PROYECTO DE PROCESADOR DE TEXTOS WEB EN JAVASCRIPT

Autor: Tomás Cano Nieto

Fecha: 24/02/2025

Nombre del Ciclo: Desarrollo de Aplicaciones Web

Curso escolar: 2º DAW

Unión Europea
Fondo Social Europeo

RESUMEN

Este proyecto tiene como objetivo el desarrollo de un procesador de textos básico basado en tecnologías web, con un enfoque especial en el uso de JavaScript para la implementación de sus funcionalidades principales. La aplicación permitirá a los usuarios escribir, editar, guardar y exportar textos en distintos formatos como .txt o .pdf, además de contar con opciones básicas de formateo como negritas, cursivas y subrayado. El proyecto también podrá ser ampliado con la integración de un framework para mejorar la estructura y funcionalidad de la aplicación.

El desarrollo del procesador de textos se centrará en la implementación de una interfaz sencilla e intuitiva, asegurando que la experiencia del usuario sea óptima. Se considerará el uso de tecnologías como localStorage para la persistencia de los documentos creados y bibliotecas externas como jsPDF para la generación de archivos PDF. Además, se evaluará el correcto funcionamiento del software, la calidad del código y la presentación final del proyecto.

Se espera que este trabajo no solo brinde una herramienta funcional, sino que también sirva como ejercicio práctico para aplicar los conocimientos adquiridos en el módulo de Desarrollo Web en Entorno Cliente, abordando conceptos clave de manipulación del DOM, eventos, almacenamiento local y exportación de datos.

ABSTRACT

This project aims to develop a basic text processor based on web technologies, with a special focus on using JavaScript to implement its core functionalities. The application will allow users to write, edit, save, and export text in different formats such as .txt or .pdf, in addition to offering basic formatting options such as bold, italics, and underline. The project may also be expanded with the integration of a framework to improve its structure and functionality.

The development of the text processor will focus on creating a simple and intuitive user interface, ensuring an optimal user experience. Technologies such as localStorage will be considered for document persistence, and external libraries such as jsPDF will be used for generating PDF files. Additionally, the correct functionality of the software, code quality, and final presentation of the project will be evaluated.

This work is expected not only to provide a functional tool but also to serve as a practical exercise for applying the knowledge acquired in the Web Development in Client-Side Environment module, covering key concepts such as DOM manipulation, events, local storage, and data export.

ÍNDICE

INTRODUCCIÓN

En el mundo digital actual, la edición y procesamiento de textos es una actividad fundamental tanto en el ámbito académico como profesional. Las aplicaciones de procesamiento de textos han evolucionado considerablemente en las últimas décadas, permitiendo a los usuarios crear, modificar y compartir documentos de manera eficiente. Sin embargo, muchas de estas herramientas dependen de software propietario o requieren instalaciones complejas. Este proyecto busca desarrollar un procesador de textos basado en tecnologías web, accesible desde cualquier navegador y que no requiera instalaciones adicionales, aprovechando el potencial de JavaScript y otras tecnologías web.

El desarrollo de una aplicación web que permita a los usuarios escribir, editar, guardar y exportar documentos representa un desafío técnico interesante. No solo se trata de implementar una interfaz intuitiva, sino también de garantizar que el procesamiento de los textos sea eficiente, funcional y extensible. La posibilidad de exportar documentos en formatos como .txt y .pdf permite a los usuarios utilizar el procesador en diversos contextos, desde la redacción de apuntes personales hasta la generación de documentos formales.

El uso de tecnologías modernas, como APIs de JavaScript y bibliotecas especializadas, juega un papel clave en este proyecto. El objetivo es demostrar la capacidad de JavaScript para construir aplicaciones web interactivas y funcionales sin necesidad de servidores complejos o software adicional. Además, este proyecto servirá como una oportunidad para explorar buenas prácticas en el desarrollo web, como la organización modular del código, la optimización de rendimiento y el diseño centrado en la experiencia del usuario.

JUSTIFICACIÓN

El motivo principal para desarrollar un procesador de textos web radica en la creciente necesidad de herramientas accesibles y multiplataforma para la edición de documentos. En la actualidad, la mayoría de los procesadores de textos populares requieren suscripción, instalaciones o están limitados a ecosistemas específicos. Este proyecto busca ofrecer una solución libre de estos inconvenientes, permitiendo que cualquier usuario con un navegador moderno pueda acceder a una herramienta funcional sin barreras de entrada.

Desde el punto de vista educativo, este proyecto es una excelente oportunidad para poner en práctica conocimientos clave sobre desarrollo web en entorno cliente. La implementación de funcionalidades como la edición de texto, el almacenamiento en localStorage y la exportación en diferentes formatos permitirá reforzar habilidades en JavaScript, manipulación del DOM y uso de APIs web. Además, trabajar con bibliotecas externas como jsPDF proporcionará una experiencia real sobre cómo integrar herramientas de terceros en un proyecto propio.

Otra razón importante para la realización de este proyecto es la aplicabilidad en escenarios reales. Profesionales, estudiantes y usuarios en general podrían beneficiarse de un procesador de textos ligero y accesible, sin necesidad de depender de software instalado o servicios en la nube con limitaciones. Además, el aprendizaje obtenido podría servir como base para futuras mejoras y expansiones, como la integración de correctores ortográficos, opciones avanzadas de formato y sincronización en la nube.

Por último, este proyecto también representa una oportunidad para demostrar la viabilidad de las aplicaciones web modernas como una alternativa real a las aplicaciones tradicionales. Con un diseño bien estructurado y un uso eficiente de los recursos del navegador, es posible crear herramientas funcionales que ofrezcan una experiencia de usuario comparable a las soluciones de escritorio, consolidando así el papel de JavaScript como un lenguaje fundamental en el desarrollo de software.

OBJETIVOS

El objetivo principal de este proyecto es desarrollar un procesador de textos web utilizando JavaScript, que permita a los usuarios escribir, editar, guardar y exportar documentos de manera sencilla. Se busca que la aplicación tenga una interfaz intuitiva y funcionalidades esenciales para el manejo básico de texto, asegurando una experiencia fluida y eficiente. Para cumplir con este propósito, se establecen los siguientes objetivos específicos:

Implementar una edición básica de texto

-Permitir al usuario escribir y modificar texto en un área de edición.
-Incluir opciones de formateo como negrita, cursiva y subrayado, accesibles mediante botones o atajos de teclado.

Gestionar documentos a nivel local

-Implementar un sistema de guardado en el navegador mediante localStorage, permitiendo que los documentos se almacenen con un título único y puedan recuperarse posteriormente.
-Crear una interfaz que facilite la visualización y gestión de los documentos guardados.

Desarrollar opciones de exportación

-Permitir la exportación del texto a archivos .txt, utilizando la API Blob de JavaScript.
-Implementar la exportación a formato .pdf mediante bibliotecas especializadas como jsPDF, asegurando un correcto formato del contenido.

Diseñar una interfaz clara y funcional

-Crear un diseño limpio y ordenado que facilite el uso del procesador.
-Asegurar que los botones y herramientas sean accesibles y comprensibles para cualquier usuario.
-Garantizar una experiencia de usuario fluida, evitando retrasos o bloqueos en la edición del texto.

Optimizar el código y su estructura

-Aplicar buenas prácticas de programación en JavaScript para mantener el código limpio, estructurado y fácil de entender.
-Dividir las funcionalidades en módulos o funciones reutilizables, facilitando futuras mejoras y mantenimiento.

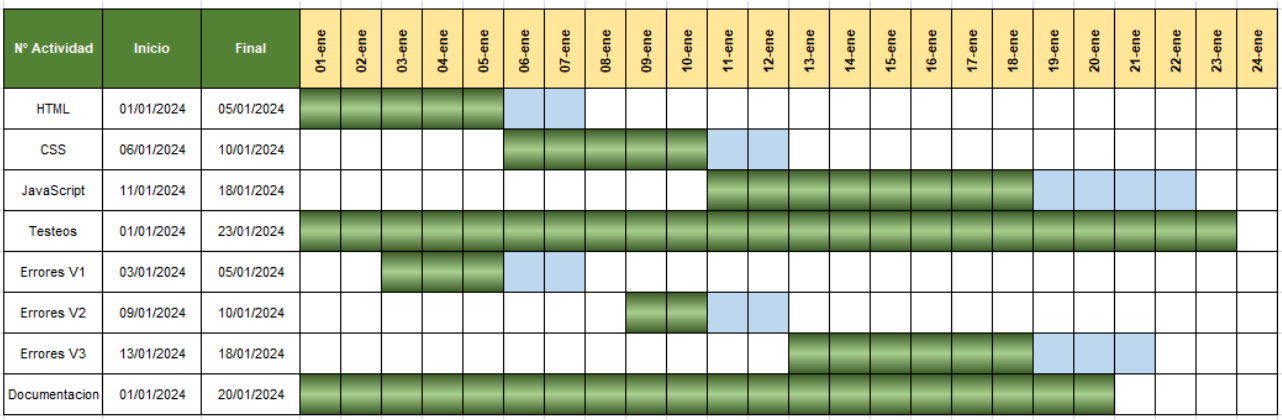
Documentar correctamente el proyecto

-Redactar una documentación detallada sobre el desarrollo del procesador de textos, explicando las decisiones técnicas y funcionalidades implementadas.
-Incluir comentarios en el código fuente para que sea comprensible y fácil de modificar en futuras versiones.

Evaluar posibles mejoras futuras

-Analizar qué características adicionales podrían implementarse en versiones posteriores, como corrector ortográfico, opciones avanzadas de formato o sincronización en la nube.

PLANIFICACIÓN DEL PROYECTO



N° Actividad	Total de Horas	Comentario
HTML	40 horas	Aproximado, aunque pensaba que me llevaría más tiempo.
CSS	36 horas	Me tomó menos tiempo del esperado.
JavaScript	52 horas	Fue una de las partes más largas, pero dentro de lo previsto.
Testeos	96 horas	Al final dediqué más horas de lo planeado aquí.
Errores V1	20 horas	Pensaba que sería más rápido, pero me llevó su tiempo.
Errores V2	12 horas	Lo solucioné antes de lo esperado.
Errores V3	40 horas	Se alargó más de lo que pensé en un principio.
Doc.	64 horas	Creía que me llevaría menos tiempo, pero fue necesario dedicarle más.

Las horas no son exactas, pero reflejan más o menos el tiempo que realmente dediqué a cada parte. También hubo ajustes sobre la marcha, ya que algunas tareas tomaron más tiempo y otras menos de lo que había previsto.

En el diagrama de Gantt, se pueden ver casillas **azules** después de la finalización de algunas tareas. Estas representan las horas que inicialmente pensé que necesitaría, pero que finalmente no fueron necesarias. Esto demuestra que en algunas partes del proyecto fui más eficiente de lo esperado, mientras que en otras tuve que dedicar más tiempo del planeado.

DESARROLLO DEL PROYECTO

En esta sección se describe el proceso de desarrollo del procesador de textos web, abordando cada una de sus funcionalidades y la forma en que han sido implementadas. Se sigue una estructura lógica desde la edición básica hasta la exportación de documentos, asegurando una experiencia de usuario fluida y efectiva.

1. EDICIÓN BÁSICA DE TEXTO

El núcleo del procesador de textos es la capacidad de escribir y editar contenido en un área de texto. Para ello, se ha implementado un editor basado en un elemento ** **, el cual permite que los usuarios escriban directamente sobre la interfaz sin necesidad de utilizar un **<textarea**>.

Funcionalidades implementadas:

- **Negrita, cursiva y subrayado:** Se han añadido botones que permiten aplicar estos estilos al texto seleccionado utilizando **document.execCommand().**
- **Accesos rápidos con teclado:** Además de los botones, se han implementado combinaciones de teclas (Ctrl + B para negrita, Ctrl + I para cursiva y Ctrl + U para subrayado), mejorando la accesibilidad.
- **Interfaz de usuario clara:** Se han diseñado botones visibles y organizados para facilitar la edición del texto sin distracciones.

2. GUARDADO EN LOCALSTORAGE

Para que el usuario pueda almacenar sus documentos sin perderlos al cerrar la página, se ha utilizado la API localStorage, que permite guardar información de forma persistente en el navegador.

Proceso de implementación:

1. **Sistema de guardado:** Al presionar el botón de "Guardar", el contenido del editor se almacena en localStorage bajo un nombre de documento definido por el usuario.
2. **Lista de documentos:** Se ha creado un apartado donde se muestran los documentos guardados previamente, permitiendo que el usuario los cargue y edite nuevamente.
3. **Eliminación de documentos:** Se ha agregado una opción para eliminar documentos almacenados si el usuario ya no los necesita.

Este sistema permite que el procesador de textos actúe como un editor funcional sin necesidad de depender de una base de datos o almacenamiento en la nube.

3. EXPORTACIÓN DE DOCUMENTOS

El procesador de textos ofrece la opción de exportar los documentos en dos formatos: .txt y .pdf, permitiendo al usuario guardar su trabajo en su dispositivo.

Exportación a .txt

Para exportar el contenido en formato de texto plano (.txt), se ha utilizado la API Blob. Se crea un archivo con el contenido del editor y se genera un enlace de descarga automática.

Exportación a .pdf

Para generar archivos PDF, se ha integrado la biblioteca jsPDF. La función toma el contenido del editor, lo convierte en un documento PDF y permite su descarga en un solo clic.

4. DISEÑO VISUAL Y USABILIDAD

La interfaz ha sido diseñada para ser sencilla y fácil de entender. Se han considerado los siguientes aspectos:

- **Estructura minimalista:** Se evita el exceso de botones y opciones innecesarias.
- **Distribución intuitiva:** Los botones de edición y exportación están organizados de manera clara.
- **Compatibilidad con distintos dispositivos:** El diseño es adaptable a pantallas de diferentes tamaños, permitiendo su uso en ordenadores y dispositivos móviles.

CONCLUSIÓN

Este desarrollo ha permitido la implementación de un procesador de textos funcional con opciones básicas de edición, guardado y exportación. A lo largo del proceso, se ha priorizado la facilidad de uso y la accesibilidad para que cualquier usuario pueda interactuar con la aplicación de forma sencilla y eficiente.

PRESUPUESTO

En este apartado se realiza una estimación detallada de los recursos necesarios para el desarrollo del procesador de textos web, teniendo en cuenta el tiempo invertido, las herramientas utilizadas y otros factores relevantes. Como se trata de un proyecto educativo, se ha optado por utilizar herramientas y tecnologías gratuitas.

1. TIEMPO INVERTIDO

El tiempo es uno de los recursos más importantes en cualquier proyecto de desarrollo de software. Se ha estimado el número de horas necesarias para cada fase del proyecto, basándose en la planificación y ejecución real del trabajo:

Fase del Proyecto	Horas Estimadas
Análisis de requisitos	6 horas
Diseño de la interfaz	10 horas
Implementación:	
- Edición básica	12 horas
- Guardado en localStorage	8 horas
- Exportación a .txt/.pdf	10 horas
Testeo y depuración	8 horas
Documentación	6 horas
Total Aproximado	**60 horas**

El tiempo estimado total es de aproximadamente **60 horas de trabajo**, aunque puede variar dependiendo de la complejidad de los ajustes y mejoras que se implementen sobre la marcha.

2. HERRAMIENTAS UTILIZADAS

Dado que este proyecto se ha desarrollado como parte de un entorno educativo, se han seleccionado herramientas gratuitas para minimizar los costos. A continuación, se detallan los recursos utilizados:

a) Lenguajes y tecnologías

- **HTML5** → Estructura del procesador de textos.
- **CSS3** → Diseño y presentación de la interfaz.
- **JavaScript** → Lógica de funcionamiento e interactividad.

b) Bibliotecas y APIs

- **localStorage API** → Para el almacenamiento de documentos en el navegador.
- **Blob API** → Para la exportación de archivos .txt.
- **jsPDF** → Para la exportación de archivos .pdf.

c) Entorno de desarrollo y herramientas auxiliares

Herramienta	Uso en el proyecto	Costo
Visual Studio Code	Edición de código y depuración	Gratis
Google Chrome DevTools	Inspección y pruebas en el navegador	Gratis
GitHub	Control de versiones y respaldo del código	Gratis
Figma / Adobe XD	Diseño de interfaz (opcional)	Gratis

Dado que todas estas herramientas son de acceso libre, no ha sido necesario incurrir en gastos adicionales en software de desarrollo.

3. COSTE ESTIMADO DEL TRABAJO (EQUIVALENCIA SALARIAL)

Para calcular un presupuesto más realista, se puede considerar cuánto costaría este desarrollo si fuera llevado a cabo por un desarrollador profesional. Suponiendo un salario promedio de **20€ por hora** para un programador junior, el costo aproximado sería:

60 horas × 20€/hora = 1.200€

Sin embargo, al tratarse de un proyecto educativo realizado por el propio estudiante, este coste no se aplica, pero sirve como referencia del valor del trabajo.

4. RECURSOS MATERIALES Y HARDWARE

El desarrollo del proyecto ha requerido el uso de equipos informáticos con las siguientes características:

Recurso	Descripción	Costo Estimado
Ordenador personal	Mínimo 8GB RAM, procesador i5 o equivalente	Ya disponible
Conexión a Internet	Para pruebas y uso de documentación online	Ya disponible
Almacenamiento en la nube	Uso de GitHub para control de versiones	Gratis

5. COSTES ADICIONALES (OPCIONAL)

Si este proyecto se escalara a un entorno profesional, se podrían considerar otros costos adicionales, como:

- **Dominio web y hosting** (opcional, si se despliega en internet).
- **Compatibilidad con dispositivos móviles** (requeriría más tiempo de desarrollo).
- **Seguridad y protección de datos** (en caso de almacenamiento en servidores).

CONCLUSIÓN

El desarrollo del procesador de textos se ha llevado a cabo con herramientas gratuitas, reduciendo así los costos del proyecto. El mayor recurso invertido ha sido el tiempo, con un estimado de **60 horas de trabajo**, lo que equivaldría a **1.200€ en el mercado laboral**.

Este presupuesto permite valorar el esfuerzo requerido para crear una aplicación funcional y útil, así como la importancia de una buena planificación para optimizar los recursos disponibles.

PROBLEMAS ENCONTRADOS Y SOLUCIONES

Durante el desarrollo del procesador de textos web, se han presentado varios desafíos tanto a nivel técnico como de diseño. A continuación, se detallan los principales problemas encontrados junto con las soluciones implementadas para superarlos.

1. PROBLEMAS EN LA EDICIÓN Y FORMATEO DEL TEXTO

Problema

Inicialmente, al aplicar formatos como negrita, cursiva o subrayado, se detectó que el estilo no se aplicaba correctamente al texto seleccionado. En algunos casos, el formato se activaba para todo el contenido en lugar de solo la parte resaltada.

Solución

Se implementó la función **document.execCommand()**, que permite modificar solo el texto seleccionado por el usuario. Sin embargo, dado que esta API está en desuso, se exploraron alternativas más modernas utilizando **contentEditable** y manipulaciones del DOM con **Range** y **Selection** para garantizar que los estilos se apliquen correctamente.

2. GUARDADO Y RECUPERACIÓN DE DOCUMENTOS EN LOCALSTORAGE

Problema

Uno de los principales problemas al usar **localStorage** fue que los documentos guardados se sobrescribían si tenían el mismo nombre, lo que llevaba a la pérdida de datos.

Solución

Se implementó un mecanismo que obliga al usuario a asignar un título único a cada documento. Además, se usó un sistema de claves dinámicas en localStorage, permitiendo que los documentos se almacenen con identificadores únicos en formato JSON.

3. EXPORTACIÓN A FORMATOS .TXT Y .PDF

Problema

Al exportar los textos en .txt usando la API Blob, algunos caracteres especiales como acentos y eñes no se guardaban correctamente debido a problemas de codificación. En la exportación a .pdf, los saltos de línea y el formato del texto no se mantenían correctamente.

Solución

- Para .txt, se especificó la codificación UTF-8 al crear el Blob:
- const blob = new Blob([texto], { type: "text/plain;charset=utf-8" });
- Para .pdf, se usó la biblioteca jsPDF, ajustando manualmente los saltos de línea mediante la función splitTextToSize(), asegurando que el texto no se desbordara fuera de los márgenes.

4. DISEÑO DE LA INTERFAZ DE USUARIO

Problema

Inicialmente, la interfaz no era lo suficientemente intuitiva, lo que dificultaba la interacción del usuario. Los botones de formato no eran visibles de manera clara y no había retroalimentación visual sobre qué opciones estaban activas.

Solución

Se implementaron mejoras en la interfaz, como:

- Resaltar los botones cuando un formato estaba activado.
- Organizar los controles en una barra de herramientas clara y accesible.
- Uso de íconos en lugar de texto para hacer la interfaz más visualmente atractiva.

5. COMPATIBILIDAD CON NAVEGADORES

Problema

Al probar la aplicación en diferentes navegadores, se detectó que algunas funcionalidades no se comportaban de la misma manera en Chrome, Firefox y Edge. En especial, el manejo de **execCommand**() tenía variaciones.

Solución

Se realizaron pruebas en múltiples navegadores y se adoptó un enfoque más estándar con **contentEditable**, asegurando que el código funcionara en la mayoría de los entornos sin depender de API en desuso.

6. PERDIDA DE DATOS EN LA SESIÓN DEL USUARIO

Problema

Si el usuario cerraba accidentalmente la pestaña o recargaba la página sin guardar, todo el contenido escrito se perdía.

Solución

Se implementó un **autoguardado en localStorage**, que guarda automáticamente el contenido cada poco segundo. Así, al recargar la página, el usuario puede recuperar su trabajo sin necesidad de guardarlo manualmente.

CONCLUSIÓN

A lo largo del desarrollo, se identificaron y resolvieron varios problemas que afectaban la funcionalidad y usabilidad del procesador de textos. La implementación de soluciones basadas en estándares modernos permitió mejorar la experiencia del usuario, garantizar la compatibilidad y optimizar la fiabilidad del sistema. Estas mejoras contribuyeron a la estabilidad y facilidad de uso del proyecto, permitiendo ofrecer un procesador de textos funcional y eficiente.

CONCLUSIONES

El desarrollo de este procesador de textos web ha sido un proceso enriquecedor que ha permitido aplicar y reforzar conocimientos clave en JavaScript, manipulación del DOM, almacenamiento local y exportación de datos. A lo largo del proyecto, se han enfrentado diversos desafíos que han sido solucionados con enfoques prácticos y el uso de tecnologías adecuadas.

¿Se cumplieron los objetivos?

Desde el inicio, el proyecto tenía como meta desarrollar una herramienta básica de edición de textos que permitiera escribir, dar formato, guardar y exportar documentos. Se han cumplido estos objetivos, logrando:

- Un editor funcional con opciones de negrita, cursiva y subrayado.
- Un sistema de almacenamiento que permite guardar y recuperar documentos en **localStorage**.
- La exportación de archivos en formatos .txt y .pdf con codificación correcta.
- Una interfaz intuitiva y accesible para el usuario.

Además, se han implementado mejoras que no estaban planteadas inicialmente, como el autoguardado en localStorage, lo que hace que la experiencia del usuario sea más fluida y segura.

Conocimientos adquiridos y reforzados

El desarrollo de este proyecto ha permitido afianzar diversos conceptos y habilidades, entre ellos:

-Manipulación avanzada del DOM: Uso de contentEditable, execCommand(), Range y Selection.

-Gestión de almacenamiento local: Implementación de localStorage para la persistencia de datos en el navegador.

-Manejo de eventos y control de interfaz: Creación de botones interactivos, cambios dinámicos en la UI y retroalimentación visual para el usuario.

-Exportación de datos: Uso de Blob para generar archivos .txt y jsPDF para .pdf, adaptando la estructura del texto en los archivos exportados.

-Compatibilidad entre navegadores: Adaptación del código para garantizar el funcionamiento correcto en diferentes entornos.

Dificultades y aprendizajes

Si bien el desarrollo del proyecto se llevó a cabo con éxito, algunos retos como la inconsistencia en la aplicación de formatos o la pérdida de datos en sesiones inesperadas obligaron a investigar soluciones más eficientes. Estos desafíos han ayudado a mejorar la capacidad de resolución de problemas y a desarrollar un código más robusto y adaptable.

Valoración personal del proyecto

El procesador de textos web desarrollado cumple con los requisitos establecidos y demuestra la capacidad de crear una aplicación funcional utilizando solo tecnologías del lado cliente. Este proyecto no solo refuerza habilidades técnicas, sino que también proporciona una base para futuros desarrollos más complejos, como la implementación de una versión con almacenamiento en la nube o una integración con bases de datos.

En conclusión, el desarrollo de este procesador de textos ha sido un ejercicio práctico altamente beneficioso, que ha permitido mejorar tanto la lógica de programación como la planificación y estructuración de un proyecto web completo.

FUTURAS MEJORAS

En el futuro, el **Procesador de Textos Web** se verá enriquecido con nuevas características que mejorarán tanto la experiencia del usuario como el rendimiento general del sistema. Estas mejoras no solo buscarán hacer la herramienta más completa, sino también asegurar que sea accesible y fácil de usar en distintos dispositivos y contextos.

1. Barra de Herramientas Personalizable

Una de las mejoras clave será permitir que los usuarios personalicen la barra de herramientas del editor. Esto les dará la posibilidad de adaptar el espacio de trabajo a sus necesidades, añadiendo o eliminando botones de las herramientas que más usen. Además de las opciones básicas de formato, se podrán incluir:

- **Ajustes de alineación de texto**.
- **Cambio de tamaño de fuente**.
- **Insertar enlaces, imágenes, y tablas**, entre otras opciones.

2. Exportación a Nuevos Formatos

El sistema de exportación se ampliará para soportar más formatos, facilitando a los usuarios guardar su trabajo en el formato que prefieran:

- **.docx** para que puedan continuar editando sus documentos en Microsoft Word o Google Docs.
- **.html**, ideal para aquellos que necesiten utilizar el contenido en páginas web o blogs.

3. Integración con Google Drive y Otras Nubes

La posibilidad de guardar documentos en la nube será una de las mejoras más útiles. A través de la integración con **Google Drive** y otros servicios de almacenamiento como **Dropbox**, los usuarios podrán acceder a sus documentos desde cualquier dispositivo y tener una copia de seguridad en línea, además de poder compartir sus archivos de forma rápida.

4. Reconocimiento de Voz para Dictado

Con el objetivo de mejorar la accesibilidad y la comodidad, se incorporará un sistema de **reconocimiento de voz** que permitirá a los usuarios dictar su texto en lugar de escribirlo manualmente. Esto no solo beneficiará a personas con discapacidades, sino también a aquellos que prefieren dictar en vez de escribir. Además, se podrán usar comandos de voz para dar formato al texto, como ponerlo en negrita o cursiva.

5. Colaboración en Tiempo Real

La posibilidad de trabajar con otras personas en el mismo documento será una mejora esencial. Con el uso de tecnologías de tiempo real como **WebSockets**, los usuarios podrán editar un documento de manera simultánea, ver los cambios al instante y dejar comentarios dentro del texto. También se implementará un sistema de **control de versiones** para poder revertir cualquier cambio si es necesario.

6. Búsqueda y Reemplazo Avanzado

Aunque el sistema actual de búsqueda y reemplazo es sencillo, se añadirán funcionalidades avanzadas que permitirán realizar búsquedas más específicas, como:

- **Búsqueda con expresiones regulares**, ideal para encontrar patrones de texto complejos.
- **Reemplazo de texto con formato**, lo que permitirá, por ejemplo, cambiar solo las palabras en negrita o en un color determinado.

7. Optimización para Dispositivos Móviles

Aunque ya funciona en teléfonos y tablets, se mejorará la experiencia en dispositivos móviles, asegurando que el editor sea **más fluido y fácil de usar**. Se implementarán ajustes de diseño para que la interfaz se adapte mejor a pantallas más pequeñas y se mejorará la interactividad táctil. Además, se optimizará el tiempo de carga para asegurar una experiencia rápida incluso en conexiones lentas.

8. Rendimiento Mejorado para Documentos Grandes

Se trabajará en mejorar el rendimiento al editar documentos muy largos o complejos. Se implementarán técnicas de optimización para la manipulación del **DOM** y se utilizarán **Web Workers** para ejecutar tareas pesadas en hilos separados, asegurando que la interfaz de usuario no se vuelva lenta o bloqueada.

9. Mayor Control sobre Estilos y Personalización

Se añadirá más flexibilidad en cuanto a la personalización de los documentos. Los usuarios podrán elegir entre más **tipos de fuentes** y **colores**, y se incluirá la opción de crear **temas personalizados** para darle a cada usuario un control total sobre la apariencia de su espacio de trabajo.

10. Accesibilidad Mejorada

Para garantizar que el editor sea accesible para todas las personas, se añadirán funcionalidades como:

- Compatibilidad con **lectores de pantalla**.
- Opciones para **ajustar el contraste** y **el tamaño de la fuente**, lo que será útil para personas con discapacidades visuales.
- **Atajos de teclado** personalizables para facilitar la navegación y la edición.

11. Seguridad y Protección de Documentos

La seguridad será una prioridad con la implementación de herramientas para proteger los documentos, como:

- **Encriptación** de archivos para mantener la privacidad.
- **Autenticación de usuario** para que solo las personas autorizadas puedan acceder a los documentos.
- Un sistema de **recuperación automática** de documentos para evitar la pérdida de trabajo en caso de caídas del sistema o cierres inesperados del navegador.

12. Integración con Servicios Externos

Finalmente, se implementará la integración con **otras plataformas y herramientas** que facilitarán aún más la productividad. Algunas ideas incluyen:

- **Integración con aplicaciones de gestión de proyectos** como Trello o Asana, lo que permitirá importar y exportar contenido fácilmente.
- **Publicación directa en plataformas de contenido** como WordPress o Medium, para aquellos que utilicen el procesador de textos para escribir entradas de blog o artículos.

13. Uso de Frameworks como Vue.js o Firebase

Para mejorar la escalabilidad y la estructura del código, se evaluará la posibilidad de usar frameworks modernos como **Vue.js** para el desarrollo del frontend. Vue.js facilitará la creación de interfaces de usuario dinámicas y más eficientes. Además, se explorará la integración con **Firebase** para la gestión de usuarios, bases de datos en tiempo real, y almacenamiento en la nube, lo que permitirá mejorar la colaboración en tiempo real y el acceso a los documentos desde diferentes dispositivos. Esta integración también facilitará la implementación de autenticación segura y otras funcionalidades relacionadas con la nube.

Estas mejoras tienen como objetivo hacer que el **Procesador de Textos Web** sea más potente, accesible y colaborativo, asegurando que se convierta en una herramienta indispensable tanto para usuarios individuales como para equipos de trabajo.

FUENTES

Bibliografía

- **MDN Web Docs**. HTML, CSS, y JavaScript. Mozilla Developer Network.
Recuperado de: https://developer.mozilla.org/es/

- **W3C**. World Wide Web Consortium. Documentación oficial sobre estándares web.
Recuperado de: https://www.w3.org/

- **JavaScript.info**. *JavaScript Guide*.
Recuperado de: https://javascript.info/

- **OWASP**. Open Web Application Security Project. Mejores prácticas de seguridad para aplicaciones web.
Recuperado de: https://owasp.org/

Webgrafía

- **Google Drive API**. Documentación oficial de la API de Google Drive.
Recuperado de: https://developers.google.com/drive

- **Dropbox API**. Documentación oficial de la API de Dropbox.
Recuperado de: https://www.dropbox.com/developers

- **ChatGPT**. Asistencia con la generación de estilos CSS y dudas sobre JavaScript. Utilizado para la generación de estilos CSS y resolución de problemas en la manipulación de PDFs mediante JavaScript.

- **Foros Especializados**. Foros de discusión sobre la generación y manipulación de PDFs en JavaScript. Se recurrió a foros como **StackOverflow** para resolver dudas específicas sobre la implementación de funcionalidades de PDF que no estaban suficientemente detalladas en la documentación online.

ANEXOS

A continuación, se detallan las principales funcionalidades implementadas en el "Procesador de Textos Web" y su descripción técnica, cómo se implementan en el código y su interacción con el usuario.

1. Formato de Texto (Negrita, Cursiva, Subrayado)

El editor permite aplicar formatos básicos de texto a través de botones de la barra de herramientas. Al hacer clic en un botón, se utiliza el método document.execCommand() para aplicar el formato al texto seleccionado en el editor.

```
function formatText(command) {
    document.execCommand(command, false, null);
}
```

2. Guardar Documento

Permite guardar el contenido del editor en el almacenamiento local del navegador utilizando localStorage. Si ya hay un documento abierto, se sobrescribe; si es nuevo, se solicita al usuario un nombre para el archivo.

```
function saveDocument() {
    const content = document.getElementById('editor').innerHTML;
    const documents = JSON.parse(localStorage.getItem('documents')) || {};

    if (currentEditingDocument) {
        documents[currentEditingDocument] = content;
        alert(`Cambios guardados en "${currentEditingDocument}".`);
    } else {
        const title = prompt("Introduce el título del documento:");
        if (!title) return alert("El título es obligatorio.");
        if (documents[title]) {
            return alert("Ya existe un documento con este título. Usa otro nombre o edita el existente.");
        }

        documents[title] = content;
        alert("Documento guardado con éxito.");
    }

    localStorage.setItem('documents', JSON.stringify(documents));
    currentEditingDocument = null;
    loadDocuments();
}
```

3. Deshacer y Rehacer Cambios

La funcionalidad de deshacer y rehacer se implementa manteniendo un historial de los cambios en el contenido del editor. Cada vez que el usuario edita el texto, se guarda el estado en un array. Los botones "Deshacer" y "Rehacer" permiten navegar por este historial.

```
// Funciones para deshacer y rehacer
function deshacer() {
    if (indiceHistorial > 0) {
        indiceHistorial--;
        editor.innerHTML = historial[indiceHistorial];
        actualizarEstadisticas();
    }
}

function rehacer() {
    if (indiceHistorial < historial.length - 1) {
        indiceHistorial++;
        editor.innerHTML = historial[indiceHistorial];
        actualizarEstadisticas();
    }
}

// Guardar el estado de la edición
function guardarEstado() {
    historial = historial.slice(0, indiceHistorial + 1);
    historial.push(editor.innerHTML); // Guardar el contenido HTML
    indiceHistorial++; // Avanzamos en el historial
}
```

4. Estadísticas de Edición (Palabras y Caracteres)

Se actualizan en tiempo real el número de palabras y caracteres en el editor. El cálculo se realiza cada vez que el usuario escribe, utilizando innerText para contar solo el texto visible, excluyendo los elementos HTML.

```
function actualizarEstadisticas() {
    const texto = editor.innerText.trim();
    const palabras = texto.split(/\s+/).filter((palabra) => palabra.length > 0);
    const caracteres = texto.replace(/\s/g, '').length;

    palabrasSpan.textContent = palabras.length;
    caracteresSpan.textContent = caracteres;
}
```

5. Personalización de Fondo del Editor

El usuario puede cambiar el color de fondo del área de edición mediante un cuadro de diálogo que solicita un código hexadecimal de color.

```
function cambiarFondoEditor() {
    const color = prompt("Introduce el color de fondo (código hexadecimal):");
    if (color) {
        editor.style.backgroundColor = color;
    }
}
```

6. Exportación a Formatos TXT y PDF

Permite al usuario exportar el contenido del editor en formato de texto plano (.txt) y en formato PDF. En el caso del PDF, se utiliza la librería jsPDF para generar el archivo.

```
function exportToTxt() {
    const content = document.getElementById('editor').innerText;
    const blob = new Blob([content], { type: 'text/plain' });
    const a = document.createElement('a');
    a.href = URL.createObjectURL(blob);
    a.download = 'documento.txt';
    a.click();
}
```

```
function exportToPdf() {
    const { jsPDF } = window.jspdf;
    const doc = new jsPDF();

    // Configuración de la fuente
    doc.setFont('Helvetica');

    // Título o encabezado del documento
    doc.setFontSize(18);
    doc.text('Documento Personalizado', 10, 10);

    // Espacio después del título
    doc.setFontSize(12);
    doc.text('Introducción:', 10, 20);
    doc.setFontSize(10);
    doc.text('Este es un documento generado desde el editor. A continuación, se muestra el contenido editado que puedes personalizar a tu gusto.', 10, 25, { maxWidth: 180 });

    // Separador de sección
    doc.setLineWidth(0.5);
    doc.line(10, 30, 200, 30); // Línea horizontal como separador

    // Espacio para el contenido principal
    doc.setFontSize(12);
    doc.text('Contenido del Documento:', 10, 40);

    // Obtener el contenido del editor
    const content = document.getElementById('editor').innerText;
    const lineHeight = 7;
    let yOffset = 45; // Posición Y inicial

    // Añadir el contenido del editor al PDF, con saltos de línea
    content.split('\n').forEach((line, index) => {
        if (yOffset >= 270) { // Si llegamos al final de la página, agregamos una nueva página
            doc.addPage();
            yOffset = 10; // Reiniciar la posición Y
        }
        doc.text(line, 10, yOffset);
        yOffset += lineHeight;
    });

    // Agregar pie de página
    doc.setFontSize(8);
    doc.text('Generado por tu Editor - Página ' + doc.internal.getNumberOfPages(), 10, doc.internal.pageSize.height - 10);

    // Guardar el PDF con un nombre personalizado
    doc.save('documento_personalizado.pdf');
}
```

7. Gestión de Documentos Guardados

Los documentos guardados en el almacenamiento local se muestran en una lista. El usuario puede buscar, editar y borrar documentos. Se utiliza el localStorage para guardar los documentos, y se cargan al iniciar la página.

```
function loadDocuments() {
    const documents = JSON.parse(localStorage.getItem('documents')) || {};
    const documentList = document.getElementById('document-list');

    documentList.innerHTML = '';

    Object.keys(documents).forEach(title => {
        const li = document.createElement('li');
        li.innerHTML = `
            <div>
                <span id="title-${title}">${title}</span>
                <div class="edit-title" style="display: none;">
                    <input type="text" id="edit-input-${title}" value="${title}" />
                    <button onclick="saveEditedTitle('${title}')">Guardar</button>
                </div>
            </div>
            <div>
                <button onclick="editDocument('${title}')">Editar</button>
                <button onclick="deleteDocument('${title}')">Borrar</button>
                <button onclick="toggleEditTitle('${title}')"><i class="fas fa-edit"></i></button>
            </div>
        `;
        documentList.appendChild(li);
    });
}

function editDocument(title) {
    const documents = JSON.parse(localStorage.getItem('documents')) || {};
    const content = documents[title];

    document.getElementById('editor').innerHTML = content;
    currentEditingDocument = title;
    alert(`Editando el documento: "${title}"`);
}

function deleteDocument(title) {
    const documents = JSON.parse(localStorage.getItem('documents')) || {};

    if (confirm(`¿Estás seguro de que deseas borrar el documento: "${title}"?`)) {
        delete documents[title];
        localStorage.setItem('documents', JSON.stringify(documents));
        loadDocuments();
        alert("Documento eliminado con éxito.");
    }
}
```

Este anexo ofrece una visión detallada de las principales funcionalidades implementadas en el "Procesador de Textos Web", que cubren desde la edición básica de texto hasta la gestión avanzada de documentos, pasando por la exportación en diferentes formatos y la personalización del entorno de trabajo.