Modularización de plataforma e-commerce en keyrus

Sergio Andrés Carrillo Muñoz

Trabajo de Grado para Optar el Título de Ingeniero de Sistemas

Director

Luis Carlos Gómez Flórez

Magíster en Ingeniería de Sistemas e Informática

Modalidad

Práctica Empresarial

Interesado

Keyrus Colombia

Universidad Industrial de Santander

Facultad de Ingenierías Físico-mecánicas

Escuela de Ingeniería de Sistemas e Informática

Bucaramanga

2024

Tabla de Contenido

Lista de tablas

Lista de figuras

Glosario

CSS: Un lenguaje de diseño que controla la apariencia y formato de un documento HTML.

DOM: En el contexto de desarrollo web, el DOM permite a los programadores acceder y manipular los elementos y contenido de una página web de manera dinámica utilizando lenguajes de programación como JavaScript.

E-commerce: El proceso de compra y venta de bienes o servicios a través de internet.

Encapsulamiento: El principio de agrupar datos y métodos en un objeto para proteger los datos y ocultar su implementación interna.

HTML: El lenguaje estándar utilizado para crear y estructurar el contenido de las páginas web.

JavaScript: Un lenguaje de programación utilizado para crear contenido interactivo en páginas web.

Módulo: Una unidad independiente de código que cumple una función específica dentro de un programa.

Refactorizar: Refactorizar es un proceso en la programación de software en el que se reestructura el código fuente de un programa sin cambiar su comportamiento externo.

Rendimiento: La eficiencia y velocidad con la que un sistema, aplicación o componente realiza una tarea específica.

Resumen

Título: Modularización de plataforma e-commerce en keyrus *

Autor: Sergio Andrés Carrillo Muñoz **

Palabras Clave: E-commerce, Modularización, Rendimiento

Descripción: En Keyrus se identificaron problemas de rendimiento en las plataformas de comercio electrónico de uno de sus clientes. Estos problemas son causados por una mala gestión en el desarrollo de nuevo código por parte de algunos desarrolladores, lo que resulta en funcionalidades confusas y sin propósitos claros. Este desorden en el código se refleja en problemas de rendimiento, escalabilidad y especialmente en los tiempos de carga. Al no estar claramente definidas algunas funcionalidades, se envían al usuario archivos innecesarios que afectan significativamente los tiempos de carga.

Para abordar esta problemática, Keyrus planificado una modularización del código. Dado el tamaño de la plataforma de comercio electrónico, se identificó la sección más problemática dentro de la plataforma junto a las principales funcionalidades que la componen y a partir de estas se reestructuro el código en módulos correspondientes a grupos de funcionalidades. Este enfoque permite que diferentes programadores trabajen en diferentes módulos de manera paralela, lo que aumenta la eficiencia y la velocidad del proceso de desarrollo. Además, se establecieron condiciones específicas para determinar qué módulos se cargarán en cada caso, lo que reduce la cantidad de archivos enviados al usuario únicamente a aquellos que sean estrictamente necesarios para el correcto funcionamiento.

---

* Trabajo de grado

** Facultad de Ingenierías Físico-mecánicas. Escuela de Ingeniería de Sistemas. Director: Luis Carlos Gómez Flórez, Magíster en Ingeniería de Sistemas e Informática.

Abstract

Title: Modularization of the e-commerce platform in keyrus *

Author: Sergio Andrés Carrillo Muñoz **

Key Words: E-commerce, Modularization, Performance

Description: At Keyrus, performance problems are identified in the e-commerce platforms of one of its clients. These problems are caused by poor management of the development of new code by some developers, resulting in confusing functionality without clear purposes. This disorder in the code is reflected in problems with performance, scalability and especially loading times. As some functionalities are not clearly defined, unnecessary files are sent to the user that significantly affect loading times.

To address this problem, Keyrus planned a modularization of the code. Given the size of the e-commerce platform, one of the most problematic sections within the platform was identified, along with the main functionalities that comprise it. From these, the module code is restructured into those corresponding to each functionality. This approach allows different programmers to work on different modules in parallel, increasing the efficiency and speed of the development process. In addition, specific conditions are specified to determine which modules will be loaded in each case, which reduces the number of files sent to the user to only those that are strictly necessary for correct operation.

* Degree Work

** Faculty of Physical-mechanical Engineering. School of Systems Engineering. Director: Luis Carlos Gómez Flórez, Master in Systems Engineering and computing.

Introducción

La adaptación a nuevas tecnologías y funcionalidades es fundamental para que las empresas sean competitivas. Para sobresalir de la competencia, se ven obligadas a diseñar soluciones que les permitan aprovechar al máximo estas nuevas tecnologías.

En este contexto, Keyrus se presenta como una oportunidad única para desarrollar habilidades en el campo del desarrollo front-end. Keyrus es una consultora internacional con sede en Levallois-Perret, Francia, especializada en el desarrollo de soluciones tecnológicas innovadoras en el ámbito digital e inteligencia de datos.

Como front-end trainee developer en Keyrus, se tiene acceso a las soluciones tecnológicas que la empresa ha diseñado para la plataforma de comercio electrónico de uno de sus clientes, así como a las diferentes herramientas tecnológicas con las que trabajan.

Durante el desarrollo de la práctica empresarial en Keyrus, se tuvo la oportunidad de abordar una problemática que está afectando el rendimiento de la plataforma de comercio electrónico mediante la modularización. A través del presente proyecto, ha sido posible poner en práctica los conocimientos adquiridos, además de las habilidades comunicativas con los demás desarrolladores.

Este documento se encuentra organizado así: en el capítulo 1 se encuentra la presentación del proyecto, seguido del marco de referencia en el capítulo 2. En el capítulo 3 se muestra el análisis de rendimiento y definición de las condiciones de carga para los módulos, seguido de la modularizarían de las funcionalidades en el capítulo 4. Finalmente, en el capítulo 5 se realizan las correcciones, validaciones y pruebas de rendimiento, seguido de las conclusiones en el capítulo final .

1. Presentación del proyecto

El constante progreso en la tecnología a nivel global ha llevado a una adaptación de los diferentes modelos comerciales a las nuevas necesidades tecnológicas. Las empresas tecnológicas son un factor clave en hacer posible esta transformación, brindando una mayor visibilidad y crecimiento digital. Keyrus es una consultora internacional especializada en proporcionar soluciones innovadoras en tecnología digital e inteligencia de datos. Asimismo, es una de las empresas tecnológicas encargadas de implementar plataformas de comercio electrónico y de impulsar la innovación y la transformación digital.

En Keyrus se identificaron problemas de rendimiento en las plataformas de comercio electrónico de uno de sus clientes. Estos problemas son causados por una mala gestión en el desarrollo de nuevo código por parte de algunos desarrolladores, lo que resulta en funcionalidades confusas y sin propósitos claros. Este desorden en el código se refleja en problemas de rendimiento, escalabilidad y especialmente en los tiempos de carga. Al no estar claramente definidas algunas funcionalidades, se envían al usuario archivos innecesarios que afectan significativamente los tiempos de carga.

Además de las implicaciones en el rendimiento, existen también implicaciones en los costos. El cliente debe asumir un costo variable que depende de la cantidad de datos transferidos desde sus servidores hasta los usuarios. Al reducir el volumen de datos innecesarios a través de la eliminación de archivos inútiles, se pueden disminuir dichos costos de transferencia. Del mismo modo, los usuarios finales también se ven afectados por estos cambios. Debido a que las comunicaciones de datos móviles son ampliamente empleadas para navegar en línea, la reducción de la cantidad de archivos innecesarios permitirá un mejor uso de este recurso y ahorro en costos asociados a su uso.

Para abordar esta problemática, Keyrus planificó una modularización del código. Dado el tamaño de la plataforma de comercio electrónico, se identificó la sección más problemática dentro de la plataforma junto a las principales funcionalidades que la componen y a partir de estas se reestructuro el código en módulos correspondientes a grupos de funcionalidades. Este enfoque permite que diferentes programadores trabajen en diferentes módulos de manera paralela, lo que aumenta la eficiencia y la velocidad del proceso de desarrollo. Además, se establecieron condiciones específicas para determinar qué módulos se cargarán en cada caso, lo que redujo la cantidad de archivos enviados al usuario únicamente a aquellos que sean estrictamente necesarios para el correcto funcionamiento.

1.1 Objetivos

1.1.1 Objetivo General

Modularizar la plataforma e-commerce en keyrus mediante el encapsulamiento del código, el uso de funcionalidades avanzadas de JavaScript (ES2015) para importar y exportar módulos de manera eficiente y la implementación de carga dinámica de módulos, contribuyendo a reducir tiempos de carga y los costos de transferencias de datos.

1.1.2 Objetivos Específicos

1. Analizar el rendimiento de la plataforma con la herramienta Lighthouse de Google, evaluando métricas de tiempo, como respuesta, carga inicial e interactividad y a partir de estos resultados seleccionar las funcionalidades a modularizar.

2. Definir condiciones específicas basadas en variables de estado del usuario dentro de la plataforma para determinar cuándo es necesario cargar cada módulo.

3. Construir los módulos correspondientes a las funcionalidades seleccionadas utilizando las tecnologías dispuestas por SAP Hybris y aprovechando las características de ECMAScript 6 para crear módulos en JavaScript, permitiendo así un encapsulamiento y organización lógica del código mejorando la eficiencia y mantenibilidad del proyecto.

4. Validar el correcto funcionamiento de los módulos implementados aprovechando los requerimientos asociados a las funcionalidades modularizadas, permitiendo de esta forma realizar pruebas de aceptación que aseguren el correcto funcionamiento de la sección.

Tabla 1.

Tabla de cumplimiento de objetivos.

Objetivos específicos	***Descripción del cumplimiento***	***Ubicación***
Analizar el rendimiento de la plataforma con la herramienta Lighthouse de Google, evaluando métricas de tiempo, como respuesta, carga inicial e interactividad y a partir de estos resultados seleccionar las funcionalidades a modularizar.	Usando la herramienta Lighthouse presente dentro del navegador de Google Chrome, se analizó el rendimiento de 4 secciones principales dentro de la plataforma. La elección de estas secciones se basó en su importancia dentro del contexto de las plataformas de comercio electrónico. Cada una de estas secciones se evaluaron de acuerdo con las métricas de rendimiento disponibles en lighthouse, así como oportunidades de mejora recomendadas. Finalmente, a partir de los resultados obtenidos en estas evaluaciones, se realizó un análisis comparativo entre las 4 secciones y se eligió el checkout como la de mayor potencial en términos de rendimiento. Específicamente, se identificó el archivo que contiene las principales funcionalidades de esta sección como el punto focal para la modularización.	Página 30 a 56
Definir condiciones específicas basadas en variables de estado del usuario dentro de la plataforma para determinar cuándo es necesario cargar cada módulo.	La sección de checkout sigue un flujo de pasos en el cual el usuario debe avanzar desde el paso 1 hasta el paso 5 para completar el proceso de checkout. En línea con esta lógica, se definieron las condiciones aprovechando la estructura HTML del checkout. Esta estructura hace uso de variables para determinar el estado del checkout en relación con cada uno de los pasos, lo que permitió identificar la ubicación del usuario en cada momento. Para obtener estas variables, se empleó el Document Object Model (DOM). Esta tecnología permitió acceder a la representación interna de la página web y extraer información en tiempo real sobre los elementos y su estado en el checkout. De esta manera, al utilizar el DOM, se tomaron decisiones informadas sobre si cargar o no un módulo en función de la posición exacta del usuario en el proceso de checkout.	Página 57 a 64 Y Página 83 a 88

De Construir los módulos correspondientes a las funcionalidades seleccionadas utilizando las tecnologías dispuestas por SAP Hybris y aprovechando las características de ECMAScript 6 para crear módulos en JavaScript, permitiendo así un encapsulamiento y organización lógica del código mejorando la eficiencia y mantenibilidad del proyecto.	Se establecieron criterios específicos que permitieron seleccionar las funcionalidades del proceso de checkout que valían la pena modularizar. Se buscó evitar modularizar funciones pequeñas y de baja complejidad que no aportaran significativamente a la mejora del rendimiento. En cambio, se enfocó en identificar aquellas áreas del código que realmente podrían beneficiarse de la modularización. Aprovechando la capacidad de exportación de código en ECMAScript 6 se logró una organización más lógica y estructurada de las funciones en archivos separados. Esta organización no solo facilitó la comprensión del código, sino que también mejoró la eficiencia al reducir la duplicación y redundancia de código innecesario. La importación de módulos se gestionó a través de una función asíncrona diseñada específicamente para que la importación fuera más sencilla. Esta función, respaldada por variables globales, garantizó que la importación de módulos se realizara de manera eficiente y sin problemas. Finalmente, la estructura de carpetas que se adoptó para organizar los módulos también fue clave para la mantenibilidad del proyecto. Cada paso del proceso de checkout tiene su propia carpeta dedicada con los módulos necesarios, lo que simplificó la localización y gestión de funciones relacionadas con cada etapa específica del flujo de trabajo. Esto no solo mejora la eficiencia en el desarrollo, sino que también facilita las futuras actualizaciones y modificaciones.	Página 65 a 91
Validar el correcto funcionamiento de los módulos implementados aprovechando los requerimientos asociados a las funcionalidades modularizadas, permitiendo de esta	Dado que las funcionalidades modularizadas fueron desarrolladas por otros miembros del equipo, Keyrus mantiene un control de la información relacionada con cada uno de estos desarrollos, la cual se almacena en forma de historias de usuario. Estas historias usuario resultan valiosas tanto durante la fase de desarrollo como para comprender los objetivos de las funcionalidades por parte de futuros desarrolladores. En el contexto de este proyecto, se identificaron las	Página 92 a 117

forma realizar pruebas de aceptación que aseguren el correcto funcionamiento de la sección.	historias de usuario asociadas a las funcionalidades modularizadas y específicamente los criterios de aceptación que contienen. Con esta información como base, se definieron casos de aceptación concretos que permitieron validar que las funcionalidades modularizadas siguieran manteniendo su comportamiento inicial, solucionando las inconsistencias antes de proceder a realizar el análisis de los resultados.	

2. Marco Referencial

La modularización es una técnica altamente efectiva que aborda la problemática del rendimiento y la mantenibilidad del código. En esta sección, se resumirán todos los conceptos clave y fundamentos teóricos necesarios para comprender qué es la modularización, qué implica y cuáles son los beneficios de su aplicación. Además, se destacarán consideraciones especialmente importantes para lograr una modularización correcta aprovechando las características incluidas en ECMAScript 6 de JavaScript.

2.1 Modularización

La modularización es un proceso fundamental en la programación que consiste en dividir un programa en módulos más pequeños y manejables. Estos módulos pueden contener funciones, variables y otros elementos necesarios para la tarea que realizan, y pueden ser desarrollados y probados por separado antes de ser integrados en el sistema completo.

Entre los beneficios de la modularización destacan (Luis Reynoso, 2022):

- **Construcción del código:** Permite que los equipos de programadores trabajen en módulos independientes, lo que facilita la colaboración y reduce el riesgo de errores.

- **Depuración del código:** Al dividir el programa en módulos, se puede aislar el problema y corregirlo de manera más eficiente.

- **Lectura del código:** Cada módulo se define con un nombre específico, lo que permite identificar fácilmente su función y propósito. Además, al tener módulos más pequeños, el código se vuelve más claro y fácil de leer, lo que facilita su mantenimiento.

- **Modificación del código:** Un pequeño cambio en los requerimientos sólo implica un cambio en uno o pocos módulos, lo que agiliza el proceso de desarrollo y evita errores innecesarios.

- **Eliminación de redundancia de código:** Al dividir el programa en módulos, se pueden localizar operaciones que ocurren en diferentes lugares del programa y agruparlas en un solo módulo, reduciendo la cantidad de código repetido y mejorando la eficiencia del programa.

Como se puede observar, la modularización del código trae consigo grandes beneficios. Sin embargo, se debe tener en cuenta otro factor que va más allá de la creación de dichos módulos, y es el cómo se cargarán estos. Los módulos se pueden cargar automáticamente al inicio, pero esto afectará considerablemente al rendimiento. Para solucionar este problema, a continuación, se definirá como cargar dichos módulos de una forma óptima.

2.1.1 Módulos en ECMAScript 6

En JavaScript, los módulos se han utilizado durante mucho tiempo, pero su implementación se realizaba a través de librerías y no estaban disponibles directamente en el lenguaje. Sin embargo, con la llegada de ECMAScript 6, los módulos se implementaron por primera vez directamente en el lenguaje (Axel Rauschmayer, 2018).

El estilo de los módulos utilizado en ECMAScript 5 y versiones anteriores, que era usado a través de librerías, fue en cierta medida integrado en ECMAScript 6. Algunas de estas características incluyen:

- Cada módulo es una pieza de código que se ejecuta una vez que se carga.
- En el código de un módulo pueden existir declaraciones, pero por defecto estas permanecen locales al módulo a menos que se declaren como exports, lo que permite que otros módulos las importen.
- Un módulo puede importar elementos de otros módulos.
- Los módulos son únicos, incluso si un módulo es importado múltiples veces, solo existe una única instancia del módulo.

Uno de los principales inconvenientes al utilizar módulos de esta manera es que la importación es estática, esto quiere decir que todos los módulos se cargan al inicio del archivo y no cuando realmente son necesarios, para abordar ese problema javascript cuenta con una funcionalidad que permite importar estos módulos dinámicamente evitando cargar cosas innecesarias. A continuación, se abordará el tema de la carga dinámica de módulos.

2.1.2 Carga dinámica de módulos

La carga dinámica (import dinámico) es una funcionalidad de JavaScript que permite especificar el nombre del archivo que se requiere importar. Pero a diferencia del import estático, el archivo no se carga automáticamente al principio. Sino que se carga solo cuando se llega a la parte del código donde se utiliza (LenguajeJS, 2022).

Esta característica es especialmente útil para la optimización y el rendimiento, ya que permite no importar y procesar archivos con código Javascript hasta que se cumpla una cierta condición, evento o acción. Además, esta funcionalidad permite retrasar la descarga, procesamiento y ejecución de código Javascript por parte del navegador hasta que sea necesario, lo que puede mejorar significativamente la eficiencia del sitio web. Por lo tanto, se puede incluir el import dinámico en condicionales, funciones o lógica diversa para mejorar el rendimiento y la eficiencia de un proyecto.

Adicionalmente para lograr una buena modularización, es fundamental tener totalmente claro qué secciones dentro de la plataforma son las que requieren una mayor prioridad. A continuación, se describe una herramienta que facilita el llevar a cabo esta tarea con éxito.

2.2 Chrome Dev Tools

Las Chrome Dev Tools son un conjunto de herramientas y funcionalidades integradas en el navegador Google Chrome que ayudan a los desarrolladores web a depurar, analizar y optimizar sitios web. Estas herramientas se encuentran directamente disponibles en el navegador y ofrecen una amplia gama de características para la inspección de sitios web. A continuación, se describirán algunas de las herramientas que se utilizan en el desarrollo del proyecto (Chrome Developers, 2023).

2.2.1 Lighthouse

Lighthouse es una herramienta desarrollada por Google que permite realizar pruebas automatizadas en sitios web y obtener un informe detallado sobre el rendimiento y la optimización del sitio en diferentes áreas clave. Al utilizar Lighthouse, se pueden identificar rápidamente las áreas que requieren mejoras y, de esta forma, centrarse en la modularización de las funcionalidades requeridas.

Lighthouse utiliza diferentes métricas que nos permiten analizar el rendimiento de la página web, para este caso específico se tomarán en cuenta principalmente las siguientes métricas (Chrome Developers, 2023):

- **Largest Contentful Paint (LCP):** Reporta el lapso de tiempo que tomó para procesar la imagen o el bloque de texto más extenso visible en la pantalla en comparación con el instante de inicio de carga de la página.

- **First Contentful Paint (FCP):** Mide cuánto tiempo le toma al navegador mostrar la primera parte del contenido DOM después de que un usuario navega en la página.

- **Total Blocking Time (TBT):** Mide el tiempo total durante la cual la página se encuentra inactiva y no responde a ninguna interacción del usuario, ya sea mediante clics del mouse, toques en pantalla o pulsaciones de teclado.

- **Time to Interactive (TTI):** TTI mide cuánto tarda una página en volverse completamente interactiva. Las siguientes características definen cuándo una página se considera completamente interactiva:

    1. Se muestra contenido útil, se mide por la primera pintura con contenido.
    2. La capacidad de respuesta de la página a las interacciones del usuario es de 50 milisegundos.
    3. Los controladores de eventos se aplican a los elementos más visibles de la página.

- **Speed Index:** Mide qué tan rápido se muestra visualmente el contenido durante la carga de la página.

- **Cumulative Layout Shift:** El Cambio de diseño acumulado mide el movimiento de los elementos visibles dentro del viewport.

Aparte de examinar estos aspectos, Lighthouse también proporciona recomendaciones para mejorar el rendimiento del sitio web. Estas recomendaciones incluyen diversas opciones

de optimización que permiten reducir el tiempo de carga, como la compresión de imágenes, la eliminación de recursos que podrían estar obstaculizando la visualización de la página o como para el caso de este proyecto, la optimización de código JavaScript. Además, el análisis del rendimiento abarca otros aspectos, como recomendaciones sobre diversos puntos que también influyen en el rendimiento, entre ellos se encuentran (IONOS, 2021):

- Utilizar formatos de imagen que requieran menor espacio de almacenamiento.
- Comprimir siempre que sea posible los textos y JavaScript.
- Optimizar la memoria caché.
- Evitar redireccionamientos y cargar las consultas más importantes previamente.
- Usar contenidos animados en formatos modernos que requieren poco espacio.
- Reducir al mínimo el volumen de datos que se utilizan en la página web.

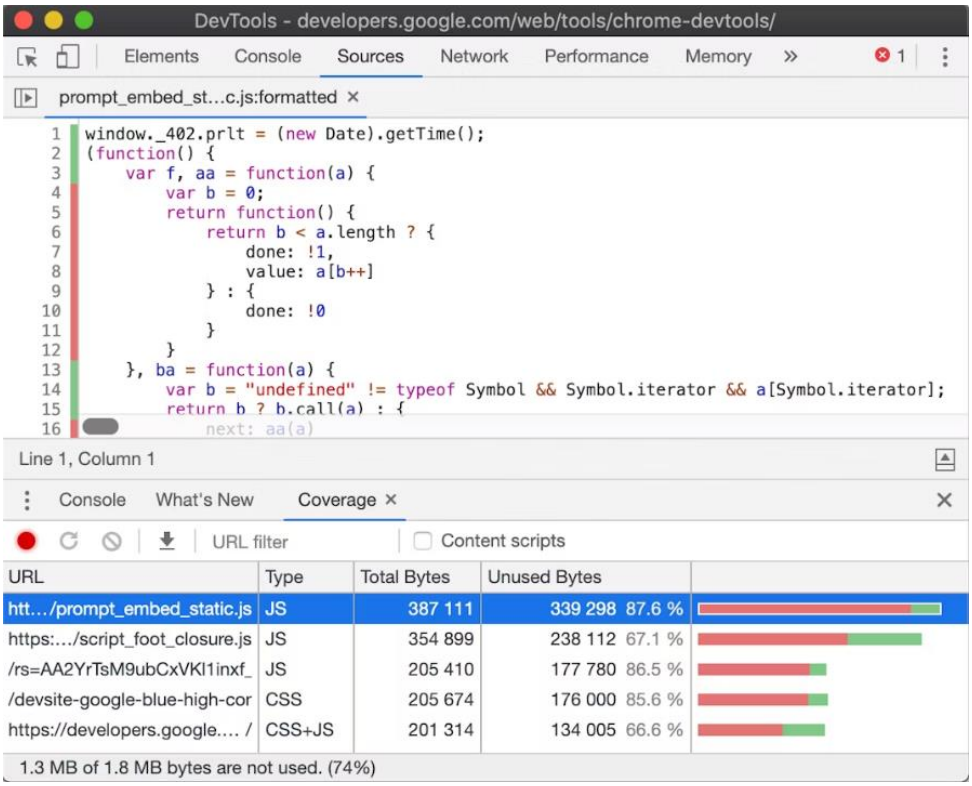
En resumen, Lighthouse es una herramienta fundamental para mejorar la calidad de cualquier proyecto de desarrollo web y puede ayudar a garantizar que el sitio web cumpla con los estándares del mercado actual.

2.2.2 Coverage

Coverage es una herramienta de cobertura disponible en las Chrome DevTools. Esta herramienta ayuda a medir el alcance del código JavaScript y css utilizado por una página web. Coverage proporciona información valiosa sobre qué partes o secciones del código se están ejecutando y cuáles no lo están, además de otros datos útiles que ayudan a comprender mejor los archivos cargados (Chrome Developers, 2023). En la **Figura 1**, se da un ejemplo del funcionamiento de la herramienta coverage.

Figura 1.

Análisis de cobertura de código con la herramienta coverage.



Nota. Tomado de Chrome Developers. (2023)

La herramienta Coverage fue especialmente útil para el desarrollo del proyecto, ya que permite identificar fácilmente las funcionalidades que requieren una mayor prioridad. Además, los resultados obtenidos en el Coverage sirven para realizar una comparación entre el estado inicial y final del proyecto, lo que permite llegar a conclusiones sobre las implicaciones reales del mismo y los resultados obtenidos en comparación con los esperados.

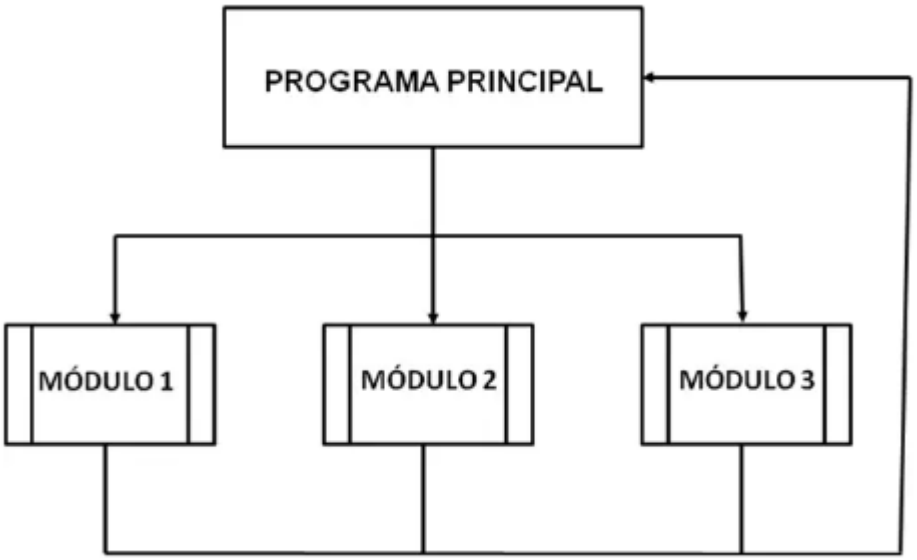
2.3 Encapsulamiento

Realizar un programa sin seguir una técnica clara y adecuada de programación comúnmente resulta en funcionalidades complejas de entender, lo que hace que su manejo y depuración sean tareas tediosas. Como se muestra en la **Figura 2**, la programación estructurada es una forma de mejorar el manejo del código al dividir el programa en segmentos o módulos

más manejables. Al realizar una correcta partición del código, se vuelve más sencillo e intuitivo comprender lo que cada módulo debe hacer (Vega. J, 2018).

Figura 2.

Lógica de la programación estructurada



Nota. Tomado de Nimble (2022)

Al agrupar las funcionalidades en un módulo, se resalta la relación existente entre ellas al separarlas del resto del programa. Esta ocultación de los detalles se conoce como encapsulación. Esto significa que, para utilizar las funciones incluidas en el módulo, no es necesario conocer los detalles de su implementación ni de ninguna de las otras funcionalidades presentes dentro del módulo. Solo es necesario tener conocimiento del archivo en el que se declararon las funciones.

La independencia modular, además de lograr el aislamiento a través del encapsulamiento, mejora el rendimiento humano y el desarrollo de módulos de forma paralela, así como la reutilización de código.

Finalmente, es fundamental comprender las herramientas tecnológicas utilizadas en el desarrollo de estas funcionalidades para lograr una modulación adecuada. A continuación, se describirán las principales tecnologías necesarias para el desarrollo.

2.4 Descripción de herramientas tecnológicas:

Durante el proceso de desarrollo de un sitio web, se utilizan diversos lenguajes, como HTML, CSS y Javascript. Estos lenguajes son fundamentales para construir la estructura principal, el estilo de los componentes y las funcionalidades presentes dentro de la página. En particular, en este proyecto se utilizan herramientas de alto nivel de abstracción, como JSTL, Less y Javascript, que permiten la integración con los diferentes componentes de la plataforma SAP Hybris. Estas herramientas, disponibles en el proyecto, permiten una mayor eficiencia en el desarrollo y aseguran la calidad del sitio web en su conjunto

2.4.1 SAP Hybris

Es una plataforma que proporciona herramientas para el comercio electrónico y el marketing digital, ayudando a las empresas a vender sus productos en línea y a mejorar su relación con los clientes. SAP Hybris ofrece una amplia gama de soluciones, desde la gestión de la experiencia del cliente hasta la gestión de precios y la gestión de contenido, entre otros. Además, es una plataforma altamente personalizable, lo que significa que las empresas pueden adaptarla a sus necesidades específicas. SAP Hybris también incluye soluciones para el comercio omnicanal, lo que permite a los clientes comprar en línea, en tiendas físicas y en otros canales de venta (SAP Hybris, SAP,2021). En general, SAP Hybris es una plataforma muy completa y poderosa para ayudar a las empresas a mejorar la experiencia del cliente y aumentar las ventas.

2.4.2 JSP (JavaServer pages) y JSTL (JavaServer Pages Standard Tag Library)

JSP y JSTL son tecnologías utilizadas en la creación de aplicaciones web en Java. JSP permite incrustar código Java en páginas web para hacerlas más dinámicas y personalizables, mientras que JSTL es una librería de etiquetas personalizables que simplifica la creación de aplicaciones web, las etiquetas JSTL están organizadas en 4 librerías (OpenWebinars, 2017):

1. **core** Comprende las funciones script básicas como loops, condicionales, y entrada/salida.
2. **xml:** Comprende el procesamiento de xml
3. **fmt:** Comprende la internacionalización y formato de valores como moneda y fechas.
4. **sql:** Comprende el acceso a base de datos

2.4.3 LESS

Es un lenguaje de hojas de estilo que permite crear variables, operaciones y funciones si es necesario. LESS se conoce también como un preprocesador de CSS, ya que los estilos definidos en este lenguaje deben compilarse en archivos en formato Less para obtener su traducción en formato CSS. De esta forma, el navegador puede leer los archivos CSS sin ningún problema. El uso de LESS permite reciclar código a lo largo del desarrollo gracias al uso de variables y funciones (Jackson. B, 2023).

Para que los demás colaboradores del proyecto puedan entender el código de estilos, se utiliza la metodología BEM (Bloque, Elemento, Modificador). BEM es una metodología ágil de desarrollo basada en componentes cuyo objetivo es dividir la

interfaz de usuario en bloques independientes para que sean escalables y reutilizables. Esta metodología propone un estilo descriptivo para nombrar correctamente las clases de las etiquetas utilizadas en la página. De esta manera, es más fácil ubicar donde se definen los estilos para cada elemento en el repositorio de trabajo.

2.4.4 JavaScript

Para que los diversos elementos de la página interactúen correctamente, es necesario contar con un lenguaje adecuado. Aunque JSP y LESS nos permiten acceder a funcionalidades avanzadas en comparación con otros lenguajes de etiquetas y estilos, no son suficientes por sí solos. JavaScript es un lenguaje de programación ligero, interpretado y compilado en tiempo de ejecución. Es el único lenguaje de programación que funciona en los navegadores y se utiliza como complemento de lenguajes de maquetado como HTML y CSS para la creación de páginas web (MDN contributors, 2022).

2.5 Metodología ágil: SCRUM

Scrum es una metodología en la cual se aplican un conjunto de buenas prácticas para trabajar en equipo, colaborativamente y obtener el mejor resultado posible de un proyecto.

En Scrum, un proyecto se ejecuta en ciclos temporales cortos y de duración fija (iteraciones que normalmente son de 2 semanas) como se presenta en la **Figura 3**, los cuales se les denomina como sprints. Cada sprint tiene que proporcionar un resultado completo.

Al aplicar la metodología Scrum, se deben llevar a cabo estas actividades (Proyectosagiles, 2022):

- **Planificación del sprint:** Se seleccionan los requerimientos a desarrollar y se elabora una lista de tareas necesarias para llevar a cabo el desarrollo de estos.

- **Ejecución del sprint:** El equipo realiza una reunión de sincronización todos los días. Se inspecciona el trabajo que los integrantes están realizando, como por ejemplo, el progreso que cada uno lleva hacia el objetivo de la iteración, los obstáculos que pueden presentarse a medida que el proyecto vaya avanzando. Todo esto para poder corregir o incentivar problemas y soluciones que permitan cumplir con la previsión de objetivos al final de la iteración.

- **Inspección y adaptación:** El equipo presenta los requerimientos completados en la iteración. En función de los resultados, se definen los requerimientos de la siguiente iteración y se realiza una retrospectiva acerca de cómo ha sido la manera de trabajar, definir los problemas que aparecieron y tratar de eliminarlos para aumentar la productividad.

Figura 3.

Proceso de Scrum.

Nota. Tomado de Nimble. (2022)

3. Análisis de rendimiento en la plataforma y definición de las condiciones de carga para los módulos

Este proyecto tiene como objetivo mejorar el rendimiento de la plataforma de comercio electrónico Alkomprar a través de la modularización. Al tratarse de una modularización, no solo se busca mejorar los tiempos de carga, sino también la escalabilidad y la comprensión del código, lo cual resulta muy valioso tanto para los desarrolladores al momento de solucionar problemas en el código como para el cliente que notara una mejoría en los tiempos que lleva integrar nuevos proyectos.

Idealmente, modularizar todo el código de la plataforma de comercio electrónico de Alkomprar sería deseable y tendría un impacto significativo en el rendimiento. Sin embargo,

esto es poco realista, teniendo en cuenta que la plataforma y su estructura son muy grandes. Por esta razón, en el desarrollo de este proyecto, el primer paso consistió en identificar en qué sección de toda la plataforma de Alkomprar se debería implementar la modularización. Lo anterior, siendo consistente con el desarrollo del objetivo específico 2, que se puede ver en la página 57.

En el siguiente apartado se abordará este tema en detalle, definiendo qué sección se identificó como la más problemática, así como las funcionalidades específicas dentro de la sección que fueron objeto de modularización.

3.1 Análisis de rendimiento por secciones

Con el objetivo de no abordar más de lo necesario se llevó a cabo un análisis del rendimiento de la página web en algunas de las secciones consideradas generalmente las más importantes en plataformas de comercio electrónico, logrando de esta forma entender el estado actual de los tiempos de carga, la cantidad de archivos cargados y rendimiento para cada una de las secciones por separado.

Debido a que en una plataforma e-commerce existen muchas secciones sobre las cuales poder llevar a cabo el proceso de modularización se tomaron en cuenta las secciones clave tomadas en el benchmark UX hecho por el Bayard Institute (Baymard Institute, 2023) los cuales se pueden ver en la **Figura 4**, donde se analizan los sitios e-commerce más importantes del mundo.

Figura 4.

Benchmark UX hecho por Bayard Institute en secciones clave de plataformas e-commerce.

Nota. La figura muestra las secciones más relevantes en plataformas de comercio electrónico por parte del Bayard Institute, además de un análisis de la experiencia de usuario en estas. Tomado de Baymard Institute. (2023)

En el benchmark se hicieron pruebas a las secciones consideradas más importantes de un sitio web de comercio electrónico, entre las cuales para este proyecto se tomaron las que mejor se relacionaban al proyecto de alkomprar:

- **Homepage:** Es la página principal del sitio web y suele ser la primera impresión que los visitantes tienen de una tienda en línea.
- **Product Page:** Cada producto suele tener su propia página dedicada, donde se muestra información detallada, como descripción, características, precio, opciones de personalización, reseñas de clientes y botones para agregar al carrito de compras.

- **Cart:** Esta sección permite a los usuarios revisar y editar los productos que han seleccionado para comprar.

- **Checkout:** Es la sección donde los clientes completan su compra. Aquí ingresan la información de envío, seleccionan el método de pago y revisan los detalles de la orden antes de confirmar la compra.

Para facilitar el análisis de cada una de las secciones, se hizo uso de Lighthouse, una herramienta que permite tomar decisiones apropiadas e informadas. Lighthouse cuenta con diferentes categorías que permiten un análisis completo de la página web. Para el caso de este proyecto, el análisis se centró específicamente en el apartado de rendimiento, el cual basa su análisis en métricas como *First Contentful Paint (FCP), Speed Index, Largest Contentful Paint (LCP), Time to Interactive (TTI), Total Blocking Time (TBT) y Cumulative Layout Shift.*

A continuación, se podrá observar el análisis realizado para cada una de las secciones mencionadas por separado. Además, se buscará contextualizar sobre el funcionamiento y las características importantes para cada una de estas secciones dentro de la plataforma, adicionalmente se agregó una vista de cada sección que permitirá una mayor claridad sobre cómo son realmente estas secciones dentro la plataforma.

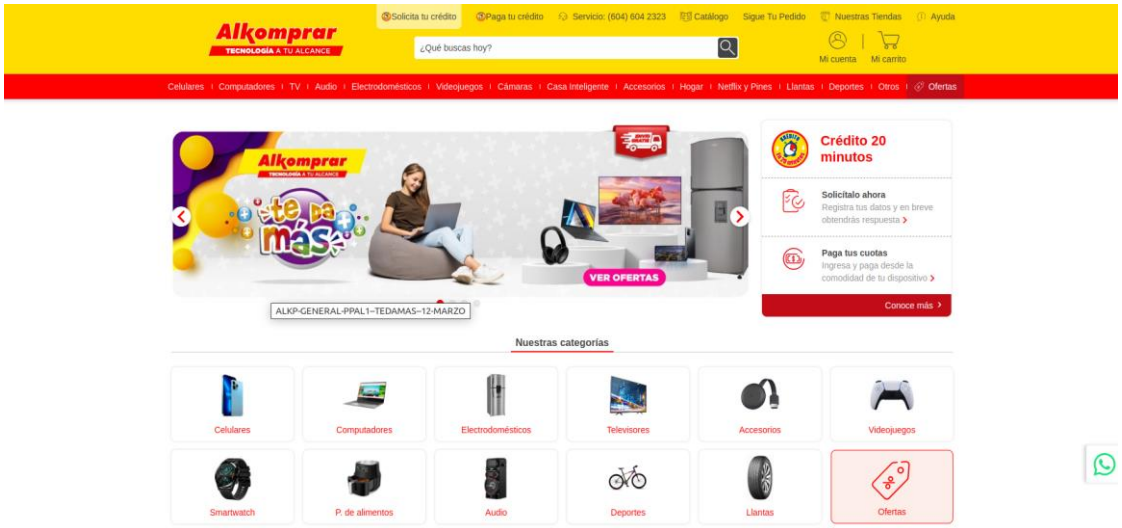
3.1.1 Home Page

Corresponde a la primera vista del usuario al entrar en la página web y es una de las principales secciones dentro de la plataforma, por lo cual esta sección es especialmente interesante y prioritaria al momento de realizar algún desarrollo que permita mejorar su rendimiento. Esta sección carga una gran cantidad de contenido multimedia el cual corresponde a las imágenes de los productos y elementos relacionados.

A continuación, en la ***Figura 5***, se presenta una vista del home en el ambiente de producción del proyecto:

Figura 5.

Vista del home page de la plataforma e-commerce de alkomprar.

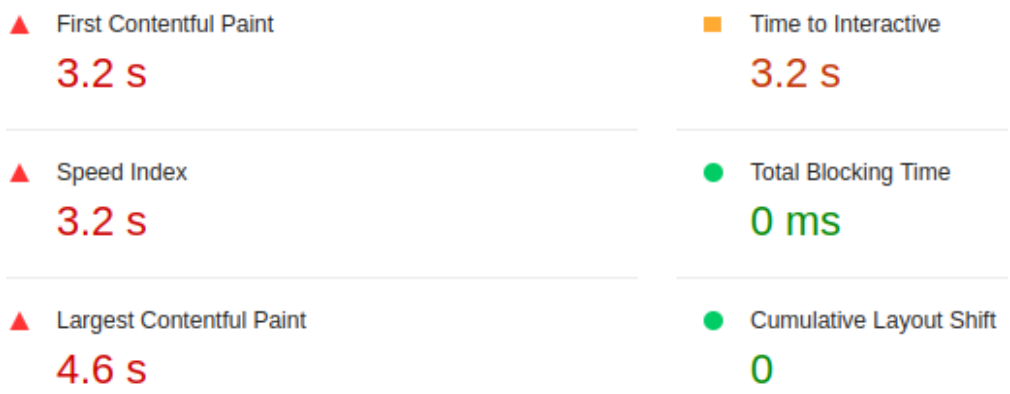


Nota. Tomado de Alkomprar (2023).

El análisis realizado con Lighthouse en las secciones se llevó a cabo en un entorno local con el objetivo de tener un mayor control sobre los cambios realizados y facilitar el análisis de los resultados. En la **Figura 6** se presentan los resultados de rendimiento para cada una de las métricas.

Figura 6.

Resultados métricos de lighthouse para el home.



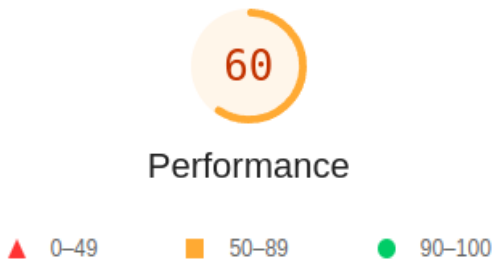
Nota. Tomado de Google Chrome, lighthouse (2023).

Se pueden observar resultados no muy positivos, especialmente en la métrica LCP la cual se relaciona con la carga de las imágenes y textos, resultado que no es de extrañar para la página principal por la cantidad de imágenes y textos que se cargan en comparación a las otras secciones.

A continuación, en la **Figura 7**, se presenta el puntaje de rendimiento general asignado a esta sección :

Figura 7.

Puntuación de performance dada por lighthouse para la sección del home



Nota. Tomado de Google Chrome, lighthouse (2023).

Adicionalmente al resultado general, lighthouse brinda ciertas oportunidades que ayudarán a mejorar las métricas y por consiguiente los tiempos de carga dentro de la página. A continuación, en la **Figura 8** se presentan estas oportunidades y el ahorro estimado en segundos:

Figura 8.

Oportunidades dadas por lighthouse para mejorar el performance en el home page.

Se pueden observar oportunidades relacionadas con JavaScript, las cuales son especialmente importantes debido a la naturaleza del proyecto. Además, se puede notar que la oportunidad de compresión de texto sería una de las que proporcionarían un mayor ahorro, aunque esta no está relacionada y probablemente no se beneficiará del proceso de modularización.

3.1.2 Product Page (PDP)

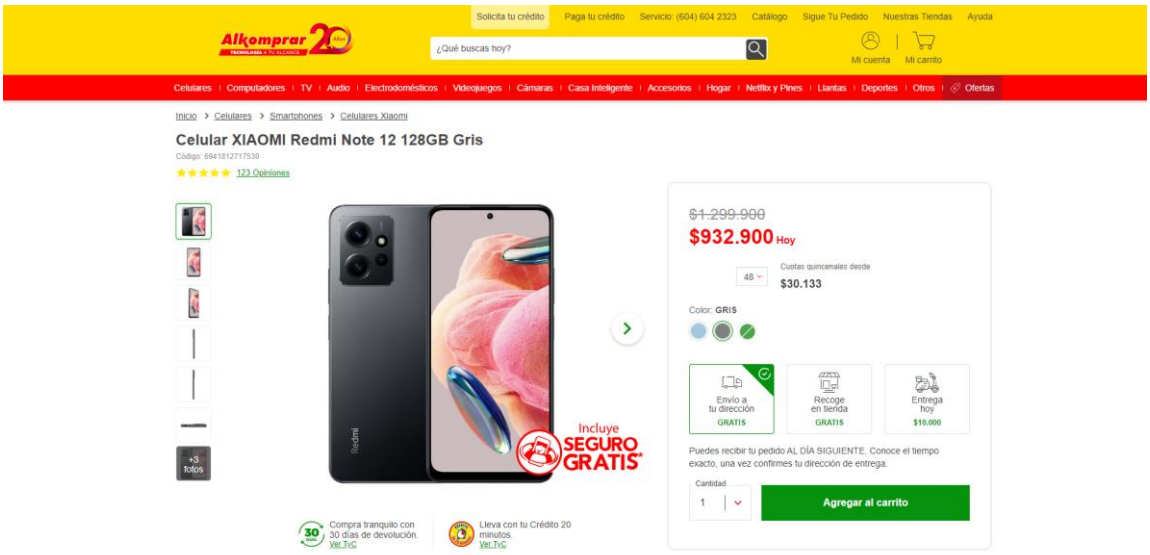
En esta sección, el usuario puede visualizar toda la información del producto seleccionado y elegir uno de los métodos de envío para avanzar en el proceso de compra. En el momento del desarrollo del proyecto, esta sección estaba sujeta a cambios que buscaban

añadir nuevas funcionalidades, además de cambios significativos relacionados con la estructura completa de la sección. Por esta razón, los resultados obtenidos para las métricas pueden no reflejar la situación actual para esta sección de la plataforma.

A continuación, en la **Figura 9**, se presenta una vista de la sección:

Figura 9.

Vista del product page de la plataforma e-commerce de alkomprar.

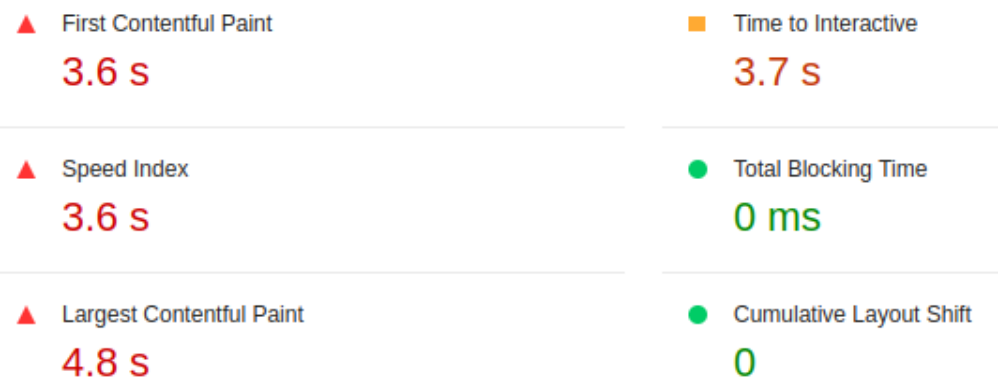


Nota. Tomado de Alkomprar (2023).

Al realizar un análisis con lighthouse localmente se obtienen los resultados de la **Figura 10**.

Figura 10.

Resultados métricos de lighthouse para la product page.



Nota. Tomado de Google Chrome, lighthouse (2023).

La sección presenta resultados muy similares a los obtenidos en la página de inicio. La métrica LCP continúa siendo la que presenta peores resultados, lo que podría inferir que la forma en que se cargan las imágenes y los textos dentro de la plataforma no es eficiente. Posterior a la métrica LCP, se encuentra la métrica TTI como la siguiente en términos de resultados deficientes. Esta métrica mide el tiempo en el que el usuario puede interactuar con los elementos de la sección. Por lo tanto, una demora significativa podría llevar al usuario a abandonar y optar por otras páginas de comercio electrónico más eficientes.

A continuación, en la **Figura 11**, se presenta el puntaje de rendimiento general asignado a esta sección.

Figura 11.

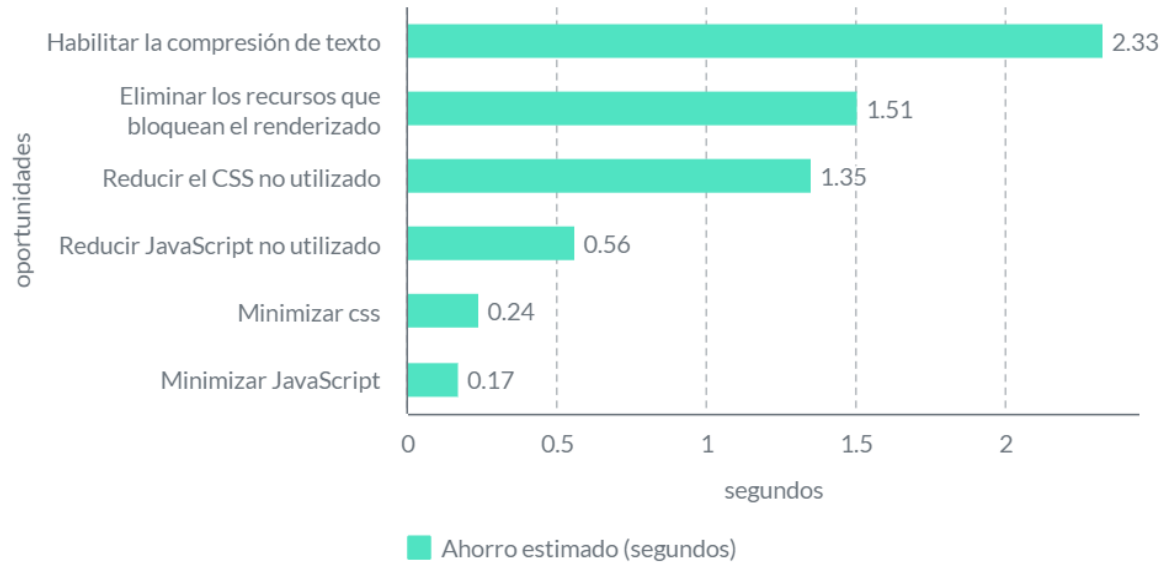
Puntuación de performance dada por lighthouse para la sección del product page.

Nota. Tomado de Google Chrome, lighthouse (2023).

Se observa una puntuación muy similar a la obtenida para la sección anterior, aunque esta sección presenta un rendimiento un poco menor. Las oportunidades para mejorar el tiempo de carga en esta sección se presentan en la **Figura 12**.

Figura 12.

Oportunidades dadas por lighthouse para mejorar el performance en la product page.



Se puede observar que la compresión de texto es una oportunidad que se presenta nuevamente como la mejor, por lo cual, se podría asumir que se trata de un aspecto común en todas las secciones dentro de la plataforma, el cual valdría la pena analizar más a fondo. Por otro lado, en cuanto al código JavaScript, se puede observar que la cantidad de tiempo ahorrado para esta sección es levemente mayor que para la página de inicio.

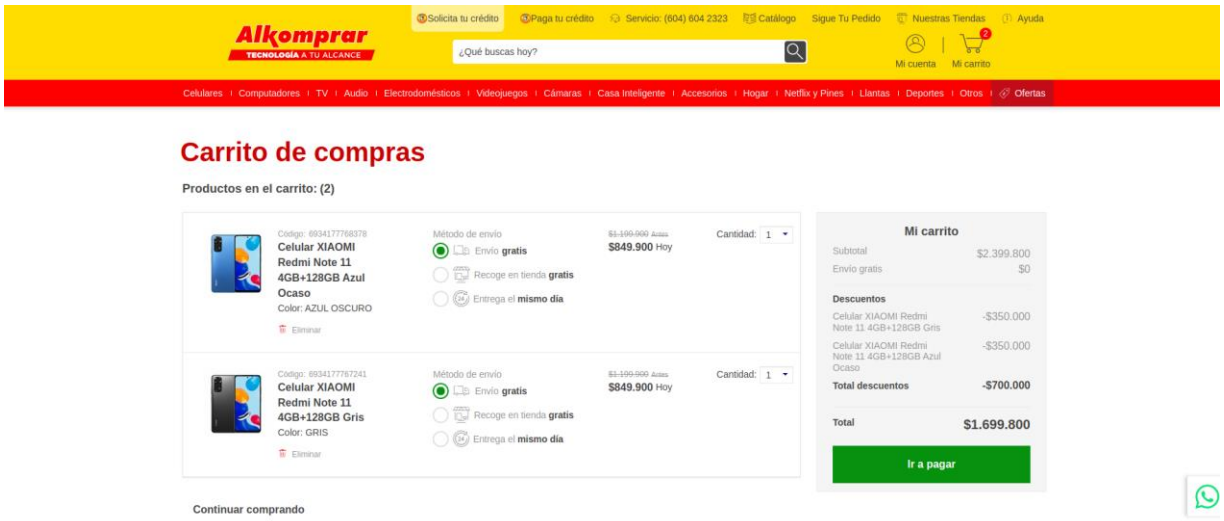
3.1.3 Cart

Esta sección se encarga de mostrar todos los productos que el usuario ha agregado al carrito. Al momento del análisis de rendimiento, el equipo de trabajo se encontraba en la etapa final de un proceso de cambio en el funcionamiento del carrito. Este proceso se basó en utilizar la técnica AJAX para lograr actualizar cada uno de los ítems sin necesidad de recargar la página. Debido a esto, el rendimiento en esta sección es mayor que en las demás y no se presentó como la mejor opción para aplicar el proceso de modularización.

A continuación, en la **Figura 13**, se presenta una vista de la sección:

Figura 13.

Vista del carrito de la plataforma e-commerce de alkomprar.

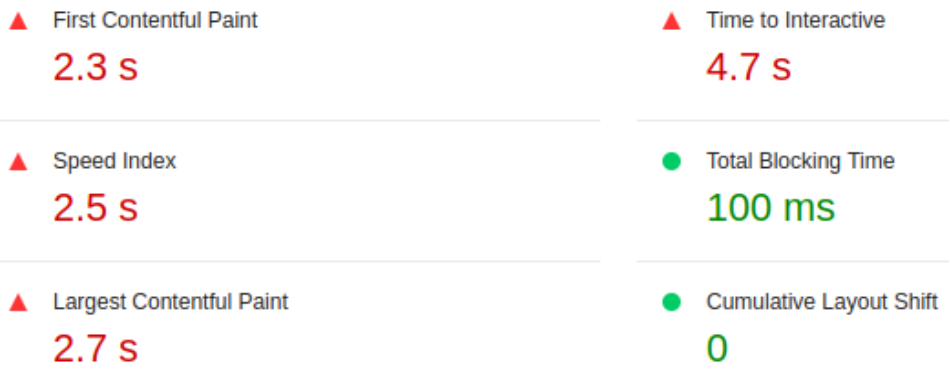


Nota. Tomado de Alkomprar (2023).

Al realizar un análisis con lighthouse localmente se obtienen los resultados de la **Figura 14**.

Figura 14.

Resultados métricos de lighthouse para el carrito.



Nota. Tomado de Google Chrome, lighthouse (2023).

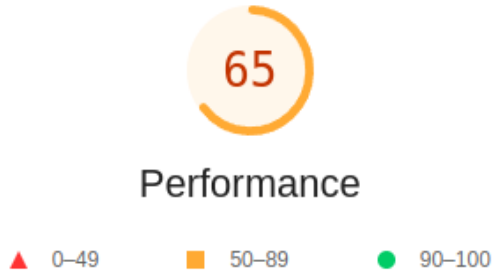
En esta sección, se puede observar que para la métrica TBT se presenta un tiempo mayor que en las secciones anteriores. Esta métrica corresponde al tiempo en que el usuario

puede interactuar con la página, por lo que un tiempo muy alto provocará que los usuarios no tengan una buena experiencia y posiblemente eviten realizar futuras compras en la plataforma.

A continuación, en la **Figura 15**, se presenta el puntaje de rendimiento general asignado a esta sección.

Figura 15.

Puntuación de performance dada por lighthouse para la sección del carrito.

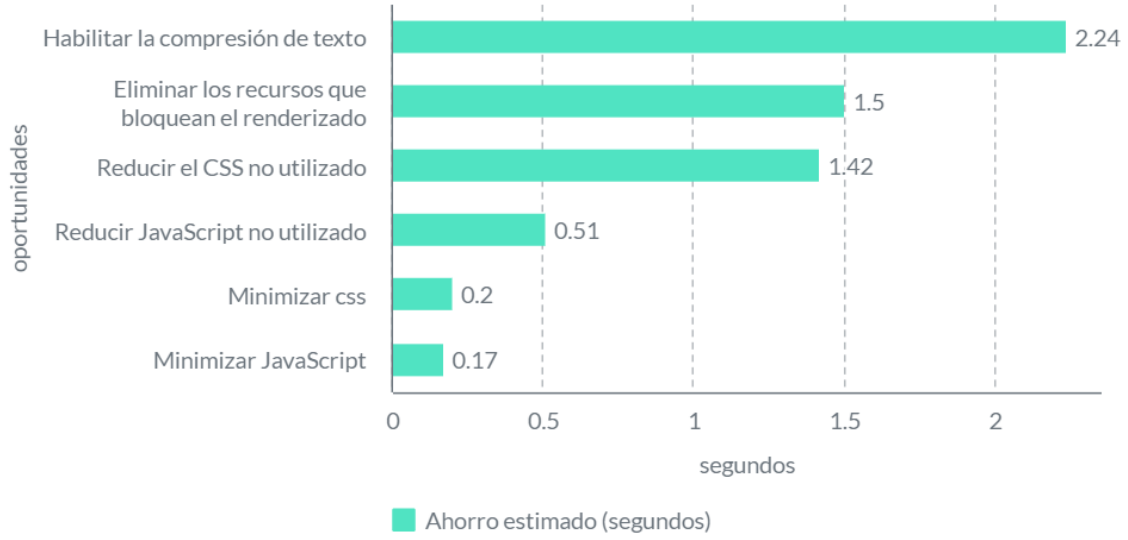


Nota. Tomado de Google Chrome, lighthouse (2023).

Rendimiento general por encima de los anteriores resultados esperados debido a que se observan mejoras en cada una de las métricas en comparación a las secciones anteriores. Las oportunidades para mejorar el tiempo de carga en esta sección se presentan en la **Figura 16**.

Figura 16.

Oportunidades dadas por lighthouse para mejorar el performance en el carrito.



Como se viene observando, la compresión de texto sigue siendo la acción que ahorraría más tiempo de carga. Por otro lado, en general, se puede observar que las oportunidades son muy similares para cada una de las secciones. Por lo tanto, si el proceso de modularización se considera que es viable con resultados satisfactorios y relevantes, podría considerarse realizar una modularización más grande que abarque las otras secciones.

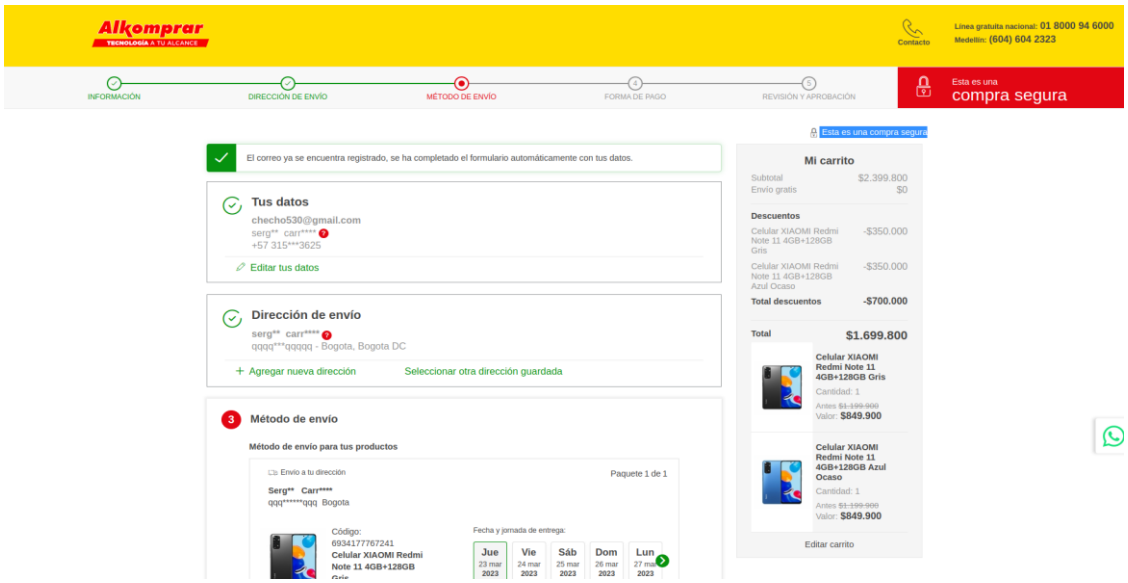
3.1.4 Checkout

Esta sección corresponde a la en que concluye el ciclo de compra de un producto por parte del usuario. Su buen funcionamiento y rendimiento son vitales para que el usuario pueda completar su compra de forma satisfactoria y no generar ningún tipo de desconfianza en el proceso. Adicionalmente, según los desarrolladores front-end, consideran esta como la sección con el código más desorganizado y poco entendible.

A continuación, en la **Figura 17**, se presenta una vista de la sección:

Figura 17.

Vista del checkout de la plataforma e-commerce de alkomprar.

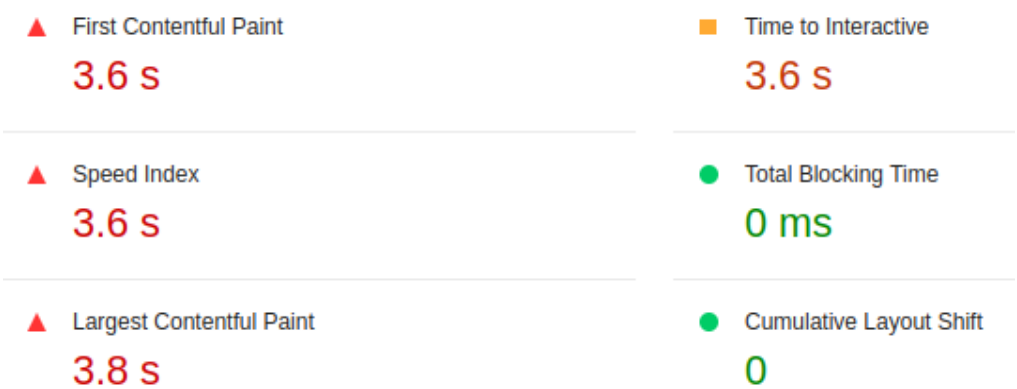


Nota. Tomado de Alkomprar (2023).

Al realizar un análisis con lighthouse localmente se obtienen los resultados de la **Figura 18**.

Figura 18.

Resultados métricos de lighthouse para el checkout.



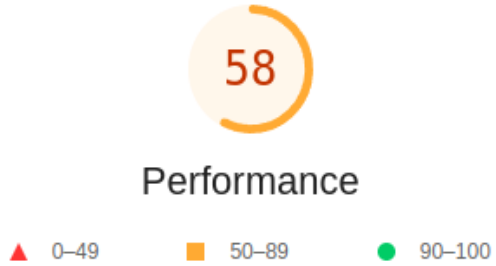
Nota. Tomado de Google Chrome, lighthouse (2023).

Resultados no muy buenos pero que en general no varían mucho respecto a los vistos en las secciones anteriores.

A continuación, en la **Figura 19**, se presenta el puntaje de rendimiento general asignado a esta sección.

Figura 19.

Puntuación de performance dada por lighthouse para la sección del checkout.

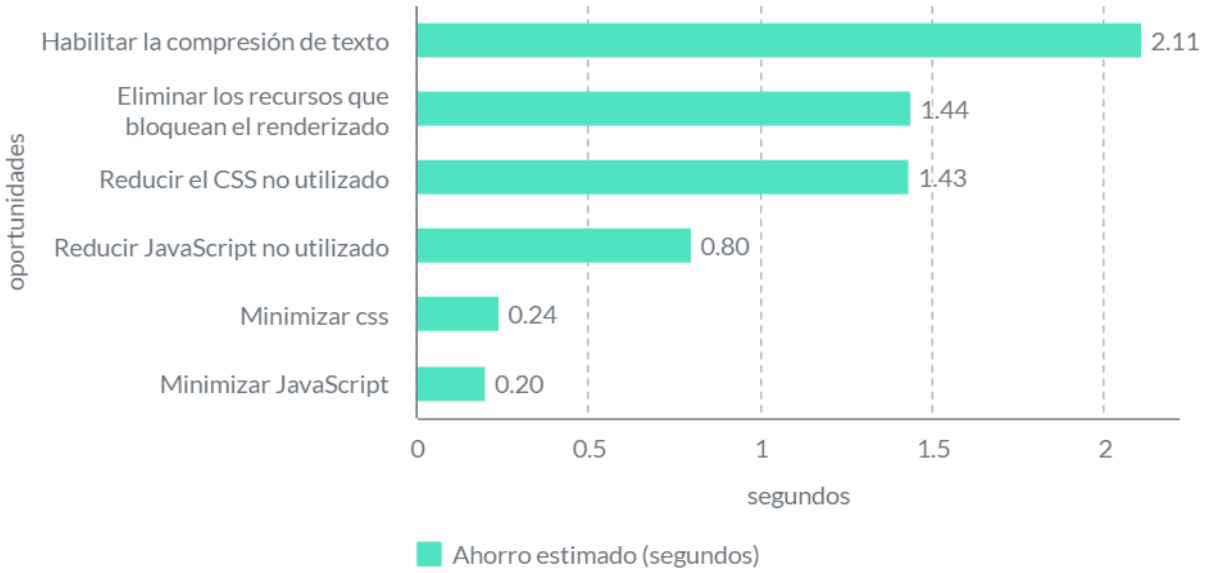


Nota. Tomado de Google Chrome, lighthouse (2023).

Rendimiento general similar al de las secciones anteriores por lo que se podría pensar que la mayoría de las secciones dentro de la página tendrán unos resultados similares en cuanto al rendimiento. Las oportunidades para mejorar el tiempo de carga en esta sección se presentan en la **Figura 20**.

Figura 20.

Oportunidades dadas por lighthouse para mejorar el performance en el checkout.



Se puede observar que para la sección del checkout, el beneficio obtenido por reducir la cantidad de JavaScript no usado es significativo. En otras palabras, mucho del código cargado para esta sección no está siendo utilizado, lo cual es en parte provocado por la desorganización y poca claridad del código. Para el caso de la sección del checkout, según lo perciben los propios desarrolladores estas son situaciones más problemáticas en comparación con las demás secciones dentro de la plataforma.

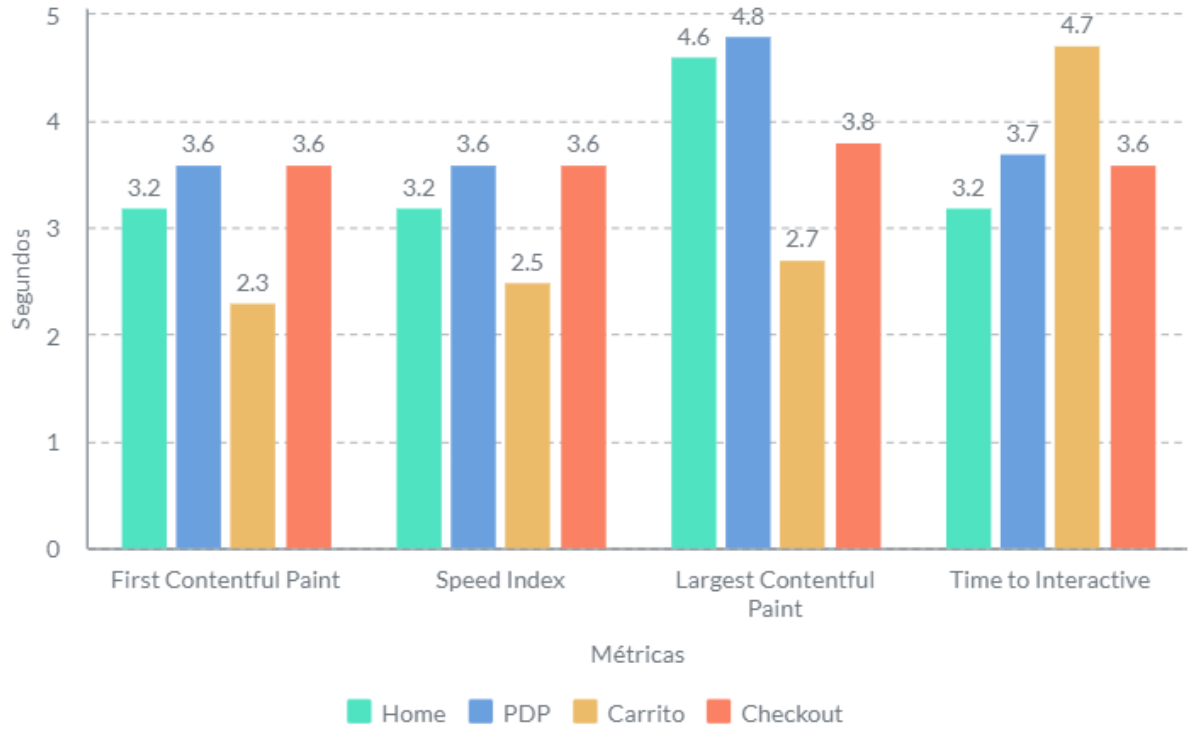
Después de analizar por separado cada una de las cuatro secciones establecidas en la Pagina 32 como las más relevantes en plataformas e-commerce, ya se vuelve más claro el estado actual de cada sección en el contexto de Alkomprar. A partir del análisis individual, ya es posible tener cierta idea de cuál sección sería la indicada en cuanto a resultados de las métricas y su papel actual en los desarrollos que se están realizando. A continuación, con el objetivo de tener una vista más clara, se realizó un análisis comparativo entre todas las secciones para, de esta forma, finalmente decidir cuál de las 4 secciones es la más indicada para implementar el proceso de modularización.

3.2 Análisis comparativo de las secciones

En la **Figura 21** se agruparon los resultados de las métricas de rendimiento de lighthouse para cada una de las secciones con el objetivo de ver más fácilmente cómo se comportan estas métricas para las diferentes secciones.

Figura 21.

Comparación del tiempo de cada una de las métricas para cada sección.

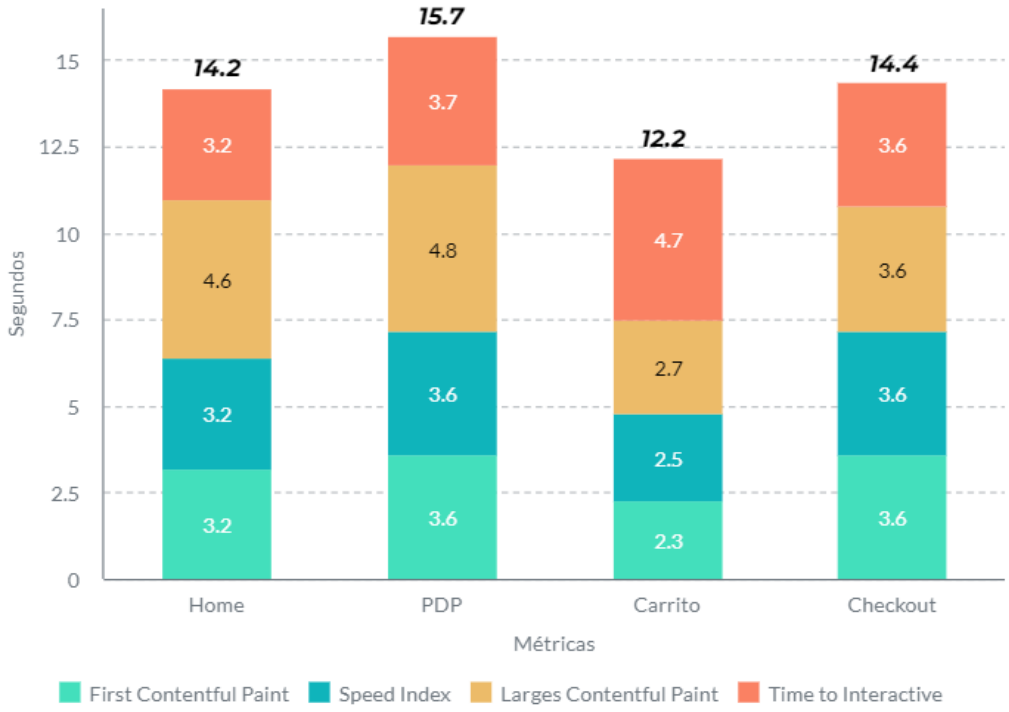


Se puede observar que, para la gráfica comparativa, se omitieron las 2 últimas métricas. Estas métricas tienen mediciones de 0 segundos para cada una de las secciones analizadas, por lo que no aportan nada a la comparación. En cuanto a los resultados de las métricas, estos se encuentran bastante variados para cada una de las secciones. Se observan casos en los que ciertas métricas para algunas secciones tienen tiempos bastante altos, mientras que para otras secciones los tiempos son relativamente bajos.

Con el fin de tener un valor único que permitiera tomar una decisión sobre cuál sección tiene los peores resultados de métricas, en **la Figura 22** se agruparon los resultados de cada una de las métricas por sección y de esta manera ver cual tiene una mayor prioridad.

Figura 22.

Tiempo acumulado de todas las métricas para cada sección.

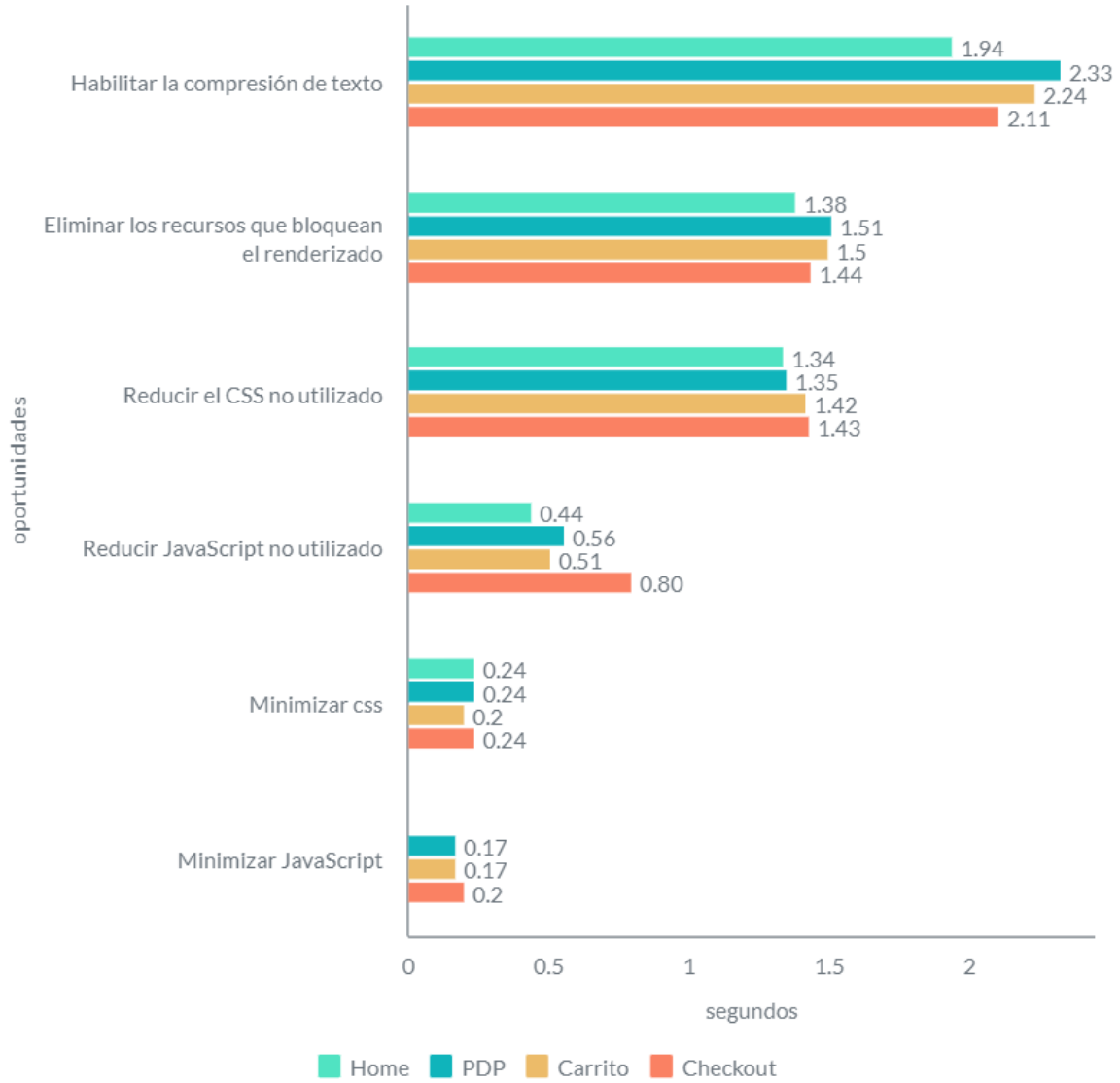


Al observar la gráfica es mucho más claro cuál sección es la peor en base a los resultados de las métricas. Esta sección sería la Product Page o PDP como se le conoce en el proyecto de Alkomprar. En cuanto a las otras secciones, estas no se encuentran muy alejadas entre sí, a excepción del carrito, el cual, a pesar de tener el tiempo más grande en la métrica de tiempo de interacción, se encuentra con un tiempo acumulado considerablemente más bajo que la sección de la Product Page.

Además de los resultados de las métricas, Lighthouse también ofrece unas oportunidades con las cuales se puede mejorar el resultado general de rendimiento de la sección. A continuación, en la Figura 23, se comparó el ahorro que se puede obtener de estas oportunidades para cada una de las secciones y, de esta forma, ver claramente qué sección tiene la mayor oportunidad de mejorar en cuanto al rendimiento.

Figura 23.

Comparación del tiempo que se puede ahorrar por oportunidad para cada sección.

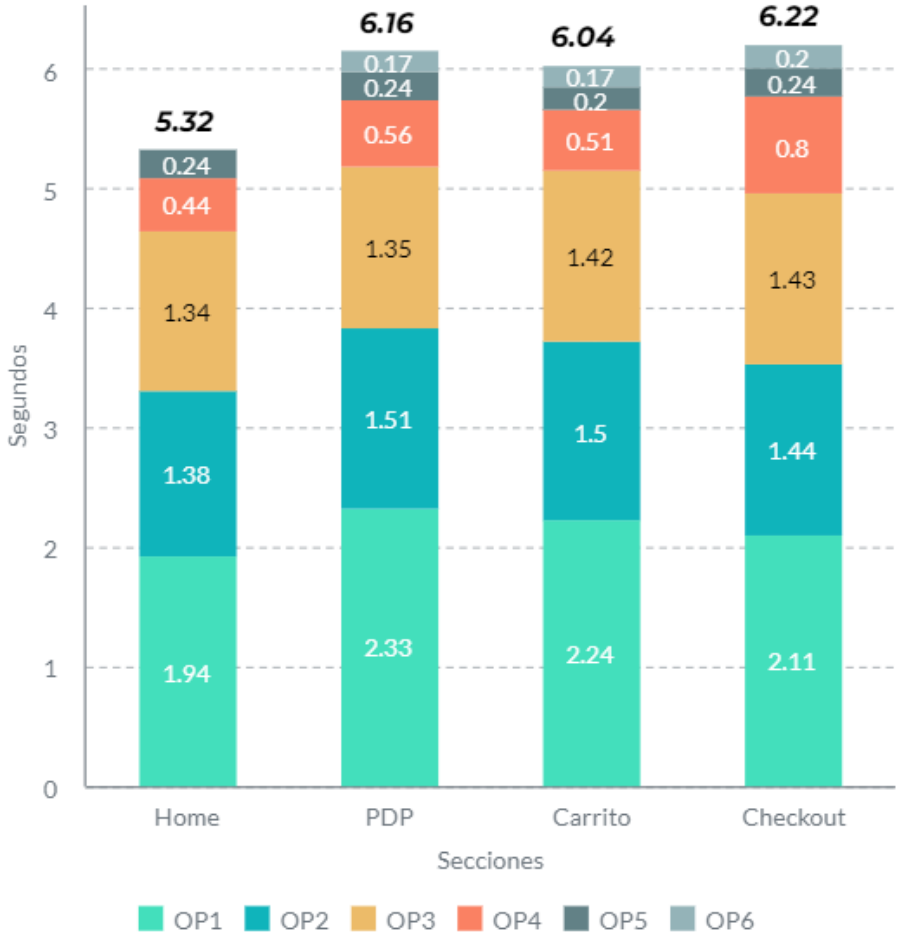


En general, todas las secciones presentan un tiempo de ahorro relativamente similar para cada una de las oportunidades identificadas por Lighthouse. Por lo tanto, en la **Figura 24**

se agrupó el ahorro correspondiente a cada sección, con el objetivo de obtener una visión más clara sobre cuál sección presenta el mayor tiempo de ahorro.

Figura 24.

Tiempo total ahorrado por todas las oportunidades para cada sección.



En términos generales, la sección del checkout y la PDP presentan un ahorro de tiempo total muy similar si se tienen en cuenta todas las oportunidades identificadas por Lighthouse. Sin embargo, debido al enfoque del proyecto, que se basó en modularizar el código aprovechando las funcionalidades de JavaScript en la versión ECMAScript 6, el análisis se centró en comparar los tiempos de ahorro relacionados a código JavaScript no utilizado. A

continuación, en la **Figura 25**, se podrá observar de manera más clara los tiempos de ahorro para las oportunidades relevantes en el contexto del proyecto.

Figura 25.

Tiempo total ahorrado por oportunidades relacionadas a código sin usar de javascript

Al dejar solo las oportunidades relacionadas con la carga innecesaria de código javascript se puede observar que la sección del checkout presenta la mayor oportunidad de mejora encontrándose significativamente por encima de las otras, debido a esto el realizar un proceso de modularización en esta sección sería una buena oportunidad de mejora .

Finalizando el análisis grupal de los resultados de las métricas y de los datos de rendimiento brindados por lighthouse se hace claro que hay 2 secciones que se encuentran con unas necesidades más altas que las otras 2, estas secciones son la Product Page (PDP) y el

Checkout, sus puntajes de rendimiento global son 56 y 58 respectivamente conformando los 2 peores resultados.

En general, la sección de la Product Page presenta una oportunidad de mejora superior si la decisión se basa únicamente en los resultados obtenidos por Lighthouse. Sin embargo, como ya se mencionó anteriormente, la sección de la Product Page al momento del desarrollo de este proyecto está experimentando un cambio en su estructura y diseño. Debido a esto, realizar la modularización en esta sección provocaría problemas con los desarrollos que están siendo llevados a cabo por otros desarrolladores. Además, es probable que gran parte del código cambie o sea modificado. Por lo tanto, en general, el contexto específico no permitió seleccionar la Product Page a pesar de los resultados observados.

Por esta razón, se decidió elegir la sección del checkout como la indicada para realizar la modularización. Además de ser la segunda mejor en cuanto a oportunidades de mejora, es también una sección que no tiene asignados desarrollos grandes que puedan dificultar la reubicación del código o la implementación de la modularización. Otro aspecto importante para tomar esta decisión es que el checkout suele ser una de las secciones en las que los usuarios abandonan más frecuentemente, con un índice de abandono de hasta el 72%, según un informe publicado por Forbes en 2023 (Anna Baluch, 2023). Por lo tanto, un buen rendimiento en esta sección es deseable.

Ya identificada la sección a trabajar, se optó por realizar un análisis de qué archivos JavaScript se están cargando. Esto con el principal objetivo de tener una mayor claridad sobre cómo se encontraba la situación actual y permitir enfocar todos los esfuerzos en los archivos más problemáticos. Debido al tamaño de la plataforma de comercio electrónico de Alkomprar,

solo el checkout estaba cargando una cantidad de código bastante elevada, por lo que dar prioridad a las partes más problemáticas fue el siguiente paso a seguir.

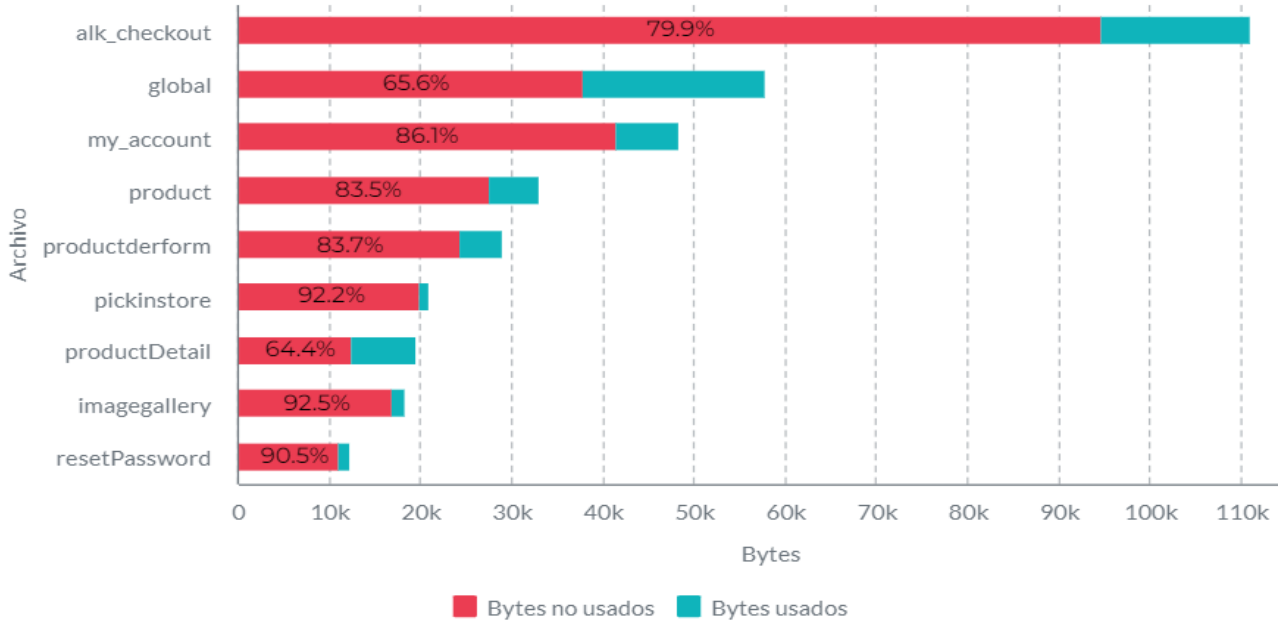
3.2.1 Análisis de los archivos cargados para la sección del checkout

Un aspecto importante antes de empezar con la modularización del código fue analizar cuántos archivos se están cargando en la sección, el tamaño de cada archivo, el tipo del archivo y muchos datos más que ayudarán a tomar decisiones correctas sobre cómo llevar a cabo la modularización. El navegador de google tiene integrado en sus chrome dev tools una herramienta llamada coverage. Esta herramienta permite analizar los archivos que está cargando una sección en específico, así como el tamaño del archivo, el tipo de archivo y el porcentaje de código dentro del archivo que realmente está siendo utilizado.

A continuación, en la **Figura 26**, con la ayuda de la herramienta coverage se realizó un análisis de los archivos cargados para la sección del checkout y específicamente para los archivos javascript:

Figura 26.

Porcentaje de uso de los archivos más grandes cargados en la sección del checkout



En la figura anterior, se presenta una lista de los archivos de mayor tamaño que se están cargando en la sección del checkout, todos son archivos de JavaScript que contienen funciones necesarias para el correcto funcionamiento de todas las características presentes en el checkout.

Al analizar la gráfica, se puede observar que varios archivos presentan porcentajes bastante elevados de código sin utilizar. Estos altos porcentajes están principalmente relacionados con la falta de organización en el código, pero también se deben al hecho de que el usuario recibe código de funcionalidades que no necesita en ese momento. Con la herramienta de cobertura coverage, es posible analizar cada uno de los archivos de forma individual, lo que permite observar las funciones definidas en cada uno y ver qué funciones se cargan y cuáles no (ver Figura 27). Gracias a esta funcionalidad, es fácil identificar qué secciones de código se utilizan en cada momento, lo cual es útil para identificar qué partes del código se deben cargar en cada momento.

Figura 27.

Cobertura para el archivo "alk_checkout".

```
redirectToAuthentication: function(returnURLparam, form) {
    var url = "/checkout/options/request/save";
    if (undefined !== form) {
        url = url + "?checkoutForm=" + form;
    }

    $.post(url, {
        returnURL: returnURLparam
    }, function() {
        var redirectUrl = ACC.config.encodedContextPath + "/checkout/options";
        window.location.replace(redirectUrl);
    });
},

checkBillingAddress: function() {
    var $jsBillingAddressFields = $(".js-billing-address-fields");
    $jsBillingAddressFields.removeClass("hidden");
    if (!$(this).find(".saveBillingAddressInMyAddressBook").prop("checked", true)) {
        $jsBillingAddressFields.addClass("hidden");
    }
},
```

Nota. Para la cobertura, las líneas resaltadas en rojo corresponden al código que no está siendo utilizado y las azules al código que sí. Tomado de Google Chrome, coverage (2023).

Debido a la gran magnitud de la plataforma de comercio electrónico, resultaba muy ambicioso trabajar en cada uno de los archivos cargados para la sección del checkout. Por lo tanto, lo más conveniente para este proyecto fue seleccionar un caso específico de los archivos que se cargan en esta sección y llevar a cabo el proceso de modularización en dicho archivo. De esta manera, es posible obtener conclusiones sobre el proyecto para un archivo en específico y analizar si la modularización resulta realmente beneficiosa para ser implementada en el resto del proyecto.

El archivo seleccionado para realizar el proceso de modularización fue el archivo con mayor cantidad de datos cargados y con mayor código sin utilizar, este archivo corresponde al primero de la **Figura 26** el cual tiene por nombre "***acc.alk_checkout***". Este archivo contiene las principales funcionalidades relacionadas al flujo que sigue el usuario a través del proceso

de checkout. Además, es uno de los archivos más grandes, duplicando en tamaño al siguiente archivo más grande y teniendo un porcentaje bastante elevado de código.

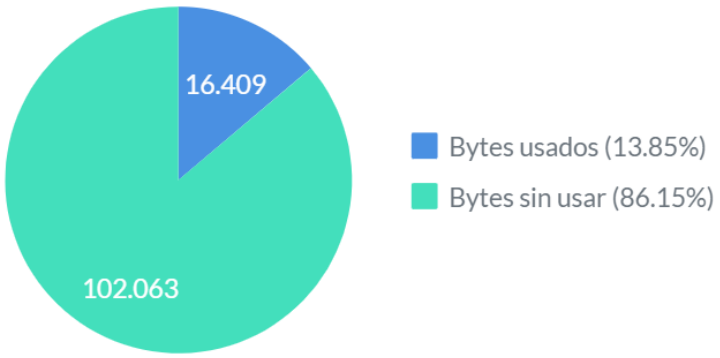
Por todas estas razones y con el objetivo de lograr un enfoque y una organización adecuada del código, se trabajó exclusivamente en este archivo. A continuación, se proporcionará una breve descripción general del archivo y se mostrará el tiempo que lleva cargar este archivo antes de empezar con el proceso de modularización.

3.2.2 Análisis del archivo (acc.alk_checkout) del checkout

Como se ha mencionado, este archivo es uno de los de mayor tamaño y también uno de los de mayor porcentaje de código sin utilizar que había para la sección del checkout. En la **Figura 28** , se presenta el porcentaje de uso para el archivo.

Figura 28.

Porcentaje de uso del archivo (acc.alk_checkout).



Más del 85% del código cargado en un inicio del proceso de checkout no es usado inmediatamente, este porcentaje corresponde a 102.063 bytes lo cual supera por mucho al tamaño completo de otros archivos que también son cargados en el checkout.

Adicionalmente en la **Figura 29** se puede observar algunos datos relacionados al tiempo que toma bajar la información del servidor:

Figura 29.

Datos del tiempo de carga para el archivo.

Tarea	***Duración en milisegundos***	
	Red lenta	***Red rápida***
Conexión estancada	0.41	2.72
Request enviada	0.092	0.082
Espera por respuesta del servidor	568.41	2.51
Descarga del contenido	485.12	6.74
Tiempo total	1054.03	11.97

El análisis fue dividido entre una red lenta y una rápida con el objetivo de lograr un mejor análisis de los resultados después del proceso de modularización.

Por último, como se mencionó anteriormente, este archivo engloba numerosas funcionalidades de diferentes tipos y para diferentes elementos dentro del checkout. Crear un módulo para cada función no era la mejor solución en términos de rendimiento. Por lo tanto, en la siguiente sección de este informe se definieron ciertos criterios que se tomaron en cuenta al momento de decidir qué módulos se crearán y qué funciones no requieren modularización

3.3 Criterios para la creación de los módulos

Antes de iniciar con la modularización se definieron criterios claros que permitieran identificar qué partes del código debían ser modularizadas y cuáles no. De esta manera, se evitó la creación innecesaria de módulos que no aportan suficiente valor. Para decidir si una funcionalidad debía ser modularizada, se consideraron los siguientes criterios:

- **Complejidad:** Si una función o sección del código es extensa y realiza múltiples tareas o contiene lógicas complejas, se considerará la modularización. Al dividir el código en módulos más pequeños y legibles facilitará su comprensión y mantenimiento.

- **Reutilización:** Si una función o sección de código se utiliza en diferentes lugares dentro del proyecto o puede ser útil en futuros desarrollos, la modularización sería beneficiosa. Al encapsular estas funcionalidades en un módulo independiente, se facilitará su reutilización.

- **Tareas independientes:** Si una función o sección de código aborda múltiples tareas que podrían tratarse de forma independiente, es probable que se requiera la modularización. Separar cada una de estas tareas en módulos independientes, enfocándose en una sola responsabilidad para cada módulo, resultará en un diseño más claro y mantenible.

- **Escalabilidad:** Si una función del proyecto es propensa a cambios o expansiones debido a futuras funcionalidades, la modularización se convierte en una opción deseable. Al modularizar, se logrará un código mucho más escalable y se facilitará la gestión de modificaciones futuras.

Teniendo en cuenta estos criterios, fue más sencillo identificar las funciones específicas que requieren una mayor prioridad en el proceso de modularización. Una vez definidos los criterios a considerar para crear los módulos, fue necesario definir las condiciones que determinen cuándo se debe cargar cada módulo, aprovechando así las ventajas de los módulos dinámicos de JavaScript.

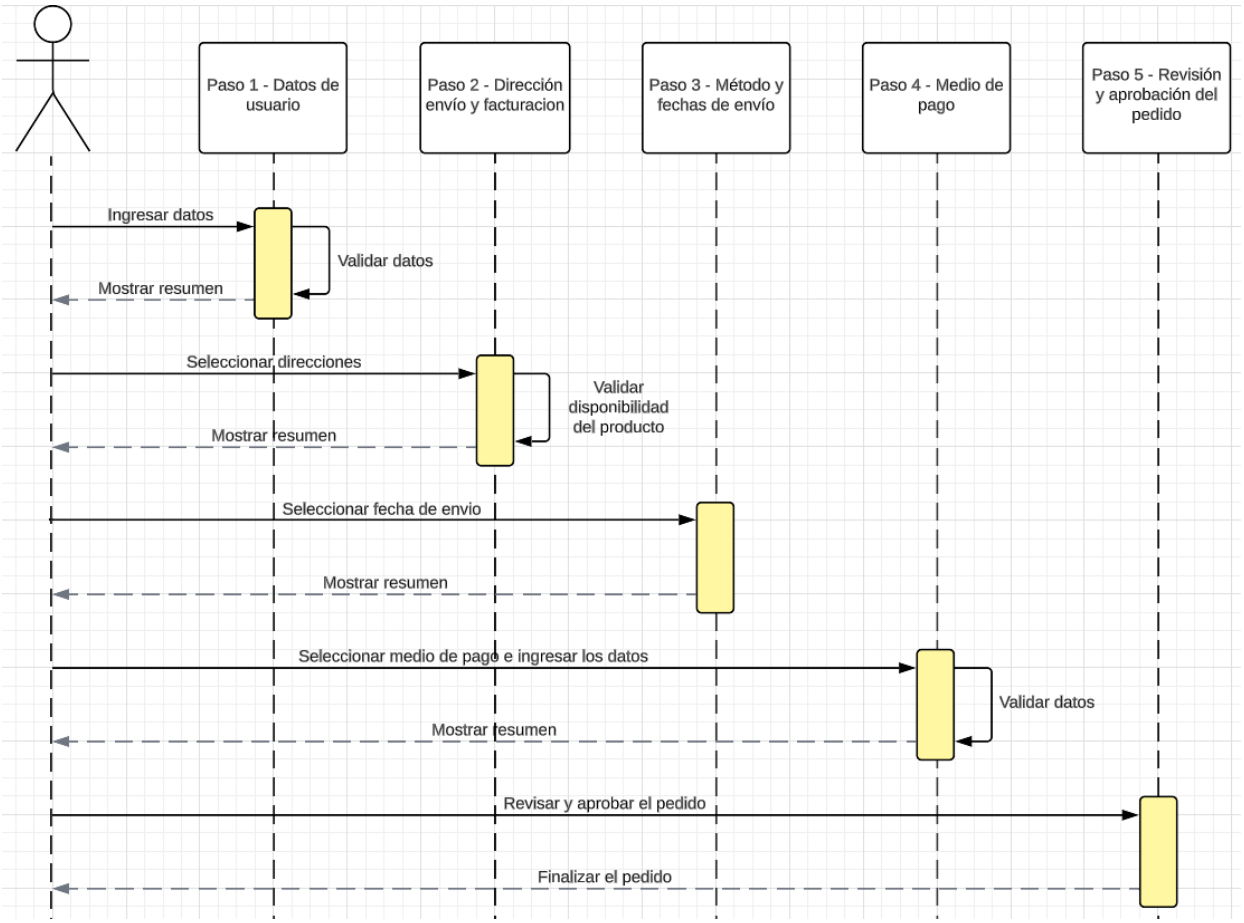
3.4 Condiciones para la carga de los módulos

Como se ha mencionado anteriormente, uno de los objetivos del proyecto fue aprovechar las características de los módulos dinámicos que ofrece JavaScript. Para lograr implementar este tipo de módulos, fue necesario definir ciertas condiciones que permitieran cargar los módulos sólo cuando fueran necesarios.

Antes de establecer las condiciones adecuadas, es fundamental comprender el proceso del checkout y los pasos que el usuario debe seguir para completar su compra. A continuación, en la **Figura 30**, se presentará una visión general de este proceso mediante un diagrama de secuencias.

Figura 30.

Diagrama de secuencias del proceso de checkout.



El proceso de checkout consta de 5 pasos principales, cada uno con sus propias características y funcionalidades asociadas. Es importante destacar que este proceso es completamente secuencial, es decir, se debe avanzar de un paso al siguiente solo cuando se haya completado el paso anterior y la información haya sido validada.

Para establecer las condiciones, fueron considerados estos pasos como puntos de referencia al momento de decidir cuándo cargar cada módulo. A continuación, se proporcionará una breve explicación de cada uno de los pasos, junto con una vista ilustrativa que ayudará a comprender mejor las implicaciones de cada uno.

Primer paso (Tus datos)

En este paso, el usuario ingresa sus datos personales. Cualquier funcionalidad que controle estos campos, como la verificación del correo electrónico o similares, debería ser cargada en esta sección. En la **Figura 31**, se muestra la vista del primer paso:

Figura 31.

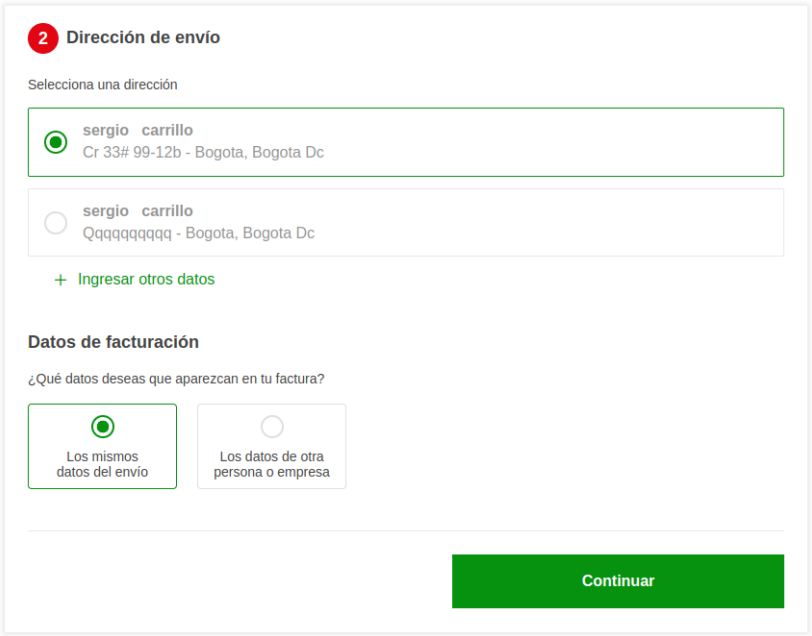
Vista primer paso del checkout.

Segundo paso (Dirección de envío y facturación)

En este paso, el usuario ingresa los datos de la dirección de envío y la de facturación en caso de ser diferentes. Al igual que en el paso anterior, existen ciertas funcionalidades que controlan este paso y que solo deberán cargarse cuando el usuario se encuentre aquí. En la **Figura 32**, se muestra la vista del segundo paso:

Figura 32.

Vista segundo paso del checkout.



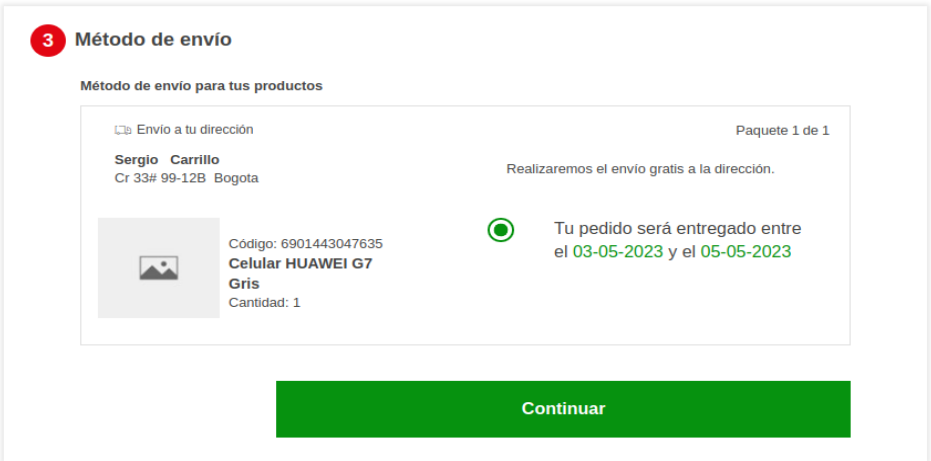
Tercer paso (Método de envío)

En este paso, se proporciona información sobre el método de envío previamente seleccionado.

En la Figura 33, se muestra la vista del tercer paso:

Figura 33.

Vista tercer paso del checkout.

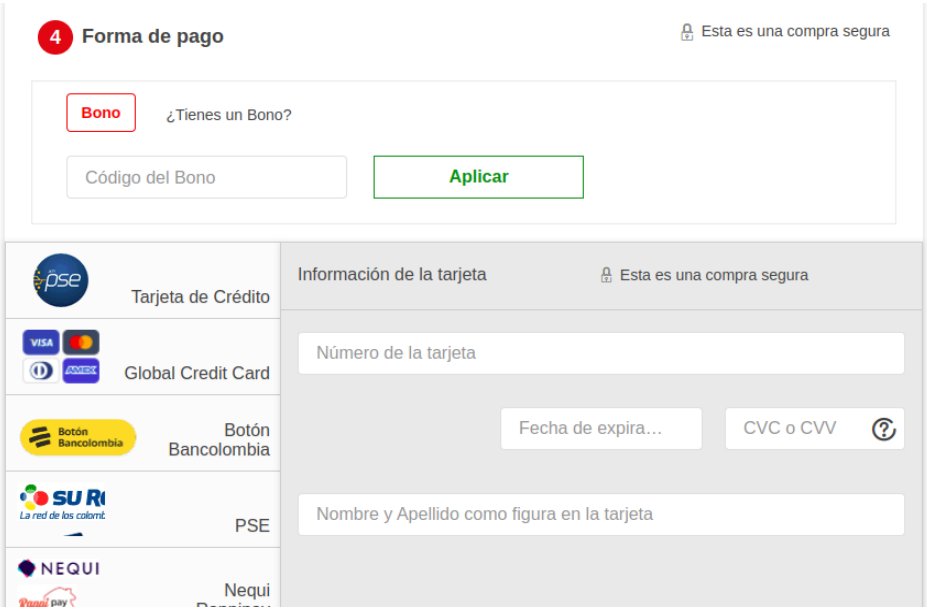


Cuarto paso (Método de pago)

En este paso, el usuario selecciona el medio de pago y llena la información requerida. Aquí se cargan todas las funcionalidades que verifican la información de las tarjetas de crédito. En la **Figura 34**, se muestra la vista del cuarto paso:

Figura 34.

Vista cuarto paso del checkout.



Quinto paso (Revisión del pedido)

Este es el último paso del flujo del checkout, en el cual el usuario revisa si el pedido es correcto y aprueba si todo está en orden. En la **Figura 35**, se muestra la vista del quinto paso:

Figura 35.

Vista quinto paso del checkout.

Como se puede apreciar, cada paso del checkout tiene características y funcionalidades propias necesarias para su correcto funcionamiento. En un comienzo, todas estas funciones estaban disponibles tan pronto como se accede a la sección del checkout. Sin embargo, esto daba lugar a una gran cantidad de código JavaScript no utilizado, como se ha observado en los resultados de cobertura.

Con el objetivo de optimizar la carga, se busca que solo se carguen las funcionalidades correspondientes al paso en el que se encuentra el usuario. Por ejemplo, si un usuario se encuentra en el paso 2, no se cargarán las funcionalidades del paso 4 al cual aún no ha llegado y al cual no se tiene la certeza de que llegará.

En conclusión, las condiciones de carga se basaron en la ubicación del usuario dentro de los pasos establecidos. Para lograr esto, se aprovecharon los elementos existentes en la estructura HTML de la página, los cuales definen clases diferentes para cada paso. Estas clases se activan y desactivan siguiendo el flujo presentado en el diagrama de secuencia de manera que el usuario pueda ver únicamente la información del paso en el que se encuentra ubicado.

Una vez concluido el análisis y de haber establecido los criterios de modularización y las condiciones de carga, se procederá en la siguiente sección a abordar la modularización del código teniendo en cuenta todas las consideraciones establecidas en esta sección.

4. Modularización de las funcionalidades

Habiendo definido claramente en qué partes se va a enfocar la modularización, se procedió a la siguiente etapa del proyecto. Esta fase consistió en la implementación de los módulos y en todos los aspectos necesarios para su correcto funcionamiento. El siguiente apartado se dividió en cada uno de los aspectos clave que fueron necesarios para llevar a cabo la modularización satisfactoriamente. Entre los cuales se encuentran la división y encapsulamiento del código, la ubicación de los módulos en un lugar apropiado, la documentación de los módulos, el exporte e importe de los módulos, y por último, la implementación de condiciones para la carga de los módulos sólo cuando sean necesarios.

En la siguiente sección, se analizarán en detalle cada uno de estos temas, así como el razonamiento detrás de las decisiones tomadas con respecto a cada uno de los apartados.

4.1 División del código en módulos

Una vez establecido el archivo de funcionalidades que iba a ser tenido en cuenta para la modularización, se procedió a revisar todo el código en busca de aquellas funciones que cumplían con los criterios definidos en la sección anterior. Este proceso de revisión de código se llevó a cabo con la colaboración del equipo de proyecto a través de la metodología Scrum.

Gracias a las constantes reuniones realizadas y a la socialización de diferentes temas relacionados con los proyectos actualmente en desarrollo y el estado actual de la plataforma, fue posible lograr un mejor entendimiento de las funciones del checkout. Esto resultó de gran ayuda al momento de revisar cada una de las funciones, ya que gracias a los conocimientos adquiridos fue más sencillo comprender cuál era la lógica detrás de estas y, a través de esa información, seleccionar las funcionalidades que cumplían con los criterios establecidos para ser tenidas en cuenta en el proceso de modularización.

Además de la comunicación constante con los miembros del equipo y desarrolladores del proyecto, la herramienta coverage también fue de gran ayuda en el proceso de familiarización con las funciones que se trabajaban, gracias a coverage fue más sencillo entender cómo se estaban cargando las funciones.

Después de revisar todo el código, se identificaron más de 50 funciones que cumplían con las condiciones establecidas previamente. Estas funciones abarcan diversos tipos, incluyendo funciones de validación, manejo de formularios y otras encargadas de controlar el comportamiento de elementos específicos en la sección del checkout.

De las funciones seleccionadas se evidenció que muchas de estas habían experimentado múltiples modificaciones a lo largo de los años, realizadas por diferentes desarrolladores. Esta situación generaba una complejidad adicional al intentar encapsular el código. Se identificaron problemas de acoplamiento e inconsistencias en la estructura y organización de algunas funciones, lo que complicaba su entendimiento, división y encapsulamiento.

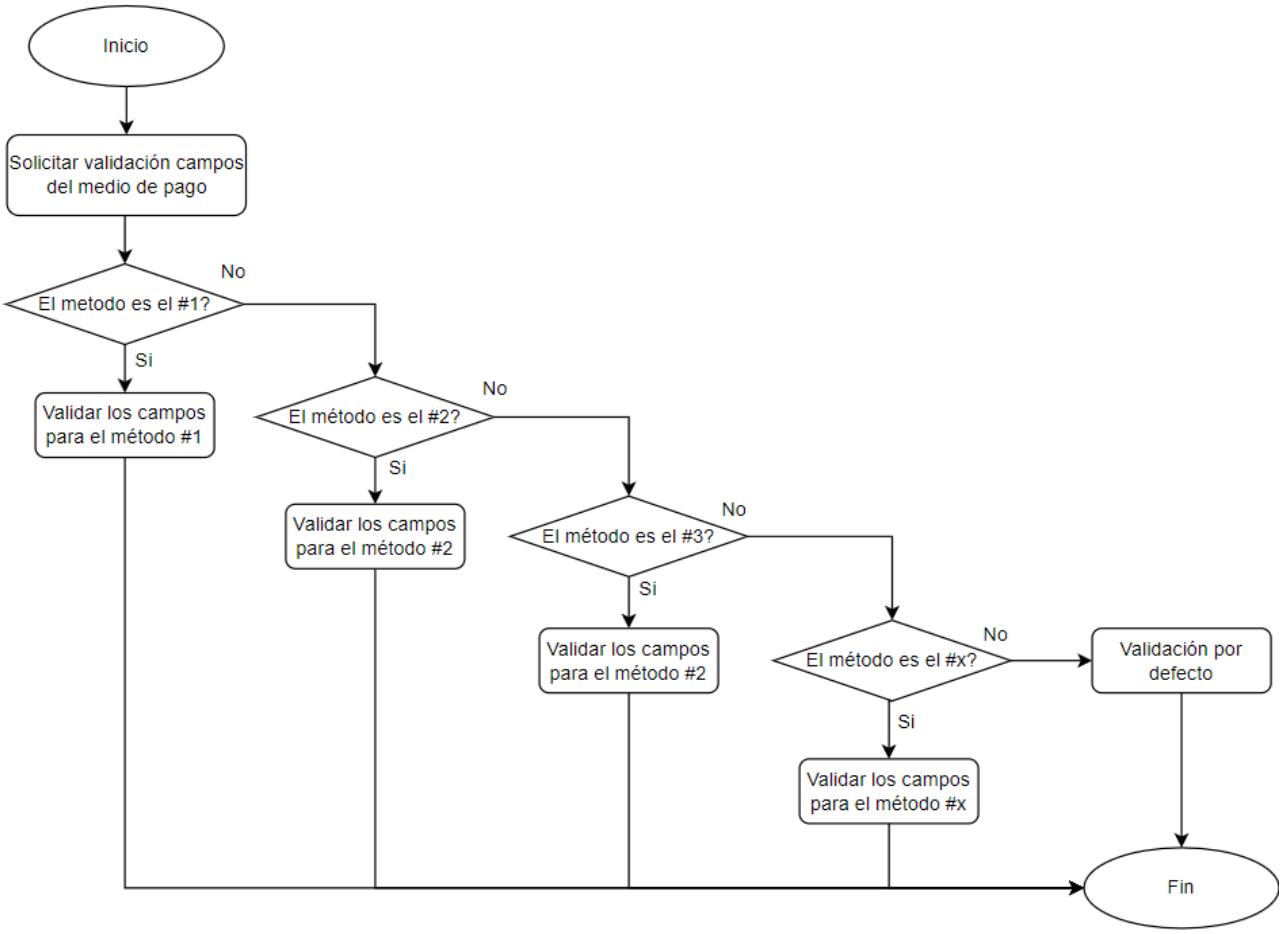
Debido a esta situación, fue necesario llevar a cabo ajustes en las funciones existentes antes de continuar con la división en módulos. El objetivo principal era optimizar su separación y facilitar su integración con otras funciones a través de los módulos. Este proceso se conoce comúnmente como refactorización del código. En este proyecto, se adoptó un enfoque básico de la refactorización, que implicó la reorganización y modificación del código con el propósito de lograr funciones más transparentes y comprensibles.

Siguiendo el enfoque de refactorización, se reescribieron algunas funciones y se reorganizaron otras, reduciendo de esta manera el nivel de acoplamiento y facilitando la división en módulos. Uno de los cambios más significativos implicó la transformación de una función extensa, compuesta por más de 500 líneas de código. Esta extensión dificultaba tanto su comprensión como su escalabilidad. Por tal motivo, se reestructuró el código con el fin de desvincularlo y definir funciones más compactas y manejables, representativas de los distintos módulos.

Esta función tiene la responsabilidad de validar las entradas de los métodos de pago provenientes del front-end. En la **Figura 36**, se describe como se llevan a cabo estas validaciones.

Figura 36.

Funcionamiento de la validación de los campos de los métodos de pago.



Como se observa en el diagrama de flujo, su funcionamiento es bastante sencillo: se inicia solicitando la validación del método de pago. Posteriormente, se procede a validar cuál fue el método de pago seleccionado, en función de lo cual se ejecuta la validación de los campos particulares, siguiendo los criterios propios de cada medio de pago.

La principal problemática de esta función se debida a su longitud, la cual excedía las 500 líneas de código, esta situación se atribuía al hecho de que todas las validaciones se encuentran ubicadas en una sola función. Esta acumulación de validaciones, sumada al hecho

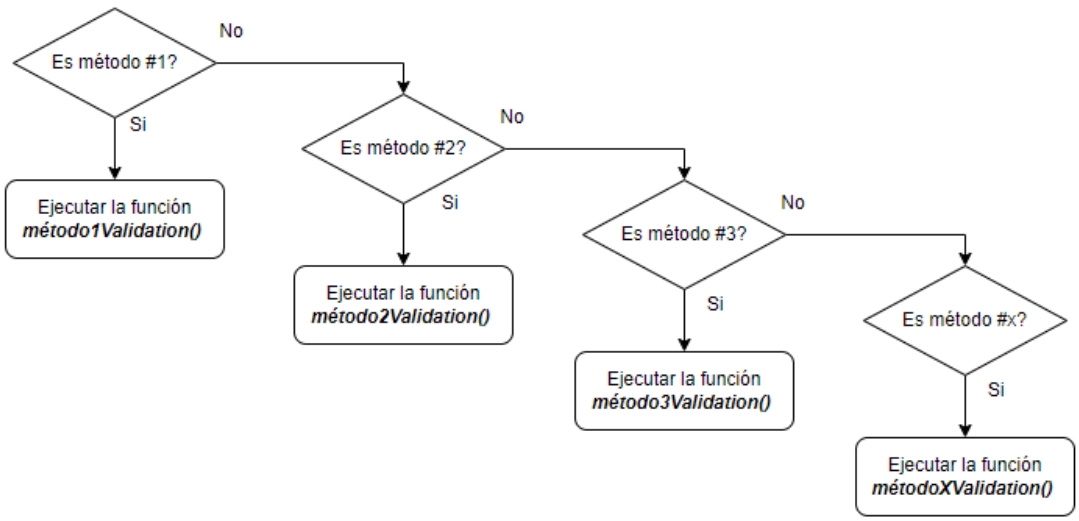
de que la función se carga desde el inicio, genera un desafío en términos del rendimiento del código.

La falta de escalabilidad y la complejidad para entender el código representaban puntos críticos en esta situación. Además, si en un momento posterior se incorporan más métodos de pago, esto provocará un aumento en la longitud del código, lo que a su vez aumentará la complejidad de la función aún más.

El principal problema de la función era que realizaba las validaciones para cada uno de los métodos de pago directamente, lo que dificultaba mucho su entendimiento. Con el objetivo de lograr un código más claro y trabajable, se decidió definir funciones específicas para las validaciones de cada uno de los métodos de pago como se observa en la **Figura 37**. De esta forma, la función principal no contendría el código de las validaciones para cada uno de los métodos, sino que sólo mantendría la estructura para decidir cuándo se requiere cada validación.

Figura 37.

Validaciones de los inputs de los medios de pago luego de reestructurar.



Cuando alguna de las condiciones se cumpla, simplemente se hará una llamada a la función de validación correspondiente al método de pago específico. Así, la función principal mantendrá un funcionamiento igual al anterior, pero será mucho más fácil de entender y visualizar el código. El principal beneficio de realizar esta separación, además de la facilidad en el entendimiento del código, es la escalabilidad. Si en el futuro se desean agregar nuevos métodos de pago o nuevas validaciones a un método de pago en específico, será mucho más fácil realizar las correcciones.

Finalmente, luego de haber sometido las funciones que se van a modularizar al proceso de refactorización, se procedió a definir un total de 18 módulos los cuales agrupan las funciones previamente seleccionadas, algunos de los módulos definidos fueron los siguientes:

- **paymentRequiredInputs:** Conformado por un grupo de función que valida que los inputs requeridos para los diferentes métodos de pago estén completos antes de continuar al siguiente paso.

- **expirationDateValidation:** A diferencia del anterior aquí se realiza una validación de que los datos de fechas de expiración sean correctos, el módulo está conformado por funciones para los diferentes métodos de pago y casos específicos.

- **paymentTabs:** Funciones que controlan el comportamiento de cada una de las tab de los medios de pago, muchas de las funciones dentro del módulo se relacionan entre sí para funcionar.

- **clearInputsPayment:** Como lo indica el nombre del módulo, está conformado por funciones relacionadas con limpiar los inputs de los diferentes medios de pago cuando se cumplan las condiciones definidas.

- **formBehaviour:** Funciones que se encargan de aspectos específicos dentro de los formularios presentes en el proceso de compra, este módulo contiene funciones encargadas de los formularios del paso 2 específicamente.

- **creditCardValidation:** Cuenta con casos específicos para la validación de los datos cuando el medio de pago es tarjeta de crédito.

Estos son solo algunos de los módulos que se definieron en base a las funciones seleccionadas. La razón para definir 18 módulos es que se buscaba crear en su mayoría módulos útiles y que estuvieran conformados por funciones que representarán una oportunidad para mejorar tanto el tiempo de carga de la sección como la escalabilidad y el entendimiento del código en general. Por esta razón, en el proceso de selección de las funciones, se dejaron de lado aquellas que no representaban una oportunidad de mejora clara, ya sea por su funcionamiento muy independiente o porque la cantidad de código dentro de esas funciones era muy poca, como el caso que se puede observar en la **Figura 38**.

Figura 38.

Función no tomada en cuenta para el encapsulamiento .

```
checkBillingAddress: function () {
    var $jsBillingAddressFields = $(".js-billing-address-fields");
    $jsBillingAddressFields.removeClass("hidden");
    if (!$(this).find(".saveBillingAddressInMyAddressBook").prop("checked", true)) {
        $jsBillingAddressFields.addClass("hidden");
    }
},
```

La anterior función es un ejemplo de las funciones que no fueron tenidas en cuenta para el proceso de refactorización y encapsulamiento del código, como se observa la función no representaba una complejidad elevada ni corresponde a una función con un tamaño muy elevado, por estas razones esta y otras muchas funciones fueron dejadas de lado. De esta forma, el objetivo fue definir módulos que representarán una mejora real, en lugar de crear módulos excesivos que no aportaran valor o incluso complicaran más la situación en términos de rendimiento y comprensión del código por parte de futuros desarrolladores.

En conjunto con esta división del código en módulos y las reestructuraciones, se llevaron a cabo pruebas unitarias para cada una de las funcionalidades modificadas. También se realizaron pruebas de aceptación basados en criterios de aceptación previamente definidos en Keyrus, las cuales se abordarán más en detalle en la sección final.

Con los módulos creados ahora se requería implementar el código correspondiente a la importación y exportación de estos, pero antes de continuar era necesario encontrar una ubicación en el repositorio y además de establecer una estructura de carpetas que permitirá ubicar fácilmente los módulos desarrollados.

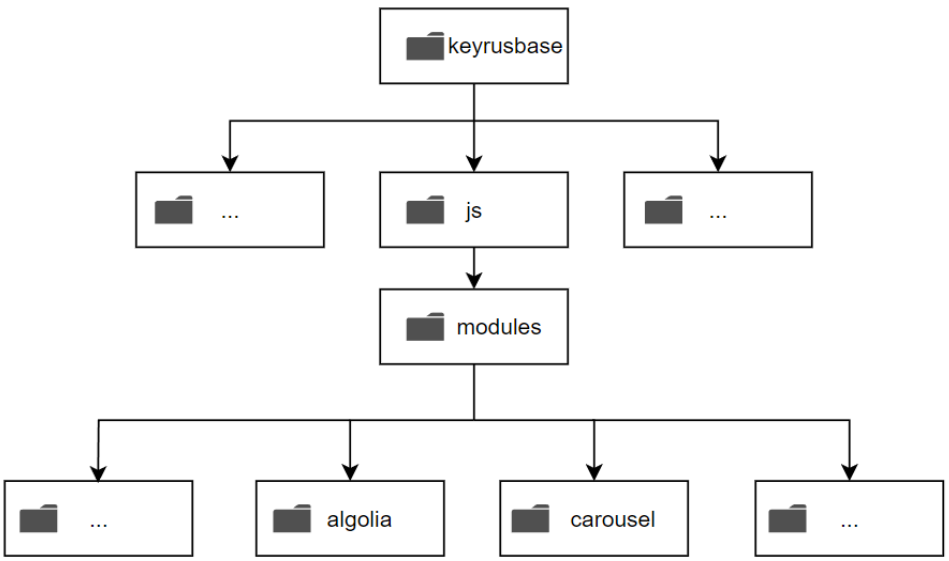
4.2 Ubicación de los módulos dentro del repositorio

La ubicación de los módulos dentro del repositorio fue una tarea relativamente sencilla y no requirió de un análisis detallado. Dado que el repositorio es bastante grande, es común que los desarrolladores utilicen atajos de teclado que permitan realizar búsquedas mediante palabras clave o directamente por el nombre del archivo. Por estas razones, la ubicación dentro del repositorio en ciertas ocasiones tiende a no ser muy recordada por los desarrolladores.

Por otro lado, una estructura de carpetas bien organizada y una ubicación adecuada del código facilitarán la identificación de lo que se ha hecho y a qué partes afecta específicamente. En el proyecto de Alkomprar, ya existía una carpeta destinada para la ubicación de los módulos, donde se encuentran pocos módulos correspondientes a proyectos específicos dentro de la plataforma como se observa en la **Figura 39**. Sin embargo, para el caso del checkout, no existe una carpeta específica donde se puedan ubicar los módulos creados.

Figura 39.

Estructura de carpetas del proyecto antes.



La estructura de carpetas del proyecto completo es bastante amplia y compleja. Para brindar una visión más enfocada en la perspectiva de los desarrolladores front-end, se proporciona una representación gráfica en la figura anterior. En esta representación, se resalta la disposición de carpetas específica utilizada por los desarrolladores front-end, lo que permite una comprensión más clara. Además, se destaca la ruta precisa que se debe seguir para acceder a los módulos javascript dentro de esta estructura, la cual empieza en la carpeta ***keyrusbase*** la cual se encuentra ya dentro de la carpeta específica dedicada al front-end, luego de ahí se encuentra una carpeta con todo el código de funcionalidades javascript y dentro de esta carpeta existe otra carpeta llamada ***modules*** la cual alberga pequeños módulos de proyecto específicos para la plataforma de alkomprar.

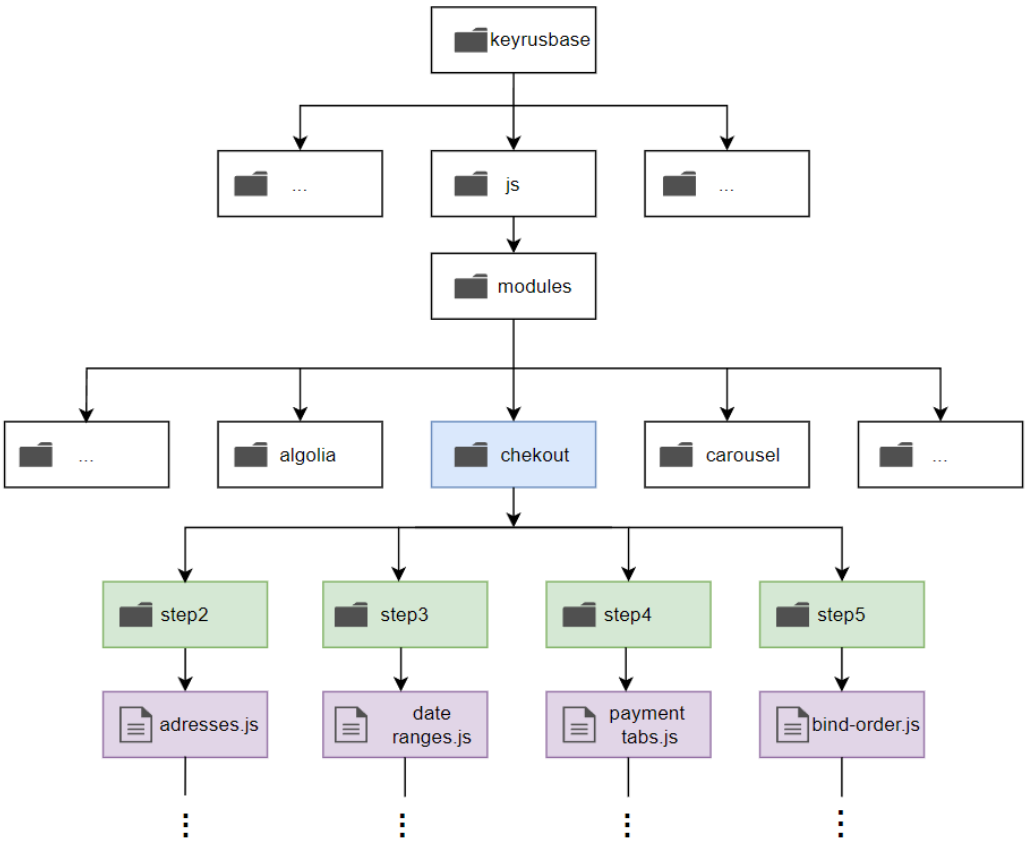
El primer paso para ubicar los módulos dentro del repositorio consistió en crear una carpeta para alojar los módulos relacionados con el proceso de checkout en la ubicación ya establecida previamente. Dentro de esta carpeta, se colocaron los distintos módulos desarrollados. Sin embargo, para lograr una mayor claridad, se crearon unas subcarpetas dentro de la carpeta principal, representando cada uno de los pasos que el usuario debe seguir en el proceso de checkout. De esta manera, resultará más visual y reconocible qué módulos afectan a cada parte del proceso de checkout, lo cual facilita su entendimiento, mantenimiento, escalabilidad y, en general, ayudará a los futuros desarrolladores a llevar a cabo sus tareas.

Gracias a Coverage, saber qué módulo se cargaba en cada paso fue bastante sencillo. Al analizar cada una de las funciones, se pudo determinar en qué partes de todo el proceso de checkout se estaban utilizando realmente.

Finalmente, con la incorporación de las carpetas relacionadas al proyecto, la estructura de directorios que proporciona acceso a los módulos se puede observar en la **Figura 40**:

Figura 40.

Estructura de carpetas del proyecto después.



Dado que el primer paso representa una etapa por la cual todos los usuarios deben pasar, ya que este es el punto de inicio del proceso de checkout, se tomó la decisión de no crear una carpeta específica para esta fase. Por lo general, cualquier desarrollo futuro relacionado con este paso, como nuevas funcionalidades o elementos, debe ser cargado desde el principio, ya que se trata del paso inicial. Por estas razones, la división en subcarpetas comenzará a partir del paso 2 en adelante en la que cada carpeta contará con sus respectivos módulos, los cuales más adelante se importarán al cumplir las condiciones específicas.

Siguiendo esta estructura, resulta sencillo identificar las funcionalidades específicas correspondientes a cada uno de los pasos. Además, facilita la ubicación de futuros módulos que puedan ser desarrollados posteriormente. Asimismo, al organizar los módulos de esta manera, se simplifica la documentación dentro del código, ya que la estructura en sí deja en claro que factores fueron tomados en cuenta para la carga de los módulos.

4.3 Importe y exporte de los módulos usando JavaScript

Después de haber encapsulado cada una de las funciones seleccionadas en sus respectivos módulos, y de haber almacenado estos módulos en una ubicación adecuada, el siguiente paso fue establecer el método de exportación e importación de dichos módulos.

En colaboración con el equipo de trabajo, especialmente con el líder de proyectos relacionados con el rendimiento de la plataforma, se desarrolló una función utilizando JavaScript asíncrono la cual se puede ver en la **Figura 41**. Esta función tiene la capacidad de importar los módulos definidos, considerando factores que facilitan su reutilización en diversas secciones del proyecto, creando de esta manera una función estandarizada y reutilizable que sea de utilidad en el caso de crear más módulos en diferentes secciones dentro de la plataforma.

Figura 41.

Función para la importación de los módulos.

```
importModule: moduleName => {
    return new Promise( executor: (resolve, reject) => {
        import(`${ACC.config.commonResourcePath}/js/modules/${moduleName}.js?v=${ACC.config.filesVersion}`)
            .then(Module => resolve(Module))
            .catch(e => reject(e))
    })
},
```

Se creó una función llamada ***importModule*** la cual se definió como un método dentro del objeto ***jsHelper*** el cual dispone de diversas funcionalidades útiles y el cual fue previamente definido como la ubicación de este tipo de funciones. En cuanto a la función, está dispone de un parámetro en el cual se especifica la ubicación del módulo que se desea cargar, adicionalmente en la función se hace uso del objeto ***Promise***, lo cual indica que el código es asíncrono, con este acercamiento se puede lograr que los módulos no sean cargados al inicio de la carga del archivo, sino que por otro lado estos sean cargados asíncronamente en el momento que sean necesarios.

La función ***importModule*** hace uso de variables las cuales se encuentran definidas en el objeto global ***ACC***, este objeto tiene diversos parámetros que facilitan la configuración de archivos garantizando un funcionamiento óptimo del código en todo momento.

En la **Figura 42**, se presenta un ejemplo del uso de la función ***importModule:***

Figura 42.

Importando una función de un módulo en específico.

```
addressProvinceAndCity: async () =>
    await jsHelper
        .importModule( moduleName: "checkout/step2/address-province-city")
        .then(({ addressProvinceAndCity }) => addressProvinceAndCity()),
```

La anterior función corresponde a un caso específico donde se hace uso de la importación de módulos usando el método ***imporModule*** del objeto ***jsHelper***, se puede observar que en esta función se hace uso de las palabras reservadas ***async*** y ***await,*** las cuales establecen que el código dentro de la función sea totalmente asíncrono. Cada una de las funciones elegidas para la modularización tendrán una estructura similar, estas harán un

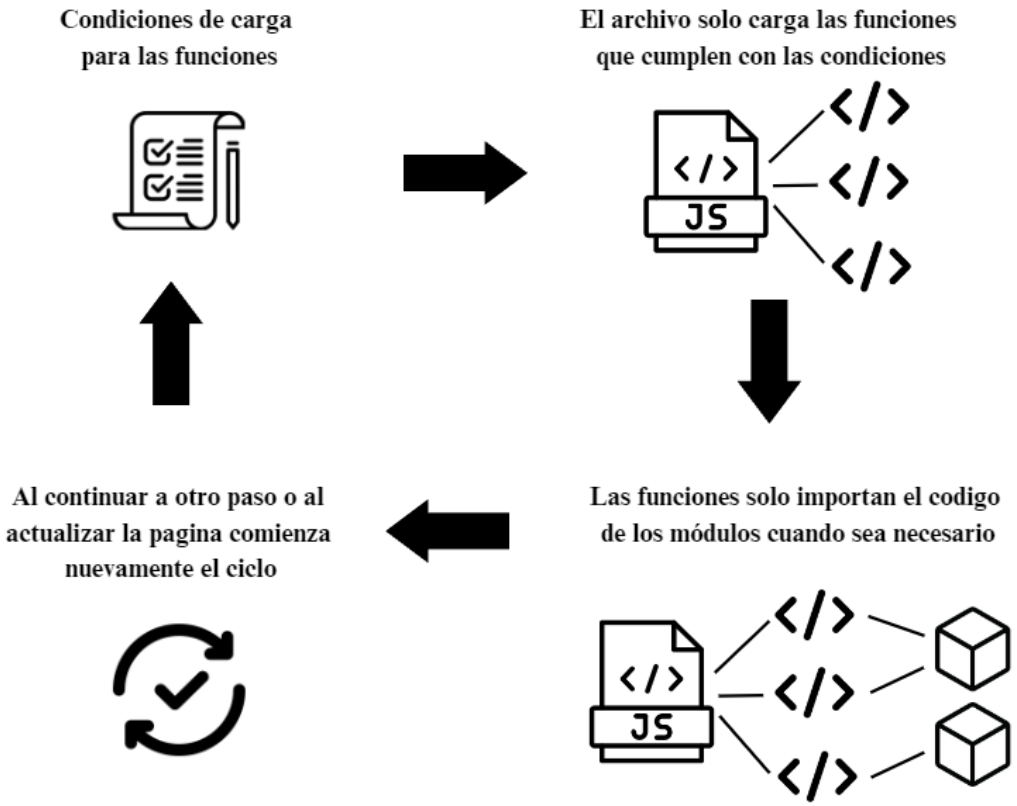
llamado a la función de importación de módulos donde el parámetro principal corresponde a la ubicación exacta del módulo dentro del repositorio.

Finalmente, con la palabra reservada ***then*** resuelve la promesa asociada al uso de código asíncrono. Para el caso puntual de la función anterior, se usó el concepto de ***destructuring*** con el objetivo de traer del módulo la función que se requiere y no todas directamente. Un aspecto que se debe tener en cuenta es que, a pesar de solo traer una función del módulo este será cargado completamente llegado ese punto, por lo tanto, al traer solo una función específica se busca mantener principalmente un código claro al saber cuál funcionalidad fue la necesaria.

Gracias a usar código asíncrono el código de las funcionalidades del archivo sólo será cargado en el momento que sea requerido, algo que resulta imposible de hacer si solo se usa importaciones normales y código síncrono. Por otro lado, para lograr el mayor rendimiento es necesario establecer condiciones que permitan filtrar aún más el código que realmente va a ser necesario, logrando de esta manera reducir al mínimo la cantidad de código javascript innecesario. En la **Figura 43**, se permite ver de manera sencilla cuál es la lógica que se tuvo en cuenta para lograr este objetivo.

Figura 43.

Cómo se carga el código del checkout después de la modularización .



La anterior figura corresponde al proceso de carga de los módulos, este proceso se organizó en 4 fases las cuales corresponde a las siguientes:

1. **Condiciones de carga:** Principalmente se deben tener unas condiciones definidas que permitan saber en qué momento se debería cargar cada una de las funcionalidades, en base a las condiciones una función aparte se encargará de cargar al usuario solo las funciones que requiere.

2. **Funciones cargadas:** Gracias a las condiciones establecidas el usuario sólo recibirá un archivo con las funcionalidades que requiere, que para este proyecto corresponde a las funcionalidades requeridas para el paso específico, así como las que no fueron tomadas en cuenta y por lo cual siempre deberán ser cargadas.

3. **Módulos:** Las funciones que fueron tomadas en cuenta para el proceso de modularización no tendrían realmente código por dentro, estas funciones solo cargaran el código que necesitan para funcionar cuando sean invocadas, gracias a esto la carga del código dentro de los módulos se retrasara lo más posible reduciendo así gran parte de código que probablemente no se llegue a usar.

4. **Nueva página:** Cada vez que el usuario recargue la página o vaya a un nuevo paso dentro del proceso de checkout el servidor le devuelve una nueva página, por consiguiente, cada vez que ocurra esto se vuelve a comenzar desde la primera fase donde nuevamente se revisan las condiciones y se establecen cuáles son las funciones que se deben cargar al usuario basado en su nueva ubicación.

Para completar este enfoque aún es necesario implementar las condiciones de carga que permitan cumplir con lo planteado anteriormente. En la siguiente sección entonces se abordará el cómo fueron implementadas estas condiciones y se revisarán brevemente una vez más el razonamiento detrás de estas.

4.4 Variables usadas para las condiciones de carga

Hasta el momento no se encuentran disponibles las condiciones que permitan reducir la carga de código innecesario, pero gracias a la notación de los módulos, como se observa en la **Figura 44**, fue posible implementar las condiciones definidas anteriormente, de una forma relativamente sencilla. Actualmente los módulos no se están importando al comienzo del archivo como sucede con los import estáticos, los módulos se importan asíncronamente en el momento que se cumplan los criterios necesarios, siendo esta la principal ventaja de usar módulos dinámicos.

Figura 44.

Import estático vs dinámico.

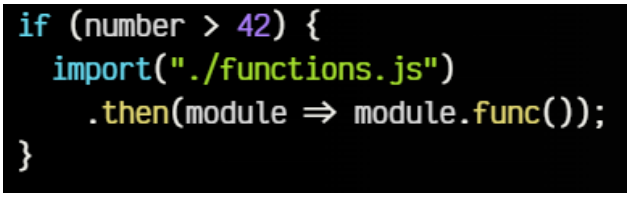
Nota. Tomado de Manz. (s.f)

Gracias a la estructura usada en el import dinámico es posible integrar condiciones que permitan cargar el código cuando se cumpla una condición específica, como, por ejemplo, cuando un usuario interactúe con un botón, cuando se llega a un valor específico o como en el caso actual, cuando el usuario se encuentre en un paso en específico del checkout.

Algunas formas de implementar estas condiciones son bastantes sencillas como se pueden ver en la **Figura 45** y en la **Figura 46**.

Figura 45.

Carga de módulo al cumplir una condición.



Nota. Tomado de Manz. (s.f)

Este es un ejemplo de cómo se pueden integrar fácilmente condiciones específicas a una importación dinámica, logrando con esto que la importación del módulo solo se realice al cumplir la condición definida la cual puede ser tan compleja como sea necesaria.

Figura 46.

Carga de módulo por evento.

```
const button = document.querySelector("button.info");
button.addEventListener("click", () ⇒ import("aditional.js"), { once: true });
```

Nota. Tomado de Manz. (s.f)

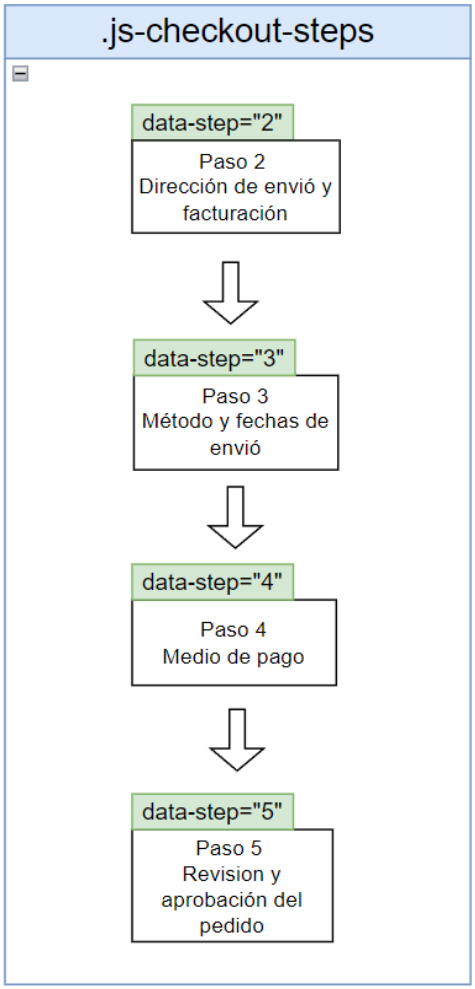
Otro de los casos más comunes es definir situaciones o eventos al momento de la importación, un ejemplo de estos casos es el visto en la figura anterior, cuando un usuario interactúa con un botón en específico entonces se realizará la importación del módulo.

Con el objetivo de implementar este tipo de cargas dinámicas es necesario previamente haber realizado un análisis que permita establecer correctamente cuando es necesario cada módulo. Gracias al análisis del proyecto realizado previamente, se llegó a la conclusión de que las condiciones de carga para los módulos debían estar dadas por la ubicación del usuario dentro de los pasos del flujo del proceso de checkout, de esta forma se logrará retrasar lo más posible la descarga y procesamiento de código por parte del navegador.

Lograr el enfoque planteado anteriormente requería de variables o elementos que permitirán saber con certeza la posición del usuario en dicho proceso, para esto se dispusieron de clases específicas en la estructura HTML. Estas clases fueron tomadas en cuenta como elementos clave que permitirán conocer la ubicación del usuario dentro del checkout. En la **Figura 47** se puede observar más claramente las clases dispuestas:

Figura 47.

Clases html usadas para la implementación de las condiciones de carga.

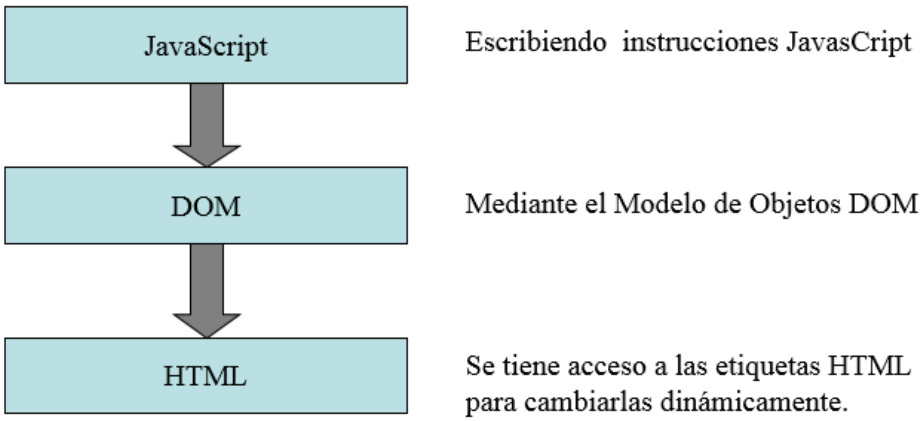


Como se puede observar en la imagen anterior, la estructura HTML usada para estas condiciones se conforma principalmente de un contenedor grande con la clase ***"js-checkout_steps"***, el cual se encarga de mostrar la información de cada uno de los pasos del checkout. Adicionalmente, dentro del contenedor principal, existen contenedores individuales para cada uno de los pasos, y cada uno de estos además tiene definido un atributo llamado ***"data-step"***, el cual representa una identificación única para los pasos.

Para identificar en qué paso se encontraba el usuario en cada momento se usaron clases que controlan el estado de cada uno de los contenedores . Estas clases son ***"active"*** y ***"complete"*** los cuales gracias a la manipulación del DOM (ver **Figura 48**) podían ser consultadas y cambiadas según fuera requerido.

Figura 48.

Manipulación del DOM.



Nota. Tomado de cetinformatica. (s.f)

Gracias a esto, era posible saber qué pasos ya había completado el usuario, además de en cuál paso se encontraba actualmente. Con la ayuda de estas clases y además del atributo ***"data-step"***, se podrían establecer las condiciones que permitan cargar al usuario solo los módulos útiles para cada paso.

Por otro lado, se puede observar que el primer paso es omitido en la representación anterior. Esto se debe a que el primer paso forma parte del punto de inicio del flujo del checkout. Por lo cual, establecer condiciones para el primer paso no sería de utilidad, ya que para este paso siempre se deberán cargar las funcionalidades que no cumplieron con las condiciones de los otros 4 pasos establecidos. O, por otro lado, cualquier funcionalidad en la cual no se tuviera

una total claridad de cuándo debe ser cargada, deberá ser cargada desde un inicio con el fin de evitar problemas en el funcionamiento del checkout.

4.5 Implementación de las condiciones para la carga dinámica de los módulos

Durante el desarrollo del proyecto, el equipo de performance se encontraba en la fase de implementación de una funcionalidad destinada a revisar las funciones presentes en un archivo y, basándose en ciertas condiciones preestablecidas, cargar únicamente aquellas que podrían ser utilizadas por el usuario.

El enfoque planteado en este proyecto optaba por la modularización para mejorar el rendimiento, permitiendo que los módulos se cargaran dinámicamente según la demanda. Por otro lado, la función desarrollada por el equipo de performance funcionaba mediante la construcción de un archivo que contenía únicamente las funciones requeridas por el usuario.

Aprovechando la nueva funcionalidad disponible se optó por que las condiciones no decidieran en qué momento se debe cargar cada módulo, por otro lado, las condiciones establecerían unos filtros que permitirán que el usuario solo recibiera las funcionalidades necesarias. La integración de esta nueva funcionalidad con las condiciones disponibles se pudo realizar de forma simple lo que facilitó en gran medida la implementación de las condiciones.

A continuación, en la **Figura 49**, se presenta la estructura de la función diseñada para esta tarea, así como una explicación de su lógica y funcionamiento.

Figura 49.

Función implementada para cargar solo las funciones necesarias.

```
function _autoload() {
    $.each(ACC, function (section, obj) {
        if ($.isArray(obj._autoload)) {
            $.each(obj._autoload, function (key, value) {
                if ($.isArray(value)) {
                    if (value[1]) {
                        ACC[section][value[0]]();
                    } else {
                        if (value[2]) {
                            ACC[section][value[2]]();
                        }
                    }
                } else {
                    ACC[section][value]();
                }
            });
        }
```

La función denominada ***autoload*** se ejecuta al comienzo de un archivo y tendrá la tarea de determinar qué funciones dentro del archivo deben ser cargadas y cuáles no. El propósito principal de la función ***autoload*** es gestionar una lista que engloba todas las funciones presentes en el archivo en cuestión. Cada elemento de esta lista contendrá dos partes distintas: el primer componente corresponde al nombre de la función, mientras que el segundo estaría representado por un valor booleano que determinaría si la función debe ser cargada o no.

Para el correcto funcionamiento del ***autoload*** se requiere que al menos el primer elemento de la lista esté presente, el segundo elemento puede o no estar definido. En el caso de no definirse el segundo elemento entonces el ***autoload*** cargará automáticamente la función sin tener en cuenta ninguna condición.

Los 2 elementos de la lista que requiere el ***autoload*** pueden verse como los parámetros requeridos y un ejemplo de estos se puede observar en **la Figura 50**. La configuración del primer parámetro solo requiere escribir el nombre de la función por lo que es bastante sencillo definirlo, por otro lado, en cuanto a la configuración del segundo parámetro, fue necesario escribir una expresión que permitiera identificar, en función de un paso específico, si el usuario se encontraba en ese punto o no.

Para obtener esta información, se disponían de diversas técnicas que en última instancia llevaban al mismo resultado deseado. En el marco de este proyecto en particular, se optó por emplear las clases HTML mencionadas previamente. Estas clases desempeñaron un papel fundamental para lograr este objetivo.

Figura 50.

Ejemplo elementos de la lista usada por la función autoload para trabajar.

```
[
    "bankPromotionNewCreditCard",
    document.querySelector( selectors: '.js-checkout-step[data-step="4"]')?.classList.contains("active"),
],
[
    "checkoutZeroInterestModal",
    document.querySelector( selectors: '.js-checkout-step[data-step="4"]')?.classList.contains("active"),
],
```

Acá se pueden observar 2 elementos presentes en la lista usada por la función ***autoload***, cada elemento de la lista está dividido en 2 partes donde la primera corresponde al nombre de la función y la segunda a la condición que se definió para saber si cargar o no. En cuanto a la condición se hizo uso del método ***querySelector,*** este método permite identificar un elemento en concreto a través del DOM. Para lograr hacer la búsqueda fue necesario definir unos

parámetros adecuados, los cuales se caracterizan por ser cada una de las clases y atributos que se han mencionado previamente:

1. ***.js-checkout-step*:** Esta clase actúa como identificación para el primer elemento requerido, el cual corresponde al contenedor principal que aloja los contenedores individuales para cada paso.
2. ***data-step:*** Este atributo está presente en los contenedores de cada paso, y tiene un valor único en cada caso.
3. ***active:*** Esta clase se asigna al contenedor del paso cuando el usuario está en ese paso en particular.

Gracias a estos elementos es realmente sencillo definir una expresión que permita saber en qué paso se encuentra el usuario, por otro lado, con la ayuda del método denominado ***contains*** se puede identificar si la clase ***active*** se encuentra presente o no en el paso. La expresión completa entonces se puede dividir en 2 de la siguiente forma:

1. ***document.querySelector('.js-checkout-step[data-step="4"]')***

   Con la ayuda del DOM se identifica el elemento que se encuentra dentro del contenedor de pasos y que su atributo de identificación para el paso sea uno específico, que para el ejemplo corresponde al paso 4. Como resultado esta expresión retorna un elemento HTML que cumpla con las condiciones.

2. ***classList.contains("active")***

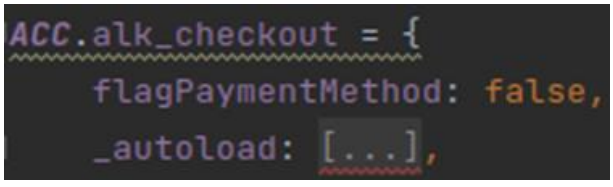
   La expresión ***classList*** retorna una lista de todas las clases del elemento HTML resultando del primer paso, luego de esa lista se verifica con el método ***contains*** sí existe la clase ***active***. Como resultado se obtiene un booleano el cual dependerá de si la clase está presente o no en la lista de clases.

Adicionalmente se debe mencionar que esta expresión hace uso de un elemento en javascript el cual es **"?"**, este se encuentra en mitad de las 2 expresiones anteriores y se asegura de que la primera expresión haya encontrado algún elemento para poder continuar con el resto, ya que dado el caso en que se intente buscar un paso, el cual no exista, el resto de la expresión no deberá ser ejecutada.

En cuanto al funcionamiento concreto de la función ***autoload***, esta aprovecha una estructura de objetos, la cual ya se encontraba en el proyecto, de esta manera el archivo trabajado ***alk_checkout*** contiene un objeto con el mismo nombre como se puede observar en la **Figura 51**, el cual es importante para el correcto funcionamiento del ***autoload***.

Figura 51.

Objeto disponible en los archivos de funcionalidad con el fin de aplicar condiciones.

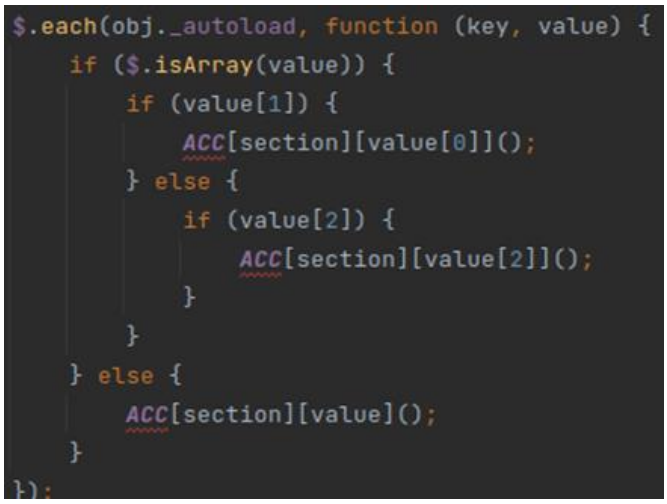


Este objeto tiene un atributo llamado ***_autoload*** el cual corresponde a una lista, esta lista es la que requiere la función ***autoload*** para su funcionamiento, además de esto, el objeto

tiene definidas todas las funciones dentro, lo cual es de gran importancia ya que gracias a esto es posible decidir con el ***autoload*** cuales de esas funciones se tendrán en cuenta y cuáles no en la construcción del archivo. En la **Figura 52**, se presentan una vista más específica de la función ***autoload*** donde se recorren los elementos de la lista.

Figura 52.

Función autoload específicamente el código que recorre los elementos de la lista.



```
$.each(obj._autoload, function (key, value) {
    if ($.isArray(value)) {
        if (value[1]) {
            ACC[section][value[0]]();
        } else {
            if (value[2]) {
                ACC[section][value[2]]();
            }
        }
    } else {
        ACC[section][value]();
    }
});
```

En cuanto al funcionamiento del ***autoload*** no es tan intuitivo de entender, el flujo completo del código se puede describir en los siguientes pasos:

1. ***$.each(ACC, function (section, obj) {...}***

   Esta línea de código hace iteraciones a través de las propiedades del objeto ***ACC,*** donde cada propiedad representa un objeto correspondiente a un archivo de funcionalidad. El parámetro ***section*** es el nombre de la propiedad y ***obj*** es el valor asociado (que es otro objeto).

2. ***if ($.isArray(obj._autoload)) {...}***

   Comprueba si la propiedad ***_autoload*** de ***obj*** es un array. Si no llegara a existir ningún elemento dentro del ***_autoload*** no se cargará ninguna de las funcionalidades y simplemente acabaría el flujo de la función ***autoload.***

3. ***$.each(obj._autoload, function (key, value) {...}***

   Si ***_autoload*** es un array, este bucle recorre cada elemento dentro del array. ***key*** representaría el índice de elemento y ***value*** es el valor en ese índice. Un ejemplo de un elemento del autoload se pudo observar en la figura 49.

Dentro de este bucle, hay 2 condiciones que ayudarán a determinar si la función debe ser cargada o no al usuario:

a***. if ($.isArray(value)) {...}***

Verifica si el valor actual es un array. Si ***value*** no es un array, simplemente ejecuta la función asociada al valor ***value*** dentro del objeto ACC. En otras palabras, si no se establece una condición para la carga de esa función siempre será cargada al usuario.

b***. if (value[1]) {...}***

Si el segundo elemento de value (value[1]) en el array es verdadero, esto quiere decir que la condición asociada se cumplió por lo que esta función si será cargada al usuario

La función ***autoload*** fue creada con ayuda del equipo de trabajo para lograr el mejor resultado de tal forma que puede ser usada por cualquier archivo y sección dentro de la plataforma, lo cual la hace reutilizable y estandarizada. Por otro lado, esto no involucra el resto de los aspectos tenidos en cuenta en este proyecto, a pesar de poder usar la función ***autoload*** en cualquier archivo aún es necesario definir condiciones adecuadas que permitan conocer cuándo deben ser cargadas las funciones, además de esto las funciones que llegarán a cumplir las condiciones no estarían modularizadas lo que provocaría que el código completo de las funcionalidades está disponible desde el principio y no cuando se requiera.

Finalmente, la modularización estaría finalizada, se pudo lograr un enfoque en el cual el código cargado se mantiene solo a lo estrictamente necesario, gracias a la combinación del importe dinámico de módulos y a unas condiciones apropiadas se pudo lograr disminuir en gran medida el tamaño del archivo y por consiguiente el tiempo. En la siguiente sección se revisarán los últimos detalles de la implementación realizada y de igual manera se presentarán los análisis realizados con el fin de comprender cuáles fueron los resultados de la modularización, así como las pruebas realizadas para validar el correcto funcionamiento del checkout.

5. Correcciones, validaciones y pruebas de rendimiento

Una vez finalizada la creación de los módulos, haberlos ubicado en un lugar adecuado e implementar las funciones necesarias para el importe, se procedió a realizar correcciones y validaciones con el objetivo de validar que todas las funcionalidades que hayan sido modificadas funcionaran correctamente.

Esta etapa consistió en la fase final del proyecto y fue de mucha importancia ya que el proceso modularización no solo implicó trasladar el código de un lugar a otro, fue necesario reescribir código además de reestructurar directamente otras secciones de código. Por estas razones validar el correcto funcionamiento era prioritario ya que de las funciones que se trabajaron dependía gran parte del funcionamiento del checkout.

Además de la validación del correcto funcionamiento, se realizaron nuevamente las pruebas de rendimiento que se llevaron a cabo al inicio de este proyecto. Estas pruebas permitieron evaluar el impacto que han tenido las acciones realizadas en el rendimiento del checkout y sacar conclusiones sobre los resultados obtenidos.

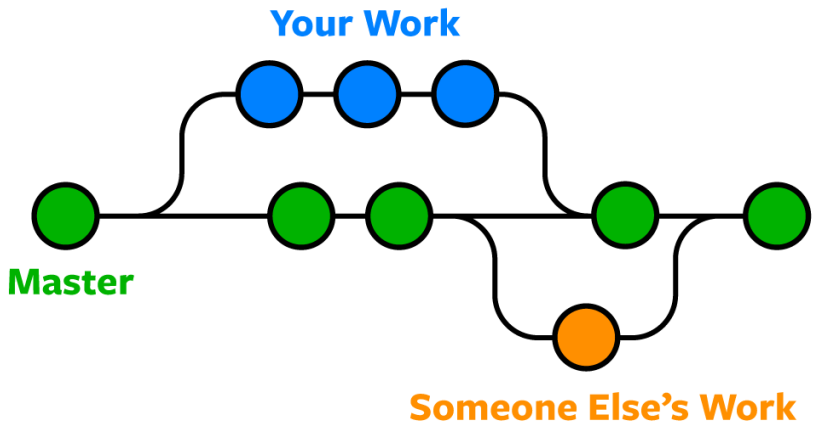
En resumen, en la siguiente sección se abordará la última parte del proceso de modularización, donde se realizaron las correcciones finales, se validó el correcto funcionamiento y llevaron a cabo las pruebas de rendimiento.

5.1 Correcciones finales

Durante el desarrollo del proyecto, las modificaciones se realizaron dentro de una rama creada especialmente para el proyecto, esta rama de git permitió obtener un entorno aparte donde era posible experimentar con las funcionalidades sin la preocupación de dañarlas definitivamente. La rama en la que se trabajó es una copia de la rama principal, conocida como ***master***, que contiene todas las funcionalidades aprobadas e implementadas (ver **Figura 53**). Conforme avanzaba el proyecto, la rama "master" se actualizaba constantemente con el nuevo código aportado por otros desarrolladores. Un ejemplo de esta situación se puede describir con la siguiente figura.

Figura 53.

Ejemplo manejo de las ramas en git .



Nota. Tomado de nobledesktop. (2021)

Al finalizar la implementación de la modularización, se hizo necesario incorporar todas las modificaciones realizadas por los demás desarrolladores a lo largo de este periodo. Las correcciones se centraron en resolver cualquier conflicto que pudiera surgir entre las diferentes versiones, siempre asegurando que el funcionamiento no se viera comprometido.

Algunas de estas correcciones se llevaron a cabo al concluir el proyecto, pero no todas se postergaron hasta el final. Periódicamente, se integraban los cambios realizados por otros desarrolladores con el código disponible en la rama correspondiente, evitando acumularlos y así simplificando el proceso final. Además, es importante destacar que las correcciones se enfocaron exclusivamente en los archivos que presentaban conflictos o inconsistencias, principalmente en el archivo en cuestión y algunos relacionados con él.

En su mayoría, estos cambios consistieron en ajustar líneas de código o modificar funciones específicas. La complejidad de estos ajustes fue relativamente baja, en gran parte debido a que

las modificaciones semanales eran de pequeña escala y cualquier problema importante que surgiera tenía la posibilidad de ser abordado en la reunión scrum semanal.

A lo largo del desarrollo del proyecto, se llevaron a cabo pruebas de aceptación de manera constante para garantizar el correcto funcionamiento de los módulos desarrollados. Estas pruebas también se realizaron durante las correcciones finales mencionadas. En la siguiente sección, se abordarán las implicaciones de estas pruebas.

5.2 Pruebas de aceptación

El funcionamiento del proceso de checkout y sus componentes ya había sido definido previamente, por lo cual solo era necesario identificar los criterios de aceptación que habían sido considerados durante el desarrollo y, a partir de estos criterios, definir casos concretos que permitieran verificar el correcto funcionamiento.

Toda la información necesaria podía encontrarse en las "historias de usuario". Estas historias habían sido utilizadas desde el inicio del proyecto y se asignaban como tareas a cada uno de los desarrolladores. Además, gracias al uso de la herramienta Jira para la gestión de proyectos y mediante una búsqueda adecuada, era posible acceder a las historias que se requerían.

La importancia de estas historias de usuario radica en la información que contienen. A través de ellas, se facilita el entender la razón de ser de cada desarrollo, los criterios de aceptación que se establecieron, el desarrollador responsable y muchos otros detalles que facilitan la comprensión de las implicaciones de cada funcionalidad.

Figura 54.

Ejemplo de historia de usuario.

Historia de Usuario	
Número: 1	**Usuario:** Cliente
Nombre historia: Cambiar dirección de envío	
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Baja
Puntos estimados: 2	**Iteración asignada:** 1
Programador responsable: José Pérez	
Descripción: Quiero cambiar la dirección de envío de un pedido.	
Validación: El cliente puede cambiar la dirección de entrega de cualquiera de los pedidos que tiene pendientes de envío.	

Las pruebas de aceptación se llevaron a cabo de manera simultánea al desarrollo del proyecto, aprovechando la información disponible en las historias de usuario. Sin embargo, lo que realmente permitía la realización de pruebas adecuadas eran los criterios de validación. Estos criterios eran esenciales para garantizar que las pruebas reflejaran adecuadamente el comportamiento esperado de cada elemento del proceso de checkout.

Basado en la información contenida en las historias de usuario y en los criterios de validación, se definieron casos específicos de pruebas de aceptación. Estos casos de prueba proporcionaban información sobre el comportamiento que se esperaba de una funcionalidad. A continuación, se presentan los casos de prueba correspondientes a los escenarios más relevantes.

Tabla 2.
Caso prueba de aceptación: Limpiar inputs de los métodos de pago.

Código: HU1_P01	**Historia de usuario:** HU_1
Nombre: Limpiar campos al cambiar de método de pago.	
Descripción: Se probará que cuando el usuario navegue entre los diferentes medios de pago, todos los campos se restablezcan al salir y entrar nuevamente.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 4 del checkout.	
Pasos de Ejecución: Al llegar al paso 4 del proceso de checkout el usuario selecciona un método de pago y completa el formulario. Posteriormente el usuario selecciona otro método de pago para después volver a entrar nuevamente al elegido inicialmente.	
Resultado esperado: Al entrar nuevamente al método elegido inicialmente, todos los campos completados deben estar vacíos. Adicionalmente cualquier estado del campo ya sea confirmado (color verde) o error (color rojo) debe ser revertido a su estado original.	
Evaluación de la prueba: Los campos están vacíos al entrar nuevamente al método elegido inicialmente, pero el estado de error (color rojo) permanecía siempre visible para los campos con errores. Problema corregido	

Tabla 3.
Caso prueba de aceptación: Notificar campos incorrectos en métodos de pago.

Código: HU2_P01	**Historia de usuario:** HU_2
Nombre: Notificar campos incorrectos en métodos de pago.	
Descripción: Se probará si se está notificando cuando los campos son completados incorrectamente en los métodos de pago.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 4 del checkout.	
Pasos de ejecución: Al llegar al paso 4 del proceso de checkout se selecciona un método de pago y se llenan los campos con información incorrecta, finalmente el usuario selecciona “continuar”.	
Resultado esperado: No se continúa al siguiente paso del proceso de checkout. Los campos involucrados en el error aparecen con un borde rojo, además debajo del campo se presenta un texto también en rojo explicando el motivo del error.	
Evaluación de la prueba: Los campos “nombre” y “número de tarjeta” no muestran el texto de error asociado. Problema corregido	

Tabla 4.

Caso prueba de aceptación: Validación tarjeta de crédito bloqueada.

Código: HU2_P02	**Historia de usuario:** HU_2
Nombre: Validar tarjeta de crédito bloqueada.	
Descripción: Se validará que la información ingresada de la tarjeta de crédito no corresponda a una tarjeta bloqueada.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 4 del checkout.	
Pasos de ejecución: Llegado al paso 4 del proceso de checkout el usuario elige el método de pago "tarjeta de crédito", en el campo "número de tarjeta" se usa la información de una tarjeta bloqueada, finalmente el usuario selecciona "continuar".	
Resultado esperado: No se continúa al siguiente paso del proceso de checkout, bajo el campo "número de tarjeta" se muestra un texto en rojo que explique que la tarjeta está bloqueada.	
Evaluación de la prueba: El texto de error para el campo no se muestra. Problema corregido	

Tabla 5.

Caso prueba de aceptación: Validación fecha de expedición.

Código: HU2_P03	**Historia de usuario:** HU_2
Nombre: Validación fecha de expedición.	
Descripción: Se probará que en el campo "fecha de expedición" para el método de pago "tarjeta de crédito" la validación de la fecha ingresada sea correcta.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 4 del checkout.	
Pasos de ejecución: Llegado al paso 4 del proceso de checkout el usuario elige el método de pago "tarjeta de crédito" y se ubica en el campo "fecha de expedición". Se pueden dar los siguientes 2 casos: 1. El usuario ingresa una fecha ya vencida. 2. El usuario ingresa una fecha que no corresponde a la tarjeta, completa el formulario y selecciona "continuar".	
Resultado esperado: 1. Al momento de completar el campo se notifica al usuario inmediatamente, a través de un borde y un texto rojos con el mensaje "fecha de expedición ya vencida". 2. No se continúa al siguiente paso del proceso de checkout, bajo el campo "fecha de expedición" se muestra un texto en rojo que explique qué fecha no corresponde.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 6.
Caso prueba de aceptación: Validación CVV para tarjeta guardada.

Código: HU2_P04	**Historia de usuario:** HU_2
Nombre: Validar el campo CVV para tarjeta guardada.	
Descripción: Se probará que la validación del campo CVV sea correcta para el caso que el usuario utilice una tarjeta de crédito guardada como medio de pago.	
Condiciones de ejecución: El usuario tiene que estar autenticado, tener una tarjeta de crédito guardada y **encontrarse** en el paso 4 del checkout.	
Pasos de ejecución: Al llegar al paso 4 del proceso de checkout se selecciona método de pago "tarjeta de crédito" y se elige una tarjeta guardada. El usuario completa el campo "CVV" incorrectamente.	
Resultado esperado: No se continúa al siguiente paso del proceso de checkout y se notifica al usuario del problema visualmente.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 7.
Caso prueba de aceptación: Manejo pestañas laterales en medios de pago.

Código: HU3_P01	**Historia de usuario:** HU_3
Nombre: Manejo pestañas laterales en medios de pago	
Descripción: Se probará que cada una de las pestañas laterales haga visible los campos relacionados a el método de pago correspondiente.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 4 del checkout.	
Pasos de ejecución: Llegado al paso 4 del proceso de checkout en el lateral izquierdo hay pestañas para cada uno de los métodos de pago, el usuario navega entre los diferentes métodos de pago.	
Resultado esperado: Al navegar por los métodos de pago la vista se debe actualizar con la información del método de pago seleccionado.	
Evaluación de la prueba: El método de pago "tarjeta alkomprar" no muestra nada al ser seleccionado en la pestaña lateral. Problema corregido	

Tabla 8.

Caso prueba de aceptación: Ciudades disponibles por departamento.

Código: HU4_P01	**Historia de usuario:** HU_4
Nombre: Ciudades disponibles por departamento.	
Descripción: Se probará que cuando el usuario esté llenando la información de envío las opciones de ciudades correspondan a las del departamento seleccionado.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 2 del checkout.	
Pasos de ejecución: Llegado al paso 2 del proceso de checkout el usuario se ubica en el campo "departamento" y elige un departamento disponible en la lista desplegable.	
Resultado esperado: La lista desplegable para el campo "ciudad" se actualiza con las ciudades correspondientes al departamento seleccionado.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 9.

Caso prueba de aceptación: Validar disponibilidad del producto.

Código: HU5_P01	**Historia de usuario:** HU_5
Nombre: Validar disponibilidad de producto.	
Descripción: Se validará que, al ingresar una ciudad con dirección de envío no disponible para un producto, se notifique al usuario y se interrumpa el proceso.	
Condiciones de ejecución: El usuario tiene que estar autenticado, en el paso 2 del checkout y tener en el carrito un artículo con envió solo disponible a "Bogotá".	
Pasos de ejecución: Llegado al paso 2 del proceso de checkout el usuario elige la opción "añadir dirección de envío" y completa el formulario eligiendo ciudad a "Bucaramanga" y selecciona finalmente "continuar".	
Resultado esperado: No se continúa al siguiente paso del proceso de checkout, se muestra una notificación con la información de que "Bogotá" es la única ciudad disponible para el envío del producto.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 10.

Caso prueba de aceptación: Métodos de pago especiales.

Código: HU6_P01	**Historia de usuario:** HU_6
Nombre: Métodos de pago especiales.	
Descripción: Se validará que los métodos de pago especiales se muestren correctamente y solo sean visibles para los asesores.	
Condiciones de ejecución: El usuario tiene que estar autenticado como asesor y encontrarse en el paso 4 del checkout	
Pasos de ejecución: Llegado al paso 4 del proceso de checkout el asesor puede observar un menú lateral con todos los métodos de pago disponibles.	
Resultado esperado: Entre los métodos de pago disponibles se deben encontrar 2 métodos especiales los cuales son "generar link" y "link manual" los cuales se deben encontrar hasta el final de las opciones disponibles.	
Evaluación de la prueba: Los métodos de pago son visibles, pero no se encuentran hasta el final de las opciones disponibles. Problema corregido	

Tabla 11.

Caso prueba de aceptación: Dirección de envío y facturación diferente.

Código: HU7_P01	**Historia de usuario:** HU_7
Nombre: Dirección de envío y facturación diferente.	
Descripción: Se validará que al tener direcciones de envío y facturación diferentes se muestre un nuevo formulario donde se pueda definir la otra dirección.	
Condiciones de ejecución: El usuario tiene que estar autenticado y en el paso 2 del checkout.	
Pasos de ejecución: Llegado al paso 2 del proceso de checkout el usuario selecciona la opción "agregar nueva dirección", completa el formulario y luego marca la opción "dirección de facturación diferente".	
Resultado esperado: Se muestra un segundo formulario debajo del formulario de dirección de envío con los mismos campos para ambos formularios.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 12.

Caso prueba de aceptación: Editar direcciones de envío guardadas.

Código: HU7_P02	**Historia de usuario:** HU_7
Nombre: Editar direcciones de envío guardadas.	
Descripción: Se validará que la opción editar esté disponible para las direcciones previamente guardadas.	
Condiciones de ejecución: El usuario tiene que estar autenticado, debe tener direcciones de envío previamente guardadas.	
Pasos de ejecución: El usuario selecciona un producto, lo agrega al carrito y se dirige al proceso de checkout hasta llegar al paso 2 correspondiente a dirección de envío y facturación.	
Resultado esperado: En el paso 2 se muestra una lista con todas las direcciones guardadas previamente, al seleccionar alguna de ellas se presentará la opción "editar" en la cual el usuario podrá cambiar datos en la dirección y además tendrá la opción "dirección de facturación diferente" al igual que al agregar una dirección nueva.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 13.

Caso prueba de aceptación: Soft-login visibilidad de información limitada.

Código: HU8_P01	**Historia de usuario:** HU_8
Nombre: Soft-login visibilidad de información limitada.	
Descripción: Se validará que cuando el usuario ingrese como Soft-login (Ingresar solo con correo y sin contraseña) la visibilidad de la información en el checkout esté limitada.	
Condiciones de ejecución: El usuario tiene que ingresar como Soft-login y debe tener información guardada de dirección de envío, datos de usuario y tarjetas.	
Pasos de ejecución: El usuario selecciona un producto, lo agrega al carrito y se dirige al proceso de checkout.	
Resultado esperado: El resumen de la información ya guardada para cada uno de los datos debe estar encriptada, cuando el usuario seleccione "editar datos" o" ingresar otros datos" aparecerá una notificación que indique que debe terminar de autenticarse con las opciones "continuar" o "regresar al checkout".	
Evaluación de la prueba: Prueba satisfactoria.	

Estos casos conforman diversas funcionalidades dentro del proceso de checkout, abarcando cada uno de los pasos y especialmente tomando las funcionalidades más relevantes. De esta manera, se concluyen las pruebas de todas las funcionalidades que fueron modificadas, asegurando que el funcionamiento cumple con los criterios establecidos desde el principio.

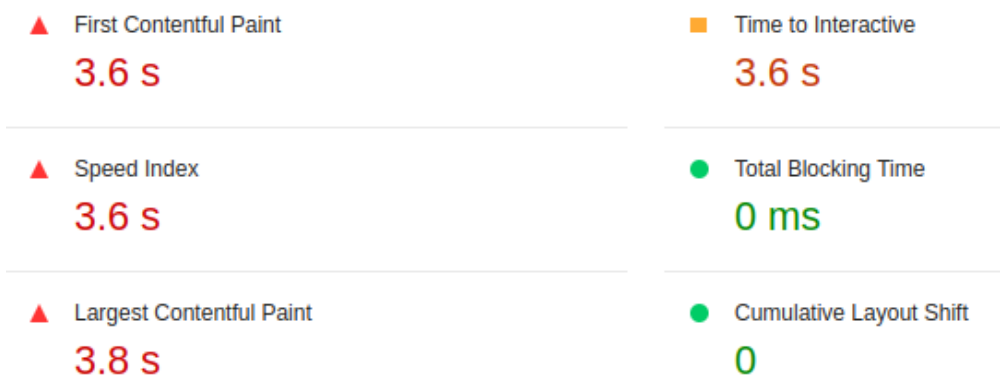
5.3 Pruebas de rendimiento

Con el objetivo de validar el impacto de la modularización en el proyecto se requirió de unas pruebas de rendimiento similares a las realizadas al comienzo del proyecto y presentadas al inicio de este informe, debido a la naturaleza del proyecto la forma más adecuada de analizar los resultados fue realizar una comparación entre los obtenidos antes y después del proceso de modularización. En base a estas comparaciones fue más sencillo observar cuales fueron realmente las implicaciones en el rendimiento que tuvo la modularización.

Las pruebas que se realizaron en esta sección fueron exactamente las mismas realizadas anteriormente, estas pruebas consistieron principalmente en usar la herramienta Lighthouse dentro de las chrome dev tools para analizar los resultados de métricas de rendimiento. La primera prueba realizada fue basada en los resultados individuales de cada una de las métricas de rendimiento, a continuación, en la **Figura 55**, se podrán observar cuáles fueron los resultados obtenidos previo al proceso de modularización.

Figura 55.

Resultados métricos de lighthouse para el checkout antes de la modularización.

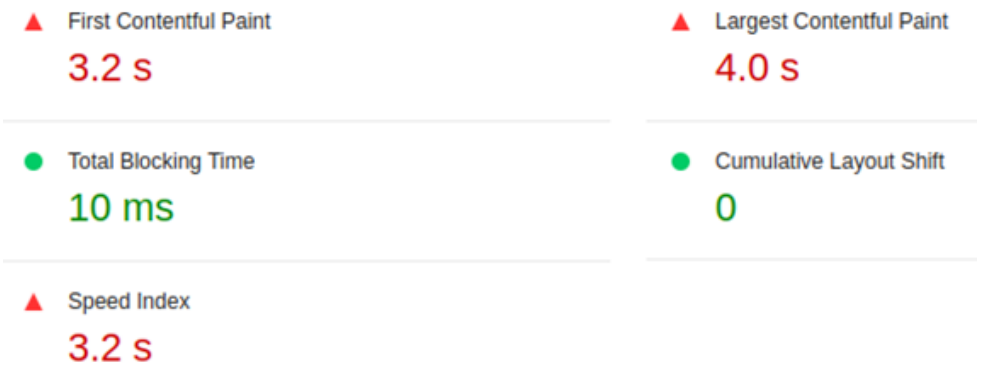


Nota. Tomado de Google Chrome, lighthouse (2023).

Durante el análisis de las métricas posterior a la modularización, la herramienta lighthouse no permitió ver la información asociada a la métrica ***Time to interactive***, debido a esta situación en la **Figura 56** se podrá notar la ausencia de esta métrica.[1]

Figura 56.

Resultados métricos de lighthouse para el checkout después de la modularización.



Nota. Tomado de Google Chrome, lighthouse (2023).

[1] Durante el análisis posterior a la modularizarían, la configuración predeterminada de la herramienta lighthouse cambio de cómo se presentaba en un comienzo, por esta razón la métrica ***Time to interactive*** no fue visible.

Los resultados obtenidos son levemente diferentes a los observados antes de la modularización, se obtuvieron unas pequeñas mejoras en algunas métricas, pero también se puede evidenciar que la métrica ***Largest Contentful Paint*** presenta un resultado menos favorable respecto al análisis anterior.

En la **Figura 57**, se presenta una comparación de los resultados obtenidos antes y después de la modularización:

Figura 57.

Comparativa de los resultados de las métricas de rendimiento.

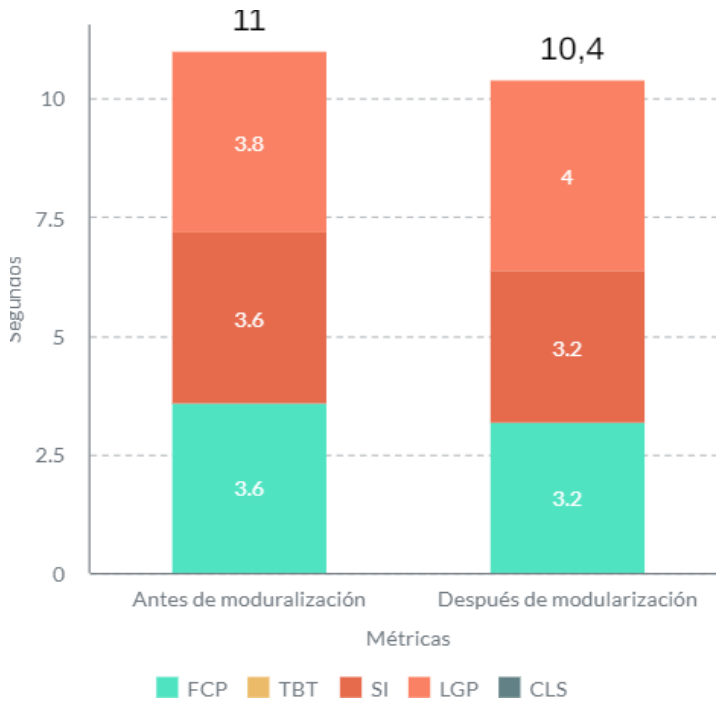
Métricas	**Antes de modularizar**	**Después de modularizar**	**Diferencia**
First Contentful Paint	3.6 segundos	3.2 segundos	- 0.4 segundos
Total Blocking Time	0 segundos	0.01 segundos	+ 0.01 segundos
Speed Index	3.6 segundos	3.2 segundos	- 0.4 segundos
Largest Contentful Paint	3.8 segundos	4 segundos	+ 0.2 segundos
Cumulative Layout Shift	0 segundos	0 segundos	0 segundos
Total	11 segundos	10.4 segundos	-0.6 segundos

En la tabla es más fácil observar cuales métricas fueron las que obtuvieron una mejora, las cuales son las métricas **First Contentful Paint** y **Speed Index.** También, en la **Figura 58**, se incluye un total del tiempo total de todas las métricas, que, aunque estas métricas son

independientes una de la otra, el total permite identificar en términos generales si la prueba de rendimiento tuvo mejores resultados antes o después.

Figura 58.

Tiempo acumulado por cada una de las métricas antes y después de la modularización.



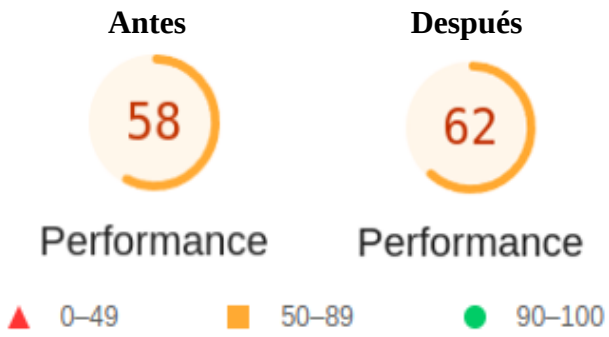
El tiempo total resultado de agrupar los tiempos de cada una de las métricas corresponde a 11 segundos para antes de la modularización y de 10,4 para después de esta. Como se mencionó anteriormente la métrica ***Time to interactive*** no fue posible obtenerla por lo que la diferencia podría variar más o menos dependiendo del resultado de esta, pero con los datos que se disponían se observa una disminución de 0.6 segundos después de realizar la modularización.

Lighthouse adicionalmente a evaluar el rendimiento de la sección con métricas individuales, también asigna un puntaje de rendimiento general para la sección, este puntaje se calcula específicamente para la sección analizada a través de ciertas pruebas definidas

internamente. A continuación, en la **Figura 59**, se podrán observar los resultados para el antes y después:

Figura 59.

Puntuación de performance por lighthouse para la sección del checkout antes vs después.



Nota. Tomado de Google Chrome, lighthouse (2023).

En base al puntaje obtenido para una sección de la plataforma, lighthouse asocia el resultado a alguno de los 3 grupos que dispone, estos se dividen en: pobre, necesita mejoras y bueno. Actualmente la diferencia es de 4 puntos respecto a los resultados obtenidos antes de la modularización lo cual no es una mejora significativa ya que aún se asigna el rendimiento en el grupo de *"necesita mejoras"*. Por otro lado, la mejora está presente y si tiene un impacto para mejorar el rendimiento de la sección, a pesar de que quizá se deba tomar un enfoque más amplio para poder observar mejoras significativas, los resultados obtenidos y el proyecto realizado permitirá definir la viabilidad del proceso de modularización.

Además de los datos de rendimiento vistos hasta el momento, lighthouse también dispone de una sección la cual presenta oportunidades de mejora, estas oportunidades mencionan diferentes acciones que ayudarían a mejorar el rendimiento además del ahorro

estimado de aplicar dichas mejoras. En la **Figura 60**, se presentan las oportunidades disponibles antes de iniciar el proceso de modularización.

Figura 60.

Oportunidades de mejora en el checkout antes de la modularización.

La gráfica contenía cada una de las oportunidades que lighthouse consideraba apropiadas, además de incluir el tiempo que se podría ahorrar de aplicar la oportunidad.

Estas oportunidades se asocian a casos concretos dentro de la sección de la plataforma que no pudieron pasar ciertas pruebas definidas por lighthouse. A continuación, en la **Figura 61**, se presentan las pruebas realizadas después de la modularización.

Figura 61.

Auditorías aprobadas en la sección del checkout después de la modularización .

Nota. Tomado de Google Chrome, lighthouse (2023).

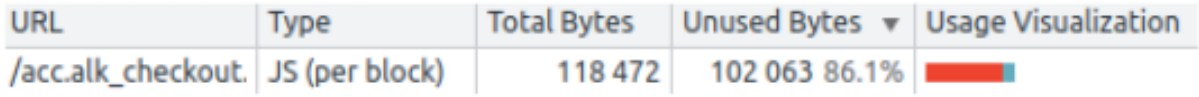
Como se observa las pruebas respecto al uso de código de javascript fueron satisfactorias, se pueden observar que aún existen ahorros potenciales, pero estos no representan una gran mejora. Adicionalmente un aspecto a destacar es que las únicas 2 oportunidades que lograron superar las pruebas y que se encontraban presentes en la gráfica de oportunidades son las relacionadas a reducción de código javascript. Debido a que el proceso de modularización tenía un enfoque exclusivo a la reducción de javascript es normal que estas oportunidades sean las únicas que en el nuevo análisis hayan superado las pruebas, y el hecho de pasar estas pruebas es un buen indicador de que los módulos realmente si ayudan mejoran el rendimiento de la sección.

La implementación de la modularización como se mencionaba anteriormente redujo en gran medida la cantidad de código javascript cargado desde un inicio, esta reducción de código se ve especialmente reflejada en el tamaño del archivo que se estaba trabajando (*alk_checkout*), adicionalmente el porcentaje de código sin utilizar también se vería afectado ya que todo el código definido en los módulos solo se cargaría al ser necesario. A continuación, en la **Figura**

62, se puede ver nuevamente cuál era el tamaño del archivo en un principio, además de cuál era el porcentaje de ese archivo que no era utilizado para el caso específico del segundo paso.

Figura 62.

Información del archivo alk_checkout antes de la modularización.



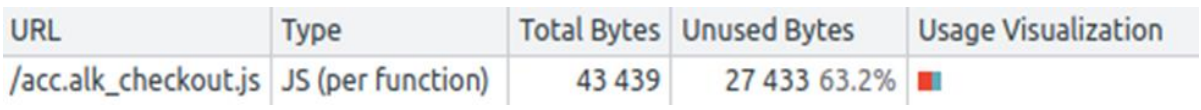
URL	Type	Total Bytes	Unused Bytes ▼	Usage Visualization
/acc.alk_checkout.	JS (per block)	118 472	102 063 86.1%	

Nota. Tomado de Google Chrome, lighthouse (2023).

El tamaño total del archivo correspondía a 118.472 Bytes lo cual es aproximadamente 2 veces más grande que el siguiente archivo de mayor tamaño, adicionalmente el 86.1% de esa gran cantidad de código no está siendo realmente usada para el ejemplo del segundo paso. Esta situación provocaba que el tiempo que se gastaba descargando todos los datos del archivo fuera innecesario ya que gran parte de esto era irrelevante. A continuación, en la **Figura 63**, se observará la misma información del archivo luego de haber realizado la modularización.

Figura 63.

Información del archivo alk_checkout después de la modularización.



URL	Type	Total Bytes	Unused Bytes	Usage Visualization
/acc.alk_checkout.js	JS (per function)	43 439	27 433 63.2%	

Nota. Tomado de Google Chrome, lighthouse (2023).

El tamaño del archivo después de realizar la modularización se redujo considerablemente con un total de 43.439 Bytes, además de la reducción neta de la cantidad de

código se logró reducir en más del 20% la cantidad de código innecesario cargado por el archivo. A continuación, en la **Figura 64**, se resumirá los resultados obtenidos de antes y después de la modularización.

Figura 64.

Comparativa de los datos del archivo antes y después de la modularización.

	Tamaño total (bytes)	**Bytes no usados (Para el paso 2)**	**Porcentaje no usado de bytes**
Antes de modularizar	118.472	102.063	86.1%
Después de modularizar	43.439	27.433	63.2%
Diferencia	75.033	74.630	22.9%
Diferencia Porcentual	63.33%	73.12%	22.9%

Se puede observar que hubo una reducción tanto en el tamaño total del archivo, así como de la cantidad de bytes no usados y el porcentaje del archivo sin utilizar. La reducción significativa del archivo se debe a que gracias a la modularización no se está cargando todo el código de una vez, por otro lado, a medida que sea necesario se van descargando los módulos. De esta forma el tamaño original del archivo de cierta forma se conserva, pero a diferencia de tener todo disponible en un solo archivo el código está distribuido en diferentes módulos.

A continuación, en la **Figura 65**, se puede observar un ejemplo de algunos de los módulos que están siendo cargados en el paso 4 del checkout:

Figura 65.

Información módulos cargados en el paso 4 del checkout.

URL	Type	Total ...	Unused Bytes	Usage Visualization
/payment-required-inputs.js?	JS (per function)	27 194	2 654 9.8%	
/payment-tabs.js?v=7.0.0-30-	JS (per function)	10 025	1 813 18.1%	
/credit-card-validation.js?v=7	JS (per function)	24 457	1 135 4.6%	
/ccv-saved-card.js?v=7.0.0-30	JS (per function)	8 134	845 10.4%	
/debit-as-credit-card.js?v=7.0	JS (per function)	12 098	549 4.5%	

Nota. Tomado de Google Chrome, lighthouse (2023).

La tabla contiene la información de 5 de los principales módulos cargados en el paso 4 del checkout. Algunos de los módulos cargados tienen tamaños más grandes que otros, principalmente debido a la complejidad de algunas funcionalidades que provocaba que el código fuera originalmente bastante extenso. Un ejemplo de esta situación se encuentra en el módulo ***payment-required-inputs***, este módulo corresponde al caso mencionado en la división del código como uno de los casos más problemáticos debido a la estructura que se manejaba.

El caso del archivo ***payment-required-inputs*** es uno de los más claros ejemplos de que cargar el código por demanda es una solución que ayuda mucho en el rendimiento de una plataforma. Este caso es muy importante debido a que las funcionalidades que conforman el módulo solo eran requeridas para la verificación de los inputs en los métodos de pago, esta verificación solo era necesaria hasta llegar al paso 4 del proceso de checkout, pero por otro lado todas estas funcionalidades se estaban cargando desde un comienzo. Las funcionalidades que contiene el módulo suman un total de 27.194 bytes, teniendo en cuenta que el archivo completo tiene una total de 118.472 bytes, aproximadamente a un 23% del código cargado al comienzo correspondía a funcionalidades relacionadas a una validación que solo se realizaría hasta llegar al penúltimo paso del proceso.

Otro aspecto importante es que para los módulos cargados y que se listan en la **Figura 66**, todos tienen un porcentaje de código sin usar bastante bajos. En la **Figura 66**, se presenta de una forma más clara esta información.

Figura 66.

Tabla de información de los módulos cargados para el paso 4 del checkout.

Módulos	***Tamaño en Bytes***	***Bytes no usados***	***Porcentaje sin usar***
payment-required	27.194	2.654	9.8%
payment-tabs	10.025	1.813	18.1%
credit-card-validation	24.257	1.135	4.6%
ccv-saved-card	8.134	845	10.4%
debit-as-credit-card	12.098	549	4.5%

Los módulos solo se cargan cuando se requieren por lo que normalmente estos porcentajes son bastante bajos, por otro lado, los módulos están conformados por varias funciones, debido a esto en el momento que se requiera solo una función del módulo este será cargado totalmente. En general este problema no debería generar mayores complicaciones ya que todos los módulos se construyeron con funcionalidades que se asocian o que requieren unas de otras, debido a esto los porcentajes de código no usado a pesar de no estar cerca de 0% serán relativamente bajos.

Adicionalmente una comparativa importante que se realizó está relacionada con el tiempo que toma el descargar el contenido del archivo principal al navegador. A continuación, en la **Figura 67**, se mostrará la información relacionada con el tiempo de descarga del contenido antes y después del proceso de modularización.

Figura 67.

Información de red sobre el archivo antes y después de la modularización.

Tarea	***Duración en milisegundos***	
	Antes	***Después***
Conexión estancada	2.72	1.11
Roques enviada	0.082	0.084
Espera por respuesta del servidor	2.51	2.65
Descarga del contenido	6.74	3.60
Tiempo total	11.97	7.36

Para este análisis las pruebas se realizaron usando la velocidad de internet local la cual corresponde a una conexión rápida. En cuanto a los resultados se pueden observar que están en milisegundos lo cual indica que los tiempos para descargar un archivo individual son bastante bajos. Por otro lado, se debe recordar que este solo es un archivo de los cientos que se deben cargar, por lo que de llegar a aplicar la modularización a otros archivos el impacto se verá reflejado de una mejor manera.

En cuanto al análisis realizado, se puede observar que el proceso completo para obtener un archivo de funcionalidades está dado por los siguientes pasos:

1. **Conexión estancada:** La conexión puede estar estancada porque existen peticiones de mayor prioridad, porque ya existen 6 conexiones TCP para ese origen, el cual es el límite o el navegador asigna brevemente espacio en la memoria caché del disco. Ninguna de estas situaciones está implicada en el proyecto realizado por lo que los resultados observados antes y después dependen exclusivamente de factores externos.

2. **Request enviada:** La petición ha sido enviada al servicio. La duración de este paso es bastante corta y es muy similar para el antes y el después, pero de igual manera que el paso anterior este depende de factores externos.

3. **Espera por respuesta del servidor:** El navegador se encuentra en espera de recibir el primer byte de respuesta. Este tiempo incluye 1 viaje de ida y vuelta de latencia y el tiempo que tardó el servidor en preparar la respuesta.

4. **Descarga del contenido:** Este valor es la cantidad total de tiempo dedicado a leer el cuerpo de la respuesta. En esta etapa es donde se ve reflejado los resultados del proceso de modularización, acá se logró disminuir el tiempo de 6.74 a 3.60 milisegundos.

Finalmente, entonces la duración total del proceso puede tomar más o menos tiempo dependiendo de factores externos, pero en cuanto al tiempo que toma descargar el contenido solicitado siempre tendrá un valor menor, esto debido a que la cantidad de código es bastante menor.

Puede parecer que una diferencia de 3.14 milisegundos no es la gran cosa, pero se debe tener en cuenta que las velocidades de red pueden variar mucho el tiempo que toma descargar el contenido, en redes lentas este tiempo se puede incrementar significativamente. A continuación, en la **Figura 68**, se realizó nuevamente el análisis, pero esta vez se simularon las velocidades de una red 3G con el objetivo de ver que tanto la velocidad de red afecta estos tiempos de carga.

Figura 68.

Información de red sobre el archivo antes y después de la modularización simulando red de baja velocidad.

Tarea	***Duración en milisegundos***	
	Antes	***Después***
Conexión estancada	0.41	0.16
Request enviada	0.092	0.097
Espera por respuesta del servidor	568.41	575.29
Descarga del contenido	485.12	255.94
Tiempo total	1.05 ***segundos***	0.831 ***segundos***

Se observa que para el caso de una red 3G con una velocidad no muy rápida los tiempos aumentan significativamente, este aumento es especialmente notorio para la espera de la respuesta del servidor y la descarga del contenido. El tiempo total necesario pa ra obtener los archivos es de **1.05 segundos** antes de la modularización y de **0.831 segundos** después, el ahorro que se obtiene es de **0.219 segundos** lo que indica que esta disminución de tiempo de carga es cada vez más significativa para velocidades red más bajas.

Las pruebas realizadas permiten observar una comparación entre los resultados antes y después del proceso de modularización. Estos resultados pueden variar levemente según situaciones específicas dentro de la plataforma y el servidor, pero estos dan una visión general del rendimiento en la sección. Es necesario analizar más a profundidad el proceso de modularización para definir si realmente cómo se llevó a cabo este proceso fue el más adecuado, así como qué aspectos se debieron tener en cuenta para obtener mejores resultados.

6. Conclusiones

Al llevar a cabo este proyecto se encontraron con ciertas situaciones que dificultaron su desarrollo. La primera dificultad se debió principalmente al hecho de que se trabajó con funcionalidades que ya llevaban tiempo desarrolladas y que constantemente eran modificadas para corregir errores o agregar nuevas características. Para entender el funcionamiento correcto de las funciones se requirió del apoyo de otros desarrolladores, por otro lado, era común que estos no tuvieran siempre la disponibilidad. Esto hizo que la tarea de modularizar se volviera complicada, ya que al realizar cambios en el código y definir los módulos, era común que el código simplemente dejara de funcionar correctamente. Como resultado, una de las tareas que requirió más tiempo era asegurar que el código siguiera funcionando correctamente y cumpliera con su propósito.

En relación con los resultados obtenidos en cuanto al ahorro de tiempo, se pudo observar una mejora, aunque no fue realmente muy notable. Dado que se estaba trabajando en un proyecto tan grande, es necesario asignar un equipo completo que se encargue de más secciones simultáneamente y abarquen una mayor cantidad de funcionalidades. A pesar de que el beneficio en tiempos de carga no fue tan significativo, el proceso de modularizarían sí ayudó a organizar el código y lograr una mayor claridad en cuanto a las funciones requeridas para cada paso del proceso de checkout. También permitió identificar la falta de consistencia en el código y arreglar bugs.

La modularizarían realizada contribuyó a mejorar levemente los tiempos de carga del checkout, lo que a su vez se traduce en ahorro de costos de transferencia de datos, ya que el código se dividió de tal manera que solo se cargaba lo necesario. Con base en estos resultados y en las mejoras logradas en el código, en el futuro será posible considerar la incorporación de

otras áreas en un proceso de modularización más ambicioso, que permita obtener mejores resultados en relación con los tiempos de carga.

En cuanto a la experiencia personal durante la práctica empresarial, esta representó una valiosa oportunidad para formar parte de la industria y comprender cómo funcionan los proyectos reales, especialmente en entornos con numerosos desarrolladores y en proyectos tan extensos como el de Alkomprar. Gracias a esta experiencia, fue posible poner a prueba y demostrar los conocimientos en el área a la que fui asignado. Además, en esta experiencia fue posible entender el extenso camino que se presenta en cuanto a la variedad de herramientas, tecnologías y metodologías de trabajo disponibles.

En síntesis, el período en Keyrus resultó ser una experiencia sumamente enriquecedora que, en mi opinión, me proporcionó una visión realista y valiosa del entorno de proyectos empresariales.

Referencias Bibliográficas

Axel Rauschmayer. (2018). *Exploring ES6.* https://exploringjs.com/es6/ch_modules.html#sec_overview-modules

Baluch, A. (2023). *38 E-commerce Statistics of 2023 [38 Estadísticas de comercio electrónico de 2023]*. https://www.forbes.com/advisor/business/ecommerce-statistics/

Baymard Institute. (2023). *222 Top E-Commerce Sites Ranked by User Experience Performance [222 principales sitios de comercio electrónico clasificados por rendimiento de la experiencia del usuario]*. https://baymard.com/ux-benchmark

Cetinformatica. (s.f). *Introducción a JavaScript.* https://cetinformatica.com/chamilo/courses/DEMO/document/learning_path/Demostracion_de_una_leccion/intro_js/intro-js.html?cidReq=DEMO&id_session=0

Chrome Developers. (s.f). *Performance Audits [Auditorías de Desempeño]*. https://developer.chrome.com/docs/lighthouse/performance/

IONOS. (2021). *¿Qué es Google Lighthouse?*.https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de-busqueda/google-lighthouse/

Jackson, B. (2023). *CSS Preprocessors - Sass vs Less*. https://www.keycdn.com/blog/sass-vs-less

Manz. (s.f). *Import dinámico en Javascript.* https://lenguajejs.com/javascript/modulos/dynamic-import/

MDN contributors. (2022). *Javascript*. https://developer.mozilla.org/es/docs/web/javascript

Nimble. (2022). *Scrum de Scrums: Qué es y qué hace falta.* https://www.nimblework.com/es/agile/scrum-de-scrums/

Nobledesktop. (2021). *Git Branches: List, Create, Switch to, Merge, Push, & Delete [Ramas de Git: enumerar, crear, cambiar a, fusionar, enviar y eliminar]. https://www.nobledesktop.com/learn/git/git-branches*

OpenWebinars. (2017). *JSP Standard Tag Library (JSTL) y Expression Language (EL).* https://openwebinars.net/blog/jsp-standard-tag-library-jstl-y-expression-language-el/

Proyectosagiles. (2022). *¿Qué es Scrum?*. https://proyectosagiles.org/que-es-scrum/

Reynoso, L. (2022). *Apunte: Modularización.* https://opendata.fi.uncoma.edu.ar/algoritmos/ApunteModularizaci%C3%B3n.html

Reyes, E. (2022). *Programación modular y estructurada.* https://es.scribd.com/document/378774239/Programacion-Modular-y-Estructurada#

SAP Hybris, SAP (2021). *SAP Company Information*. https://www.sap.com/products/crm/e-commerce-platforms.html

Vega, J. (2018). *Lenguaje de programación estructurada.* http://ri.uaemex.mx/bitstream/handle/20.500.11799/104235/secme-17408_1.pdf?sequence=1