1. Análisis y Diseño Preliminar

1.1. Introducción a Electron JS

Electron JS es un marco de desarrollo de software de código abierto capaz de crear aplicaciones de escritorio multiplataforma utilizando tecnologías web como HTML, CSS y JavaScript. Desarrollado por GitHub y ampliamente utilizado en la industria.

Características Principales

Arquitectura Híbrida: Proceso principal y procesos de renderizado.

Proceso Principal (Main Process): El main process es el núcleo de Electron. Controla la vida útil de las ventanas y maneja las operaciones nativas del S.O. Corre en un entorno Node.js, lo que le permite acceder a las APIs del sistema.

Proceso de Renderizado (Renderer Process): Cada ventana de la aplicación tiene su propio proceso de renderizado, similar a los navegadores web. Acá es donde se cargan y ejecutan las páginas HTML, CSS y JavaScript.

Tecnologías Web: Utiliza HTML5, CSS3 y JavaScript, y también puede integrarse con otros frameworks como **React**, **Vue** y **Angular**, para crear interfaces de usuario más dinámicas.

Acceso a APIs Nativas: Posee APIs nativas para realizar tareas específicas del sistema operativo, como crear notificaciones, abrir archivos, gestionar ventanas, crear menús, etc. Esto permite que las aplicaciones tengan un comportamiento más integrado con el sistema operativo en el que se ejecutan, ya que puede interactuar directamente sobre este.

Multiplataforma: Otra de sus grandes ventajas es compilar una sola aplicación que puede ser distribuida y ejecutada en los distintos Sistemas Operativos **Windows**, **macOS** y **Linux**. Esto reduce el esfuerzo de desarrollo, el tiempo y mantenimiento de tener que gestionar diferentes bases de código para cada S.O.

Distribución y Empaquetado: Electron permite empaquetar aplicaciones de forma sencilla utilizando herramientas como **electron-builder** o **electron-forge**, que nos ayudan a facilitar la creación de instaladores y ejecutables para las diferentes plataformas.

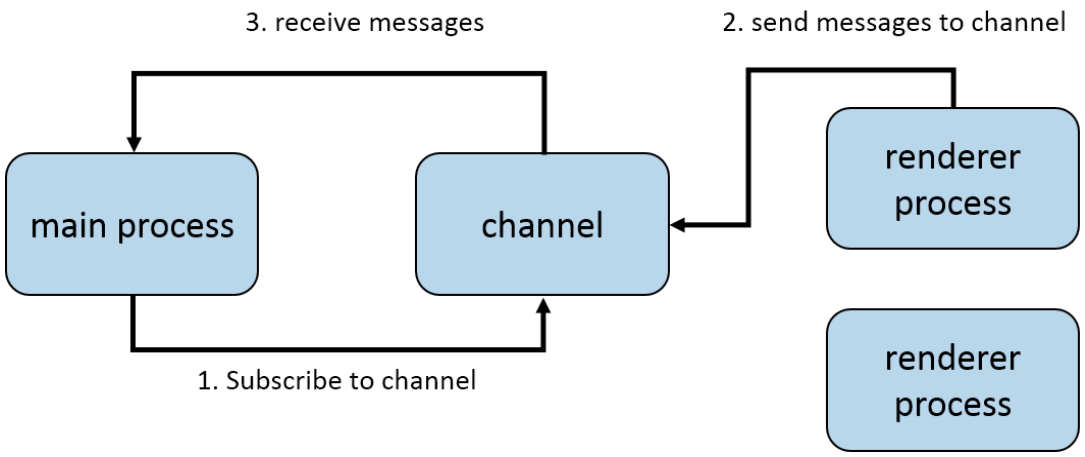
Ventajas y Desventajas

Ventajas	**Desventajas**
Código único para múltiples plataformas	Alto consumo de recursos
Fácil integración con tecnologías web	Tamaño grande de las aplicaciones
Acceso a APIs del Sistema Operativo	

Algunas aplicaciones hechas con Electron JS

- Visual Studio Code
- Slack
- Atom
- Discord

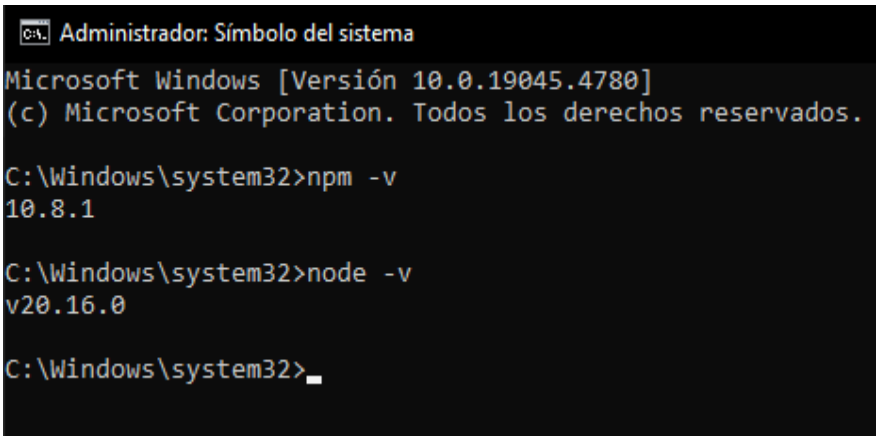
1.2. Diagrama de arquitectura de la aplicación



2. Configuración del Entorno de Desarrollo

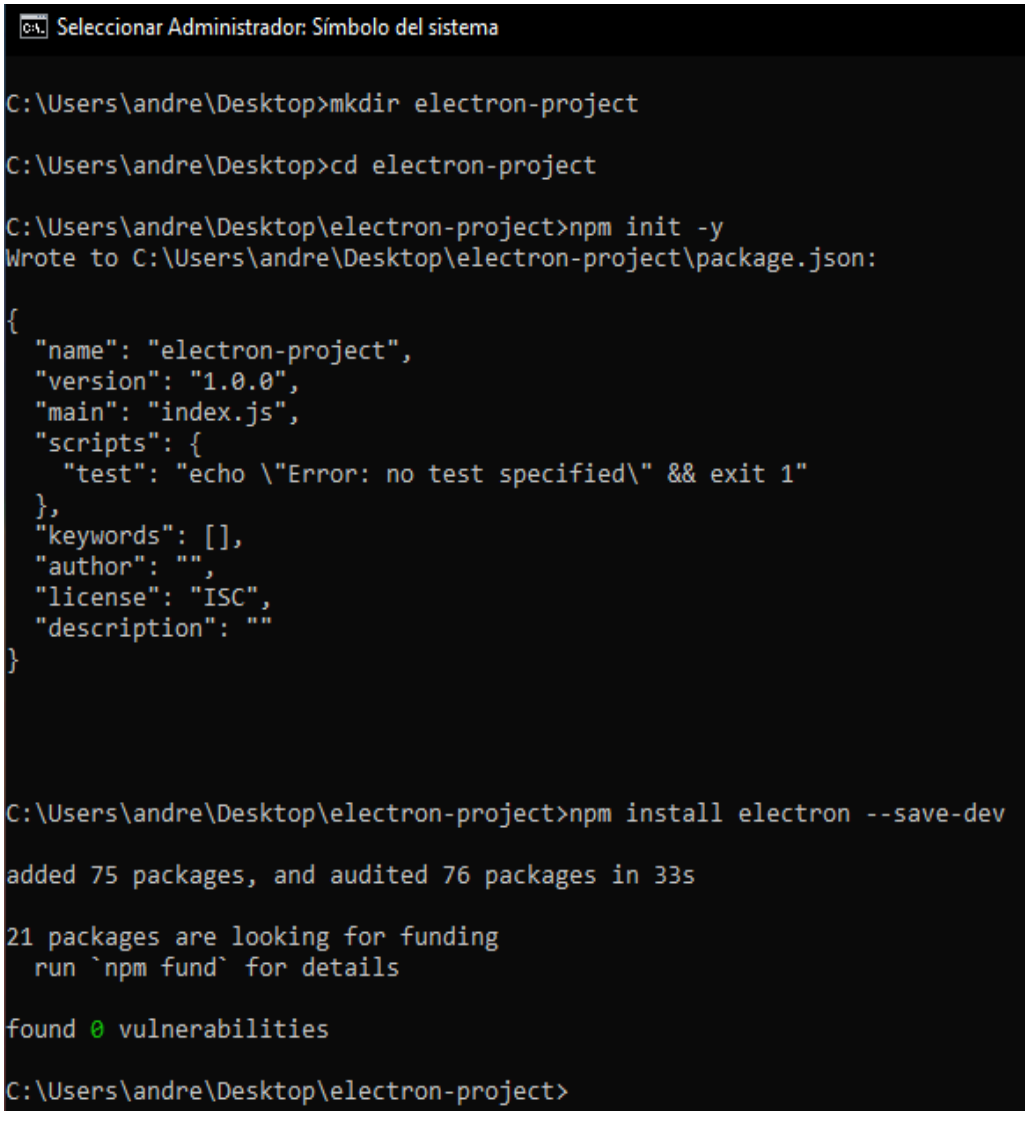
2.1. Instalación de Node.js & Node Package Manager (npm)

- Se instaló Node.js en el equipo de escritorio y portátil para trabajar desde ambos lados en un repositorio remoto
- Siempre verificamos la correcta instalación ejecutando en consola



2. 2 Inicialización del proyecto Node.js y configuración de Electron como dependencia

- Para inicializar nuestro proyecto primero creamos el directorio (en caso de no tenerlo), accedemos a él, dentro inicializamos el proyecto con Node.js e instalamos electron como dependencia.



3. Desarrollo de la Aplicación Base

3.1. Implementando estructura básica de una aplicación Electron

3.2.1 `main.js` - Proceso Principal

En mi proyecto, el archivo `main.js` actúa como el núcleo del proceso principal de Electron. Desde aquí se manejan varias tareas cruciales:

- **Inicialización de la Aplicación**: Este archivo, configura y arranca la aplicación, asegurándose de que todo esté listo para la creación y visualización de la ventana principal. Algo que es esencial para que la aplicación se inicie correctamente y funcione sin problemas desde el comienzo.
- **Creación de Ventanas**:En `main.js` se define y crea la ventana principal de la aplicación. Estableciendo propiedades como el tamaño y el título de la ventana, la URL del archivo HTML que se cargará, etc.
- **Gestión de Eventos**: Este archivo también gestiona eventos globales, como la apertura y el cierre de ventanas. Esto ayuda a manejar el ciclo de vida de la aplicación de manera efectiva, asegurando que la aplicación responda adecuadamente a las acciones del usuario y a otros eventos importantes.
- **Comunicación entre Procesos**: `main.js` facilita la comunicación entre el proceso principal y el proceso de renderizado. Utilizo el módulo `ipcMain` para recibir mensajes del proceso de renderizado y enviar respuestas. Esto es crucial para coordinar la interacción entre diferentes partes de la aplicación y mantener una comunicación fluida.

3.2.2 `index.html` - Interfaz de Usuario

El archivo `index.html` es donde se diseña y estructura la interfaz de usuario de mi aplicación. Aquí están las principales características y beneficios:

- **Estructura HTML**: En `index.html`, defino la estructura básica de la página que se mostrará en la ventana principal. Esto incluye elementos clave como botones, formularios y áreas de visualización. Tener una estructura clara y bien organizada es fundamental para una interfaz de usuario intuitiva.
- **Enlace con `renderer.js`**: Este archivo también incluye `renderer.js`, que se encarga de la lógica del proceso de renderizado. Al enlazar el script en el HTML, aseguro que la lógica de JavaScript esté disponible para manipular el DOM y responder a las acciones del usuario.
- **Estilos y Diseño**: Enlazaremos nuestro `index.html` con nuestra hojas de estilos CSS. Esto me permite aplicar estilos y mejorar la presentación visual de la interfaz, haciendo que la aplicación sea más atractiva y fácil de usar.

3.2.3 `renderer.js` - Proceso de Renderizado

En `renderer.js`, manejo la lógica del proceso de renderizado y la interacción con el DOM. Las principales características y beneficios son:

- **Manipulación del DOM**: Utilizamos `renderer.js` para interactuar con los elementos HTML en `index.html`, actualizando la interfaz en respuesta a las acciones del usuario o eventos específicos. Esto incluye tareas como modificar contenido, manejar eventos de clic, y actualizar elementos de la página. Esto me permite ofrecer una experiencia de usuario dinámica y receptiva.
- **Comunicación con el Proceso Principal**: Con `renderer.js`, también me comunico con el proceso principal utilizando el módulo `ipcRenderer`. Envío mensajes al proceso principal y recibo respuestas o datos. Esto facilita la sincronización de datos y la gestión de eventos entre el proceso principal y el proceso de renderizado.
- **Lógica de Aplicación**: Implemento la lógica específica de la aplicación en este archivo, como la validación de formularios, el manejo de datos y la interacción del usuario. Esto me permite definir el comportamiento de la aplicación y asegurar que las funcionalidades se ejecuten de manera correcta y eficiente.

4. Implementación de Funcionalidades Core

4.1. Desarrollar una aplicación de gestión de tareas (Todo List)

Para la gestión de tareas en mi aplicación, opté por utilizar una combinación de **elementos HTML** y técnicas de **manipulación del DOM** para crear una experiencia de usuario efectiva y cómoda. Inicialmente, enfrenté algunos inconvenientes con los métodos tradicionales de solicitud de datos al usuario mediante **prompts**, que resultaron ser **desactualizados** e **invasivos**. Decidí en su lugar emplear un campo de entrada (`input`) de tipo texto, integrado en la interfaz HTML, que resulta ser más fácil de crear, más ágil y proporciona una experiencia más cómoda para el usuario.

En `index.html`, diseñé la estructura de la interfaz añadiendo distintos elementos y asignándoles identificadores (`ID`) y clases específicas. Esto no solo facilita el acceso y la manipulación de estos elementos en JavaScript, sino que también permite aplicar estilos CSS de manera más eficiente. Utilizamos `getElementById` en `renderer.js` para acceder a estos elementos del DOM, lo que me permite modificarlos y actualizarlos en función de las interacciones del usuario.

4.2.2 Funciones Básicas para la Gestión de Tareas

Desarrollé varias funciones básicas para gestionar las tareas, cada una diseñada para ser reutilizable y modular:

- **Función `eliminarTarea()`**: Esta función, es la más sencilla, recibe dos parámetros: `lista` y `nuevaTarea`. Aquí, `lista` se refiere al elemento `<UL>` que contiene todas las tareas, mientras que `nuevaTarea` es el elemento `<LI>` específico que representa la tarea seleccionada. Al eliminar `nuevaTarea` de `lista`, aseguro que la tarea completa sea eliminada del listado.

- **Función `marcarTarea()`**: Esta función recibe dos parámetros: `textoTarea` y `completed` (con valor predeterminado `false`). `textoTarea` representa el elemento `<SPAN>` que contiene el texto de la tarea, y `completed` indica si la tarea está marcada como completada. Inicialmente, las tareas nuevas no están completadas, pero si se carga una lista previamente guardada, el valor de `completed` se ajusta según el estado de la tarea.

- **Función `editarTarea()`**: Esta función es más compleja y maneja la edición de tareas. Recibe varios parámetros y reemplaza el `<SPAN>` que contiene el texto de la tarea con un `<INPUT>` de tipo texto para permitir la edición.

  Durante la edición, el texto del botón “Editar” cambia a “Guardar” y su funcionalidad se actualiza para confirmar los cambios, validando que el campo no esté vacío. Los botones “Eliminar” y “Marcar Tarea” se reemplazan temporalmente, y se agrega un botón “Cancelar” para revertir los cambios, que permite volver el valor del elemento `<SPAN>` a su estado previo a la edición. Tras finalizar la edición, los botones vuelven a su estado original.

4.2.3 Función Principal para la Gestión de Tareas

- **Función `crearNuevaTarea()`**:Creé una función “padre” que sirve para gestionar la creación de nuevas tareas y coordinar las funciones anteriores. Esta función se activa cada vez que el usuario hace clic en el botón `AgregarTarea` y realiza una validación antes de proceder. También se utiliza para cargar automáticamente las tareas desde la función `CargarTareas()`, asegurando que la lista se actualice con las tareas previamente guardadas.

4.2.4 Persistencia de Datos

Para la persistencia de datos, utilizo `LocalStorage` para guardar y cargar las tareas. Implementé dos funciones específicas para este propósito:

- **Función `guardarTareas`**: Esta función se llama cada vez que se realiza un cambio en la lista de tareas. Al mantener esta funcionalidad en una función separada, puedo asegurarme de que los datos se actualicen de manera eficiente y constante.
- **Función `cargarTareas`**: Esta función se invoca cuando la aplicación se carga, recuperando la lista de tareas previamente guardadas. Si no hay tareas guardadas, la función inicializa la lista con un arreglo vacío (`[]`), garantizando que la aplicación se inicie con una lista limpia.

5. Integración de APIs Nativas de Electron:

5.1. Proceso de integración y ventajas de utilizar APIs nativas

Sistema de menús nativos para navegación y acciones

Para mejorar la experiencia del usuario en mi aplicación, integré un sistema de menús nativos utilizando la API `Menu` de Electron. Este sistema de menús proporciona un acceso sencillo a diversas funcionalidades y acciones dentro de la aplicación.

- **Creación del Menú Superior**: En `main.js`, importé el módulo `Menu` de Electron y diseñé un menú superior que incluye opciones básicas como Deshacer, Rehacer, Cortar, Copiar y Pegar. Estas funcionalidades son esenciales para la edición y manipulación de tareas y permiten al usuario gestionar su trabajo de manera más eficiente.
- **Opciones Adicionales**: También incluí opciones para Importar, Exportar y Salir del programa. La funcionalidad de Importar y Exportar se implementará en combinación con la API de Diálogos de archivo, facilitando al usuario la gestión de listas de tareas. Este enfoque permite una integración fluida y coherente de las diferentes funcionalidades, brindando una experiencia de usuario más completa y profesional.

Ventajas:

- **Experiencia de Usuario Mejorada**: Los menús nativos ofrecen una interfaz familiar y accesible, mejorando la usabilidad de la aplicación.
- **Acceso a Funcionalidades Clave**: Permiten al usuario acceder rápidamente a funciones importantes sin necesidad de navegar por la interfaz principal.
- **Consistencia con el Sistema Operativo**: Los menús nativos se integran de manera coherente con el sistema operativo, proporcionando una experiencia visual y funcional consistente.

Integración de Notificaciones del Sistema

Para alertar a los usuarios sobre el estado de sus tareas, utilicé la API `Notification` de Electron. Esta integración permite enviar notificaciones del sistema cada vez que una tarea es marcada como completada.

- **Implementación de Notificaciones**: En lugar de crear una función de notificación separada, opté por reutilizar la función existente para marcar tareas como completas. Después de que una tarea es completada, se genera una notificación utilizando la API `Notification`. Esta notificación incluye un mensaje que informa al usuario sobre la tarea completada.

Ventajas:

- **Notificaciones en Tiempo Real**: Permiten al usuario recibir alertas instantáneas sobre el progreso de las tareas, mejorando la visibilidad y el seguimiento.
- **Reducción de Interrupciones**: Al integrar la notificación con la función de completar tareas, se evita la necesidad de manejar funciones adicionales, manteniendo el código más limpio y manejable.
- **Mejora en la Experiencia del Usuario**: Las notificaciones proporcionan una forma directa y efectiva de mantener al usuario informado sin interrumpir su flujo de trabajo.

Integración de Diálogos de Archivo

La API de Diálogos de archivo se utilizó para permitir la importación y exportación de listas de tareas, facilitando la gestión de datos de manera flexible y eficiente.

- **Configuración de Diálogos de Archivo**: Implementé diálogos de archivo para que los usuarios puedan seleccionar y guardar archivos de tareas. Esta funcionalidad se integrará con el menú de Importar y Exportar, permitiendo al usuario cargar listas de tareas desde archivos y guardar sus listas actuales en formatos compatibles.

Ventajas:

- **Facilita la Gestión de Datos**: Permite al usuario importar y exportar listas de tareas de manera sencilla, ofreciendo flexibilidad en la gestión de datos.
- **Interacción Intuitiva**: Los diálogos de archivo proporcionan una interfaz familiar para seleccionar y guardar archivos, mejorando la facilidad de uso.
- **Compatibilidad y Flexibilidad**: Al permitir la manipulación de listas de tareas mediante archivos, se facilita la interoperabilidad con otras aplicaciones y plataformas.

6. Optimización de la Interfaz de Usuario

6.1 Aplicar Principios de Diseño UX/UI en la Interfaz de la Aplicación

He aplicado principios de diseño UX/UI para mejorar la experiencia del usuario y optimizar la interfaz de usuario

- **Principios de Diseño UX/UI Aplicados**:
    - **Consistencia**: Aseguré que todos los elementos de la interfaz (botones, menús, campos de entrada) sigan un diseño y estilo uniforme. Esto facilita el aprendizaje y uso de la aplicación por parte del usuario.
    - **Claridad y Simplicidad**: Mantengo la interfaz libre de elementos innecesarios, asegurando que los usuarios puedan realizar tareas sin distracciones. Las acciones principales están claramente visibles y accesibles.
    - **Feedback Inmediato**: Implementé respuestas visuales y sonoras para las acciones del usuario, como notificaciones y mensajes de confirmación, para que el usuario sepa que su acción ha sido completada y procesada.
    - **Accesibilidad**: Me aseguré de que los elementos de la interfaz sean accesibles para todos los usuarios, incluidos aquellos con discapacidades, utilizando un contraste adecuado y etiquetas descriptivas.

Impacto en la Usabilidad:

- **Facilita el Aprendizaje**: La consistencia y simplicidad permiten que los usuarios comprendan rápidamente cómo usar la aplicación.
- **Mejora la Eficiencia**: El feedback inmediato y los elementos bien diseñados reducen la necesidad de errores y ayudan a completar tareas de manera más eficiente.
- **Acceso Universal**: La accesibilidad garantiza que la aplicación pueda ser utilizada por una audiencia más amplia.

6.2 Implementar un Diseño Responsivo Utilizando CSS Moderno

Para asegurar que la aplicación se vea bien en diferentes dispositivos, implementé un diseño responsivo utilizando técnicas modernas de CSS.

- **Uso de Flexbox**:
    - Utilicé Flexbox para alinear y distribuir espacio entre los elementos dentro de contenedores flexibles. Esto facilita la creación de diseños que se ajustan dinámicamente a diferentes tamaños de pantalla sin la necesidad de incluir mediaQueries específicos para cada resolución.

Impacto en la Usabilidad:

- **Adaptabilidad**: El diseño responsivo garantiza que la aplicación se vea y funcione bien en dispositivos de diferentes tamaños, desde monitores de baja resolución hasta monitores de alta resolución.

6.3 Documentar las Decisiones de Diseño y su Impacto en la Usabilidad

La documentación de las decisiones de diseño detalla cómo las elecciones realizadas afectan la experiencia del usuario.

- **Decisiones de Diseño**:

    - **Aplicación de Principios UX/UI**: Elegí principios como consistencia y claridad para garantizar una experiencia de usuario fluida y eficiente.
    - **Uso de Flexbox** : Opté por Flexbox para asegurar un diseño adaptativo y flexible, que se ajusta a diversos tamaños de pantalla sin comprometer la usabilidad.

Impacto en la Usabilidad:

- **Eficiencia y Comodidad**: Las decisiones de diseño contribuyen a una interfaz intuitiva y fácil de usar, mejorando la eficiencia en la realización de tareas y la satisfacción del usuario.
- **Adaptación a Diferentes Dispositivos**: La implementación de un diseño responsivo asegura que la aplicación sea accesible y funcional en todos los dispositivos, proporcionando una experiencia de usuario consistente y agradable.

7. Empaquetado y Distribución

7.1 Investigar Herramientas de Empaquetado

Para el empaquetado de la aplicación, utilizamos la herramienta `electron-builder`.

- **Electron-builder**: Esta herramienta simplifica la creación de paquetes de instalación para diferentes sistemas operativos y proporciona opciones avanzadas de configuración y personalización.

7.2 Configurar el Proceso de Empaquetado para Múltiples Plataformas

Configurar el proceso de empaquetado incluyó la creación de archivos de configuración específicos para cada plataforma objetivo.

- **Configuración para Windows y macOS**: Utilicé `electron-builder` para configurar los archivos de paquete (`.exe` para Windows y `.dmg` para macOS), adaptando las opciones de configuración según las necesidades de cada sistema operativo.

7.3 Generar Ejecutables para Al Menos Dos Sistemas Operativos Distintos

Generé ejecutables para Windows y macOS, asegurando que la aplicación estuviera disponible en las dos plataformas principales.

- **Windows**: Creé un instalador `.exe` utilizando `electron-builder`.
- **macOS**: Generé un archivo de imagen de disco `.dmg` para facilitar la instalación en sistemas macOS.

- **Herramientas Utilizadas**: `electron-builder` para empaquetado y generación de instaladores.
- **Pasos de Configuración**:
    - Instalación y configuración de `electron-builder`.
    - Creación de archivos de configuración para Windows y macOS.
    - Generación de paquetes e instaladores para ambas plataformas.

8. Documentación en el Código Fuente

- **Comentarios Claros y Precisos**: Explicaciones en el código sobre el propósito y funcionamiento de las funciones y módulos.
- **Documentación de API**: Descripción de las funciones y clases disponibles para otros desarrolladores.
- **Convenciones de Codificación**: Adherencia a buenas prácticas y estilos de codificación.

9. Control de Versiones y Colaboración

9.1 Utilizar Git para el Control de Versiones

Utilicé Git para gestionar las versiones del proyecto, permitiendo un control efectivo de los cambios en el desarrollo.

- **Comandos Básicos**: Uso de comandos como `git add, git commit`, `git push` y `git pull` para manejar versiones y sincronizar el código.

9.2 Crear un Repositorio en GitHub o GitLab

Creé un repositorio en GitHub, incluyendo:

- **README.md Completo**: Un archivo README con una descripción del proyecto, instrucciones de instalación y uso, y detalles sobre el proceso de desarrollo.