

## Lenguaje de programación:

Lenguaje	Ventajas	Desventajas
JavaScript	<ul style="list-style-type: none"> <li>- Conocimientos previos: Todos los integrantes del equipo manejan JavaScript previo a la realización del proyecto. Además, se tiene un conocimiento sólido sobre el Modelo Vista Controlador, y los paradigmas para desarrollo de frontend, backend e integración en el ambiente de JavaScript, así como conocimiento de múltiples de sus frameworks. Esto permite que sea uno de los lenguajes analizados con una mayor velocidad de desarrollo de los analizados.</li> <li>- Agilidad: Además del conocimiento previo, los frameworks de JavaScript fueron diseñados para pensar en la agilidad del desarrollo.</li> <li>- Asincronía: JavaScript no es concurrente, pero si logra optimizar la ejecución de computación mononúcleo de mejor manera que otros de los lenguajes propuestos para el proyecto, bajo el concepto de asincronía. En JavaScript se pueden definir procesos asíncronos, promesas, callbacks entre otras herramientas que permitan mantener la interfaz de usuario reactiva y funcional, mientras se ejecutan otros procesos de backend.</li> <li>- Gestión de memoria: JavaScript gestiona la memoria de manera automática, liberando carga cognitiva y laboral a los desarrolladores en este aspecto.</li> <li>- Interfaz: Múltiples frameworks de desarrollo de JavaScript se han desarrollado por años para permitir la construcción de interfaces gráficas reactivas y altamente personalizadas, aprovechando al máximo el asincronismo para una buena experiencia de usuario. Esta es una prioridad fundamental para la idea de proyecto propuesta en este</li> </ul>	<ul style="list-style-type: none"> <li>- Fiabilidad: El código de JavaScript es fácil de leer, es estético, y permite identificar fácilmente errores de lógica de negocio. No obstante, presenta desafíos superiores en testing, cuando se presentan errores de compilación. Esto es principalmente por el tipado dinámico, lo que requiere un proceso de detección de errores en tiempo de ejecución, que es más desafiante y requiere un mayor consumo de tiempo.</li> <li>- Memoria: El uso de motores de ejecución para Máquinas Virtuales de JavaScript favorece una interfaz rápida y reactiva, pero también genera proyectos más pesados que los propuestos por otros lenguajes de programación.</li> <li>- Arquitectura: Por diseño, las aplicaciones de JavaScript son modulares, y buscan que múltiples componentes individuales interactúen entre sí para el funcionamiento del proyecto global. No obstante, existen herramientas como Electron que permiten la integración de los distintos frameworks de JavaScript en un solo entorno monolítico, pensado particularmente para aplicaciones de escritorio, lo que permite circunnavegar este defecto.</li> </ul>

	ejercicio.	
Python	<ul style="list-style-type: none"> <li>- Conocimientos previos: Python es uno de los 2 lenguajes estudiados que son manejados por los 4 integrantes del equipo. Teniendo en cuenta las claras restricciones de tiempo, y la necesidad de aplicar metodologías ágiles, para una entrega de código funcional en un periodo corto de tiempo, las ventajas funcionales de un lenguaje pueden no ser suficientes para compensar la desventaja de capacitar a los miembros faltantes, para que el equipo sea productivo.</li> <li>- Código limpio: La sintaxis de Python es muy sencilla y limpia, lo que significa que un proyecto desarrollado en este permitiría una visualización fácil del código. Por ejemplo, el uso de indentación en el lenguaje obliga a visualizar la jerarquía del código.</li> <li>- Gestión de memoria: Python gestiona de manera automática la memoria, lo que hace más rápido el desarrollo, y facilita la implementación de lógica de negocio o interfaz visual.</li> <li>- Bases de datos: El lenguaje y su ecosistema tienen herramientas adicionales de integración de bases de datos como SQLite, lo que facilita su implementación y provee seguridad en el proceso de desarrollo.</li> </ul>	<ul style="list-style-type: none"> <li>- Rendimiento: Es bien conocido que las ventajas que tiene Python en ser accesible a los desarrolladores, viene con costos computacionales. Entre los causantes de estos se encuentra el tipado dinámico (aunque ha tenido grandes mejoras en los últimos años). Otra de las razones de sus deficiencias de rendimiento es el GIL (Global Interpreter Lock). Esto prácticamente impide (o limita severamente) el rendimiento de un proyecto en un equipo con computación paralela, particularmente severo en proyectos de amplia envergadura. Los frameworks de frontend más ágiles de Python, como PyQt, no son nativos de Python, por lo que no son válidos para el desarrollo del proyecto, impidiendo la navegación alrededor de este defecto.</li> <li>- Fiabilidad: El tipado dinámico logra una mayor agilidad en el desarrollo, pero también deja al proyecto con vulnerabilidades por errores de tipado. El testing de un proyecto en Python también presenta problemas de rendimiento, peculiarmente teniendo en cuenta que el tipado dinámico es definido en tiempo de ejecución, y no en tiempo de compilación. Esto significa que el tiempo para detectar errores es mayor.</li> </ul>
Java	<ul style="list-style-type: none"> <li>- Código limpio: Java favorece la implementación de código limpio. Principalmente por el tipado estático, y su naturaleza orientada a objetos, es sencillo entender la lógica del programa, centrado en el paradigma POO.</li> <li>- Rendimiento: Aunque no es la herramienta superior de las evaluadas, los programas desarrollados en Java suelen tener muy buen rendimiento. Esto es porque la estrategia de ejecución por la Máquina Virtual de Java permite una buena velocidad de ejecución,</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Hay integrantes que han desarrollado proyectos en Java, pero no todos los integrantes dominan el lenguaje. Sería necesario proveer tutoría a los integrantes con menos experiencia en este lenguaje.</li> <li>- Memoria: Tanto el paradigma de la Programación Orientada a Objetos, como el uso de Máquinas Virtuales, tienen como desventaja un mayor uso de memoria en sus proyectos. Java combina ambos factores, lo que se manifiesta como proyectos más pesados, lo que puede ser un factor detractor</li> </ul>

	<p>especialmente comparado con los lenguajes de código nativo.</p> <ul style="list-style-type: none"> <li>- Portabilidad: La implementación de la Máquina Virtual significa que la ejecución del proyecto en otro equipo depende de la Máquina Virtual que tenga instalada, y no de sus características nativas. Esto facilita significativamente el proceso de asegurar que el proyecto sea ejecutable por distintos computadores.</li> <li>- Gestión de memoria: Java también tiene gestión automática de memoria. Para un proyecto con lógica de negocio sencilla, sin requerimientos específicos de hardware, se prefiere obviar este aspecto del desarrollo.</li> <li>- Fiabilidad: La gestión de memoria automática y el tipado estático exigente de Java hacen los proyectos más resistentes a fallas, y el testing más fácil de enfocar a lógica de desarrollo.</li> </ul>	<p>en un proyecto intensivo en GUI como el presente.</p> <ul style="list-style-type: none"> <li>- Velocidad en caliente: La velocidad de compilación de Java es muy competitiva, pero la velocidad de iteración de ejecutables es deficiente, comparada con otros lenguajes diseñados para esto. Java funciona muy bien en aplicaciones tipo CRUD o lógica de negocio, pero puede no ser óptimo para aplicaciones intensivas en GUI, comparado con otros lenguajes diseñados para esta labor.</li> </ul>
C++	<ul style="list-style-type: none"> <li>- Rendimiento: C++ es uno de los lenguajes con mayor velocidad de ejecución, y gestión de memoria comparados en este ejercicio.</li> <li>- Frameworks: C++ tiene múltiples Frameworks nativos altamente optimizados para manejo de GUIs interactivas. Entre estos Frameworks se encuentran Qt y wxWidgets. Para el ejercicio planteado, el enfoque en Frameworks con buenos componentes estéticos y rendimiento es una de las prioridades en la elección.</li> <li>- Control de bajo nivel: No tan necesario para el proyecto. C++ presenta grandes ventajas en la gestión de memoria personalizada, y el control de elementos de hardware.</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Algunos integrantes del equipo saben manejar C++, por ejemplo, para la asignatura de Algoritmos. No obstante, no todos los integrantes son proficientes, y C++ se caracteriza por una elevada curva de aprendizaje, lo cual no es práctico para los requerimientos de tiempo del proyecto.</li> <li>- Código limpio: C++ puede implementar técnicas de código limpio. Pero hacer esto es más desafiante, ya que elementos como la gestión de memoria por naturaleza pueden ser difíciles de leer.</li> <li>- Depuración y testing: El testing en C++ es notoriamente más difícil, especialmente porque la depuración de bajo nivel, manejo de registros o punteros requiere herramientas y conocimientos particulares, lo que además es intensivo en consumo de tiempo, especialmente para un equipo más inexperto.</li> </ul>

C#	<ul style="list-style-type: none"> <li>- Código limpio: La sintaxis de C# está diseñada para ser accesible y fácil de aprender. También se caracteriza por tener tipado estático.</li> <li>- Rendimiento de compilación: Similar a Java, C# ejecuta su código por medio de una Máquina Virtual. Esto lo hace por medio del CLR (Common Language Runtime) que se encarga de traducir el código intermedio generado en el proceso de compilación, para ejecutar el código traducido en una Máquina Virtual. El proceso de traducción es en tiempo real, lo que lo hace particularmente eficiente.</li> <li>- Manejo de memoria: C# incluye manejo de memoria automático, por lo que se agiliza este proceso de desarrollo.</li> <li>- Asincronía: Las herramientas de ejecución asíncrona son particularmente útiles para aplicaciones interactivas, como el proyecto de aplicación de escritorio considerado para este proyecto.</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Ninguno de los integrantes del equipo tiene conocimientos previos en C# o en el entorno de desarrollo de Windows. Por lo tanto, se tienen que adquirir muchos conocimientos previos para poder implementar correctamente un proyecto en este lenguaje.</li> <li>- Compatibilidad: C# fue diseñado exclusivamente para compatibilidad nativa con Windows. En la actualidad hay herramientas como .NET MAUI y Avalonia UI que permiten ejecutar proyectos multiplataforma, pero todavía presentan desafíos significativos. Esto es particularmente notorio para el equipo de desarrollo, puesto que uno de los integrantes desarrolla en Ubuntu, lo que presenta dificultades adicionales para el desarrollo del proyecto.</li> <li>- Memoria: Similar a Java, el uso de Máquina Virtual para el proceso de compilación presenta desafíos adicionales en el consumo de memoria. Esto no es particularmente necesario para un proyecto con énfasis en GUI como el presente.</li> </ul>
Dart	<ul style="list-style-type: none"> <li>- Código limpio: DART implementa muchas herramientas de fiabilidad en la escritura del código, como lo son el tipado estático, la seguridad nula o el uso de isolates para hacer la concurrencia de manera explícita. Todo esto favorece un código fácil de leer para entender la lógica de ejecución</li> <li>- Rendimiento en compilación: DART funciona con Compilación Anticipada (AOT). Esta consiste en compilar un código de Máquina Virtual a lenguaje de máquina. Este proceso adicional de compilado permite optimizar el uso de memoria del programa compilado, manteniendo ventajas de la Máquina Virtual, como la velocidad de compilación y compatibilidad entre dispositivos.</li> <li>- Rendimiento en tiempo real: El sistema secuencial de compilación JIT seguido de</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimiento previo: Ninguno de los integrantes del equipo tiene experiencia en Dart o Flutter. No es tan difícil de aprender desde cero como otros lenguajes y frameworks comparados en este documento, pero si puede ser un factor detractor en el contexto actual de producción de software ágil.</li> <li>- Herramientas para escritorio: Dart fue diseñado para el desarrollo de aplicaciones móviles. Flutter tiene compatibilidad y herramientas para desarrollar aplicaciones de escritorio nativas, como las de este proyecto, pero el marco de herramientas disponibles para aplicaciones de escritorio no es tan maduro y desarrollado como el de aplicaciones móviles. Es posible encontrarse con limitaciones de funcionalidad, o herramientas que requieran un testing más intensivo que frameworks optimizados para</li> </ul>

	<p>compilación AOT es particularmente optimizado para GUIs interactivas en tiempo real, lo que se asemeja a las necesidades de nuestro proyecto. También está diseñado para permitir concurrencia, otra herramienta útil en nuestro proyecto, como se discute en JavaScript.</p> <ul style="list-style-type: none"> <li>- Interfaz gráfica: El framework Flutter de DART tiene múltiples librerías y herramientas diseñadas para desarrollar interfaces gráficas estéticas y reactivas, lo que también es una prioridad para la muestra del recetario de cócteles que planteamos desarrollar.</li> </ul>	aplicaciones de escritorio
Rust	<ul style="list-style-type: none"> <li>- Código limpio: Rust introduce muchos conceptos únicos del lenguaje de programación. No obstante, estos conceptos fuerzan a la implementación de código limpio y seguro. Los códigos funcionales de Rust permiten una organización y escritura intuitiva del código, facilitando la lectura y comprensión, siempre y cuando se entienda el paradigma que envuelve a Rust.</li> <li>- Rendimiento: Rust fue diseñado buscando una velocidad de compilación y ejecución máxima. Una de las maneras que la logra es implementando abstracciones de costo cero. Las abstracciones de costo cero son características del lenguaje que pueden ser compiladas directamente, sin requerir interpretación, y por tanto compilando de manera directa y más veloz. Esto es particularmente útil en el desarrollo de GUIs interactivas, como se propone en nuestro proyecto.</li> <li>- Fiabilidad: Rust introduce el concepto de escudos, y de manera general, un nuevo nivel de seguridad. A diferencia de lenguajes de bajo nivel como C++, los escudos de Rust permiten evitar de manera absoluta problemas de bajo nivel, como punteros nulos o incorrectos, fugas de memoria o condiciones de carrera</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Ninguno de los integrantes tiene conocimiento previo sobre Rust. De los lenguajes seleccionados, este es el que tiene una curva de aprendizaje más compleja, porque maneja muchos conceptos exclusivos del lenguaje. Este proceso de capacitación es particularmente poco deseable en el contexto actual de desarrollo ágil con fechas de entrega exigentes y alta producción de código en poco tiempo</li> <li>- Herramientas para escritorio: Rust no ha sido desarrollado de manera nativa, pensando en aplicaciones de escritorio nativas. Particularmente, Rust no fue desarrollado para frontend. Existen frameworks nuevos para incluir frontend en Rust, con muy buenos diseños estáticos, pero la implementación y fiabilidad de de estos frameworks todavía está en proceso de desarrollo. Esto hace más difícil cumplir el requerimiento de desarrollar toda la aplicación en un solo lenguaje de programación</li> </ul>

	<p>entre procesos. Esto simplifica de manera significativa el proceso de testing, permitiendo enfocarlo a lógica de negocio, que para las rúbricas del proyecto es lo más fundamental. Esto se logra a cambio de aumentar el tiempo de desarrollo del proyecto, así como los conocimientos necesarios del desarrollador.</p> <ul style="list-style-type: none"> <li>- Interfaz gráfica: El framework Tauri ha sido desarrollado para permitir el uso de Rust en aplicaciones de escritorio nativas. Implementa bien la reactividad, y tiene muy buenas herramientas de diseño e interfaz amigable. Por tanto, un proyecto con este framework puede tener resultados muy satisfactorios, cumpliendo con los requerimientos del proyecto.</li> </ul>	
Kotlin	<ul style="list-style-type: none"> <li>- Código limpio: Kotlin fue desarrollado para implementar el paradigma POO/tipado fuerte de Java, proponiendo adicionalmente una sintaxis más rápida de escribir y fácil de leer que Java. En este aspecto, se presenta como superior a Java, y una de las mejores opciones de los lenguajes comparados.</li> <li>- Rendimiento en compilación: Por defecto, Kotlin compila usando la Máquina Virtual de Java. Esto incluye las ventajas en ejecución mencionadas previamente en Java.</li> <li>- Rendimiento en caliente: Adicional a la Máquina Virtual, Kotlin puede implementar LLVM, el cual traduce el código intermedio de la Máquina Virtual de Java a código de máquina, permitiendo también obtener los beneficios de rendimiento en caliente mencionados en tiempo real, lo que es particularmente útil en una aplicación de GUI interactiva como la nuestra.</li> <li>- Fiabilidad: Kotlin tiene múltiples herramientas adicionales a Java para facilitar la seguridad y evitar errores detectables en testing, manteniendo su flexibilidad superior a Java. La más</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Ninguno de los integrantes del grupo tiene conocimientos previos sobre Kotlin. Por una parte, es uno de los lenguajes más fáciles de aprender desde cero de los mencionados en este análisis, especialmente cuando los desarrolladores tienen experiencia con Java (esto si se cumple en el equipo de desarrollo). Por otra parte, como hay integrantes que no han manejado Java previamente, es necesario reforzar conceptos del paradigma POO y el tipado fuerte de Java antes de aprender la sintaxis de Kotlin.</li> <li>- Herramientas para escritorio: El framework Compose Multiplatform permite el desarrollo de aplicaciones de escritorio nativa, pero es un framework menos maduro que muchos de otros lenguajes ya analizados, lo que plantea limitaciones en el proceso de diseño, desarrollo y testing.</li> </ul>

	<p>notoria de estas es la seguridad nula. Contrario a lo que se asume de su nombre, la seguridad nula es una herramienta que fuerza al programador a declarar cuando una variable tiene el potencial de retornar un valor nulo. Esto permite evitar excepciones innecesarias y difíciles de entender, al forzar al compilador de Kotlin a verificar la nulidad del objeto en tiempo real, permitiendo un grado de flexibilidad en el proceso de desarrollo, agilizando la producción de código funcional, especialmente en un equipo inexperto.</p> <ul style="list-style-type: none"> <li>- Compatibilidad: Kotlin ofrece compatibilidad absoluta y garantizada con Java, y sus respectivas máquinas virtuales, facilitando las necesidades de despliegue y compatibilidad multiplataforma del proyecto. Además, tiene compatibilidad con otras Máquinas Virtuales, por ejemplo, la de JavaScript (esto es una ventaja de JavaScript directamente).</li> </ul>	
--	---	--

Para la decisión final, se escogieron tres finalistas 🏆 basados en los requerimientos de capacitación del equipo. Para un lenguaje de programación con altos conocimientos previos 📖, la decisión fue entre JavaScript y Python 🐍, decidiéndonos últimamente por JavaScript en esta categoría. La razón es que JavaScript es notoriamente mejor desarrollado para la implementación de aplicaciones estéticamente agradables 🎨 y con grandes requerimientos de interactividad ⚙️. Las ventajas de código limpio y velocidad de desarrollo de Python son cubiertas de manera aceptable por los frameworks de JavaScript, cuyo potencial de un producto final de alta calidad ⭐ es superior en este campo.

Para programas de capacitación rápida ⚡, se compararon Java ☕, C# y Kotlin. Entre estos tres, se escoge Kotlin como lenguaje candidato. Aunque requiere conocimientos adicionales a las necesidades del equipo con Java, la sintaxis es más fácil de usar 🖋️, sobre todo en proyectos sencillos, por lo que el proceso mayor de aprendizaje puede ser compensado por una mayor velocidad de desarrollo 🚀. Es más rápido de aprender, y presenta mejor compatibilidad con Ubuntu 🐧 que C# (el uso de Ubuntu por parte del equipo fue un detractor mayor para usar C#). Las características de código seguro 🔒 de Kotlin, sumados a su sintaxis sencilla lo hacen una elección muy adecuada para necesidades de producción rápida 🔧, si no se puede usar un lenguaje que ya se domina.

Para programas de capacitación extensiva 📖 (C++, Dart y Rust 🦀), se escoge como candidato a Dart. Dart requiere una mayor capacitación en sintaxis y lógica que C++, ya que de este sí hay experiencia. Pero la experiencia del equipo comparada con las necesidades de construir una aplicación e interfaz gráfica 🖥️ es bastante escasa, y los desafíos a futuro

en el proceso de testing 🧪 y debugging 🐛 con C++ son notoriamente superiores a los previstos con Dart y Rust, por lo que C++ es descartado. Adicionalmente, la curva de aprendizaje 📈 de Rust es mucho más pronunciada que Dart, y para las necesidades de tiempo del proyecto es simplemente inviable 🚫 crear una aplicación de escritorio nativa funcional con Rust, que cumpla los requerimientos funcionales y no funcionales 📄 ya definidos.

Finalmente, se escoge entre los candidatos actuales, JavaScript, Kotlin y Dart. La decisión final es utilizar JavaScript 🧑‍💻. El primer descartado es Dart. Aunque la curva de aprendizaje prevista 📈 sea menor que Rust, sigue siendo sustancial comparada con JavaScript y Kotlin, lo que dificulta la elección de este lenguaje. Adicionalmente, Dart todavía es inmaduro para aplicaciones de escritorio 🖥️, lo que limita su potencial real comparado con JavaScript y Kotlin. El principal factor detractor de JavaScript es que los frameworks clásicos no permiten construir aplicaciones monolíticas nativas 📦. Pero esta desventaja es compensada si se utiliza el framework Electron ⚡ como empaquetador. A pesar de esto, el uso del empaquetador es ligeramente menos eficiente en ejecución 🐢, y mucho menos eficiente en consumo de memoria 🧠 que Kotlin. Pero a diferencia de JavaScript, las herramientas de framework ya desarrolladas por Kotlin 🔧 para construir interfaces estéticas y altamente reactivas en escritorio son mucho más inmaduras que las herramientas de JavaScript. Por tanto, especialmente con un equipo relativamente inexperto 👥, aprovechar las herramientas ya existentes permite crear un proyecto con una interfaz muy adecuada 🍷 para los requerimientos de manera notoriamente más ágil ⚙️. Esta comparación entre JavaScript y Kotlin concluye con la elección de JavaScript como único lenguaje de programación ✅.

## Frameworks:

Framework	Ventajas	Desventajas
Electron	<ul style="list-style-type: none"> <li>- Compatibilidad: Se fundamenta en empaquetar un proyecto JavaScript en un navegador Chromium, el cual puede ser ejecutado en una aplicación de escritorio. Este estilo particular de empaquetamiento le permite ser compatible en multiplataforma, particularmente útil si el equipo de desarrollo está dividido entre Windows y Ubuntu.</li> <li>- Interactividad: Así como Node es naturalmente monohilo, el empaquetamiento de programas Node en Electron permite realizar la ejecución de múltiples procesos Node en paralelo. Usualmente ejecuta de manera constante un proceso</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Ninguno de los integrantes del equipo de trabajo tiene experiencia en aplicaciones de escritorio usando JavaScript, ni en herramientas empaquetadoras. Existen herramientas más fáciles de aprender que Electron.</li> <li>- Memoria: De los tres empaquetadores analizados en este documento, Electron es cómodamente el que exige mayor consumo de memoria de los tres. El uso del navegador Chromium precisamente tiene un consumo de memoria significativo, incluso en las instancias más basales de la aplicación.</li> <li>- Rendimiento: Las aplicaciones de Electron compilan rápidamente, y permiten una reactividad muy buena. Pero para conseguir</li> </ul>



	<p>principal, que mantiene activa la ventana, y crea un nuevo hilo de ejecución Node por cada requerimiento funcional inmediato (renderización, comunicación con base de datos, etc.). Por este paralelismo, Electron es muy útil para aplicaciones que requieran alta interactividad con el usuario, como es el caso de nuestra aplicación.</p> <ul style="list-style-type: none"> <li>- Interfaz gráfica: Electron presenta amplio soporte para herramientas frontend como React o Angular. Por tanto, permite implementar de manera fiel los diseños estéticos construidos en frontend especializados, asegurando una buena experiencia de usuario (prioridad fundamental del proyecto).</li> <li>- Arquitectura: Por naturaleza, Electron construye arquitecturas monolíticas, lo que permite la construcción de una aplicación basada en el paradigma Modelo-Vista-Controlador, de una manera contenida que cumpla las necesidades arquitectónicas del proyecto.</li> </ul>	<p>este buen rendimiento, presenta necesidades mayores en consumo de CPU y RAM, lo que hace al proyecto final levemente inaccesible para usuarios con computadores más modestos.</p> <ul style="list-style-type: none"> <li>- Seguridad: La integración de un entorno de aplicativo web en una aplicación de escritorio, implica comunicación con el Sistema Operativo y Sistema de Archivos del usuario. En proyectos comerciales, esto genera riesgos en la seguridad del usuario, dependiendo de cómo se implementan estas interacciones en el backend. Como nuestro proyecto es completamente autocontenido, esta no es una preocupación decisiva.</li> </ul>
NW.js	<ul style="list-style-type: none"> <li>- Conocimientos previos: Ninguno de los integrantes del equipo tiene experiencia usando empaquetadores para aplicaciones de escritorio. No obstante, entre los tres empaquetadores analizados, NW es claramente el más fácil de interpretar.</li> <li>- Memoria: El renderizado en el aplicativo de escritorio se realiza directamente sobre el DOM, en vez de utilizar un Virtual DOM.</li> <li>- Interfaz gráfica: Tiene una interacción más directa con Node que Electron, lo que permite el desarrollo de interfaces altamente reactivas con un menor consumo de memoria.</li> <li>- Bases de datos: NW tiene su propia herramienta de integración a SQLite, lo que hace el uso de bases de datos relacionales particularmente sencillo.</li> </ul>	<ul style="list-style-type: none"> <li>- Arquitectura: La implementación de NW se fundamenta en la implementación de APIs de otros programas de JavaScript de backend, en vez de desarrollar la lógica de negocio directamente en el software. Por tanto, no es una opción idónea para un proyecto como el nuestro, que no puede centrarse en el llamado a APIs.</li> <li>- Seguridad: Las vulnerabilidades de seguridad al usuario en NW son todavía mayores que las de Electron. Tiene la desventaja añadida de interactuar directamente con el DOM, lo que lo hace vulnerable a usuarios introduciendo código malicioso en el programa.</li> <li>- Testing: El proceso de testing es mucho más difícil de aplicar, debido a la dependencia de APIs en la ejecución del proyecto.</li> </ul>
Tauri	<ul style="list-style-type: none"> <li>- Código limpio: El uso de Rust, y sus características principales como los escudos es muy dado para la implementación de código limpio, tal como se comentó en la sección de</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos:</li> <li>- Lenguaje: Tauri está escrito parcialmente en Rust, y necesita código Rust para funcionar. Por tanto, para los requerimientos del proyecto de manejar sólo un idioma, Tauri</li> </ul>

	<p>Rust de la sección anterior.</p> <ul style="list-style-type: none"> <li>- Rendimiento en compilación: No requiere un motor de navegación que ejecute el aplicativo, a diferencia de Electron y NW. Esto le permite compilar y ejecutarse más rápidamente que los otros dos empaquetadores, porque compila directo a lenguaje de máquina.</li> <li>- Memoria: Tauri es notoriamente mejor empaquetando un proyecto completo de aplicación de escritorio que los empaquetadores previos.</li> <li>- Bases de datos: Incluye tauri-plugin-sql, que es un plugin SQLite, lo que facilita la integración de bases de datos relacionales.</li> <li>- Aplicabilidad a escritorio: Tauri fue diseñado para aplicaciones de escritorio, lo que hace el proceso de despliegue y desarrollo de requerimientos más sencillo que con los otros dos empaquetadores.</li> </ul>	<p>sólo sería viable si se decide desde el inicio construir todo el proyecto en Rust.</p> <ul style="list-style-type: none"> <li>- Arquitectura: Tauri se enfoca en empaquetar frontend, backend, y coordinar sus interacciones, por lo que tendrían que realizarse modificaciones a la implementación para adaptarse a la arquitectura monolítica.</li> </ul>
React	<ul style="list-style-type: none"> <li>- Conocimiento previo: Múltiples integrantes del grupo han desarrollado proyectos previamente con React, por lo que se conocen los conceptos fundamentales necesarios para desarrollar un proyecto de frontend con React.</li> <li>- Código limpio: React busca la mayor separación de responsabilidades posible entre componentes. Esto se alinea de forma directa con los principios de desarrollo SOLID, y una fácil implementación de código.</li> <li>- Modularidad: El framework React gira en torno al concepto de componentes. Los componentes son piezas de código reutilizables e independientes entre sí, que se pueden implementar de manera flexible. Esta flexibilidad le permite adaptarse a distintas arquitecturas, como lo pueden ser el MVC, Flux o Redux.</li> <li>- Interfaz gráfica: React tiene un entorno robusto de herramientas para el desarrollo de GUIs estéticamente agradables, personalizables y altamente interactivas.</li> </ul>	<ul style="list-style-type: none"> <li>- Memoria: React utiliza un virtual DOM para renderizar sus componentes. Un Virtual DOM es una representación virtual del DOM tradicional que usan los navegadores para renderizar una página o aplicativo web.</li> <li>- Rendimiento: El proceso de ejecución usando un virtual DOM hace su compilación y ejecución más lenta que algunos de los otros frameworks de frontend mencionados en esta comparación.</li> <li>- Arquitectura: Los patrones de uso más comunes de React requieren el uso de APIs. Además, no está diseñado para arquitecturas monolíticas, por lo que toca realizar adaptaciones para integrarlo de una manera aceptable para el proyecto.</li> </ul>

Vue	<ul style="list-style-type: none"> <li>- Curva de aprendizaje: Vue tiene un estilo de desarrollo incremental, que lo hace más sencillo de aprender que otros frameworks de frontend de esta comparación, sobre los cuales el equipo de desarrollo no tiene conocimientos previos.</li> <li>- Código limpio: Similar a React, Vue se fundamenta en la modularidad, y la separación máxima de responsabilidades entre módulos, haciéndolo un framework ideal para implementar código limpio.</li> <li>- Flexibilidad: Además de la utilidad para código limpio, Vue es bastante adaptable a múltiples tipos de proyectos, incluyendo aplicaciones contenidas de escritorio, como es el caso de este proyecto.</li> <li>- Reactividad: Las GUIs construidas con GUI son altamente reactivas, y tiene múltiples herramientas para diseñar interfaces estéticamente agradables.</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Los integrantes del equipo de trabajo no tienen experiencia previa con Vue.</li> <li>- Arquitectura: Vue es diseñado para aplicarse en el patrón arquitectónico Model-View-ViewModel (MVVM). Este es un patrón óptimo para el desarrollo de interfaces interactivas, pero se sale del requerimiento de implementar una arquitectura monolítica.</li> <li>- Memoria: Igual que React, Vue se ejecuta utilizando un Virtual DOM, lo que representa necesidades adicionales de memoria. También tiene desventajas de rendimiento, comparados con frameworks que compilan en tiempo de construcción.</li> </ul>
Angular	<ul style="list-style-type: none"> <li>- Código limpio: Angular es un framework MVC completo, basado en módulos, lo que también hace a un proyecto basado en Angular ideal para un ejercicio de código limpio.</li> <li>- Testing: Angular está desarrollado en TypeScript. Aunque se puede desarrollar el código de Angular en JavaScript, el uso de TypeScript favorece una aplicación de tipado más estricta que los frameworks de JavaScript puro. Esto implica un proceso más rápido y sencillo de testing.</li> <li>- Interfaz gráfica: Angular permite la construcción de interfaces gráficas estéticas, complejas y reactivas, lo que es un aspecto prioritario para el desarrollo de este proyecto.</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Entre los frameworks de frontend estudiados, Angular tiene de lejos la curva de aprendizaje más pronunciada. Ninguno de los integrantes del equipo tiene experiencia previa con Angular. Por tanto, el aprendizaje de los conceptos generales del framework, adicional a la adaptación a los requerimientos del proyecto tiene necesidades de capacitación adicional.</li> <li>- Memoria: Angular utiliza un DOM regular, que incluye requerimientos importantes de memoria en el aplicativo final. También significa que requiere actualizaciones frecuentes de renderización en una aplicación interactiva, por lo que también puede ser una desventaja en rendimiento en tiempo real.</li> <li>- Arquitectura: Angular fue desarrollado para proveer un framework optimizado para Single Page Applications (SPA), lo que dista del proyecto que estamos planteando. Se puede usar un empaquetador para desplegar la SPA en una aplicación de escritorio, pero no está diseñada con este objetivo. Por tanto, presenta desafíos adicionales en la implementación del proyecto para cumplir los requerimientos.</li> </ul>
Svelte	<ul style="list-style-type: none"> <li>- Curva de aprendizaje:</li> <li>- Memoria: Svelte fue diseñado para</li> </ul>	<ul style="list-style-type: none"> <li>- Conocimientos previos: Ninguno de los integrantes del equipo tiene experiencia</li> </ul>

	<p>evitar código repetido en la mayor medida posible, lo que permite construir aplicativos significativamente más ligeros que frameworks más estrictos, como lo puede ser React.</p> <ul style="list-style-type: none"> <li>- Rendimiento: Svelte se ejecuta por medio de código JavaScript altamente optimizado durante el proceso de construcción, lo que hace el tiempo de compilación y velocidad de ejecución superior a otros frameworks de frontend evaluados en este comparativo.</li> <li>- Flexibilidad: La sintaxis de Svelte es más flexible, lo que permite una mayor personalización en el diseño e implementación de la interfaz gráfica del proyecto.</li> </ul>	<p>previa con Svelte, aunque es uno de los frameworks más amigables para desarrolladores novatos.</p> <ul style="list-style-type: none"> <li>- Código limpio: Svelte es un framework mucho más cercano a JavaScript nativo que todos los frameworks mencionados anteriormente. Aunque esto le permite ser más sencillo a la hora de construir elementos del aplicativo, es mucho más desorganizado, lo que lo hace una opción bastante indeseable para desarrollo modular, implementando las prácticas deseadas de código limpio.</li> </ul>
Node	<ul style="list-style-type: none"> <li>- Concurrencia: Node utiliza un motor V8 de Chrome para la ejecución. Implemente un bucle de eventos, que le permite manejar los procesos demandados de manera concurrente, evitando cualquier tipo de bloqueo, ofreciendo una ejecución más eficiente y fiable.</li> <li>- Bases de datos: Node incluye su propio paquete de SQLite, el cual podemos usar para conectar el backend del aplicativo con una base de datos relacional de manera sencilla y confiable.</li> <li>- Aplicabilidad: Entre las funcionalidades backend de Node, se tiene la comunicación con el sistema de archivos del usuario, u otras funcionalidades del sistema operativo. Por lo tanto, es particularmente buena elección para backend de aplicaciones de escritorio.</li> </ul>	<ul style="list-style-type: none"> <li>- Rendimiento: Aunque Node está altamente optimizado para favorecer la concurrencia y asincronía, para hacer la ejecución en tiempo real más eficiente, y dada a la interactividad, desde su inepción, Node es monohilo. Esto siempre conlleva a limitaciones en la capacidad bruta de ejecución de sus aplicaciones.</li> <li>- Arquitectura: Node está diseñado como un framework de backend local, independiente de frameworks que lo llaman, como el frontend. Esto debe ser moldeado para la arquitectura monolítica de nuestro proyecto.</li> </ul>
Vite	<ul style="list-style-type: none"> <li>- Conocimientos previos: Uno de los integrantes del equipo tiene experiencia previa usando Vite, por lo que se pueden aplicar fácilmente las ventajas de empaquetamiento que ofrece en nuestro proyecto.</li> <li>- Rendimiento: Con Vite, el servidor de desarrollo implementado incluye pre-empaquetado de dependencias con Esbuild y mejoras sobre módulos ECMAScript nativos</li> <li>- Memoria: La construcción con Vite</li> </ul>	

	<p>ofrece mejoras significativas en el consumo de memoria de la aplicación final. Esto es por el uso de Rollup, una herramienta de empaquetación de código, que permite una construcción equivalente mucho más ligera.</p> <ul style="list-style-type: none"> <li>- Flexibilidad: Vite tiene compatibilidad con todos los frameworks de frontend, backend y empaquetamiento analizados en esta sección, por lo que su inclusión no representa un desafío en el acoplamiento adecuado de este framework.</li> <li>- Testing: El proceso de construcción ligero que permite Vite, hace el proceso de testing notablemente más rápido, especialmente en el testing de ejecución, ya que la ejecución del proyecto en producción es mucho más rápida con Vite. Por ejemplo, permite un arranque más rápido del servidor, permitiendo un testing de casos de prueba más veloz.</li> </ul>	
--	--	--

Para poder utilizar JavaScript como lenguaje de programación único 🖥️ para el proyecto, es necesario usar un empaquetador 📦, de manera que se puedan aplicar los frameworks, y conectarlos entre ellos, en una arquitectura monolítica 🏛️. Para la empaquetación, se considera la opción de usar Electron ⚡, NW.js y Tauri 🦀. Tauri, a pesar de que en la comparación parece ser fácilmente la mejor opción en términos de rendimiento ⚙️ y memoria 🧠, queda descartado para implementar en el proyecto. Esto es porque Tauri necesita Rust, lo que rompe con el requisito de usar sólo un lenguaje 🚫 en el proyecto. De manera similar, implementar NW.js tampoco se alinea perfectamente con los lineamientos del proyecto 📋, debido a que está diseñado para la implementación de APIs 🔗, lo cual tampoco es el objetivo.

Evalutando los frameworks de frontend 🎨, y comparando sus fortalezas y debilidades ⚖️, nos hemos decidido por construir el frontend en React ⚛️. El primer framework en ser descartado es Angular 🅒. La principal razón para no usar Angular en este proyecto es que, de los frameworks de frontend estudiados, Angular es fácilmente el más pesado 🐘, y no es necesariamente el más rápido 🚀. Además de esto, Angular fue diseñado para SPAs 🌐, y nuestro proyecto no es un SPA, por lo que sería necesario realizar múltiples adaptaciones 🔧 para implementar los requerimientos específicos. Adicional a esto, el equipo no tiene experiencia 👥 con Angular, lo que termina siendo más insidioso que eficiente, sin grandes ganancias en rendimiento.

Por razones parecidas, se decide no utilizar Vue 🍷 para el proyecto. Aunque Vue sí tiene herramientas para aplicativos de escritorio 🖥️, su arquitectura no es ideal para una arquitectura monolítica 🏛️, requiriendo sus propias adecuaciones. De los estudiados, Svelte ⚙️ parece ser la mejor opción en términos de memoria y rendimiento puro 📊, por lo que fue una de las opciones preliminares. La otra opción preliminar es React ⚛️, ya que el

equipo tiene conocimientos previos 📖 y presenta buenas herramientas para diseño estético 🎨 y reactivo ⚡, además de tener una arquitectura amigable para los requerimientos del proyecto.

El aspecto final, detractor de la implementación de Svelte, fue la facilidad de aplicar principios de código limpio 🧹, como los principios SOLID 🏗️. Aplicar estos principios es más fácil en React, que tiene una estructura modular 📁. En cambio, Svelte tiene una sintaxis más flexible 🖋️, pero no tan propicia para mantener código limpio y organizado 📏. Por tanto, elegimos usar React para el frontend de nuestro proyecto.

Finalmente, debido al uso de Electron ⚡ y React ⚛️, decidimos implementar el backend 🔄 del proyecto directamente con Node 🟢. Node presenta compatibilidad total 🔗 con Electron y React, por lo que no presenta dificultades adicionales en este aspecto. La lógica de negocio 📊 del aplicativo es sencilla, por lo que no se observan requerimientos adicionales que no puedan ser manejados por programas de Node, conectados e integrados en la arquitectura monolítica 🏛️.

Tanto React como Electron presentan desventajas en memoria 🧠 y rendimiento puro 📉, pero esto puede ser atenuado por el uso de Vite ⚡ como constructor del proyecto 🛠️. Usar Vite facilita el proceso de testing 🧪, adicional al hecho de que mejora el rendimiento 🚀, y no presenta incompatibilidades ⚙️ con los frameworks elegidos para el proyecto.

## Base de datos

La elección de SQLite 🗄️ como sistema de gestión de bases de datos 📊 para el proyecto ZFCocteles 🍹 es una decisión estratégica 📌 que se alinea perfectamente con la escala y los objetivos 🎯 del aplicativo. El proyecto está concebido como un recetario interactivo 📖⚡ y gratuito, enfocado en la gestión de cócteles 🍹 y la creación de contenido por parte del usuario 👤.

Dado que el sistema no gestionará transacciones de venta 💰 ni pedidos 📝 y la concurrencia de usuarios no será masiva 👤, la simplicidad y eficiencia ⚙️ de SQLite superan las ventajas de sistemas más robustos como MySQL 🐉 o PostgreSQL 🐘. SQLite opera sin un servidor dedicado 🖥️, almacenando toda la base de datos en un único archivo 📁, lo que simplifica enormemente la configuración ⚙️, el despliegue 🚀 y el mantenimiento 🛠️, un factor clave para un equipo de desarrollo pequeño 👥.

La naturaleza del proyecto 🍹, centrado en funcionalidades como un catálogo de cócteles 📖🍹, un creador de recetas 🖋️🍹 y guías de preparación 📋, no demanda la complejidad de un sistema cliente-servidor 🔗. La arquitectura de ZFCocteles 🏛️, que prioriza una experiencia de usuario intuitiva 🎨 y una rápida implementación de un Producto Mínimo Viable (MVP) ⚡, se beneficia directamente de la facilidad de uso de SQLite 🗄️.

Los requerimientos funcionales 📋, como filtrar cócteles por criterios 🔍🍹, guardar favoritos ⭐ y mostrar historiales 📈, pueden ser manejados eficientemente con las capacidades SQL estándar 💾 que SQLite ofrece, sin incurrir en la sobrecarga

administrativa 📁 que implica la gestión de servidores 🖥️, usuarios 👤 y permisos 🔒 en bases de datos como PostgreSQL o MySQL.














Además, la portabilidad 🚚 de una base de datos contenida en un solo archivo 📦 es ideal para el ciclo de vida planificado 📅 del proyecto. Con una versión inicial enfocada en navegadores web 🌐 y la posibilidad futura de expandirse a aplicaciones móviles 📱 si la demanda lo justifica, SQLite facilita la transición y el desarrollo en diferentes plataformas ⚙️.

Mientras que MySQL o PostgreSQL son soluciones excelentes para aplicaciones a gran escala 🏢 con altas cargas transaccionales ⚡ y concurrencia 🏃, para ZFCosteles 🍸 representaría una complejidad innecesaria 🚫. La escalabilidad 📈 del proyecto está definida como "controlada" 🛑, enfocándose primero en las funciones esenciales ✅. SQLite es más que suficiente para soportar el MVP 🚀 y las futuras iteraciones 🔄 previstas, como las herramientas interactivas 🔧 o la modificación de recetas 🍷, demostrando ser la opción más pragmática y eficiente 💡 para los requerimientos actuales del sistema 📊.

## Tecnologías utilizadas

Las tecnologías auxiliares ⚙️ que se van a implementar en el proyecto son:

- Better SQLite 3 📄: Esta es una base de datos embebida 📦, implementada a forma de módulo nativo 📦 en nuestro proyecto. Este módulo permite que la instalación e interacción 🔗 del proyecto con la base de datos sea bastante sencilla y rápida ⚡. La gran desventaja es que no funciona para bases de datos masivas 📈, o con una necesidad de actualización frecuente 🔄 por parte de los usuarios, pero estas no son prioridades para nuestra aplicación, por lo que la velocidad 🚀, portabilidad 🚚 y limpieza en su integración 🔧 son más importantes.
- Vitest 🪄: Herramienta auxiliar de Vite ⚡ para desarrollar y ejecutar tests 📝 de nuestra aplicación. Es una herramienta más reciente 🆕, por lo que puede presentar falencias en casos límite ⚠️, pero fue diseñada específicamente para proyectos que usen Vite, por lo que es la opción más lógica 📌 para apoyar la fase de testing.
- ESLint 🪄: Nuestro equipo ha decidido integrar un linter 🪄, y ha decidido usar ESLint para esta labor. Usar ESLint hace la labor de desarrollo mucho más productiva 💡, al permitir detección de errores en tiempo real ⌚ y enforzar prácticas de código limpio ✨.
- Prettier 🍷: Usamos Prettier integrado con ESLint para formatear el código de manera consistente y automática ⚙️. Al definir la configuración de ESLint y Prettier en las etapas iniciales 📄 del proyecto, apoyamos el uso de un estándar aplicado a todo el proyecto 🪄, sin requerir revisiones extensas 📋 del proceso de desarrollo de cada colaborador.
- concurrently ⚡: Con esta herramienta, no sólo podemos definir scripts complejos de ejecución 📄 como `npm run dev`, sino que además podemos configurarlos para correr en paralelo 🏃, y hacer el arranque o la ejecución de la aplicación más eficiente 🚀. No tiene herramientas avanzadas de orquestación 📁, pero para el tamaño del proyecto, el simple uso del paralelismo ✅ permite definir nuestros scripts de una manera más eficiente.

- cross-env : Esta herramienta ofrece un soporte sencillo para múltiples variables de entorno , lo que es particularmente importante para el proceso de testing  del proyecto y para probar la compatibilidad multiplataforma   del aplicativo.
- electron-rebuild : Dado que este es nuestro primer proyecto realizado en Electron , usar electron-rebuild facilita la construcción del proyecto . Este módulo se encarga de compilar los distintos módulos de Electron de forma nativa , facilitando la automatización  de las distintas herramientas de Electron, y liberando carga operativa  al equipo de trabajo. Dado que no se plantean modificaciones profundas  a sus módulos, electron-rebuild puede realizar este proceso sin mayores dificultades .