Lezama López Yael

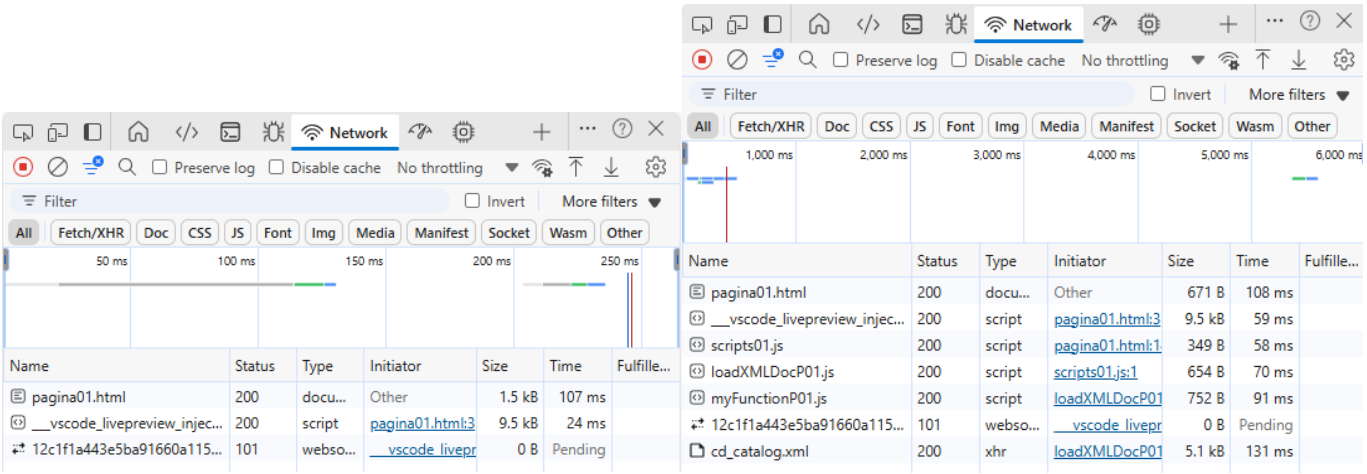
Análisis de Refactorización

Página 01

Para optimizar el código y mejorar su estructura, se dividió el script principal en tres archivos JavaScript con responsabilidades específicas. El archivo loadXMLDocP01.js se encarga de cargar el documento XML denominado cd_catalog.xml, que contiene la lista de discos compactos, y preparar la tabla que se mostrará en la página; myFunctionP01.js procesa el archivo XML cargado previamente, extrayendo los valores de las etiquetas relevantes para distribuirlos en las columnas correspondientes de la tabla, tales como “cd” y “artista”. Finalmente, scripts01.js funciona como controlador principal, ya que asigna un EventListener a un botón de la interfaz, el cual invoca la función de carga cuando se presiona.

La primera página, con todos los scripts integrados dentro de un documento HTML, reduce ligeramente el tiempo de carga, registrando un promedio de 133ms. Aunque a primera vista disminuye el número de peticiones HTTP y puede parecer más eficiente, dificulta el mantenimiento y escalabilidad del proyecto, ya que todo el código está dentro de un mismo archivo.

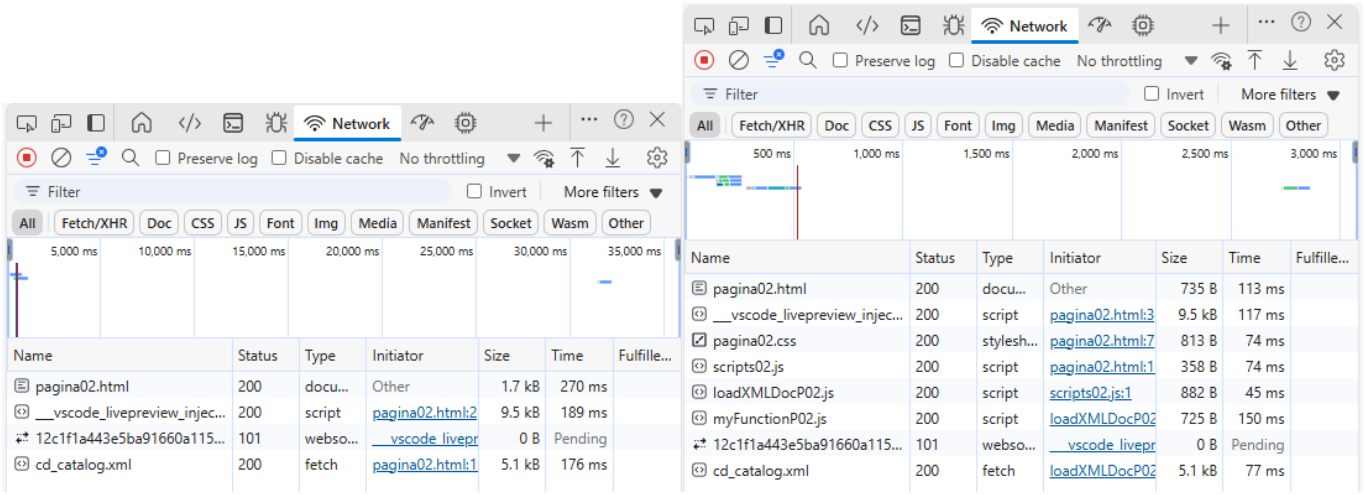
Por otro lado, en la segunda imagen se han modularizado los scripts, aumentando el número de solicitudes y el tiempo total de carga. Sin embargo, permite un mayor ordene en el desarrollo, facilita el mantenimiento de código y promueve la reutilización, ya que cada módulo cumple con una función específica.



Página 02

El archivo loadXMLDocP02.js tiene como propósito cargar el documento cd_catalog.xml, que contiene una lista de discos compactos, y genera la tabla que se muestra en la página, utilizando una lógica distinta a la empleada en versiones previas. Por su parte, myFunctionP02.js se encarga de procesar dicho archivo XML, extrayendo la información de las etiquetas relevantes para organizarla en columnas como “cd” y “artista”, también mediante un enfoque distinto al utilizado anteriormente. Finalmente, scripts02.js actúa como el archivo principal del sistema, donde se asigna un EventListener a un botón de la interfaz que invoca la función de carga definida en loadXMLDocP02.js.

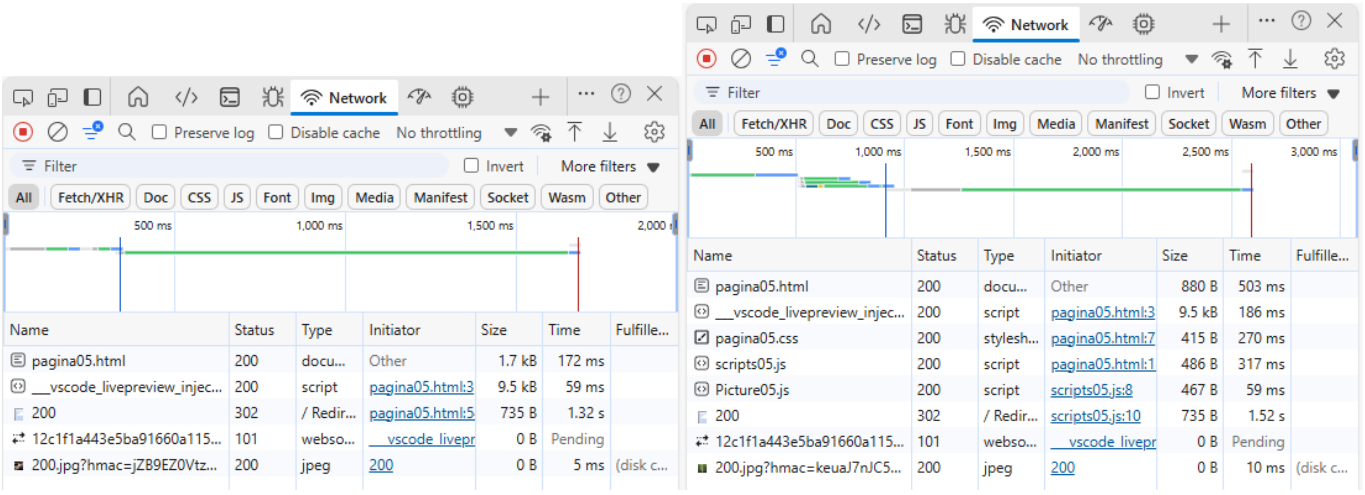
Al analizar el tráfico de red en las dos versiones mostradas, se identifica que en la primera imagen los scripts han sido integrados directamente en el archivo HTML, lo que reduce las peticiones a 4 y el tiempo de carga a 132 ms, aunque este valor también se ve influenciado por la inclusión de una hoja de estilos CSS. Por otro lado, en la segunda imagen los scripts están distribuidos en archivos externos, incrementando el número de solicitudes a 8 y eleva el tiempo de carga a 263 ms, debido al mayor número de recursos que deben ser descargados. Similar al caso anterior, la primera opción limita la escalabilidad y mantenibilidad del código a pesar de presentar un mejor rendimiento.



Página 05

La función contenida en imagen05.js crea un elemento <figure> que incluye una imagen y un pie de imagen con la leyenda "john", tomando como parámetro un ID de imagen. En picture05.js, se implementa exactamente la misma lógica, pero utilizando la palabra clave const en lugar de function, lo que responde a una práctica moderna y más segura de declaración. Finalmente, el archivo scripts05.js actúa como controlador principal, ya que desde este se invoca la ejecución de los dos scripts anteriores para que las imágenes se generen y aparezcan correctamente en la interfaz

En la primera imagen se reduce el número total de solicitudes de red a 4. El tiempo de carga promedio registrado es de 95 ms, una mejora en rendimiento debido a la menor cantidad de archivos externos requeridos. En contraste, la segunda imagen refleja la versión modular del mismo proyecto, con los tres scripts separados y vinculados desde el HTML. Esto incrementa las solicitudes a 7, incluyendo también recursos como una hoja de estilo CSS (pagina05.css) y una imagen (200.jpg). El tiempo de carga, aunque ligeramente superior (126 ms), se mantiene en un rango eficiente. Sin embargo, esta estructura modular facilita considerablemente el mantenimiento del código y su escalabilidad, permitiendo modificar o reutilizar funciones sin afectar el resto del sistema, en línea con buenas prácticas de desarrollo web.

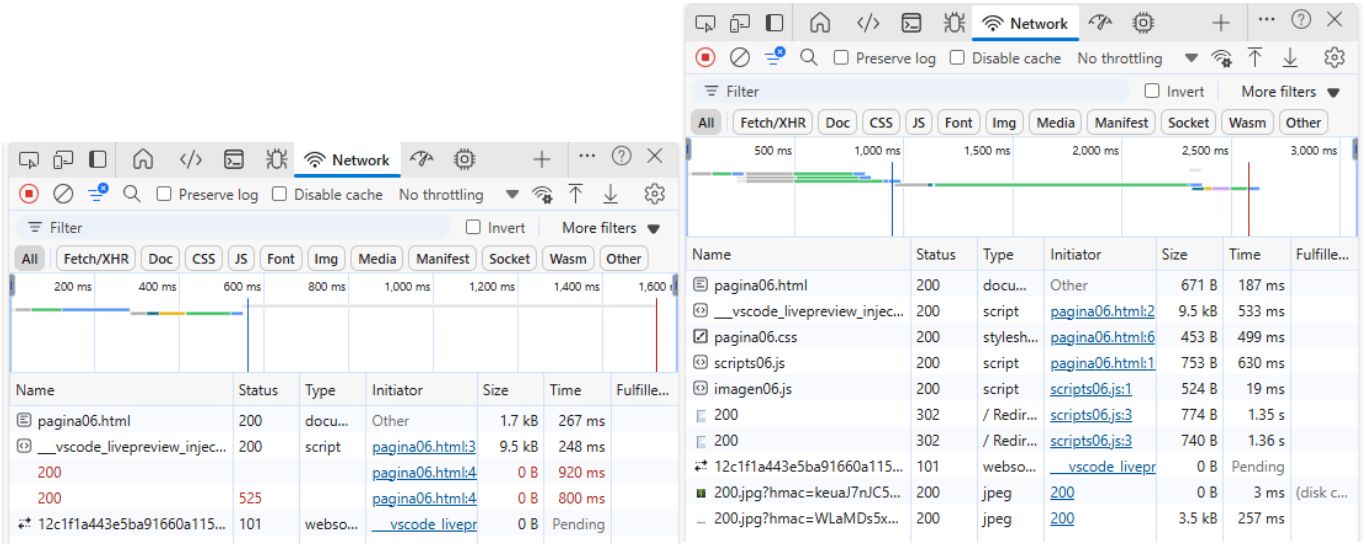


Página 06

El archivo imagen06.js implementa la creación dinámica de dos elementos <figure>, cada uno conteniendo una imagen cuya fuente se define mediante un ID proporcionado como parámetro, así como un pie de imagen cuyo texto es configurable por el usuario. Adicionalmente, este script incorpora una funcionalidad interactiva que modifica la apariencia y el color de la imagen al recibir un evento de clic. Por otro lado, scripts06.js actúa como el punto de entrada principal, encargándose de invocar la ejecución de la lógica definida en imagen06.js para renderizar las imágenes en la interfaz de usuario.

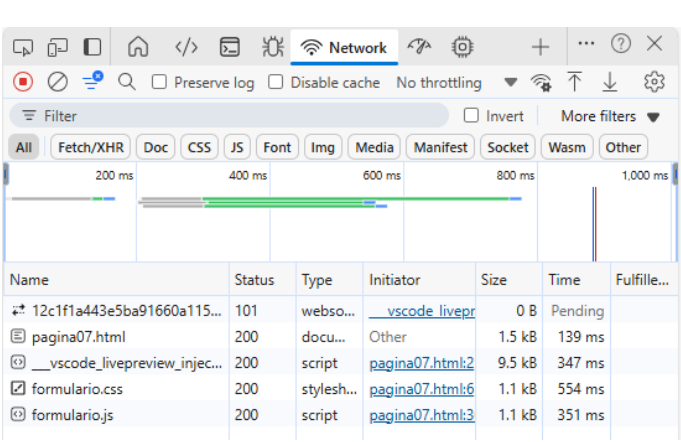
La segunda imagen muestra una carga de página con un total de 10 solicitudes de red, resultando en un tiempo de carga que se extiende hasta aproximadamente 1.36 segundos debido a la finalización tardía de ciertos recursos. Esta versión incluye la carga de múltiples scripts, una hoja de estilos y dos imágenes JPEG. En contraste, la primera imagen muestra una carga significativamente más ligera, con solo 5 solicitudes y un tiempo de carga más rápido, alrededor de 920 ms. Esta instancia omite la carga de la hoja de estilos y los scripts adicionales, así como las imágenes presentes en la primera captura.

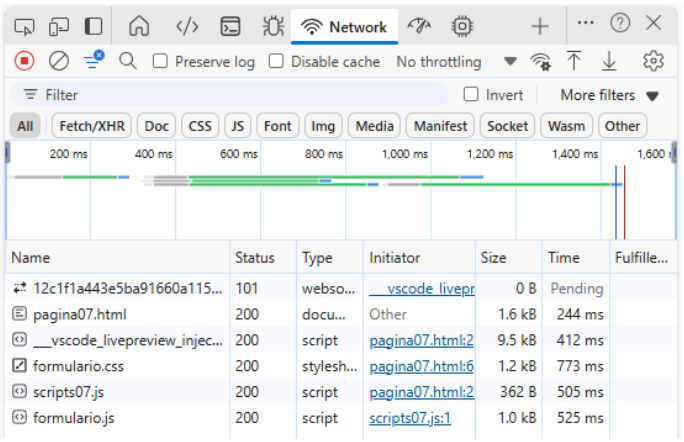
La versión modularizada, a pesar de tener más solicitudes y cargar más recursos, no presenta un tiempo de carga drásticamente superior, lo que sugiere una posible optimización en la entrega de algunos de sus activos o diferencias en el tamaño de los recursos no presentes en la primera carga. La presencia de redirecciones en la segunda carga y un error en la primera también influyen en los tiempos observados. En resumen, la primera configuración parece priorizar una carga inicial más rápida al omitir ciertos recursos, lo que podría impactar en la funcionalidad o la presentación visual completa de la página en comparación con la segunda.



Página 07

La primera imagen muestra una carga con 5 solicitudes, incluyendo HTML, un script de Live Preview, CSS (formulario.css) y el script formulario.js, completándose en ~554 ms. La segunda imagen, tras la adición de scripts07.js como main que invoca formulario.js, presenta 6 solicitudes con un tiempo de carga que se extiende a ~773 ms. La inclusión del script main (scripts07.js) incrementa ligeramente el número de peticiones y el tiempo de carga total, reflejando el costo de cargar y ejecutar la lógica adicional para la gestión del formulario.





Conclusión

Tras analizar las diferencias en los tiempos de carga de ambas versiones, se concluye que la modularización del código, aunque conlleva un mayor número de solicitudes y un ligero aumento en los tiempos de carga, ofrece ventajas significativas en términos de mantenibilidad, escalabilidad y reutilización del código. En cambio, las versiones que contienen todos los scripts y estilos directamente en el archivo HTML, logran tiempos de carga más bajos y menor cantidad de peticiones, lo cual puede ser beneficioso para aplicaciones pequeñas o estáticas donde el rendimiento inmediato es prioritario.

Sin embargo, las diferencias de tiempo entre versiones también dependen de factores como la cantidad y tipo de recursos cargados (CSS, imágenes, redirecciones), el peso de los archivos y la presencia de errores o procesos en segundo plano. Por lo tanto, aunque las versiones embebidas pueden parecer más eficientes en términos de carga, su desventaja radica en que dificultan la organización del código y su crecimiento futuro. Por otro lado, aunque las versiones modularizadas requieran una inversión inicial mayor, son más adecuadas para proyectos medianos o grandes, donde la claridad y división de responsabilidades son clave para un desarrollo sostenible.