E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

Análisis del renderizado web y migración de SPA (Single Page Application) a SSG (Static Site Generation) en una empresa de servicios digitales

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Álvaro del Pazo Batista

Tutora: Edurne Barrenechea Tartas

Pamplona, 30 de mayo de 2022

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen de contenido

El presente Trabajo Fin de Grado, ha sido realizado en la empresa de desarrollo de software, Biko2 S.L. En este trabajo realizamos un estudio sobre el renderizado web y sus distintos tipos, haciendo hincapié en sus ventajas y desventajas según diversos criterios. En paralelo, también analizamos en qué situaciones conviene utilizar un renderizado u otro y con qué frameworks y herramientas. Finalmente, implementamos, como petición de un cliente, un caso práctico de migrado de una aplicación web de un tipo de renderizado a otro.

Palabras clave

Static-Site Generation, Server-Side Rendering, Client-Side Rendering, Content Delivery Network, Web estática, SEO, Single Page Application, Framework, Migración, Renderizar

Contenido

1. Introducción

El presente Trabajo fin de Grado (TFG), ha sido realizado en la empresa de desarrollo software, Biko2 S.L., empresa asentada en Pamplona, Navarra. Biko, forma parte del sector de consultoría tecnológica, dedicándose a un amplio espectro de áreas de trabajo, siendo la primaria el Desarrollo de Aplicaciones Web. No obstante, también ofrecen otros servicios como el de diseño de interfaces y experiencias de usuario, construcción de software ágil a través de metodologías de código limpio, entre otras, primando siempre la creación de un buen software de calidad.

Actualmente, cuenta con una sede central situada en el Polígono de Mutilva, Pamplona, donde se fundó y a medida que crecieron, expandieron su presencia en otras ciudades. En primer lugar, Madrid, donde poseen otra oficina, que es empleada en ocasiones para la realización de eventos multitudinarios, seminarios de formación o puntos de encuentro entre clientes y empresa. Por otro lado, en un sector más internacional, también tienen presencia en Lima, Perú con intenciones de seguir expandiéndose a medida que la empresa siga creciendo.

Desde el año 2000, con el despertar de la era digital moderna, las páginas web han tomado una parte fundamental en sentar las bases del Internet que conocemos hoy en día. Ya por aquel entonces se consideraba algo revolucionario, ahora tenemos el concepto de página web como algo muy común y algo a lo que la mayoría de la población accede en su día a día sin ser consciente de ello. Pese a que la mayoría de los usuarios conocen lo qué es una página web, no es tan común el conocer cómo se renderiza la misma en nuestro navegador, es decir, cómo todos esos datos que la contienen son obtenidos y cargados. Existen diversas formas que han ido surgiendo y evolucionando a lo largo de la historia de la navegación web, y el propósito de este TFG es realizar un análisis de ellas, evaluar su desempeño en distintas situaciones y cómo se llevan a cabo dichos renderizados con un viaje por distintos frameworks que permiten la realización de ellos. Además, se analizará qué método es mejor atendiendo a distintos criterios y situaciones posibles. Aplicaremos este estudio para migrar una aplicación web de un tipo de renderizado a otro para conseguir un determinado objetivo y estudiaremos un caso real de una migración de proyecto de la empresa Biko.

2. Renderizar

Previa a la inmersión dentro de este trabajo de investigación, es recomendable tener en cuenta unos conocimientos previos para poder sentar las bases del análisis. Antes que nada, estaría bien comenzar con qué definición le damos a la palabra *renderizado*, el arte de *renderizar*. Muchos de ustedes sabrán, que es una palabra utilizada en el mundo de la tecnología y posee varios significados según el área que estemos tratando. En pocas palabras, la palabra renderizar proviene del inglés "*render*" y hace alusión al proceso por el cual se genera una imagen a partir de un modelo 2D o 3D, con la ayuda de programas informáticos. Una vez entendido esto, pasamos a migrar este concepto al sector del Desarrollo Web. La lógica nos llevaría a pensar, que el renderizado web es el proceso por el cual se genera una web en un navegador gracias a la ayuda de programas informáticos. Este renderizado es capaz de llevarse a cabo gracias a la transformación de los archivos HTML, CSS y JavaScript en el dispositivo final del usuario o en el servidor [1].

2.1. El navegador y el renderizado

En primer lugar, para una mejor comprensión del renderizado, detallamos en qué ocasiones los navegadores reaccionan y renderizan una página. La primera de ellas sería con la carga inicial o recarga de esta. La segunda, sería a través de un cambio producido por JavaScript, abarcando todas las llamadas asíncronas que pueden ser realizadas por medio de técnicas como AJAX. Posteriormente, una vez se haya llevado a cabo la reacción en el navegador, éste comenzará a mostrar la página a través del método de renderizado establecido y acorde a distintos algoritmos de cálculo, que permiten estilar la página web ajustándose a las dimensiones del dispositivo del usuario final.

Los algoritmos de cálculo son generales, empezando por calcular los estilos de cada elemento del HTML, aplicando cada uno de ellos en cascada atendiendo siempre a las reglas que rigen en cada uno. Posteriormente, se representa cada elemento HTML en su posición, con su tamaño y ocupando el espacio que se le ha sido asignado. Una vez realizadas las dos fases anteriores, se realiza una tercera fase que es el rellenado en memoria de los píxeles con los textos, los colores, las imágenes y el resto de los componentes. Finalmente, se compone todo en sus respectivas capas y se plasma en pantalla respetando el orden de estas, para no alterar la visualización de la página original.

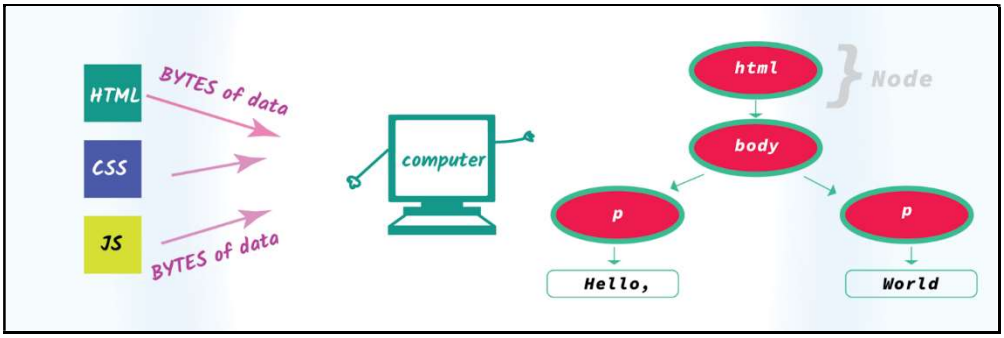

Fig. 1.1 - Reacción del navegador a una web

En la Fig. 1.1, visualizamos cómo se transforma los archivos HTML, CSS y JavaScript a componentes representables en el navegador. El navegador realiza un

proceso intermedio, en el cual genera en primer lugar el DOM y luego el CSSOM del proyecto en cuestión. Acto seguido, el DOM y CSSOM se unifican para crear el árbol de renderizado que vemos a la derecha en la figura previamente citada. Finalmente, se aplican los estilos CSS a los componentes del árbol de renderizado, para terminar, mostrando esos nodos en la pantalla, pudiendo así, observar la página web en nuestro navegador [2].

2.2 Tipos de renderizado

Una vez establecido cómo funciona un navegador y qué es el renderizado, analizaremos los distintos tipos de renderizados que existen. En líneas generales, podemos agruparlos principalmente en dos clases las cuales engloban a la gran mayoría: los renderizados por parte del servidor y los renderizados por parte del cliente, comúnmente conocidos como Server-Side Rendering y Client-Side Rendering respectivamente. A continuación, explicamos cada uno de ellos, analizando sus ventajas y desventajas en según qué condiciones y escenarios.

2.2.1 Server-Side Rendering

Comenzando con el Server-Side Rendering, debemos tener en cuenta que esta técnica se encarga de generar todo el HTML de la página del cliente en el servidor. Como ejemplo, tenemos la página web de Google, que sigue el tipo de renderizado por parte del servidor. Al solicitar la página, enviando la petición por parte del cliente al servidor, esta se pre-renderiza en el lado del servidor y nos envía el código HTML de la web que nuestro navegador plasmará y hará operativa.

Anteriormente se asociaba el SSR al lenguaje de PHP, cosa que ya no es necesariamente cierta. Hoy en día tenemos diversas formas de programar y crear un sitio web siguiendo este modelo de renderizado. Frameworks como Node.js o Next.js son unos ejemplos de herramientas que podemos emplear.

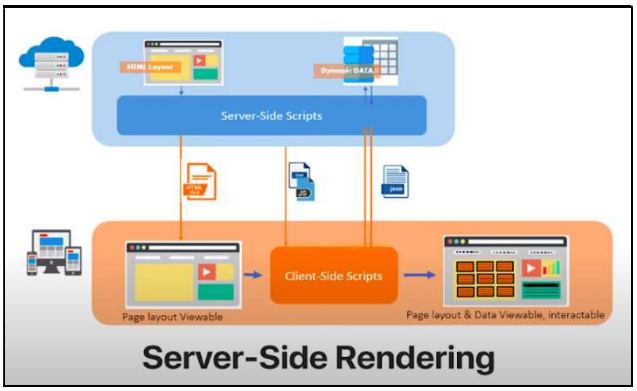

Fig. 2.1 - Funcionamiento del SSR

En la Fig. 2.1, visualizamos este modelo de renderizado, que permite que la información y datos necesarios para el renderizado de la página sean enviados directamente en un único viaje al cliente en cada petición que el servidor reciba. En consecuencia, se evitan viajes adicionales para la obtención de estos datos imprescindibles para el renderizado. Podemos concluir, que éste proporciona un rápido tiempo de respuesta, además de actuar rápidamente para hacer visible y usable la web solicitada por el usuario.

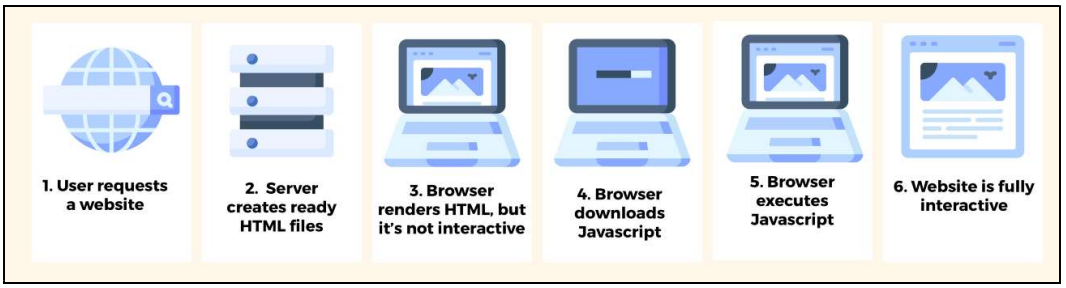

Fig. 2.2 - Orden del funcionamiento del Server-Side Rendering

La Fig. 2.2 resume brevemente el proceso que se lleva a cabo para obtener la página una vez ésta ha sido solicitada al servidor. Primeramente, el servidor pre-renderiza los archivos HTML necesarios y se los envía al navegador, logrando una página web todavía no interactiva. Posteriormente, el navegador descarga y ejecuta cualquier JavaScript acoplado para terminar con el proceso, permitiendo la interactividad en la página web. De esta manera, conseguimos una sensación de progreso inmediata, dónde el tiempo de carga es bajo, evitando el riesgo de abandono inicial por parte del usuario, generalmente causado por largos tiempos de carga inicial.

No obstante, relegar el proceso inicial de renderizado al servidor en lugar del cliente también tiene sus inconvenientes. El proceso de renderizado de una página web, mostrado en la Fig. 2.2 se realiza por cada petición recibida, ya sea de carga o de recarga de la página. Por ende, el rendimiento en páginas que no reciben multitud de peticiones o cuyos usuarios interactúan bastante, pueden verse perjudicadas con este tipo de renderizado. Podríamos estar causando una saturación del servidor y afectar negativamente a las ventajas que este tipo de renderizado posee. No obstante, otro tipo de páginas como blogs y webs estáticas sí que se verán beneficiados por este tipo de renderizado ya que las interacciones con los usuarios son mínimas.

Podemos concluir, que el cliente no interpreta nuestro código, sino que manda una petición a un servidor y este servidor es el que lo interpreta y renderiza. De esta forma, el SEO rendirá mejor ya que la rápida respuesta beneficia al posicionamiento provocando que los crawlers rastreen la página con mayor facilidad. Además, ya que el servidor es mucho más rápido a la hora de renderizar la página, éste mejora la experiencia de usuario, ya que podemos enviar una primera vista previa de la finalización del renderizado.

2.2.2 Static-Site Generation

Dentro del renderizado web del lado del servidor, también tenemos el Static-Site Generation (SSG). Pese a que comparta varias similitudes con el Server-Side Rendering, el Static-Site Generation toma una aproximación ligeramente distinta a la hora de renderizar. En estas situaciones, optar por un sitio web generado de forma estática, provoca eliminar el componente de dinamismo que pueda tener una página web que, según en qué circunstancias, puede ser una ventaja muy grande. Cabe recordar que, en los comienzos de Internet, éste estaba lleno de sitios webs estáticos, previo a cualquier avance en los frameworks y renderizados que tenemos y conocemos hoy en día.

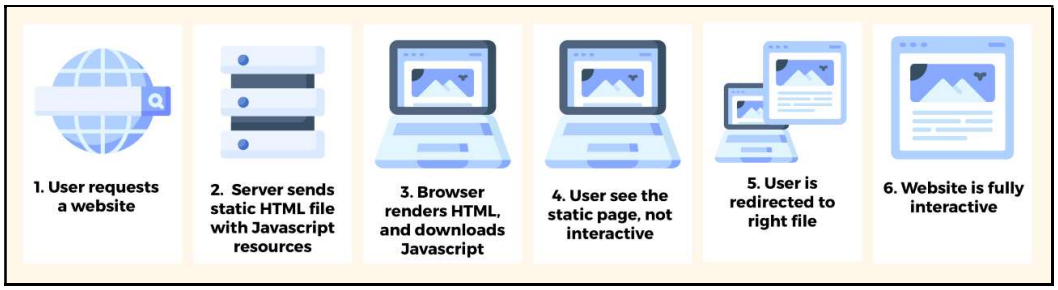

Fig. 2.3 - Orden del funcionamiento del Static-Site Generation

Referente al concepto de web estática, esta la podemos asemejar a la empleada en la creación de blogs y páginas informativas o páginas carentes de dinamismo. En otras palabras, una web generada de forma estática no será más que eso, contenido estático que se halla en algún repositorio. Se rige principalmente por dos conceptos claros, siendo estos los siguientes:

- Las capas que componen la página deben estar desacopladas, sobre todo en el Frontend y Backend, es decir, la obtención de los datos debe ser independiente a la visualización de estos.
- Todo el Frontend debe estar compuesto por páginas estáticas permitiendo la pre-renderización de estas.

El orden que se lleva a cabo cuando se realiza una petición a una web estática es la que podemos observar en la *Fig. 2.3*, donde comienza con el usuario inicial solicitando la página web. A continuación, el servidor envía todos los archivos estáticos previamente generados, como el HTML y JavaScript para que acto seguido, el navegador lo plasme tal cual, permitiéndole al usuario ver la página estática. Finalmente, los archivos JavaScript se ejecutan, haciendo la página web interactiva [3]. Podemos observar una clara diferencia entre el renderizado por parte del servidor y el estático, siendo esta que, en el estático, en lugar de generar la página para cada solicitud como el SSR, esta se entrega directamente al usuario desde el servidor reutilizándola en cada petición, ya que el HTML se genera al desplegar el proyecto y no en cada petición realizada al servidor (SSG).

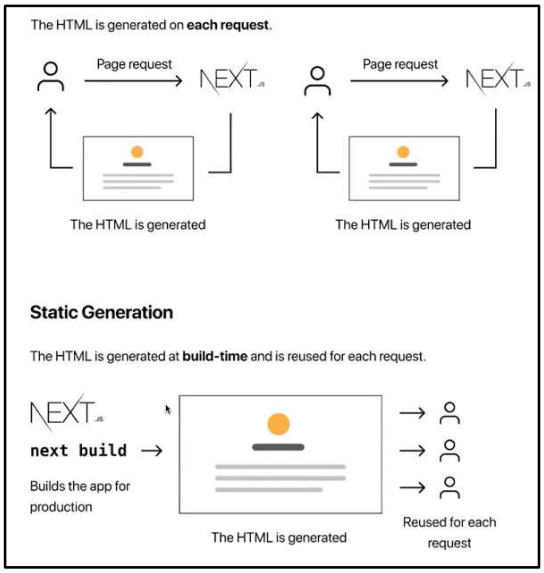

Fig. 2.4 – Diferencias entre Server-Side Rendering y Static-Site Generation

Una de las ventajas de los sitios web estáticos, es la posibilidad de desplegar nuestra página en un Content Delivery Network (CDN), es decir, en una red de distribución de contenidos. Al emplear estos servicios en nuestra página, podemos optimizar e incrementar la experiencia del usuario final, aumentando la velocidad con la que recibe y se plasma la web en su navegador. Esta funcionalidad se consigue al tener distintos servidores que contienen el HTML del servidor principal, por lo que, al hacer la petición al servidor central, en realidad, se la hacemos al servidor CDN más cercano al nuestro. Gracias a este servicio, podemos descentralizar el servidor principal y liberarlo de carga y saturación, puesto que las únicas peticiones que recibirá serían de los CDN para actualizar su sitio estático en caso de que fuera necesario.

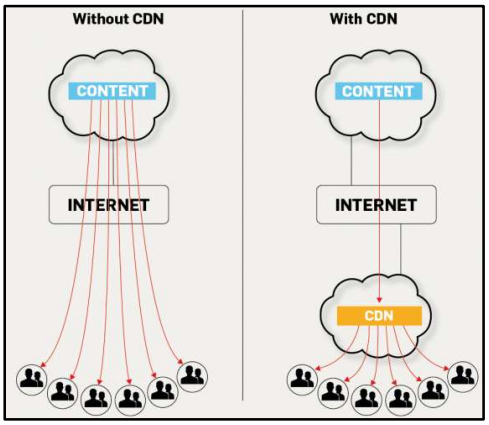

Fig. 2.5 – Funcionamiento de un servicio con y sin CDN

La generación de sitios web estáticos puede considerarse como la creación del archivo HTML con todo el contenido necesario. No obstante, el framework de React también nos ofrece la posibilidad de crear sitios estáticos usando un poco de JavaScript. Como alternativa, podemos usar también un Sistema de Administración de Contenidos (CMS) como WordPress para poder llevar todo esto a cabo debido a que los blogs entran dentro de esta categoría. La página web de Wikipedia, visible en la Fig. 2.6, sería un claro ejemplo de una web estática, carente de gran dinamismo y cuyo contenido es totalmente estático.

Fig. 2.6 – Página web estática de Wikipedia

A la hora de tomar la decisión de utilizar una aproximación estática a nuestra página web, debemos tener en cuenta, como hemos establecido anteriormente, que sirven para páginas con ausencia de dinamismo, cuyo contenido sea estático y poco

cambiante con el tiempo. Posee varias ventajas como su gran funcionalidad, su poco tiempo de cargar ya que el HTML ya está pre-renderizado, mejora el posicionamiento del SEO notablemente puesto que los *crawlers* de Google pueden hacer una mejor lectura de una web ya pre-renderizada que una que no lo ha sido aún. Además, no requieren mucho mantenimiento ni espacio de alojamiento.

2.2.3 Single Page Applications

Por otro lado, también tenemos las Single Page Applications, acortado por SPA. A grandes rasgos, una SPA es aquella página web que no necesita ser refrescada ni recargada para su uso en el navegador. Podemos tomar la página de Gmail como ejemplo, todo está ocurriendo dentro del mismo sitio, sin cambiar en ningún momento ni a la hora de escribir un correo o leer los recibidos. La forma de funcionar es bastante sencilla, el navegador renderiza los archivos HTML, CSS y JavaScript que componen la web una única vez al momento de la petición del cliente y solamente se intercambia con el servidor los datos a mostrar. Esta forma de renderizado web permite priorizar la velocidad, además de permitir que las interacciones responsivas del usuario sean más fluidas y cómodas tomando menos recursos del servidor.

El renderizado de las SPA no es complejo puesto que solo una sección de la página necesita ser actualizada y no su totalidad. La actualización llevada a cabo se puede realizar a través de llamadas al API u otro gestor de contenidos con JavaScript.

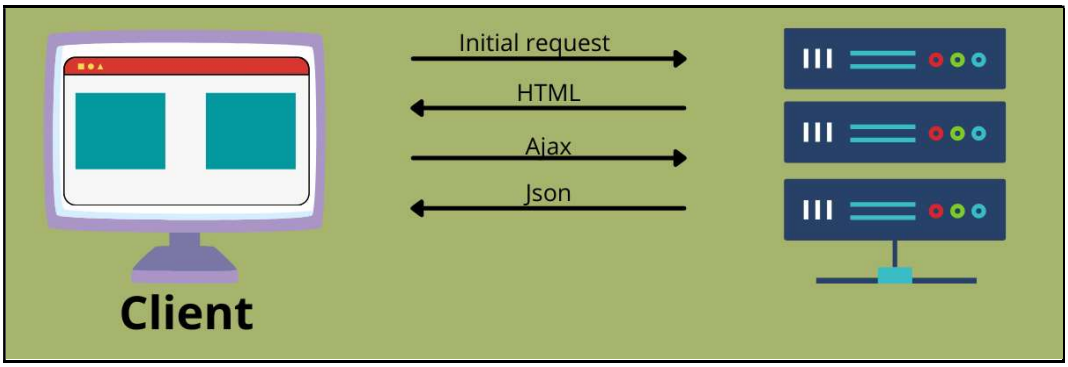

Fig. 2.7 - Orden de peticiones en Single Page Applications

Al observar la *Fig. 2.7*, vemos reflejado el proceso por el cual renderizamos una SPA al solicitarla al servidor [4]. El proceso comienza con el usuario inicial cuando realiza la petición al servidor que tiene alojada la SPA. Acto seguido, este renderiza la página y devuelve un HTML que contiene el esqueleto de la página web sin los datos a mostrar aún. Finalmente, a través de llamadas al API usando AJAX, recibiremos el JSON que precisamos para ir actualizando las distintas secciones de nuestra página web, poblando la página con todo lo necesario. Podemos ver el ejemplo de una SPA a continuación, concretamente, la de Twitter.

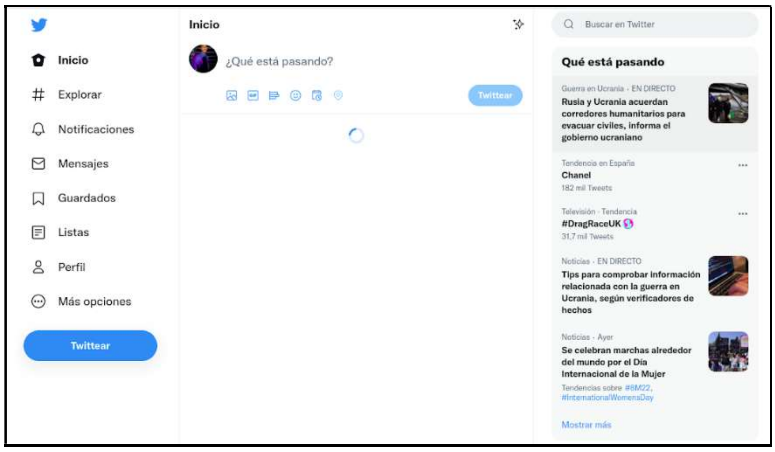

Fig. 2.8 - Twitter con la carga inicial de componentes

Al observar la *Fig. 2.8*, visualizamos que al realizar la petición de la página web, todos los componentes principales se han cargado rápidamente, brindando así un tiempo de respuesta casi instantáneo. Sin embargo, el contenido todavía se está procesando. Esta página es en la que estaremos en todo momento, lo único que irá cambiando es su contenido, que será actualizado mediante técnicas de code-splitting y gracias a llamadas AJAX al servidor, que devuelve los datos en formato JSON, obteniendo así la siguiente página visible en la *Fig. 2.9*.

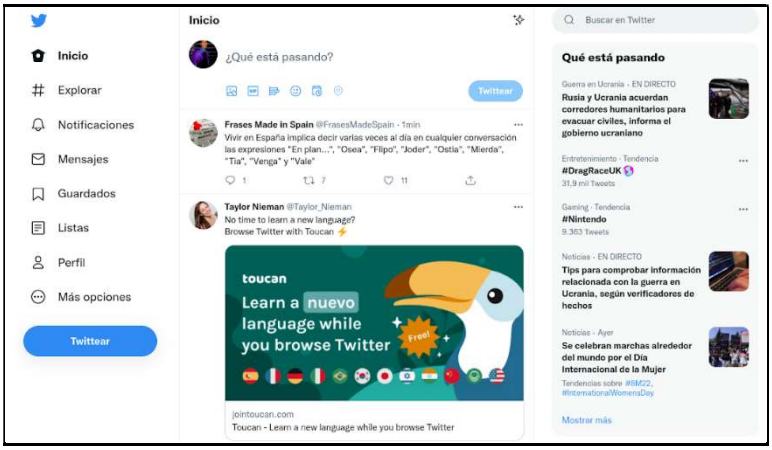

Fig. 2.9 - Twitter con los componentes y JSON cargados

Podemos concluir que, referente a la velocidad, la SPA es muy eficiente, considerando que no necesita realizar recargas de páginas ni excederse en llamadas al servidor. Este hecho conlleva que se mejore la experiencia de usuario, aspecto que trataremos en mayor profundidad en el siguiente apartado del TFG. Además, tenemos un abanico más extenso de ventajas que nos da las SPA, por ejemplo, la posibilidad de almacenar todo en caché de forma muy sencilla, minimiza también la utilización de recursos, simplifica muchísimo el desarrollo permitiendo de esta forma facilitar su depuración.

No obstante, no todo son ventajas, también hay varios inconvenientes que hay que tener en cuenta a la hora de elegir la forma de desarrollar. Uno de los factores más importantes que se ve afectado negativamente al seguir el modelo de SPAs, es el posicionamiento con el SEO. Al ser una única página, esta carece de indexado, metadatos y enlaces propios, propiedades que causan que los crawlers de Google puedan procesar mejor y de forma eficiente la página web. Además, pese a ser veloz al

cargar el HTML inicial, el tiempo transcurrido para la obtención de los datos necesarios, puede ocasionar retrasos y lentitudes en la carga final del sitio web.

	Single Page Application	Multi Page Application
Definition	Single Page Apps are web applications that have a single web page. The browser only reloads particular sections of the page when it receives user requests.	Multi-page web apps are web applications that include multiple web pages. They are reloaded when the user navigates or requests new information.
Speed	• Initial load time can be high • Smooth navigation and fast speeds after loaded	• Comparatively slow • Requires good internet connection especially if the web pages include a lot of graphical elements
Cross Platform	SPA is cross-platform. The backend code of web apps can be used for creating a mobile or desktop application.	MPAs can be cross-platform if built using the right frameworks but the same code cant be reused for developing mobile apps.
Offline	Works offline once loaded as it can cache	Requires internet connectivity to function
Memory Leaks	SPAs can run for hours on the user's device. This can lead to high memory consumption	MPAs have a lifetime of minutes as they will be reloaded. Less chance of memory leaks.
Security	Hard to secure SPAs against cyber attacks	Can be protected against vulnerabilities.
Code Separation	Backend developers and frontend developers have separate and independent code bases.	There is not a clear separation between frontend and backend.
When To Use	• Planning to build mobile apps • Don't require SEO • Software-as-a-Service (SaaS) solutions • closed community requiring logins • social networks	• Enterprises that provide a wide array of services and products • SEO-friendly web platforms • eCommerce stores • Product websites • Blogs
Examples	PayPal, Pinterest, Gmail, Facebook	eBay, Amazon, Google Docs

Fig. 2.10 - Tabla comparativa de SPA vs MPA

Observando la tabla de la *Fig. 2.10*, podemos observar que también podemos tener Multi Page Application (MPA). A diferencia de recargar únicamente secciones de la página como es propio de las SPA, las MPA tienen diversas páginas que son recargadas cada vez que el usuario solicite visitarlas. Las páginas de Amazon o Documentos de Google son ejemplos de una MPA ya que proporcionan una navegación entre más de una única página web, a diferencia de las SPA [5].

2.2.4 Client-Side Rendering

A continuación, estudiaremos el renderizado por parte del cliente, el Client-Side Rendering, que como podremos apreciar, no difiere mucho de las SPAs. Dicho renderizado se resume en tres acciones básicas, siendo la primera, realizar la petición al servidor, el servidor luego envía un *barebone* (esqueleto) del HTML y el resto será renderizado en el cliente. Podemos ver en la Fig. 2.11 el orden en mayor detalle de este tipo de renderizado, donde la mayor carga lógica y de datos, la relegamos al cliente en lugar de pre-renderizar todo desde el servidor [3].

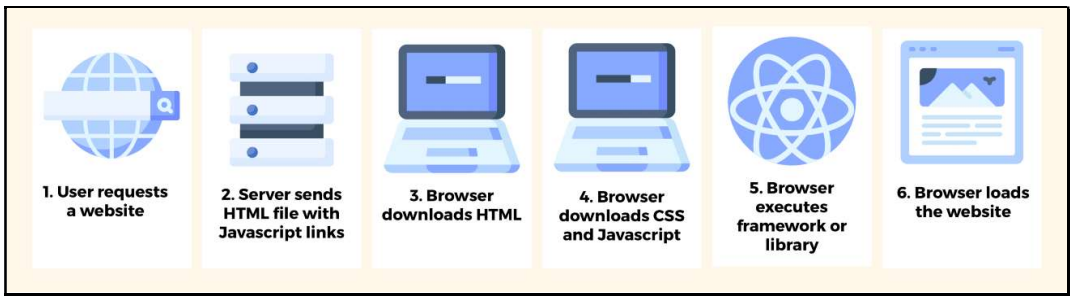

Fig. 2.11 - Orden del funcionamiento del Client-Side Rendering

Una aplicación programada en React suele seguir esta fórmula y podremos diferenciarla fácilmente del resto analizando su código fuente. A continuación, en la *Fig.*

2.12, tenemos una aplicación web en React de creación propia que usaremos para analizar este apartado.

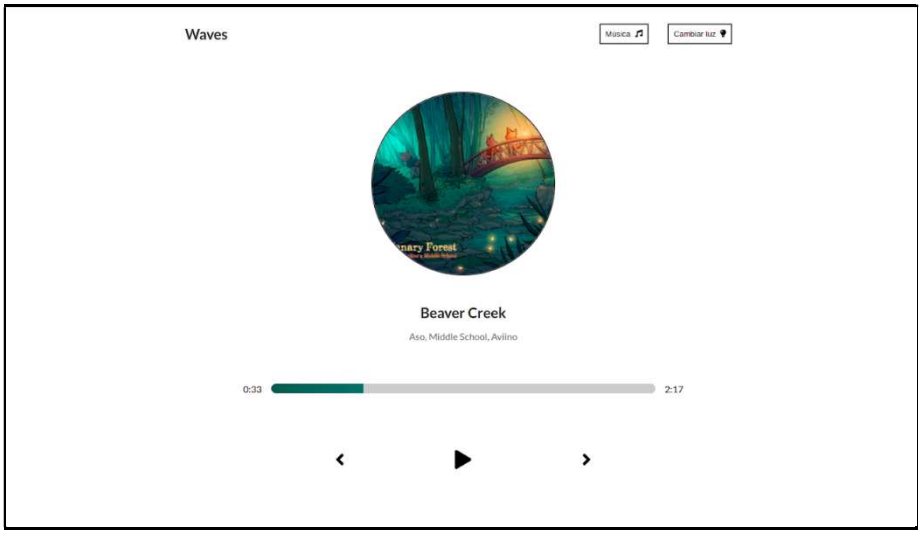

Fig. 2.12 - Aplicación Web de Reproductor de Música

En primer lugar, si inspeccionamos su código fuente, observaremos un fichero HTML simple que no contiene toda la web app en cuestión, solamente los aspectos básicos. Una vez que este fichero, denominado backbone, llega al cliente, este es el encargado de ejecutar todo el JavaScript y renderizar la página correctamente y en su totalidad. En la *Fig. 2.13*, visualizamos el código fuente resultante a la hora de inspeccionar la página web.

Fig. 2.13 - Código fuente de la App de la Fig. 2.12

Apreciamos en la *Fig. 2.13*, que no tenemos nada del cuerpo creado, esta responsabilidad se relega al cliente que debe ejecutar el JavaScript que ha sido pasado junto al barebone del HTML, que ya contiene los meta e información de la cabecera. Debido a que el renderizado ocurre en el cliente y gracias a JavaScript, si lo tenemos inhabilitado en nuestro navegador, nos será imposible renderizar la página y obtendremos el error visible en la *Fig. 2.14*. Este aspecto puede ser una desventaja considerable a la hora de decidir el tipo de renderizado que seguir ya que no contamos con la independencia del tipo de navegador del usuario final.

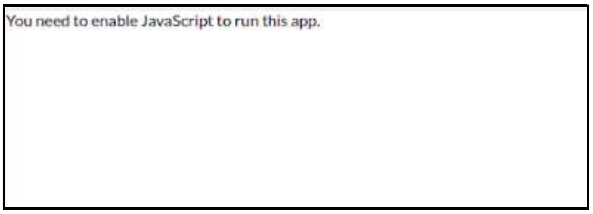

Fig. 2.14 - Error obtenido con JavaScript desactivado

Sin embargo, si se cumplen los requisitos y tenemos el JavaScript activo, el renderizado debería llevarse a cabo correctamente. Al inspeccionar la página con las herramientas de desarrollador al pulsar F12, obtenemos el siguiente código, que ya

representaría en su totalidad a la página web completamente renderizada, como podemos observar en la siguiente Fig. 2.15.

Fig. 2.15 - Código HTML de la App de la Fig. 2.12

Debido a que ejecutamos en el cliente, esto implica que cada vez que llevemos a cabo la petición, estaremos refrescando el contenido que puede haber sido modificado, permitiéndonos estar siempre actualizado y tener la última versión del sitio. Así mismo, traspasar toda la carga y lógica al cliente provoca una menor saturación en el servidor, ya que no tiene que estar gestionando tanta carga como en un sitio renderizado como tal. No obstante, esa rapidez por parte del servidor podría implicar lentitud por parte del cliente, es decir, cada dispositivo es diferente y por ello, también lo es la memoria y velocidad de este. Con esto, podemos asumir que la velocidad de renderizado depende de cada uno de los dispositivos donde se esté realizando la petición.

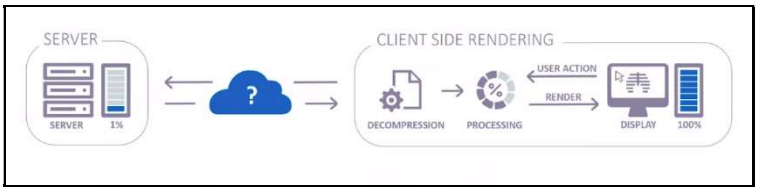

Fig. 2.16 - Funcionamiento del Client-Side Rendering

Si analizamos el efecto que el tipo de renderizado tiene sobre el SEO, podremos concluir que Client-Side Rendering no es una de las mejores opciones si estamos buscando tener un buen posicionamiento. Usando la lógica, podremos entender el por qué. Previamente hemos establecido que, al hacer la petición, solo obtenemos el barebone del HTML, esto implica que no tenemos la web renderizada aún, por lo que la araña de Google no puede analizar nada porque no hay nada que analizar. Pese a este inconveniente, Google ha ido mejorando su araña y la magnitud de este problema ha ido decreciendo con el tiempo, pero sigue siendo recomendable, no emplear renderizado de lado del cliente si queremos priorizar el SEO.

2.2.5 Renderizado Híbrido

En algunos proyectos se puede llevar a cabo una aproximación híbrida, mezclando varios tipos de renderizado según qué sección o aspecto de la web queramos tratar. Distintos frameworks que veremos en los próximos apartados, nos permiten realizar esta fusión de renderizados en nuestra web, dedicando renderizado estático a algunas zonas y dinámico (de servidor) en otras [6]. Por ejemplo, si tenemos una página de nuestra tienda online, podríamos generar nuestra sección de “Sobre nosotros” o la de “Inicio” de forma estática (SSG). Por otro lado, nuestra página de productos sería generada por lado del servidor (SSR) para, como hemos comentado anteriormente, mantener nuestra sección constantemente actualizada y limitar la carga que le enviamos al cliente.

Sin embargo, el renderizado híbrido también tiene sus inconvenientes y el principal es la alta complejidad que este modelo presenta. Este problema provoca que el modelo de renderizado híbrido no sea una elección popular en muchísimas de las páginas webs de hoy en día.

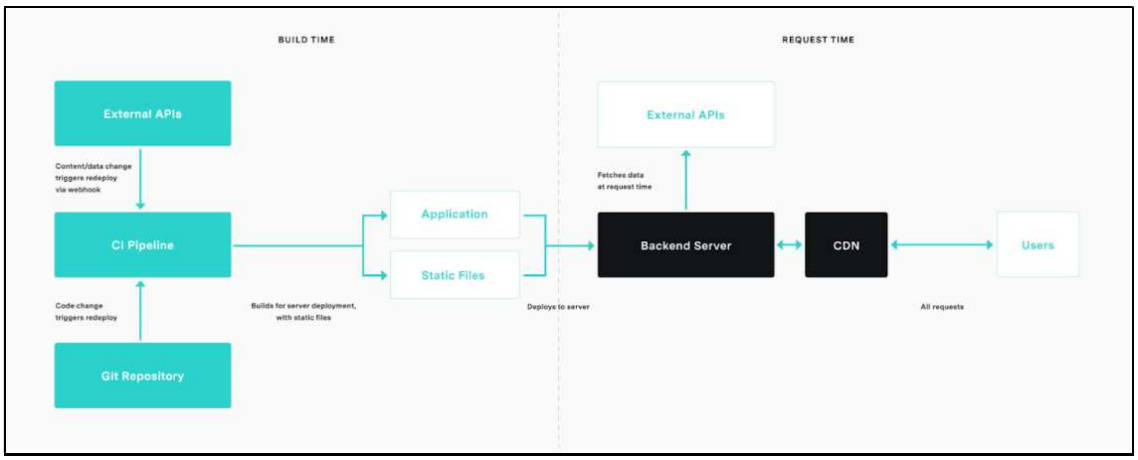

Fig. 2.17 - Funcionamiento del Renderizado Híbrido

2.3 Velocidad

Una vez analizado estas formas de renderizado, la elección de cuál de ellos vamos a elegir para utilizar en nuestro proyecto se vuelve una cuestión muy importante. Uno de los principales factores que debemos tener en cuenta a la hora de la realización de nuestro proyecto, es la velocidad de carga de la página. Según estudios llevados a cabo por Google junto a otras pequeñas empresas [7], se recabó la siguiente información, visible en la *Fig. 3.1* referente al abandono de la página web con respecto al tiempo de carga de esta.

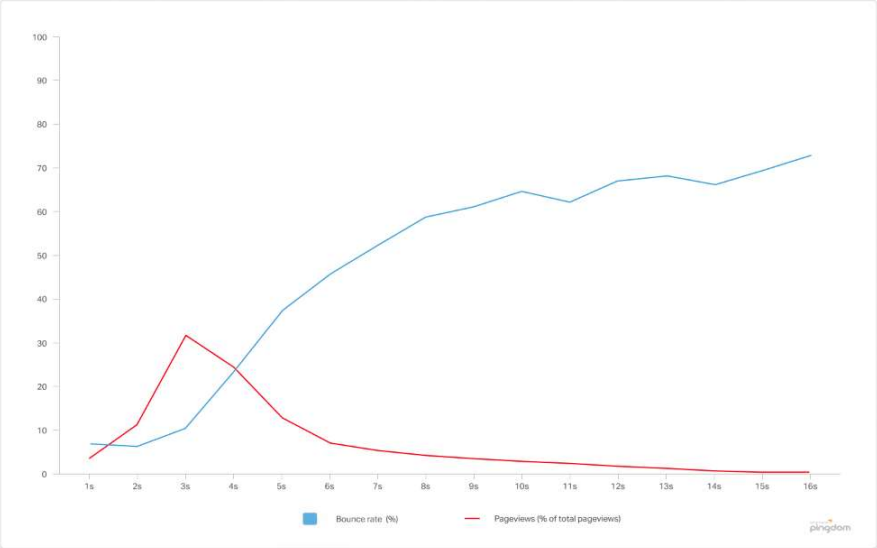

Fig. 3.1 - Porcentaje de abandono con respecto al tiempo de carga

Visible en la Fig. 3.1, observamos que la línea azul representa el porcentaje de abandono de usuarios en una página web a medida que el tiempo de carga de esta va aumentando. Una interpretación de la gráfica nos lleva a la conclusión que lo óptimo debería ser 3 segundos de carga o menos, puesto que después de eso, el porcentaje de abandono crece notablemente, pasando de un 10% de usuarios a un casi 40% [8]. A continuación, otro estudio de Google, visible en la *Fig. 3.2*, corrobora esta información, a partir de los 5 segundos de carga, la cantidad de usuarios que abandonan la página es demasiado grande, siendo esto motivo de pérdidas para muchas empresas [9].

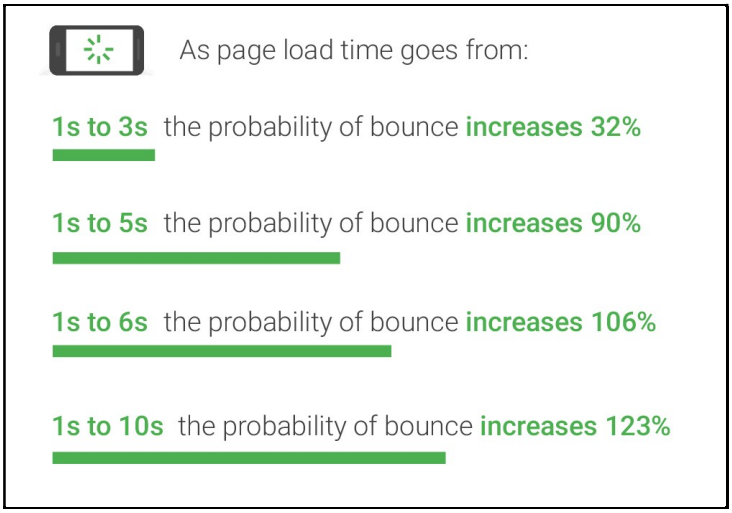

Fig. 3.2 - Aumento del abandono con relación al tiempo de carga

3. Frameworks

Una vez tomada la decisión del tipo de renderizado que emplearemos en nuestro proyecto, la siguiente decisión que hay que tomar sería qué herramientas emplear para llevarlo a cabo. Para ello, tenemos diversos frameworks que nos hará el trabajo más sencillo, no obstante, es crucial la elección de cuál de ellos emplear.

3.1 Next.js

El framework de Next.js es útil a la hora de migrar páginas de renderizado del lado del cliente, o SPAs, a renderizado por parte del servidor. Puede ser un framework potente si tratamos de mejorar la velocidad de carga, entonces realizar la migración se vuelve un aspecto necesario. No tendríamos que renderizar todo desde el servidor, pero convendría que las partes más importantes de la misma sí lo fueran, para permitir así un buen posicionamiento del SEO.

Con respecto a las bibliotecas que podemos utilizar para trabajar con estas tecnologías, tenemos tanto React y Vue, que han sido empleadas en bastantes ocasiones como opción principal por muchos desarrolladores para crear sitios renderizables en el cliente. Con el paso del tiempo, los frameworks de Next y Nuxt surgieron como alternativa para permitir el desarrollo del lado del servidor. A la hora de establecer la popularidad que Next.js tiene en cuanto a frameworks de desarrollo web de servidor, tenemos la *Fig. 3.1* que visualiza la popularidad entre distintos tipos de frameworks con el paso del tiempo.

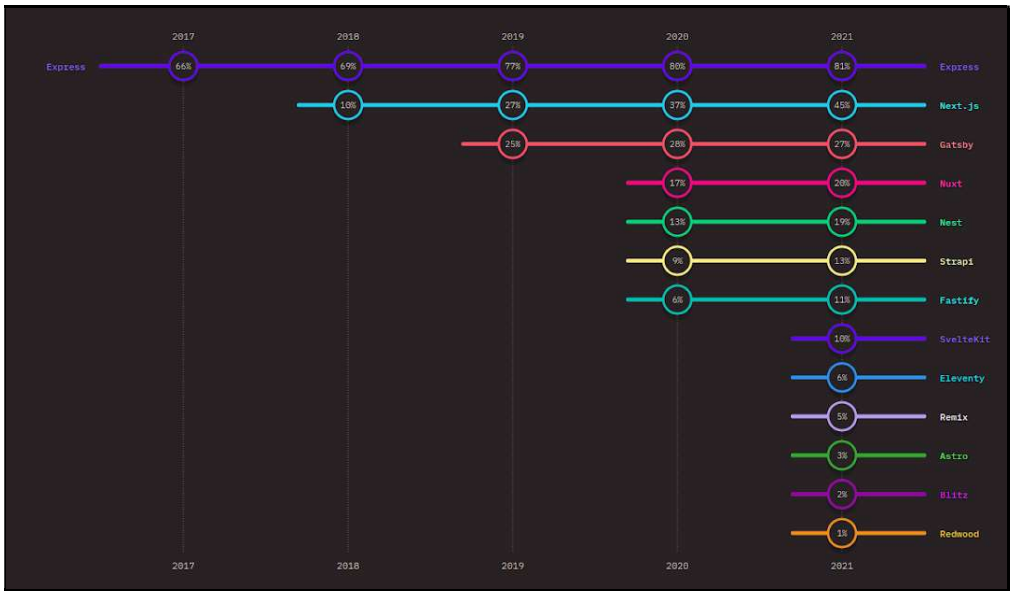

Fig. 3.1 - Popularidad de los frameworks con el paso del tiempo

Como visualizamos en la *Fig. 3.1*, teniendo en cuenta solo los frameworks de JavaScript para desarrollo en la parte del servidor, Next.js se lleva la medalla de oro en popularidad, seguida por Gatsby [10]. Además, su tendencia al aprendizaje sigue al alza con el paso de los años, claramente visible en la gráfica de la *Fig. 3.2*, donde los usuarios a cada vez son más conscientes del framework y muestran más interés por el mismo.

Fig. 3.2 - Popularidad del framework Next.js con el paso del tiempo

No obstante, pese a que la popularidad tanto de Next.js como Gatsby es bastante alta, sobre todo a partir de 2020, donde se notó el *boom* de ambos frameworks y a la hora de la verdad, la decisión de cuál de las dos utilizar puede ser complicada. Por ello, es importante tomar en cuenta las ventajas e inconvenientes de cada uno de ellos, considerando también el proyecto para el cual lo vamos a emplear y analizando sus pros y contras.

Con respecto a Next.js, debemos tener en cuenta que se trata de un framework de React, por lo que su usabilidad puede ser más sencilla si previamente hemos utilizado React en proyectos anteriores. A su vez, el uso de este lenguaje nos facilita mucho la labor de programar el renderizado, ya que tiene un enfoque isomórfico, permitiendo el renderizado tanto en el cliente como en el servidor, ocupándose de la lógica detrás del servidor independientemente y por cuenta propia. Además de todo esto, con el nuevo build que Next.js ha proporcionado a la comunidad, también se permite la creación de sitios estáticos (Static-Site Generation) con el framework, aumentando las posibilidades de creación.

3.2 Gatsby

Gatsby se considera un framework de código abierto basado en React, al igual que Next, que ayuda y facilita la creación de sitios web principalmente estáticos, pero no necesariamente.

Al hablar de Gatsby, debemos tener en cuenta, que ha sido creado para permitir y facilitar la creación de sitios estáticos, es decir, Static-Site Generation, también de React. Estudios confirman que Gatsby es un framework bastante sencillo de utilizar y para hacerlo funcionar simplemente desplegarlo en Netlify o en un CDN sería más que suficiente para que todo funcione correctamente. En cambio, con un sitio renderizado por lado del servidor, debemos tener un servidor acorde al proyecto tanto en recursos como en velocidad o incluso tener varios en plena disposición para poder ofrecer el servicio correctamente. Hoy en día, Gatsby sigue siendo empleado, pero en menor medida, sus capacidades se ven limitadas por el auge de tecnologías alternativas como Next. Podemos ver reflejada esta tendencia en la Fig. 3.3, donde visualizamos la tendencia al alza de las personas que no repetirían con este framework y del desinterés sobre el mismo.

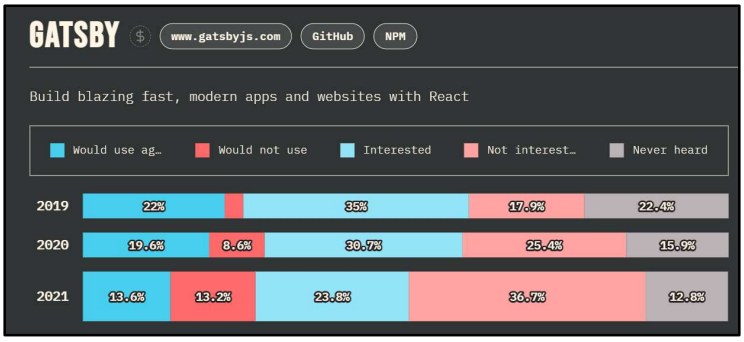

Fig. 3.3 - Popularidad del framework Gatsby con el paso del tiempo

En cuanto a la carga de datos, Next.js es la mejor opción si nuestra página sigue páginas del estilo de Twitter o Reddit, es decir, que tengan actualizaciones de los datos con bastante regularidad. Gatsby, por el contrario, se centra en el desarrollo estático sin tener en cuenta el posible dinamismo que pueda haber, esto no descarta que no se pueda hacer también páginas con actualizaciones, pero la carga de trabajo será notablemente mayor y puede llegar en ocasiones a saturar y ralentizar el servicio.

4. Caso práctico de migración

Una vez realizado el análisis sobre los distintos tipos de renderizado, además de tecnologías de framework que permiten su implementación, pondremos todo en ejecución con un caso práctico. Antes de comenzar, introduciremos dicho caso práctico y sentaremos las bases de este. Nos centraremos en el framework de Next.js de React para migrar una aplicación web que se renderiza en el lado del cliente (CSR), a renderizarse en el lado del servidor (SSR). El propósito de este migrado reside en poner en práctica el conocimiento aprendido, además de mejorar el desempeño y rendimiento de nuestra aplicación con el SEO, liberar al usuario de carga lógica y agilizar la recepción de la página en el lado del usuario. El resultado será una página web que se entregue rápidamente al usuario final limitando la dependencia a la velocidad de conexión y recursos de cada usuario.

4.1 Análisis del proyecto

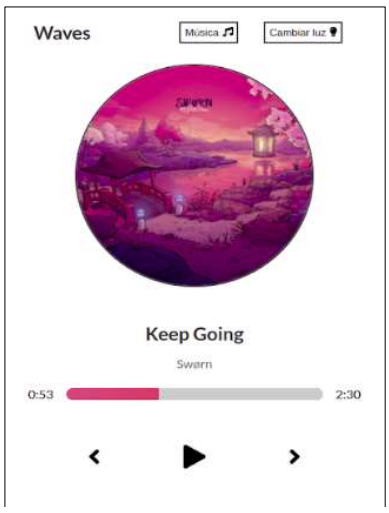

Fig. 4.1 - Aplicación de música de React

Realizaremos la migración del proyecto que ya ha sido introducido y expuesto anteriormente en el TFG. En la Fig. 4.1 visualizamos el reproductor de música creado en React, siguiendo el modelo de renderizado por parte del cliente. Utilizaremos el framework de Next.js para esta migración puesto que, para los desarrolladores, Next.js presenta ciertas ventajas interesantes a la hora de programar. Además, siendo Next.js el framework más utilizado y popular por lo que el aprendizaje y utilización de este, nos puede ayudar y beneficiar en futuros proyectos.

Comenzaremos con un pequeño análisis y comprensión del proyecto con el que vamos a trabajar, llevando a cabo un aterrizaje que permita el correcto y completo entendimiento de este. En primer lugar, analizaremos los distintos componentes que forman nuestra aplicación web, podemos ver la división en la *Fig.*

4.2, donde cada color representa un componente distinto, atendiendo a la guía debajo de la siguiente figura.

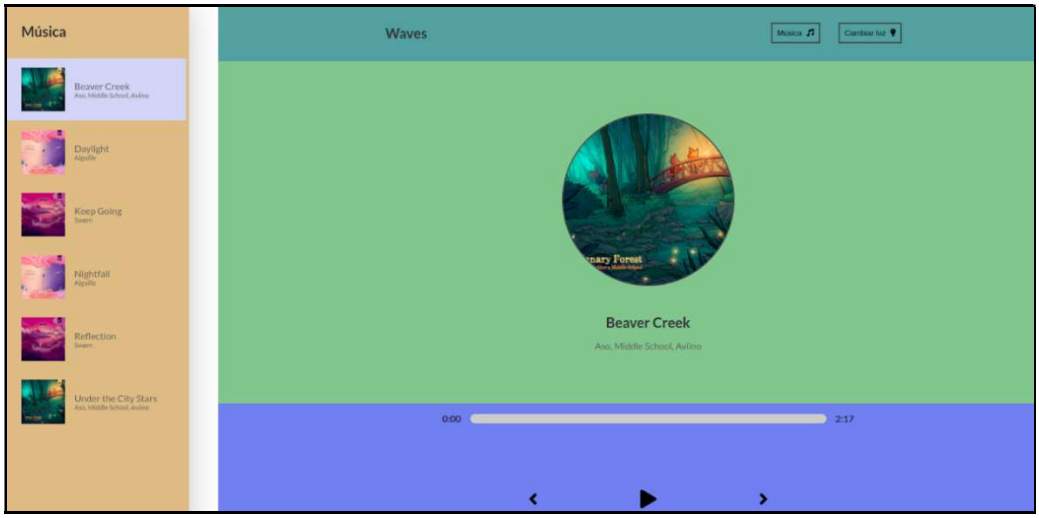

Fig. 4.2 - Web App clasificada por componentes

Tenemos la aplicación que engloba todos los componentes, estos son los siguientes atendiendo a su respectivo color:

- **Naranja** - Librería musical (desplegable con el botón de Música).
- **Lila** - Ficha de la canción.
- **Verde oscuro** - Cabecera con botones de música y cambio de color del tema.
- **Verde claro** - Información de la canción actual seleccionada.
- **Azul** - Información de la reproducción junto a los controles.

Cada uno de ellos posee una función distinta y en su conjunto forman la aplicación. Dichos componentes han sido estilados mediante SASS. La elección de este tipo de estilado viene basada por petición de la empresa ya que defienden que seguir el modelo SASS dota al proyecto de independencia frente a la arquitectura que pueda estar empleando. Además, SASS ofrece una mayor sencillez referente al nombrado de cada clase dentro de los componentes, haciendo el trabajo más fácil que si estuviéramos empleando CSS puro.

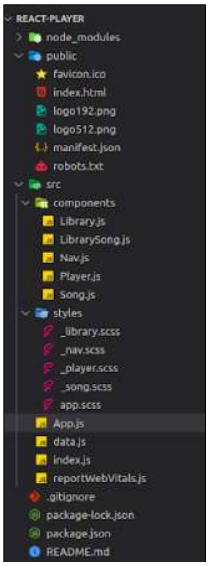

Fig. 4.3 - Distribución de archivos del proyecto

En cuanto a la distribución de los archivos del proyecto, la podemos visualizar en la *Fig. 4.3.* Observamos que tenemos en una carpeta todos los componentes necesarios, otra carpeta de estilos cuyo contenido refleja el diseño que va a seguir cada uno de los componentes y sus respectivos contenidos. Además, tenemos un archivo llamado *data.js* que contiene las canciones libres de copyright que reproduce nuestro reproductor de música. La existencia de este archivo propio nace del problema de encontrar un API que contenga música libre de derechos, evitando así problemas con los derechos de autor de artistas.

Así mismo, realizaremos un pequeño análisis de la página con la herramienta de desarrollador Lighthouse que existe en los navegadores. Dicho análisis del site, nos devuelve información basado en el rendimiento de la página, su accesibilidad, las buenas prácticas, el SEO y si cumple los requisitos para ser una Aplicación Web

Progresiva. Podemos hacer el análisis tanto para dispositivos de sobremesa como para dispositivos móviles. En las siguientes Fig. 4.4 y Fig. 4.5 apreciamos los resultados de nuestra página en cada una de las categorías.

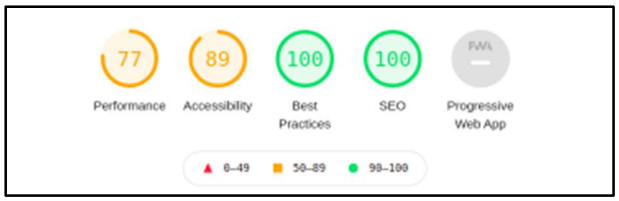

Fig. 4.4 – Resultados de Lighthouse en dispositivos de sobremesa

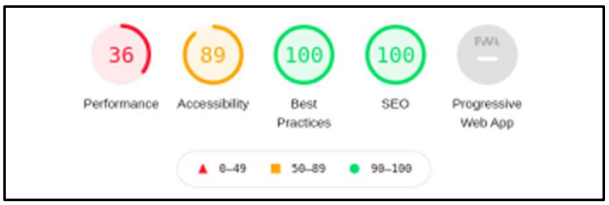

Fig. 4.5 – Resultados de Lighthouse en dispositivos móviles

En dichas figuras, apreciamos que el rendimiento en sobremesa y sobre todo en dispositivos móviles es notablemente mala y debería ser optimizada para hacer el site más eficiente. Por ende, podemos desglosar el panel de rendimiento de las herramientas de Lighthouse, para obtener información más precisa sobre qué aspectos deberíamos mejorar para que pase con mejor calificación los tests de rendimiento.

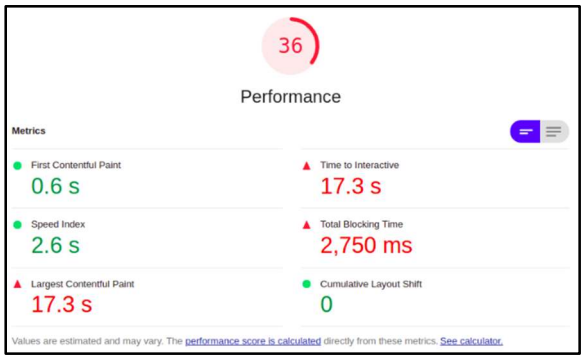

Fig. 4.6 – Desglose del rendimiento de Lighthouse

En la Fig. 4.6 visualizamos el desglose de la información recolectada por la herramienta de Lighthouse para el test del rendimiento del site [12]. Vemos que tenemos aspectos positivos, pero también negativos que afectan bastante al rendimiento final. Empezando con los positivos, tenemos un First Contentful Paint (Primer Despliegue de Contenido) muy rápido, es decir, el tiempo que tarda el primer texto o imagen es representado en el site tras la carga de la página es de menos de 1 segundo. El siguiente aspecto positivo es el Speed Index (Índice de Velocidad) que mide cómo de rápido se visualiza el contenido de la página durante la etapa de carga de esta, transcurriendo en menos de 3 segundos. Finalmente, tenemos el Cumulative Layout Shift que determina la cantidad de desplazamientos de contenido inesperado en nuestra página, siendo cero en nuestro caso.

Visualizando nuevamente la Fig. 4.6, observamos que tenemos 3 resultados muy negativos que nos afectan desfavorablemente. En primer lugar, tenemos el Time to Interactive (Tiempo para Interactividad) que nos indica el tiempo que tarda la página en ser interactiva al completo, en nuestro caso obtenemos nuestra peor métrica con unos

17 segundos de tiempo. Como hemos establecido en apartados anteriores, este punto es primordial a la hora de asegurar que los usuarios no abandonen la página al entrar, por eso este punto es el 35% de la nota final del rendimiento del site. De la misma manera, el Largest Contentful Paint, es decir, el tiempo que tarda el texto o imagen más grande del site en ser representado en el site también es de 17 segundos. Por último, tenemos el Total Blocking Time (Tiempo de Bloqueo Total) que nos mide el tiempo que la página está bloqueada para que no responda a la entrada del usuario, ya sean toques de pantalla, pulsaciones de teclado o clicks. La suma que aparece en esta categoría, se calcula sumando la parte de bloqueo de todas las tareas largas entre el primer despliegue de contenido y el tiempo de interacción. Cualquier tarea que tarde más de 50ms, se considera una tarea larga y la diferencia entre esos 50ms y el tiempo real se suma al Tiempo de Bloqueo Total, es decir, la página está bloqueada a entradas del usuario durante 2750ms, una cantidad de tiempo excesivamente alta.

Posterior al análisis de las métricas del site, podemos sacar en conclusión que una migración al renderizado por parte del servidor, es decir, Server-Side Rendering, puede hacer que las métricas mejoren favorablemente. Como hemos establecido en toda la parte de investigación de los tipos de renderizado, trasladamos la carga del cliente, al servidor, creando así un sitio más rápido y eficiente, manteniendo aun así el buen SEO, las buenas prácticas y la accesibilidad.

Sentada ya la base y conocimiento básico del proyecto, podremos pasar a comenzar con la migración con Next. El objetivo principal de esta migración es que la página se renderice en el lado del servidor en lugar del cliente como está programado ahora mismo. Las grandes ventajas de esta migración es que permitimos el pre-rendering de la web, además de mejorar el rendimiento, que sea más universal además de que tendremos la independencia de si nuestro usuario tiene JavaScript activado en el navegador o no.

Comenzamos abriendo Visual Studio Code para preparar el entorno de trabajo en el que vamos a trabajar. Abriendo una nueva terminal, introducimos los comandos específicos para generar un nuevo proyecto de Next. Acto seguido, tendremos ya lista una plantilla web en Next que podremos modificar y codificar acorde a nuestro proyecto.

```
“ npx create-next-app waves
npm run dev ”
```

Al analizar el contenido que se nos ha generado en las carpetas, veremos que tenemos varias cosas interesantes y que nos podrían servir. En primer lugar, vemos que tenemos una carpeta llamada “*pages*” que contiene las páginas que usará nuestra página web en su totalidad (solo tenemos el index por ahora) y además un API que ya se nos ha creado por defecto. Dentro de la carpeta “*public*”, se guardarán todos aquellos recursos públicos de la página web que serán utilizados, por ejemplo, imágenes o favicons. Finalmente, tendremos la carpeta “*.next*” que se genera una vez hayamos ejecutado el proyecto con el comando “npm run dev”, esta carpeta contiene una construcción que agrupa todos los estáticos del servidor y del cliente, contiene también una caché que permite que los cambios que se realicen, sean aplicados casi de inmediato.

La facilidad y comodidad de realizar un proyecto empleando Next.js se hace notar desde el primer momento gracias a la gestión de errores. Por ejemplo, podemos estar intentando modificar la landing page (index.js) para cambiarla a nuestro propio diseño y al modificar el código, pensamos que aparentemente se ejecuta y visualiza todo correctamente. No obstante, hemos cometido un fallo, pero este fallo no es visible

desde la terminal, sino que la propia web nuestra nos indica en dónde está el error y de qué se trata dicho error, la *Fig. 4.7* muestra un ejemplo. A la hora de programar, estas pequeñas cosas marcan la diferencia entre un framework y otro, haciendo la gestión de errores más sencilla.

Fig. 4.7 - Pantalla de error al compilar en Next.js

4.2 Migración de cabecera y metadatos

Uno de los aspectos más importantes de una página web, que afecta directamente al SEO, son los metadatos y su información de cabecera. Este contenido se encuentra en la cabecera de la página web, englobado en la etiqueta *head*. Dentro de nuestro proyecto en Next, podemos observar en index.js que se ha importado un componente llamado Head que luego es devuelto en la función Home (función que devuelve todo el contenido de la página web principal).

Observamos que dentro del componente Head, las etiquetas usadas dentro son las mismas que las que nos encontraríamos en una etiqueta head de un proyecto en CSR (Client-Side Rendering). En consecuencia, podemos trasladar las etiquetas que teníamos en nuestro proyecto versión CSR a este de SSR como apreciamos en la *Fig. 5.8*. De esta forma, la parte de los metadatos básicos quedaría lista y pasaremos al siguiente aspecto de la migración, que sería los componentes.

Fig. 4.8 - Zona seleccionada del componente HEAD

4.3 Detalles de la migración

4.3.1 Detalles de la migración I - Estilos

Debemos tener en cuenta, la forma en la que Next.js trata las distintas formas de estilar todo un proyecto. La primaria y la que emplearemos en nuestra migración es la utilización de clases. Su funcionamiento comienza dentro de un componente, podemos escribir el estilado en CSS en línea o en otro archivo que guardaremos dentro de la carpeta *styles*. Nos vamos a quedar con la última opción, la de la carpeta *styles*, donde podemos ver que hay varias clases generadas para según qué aspecto de la

web. Un vistazo rápido a lo que se ha creado por defecto, nos muestra distintas clases para elementos como el main, footer, title… Tenemos otra opción también, que es en lugar de estilar por clases creadas, estilar las etiquetas en cuestión, ahorrándonos tener que realizar el nombrado de las clases, actividad que puede ser bastante tediosa en ocasiones, ya que conviene seguir unos estándares de nombrado. Lo que haremos, es utilizar la etiqueta styles de HTML junto a "*jsx global*" para poder codificar de esta manera. En estos casos, lo que hace Next es asignarle un hash único a esa etiqueta que estemos estilando, el problema surge si existe más de una etiqueta en nuestra página web y solamente queremos estilar una, ahí sí que tendríamos que utilizar las clases.

Fig. 4.9 - Estilado con SASS por clases y etiquetas

Apreciamos en la Fig. 4.9, que tenemos estilado de clases, como la clase *library*, *library-song* o *song-description*. A su vez, también podemos ver que, dentro de cada clase, tenemos estilados distintas etiquetas, estos estilados sólo afectarán a aquellas etiquetas contenidas dentro de las clases, impidiendo una generalización a todas las etiquetas del proyecto. Siguiendo esta aproximación con respecto al estilado, facilitamos el nombrado de las clases ya que reducimos el número de clases empleadas, permitiendo emplear estilado de etiquetas locales dentro de cada clase generada.

4.3.2 Detalles en la migración II - Paginación

Como actualmente estamos trabajando con una SPA, no queremos que al hacer click sobre un link, este nos redireccione a otro sitio y cargue o recargue otra página, ya que eso no sería el funcionamiento correcto de una SPA. Por ello, en lugar de utilizar etiquetas de anclaje (<a>) para tratar los enlaces, utilizaremos el componente de Link que Next.js emplea. De esta manera, cuando hagamos click en un enlace, no recargará la página y hará el code splitting, mostrando el contenido necesario en cuestión.

El code splitting, es una técnica utilizada en muchas aplicaciones front end en la que agrupamos distintas secciones de código según la zona o funcionalidad que representen. De esta forma, creamos distintos paquetes, por ejemplo, tendremos el paquete global que estará presente en toda la aplicación y luego paquetes más específicos como por ejemplo inicio o sobre nosotros. Gracias al componente *Link* podremos llevar a cabo el code splitting, por lo que podremos agrupar otras secciones que serán renderizadas al hacer *click*, siguiendo el típico funcionamiento de una SPA como vimos con el ejemplo de Twitter anterior.

4.4 Obtención de datos del API

Para esta segunda parte, queremos que los datos se recuperen del servidor previo al renderizado de los componentes, para poder mostrarlos correctamente. Para ello, Next.js tiene el método *getServerSideProps* que es útil en estos casos. Además, este método nos devuelve un HTML con los datos obtenidos, que luego podrán ser mostrados en el código fuente de la página. Este envío de datos se llama *hidratación*, que consiste en que el cliente, a través de esos datos enviados, sea capaz de renderizar lo mismo que el servidor ha renderizado con todos esos datos incorporados.

El siguiente paso, es traspasar los datos que contiene nuestra aplicación web, al API que la aplicación en Next nos proporciona. En esta API introduciremos todos los datos que la aplicación va a utilizar y mediante el método *getServerSideProps,* que anteriormente hemos comentado, obtendremos todo lo necesario a través del método *fetch*. Lo que devolvemos con el método de *getServerSideProps*, son los *props* que el componente contendrá y los que tendremos que utilizar en este. Debemos tener en cuenta que este método solo es posible utilizarlo en componentes de página completa, no en componentes individuales como son los de React, es decir, los que representan aspectos visuales independientes de la página. Sin embargo, a continuación, veremos cómo tratar este aspecto y hacer de nuestra web una web funcional con el propósito destinado.

Como hemos establecido anteriormente, vamos a hacer uso del API por defecto que se nos ha generado cuando hemos creado el proyecto. Procederemos a insertar en *data.js* (archivo del API), que está situado dentro de la carpeta *api*, toda la información de las canciones que nuestro reproductor de música emplea. Tras la realización de estos pasos, volveremos a la página principal donde renderizamos la web y utilizaremos *getServerSideProps* para obtener los datos en el servidor (SSR). Procederemos a hacer una llamada al API y le pasamos a la página todos los objetos devueltos como *props* para que puedan ser tratados, como podemos visualizar en la *Fig. 4.10* y *4.11*.

De esta forma, estamos consiguiendo que todos los datos sean recolectados en el servidor y la página se renderice con los datos antes de ser devuelta al cliente y usuario final.

```
73
74  export async function getServerSideProps() {
75    const res = await fetch('http://localhost:3000/api/data');
76    const data = await res.json();
77    return { props: { data } }
78  }
```

Fig. 4.10 - Código que incluye la obtención de datos del servidor


```
[{"name":"Beaver Creek","cover":"https://chillhop.com/wp-
content/uploads/2020/09/0255e8b8c74c90d4a27c594b3452b2daafae608d-1024x1024.jpg","artist":"Aso, Middle School,
Aviino","audio":"https://mp3.chillhop.com/serve.php/?mp3=10075","color":["#205950","#2ab3bf"],"id":"02f32f40-7b98-
46fa-9458-deef3244a1b6","active":true},{"name":"Daylight","cover":"https://chillhop.com/wp-
content/uploads/2020/07/ef95e219a44869318b7806e9f0f794a1f9c451e4-
1024x1024.jpg","artist":"Aiguille","audio":"https://mp3.chillhop.com/serve.php/?mp3=9272","color":
["#EF8EA9","#ab417f"],"id":"efee5fa2-1afc-469f-9d96-6ea87f5db91c","active":false},{"name":"Keep
Going","cover":"https://chillhop.com/wp-content/uploads/2020/07/ff35dede32321a8aa0953809812941bcf8a6bd35-
1024x1024.jpg","artist":"Swørn","audio":"https://mp3.chillhop.com/serve.php/?mp3=9222","color":
["#CD607D","#c94043"],"id":"3059e957-2164-4019-bd3b-1b240e1a07be","active":false},
{"name":"Nightfall","cover":"https://chillhop.com/wp-
content/uploads/2020/07/ef95e219a44869318b7806e9f0f794a1f9c451e4-
1024x1024.jpg","artist":"Aiguille","audio":"https://mp3.chillhop.com/serve.php/?mp3=9148","color":
["#EF8EA9","#ab417f"],"id":"b7c9d132-bad3-4269-93d4-9f0fbf543062","active":false},
{"name":"Reflection","cover":"https://chillhop.com/wp-
content/uploads/2020/07/ff35dede32321a8aa0953809812941bcf8a6bd35-
1024x1024.jpg","artist":"Swørn","audio":"https://mp3.chillhop.com/serve.php/?mp3=9228","color":
["#CD607D","#c94043"],"id":"9155b477-b4e8-412b-be6a-4eaa4a15818f","active":false},{"name":"Under the City
Stars","cover":"https://chillhop.com/wp-content/uploads/2020/09/0255e8b8c74c90d4a27c594b3452b2daafae608d-
1024x1024.jpg","artist":"Aso, Middle School, Aviino","audio":"https://mp3.chillhop.com/serve.php/?
mp3=10074","color":["#205950","#2ab3bf"],"id":"1925d2cf-a0c8-4f6f-a5d4-b455563e51cb","active":false}]
```


Fig. 4.11 - Datos del API

Actualmente nuestro API está dentro de nuestro localhost, pero podremos subir esta API a alguna plataforma que nos permita alojarla y ejercer de host y publicarla para poder acceder mediante Internet. A efectos prácticos de la migración, la mantendremos en local.

4.5 Migración de los componentes básicos

Posterior a la realización de los pasos previos, pasaremos a migrar los componentes básicos que forman la aplicación previamente analizada en la *Sección 4.1*. Aparentemente parece ser una tarea sencilla, ya que simplemente tendremos que trasladar un componente de un proyecto al otro. Comenzaremos por crear una carpeta de componentes dentro de nuestro proyecto en Next, para acto seguido crear el primer fichero, el de Library.js. Dentro del fichero simplemente trasladaremos lo que teníamos programado de ese componente en la versión CSR al nuevo proyecto de Next. A continuación, repetiremos el mismo proceso para todos los componentes restantes.

Al terminar de migrar los componentes básicos de la aplicación web, tendremos que volver a la página de index.js y reconfigurar esta sección con todos los componentes que tenemos para armar la aplicación. Gran parte del contenido lo podemos obtener de la aplicación web original. En primer lugar, trasladaremos el código que tenemos del anterior proyecto, instanciando los componentes que hemos creado anteriormente. A su vez, también trasladaremos las variables necesarias para la lógica detrás del funcionamiento de la aplicación. El único sutil cambio que tendremos que tomar entre proyectos es cómo obtenemos los datos del API. En el proyecto CSR los obteníamos gracias a un JSON interno, sin embargo, en el proyecto SSR, haremos la llamada al API, con el método *getServerSideProps*. Acto seguido, como ahora le estamos pasando todos los datos recolectados al componente de la página como props, simplemente inicializamos la variable de canciones con esos datos.

Podemos apreciar la diferencia entre proyectos en la siguiente *Fig. 4.12*. A la izquierda tenemos el componente App que engloba a todo sin pasarle ningún prop. Sin embargo, a la derecha observamos que, a nuestro componente de página, le estamos pasando un prop de data, que ha sido obtenido con *getServerSideProps,* para finalizar inicializando nuestras canciones (variable *song* y *setSongs*) con esos datos.

```
function App() {
  const [songs, setSongs] = useState(data());
  const [currentSong, setCurrentSong] = useState(songs[0]);
  const [isPlaying, setIsPlaying] = useState(false);
  const [songInfo, setSongInfo] = useState({
    currentTime: 0,
    duration: 0,
    animationPercentage: 0,
  });
  const [libraryStatus, setLibraryStatus] = useState(false);
  const [lightingMode, setLightingMode] = useState(false);
  const audioRef = useRef(null);
  const timeUpdateHandler = (e) => {
    const current = e.target.currentTime;
    const duration = e.target.duration;
```

```
export default function Home({ data }) {
  const [songs, setSongs] = useState([data])
  const [currentSong, setCurrentSong] = useState(songs[0]);
  const [isPlaying, setIsPlaying] = useState(false);
  const [songInfo, setSongInfo] = useState({
    currentTime: 0,
    duration: 0,
    animationPercentage: 0,
  });
  const [libraryStatus, setLibraryStatus] = useState(false);
  const [lightingMode, setLightingMode] = useState(false);
  const audioRef = useRef(null);
  const timeUpdateHandler = (e) => {
    const current = e.target.currentTime;
    const duration = e.target.duration;
```

Fig. 4.12 - Declaración de variables en CSR y SSR respectivamente

Si procedemos a ejecutar nuestro proyecto, con todos estos pasos previos realizados, deberíamos obtener una pantalla con todos los componentes renderizados, pero carentes de estilado. El siguiente paso sería migrar el estilado, ya que la lógica debería funcionar sin problema tras la realización de los pasos previos. En la siguiente *Fig. 4.13* visualizamos el estado de la web en el actual punto de migrado. Como podemos observar, el siguiente paso sería aplicar los estilos para poder modificar y adaptar cada componente a su espacio y tamaño correspondiente.

Fig. 4.13 - Aplicación web con componentes y sin estilado alguno

4.6 Migración de los estilos de componente

El siguiente punto sería trasladar los distintos estilos que modifican cada componente y sus etiquetas. Queremos convertir el esqueleto de la *Fig. 4.13* a la versión estilizada y previamente mostrada en los comienzos del trabajo. Para poder realizar lo previamente dicho, comenzaremos por fijarnos en cómo Next.js procede a utilizar los estilos, concretamente el formato SASS.

En la siguiente *Fig. 4.14*, vemos el estilado por defecto que Next nos ha creado, donde en la carpeta de estilos, tenemos dos ficheros, un *global.css* y un módulo para Home. Además, estos ficheros están en formato CSS en lugar de SCSS. Por norma general, estos ficheros aplican estilos distintos, siendo el fichero global el que aplica los estilos globales y el módulo Home estila los elementos del componente página *index.js* es decir, la página Home.

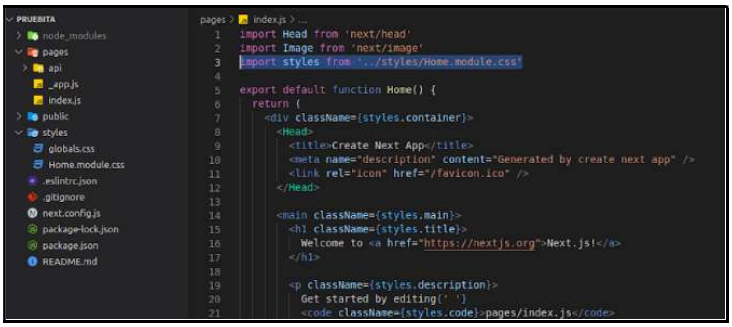

Fig. 4.14 - Carpeta de estilos e importación al fichero

Con respecto a nuestro proyecto, podremos utilizar módulos para cada componente y separar los diseños de cada uno de ellos, o la segunda opción sería agrupar todos los estilos y ponerlos como globales. Esta última opción nos dejaría el resultado visible en la *Fig. 4.15*, como podemos apreciar, existen varios errores a la hora de trasladar el estilado de un proyecto a otro. Comenzando por las animaciones, apreciamos que al hacer click sobre los botones, el menú de canciones no aparece y el cambio de color de tema cuando se le da a la bombilla tampoco. A esto le tenemos que añadimos que los iconos están con unos tamaños desproporcionados.

Una posible explicación de los errores a la hora de migrar el estilado, es el hecho de que todo se haya puesto como global cuando hacer una división por módulos se entendería como una aproximación más idónea. Por este motivo, realizaremos la división de los estilos por módulos, para aplicarlo directamente a cada componente de forma independiente en lugar de un estilo global. En global simplemente dejaremos aquellos estilados de etiquetas básicas como el body, los headers, inputs y el nav.

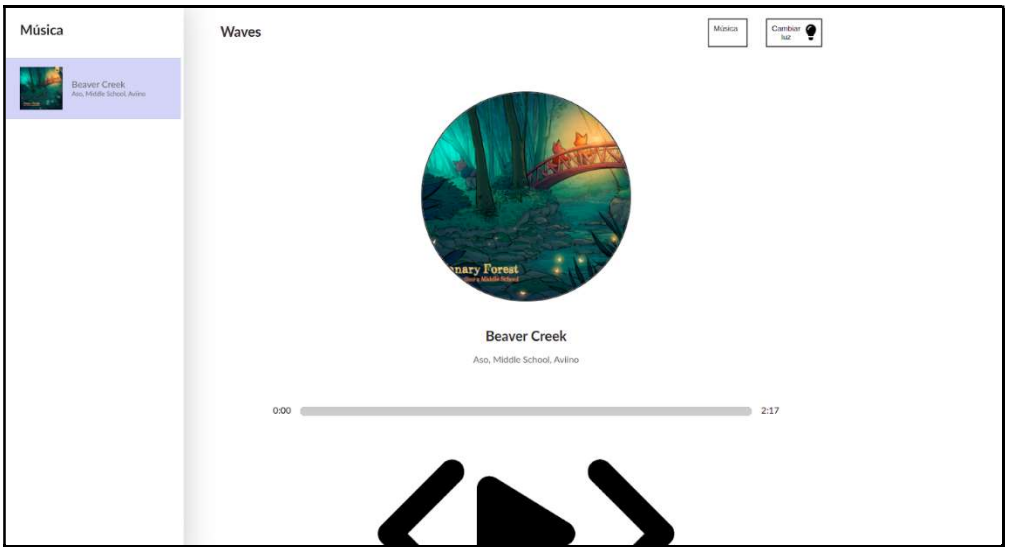

Fig. 4.15 - Problemas en el estilado de componentes

El siguiente paso que considerar, sería intentar solventar la anomalía del tamaño de los iconos. En el proyecto original, todo icono empleado, pertenece al componente de *FontAwesome* de React. Al realizar una investigación del problema, llegamos a la conclusión de que Next.js procesa el CSS de forma distinta que un proyecto de React, ya que los iconos de *FontAwesome* en el servidor, crean un estilo que no coincide con el del cliente, por lo que se tiene que aplicar una serie de configuraciones previas para evitar este problema. Para conseguir el correcto funcionamiento, añadiremos en el fichero global, el *_app.js* las siguientes importaciones para obtener el comportamiento deseado [13].

```
“ import {config} from ‘@fortawesome/fontawesome-svg-core’
import ‘@fortawesome/fontawesome-svg-core/styles.css’
config.autoAddCss = false ”
```

Con esta incorporación al fichero, los iconos pasarían a mostrarse correctamente, escalando el tamaño que tenían marcados por defecto, en lugar de mostrarse con un tamaño desproporcional. Podemos visualizar el resultado y el progreso de la migración, en la siguiente *Fig. 4.16*. Gracias al código añadido, hemos provisto a nuestro proyecto de las ayudas necesarias para hacer que su SCSS actúe como un CSS normal. No obstante, aún queda arreglar varios problemas que la aplicación todavía presenta.

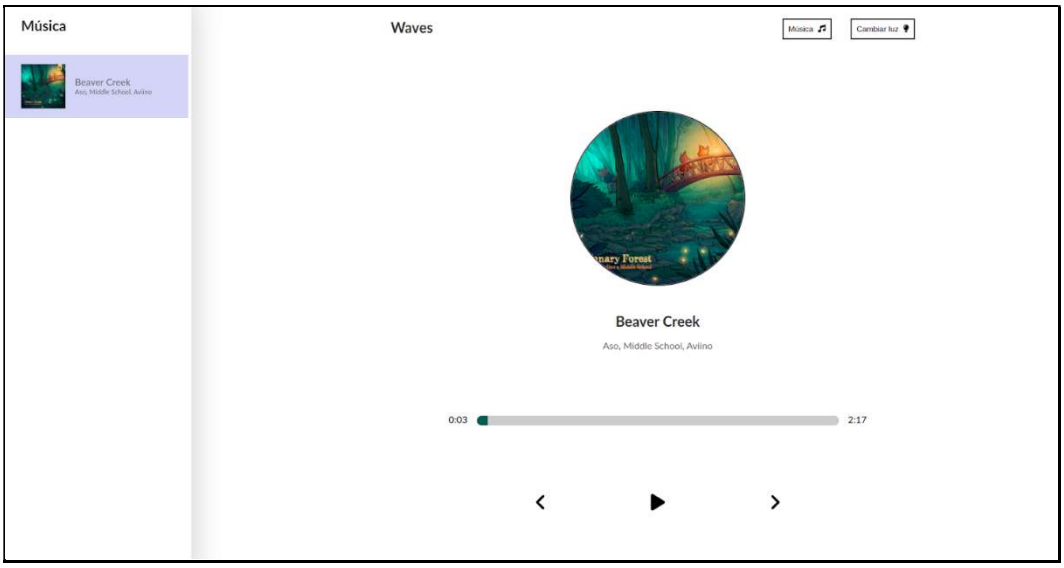

Fig. 4.16 - Web App en Next.js con los iconos arreglados

4.7 Problemas en la migración

4.7.1 Obtención de más datos

Actualmente en nuestro proyecto, tenemos únicamente una canción guardada y que podemos reproducir. Esto es normal, puesto que nuestra API en estos momentos solamente posee una canción, por lo que procederemos a trasladar las otras como objeto que devuelve el JSON. En primer lugar, dentro del archivo del API, *data.js*, insertaremos el resto de las canciones en forma de un objeto que las contenga a todas.

Comenzaremos realizando unas pequeñas modificaciones a la función de *getServerSideProps* que tratamos al comienzo de la migración. Tendremos que modificar la forma en la que devolverá los datos hacia el componente. En lugar de devolver directamente los objetos, devolveremos un único objeto que contendrá a todos, visible en la *Fig. 4.17*, donde comparamos la aproximación que teníamos antes, con la nueva de ahora.

```
73
74  export async function getServerSideProps() {
75    const res = await fetch('http://localhost:3000/api/data');
76    const data = await res.json();
77    return { props: { data } }
78  }
```

```
89
90  export async function getServerSideProps() {
91    const res = await fetch('http://localhost:3000/api/data');
92    const data = await res.json();
93    return { props:
94      { songsData: data, },
95    }
96  }
```

Fig. 4.17 - Comparativa de getServerSideProps

El siguiente cambio que debemos realizar para poder permitir que se recepcionen todas las canciones en el servidor, viene por modificar lo que le pasamos al componente por props. Podemos realizarlo de dos sencillas maneras, la primera sería directamente pasarle las props al componente, y la segunda sería desestructurando su contenido, pasando como parámetro *{songsData}* en lugar de props. Por motivos de

limpieza y legibilidad del código, emplearemos esta segunda forma para obtener las canciones del API, resultando en un proyecto ya casi completamente funcional, observable en la siguiente *Fig. 4.18*.

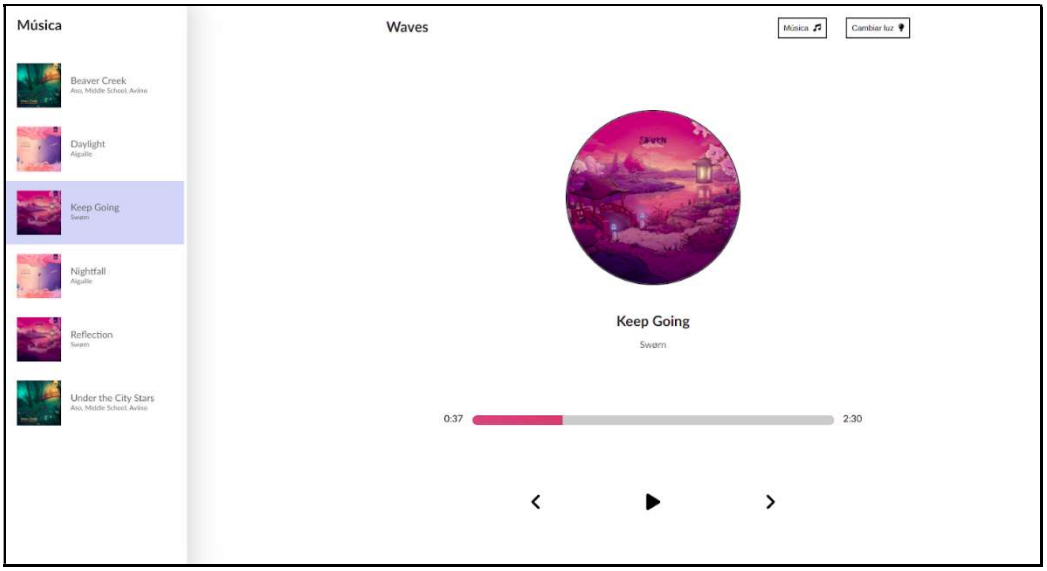

Fig. 4.18 - Web App con todos los datos incorporados

Este punto es bastante importante, puesto que el método de *getServerSideProps* es el necesario a la hora de recolectar los datos. Como hemos expuesto anteriormente, este método recoge los datos exclusivamente en el lado del servidor, nunca en el cliente, además de que el código de la función no será incluido en el JavaScript enviado por el servidor a la hora de renderizar la web. Por este motivo, este método es utilizado únicamente para realizar pre-rendering y no para recolección de datos en el lado del cliente. Además, esta función será ejecutada en el tiempo de petición por lo que no es hasta que recibe una petición, que esta función entra en ejecución.

4.7.2 Desequilibrio en el cambio de tema

Abarcado un problema, nos queda otro por resolver, puesto que aparentemente el diseño reacciona correctamente, pero al cambiar el tema dándole click al botón de “Cambiar luz”, observamos que hay un fallo que provoca que no se cambie el color de todos los componentes, solo de parte de ellos, podemos verlo en la siguiente *Fig. 4.19* de a continuación.

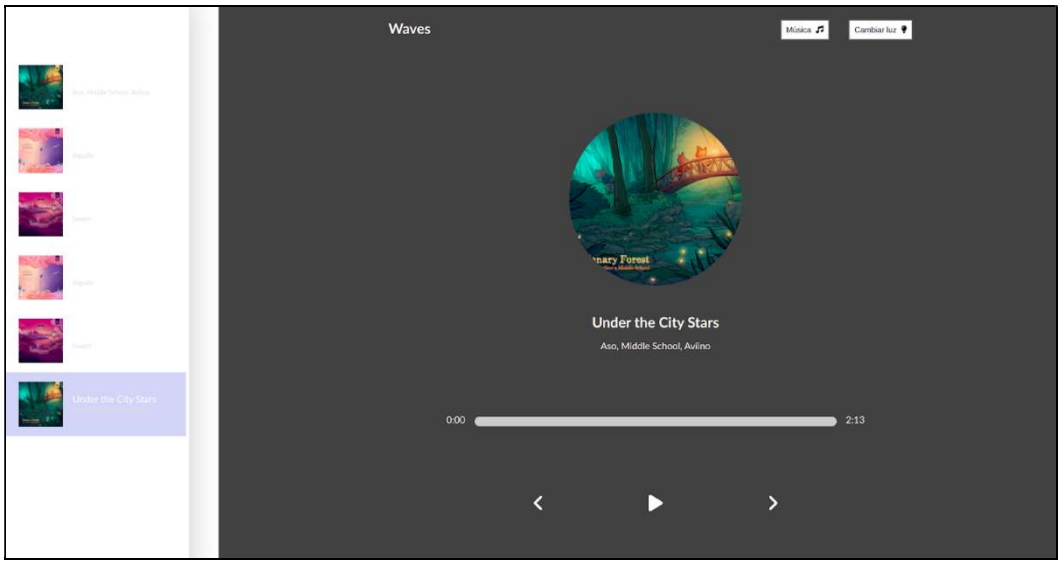

Fig. 4.19 - Web App con el fallo del cambio de tema

Un rápido análisis de la situación puede ayudarnos a determinar la causa de este problema. En primer lugar, debemos tener en cuenta que el cambio de estilos se realiza gracias a una clase llamada *darkMode*, que modifica tanto etiquetas HTML como otras clases, en la siguiente *Fig. 4.20* podemos observar la clase en cuestión.

Podemos ver que modificamos headers, botones, y varias clases, entre ellas, el contenedor de la canción, la biblioteca de música y la selección. Si nos fijamos en la *Fig. 4.19,* los componentes que no se comportan correctamente, son los citados justo ahora, por lo que podemos deducir que el problema puede estar aquí. De la imagen, podemos deducir que solamente se cambia aquellas propiedades cuyos estilos están definidos en el archivo global, es decir, las etiquetas de HTML, excluyendo las clases. Éstas se encuentran en un módulo propio privado. Por este motivo, procederemos a una solución que se basará en incorporar los estilados oscuros a cada uno de sus módulos correspondientes. A continuación, procederemos a trasladar cada uno de los estilos de las clases de la *Fig. 4.20* dentro de sus respectivos módulos, pero renombrándolos como la versión oscura del mismo.

```
.library-active {
  margin-left: 20%;
}

.darkMode {
  background: rgb(65, 65, 65);
  color: white;
  button {
    background: white;
    border: none;
    cursor: pointer;
    border: 2px solid rgb(100, 100, 100);
    padding: 0.5rem;
    transition: all 0.3s ease;
    &:hover {
      background: white;
      color: rgb(65, 65, 65);
    }
  }
  h1,
  h2,
  h3 {
    color: white;
  }
  h4 {
    font-weight: 400;
    color: rgb(224, 224, 224);
  }
  .song-container img {
    border: 2px solid white;
  }
  .library {
    background-color: rgb(65, 65, 65);
    box-shadow: 2px 2px 50px rgb(95, 95, 95);
  }
  .selected {
    background-color: rgb(167, 167, 167);
  }
}
```

Fig. 4.20 - Código de la clase darkMode

Acto seguido, tendremos que pasar la variable booleana que registra el estado del tema de la página web a través de props a los componentes que se verán afectados por esta modificación. Ahora podremos cambiar el tema, ajustándonos al valor que la variable tendrá en todo momento, es por eso, que tendremos que emplear operadores ternarios para declarar el *className* de nuestra etiqueta contenedor que alberga el componente entero. En la siguiente *Fig. 4.21*, podemos observar que en el *div* que contiene todo el componente, el className tiene un operador ternario que depende del valor de *lightingMode*. El funcionamiento comienza si la variable está activa, el componente tendrá el estilado oscuro y si está desactivada, el estilado claro.

```jsx
import React from "react";
import styles from "../styles/Song.module.scss";

const Song = ({ currentSong, isPlaying, lightingMode }) => {
  return (
    <div className={`${
      lightingMode ? styles.darkModeSongContainer : styles.songContainer
    }`}

    >
      <img
        className={`${isPlaying ? styles.songActive : ""}`}
        alt={currentSong.name}
        src={currentSong.cover}
      ></img>
      <h2>{currentSong.name}</h2>
      <h3>{currentSong.artist}</h3>
    </div>
  );
};

export default Song;

```

Fig. 4.21 - Componente de Song que presenta el operador ternario

Este proceso lo repetiremos para los componentes restantes que se han visto involucrados en el problema, es decir, la librería y el componente canción de la librería. Una vez incorporado estos aspectos, la aplicación debería funcionar correctamente y arreglar el error que teníamos en un primer lugar. Una vez todo esté compilado y ejecutado, al probar al hacer *click* sobre el botón de Cambiar Luz, el resultado debería ser el siguiente visible en la *Fig. 4.22* inferior.

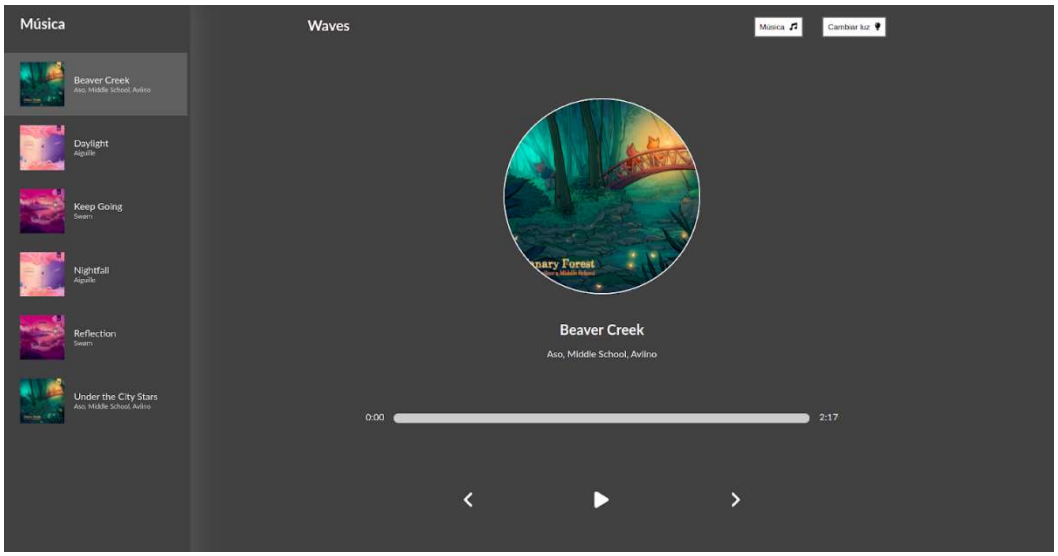

Fig. 4.22 - Resultado final con el tema oscuro funcionando correctamente

4.8 Validación de la migración

Una vez se ha realizado todo el trabajo de migración, podemos llevar a cabo una prueba para comprobar que realmente la página se genera en el servidor y no en el cliente. Como establecimos en el apartado 2 de esta sección, si inspeccionamos el código fuente de una página, podemos determinar si está siendo generado en el servidor o en el cliente. En ese mismo apartado, establecimos que, si obtenemos un código fuente casi vacío, sin contener la totalidad de la página, estaríamos ante un renderizado por parte del cliente. Por el contrario, si el código fuente ya presenta la página en cuestión, estaríamos ante un renderizado por parte del servidor, ya que pre-renderiza la página ahí y posteriormente nos la envía. En la siguiente *Fig. 4.23*, podemos ver el código fuente de la página original y en la *Fig. 4.24*, el código fuente de la página web migrada. Observaremos que el código fuente es bastante distinto, uno no conteniendo la página y el otro sí, concluyendo de esta forma que hemos logrado el objetivo de la migración.

A su vez, podemos observar que aparte de tener el código de la página (el pre-renderizado por parte del servidor), este no muestra el código JavaScript necesario para añadir la lógica y funcionamiento, ni si quiera el código de *getServerSideProps,* simplemente muestra los datos recolectados del API que emplea en la página web. En cambio, el proyecto original, sí que muestra todas estas funciones, además del código que contiene la lógica detrás del site.

En términos del SEO, la migración que acabamos de hacer es muy beneficiosa, ya que permite que la araña de Google rastree y procese la página web desde el código fuente, cosa que no ocurre con el renderizado por parte del cliente ya, que la página no está disponible desde el inicio.

Fig. 4.23 - Código fuente de la página web original previa migración

Fig. 4.24 - Código fuente de la web migrada

Al terminar la migración, podemos comprobar nuevamente si ha habido mejoras en base al rendimiento, accesibilidad, buenas prácticas y SEO como hicimos con el proyecto original [12]. Al ejecutar LightHouse de las herramientas del navegador, podemos comprobar en las siguientes *Fig. 4.25* y *Fig. 4.26* que el rendimiento ha aumentado del 77 al 91 en sobremesa, es decir, un incremento de casi el 20% y un 88% en dispositivos móviles, de un 36 a un 68. La accesibilidad decrece sutilmente, pero esto puede ser solventado fácilmente realizando las modificaciones que la herramienta de LightHouse nos proporciona. En cuanto a las buenas prácticas y al SEO, estas se han mantenido al 100%, siendo esto un aspecto positivo puesto que no se han visto afectado negativamente a la hora de la migración. Se puede realizar una mayor optimización del

SEO midiéndolo con otras herramientas externas a las herramientas de desarrollo que el navegador nos ofrece.

Fig. 4.25 – Resultados de Lighthouse en dispositivos de sobremesa

Fig. 4.26 – Resultados de Lighthouse en dispositivos móviles

5. Caso práctico: Canal March

Biko2 S.L. es una empresa que actualmente está trabajando en varios proyectos a la vez, cada uno siendo manejado por un equipo de desarrolladores distinto. En mi estancia en la empresa, he formado parte de uno de esos equipos, que tiene como objetivo realizar una migración de la página web de un portal de cultura y uno de los principales proyectos de Biko2, Canal March, descrito en ocasiones como el Netflix de la cultura. A continuación, detallaremos un poco el proceso que se llevó a cabo desde la toma de contacto con el proyecto hasta el punto actual.

Canal March es uno de los mayores portales audiovisuales en español, conteniendo más de 5000 vídeos y audios de conciertos, conferencias y exposiciones almacenada en los archivos de la fundación. En la Fig. 5.1, podemos visualizar la página principal de Canal. Inicialmente, en la etapa de generación de la página web, se decidió realizarlo en una SPA, ya que se consideró darle un enfoque de aplicación en lugar de una página web de consumo de contenido estático. De esta forma, se creó la página web enfocándose en una SPA empleando React con CRA (create react app). Con el paso del tiempo, las necesidades de negocio cambian de cierta manera, donde las prioridades se ven cambiadas. En primer lugar, el contenido compartido por redes se ve afectado por la SPA, no podemos lograr pequeñas visualizaciones del contenido al compartir en redes sociales como WhatsApp o Twitter. Así mismo, también se observan problemas de rendimiento y con el SEO, que se ve afectado negativamente por ser una SPA. Con motivo de estos problemas generados y el cambio de las necesidades de negocio por parte de la empresa, Biko2 se reúne con la asociación March, dueños de Canal March, para abordar esta situación e intentar buscar soluciones.

Fig. 5.1 – Página principal de Canal March

Dentro de Biko, existe un modelo de enfoque de soluciones llamado Bitween Suite [14] que es empleado en este tipo de proyectos. El procedimiento comienza por realizar un análisis sobre una necesidad real, en este caso, la solución a los problemas citados anteriormente de Canal March. Esta investigación puede enfocarse, acorde a este caso, en la utilización de una nueva tecnología como es el caso de NextJS. El siguiente paso sería determinar un objetivo a cumplir, siendo en este caso, principalmente, la estatificación de la aplicación, para tener el site con todos sus contenidos pre-generados, consiguiendo así un site mucho más rápido, con sus contenidos pre-generados, otorgando seguridad, y partiendo de un HTML plano. Además, es más barato en infraestructura, mejoramos bastante el SEO y obtenemos *builds* bastante rápidas.

La elección de NextJS a la hora de ser el framework encargado de realizar la migración, parte de la base de querer conocer y estar siempre a la última en cuanto a tecnologías. Hemos establecido anteriormente en el TFG la popularidad de NextJS y el auge con el paso de los años. En este sentido, Biko2 ha querido implementarlo también en uno de sus proyectos. La implicación de utilizar NextJS, permite no tener que empezar todo desde cero, sino que permite una migración incremental, permitiendo solucionar tantas cosas como el redireccionamiento de páginas, el split-coding y la gestión del servidor. Permite el pre-rendering que es muy importante en este caso y el Next Head como punto principal para mejorar el SEO.

Por este motivo, el planteamiento hacia la migración parte de la base de comenzar por un modelo híbrido, transformando aquellas secciones de la página a estático y manteniendo la apariencia de una SPA. De esta forma, no comenzamos con un desarrollo desde cero, sino que es iterativo e incremental, compatible con la arquitectura y las buenas prácticas de cómo planteamos en Biko las SPA.

Al comenzar la migración, teníamos que establecer y priorizar los aspectos que íbamos a realizar. En primer lugar, como también hicimos en el proyecto de React del reproductor musical, vamos a hacer funcionar la SPA actual en una app NextJS, ajustando la estructura vieja a la nueva y analizando los problemas encontrados para intentar darles una solución. Debido a que normalmente en Biko, se trabaja siguiendo un modelo de arquitectura hexagonal, las responsabilidades del proyecto de Canal

March, han sido separadas en capas, la capa *core* para obtener los datos, y la capa *views* encargada de pintar todos los componentes visuales. Actualmente, la web está separada en vistas y obtención de datos, siendo esta última la que realiza peticiones a Drupal (donde se almacena el contenido) y se obtiene todo. Lo que hicimos, fue llevar los componentes tal y como estaban en el proyecto de la SPA de React, al proyecto de Next, comprobando así que la ejecución inicial no diera problemas y sentar la base para continuar desde este punto.

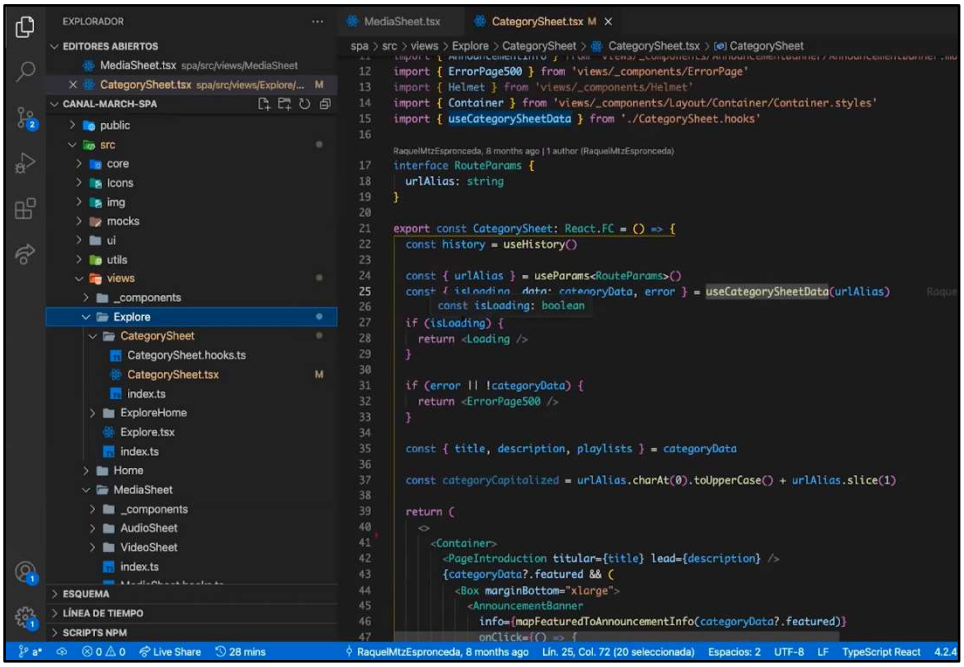

Fig. 5.2 – Componente de CategorySheet del proyecto

Uno de los primeros problemas que nos encontramos es que tenemos que utilizar el *routing* de Next y no el de React (react routing) ya que no funcionan de la misma forma. Por ende, la navegación entre distintas secciones de la site se verá afectadas si no hacemos nada al respecto. Para hallar una solución a este problema, es importante saber que el enrutado de React y de Next siguen modelos distintos, mientras que en React empleamos el componente *react-router*, en Next, tenemos una carpeta *pages* que engloban todas las páginas del site. Como se visualiza en la Fig. 5.2 vemos que utilizamos el componente *CategorySheet* en nuestro proyecto en React para hacer funcionar el enrutado, por lo que, en nuestro proyecto en Next, implementaremos este componente también, pero modificándolo y ajustándolo para que funcione correctamente. Para conseguirlo, hemos creado un *wrapper* que recoge al componente *CategorySheet* y metemos ese *wrapper*, dentro de la estructura de carpetas que nos obliga a crear NextJS. Acto seguido, modificamos el paso de datos necesarios del URL a nuestro componente, por lo que modificaremos el componente para que acepte como propiedad la *category*, consiguiendo que luego pueda hacer la petición directamente al *core* para rellenar la página, como se puede visualizar en la Fig. 5.3.

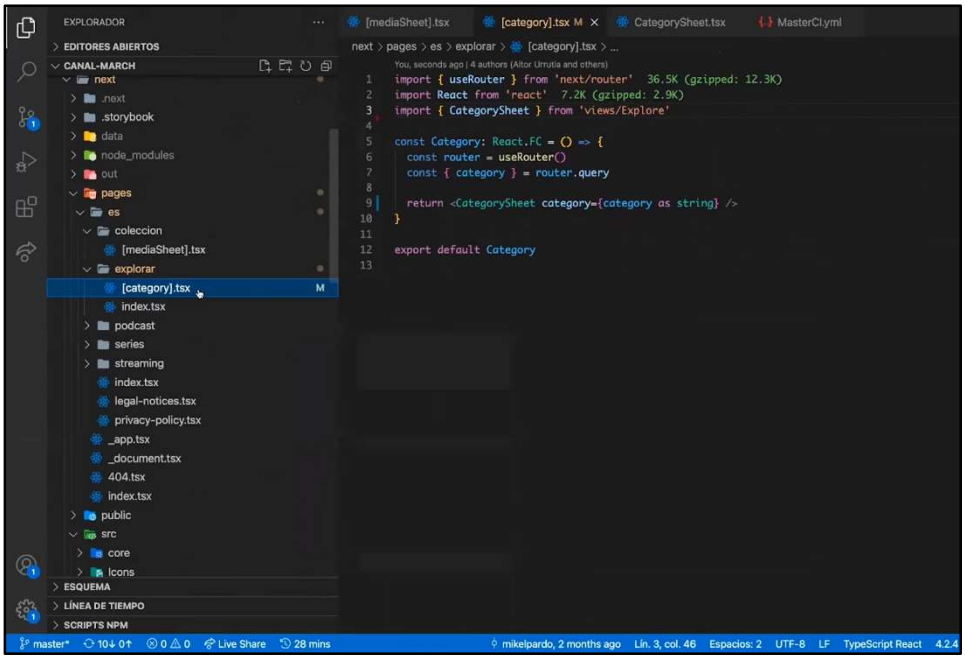

Fig. 5.3 – Componente modificado para aceptar paso de parámetros

Otro de los problemas, como fue en el caso del reproductor musical previamente realizado, fue el de los estilos de los componentes (*styled-components*). En el servidor se creaba un estilo que no coincidía con el del cliente, por lo que tuvimos que aplicar una serie de configuraciones previas para evitar este problema [13]. Otro de los obstáculos que se afrontaron fue la de la ausencia de métodos del navegador a la hora de programar, ya que inicialmente la aplicación había sido pensada para ejecutarse en un navegador y no en Next, por lo que se tuvo que migrar a *useEffect* en lugar de utilizar métodos como *window* o *toolbar*.

“ import {config} from ‘@fortawesome/fontawesome-svg-core’
import ‘@fortawesome/fontawesome-svg-core/styles.css’
config.autoAddCss = false ”

En la siguiente iteración del proyecto, tras haber solucionado los problemas de la iteración anterior, planteamos el nuevo objetivo a realizar. El siguiente paso conllevaría la estatificación de los datos y contenido que pobla el site. Para conseguirlo, NextJS nos brinda dos métodos importantes para la realización de esta iteración, *getStaticPaths* y *getStaticProps*. Estos métodos nos dan la opción de ejecutar SSG a la hora de la compilación del site. Con el primer método, lo que obtenemos son todas las URLs que queremos compilar con este componente, por ejemplo, el componente de *Media*. El método nos devolverá todas las rutas que compilaremos con el componente *Media*. A continuación, emplearemos el siguiente método que itera sobre todas las rutas obtenidas previamente y conseguiremos la información para poder montar las cabeceras de cada una de ellas. Una vez obtenida la información, lo que hacemos es pasársela al componente como *props*, teniendo así la información necesaria para montar el componente de *Helmet* (que contiene el Head de NextJS) generando así la parte estática de la site, visible en la Fig. 5.4. De esta manera, logramos tener estatificada las cabeceras para poder mejorar el SEO de la página.

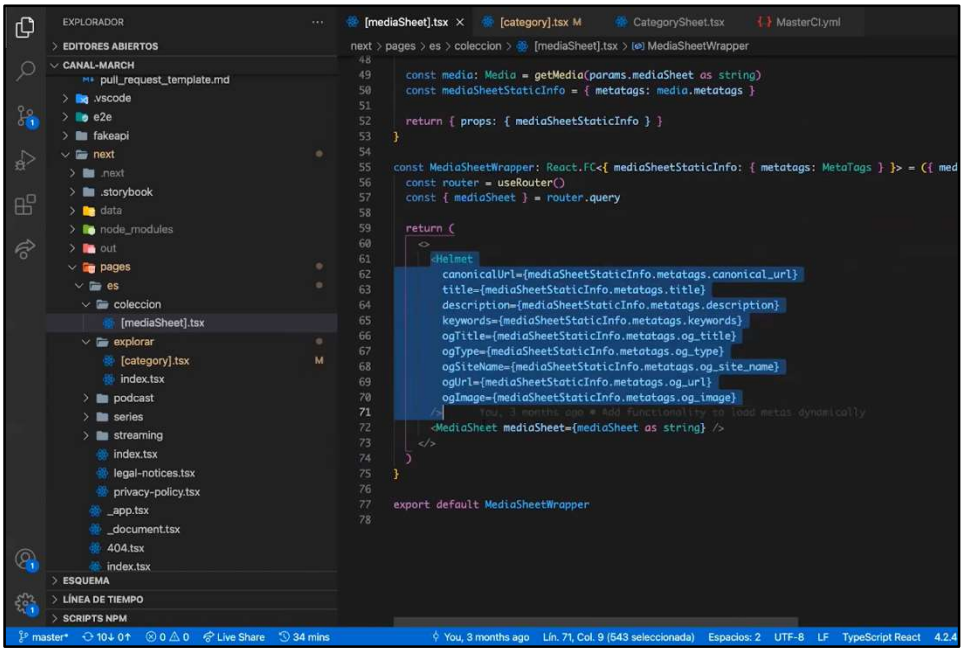

Fig. 5.4 – Componente de Helmet montado

El siguiente punto por tratar, fue la generación de las *builds*. En nuestro proyecto inicial, lo que hacíamos era obtener directamente todos esos datos desde el Drupal que los almacena. El problema reside a la hora de la generación de la *build*, puesto que a causa del gran número de peticiones y contenido que tiene el site, el *build* tarda una cantidad de tiempo inviable. Inicialmente, se solicitaba a través de llamadas al API el contenido con la información de las cabeceras para poder renderizarlas. Con este método, la generación de la *build* tardaba en torno a unas 2 horas. Siguiendo el modelo Bitween Suite de la empresa, se nos ocurre que, en lugar de hacer peticiones a un API, generaremos un JSON que contenga toda la información dividida por contenido, por ejemplo, si solicito los contenidos de tipo vídeo, el JSON contendrá todos los videos, con la información del URL y sus respectivas cabeceras. Esta aproximación fue una mejora considerable, reduciendo el tiempo de compilación a media hora, pero nos quedábamos con un fichero que pesaba mucho, por lo que se intentó profundizar más para obtener una solución con mejor rendimiento.

Al darle una vuelta, se decidió crear un fichero por cada uno de los medios que se necesitaba, todos incluidos dentro de un .zip que ha sido generado desde Drupal. De esta forma, el zip acabará conteniendo todos los datos necesarios para poder generar nuestro site, permitiendo una build mucho más rápida, con un site ya *pre-generado* además de tener un HTML por cada site. Consecuentemente, conseguimos meter los *metatags* en las cabeceras de cada página, logrando un HTML plano con unas cabeceras mejoradas para el rendimiento del SEO. Tras realizar esto, nos quedamos con unos sites que funcionan todavía como una SPA, pero hemos logrado realizar una pre-generación del contenido, estatificando el marco y las cabeceras de las sites, solucionando uno de los problemas iniciales que la empresa solicitaba.

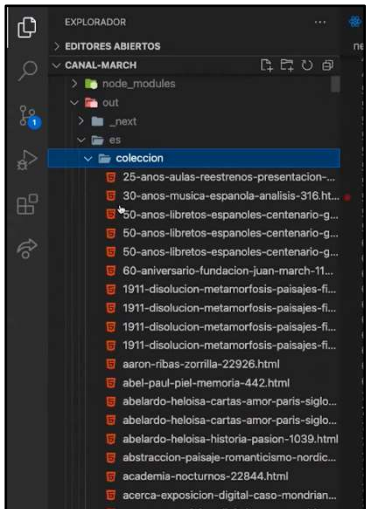

Fig. 5.5 - Páginas pre-generadas del proyecto

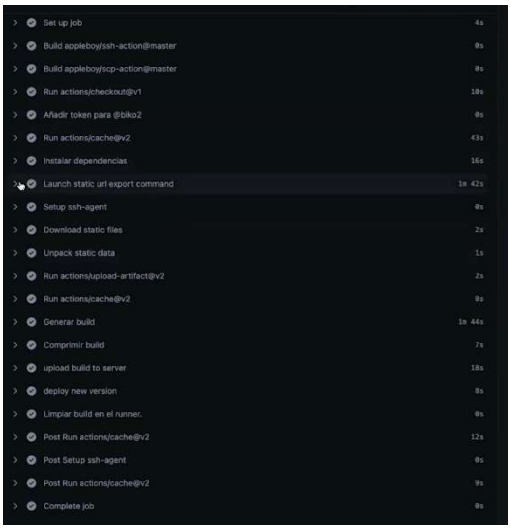

Fig. 5.6 – Pipeline de trabajo para la actualización y generación del build

En la Fig. 5.6 se visualiza el pipeline que se ejecuta dos veces diariamente para actualizar el contenido de la plataforma de Canal March, es por ese motivo que se precisa un tiempo de build rápido para que sea lo óptimo. Podemos observar que, en el pipeline, comenzamos con la preparación del entorno e instalamos las dependencias necesarias para el proyecto. Acto seguido, nos conectamos al servidor Drupal para lanzar el comando que prepara la exportación de JSON y los mete en un .zip, para acto seguido descargarlo en nuestro site y desglosarlos. Finalmente, generamos la build del proyecto, procesando correctamente todas las páginas de contenido, como podemos ver en la Fig. 5.7, y la subimos al servidor de Canal March, cambiando la antigua por la nueva.

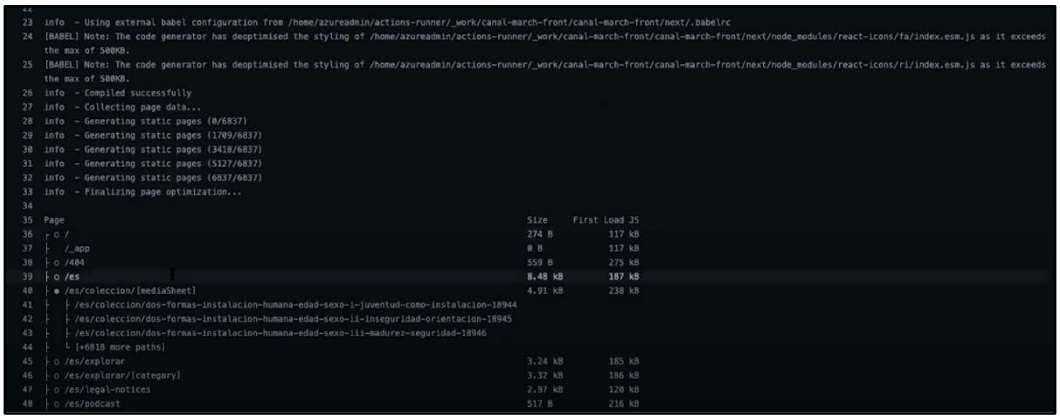

Fig. 5.7 – Proceso de creación de la build del proyecto

En cuanto al proyecto, las mejoras realizadas con NextJS han servido para enfocarse y adaptarse mejor al modelo de negocio cambiante que ha sufrido este proyecto, la Fig. 5.8 lo corrobora con la mejora considerable en todos los aspectos, pero sobre todo en rendimiento y SEO, que era la petición del cliente. Actualmente, este proyecto no ha terminado y se encuentra en la siguiente etapa de desarrollo que comenzará en los próximos meses, por lo que, hasta el momento, la migración no ha terminado. Pese a ello, hemos podido aun así sacar algunas conclusiones con respecto a lo ya realizado y las ganancias que hemos obtenido. En primer lugar, el aprendizaje e incorporación de una nueva herramienta muy potente y versátil como es NextJS en el proyecto, ha sido una ventaja y una gran incorporación a las herramientas que pueden

ser utilizadas nuevamente en futuros proyectos o incluso actuales. Otras conclusiones que podemos sacar de este trabajo, son los beneficios que conlleva el construir aplicaciones *frontend* mantenibles, agnósticas al *framework*, siguiendo buenas prácticas, permitiendo realizar un desarrollo iterativo e incremental adaptándonos al cambio, como ha sido este caso.

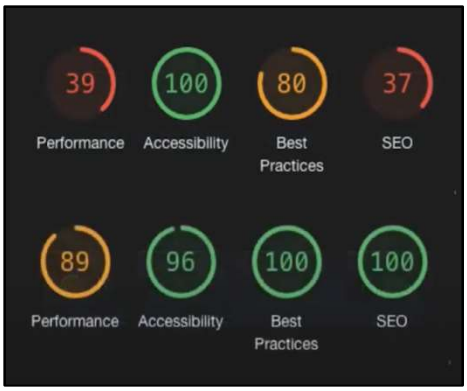

Fig. 5.8 – Comparativa de las métricas antes y después de la migración

Por otro lado, también concluimos que la aproximación de utilizar NextJS ha sido un acierto, puesto que hemos visto que es una herramienta muy versátil y con facilidad máxima para desarrollar y desplegar en producción. A su vez, también nos permite tomar un enfoque más híbrido y no meramente exclusivo hacia una SPA o un site pre-generado, así mismo permitiendo retrasar la decisión del tipo de renderizado lo máximo posible, ya que no existe la necesidad de decidir desde el principio si tomaremos una aproximación de SPA, SSG o SSR. En otras palabras, no se pierde el dinamismo existente en una SPA al combinar SPA con SSG en nuestro site, puesto que no tenemos que refrescar la página para acceder a distintas zonas de esta.

6. Conclusiones y líneas futuras

Tras la realización de este TFG, podemos concluir que existe una extensa cantidad de tipos de renderizado web disponibles para los proyectos que decidamos llevar a cabo. Así mismo, hemos establecido ciertas ventajas y desventajas en cada uno de ellos y determinado en qué ocasiones viene bien emplearlos. Tras el análisis, estudiamos distintos frameworks para la implementación de dichos renderizados y también determinamos las ventajas e inconvenientes de los mismos. Acto seguido, empleamos uno de los frameworks en dos casos prácticos, un proyecto personal como método de aprendizaje de Next.js y el otro en un proyecto profesional de la empresa. No fue hasta la parte práctica que vimos la potencia y el alcance de este framework de React, permitiendo una gran facilidad y sencillez a la hora del aprendizaje inicial del mismo. La curva de aprendizaje del framework y la cantidad de recursos online que hay para recurrir en caso de problemas, hace que emplear esta tecnología multidisciplinar, sea una tarea no muy compleja y aplicable a casi todos los tipos de renderizado. Además, gracias a su carácter intuitivo, lo vuelve una herramienta muy buena a la hora de utilizarlo en futuros trabajos, por ejemplo, en Biko2, NextJS ha entrado para quedarse y no será la última vez que será empleada en sus proyectos. Con respecto al caso práctico de migración de Canal March, esta se encuentra actualmente parado hasta futuras

indicaciones por parte de la empresa, ya que la necesidad inicial solicitada, ha sido cubierta y optimizada. No obstante, mientras tanto, mi equipo de desarrollo está aterrizando en un nuevo proyecto con otra empresa, que empleará también la tecnología de NextJS. En líneas generales, este TFG ha ayudado a tener una mejor comprensión sobre los distintos tipos de renderizado, además de haber proporcionado mayor y mejor información y conocimiento sobre una tecnología framework bastante nueva y que está en continuo auge.

Bibliografía

[1] Ramón Saquete. *¿Qué es el renderizado (rendering) o representación gráfica de la página?*

https://www.humanlevel.com/diccionario-marketing-online/renderizado-rendering-o-representacion-grafica-de-la-pagina

[2] Sodano Pascazi, F.L (2019). "Cómo funciona el renderizado del navegador -detrás de escenas" en Github, 14 de octubre.

https://github.com/r-argentina-programa/traducciones/blob/master/src/renderizacion-navegadores.md

[3] Danilec, A. (2020). "Client-Side Rendering or Server-Side Rendering - What Is the Best Solution for Your Next Application?" en Duomly, 21 de septiembre.

https://www.blog.duomly.com/client-side-rendering-vs-server-side-rendering-vs-prerendering/

[4] Palomares, K. (2022). "¿Qué es una web SPA?" en Kiko Palomares, 2022.

https://www.kikopalomares.com/blog/que-es-una-web-spa-single-page-application

[5] Baryshevskiy, A. (2021). "The Battle of the Web Apps" en MindStudios, 3 de agosto.

https://themindstudios.com/blog/spa-vs-mpa/

[6] Marshall, T. (2021). "Pre-rendered, server-rendered, or hybrid: Which should I use?" en Kontent by Kentico, 16 de junio.

https://kontent.ai/blog/pre-rendered-server-rendered-or-hybrid-which-should-i-use/

[7] Google. *Think with Google Marketing Strategies.*

https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-bounce-rate-statistics/

[8] Pingdom. (2018). "Does Page Load Time Really Affect Bounce Rate?" en Pingdom, 18 de enero.

https://www.pingdom.com/blog/page-load-time-really-affect-bounce-rate/

[9] Google. *How you stack up to new industry benchmarks for page speed.*

https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

[10] The State of JavaScript. *Rankings on back-end Frameworks.*

https://2021.stateofjs.com/en-US/libraries/back-end-frameworks

[11] Anderson, S. (2021). "How fast should a website load in 2022?" en hobo SEO Consultancy, 16 de diciembre.

https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/#:~:text=In%20studies%2C%20Page%20Time%20Load%20goes%20from%201s%20to%206s,web%20page%20was%203.21s.

[12] Google Development Tools. *Lighthouse Performance Scorings.*

https://web.dev/performance-scoring/

[13] Plumber, T. (2022). “How to use Font Awesome with Next.js” en KindaCode, 26 de enero.

https://www.kindacode.com/article/how-to-use-font-awesome-icons-in-next-js/

[14] Biko2. *Bitween Suite, the delightful Drupal and React connector.*

https://www.biko2.com/bitween/

[15] Biko2, “Webinar - ¿Cómo crear un hybrid SPA con NextJS?” en YouTube

https://www.youtube.com/watch?v=T6Cia1BCRTg&ab_channel=Biko