PEC 3.
RENDIMIENTO WEB

Sergi Rourera Llauradó

2n cuatrimestre – 08/01/2021

Trabajo realizado para la asignatura **HERRAMIENTAS HTML & CSS**

Universitat Oberta de Catalunya

Índice

Introducción

Toda la PEC 3 se ha basado en el código obtenido de la PEC 1 + PEC 2. Donde se consiguió crear una plataforma de libros sobre un boilerplate realizado en los módulos de la asignatura y que configura varios plugins y optimizaciones eficientes para el sitio web.

Toda la infraestructura del proyecto se levanta en Netlify, que proporciona un *Continuous Deployment* manejado desde Github. El código *live* deployado se encuentra en la siguiente URL:

https://musing-panini-9fe2a9.netlify.app/

En todo momento se ha utilizado el control de versiones Github para desarrollar, así pues, todas las optimizaciones realizadas en esta práctica se pueden encontrar en el repositorio siguiente, junto con todo el background realizado en la PEC1 y PEC2:

https://github.com/srourera/PEC3-HTML-CSS

Informe de optimización

Primer análisis:

Se ha analizado el sitio web utilizando Google PageSpeed Insights, con un resultado bastante óptimo de **84** en mobile y **95** en desktop.

Aún con este resultado, la plataforma de análisis indica que la página tarda unos 3 segundo en mostrar el primer contenido, lo cual marca como naranja, ya que podría optimizarse.

Aunque la plataforma de análisis no ha mostrado demasiadas propuestas de mejora, si que propone eliminar aquellos recursos de bloquean el renderizado, así como CSS y JavaScript. Además, indica la posibilidad de optimizar mediante los tamaños de imagen y el formato de las imágenes.

Por otro lado, sugiere también utilizar la propiedad *font-display* de CSS para hacer que el texto aparezca, aunque no se haya descargado la fuente.

Después de revisar en Network el *timeline* (imagen 1), se ha decidido intentar utilizar *font-display* para las fuentes (excepto font-awesome) e intentar hacer la carga del script de font-awesome de forma asíncrona para evitar este primer bloqueo en el renderizado.

En caso de que el renderizado también dependiera de las fuentes de google (por culpa del script de font-awesome no queda claro en el *timeline*) también se intentarían importar de forma asíncrona.

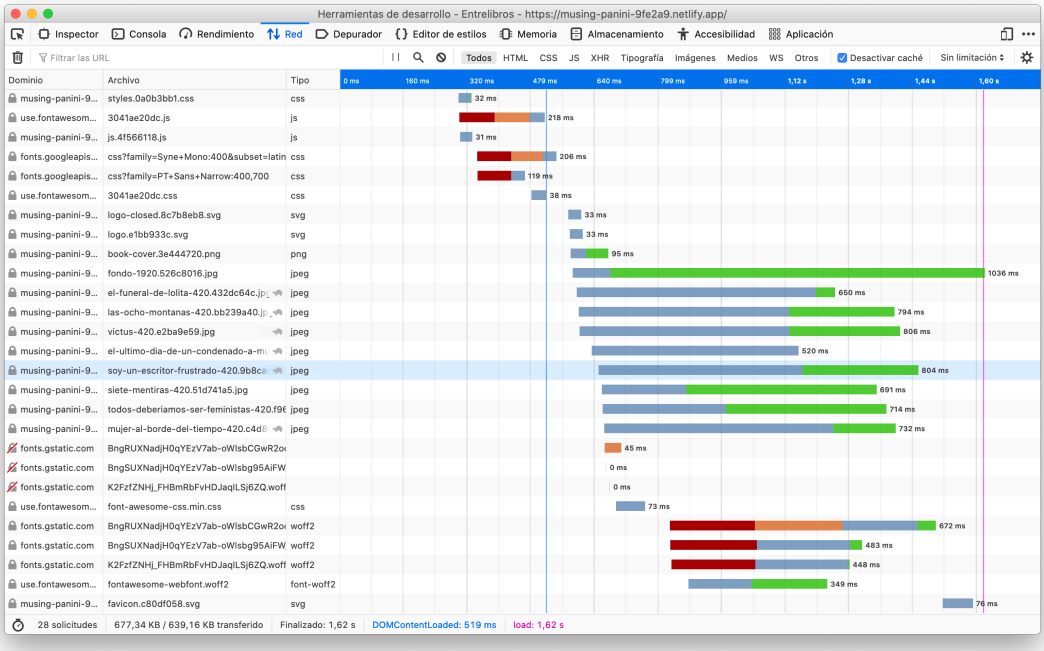
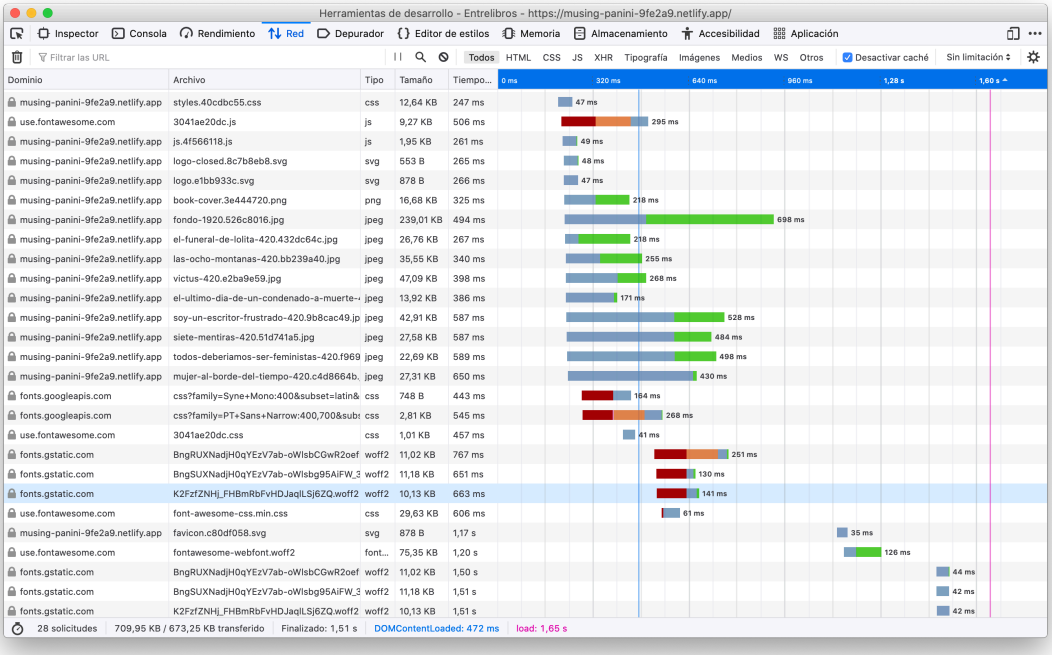


Imagen 1

Segundo análisis:

Tras modificar el script de font-awesome, haciéndolo asíncrono, y de aplicar font-display a las fuentes para evitar el bloqueo del texto hemos visto que las imágenes y otros recursos empezaban a descargarse mucho antes, ya que no tenían que esperar al fichero js.



Con el cambio, en el análisis de Google PageSpeed Insights, se ha subido a 96 en desktop, sin embargo, ha bajado a 84 en mobile y aún no está claro el motivo.

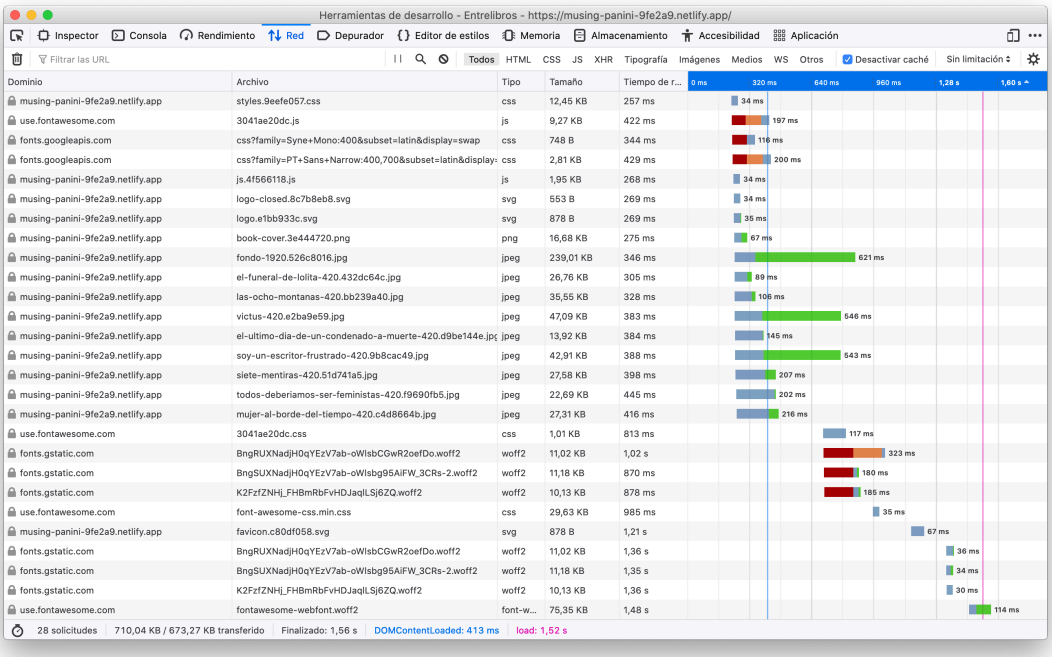
Bien es cierto, que el análisis ha dejado de avisarnos del bloqueo de ficheros js en el renderizado, e incluso ha dejado de sugerirnos que apliquemos el font-display en las fuentes.

Aún así, sigue sugiriendo que quitemos el import bloqueante de css, con lo cual no estoy tan de acuerdo ya que un primer renderizado sin estilado que acaba transformando a la página final con estilo, me parece poco bonito. No se intentará optimizar demasiado por esta rama, ya que de todos modos, es de los procesos que menos afecta ya que tiene un tiempo de respuesta pequeño.

Por otro lado, se ha decidido probar de quitar el plugin de google-font para postcss para evitar importar la fuente desde css y poder hacerlo directamente desde el html, para poder paralelizar el proceso.

Tercer análisis

Tras quitar el plugin de google-font, la carga de las fuentes se paraleliza:

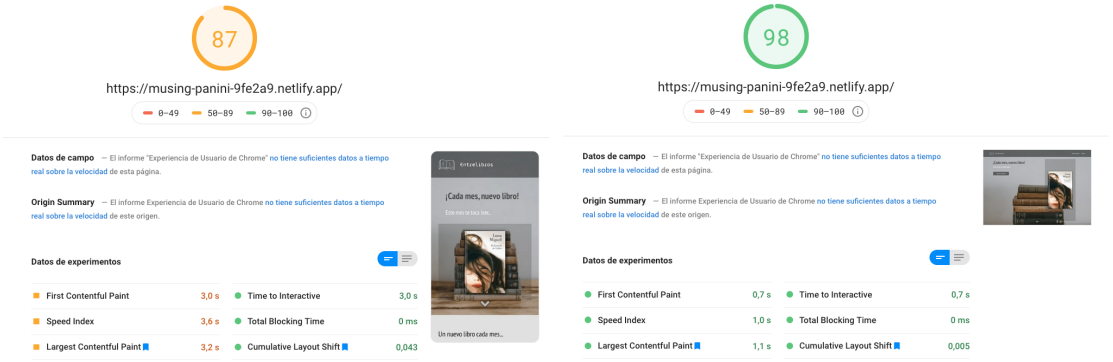


En este caso, pero, el análisis de Google PageSpeed Insights sigue indicándonos la misma puntuación.

Por último, sigue indicando que las imágenes son demasiado grandes, y aunque ya se realizó en la PEC anterior toda la gestión de imágenes responsivas, se le va a echar un vistazo, quizá intentando cargarlas de forma *lazy*.

Resultado final

El resultado final tras aplicar lazyload a las imágenes (aunque hay ciertos navegadores que no lo soportan y simplemente lo ignorarán), el resultado ha acabado siendo bastante favorable con un **87** en mobile y un **98** en desktop.



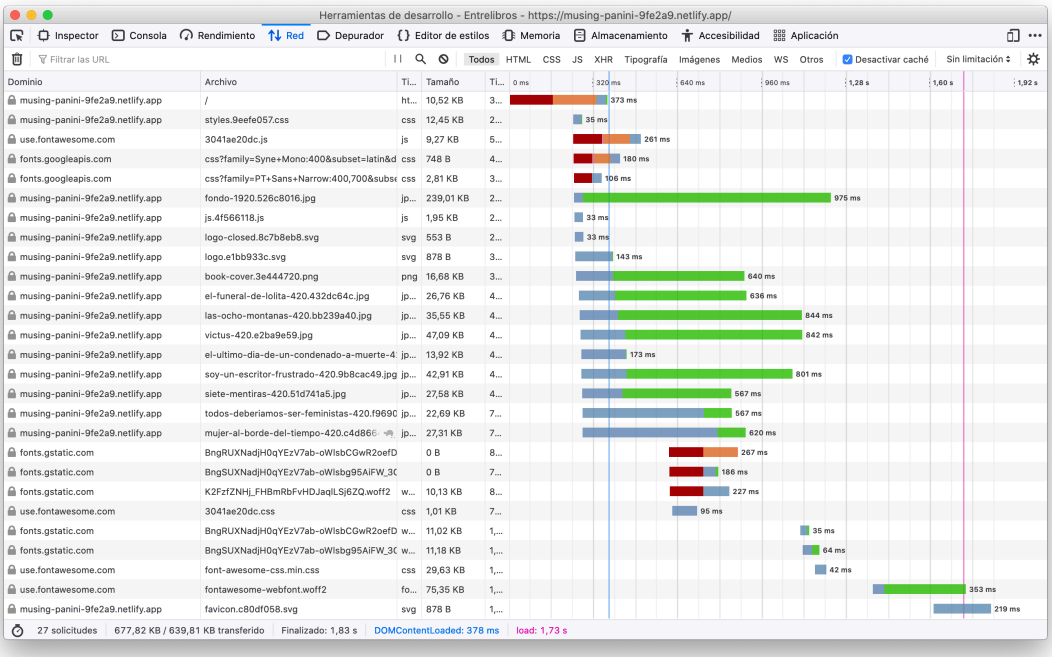
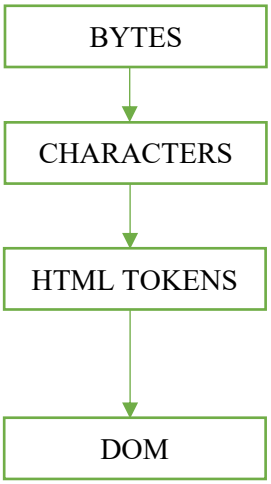


Tabla con Regular 2G

Título	URL	Tiempo de carga	Peso total	Peso transferido	Cantidad de recursos
Entrelibros	URL	31,13s	637,90 kb	616,60 kb	25
Libros mensuales	URL	33,34s	0,99 mb	999,40 kb	35
Autoconocimiento	URL	5,55s	144,76 kb	128,93 kb	17
Feminismo	URL	8,05s	245,68 kb	229,32 kb	18
Misterio	URL	7,12s	296,52 kb	280,74 kb	18
Narrativa	URL	7,07s	171,52 kb	155,65 kb	17
Novela histórica	URL	8,98s	267,12 kb	250,31 kb	19
Novela negra	URL	6,05s	183,59 kb	167,66 kb	17
Poesía	URL	5,01s	214,41 kb	198,61 kb	18
Romántico	URL	5,16s	153,66 kb	137,86 kb	17
Diciembre 2019	URL	8,34s	243,52 kb	223,77 kb	23
Febrero 2020	URL	8,67s	258,01 kb	238,68 kb	22
Mayo 2020	URL	11,60s	349,37 kb	329,79 kb	23
Junio 2020	URL	10,91s	325,60 kb	306,24 kb	22
Julio 2020	URL	9,54s	288,25 kb	268,64 kb	22
Septiembre 2020	URL	12,32s	372,51 kb	351,69 kb	24
Octubre 2020	URL	9,53s	276,02 kb	256,55 kb	22
Noviembre 2020	URL	8,28s	249,27 kb	229,88 kb	22

Preguntas de teoría

Describe, de forma esquemática, cuáles son las principales fases de renderizado de una web.



En primer lugar, el *Browser Engine* descarga del servidor (o disco local) los *bytes* del fichero HTML necesario para esa URL en concreto.

Seguidamente, usando el *encoding* especificado (UTF-8, por ejemplo) convierte estos *bytes* en caracteres individuales procesables.

Siguiendo los estándares de HTML5, convierte las cadenas de caracteres a tokens que representarán cada una de las aperturas y cierres de las etiquetas HTML.

Finalmente, de estos tokens crea objetos con sus respectivas reglas y propiedades y los lee de arriba abajo creando lo que se conoce como Document Object Model, que se trata de una estructura de árbol con relaciones padre-hijo.

Durante este último proceso, el navegador va detectado los recursos necesarios para construir la página web, ya sean imágenes, estilos, código javascript o cualquier otro recurso, como fuentes, que se descargarán en diferentes *threads* para conseguir evitar cuellos de botella. Sin embargo, la construcción del DOM puede quedar bloqueada por, por ejemplo, los recursos javascript que no se declaren explícitamente como *async*.

Además del DOM, el navegador debe generar un CSSOM cuando los recursos necesarios de estilado han sido descargados, y realiza un proceso casi idéntico para procesar el CSS, creando un Object Model de estilado, que debe adjuntarse al DOM. Hasta que no están ambos preparados el renderizado queda bloqueado.

Desde un punto de vista de rendimiento, ¿tiene más sentido situar los tags STYLE dentro del HEAD de la web, o dentro del FOOTER? Y desde el punto de vista del desarrollador, ¿cuáles son los compromisos que ello implica?

Usualmente, los tags de estilos deben ser bloqueantes para la renderización, para evitar malos aspectos visuales temporales en la página web. Es por eso, que en el punto de vista de rendimiento, como antes se pidan los recursos CSS, antes se habrán descargado y antes podrá crearse el CSSDOM. En este ámbito, es necesario situarlos en el *head* debido a que es la primera parte que *parsea* el *Browser Engine.*

Sin embargo, si no nos importa este aspecto del diseño de nuestra web, si los situamos en el footer, el primer renderizado de nuestra página web ya se habría realizado y tendríamos contenido incluso antes de obtener el estilado. Aún así, podemos conseguir un estilado asíncrono con otros métodos más limpios y eficaces como utilizar media queries como atributo al enlazarlo al html.

De todos modos, si se quisiera realizar de forma asíncrona para optimizar el rendimiento, debería tenerse en cuenta un estilado base que no se realizara asíncrono para proveer por lo menos un mínimo de usabilidad.

¿Y en el caso de los elementos SCRIPT?

En el caso del código javascript, suele cambiar un poco el enfoque, ya que, al igual que los recursos de estilado, los scripts también son bloqueantes del DOM si no se especifica lo contrario.

Es por eso por lo que los scripts sí que suelen situarse en la parte inferior del body, para permitir al *Browser Engine parsear* toda la información que necesita para renderizar el contenido de la página web sin quedarse bloqueado por la descarga de un pesado recurso javascript.

Además, si se situara los scripts javascript en el head, si al ejecutarse accedieran a ciertos elementos del DOM que aún no están renderizados, evidentemente no los encontraría.

¿Qué diferencia hay entre los valores de las columnas "transferred" y "Size" en la pestaña Network de las herramientas de desarrollo de Firefox Developer Edition? ¿Qué otras columnas son útiles para las personas encargadas de mejorar el rendimiento web?

Los valores de las columnas *transferred* y *size* solo difieren cuando el recurso se ha comprimido, ya que *transferred* indica la cantidad exacta de bytes traspasados y, en cambio, *size* indica el tamaño real del recurso descargado.

Principalmente, la columna más útil es la del timeline, que permite ver todo el transcurso de lo que se ha ido descargando, lo que ha bloqueado a cada recurso, etc.

Otras columnas interesantes serían, por ejemplo, la de response time y la de tipo de archivo.

Conclusiones

Ha sido una práctica muy importante, conocer todos los procesos realizados por el navegador al renderizar una página nos permite tener una visión mucho más clara y entender muchos de los conceptos que aplicábamos porque sí, simplemente porque alguien lo dice (como por ejemplo situar los scripts al final del body).

Ya había usado con anterioridad la sección de Network, pero creo que de ahora en adelante podré sacarle muchísimo más provecho ya que tengo más claras cada uno de sus apartados y funciones.