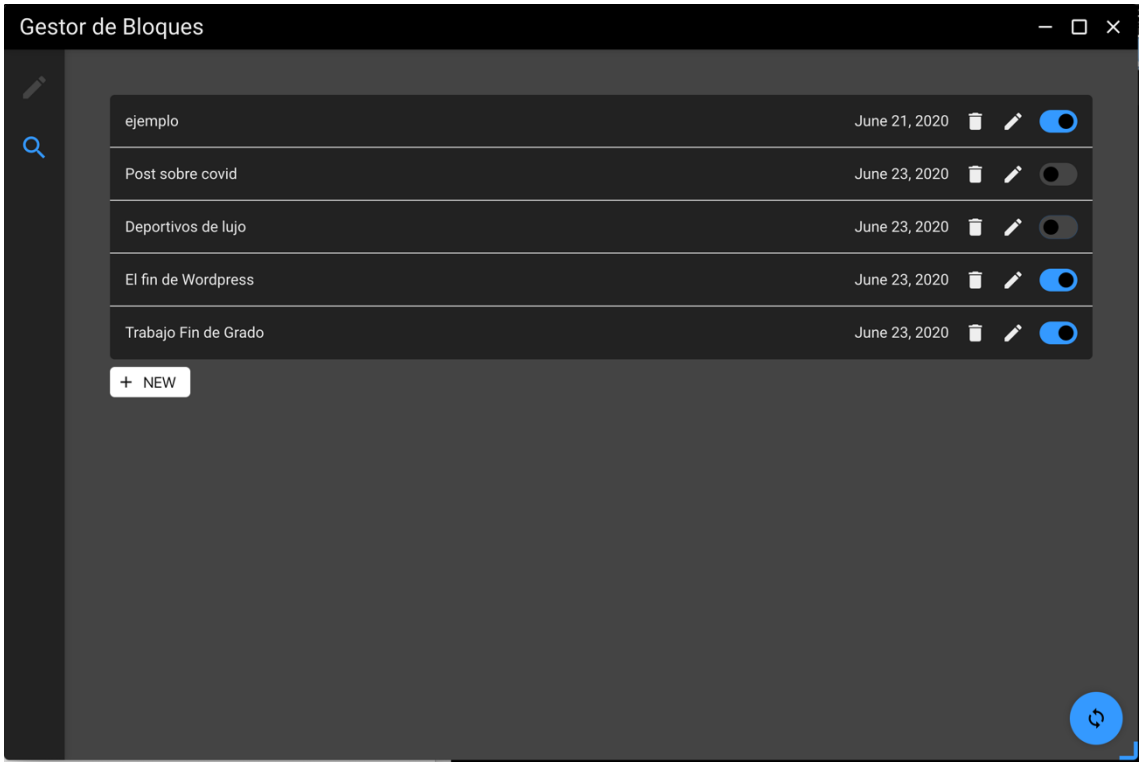


Ilustración 16: Módulo finder

Cuando entramos en este módulo, el código hace un escáner del directorio donde se encuentran los ficheros JSON que se generan cada vez que creamos una nueva página, por cada uno de estos ficheros, añade una nueva entrada en la interfaz con los datos de la página en cuestión que son obtenidos de este mismo fichero JSON, en concreto el nombre de la URL y la fecha en la que la página ha sido creada.

7.10.1 Eventos sobre páginas

Sobre cada una de estas entradas, es decir, por cada una de estas páginas que han sido previamente creadas desde el editor, tenemos tres opciones posibles.

Si el usuario desea eliminar una página que no va a necesitar más, tiene el botón de la papelera. Cuando se pulsa sobre él, la aplicación busca el fichero JSON y HTML correspondientes a esta entrada y son eliminados, a su vez elimina del DOM la entrada en la interfaz.

Si queremos añadir nuevo contenido o simplemente editar el ya existente en una página que ya está creada, tenemos el botón de editar. Editar una página consiste en mandar al usuario al módulo de editar, pero previamente estableciendo en una *cookie* determinada para ello, el identificador de la página que se quiere editar. Cuando el módulo editor carga, comprueba si esta *cookie* tiene algo, en caso afirmativo, en vez de cargar de cero, utiliza el fichero JSON de la página a editar y se reconstruyen sobre la interfaz los bloques existentes en este artículo.

El *switch* azul y negro que apreciamos en la ilustración 16, es utilizado para marcar si una página queremos que sea sincronizada con el servidor en la próxima sincronización, en caso de estar desactivada esta no será subida y si ya lo estaba, será eliminada del servidor. Para poder almacenar la información de qué página hay que subir y cuál no, utilizamos también su fichero JSON en el que una propiedad *booleana* es modificada entre *true* y *false*, cada vez que pulsamos el *switch*.

Por último, encontramos el botón “new”, este nos permitirá crear páginas nuevas desde cero. Si editamos una página en el editor, por mucho que nos movamos a otro módulo y volvamos, el editor seguirá teniendo el contenido que estabas escribiendo, para restablecer el editor y empezar una nueva página es necesario acceder desde este botón dentro del módulo finder.

7.10.2 Sincronización

La parte más importante de la aplicación es la sincronización entre los ficheros creados por el gestor de contenido en el ordenador personal del usuario y el servidor web. Al igual que el resto de las comunicaciones con el servidor de la aplicación, la sincronización se lleva a cabo por instrucciones SHH.

Para iniciar la sincronización hay que pulsar sobre el botón de sincronizar situado en la parte inferior derecha de la ilustración 16. Se recorre cada una de las páginas existentes en el gestor, si estas tienen la propiedad “online” con el valor *true* se procede a subir el fichero HTML correspondiente al servidor. Esta subida solo se produce tras comprobar que el archivo no se encuentra ya en el servidor, esto lo sabemos por otra propiedad distinta del JSON que indica si ha sido subida al servidor en algún momento, o si se encuentra en el servidor y comparando la fecha en la que se subió con la fecha de la última modificación del artículo, esta

última es mas reciente que la anterior. Todos los artículos que estén como *online* los almacenamos en un *array* para la construcción de la página principal del *blog*.

Cuando un artículo este marcado como no sincronización, no vale con no hacer nada ya que puede que este ya hubiese sido sincronizado y se encuentre *online*, por lo tanto, si el JSON indica que el artículo está en el servidor, mediante SSH eliminaremos el fichero en el servidor.

A la hora de sincronizar no vale solo con subir los ficheros de cada uno de los artículos, si no que e necesario generar una página *index* que muestre las entradas del *blog* y se pueda acceder a cada uno de los artículos. Cada vez que sincronizamos, la página principal del *blog* que existe en ese momento es eliminada y se genera una nueva de cero. Por cada uno de los artículos existentes se utiliza del *template* el “entry.shtml” como explicamos ya en la sección 7.9.3, y por encima y por debajo el resto del *template* de la web. Una vez este fichero HTML es creado, al igual que los otros, es subido al servidor, quedando el *blog* completamente actualizado.

8 Mejoras futuras de la aplicación

Gestor de bloques es una pequeña demo funcional de un gestor de contenido más potente que seguirá en desarrollo a posteriori de este trabajo de fin de grado. Como toda demo y por limitaciones de tiempo para un TFG hay una serie de características no implementadas en esta demo pero que merecen la pena mencionar.

8.1 Más tipos de bloques

Como hemos comentado, esta demo solo contiene cuatro tipos distintos de bloques: título, subtitulo, cabecera y texto. Sin embargo, a la hora de que los usuarios puedan escribir un artículo hay que proveerles de algunas herramientas más, como podrían ser los bloques de imágenes o de códigos fuente, si el *blog* es de un desarrollador.

Un bloque de imagen es un poco más complejo que cualquier bloque que sea de tipo de texto por dos motivos, la previsualización de la imagen en el bloque una vez insertada por el usuario y por la funcionalidad de como insertar esa imagen. Para solventar esta segunda lo más profesional posible, habría que maquetar un input de tipo “file” con una sección amplia que la que también admitiese el arrastrar un fichero dentro (*drag and drop*), Electron incorpora en su API una funcionalidad específica para esto.

8.2 Módulo productos

Un gran porcentaje de los sitios web en la actualidad son comercios electrónicos, pequeñas empresas que venden sus productos a través de una página web. Uno de los objetivos con este gestor en un futuro será la incorporación de un nuevo módulo que permita crear un catálogo de productos, esta sería la forma de rentabilizar la aplicación, proporcionarles a los usuarios una pasarela de pago desde el gestor que es propia y cobrar una comisión por cada venta que se realice en páginas web creadas con mi aplicación.

8.3 Modo offline y sincronización asíncrona

Una de las ventajas de ser una aplicación de escritorio es que se podría implementar una modalidad para hacer uso de ella sin necesidad de tener una conexión a internet. Esto se podría implementar utilizando el último fichero “users.json” descargado del servidor para el *login*, todos los cambios sobre las páginas quedarían registrado sobre un JSON con el historial de cambios que posteriormente podría ser contrastado con el servidor y posibles modificaciones que hayan hecho otros usuarios para solucionar posibles conflictos. En el momento en el que el usuario volviese a tener conexión se sincronizaría automáticamente con el servidor todos os cambios realizados durante el uso *offline*.

Actualmente durante el proceso de sincronización la aplicación queda congelada mientras se realiza este proceso, si volumen de artículos es muy grande, este proceso puede tardar varios minutos. Es interesante que la ejecución de este proceso se haga de manera asíncrona con respecto del resto de la aplicación, de esta forma podríamos seguir usándola mientras sincroniza.

9 Pruebas de rendimiento

Una vez la aplicación estuvo totalmente funcional, me pareció interesante hacer una comparativa entre una página de mi *blog* generada con mi gestor de contenido y el mismo artículo hecho en un WordPress recién instalado, para obtener los tiempos de procesamiento a nivel de servidor, carga y ejecución de la página. Esto me permite sacar conclusiones sobre cual de los dos gestores obtiene un mayor rendimiento.

En la comparativa hay que tener en cuenta que “le estoy dando ventaja” a WordPrees, ya que éste se encuentra sin ningún *plugin* instalado. Para poder realizar la comparativa en igualdad de condiciones, en cuanto a SEO, sería necesario como mínimo instalar el *plugin* de “Yoast”. Para replicar la interfaz de mi página web exactamente en WordPress necesitaría cargar *plugins* adicionales y personalizaciones que sumarían más tiempo aún a este proceso de carga.

Para realizar las pruebas he utilizado dos servidores iguales con el proveedor Arsys, en concreto dos servidores virtuales CentOS 8 con Plesk instalado, que no es más que un panel amigable para manejar por debajo Apache. En lo que a la resolución DNS se refiere, he forzado al ordenador a utilizar un servicio de resolución DNS 8.8.8.8 y 8.4.4.8, es decir, los DNS de Google, que *cachean* la resolución del DNS *lookup*, haciendo que estos tiempos sean nulos (0 ms) y por ello descartando esta variable de los resultados obtenidos.

Para poder registrar el tiempo empleado para procesar cada recurso cargado en la página web, he utilizado Chrome DevTools en una pestaña de incógnito, así me aseguro de que en la cache de mi navegador no hay recursos de las webs y se hace una petición desde cero [23].

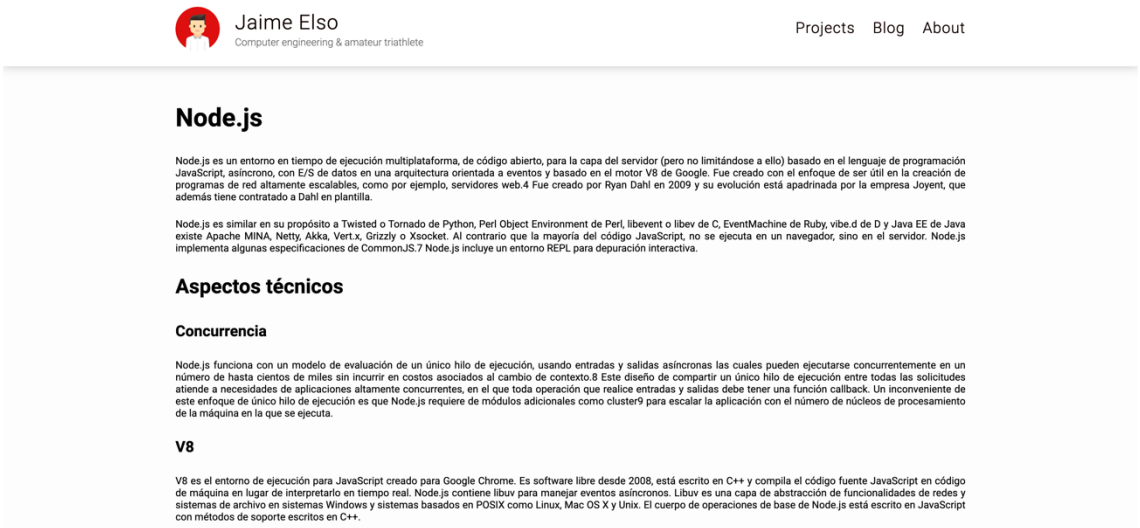


Ilustración 17: Articulo en mi web

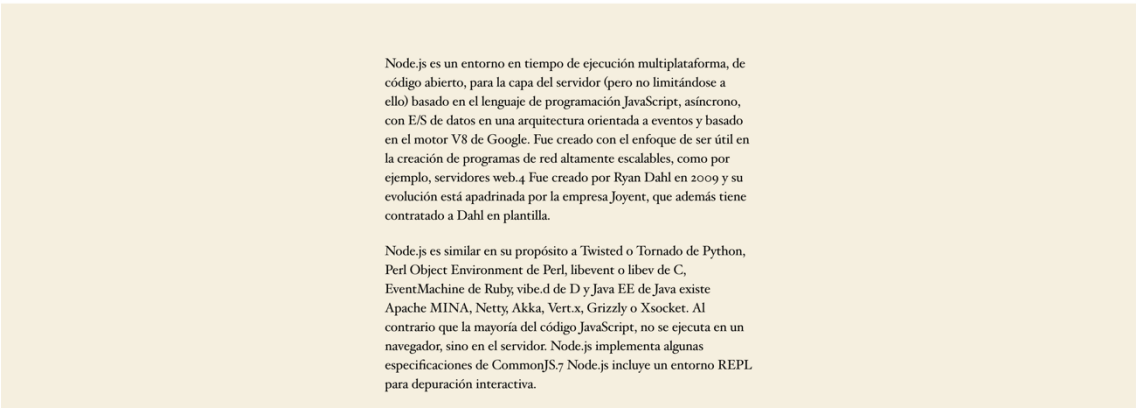


Ilustración 18: Articulo en WordPress

En las ilustraciones 17 y 18 podemos ver el mismo post con el mismo texto. La ilustración 17 es mi página web generada con el gestor desarrollado en este TFG. La ilustración 18 es una página generada por WordPress con el mismo contenido que el artículo de mi página web.

Cuando utilizamos la pestaña Network de las Chrome DevTools hay tres mediciones de tiempo distintas que podemos sacar y que es importante diferenciar:

- *DOM Content Loaded* representa el tiempo que tarda la página web desde que el navegador realiza la petición de la URL hasta que el documento HTML es cargado por completo en el navegador e interpretado, este tiempo se toma antes de solicitar recursos extra que necesite como imágenes u hojas de estilo [24].
- *Load* es el tiempo de *DOM Content Loaded* añadiendo también el tiempo empleado en solicitar y descargar los recursos adicionales que están solicitados en el HTML.
- *Finish* marca el tiempo en el que se ha terminado de representar gráficamente la página en el navegador.

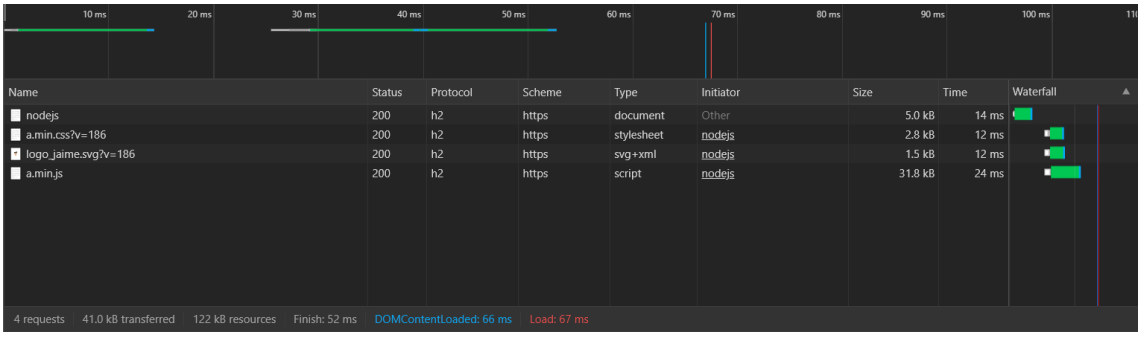


Ilustración 19: Tiempos de carga mi web

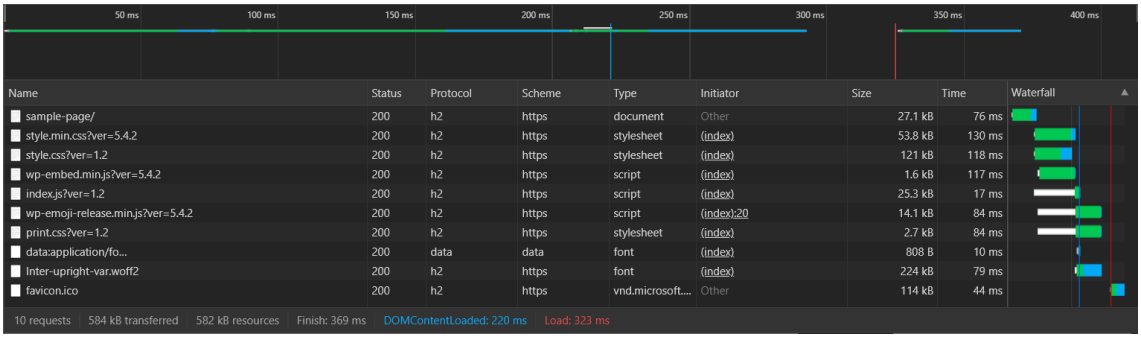


Ilustración 20: Tiempos de carga WordPress

La ilustración 19 corresponde a los resultados obtenidos de la página web generada con mi gestor, mientras que la ilustración 20 muestra los resultados de la página generada con WordPress. Las muestras tomadas corresponden a una medida tomada en un momento puntual, cada vez que haces una petición estos tiempos fluctúan, pero se mantiene la misma tendencia por lo que es suficientemente relevante como para poder comentar los resultados. Es apreciable la diferencia, en cuanto a tiempos, espacio en disco ocupado por los recursos que componen la página y número de peticiones, siendo que la página generada con mi gestor ha cargado en aproximadamente un sexto del tiempo y transfiriendo menos de un 10% de los datos que la página generada por WordPress.

En cuanto a número de peticiones se refiere, al emplear código hecho por y para esta página puedo reducir el número de archivos necesarios y optimizar su compresión lo que supone menos peticiones al servidor y menor tamaño. WordPress en cambio, carga un número de recursos definidos por el propio gestor y la plantilla empleada siendo esta una de las “plantillas más ligeras”. La diferencia es apreciable, pasando de diez peticiones a cuatro. Con esta optimización, se reduce en un 80% los datos transferidos, 122 kB de recursos en mi página web mientras que WordPress está transfiriendo 582 kB ya que, además, está transfiriendo más código que el necesario para mostrar esta página en específico. Esta diferencia en los recursos transferidos, a medida que se haga el sitio web más complejo, tomará más relevancia obteniendo así unos tiempos de respuesta mejores que un sitio WordPress convencional.

Obtener un *DOM Content Loaded*, *Load* y *Finish* lo más bajo posible, permite que los elementos de la página estén disponibles antes para el navegador. Desde el punto de vista de la interacción de un usuario humano con la página web, esto es importante porque, tal y como se discute a continuación, la experiencia del usuario es prácticamente inmediata (eventos controlados por JavaScript…). Por otro lado, de cara a los buscadores, es una ventaja competitiva ya que le tienen que dedicar menos tiempo a analizar la página y la puntuación obtenida es mejor.

Con el objetivo de poner en contexto los tiempos obtenidos respecto a la reacción de un ser humano ante un evento que le requiera una interacción es necesario disponer de datos sobre el tiempo de respuesta promedio de los humanos.

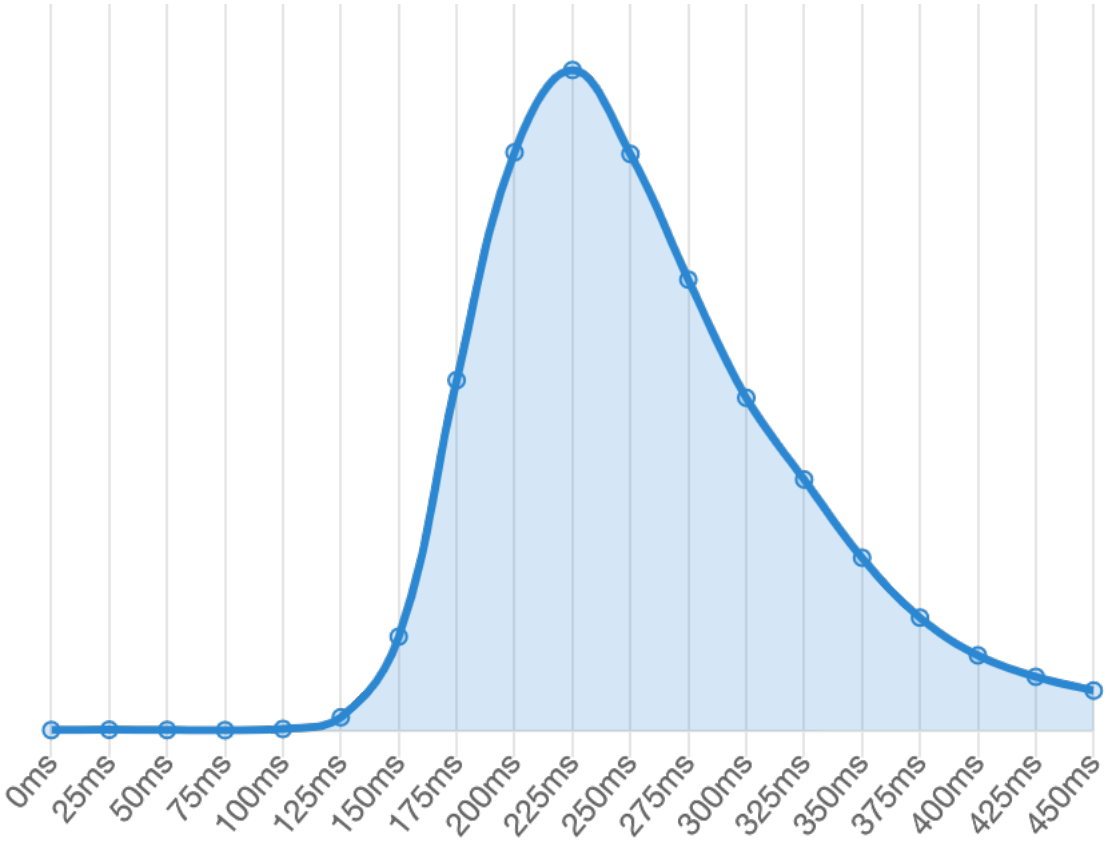


Ilustración 21: Tiempo de respuesta humano [25]

La gráfica que vemos en la ilustración 21, muestra el tiempo de respuesta de un ser humano ante un elemento que le requiera una interacción, ha sido obtenida tomando 81 Millones de muestras y la media es de 284 ms. Esto implica que las interacciones con mi página web no requieren esperas por parte del usuario, al haber obtenido un tiempo de 52 ms de representación gráfica (*Finish*), dicho usuario no deberá esperar a que cargue un contenido para poder interactuar con él. El estándar de video, 24 *frames* por segundo, nos demuestra que el ojo humano no consigue apreciar saltos por debajo de 42 ms, por lo cual el tiempo de carga entre que se termina de descargar el HTML (14 ms) y el tiempo que tarda en mostrar la totalidad de la página (52 ms) da una diferencia de 38 ms, que sería aproximadamente 0,9 *frames*, no es apreciable, dando la sensación de una carga "instantánea". Esto mismo en la carga de WordPress si es apreciable, ya que el tiempo entre que se descarga el HTML (76 ms) y el *Finish* (369 ms) da una diferencia de 293 ms que serían aproximadamente siete *frames*, por lo que el ojo humano notaría la diferencia. En una página sencilla estos datos en ambos casos son bajos, pero, a medida que aumente la complejidad, la diferencia de tiempos es mucho más relevante.

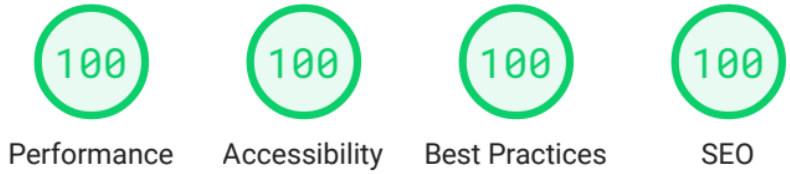


Ilustración 22: Lighthouse mi página

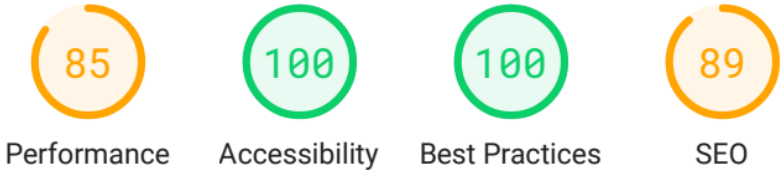


Ilustración 23: Lighthouse misma página WordPress

Estas mejoras de tiempo y optimización de recursos que hemos comentado se ven reflejadas en los informes generados por la herramienta Lighthouse en las ilustraciones 22 y 23, sus informes completos se encuentran en el Anexo C y el Anexo D de la memoria. Vemos como he obtenido en la página generada por mi gestor de contenido un cien de valoración, mientras que en la página generada por WordPress tan solo ha obtenido un ochenta y cinco.

Si miramos en el informe completo, vemos como Lighthouse está valorando positivamente en mi página web haber cargado únicamente las reglas de estilos CSS que son completamente necesarias para el documento, además de cargarlas de forma ordenada. Esto mismo aplica al JavaScript, siendo únicamente las funciones empleadas y no forzando a la página a tener que *reenderizarse* de nuevo durante la ejecución.

El resultado obtenido en el SEO se debe a que en mi página se ha generado con todo el contenido necesario a nivel de optimización para motores de búsqueda. Se han cargado los metadatos necesarios para la correcta identificación de la página, mientras que el WordPress al no haber sido configurado con *plugins* adicionales (para no empeorar aún más el resultado de WordPress en rendimiento) no dispone de estos elementos.

10. Conclusión

Este proyecto surgió para solventar una problemática que me encontré desarrollando mi página web y que ninguna de las aplicaciones o tecnologías que evalué me daban una solución que me pareciese indicada para el uso que yo iba a darle. Aprovechando la necesidad y la motivación por profundizar en esta área, nace la idea de crear esta aplicación de escritorio compatible con todos los sistemas operativos y todos los tipos de alojamiento web.

A lo largo del proceso del TFG he profundizado en tecnologías actuales como NodeJS y Electron, siguiendo la tendencia del mercado, portar tecnología web a aplicaciones de escritorio. He podido entender mejor el comportamiento de los navegadores, sobre todo de Chrome, navegador utilizado por Electron para visualizar la interfaz y motor de JavaScript empleado por NodeJS como intérprete. También elementos que influyen de cara a la valoración de una página web, especialmente a través de las herramientas Lighthouse y Chrome Dev Tools, a las que he sacado el máximo partido para analizar los resultados obtenidos y descubriendo en muchos casos elementos que se valoran y no son inicialmente tenidos en cuenta.

En un principio, pensar en desarrollar una herramienta que te permita crear y gestionar contenidos estáticos parece una idea simple. La realidad es muy diferente cuando entras en materia. Los elementos que componen el DOM [26], así como los recursos que lo modifican, generan una página web con una estructura compleja y hay que tenerla en cuenta a la hora de ser generada, hace que el desarrollo se complique. A todo esto, hay que sumarle los componentes de conexión con el servidor, módulo de edición, gestión de usuarios e interfaz gráfica que han resultado ser más complejos de lo que me imaginaba inicialmente.

Con mi TFG he confirmado mi hipótesis inicial, pese a que las pruebas realizadas en este proyecto no sean suficientes como para poder asegurarlo categóricamente, WordPress con la configuración por defecto y sin emplear *plugins* adicionales, a la hora de crear una página web sencilla, no ha resultado ser la aplicación más eficiente, generando una página mas lenta y con menos valoración SEO.

La aplicación que he desarrollado, en cambio, ha resultado ser un muy buen generador de páginas web, logrando un excelente rendimiento y un buen posicionamiento SEO en cada publicación. La carga de estas páginas es visualmente instantánea, 56 milisegundos, se reduce la transferencia de datos a los justos y necesarios para cada página y se reduce al máximo el número de peticiones al servidor (como se puede ver en el apartado 8).

Bibliografía

[1] Google, «Google Developers,» 25 OCTUBRE 2017. [En línea]. Available: https://developers.google.com/web/tools/lighthouse?hl=es. [Último acceso: 04 JULIO 2020].

[2] Google, «Google Developers,» 12 JUNIO 2020. [En línea]. Available: https://web.dev/performance-scoring/. [Último acceso: 4 JULIO 2020].

[3] Google, «WEB DEV,» 12 JUNIO 2020. [En línea]. Available: https://web.dev/performance-scoring/. [Último acceso: 5 JULIO 2020].

[4] J. GARCIA, «Departamento de Internet,» 9 MAYO 2011. [En línea]. Available: https://www.departamentodeinternet.com/que-es-un-cms-y-que-ventajas-tiene/. [Último acceso: 2 JUNIO 2020].

[5] M. W. DOCS, «Mozilla,» [En línea]. Available: https://developer.mozilla.org/es/docs/DOM. [Último acceso: 02 JULIO 2020].

[6] V. Ottervig, «enonic,» 3 JUNIO 2019. [En línea]. Available: https://enonic.com/blog/the-history-of-cms--what-has-happened#:~:text=The%20first%20CMS%2Dlike%20technologies,Pages%20(JSP)%20in%201999.. [Último acceso: 6 JUNIO 2020].

[7] SITECORE, «SITECORE,» [En línea]. Available: https://www.sitecore.com/knowledge-center/digital-marketing-resources/cms-vs-dxp-whats-the-difference. [Último acceso: 6 JUNIO 2020].

[8] NIBUSINESSINFO, «NIBUSINESSINFO,» [En línea]. Available: https://www.nibusinessinfo.co.uk/content/different-types-content-management-systems. [Último acceso: 6 JUNIO 2020].

[9] A. Brazell, WordPress Bible, 2010.

[10] Webtematica, «Webtematica,» [En línea]. Available: https://webtematica.com/historia-de-wordpress. [Último acceso: 5 JUNIO 2020].

[11] R. Ch., «Techjury,» 2 JUNIO 2020. [En línea]. Available: https://techjury.net/blog/percentage-of-wordpress-websites/#gref. [Último acceso: 5 JUNIO 2020].

[12] M. Rončević, «ARTBEES,» 29 MARZO 2018. [En línea]. Available: https://themes.artbees.net/blog/how-wordpress-works/. [Último acceso: 6 JUNIO 2020].

[13] P. Wałuszko, «Cyber Fores,» 10 JUNIO 2019. [En línea]. Available: https://cyberforces.com/en/wordpress-most-hacked-cms#:~:text=According%20to%20a%202017%20study,83%25%20of%20compromised%20CMS%20platforms.. [Último acceso: 5 JULIO 2020].

[14] WebAssembly, «WebAssembly,» [En línea]. Available: https://webassembly.org/. [Último acceso: 4 JUNIO 2020].

[15] Educacionit, «EDUCACIONIT,» 26 DICIEMBRE 2018. [En línea]. Available: https://blog.educacionit.com/2018/12/26/diferencia-entre-lenguajes-de-scripting-lenguajes-de-marcado-y-lenguajes-de-programacion/#:~:text=Los%20ejemplos%20de%20lenguajes%20de%20script%20utilizados%20com%C3%BAnmente%20incluyen%20JavaScript,principalmente%2. [Último acceso: 8 JUNIO 2020].

[16] UNIWEBSIDAD, «UNIWEBSIDAD,» [En línea]. Available: https://uniwebsidad.com/libros/javascript/capitulo-1/breve-historia. [Último acceso: 8 JUNIO 2020].

[17] S. Tilkov y S. Vinoski, «Node.js: Using JavaScript to Build High-Performance Network Programs,» *IEEE Internet Computing,* vol. 14, nº 6, p. 80, 20120.

[18] NODEJS, «NODEJS,» [En línea]. Available: https://nodejs.org/es/about/. [Último acceso: 8 JUNIO 2020].

[19] ElectronJS, «ElectronJS,» [En línea]. Available: https://www.electronjs.org/apps. [Último acceso: 5 JULIO 2020].

[20] Electron, «ELECTRONJS,» [En línea]. Available: https://www.electronjs.org/docs/tutorial/application-architecture. [Último acceso: 8 JUNIO 2020].

[21] ElectronJS, «ELECTRONJS,» [En línea]. Available: https://www.electronjs.org/docs/api/ipc-renderer. [Último acceso: 8 JUNIO 2020].

[22] ElectronJS, «ELECTRONJS,» [En línea]. Available: https://www.electronjs.org/docs/api/ipc-main. [Último acceso: 8 JUNIO 2020].

[23] Google, «Network Analysis Reference,» 4 JUNIO 2020. [En línea]. Available: https://developers.google.com/web/tools/chrome-devtools/network/reference. [Último acceso: 9 JULIO 2020].

[24] M. W. Docs, «Mozilla,» 30 ABRIL 2019. [En línea]. Available: https://developer.mozilla.org/es/docs/Web/Events/DOMContentLoaded. [Último acceso: 25 JUNIO 2020].

[25] Humanbenchmark, «Humanbenchmark,» [En línea]. Available: https://humanbenchmark.com/tests/reactiontime. [Último acceso: 29 Junio 2020].

[26] w3schools, «w3schools.com,» [En línea]. Available: https://www.w3schools.com/js/js_htmldom.asp. [Último acceso: 9 JULIO 2020].

Anexo A: Lighthouse WordPress vacío

https://florespanama.net/

Performance	Accessibility	Best Practices	SEO
94	100	100	90

Progressive Web App: PWA

0–49 50–89 90–100

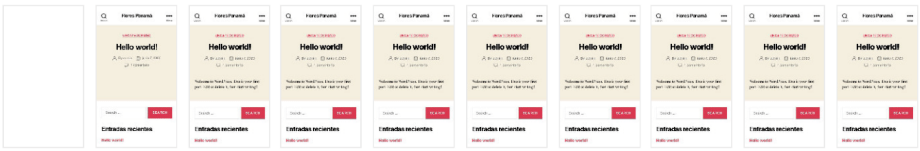
94

Performance

Metrics

First Contentful Paint	2.4 s	Time to Interactive	2.5 s
Speed Index	2.4 s	Total Blocking Time	40 ms
Largest Contentful Paint	2.7 s	Cumulative Layout Shift	0

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.



Opportunities — These suggestions can help your page load faster. They don't directly affect the Performance score.

Opportunity	Estimated Savings
▲ Eliminate render-blocking resources	1.09 s
▲ Enable text compression	0.9 s
■ Remove unused CSS	0.75 s
■ Minify CSS	0.15 s

Diagnostics — More information about the performance of your application. These numbers don't directly affect the Performance score.

- ▲ Serve static assets with an efficient cache policy — 7 resources found
- ● Avoid chaining critical requests — 5 chains found
- ● Keep request counts low and transfer sizes small — 8 requests • 459 KB
- ● Largest Contentful Paint element — 1 element found
- ● Avoid large layout shifts — 4 elements found

Passed audits (20)

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of

accessibility issues can be automatically detected so manual testing is also encouraged.

Additional items to manually check (10) — These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

Passed audits (22)

Not applicable (19)

Best Practices

Passed audits (14)

SEO

These checks ensure that your page is optimized for search engine results ranking. There are additional factors Lighthouse does not check that may affect your search ranking. Learn more.

Content Best Practices — Format your HTML in a way that enables crawlers to better understand your app's content.

▲ Document does not have a meta description

Additional items to manually check (1) — Run these additional validators on your site to check additional SEO best practices.

Passed audits (9)

Not applicable (3)

Progressive Web App

These checks validate the aspects of a Progressive Web App. Learn more.

Fast and reliable

● Page load is fast enough on mobile networks

▲ Current page does not respond with a 200 when offline

▲ `start_url` does not respond with a 200 when offline

No usable web app manifest found on page.

Installable

- Uses HTTPS
- Does not register a service worker that controls page and `start_url`
- Web app manifest does not meet the installability requirements
  Failures: No manifest was fetched.

PWA Optimized

- Redirects HTTP traffic to HTTPS
- Is not configured for a custom splash screen Failures: No manifest was fetched.
- Does not set a theme color for the address bar.
  Failures: No manifest was fetched, No `<meta name="theme-color">` tag found.
- Content is sized correctly for the viewport
- Has a `<meta name="viewport">` tag with `width` or `initial-scale`
- Contains some content when JavaScript is not available
- Does not provide a valid `apple-touch-icon`
- Manifest doesn't have a maskable icon No manifest was fetched

Additional items to manually check (3) — These checks are required by the baseline PWA Checklist but are not automatically checked by Lighthouse. They do not affect your score but it's important that you verify them manually.

Runtime Settings

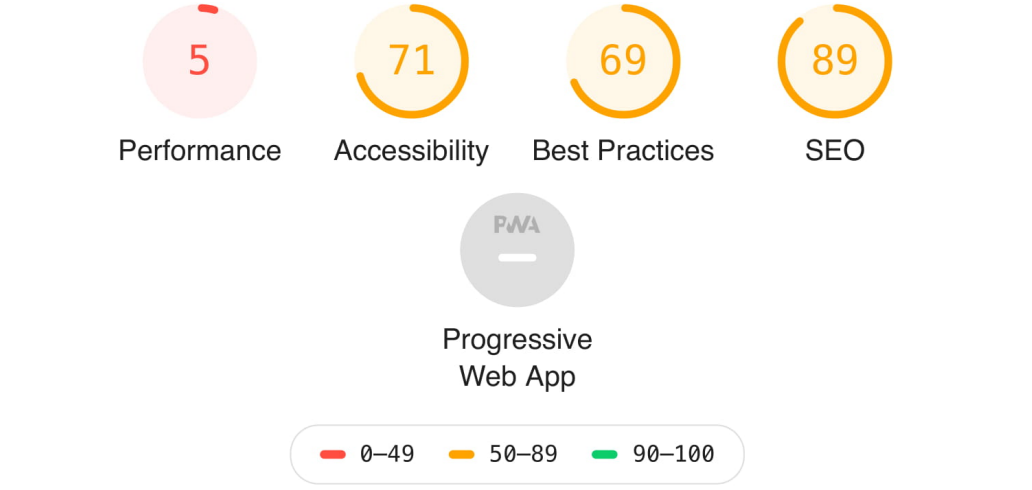
URL	https://florespanama.net/

Fetch Time	Jun 7, 2020, 12:33 PM GMT+2
Device	Emulated Moto G4
Network throttling	Unknown
CPU throttling	Unknown
Channel	lr
User agent (host)	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/83.0.4103.93 Safari/537.36
User agent (network)	Mozilla/5.0 (Linux; Android 6.0.1; Moto G (4)) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3963.0 Mobile Safari/537.36 Chrome-Lighthouse
CPU/Memory Power	640

Generated by **Lighthouse** 6.0.0 | File an issue

Anexo B: Lighthouse “techcrunch.com”

https://techcrunch.com/



Performance

Metrics			
▲ First Contentful Paint	5.0 s	▲ Time to Interactive	19.7 s
▲ Speed Index	13.3 s	▲ Total Blocking Time	4,640 ms
▲ Largest Contentful Paint	18.1 s	▲ Cumulative Layout Shift	0.996

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

Opportunities — These suggestions can help your page load faster. They don't directly affect the Performance score.

Opportunity	Estimated Savings
▲ Remove unused JavaScript	3.6 s
▲ Eliminate render-blocking resources	2.74 s
▲ Properly size images	1.2 s
■ Remove unused CSS	0.45 s
■ Enable text compression	0.15 s

Diagnostics — More information about the performance of your application. These numbers don't directly affect the Performance score.

- ▲ Ensure text remains visible during webfont load
- ▲ Reduce the impact of third-party code — Third-party code blocked the main thread for 710 ms
- ▲ Avoid `document.write()`
- ▲ Minimize main-thread work — 15.5 s
- ▲ Reduce JavaScript execution time — 12.1 s
- ▲ Serve static assets with an efficient cache policy — 65 resources found
- ■ Avoid enormous network payloads — Total size was 3,077 KB
- ● Avoid chaining critical requests — 21 chains found
- ● Keep request counts low and transfer sizes small — 123 requests • 3,077 KB
- ● Largest Contentful Paint element — 1 element found
- ● Avoid large layout shifts — 5 elements found

Passed audits (13)

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

ARIA — These are opportunities to improve the usage of ARIA in your application which may enhance the experience for users of assistive technology, like a screen reader.

- ▲ `[aria-*]` attributes do not have valid values

Contrast — These are opportunities to improve the legibility of your content.

- ▲ Background and foreground colors do not have a sufficient contrast ratio.

Names and labels — These are opportunities to improve the semantics of the controls in your application. This may enhance the experience for users of assistive technology, like a screen reader.

- ▲ Form elements do not have associated labels
- ▲ Links do not have a discernible name

Best practices — These items highlight common accessibility best practices.

- ▲ `[user-scalable="no"]` is used in the `<meta name="viewport">` element or the `[maximum-scale]` attribute is less than 5.

Additional items to manually check (10) — These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

Passed audits (16)

Not applicable (20)

Best Practices

Trust and Safety

- ▲ Links to cross-origin destinations are unsafe
- ▲ Includes front-end JavaScript libraries with known security vulnerabilities — 5 vulnerabilities detected

User Experience

- ▲ Displays images with inappropriate size

General

- ▲ Browser errors were logged to the console

Passed audits (10)

SEO

These checks ensure that your page is optimized for search engine results ranking. There are additional factors Lighthouse does not check that may affect your search ranking. Learn more.

Content Best Practices — Format your HTML in a way that enables crawlers to better understand your app's content.

▲ Links do not have descriptive text — 3 links found

Mobile Friendly — Make sure your pages are mobile friendly so users don't have to pinch or zoom in order to read the content pages. Learn more.

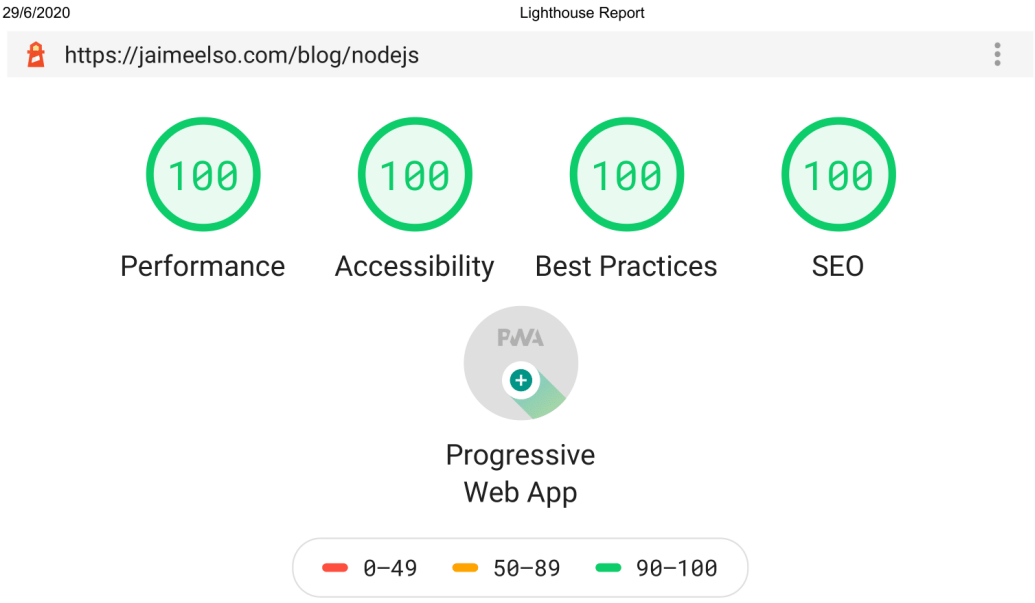
■ Tap targets are not sized appropriately — 76% appropriately sized tap targets

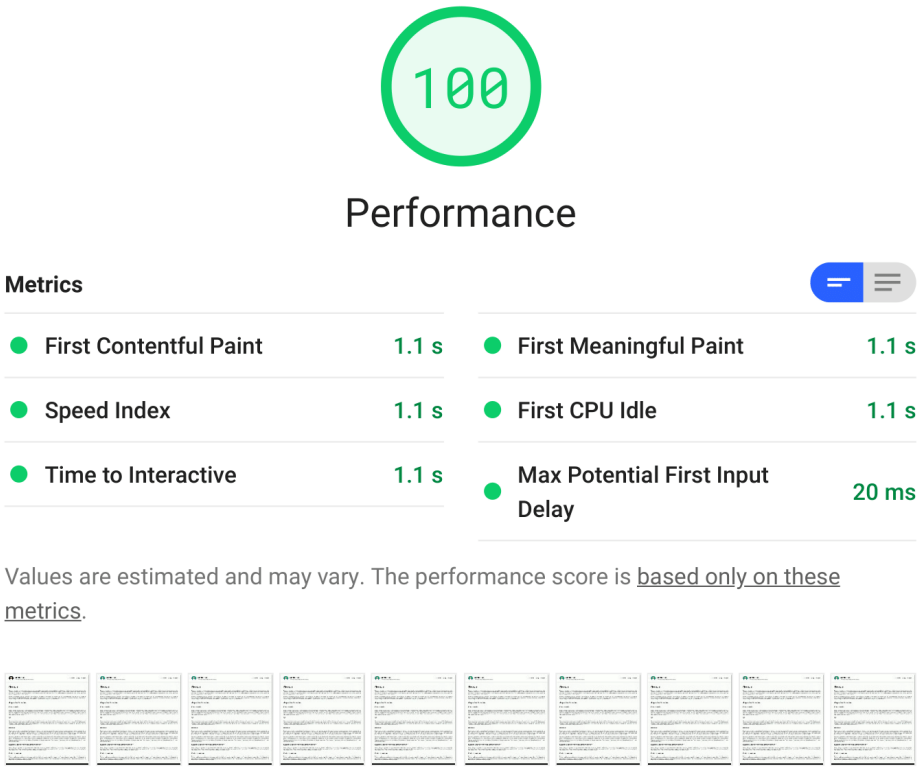
Additional items to manually check (1) — Run these additional validators on your site to check additional SEO best practices.

Passed audits (10)

Not applicable (1)

Anexo C: Lighthouse mi página





Diagnostics — More information about the performance of your application. These numbers don't directly affect the Performance score.

- Avoid chaining critical requests — 2 chains found
- Keep request counts low and transfer sizes small — 5 requests • 56 KB

Passed audits (22)

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

Additional items to manually check (11) — These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

Passed audits (18)

Not applicable (17)

Best Practices

Passed audits (15)

SEO

These checks ensure that your page is optimized for search engine results ranking. There are additional factors Lighthouse does not check that may affect your search ranking. Learn more.

Additional items to manually check (1) — Run these additional validators on your site to check additional SEO best practices.

Passed audits (10)

Not applicable (3)

Progressive Web App

These checks validate the aspects of a Progressive Web App. Learn more.

Fast and reliable

- Page load is fast enough on mobile networks
- Current page responds with a 200 when offline
- ▲ `start_url` does not respond with a 200 when offline
  The start_url did respond, but not via a service worker.

Installable

- Uses HTTPS
- Registers a service worker that controls page and `start_url`
- Web app manifest meets the installability requirements

PWA Optimized

- Redirects HTTP traffic to HTTPS
- Configured for a custom splash screen
- Sets a theme color for the address bar.
- Content is sized correctly for the viewport
- Has a `<meta name="viewport">` tag with `width` or `initial-scale`
- Contains some content when JavaScript is not available
- Provides a valid `apple-touch-icon`

Additional items to manually check (3) — These checks are required by the baseline PWA Checklist but are not automatically checked by Lighthouse. They do not affect your score but it's important that you verify them manually.

Runtime Settings

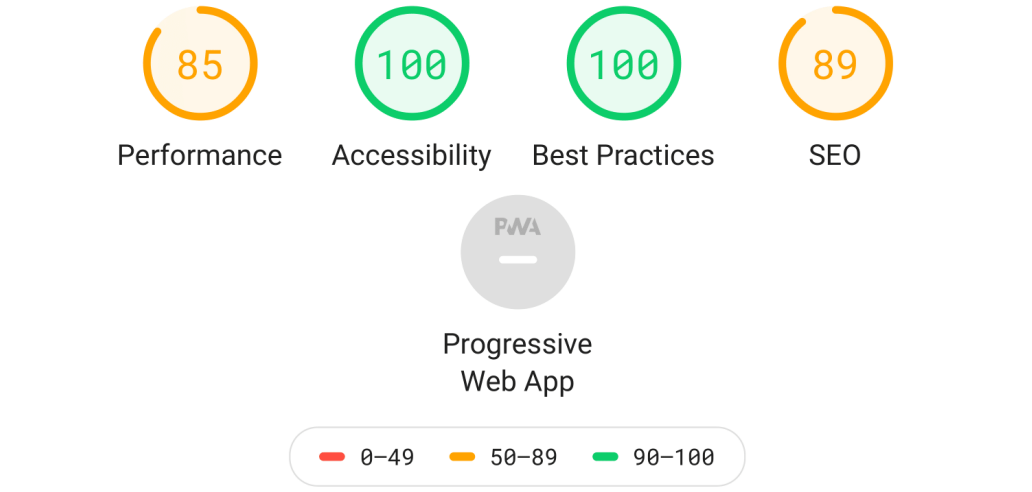
URL	https://jaimeelso.com/blog/nodejs
Fetch time	Jun 29, 2020, 6:14 PM GMT+2
Device	Emulated Desktop
Network throttling	150 ms TCP RTT, 1,638.4 Kbps throughput (Simulated)
CPU throttling	4x slowdown (Simulated)
User agent (host)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36
User agent (network)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3694.0 Safari/537.36 Chrome-Lighthouse
CPU/Memory Power	1656

Generated by **Lighthouse** 5.7.1 | File an issue

/*# sourceURL=third_party/lighthouse/report-assets/template.html */

Anexo D: Lighthouse misma página en WP

https://florespanama.net/index.php/sample-page/



Performance

Metrics

First Contentful Paint	2.7 s	First Meaningful Paint	4.1 s
Speed Index	2.7 s	First CPU Idle	4.1 s
Time to Interactive	4.1 s	Max Potential First Input Delay	50 ms

Values are estimated and may vary. The performance score is based only on these metrics.

Opportunities — These suggestions can help your page load faster. They don't directly affect the Performance score.

Opportunity	Estimated Savings
▲ Remove unused CSS	1.05 s
▲ Eliminate render-blocking resources	1.02 s
■ Minify CSS	0.15 s

Diagnostics — More information about the performance of your application. These numbers don't directly affect the Performance score.

- ▲ Serve static assets with an efficient cache policy — 7 resources found
- ● Avoid chaining critical requests — 5 chains found
- ● Keep request counts low and transfer sizes small — 10 requests • 719 KB

Passed audits (18)

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

Additional items to manually check (11) — These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an

accessibility review.

Passed audits (19)

Not applicable (16)

Best Practices

Passed audits (15)

SEO

These checks ensure that your page is optimized for search engine results ranking. There are additional factors Lighthouse does not check that may affect your search ranking. Learn more.

Content Best Practices — Format your HTML in a way that enables crawlers to better understand your app's content.

▲ Document does not have a meta description

Additional items to manually check (1) — Run these additional validators on your site to check additional SEO best practices.

Passed audits (8)

Not applicable (4)

Progressive Web App

These checks validate the aspects of a Progressive Web App. Learn more.

Fast and reliable

● Page load is fast enough on mobile networks

▲ Current page does not respond with a 200 when offline

▲ `start_url` does not respond with a 200 when offline

No usable web app manifest found on page.

Installable

● Uses HTTPS

- ▲ Does not register a service worker that controls page and `start_url`
- ▲ Web app manifest does not meet the installability requirements
  Failures: No manifest was fetched.

PWA Optimized

- ● Redirects HTTP traffic to HTTPS
- ▲ Is not configured for a custom splash screen Failures: No manifest was fetched.
- ▲ Does not set a theme color for the address bar.
  Failures: No manifest was fetched, No `<meta name="theme-color">` tag found.
- ● Content is sized correctly for the viewport
- ● Has a `<meta name="viewport">` tag with `width` or `initial-scale`
- ● Contains some content when JavaScript is not available
- ▲ Does not provide a valid `apple-touch-icon`

Additional items to manually check (3) — These checks are required by the baseline PWA Checklist but are not automatically checked by Lighthouse. They do not affect your score but it's important that you verify them manually.

Runtime Settings

URL	https://florespanama.net/index.php/sample-page/
Fetch time	Jun 29, 2020, 6:13 PM GMT+2
Device	Emulated Desktop
Network throttling	150 ms TCP RTT, 1,638.4 Kbps throughput (Simulated)
CPU throttling	4x slowdown (Simulated)

User agent (host)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36
User agent (network)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3694.0 Safari/537.36 Chrome-Lighthouse
CPU/Memory Power	1546

Generated by **Lighthouse** 5.7.1 | File an issue

/*# sourceURL=third_party/lighthouse/report-assets/template.html */