UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU EN ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Desenvolupament d'aplicacions d'escriptori amb Frameworks web i Tauri IPC

Autor: Eduard Graners i Castellet

Director: Dr. Eloi Puertas Prats
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 13 de juny de 2023

Abstract

English

This work focuses on the development of a file cleaner for MacOs systems using the Tauri framework.

The first part of the work aims to analyze the advantages of using Tauri to standardize user interfaces and inter-process communication. A comparison is made between Tauri IPC and the previous standard (Electron), highlighting that Tauri's IPC offers a different and more efficient approach to inter-process communication. It also emphasizes Tauri's ability to standardize user interfaces and adapt to different execution environments.

The history and evolution of technologies and methodologies for developing desktop applications will also be presented, starting with the libraries of classical languages, moving on to Electron and its conceptual similarities with Tauri, and ending with Tauri as a possible future standard for developing this type of applications.

The second part of the work focuses on the development of the cleaner and its implementation using Tauri. The most relevant aspects of the development, from planning to implementation and testing, are detailed. The work concludes that using Tauri is an efficient and innovative option for application development, not only for MacOs but also for other operating systems. It offers a new approach to inter-process communication and standardization of user interfaces.

Català

Aquest treball es centra en el desenvolupament d'un *cleaner* d'arxius per a sistemes MacOs mitjançant el *framework Tauri*.

La primera part del treball té com a objectiu analitzar els avantatges d'utilitzar *Tauri* per a estandaritzar les interfícies d'usuari i la comunicació inter-procés.
Es realitza una comparativa entre *Tauri IPC* i l'estàndard anterior (*Electron*), i es destaca que l'IPC de *Tauri* ofereix una aproximació diferent i més eficient per a la comunicació interprocés. Així mateix, es remarca la capacitat de *Tauri* per a estandaritzar les interfícies d'usuari, així com la seva adaptabilitat a diferents entorns d'execució.

Es mostrarà també la història i evolució de les tecnologíes i metodologies per desenvolupar aplicacions d'escriptori començant per les llibreries dels llenguatges clàssics, passant per *Electron* i les seves semblanços conceptuals amb *Tauri* i acabant amb aquest com a possible futur estàndard per desenvolupar aquest tipus d'aplicacions.

La segona part del treball es centra en el desenvolupament del **cleaner** i la seva implementació mitjançant *Tauri*. Es detallen els aspectes més rellevants del desenvolupament, des de la planificació fins a la implementació i les proves. El treball conclou que l'ús de *Tauri* és una opció eficient i innovadora per al desenvolupament

d'aplicacions tant per a MacOs com per altres sistemes operatius i que ofereix una nova aproximació per a la comunicació interprocés i per a l'estandardització de les interfícies d'usuari.

Resum

El projecte tracta del desenvolupament d'un cleaner d'arxius per a MacOs utilitzant el *framework Tauri*. Es presenta una anàlisi dels avantatges de *Tauri* en termes d'estandardització de les interfícies d'usuari i comunicació interprocés, destacant la seva eficiència i adaptabilitat. La implementació pràctica del cleaner abasta des de la concepció de la idea fins a la posada en producció, demostrant les habilitats i competències de l'enginyer informàtic.

A través d'aquest projecte, s'evidencia la capacitat de l'enginyer per aplicar els coneixements teòrics en un context pràctic i proporcionar una solució robusta i eficaç per a les necessitats dels usuaris. En conclusió, *Tauri* es desmarca com una opció eficient i innovadora per al desenvolupament d'aplicacions d'escriptori amb una aproximació diferent en termes d'estandardització i comunicació inter-procés.

Agraïments

Vull agrair als meus pares el seu suport incondicional. Al meu tutor, Dr. Eloi Puertas Prats, l'ajuda en la investigació i desenvolupament del treball, i a l'equip de desenvolupadors de *Tauri* la proximitat i ajuda brindada als usuaris del seu *framework*.

Índex

1 Introducció **1**

1.1 Context i motivació 1

1.2 Objectius 2

2 Estat dels cleaners per MacOs **3**

3 Gestió de tasques i planificació de la feina **3**

4 Història del desenvolupament d'interfícies **5**

4.1 Llibreries per al desenvolupament d'interficies d'usuari 5

4.2 Electron: Primera i actual solució 6

4.3 Tauri: Aproximació a un millor paradigma 11

5 Electron vs Tauri: Comparativa directa **21**

6 CleanMood: Concepció de la idea **23**

7 Estructura de l'aplicació **23**

7.1 Funcionalitats a implementar 24

7.2 Front-End 24

7.3 Back-End 35

8 Funcionament global de la apliació **38**

9 Funcionalitats futures **40**

10 Conclusions **41**

1 Introducció

1.1 Context i motivació

El desenvolupament d'aplicacions d'escriptori ha estat un àmbit clau de la informàtica des de l'arribada dels ordinadors personals a les cases del gran públic. Amb l'expansió d'Internet, les aplicacions distribuïdes i les aplicacions per a smartphones han pres el protagonisme de la informàtica d'usuari. Les empreses de software inverteixen gran part dels seus recursos en el desenvolupament d'aquestes aplicacions. Aquest canvi ha provocat una gran falta d'I+D per a la creació d'interfícies per als llenguatges clàssics.

Com és que *Python* no té una llibreria dedicada a desenvolupar interfícies d'usuari amb una estètica actualitzada a les tendències del moment? *Java* disposa de Swing, però no és un estàndard de la indústria ni té una curva d'aprenentatge per la qual valgui la pena invertir-hi temps. Aquests són només dos exemples, però aquest patró es repeteix en la resta de llenguatges.

La informàtica la podem separar en dos grups en base a dos conceptes senzills: el que veu l'usuari i el que no. Aquesta diferenciació és important per a la motivació d'aquest treball. La gran quantitat de tecnologíes disponibles per crear tot el que l'usuari no veu és beneficiosa per a la indústria, cada una compleix una funció o un conjunt de funcions. Algunes es solapen i algunes es complementen. Volem tractar un gran conjunt de dades? Podem fer servir *Python* amb les llibreries de Numpy i Pandas. Volem un ús molt precís de la memòria? Fem servir *C++*. Com aquests, una gran quantitat d'exemples. A l'usuari no li importa si la gestió de la memòria està optimitzada, almenys no en aquests paràmetres. L'usuari vol que l'aplicació compleixi les seves funcions i sigui ràpida, tot i que l'optimització i la velocitat estiguin directament relacionades.

Què veu l'usuari i, per tant, n'hem d'estandaritzar el seu desenvolupament? La interfície. Si tenim un conjunt estàndard d'eines per crear interfícies, que a més a causa de la seva naturalesa ens permetin partir des de la unitat més bàsica de representació gràfica, és a dir, el pixel, podrem crear qualsevol tipus d'interfície que compleixi els requeriments indicats per a cada situació.

Per sort, en l'àmbit del desenvolupament web, aquest paradigma es compleix. Disposem del 'big three' de tecnologíes de desenvolupament d'interfícies d'usuari: HTML per definir la estructura, CSS per a la estètica i *JavaScript* per al comportament.

La informàtica és per a l'usuari. Amb aquesta mentalitat vaig voler desenvolupar un dia el cleaner del qual tracta la part pràctica del treball. El primer en el que vaig pensar va ser: 'segur que *Python* té una llibreria per fer interfícies modernes'. Quan em vaig adonar que no era així, decebut, vaig començar a investigar. Amb què deuen estar programades les aplicacions de Discord o Slack? Les seves interfícies per a l'esciptori tenen una estètica idèntica a les seves versions per a la web.

Aqui vaig trobar l'actual estàndard de la indústria. El raonament de per què no fer servir el 'big three' per desenvolupar tots els tipus d'interfícies l'havien fet

també els desenvolupadors d'*Electron*.

Aquesta tecnologia s'explicarà amb més detall en el seu propi apartat. Per desenvolupar l'aplicació d'aquest treball s'ha fet servir *Tauri*, un *framework* que millora *Electron* en tots els sentits.

La motivació, per tant, d'aquest treball, és mostrar com ha estat, és i serà l'estàndard de la indústria per al desenvolupament d'interfícies d'usuari.

1.2 Objectius

- **Objectiu de la part teòrica:**
  1. Analitzar la situació actual del desenvolupament d'interfícies d'usuari en els llenguatges clàssics per a aplicacions d'escriptori.
  2. Exposar l'alternativa més popular, *Electron*, i destacar les seves característiques i limitacions.
  3. Justificar per què *Tauri* representa una millor alternativa per al desenvolupament d'aplicacions d'escriptori, amb avantatges significatius en termes d'estandardització, rendiment i eficiència.
- **Objectiu de la part pràctica:**
  1. Definir la idea i el procés de desenvolupament d'un *cleaner* d'arxius per a MacOs com a cas pràctic.
  2. Establir l'estructura de fitxers, components i classes del projecte, utilitzant les millors pràctiques de desenvolupament.
  3. Justificar l'elecció de React com a *framework* per a la implementació de l'aplicació cleaner, basant-se en els seus avantatges i la seva popularitat.
- **Objectiu final:**
  1. Crear una aplicació a manera de demostració que abasti totes les fases principals de desenvolupament, des de la concepció de la idea fins a l'estat previ a la posada en producció.
  2. Demostrar les habilitats i competències d'un enginyer informàtic en la implementació d'aquest projecte, utilitzant el *framework Tauri* per desenvolupar una aplicació ben estructurada.
  3. Exhibir una comprensió profunda de les funcionalitats, les característiques i els avantatges de *Tauri*, així com la capacitat per afrontar reptes tècnics i superar-los amb èxit al llarg de tot el procés de desenvolupament.
  4. A través d'aquesta aplicació, evidenciar la capacitat d'un enginyer informàtic per aplicar els coneixements teòrics en un context pràctic, resolent problemes reals i proporcionant una solució robusta i eficaç per a les necessitats dels usuaris.

2 Estat dels cleaners per MacOs

L'usuari, ja sigui expert o no, tendeix en molts casos a tenir el seu sistema d'arxius desordenat. Es tenen molts arxius repetits, sense utilitar, d'un tamany innecessari pel poc ús que sel's hi dona, etc.

Per tal de posar ordre a l'ordinador personal de l'usuari, van sorgir fa anys els *cleaners*, programes que busquen optimitzar el rendiment de l'ordinador i desfer-se d'arxius innecessaris. En aquest nínxol de mercat i en el cas concret de MacOs, trobem només un cleaner que sigui popular, estètic i funcional, CleanMyMacX, de la companyia MacPaw.

Aquest *cleaner* està desenvolupat amb Objective-C, *C++* i Swing. Aquest últim és d'ús exclusiu per a software d'apple. Si investiguem, veurem que moltes aplicacions desenvolupades per a sisemes MacOs exclusivament estan desenvolupades amb aquestes tecnologies. En el cas que MacPaw decidís fer una versió del seu software per a Windows o Linux, haurien de fer servir tecnologíes d'interfície diferents. És aquest factor limitant que impedeix masteritzar el desenvolupament d'aplicacions d'escriptori, ja que has d'estar molt enfocat a un sistema concret abans de desenvolupar. Fent servir *Tauri*, podrem programar una solució única i *cross-platform*.

Tauri com a *framework* és relativament nou. La primera **release** és del 2021, per tant, en un escenari optimista, ens trobem en un paradigma de pre-estandardització d'aquesta tecnología. Els motius pels quals MacPaw no va fer servir *Electron* per desenvolupar el seu **cleaner** només el coneixen ells, però pot ser que *Electron* consumeix molta memòria i produeix executables que ocupen molt espai en disc.

Fer servir *Tauri*, soluciona tots aquests problemes i és per això que aquest *framework* i el concepte que l'acompanya podrien convertir-se en l'estandard de la indústria.

3 Gestió de tasques i planificació de la feina

Investigació prèvia al desenvolupament

L'explicació feta a l'apartat d'introducció és fruit d'un procés d'investigació llarg i amb molt d'interès per trobar una solució per al desenvolupament d'interfícies d'usuari en aplicacions d'escriptori.

La idea de com desenvolupar la interfície d'aquest projecte ha passat per les mateixes tres etapes que l'explicació teòrica del perquè fer servir *Tauri*, és a dir, intentar desenvolupar-la amb una llibreria de *Python* com *Tkinter*, intentar desenvolupar-la amb *Electron* i al final descobrir *Tauri* i implementar-la amb aquest darrer *framework*.

Tot aquest procés d'investigació ha comportat pràcticament un any i mig, des dels inicis del 2021 fins a mitjan 2022.

Desenvolupament i Memòria

Un cop estudiat el *framework* i plantejats els objectius i l'estructura del projecte, ja s'ha pogut començar a desenvolupar.

El desenvolupament del projecte, com es pot veure a la Figura 1, ha anat des del febrer del 2023 fins al juny del mateix any

La memòria ha estat redactada en dues etapes: la part teòrica, des de l'abril de 2023 al maig de 2023 i la part pràctica, un cop ja desenvolupat el codi, ha estat redactada des de finals de maig del 2023 fins a principis de juny del mateix any.

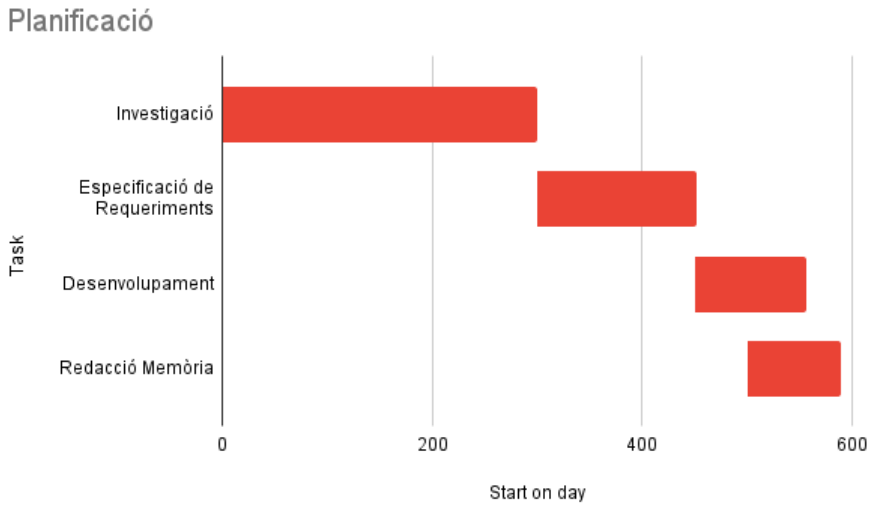

Figura 1: Planificació de la feina en funció dels dies des de la concepció de la idea

Metodologia del desenvolupament

Per desenvolupar l'aplicació, un cop finalitzada la investigació pertinent, s'ha fet servir l'estratègia *Kanban* per organitzar tasques i s'han utilitzat també conceptes de la metodología *Agile* per al compliment d'aquestes tasques.

El desenvolupament s'ha dividit en quatre *sprints* de dues setmanes, és a dir, períodes de temps de catorze dies en els quals s'han d'implementar funcionalitats.

Podem veure a la Figura 2 com s'han organitzat les tasques de desenvolupament. Al final de cada *sprint* hi ha les tasques completades en aquell període de temps. Podem veure també algunes tasques que no s'han completat a la columna de la dreta de la Figura 2.

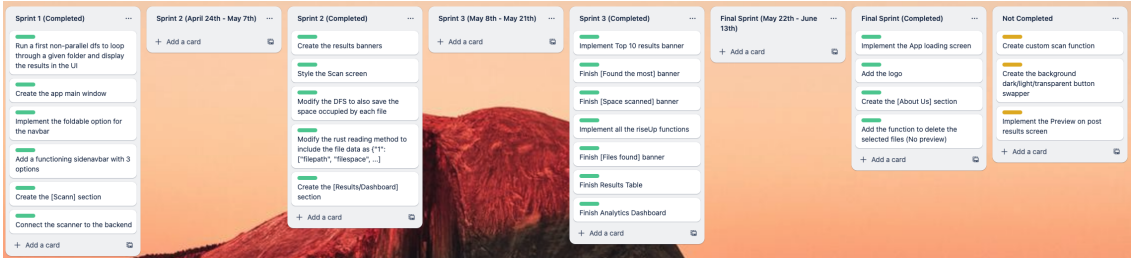

Figura 2: Kanban: Tasques realitzades/per realitzar

4 Història del desenvolupament d'interfícies

4.1 Llibreries per al desenvolupament d'interficies d'usuari

Per poder veure on anem, primer mirem d'on venim. Farem un repàs de les principals llibreries per desenvolupar UI de tres dels llenguatges més populars: *Python*, *Java* i *C++*

Python

Al famós llenguatge interpretat, *Python*, se li esperaria una llibreria actualitzada per desenvolupar inerfícies modernes. En el seu lloc trobem Tkinter, la llibreria més popular d'interfícies per a *Python*. Aprofundim l'èstètica que proporciona:

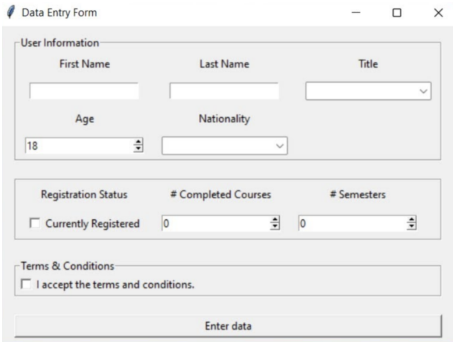

Figura 3: tkinter interface

Fent una ullada ràpida a la Figura 3, podem veure que la interfície té un *look-and-feel* molt semblant a Windows 98. La velocitat amb la qual les tendències estètiques canvien ha augmentat molt.
Les possibilitats que ofereix el 'big three' de tecnologíes web és àmpliament superior a aquesta estètica pre 2000. Resulta evident que el fet que *tkinter* no sigui millor no és per falta de capacitat tecnològica, sinó per falta d'interès de l'entorn de desenvolupament del llenguatge per dotar-lo de bones eines de construcció d'interfícies, simplement no és l'objectiu de *Python*.

Java

El cas de *Java* és molt semblant. La llibreria de referència és *Swing* i el *look-and-feel* que dona és el mateix. Queda clar el poc futur que tenen a nivell estètic les dues llibreries anteriors.

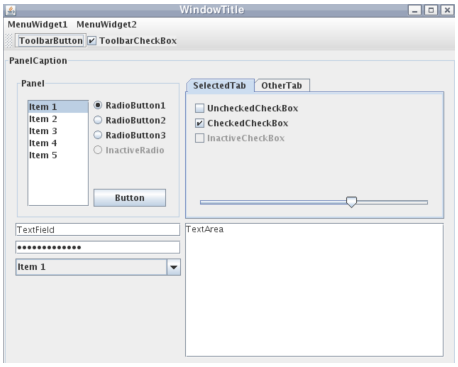

Figura 4: swing interface

C++

Centrant-nos en *C++*, *Qt* proporciona més eines per al desenvolupament d'aquestes interfícies. Es poden aconseguir millors resultats però la corba d'aprenentatge del *framework* (no del llenguatge) és molt elevada en comparació a la dels seus homòlegs i la recompensa és poder desenvolupar interfícies per un sol llenguatge. No és, per tant, un coneixement que ni per quota de mercat ni per recompensa, valgui la pena apendre.

4.2 Electron: Primera i actual solució

Introducció a Electron

Estudiem com funciona Electron:

Electron és un *framework* open-source molt popular per desenvolupar aplicacions d'escriptori *cross-platform* utilitzant el "big-three"de tecnologies web *HTML*, *CSS* i *JavaScript*. *Electron* permet als desenvolupadors crear aplicacions d'escriptori que poden funcionar en múltiples sistemes operatius com Windows, macOS i Linux, utilitzant una única base de codi.

Electron també proporciona funcionalitats per a la gestió de finestres i la seva navegació. Es poden crear finestres principals i finestres secundàries, gestionar esdeveniments i controlar-ne la visibilitat i el seu estat. Això permet crear aplicacions amb múltiples finestres i fluxos de treball complexos.

A causa de la naturalesa del *framework*, podem veure que per a algunes de les aplicacions d'escriptori més utilitzades aquesta ha estat la solució preferida ja que els ha permès reutilitzar de manera pràcticament idèntica la seva interfície d'usuari

que ja estava desenvolupada per al seu servei web. Algunes de les aplicacions més populars que en la seva versió d'escriptori fan servir *Electron*, són: Visual Studio Code, Slack, Discord, Skype i Microsoft Teams

Més enllà del seu principal atractiu, l'estandardització d'interficies, també s'ha de poder desenvolupar un back-end amb la suficient potència i adaptabilitat per gestionar grans volums de dades i processos. Aprofundim en el funcionament intern del *framework* tot repassant els seus principals components:

Main Window:

Com a finestra capaç de renderitzar continguts desenvolupats amb tecnologies web, *Electron* fa servir una instància de **chromium**

Chromium és un projecte *open-source* creat i mantingut per Google i és la base sobre la qual es construeix el navegador Google Chrome. És un motor de renderització de pàgines web que permet interpretar i mostrar continguts web d'una manera eficient i precisa. *Electron* fa servir una instància de Chromium com a motor de renderització per a la finestra principal de l'aplicació *(Main Window)*. Això significa que quan executem aplicacions com Visual Studio Code (VSCode) o Skype, en realitat estem veient informació renderitzada en una finestra de navegador.

Aprofitar Chromium com a motor de renderització per a les aplicacions d'*Electron* garanteix un gran rendiment i velocitat en la renderització del contingut, ja que Chromium està pensat i optimitzat per a aquesta funció específica.

Back-end technology:

Electron s'integra amb Node.js, un entorn d'execució de *JavaScript*. Aquesta integració permet als desenvolupadors utilitzar mòduls i API de Node.js a les seves aplicacions permetent accedir a funcionalitats del sistema operatiu i realitzar operacions del sistema de fitxers.
Per norma general, *JavaScript* no té llibreries que permetin l'accés al *filesystem*. Per motius de seguretat no es pot permetre que un llenguatge que s'executa al navegador de l'usuari i, per tant, al client, accedeixi al seu sistema sense el permís explícit de l'usuari. NodeJs en el context d'aplicacions distribuïdes, és a dir, en el seu principal context d'ús, serveix per desenvolupar servidors amb *JavaScript* i d'aquesta manera fer servir només un llenguatge de programació tant per a client com per a servidor. És una tecnología molt utilitzada en la implementació de microserveis per a aplicacions distribuïdes.

El fet que NodeJs permeti fer servir *JavaScript* com a llenguatge de Back-end és el que també permet que en aquest entorn, aquest llenguatge pugui comportar-se de forma semblant a com es comportaria un llenguatge clàssic.
Sabent això, amb *Electron* ja disposem de dos entorns on desenvolupar tant el Front-end com el Back-end i, per tant, crear una aplicació d'escriptori.

Main & Renderer processes:

La principal diferència entre l'ús de llibreries de desenvolupament d'interficies d'usuari dels lenguatges clàssics i l'ús de *frameworks* com *Electron* o *Tauri* és que en aquesta segona opció s'implementa l'aplicació de forma distribuïda, és a dir, hi ha dos processos independents que s'han de comunicar. Per norma general, els processos independents s'executen en diferents terminals. En aquest cas, això no és així. Amb *Electron* o *Tauri* s'implementen aplicacions distribuïdes en el sentit més estricte de la definició, però no es distribueixen.
En tractar-se d'aplicacions d'escriptori, els dos processos independents s'executaran al mateix terminal, és a dir, al ordinador de l'usuari.

Aquests processos són el **Renderer Process** i el **Main Process**. Referint-nos a conceptes típics del software distribuït, definim:

- **Renderer Process**: Encarregat de renderitzar a la finestra del navegador la interfície d'usuari, és a dir, el *Front-end*
- **Main Process**: encarregat del processament i emmagatzematge de dades, així com de la lògica i funcionament intern sense la interacció directa amb l'usuari final, és a dir, el *Back-end*

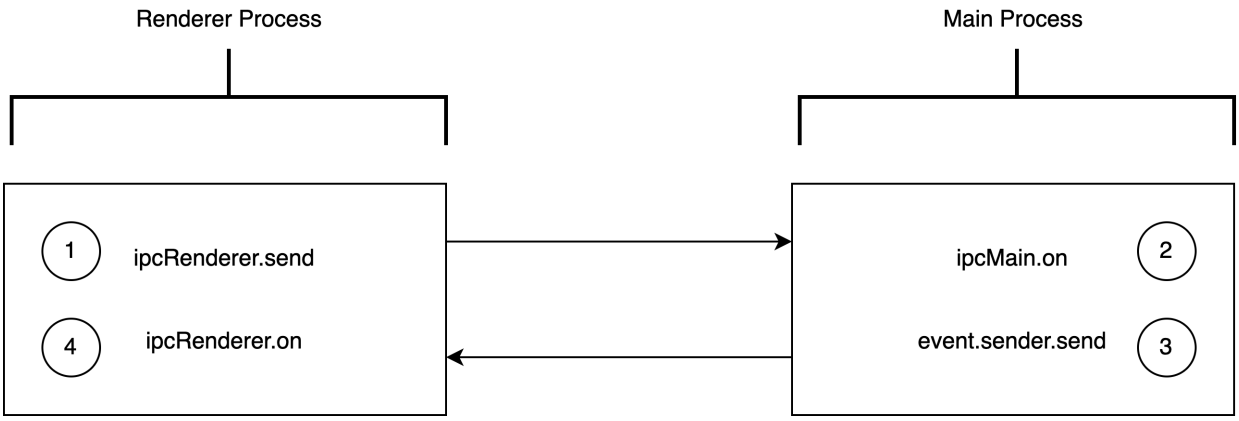

Figura 5: Electron IPC

Electron IPC: Comunicació Renderer - Main

Aprofundim sobre com funciona la comunicació IPC del diagrama anterior:

Per mostrar codi farem servir un exemple en el qual s'envia un simple string *Hello World!*
Des del **Renderer Process** fent servir la llibreria ipcRenderer enviem el missatge amb el mètode *send* de ipcRenderer.

1. **Serialization**: els arguments passats per paràmetre a la funció (1) *ipcRenderer.send()* es serialitzen típicament en format JSON per enviar-los a través de l'IPC.

```
const { ipcRenderer } = require('electron');

ipcRenderer.send('message', 'Hello from renderer!');
```

Figura 6: (1) - ipcRenderer: send

2. **Message Packaging**: les dades serialitzades, s'empaqueten en un objecte 'missatge'. Aquest objecte missatge conté la informació necessària perquè el **Main Process** identifiqui i gestioni el missatge.
   En el cas de missatges petits, s'especifica un canal de comunicació que ve donat pel Sistema Operatiu.

3. **Data Writting**: un cop les dades estan serialitzades i empaquetades, són escrites en un espai de memòria compartit (*shared memory*) al qual ambdós processos poden accedir.

4. **Data Reading**: El **Main Process** rep i llegeix el missatge de la memòria compartida i executa l'esdeveniment (2) *ipcMain.on*

5. **Deserialization**: El **Main Process** desserialitza el missatge i el torna al seu estat original.

6. **Main Process Response**: El **Main Process** després de processar el missatge rebut, pot enviar un missatge de confirmació-resposta al **Renderer Process** mitjançant la comanda (3) *event.sender.send*

```
const { ipcMain } = require('electron');

ipcMain.on('message', (event, message) => {
  console.log(message);

  event.sender.send('response', 'Hello from main!');
});
```

Figura 7: (2) - ipcMain: on & (3) - event.sender: send

7. **Renderer Process Handling**: Al rebre el missatge de resposta del **Main Process** el **Renderer Process**, pot reaccionar escoltant l'esdeveniment amb la comanda (4) *ipdRenderer.on*

```
const { ipcRenderer } = require('electron');

ipcRenderer.on('response', (event, message) => {
  console.log(message);
});
```

Figura 8: (4) - ipcRenderer: on

Packaging & Distribution

Amb les anteriors explicacions hem vist com *Electron* renderitza la interfície d'usuari, com gestiona el back-end i com es realitza la comunicació entre els seus dos principals processos. Posant en pràctica aquests coneixements, ja es pot desenvolupar una aplicació amb *Electron*. Falta, però, l'últim pas: la distribució del software. En trobar-nos en un paradigma d'aplicacions *cross-platform*, necessitarem una forma genèrica d'instal·lar i actualitzar l'aplicació.
Aquests són els passos a seguir per aconseguir-ho:

1. **Packaging preparation**: en tractar-se d'una aplicació en la qual el back-end està desenvolupat amb NodeJS, abans de realitzar el *packaging* ens hem d'assegurar que tots els mòduls i dependències de l'entorn NodeJs estan instal·lats.

2. **Package Building**: Completat el pas anterior, farem servir *Electron Forge* per generar els executables de cada sistema operatiu al qual vulguem distribuir l'aplicació.
   Fent servir un conjunt de comandes es llegeixen els diferents arxius de configuració orientats a cada sistema operatiu i es genera el corresponent executable. També, i de forma opcional, es pot especificar l'arquitectura del sistema sobre el qual s'executarà cada versió de l'aplicació.

3. **Distribution**: *Electron* dona l'opció de replicar el comportament de l'aplicació al ser publicada per poder fer *testing*. Un cop comprovat el correcte funcionament de tots els processos, la distribució final corre a càrrec del desenvolupador.

Conclusions: Electron

Hem repassat tots els principals components que formen una aplicació *Electron* i els passos necessaris per desenvoluparne una. És mostra suficient de la qualitat del *framework* el fet que Skype, una aplicació de més de 2.000 M d'usuaris, estigui desenvolupat en *Electron*?
En ser l'estàndard de la indústria, com a mínim garanteix que una aplicació de la magnitud de Skype funcioni correctament. Tot i això, el fet de tenir el domini

de la quota de mercat pot implicar una poca necessitat de millora i *Electron* té carències evidents, com l'alta consumició de recursos i memòria i el poc rendiment que provoca en les aplicacions.

4.3 Tauri: Aproximació a un millor paradigma

Introducció a Tauri:

Tauri i *Electron* comparteixen una característica fonamental: ambdós són *frameworks* orientats a complir la mateixa funció, la de crear aplicacions *cross-platform* d'escriptori desenvolupant la interfície d'usuari amb tecnologíes web.
Són les carències esmentades en l'apartat anterior les que *Tauri* ve a solucionar. Explicarem el framework i per acabar farem una comparativa directa d'avantatges i desavantatges de l'un i l'altre, on es conclourà que *Tauri* és superior.

Main Window:

El ser un framework *cross-platform* existeix la possibilitat d'adaptar segons quins aspectes del desenvolupament a cada sistema operatiu. Per al renderitzat de la *main window Tauri* pren dues aproximacions.

En el cas de sistemes Windows, com a *Electron*, *Tauri* fa servir **chromium** amb tots els avantatges que això implica, com ja s'ha explicat en l'apartat d'*Electron*. En el cas dels sistemes basats en Unix, és a dir, Linux i MacOs, *Tauri* fa servir com a *main window* el motor de renderització de pàgines web desenvolupat per Apple i de caràcter *open-source* **WebKit**.

La principal diferència entre WebKit i Chromium la trobem en el que abasten i l'enfocament de cada projecte. WebKit s'enfoca principalment en el motor de renderització que proporciona la tecnologia fonamental per mostrar contingut web. Proporciona un bon rendiment, una bona gestió de memòria i una renderització no només eficient sinó que d'una molt alta qualitat. Safari, el navegador d'Apple està basat en WebKit.

Chromium abasta una gamma més àmplia de funcionalitats més enllà del motor de renderització. Inclou components addicionals com el motor de *JavaScript* V8, el motor de renderització Blink (projecte que sorgeix d'una bifurcació de WebKit),la gestió del *stack* del **Model OSI**, funcions de seguretat i l'inspector.

Tot i que tant WebKit com Chromium són projectes de codi obert, tenen diferents orígens i comunitats de desenvolupament. WebKit va ser creat inicialment per Apple i el seu desenvolupament està impulsat en gran mesura pels enginyers i col·laboradors d'Apple. Chromium, iniciat per Google, té una comunitat més àmplia de col·laboradors, inclosos enginyers de diverses empreses, i serveix de base per a diversos navegadors, inclòs Google Chrome.

Podem concloure que ambdues tecnologíes són d'una gran qualitat. Això ho demostra el fet que tots els navegadors estan implementats fent-ne servir una o

l'altra. Chromium és més comlpet i WebKit està més orientat a una gran qualitat de renderitzat.

Back-end Technology:

En aquest apartat trobem la primera gran diferència de *Tauri* amb *Electron*. Al desenvolupar una aplicació amb *Tauri*, el *back-end* s'haurà de programar amb *Rust*. A *Electron* a l'hora de desenvolupar el *back-end*, per les tecnologíes que fa servir, es té la sensació que s'està desenvolupant un servidor que s'ha adaptat a fer de back-end d'una aplicació d'escriptori. Tècnicament ambdós *frameworks* treballen d'aquesta manera. En el fons s'està desenvolupant un client i un servidor, el fet diferencial és que rust no està només orientat a aquest aspecte com NodeJS i, per tant, atorga al desenvolupador més sensació de llibertat i control sobre el que es pot aconseguir.

Aprofundim en la naturalesa i característiques de *Rust*:
Rust és un llenguatge de programació compilat desenvolupat per Mozilla i amb la primera *release* de l'any 2010.
Des del seu llançament, any rere any ha augmentat en popularitat. Conegut com *'the most beloved language'* segons una enquesta realitzada a més de 80.000 enginyers.
A què es deu el seu èxit, tot i la dificil corba d'aprenentatge que té?
En destaquem els següents punts:

1. **Seguretat**: Rust posa un gran èmfasi en la seguretat del sistema i està orientat a prevenir errors de programació i gestió de memòria. Utilitza un sistema de tipat, de *borrowing* i d'*ownership* molt estrictes.
   Per defecte, una variable de *Rust* és inmutable! Per poder-la modificar s'ha d'especificar explícitament.

   - La variable sempre valdrà 10: `let x = 10;`
   - Afegint la paraula reservada `mut`, indiquem al compilador que la variable pot mutar el seu valor: `let mut x = 10;`

   Aquest tret característic de *rust* es suma a tot un conjunt d'altres característiques que orienten al llenguatge a gestionar la memòria de forma molt precisa. A consequència d'això, i com és típic dels llenguatges de programació, com més capacitat de gestió de memòria, més complexitat per aprendre el llenguatge.
   TOt i això, *Rust* s'ha consolidat com un dels llenguatges més utilitzats del món en només tretze anys de vida.

2. **Portabilitat**: Famós és l'eslogan de *Java 'Write once, run everywhere'* que abandera la portabilitat del llenguatge. En el cas concret de *Java* es deu a la JVM *Java Virtual Machine* que interpreta el codi i no cal recompilar-lo. Amb *rust* ens trobem en una situació semblant en referència que el codi escrit en aquest llenguatge es pot executar en qualsevol màquina. La diferència, però,

és que *rust* no necessita cap màquina virtual instal.lada en el sistema.
En executar la instrucció `rustc programm_name` el compilador crea un arxiu de codi màquina directament, i així s'estalvia problemes de compatibilitat
Aquesta adaptabilitat el fa un llenguatge ideal sobre el qual construir un framework com *Tauri*

3. **Concurrència**: La conjunció de *borrowing* i d'*ownership* són de gran ajuda al conjunt de llibreries i funcions natives del propi llenguatge per a la programació paral.lela. És l'accés restringit a segons quin tipus de dades i variables en funció del temps una de les característiques d'un model concurrent segur i fiable. La combinació d'aqusts factors amb l'ús de barreres tant implicites com explícites faciliten aquest tipus de programació en *rust*.

4. **Comunitat**: El tenir una acceptació tant gran entre els desenvolupadors es genera una situació en la qual la comunitat construeix, desenvolupa i documenta una gran quantitat de codi en *rust*.

Aquest conjunt de característiques donen una solvència *rust* com a llenguatge que ha fet que els desenvolupadors de *Tauri* l'escollissin com a llenguatge sobre el qual establir el seu framework.

Tauri Processes:

En el marc on ens trobem, el de fer servir aplicacions distribuïdes com a aplicacions d'escriptori, la comunicació entre els dos processos no es fa mitjançant HTTP requests. Amb *Electron* hem vist que es feia mitjançant l'escolta d'esdeveniments i compartint memòria entre processos. *Tauri* funciona d'una manera semblant i aprofundirem més en el seu funcionament, ja que, com que és un framework relativament nou, els propis desenvolupadors donen aquesta informació directament a qui la demani i, per tant, se'n pot obtenir de molt precisa.

Tauri com a tal té només un procés, el *Main Tauri Thread*, a partir d'ara el *MTT*, que seria l'equivalent al *Main Process* d'*Electron*. Per altra banda, l'encarregat de renderitzar és el *webview process*, el seu homòleg a *Electron* és el *Renderer Process*.

Ambdós processos actuen com a procés principal en el seu respectiu àmbit, i per tant poden tenir subprocessos càrrec seu. Observem un primer gràfic per il·lustrar aquesta explicació:

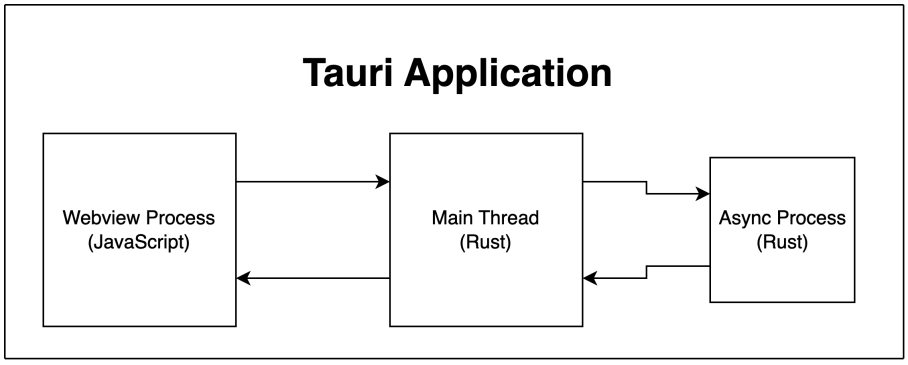

Figura 9: Processos inclosos en una aplicació Tauri

A l'esquerra de la Figura 7 Tenim el *Webview Process* que es comunica directament amb el *MTT* col.locat al centre de la figura. Totes les comunicacións entre *front-end* i *back-end* es tradueixen en comunciacions entre aquests dos processos. En el cas d'un procés asíncron, si és provocat pel *front-end*, se n'encarregara el propi *Webview Process* però si es tracta d'operacions del *back-end*, un subprocés del *MTT*, el *Async Process* és el que ho gestionarà. Cal remarcar que entre el *front-end* i el *async process* no hi ha comunicació directa i, per tant, haurà de passar sí o sí pel *MTT*.

Si aprofundim en el contingut de la Figura 9, podem veure una representació més exacta de l'estructura de processos a la Figura 10

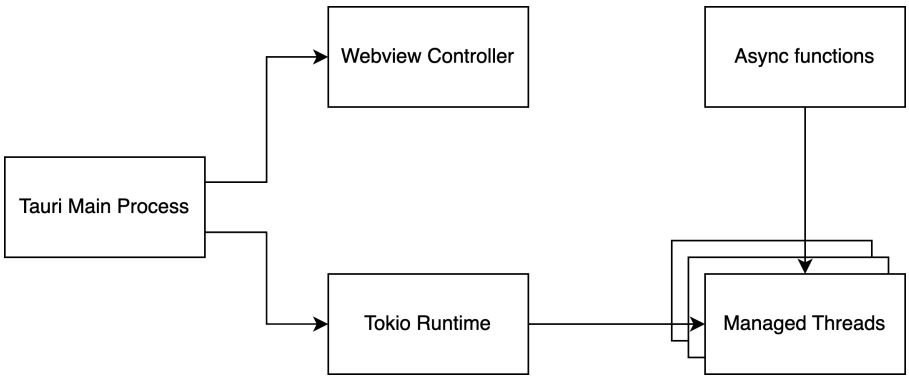

Figura 10: Detall dels processos d'una aplicació Tauri. Inclusió de Tokio Runtime

L'encarregat de gestionar els processos asíncrons de *rust* és el *Tokio Runtime*.

Suposem que volem enviar una ordre des del *front-end* que llegeixi un arxiu com un *log*, de forma asíncrona. Els *logs* poden tenir molta informació, ocupar molt d'espai a disc i per tant requerir un temps considerable per llegir-se, sobretot si es llegeixen des d'un disc dur mecànic lent o d'un dispositiu extern.

Si la lectura de fitxers de l'aplicació es realitzés de forma síncrona, obrir un fitxer gran podria bloquejar l'aplicació, i impediria l'usuari interactuar-hi fins que acabés l'operació. Si l'usuari intentés obrir diversos fitxers grans alhora, l'aplicació podria deixar de funcionar en saturar la memòria.

Aquí és on entra en joc Tokio, amb l'execució de tasques asíncrones.
Fent servir la lectura asíncrona, l'aplicació pot començar a llegir els fitxers de forma paral.lela i, immediatament després, cedir el control al *thread* principal. D'aquesta manera, l'aplicació seguirà rebent els inputs de l'usuari i, a mesura que es vagin completant les operacions de lectura dels fitxers, el contingut de cada fitxer es mostrarà a la interfície d'usuari si és necessari.
La figutra 9 representa la gestió de les tasques de *Tokio Runtime*. Tenim tres elements que interactuen entre si.

- **Tokio Runtime**: El*Tokio Tuntime* és l'encarregat de gestionar i de programar les *tasks*. S'encarrega també de gestionar els recursos de I/O i altres aspectes de baix nivell.

  La *runtime* utilitza la tècnica de *cooperative multitasking*, és a dir, les tasques s'encarreguen de cedir explícitament el control del temps d'execució del *Tokio runtime* si es troben operacions bloquejants.

- **Executor**: L'executor rep les tasques i decideix quan i on executar-les. Manté una cua amb les que es troben en espera d'execució i selecciona quina s'ha d'executar en funció de l'algorisme de planificació. Un cop seleccionada la següent *task* a executar, se li assignen els recursos del sistema, és a dir, el temps de CPU i memòria que siguin necessaris.

  És a l'*executor* a qui les tasques cedeixen el control durant l'execució si és necessari. D'aquesta manera l'*executor* pot donar prioritat a altres tasques preparades i per tant aprofitar al màxim els recursos disponibles.

- **Task**: Les operacions asíncrones a Tokio s'anomenen *tasks*. Una *task* és semblant a un *Thread* en el sentit que és una unitat independent d'execució, és a dir, cada *task* pot ser programada i executada de manera separada sense dependre de la resta. Les *tasks* són molt més lleugeres que els *threads*. Per tant, se'n poden executar moltes més.

  L'execució de *tasks* es separa per punts d'espera (*.await*). Quan es fa l'*.await* d'una operació asíncrona, la tasca cedeix el control de nou a *Tokio Runtime*

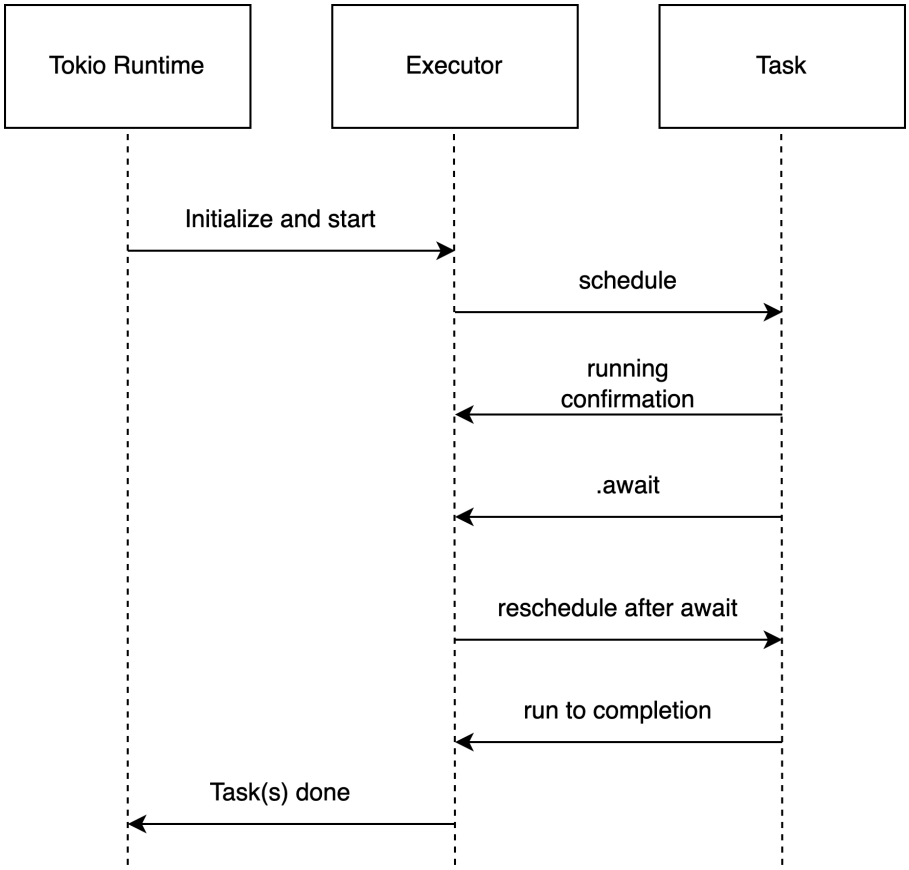

Figura 11: Diagrama de flux d'una task de tokio runtime

Mitjançant un exemple expliquem el diagrama de la Figura 11:

1. **Initialize and start** *(d'Executor a Tokio Runtime)*: el *runtime* inicia l'*executor*, i configura l'entorn per a l'execució de les tasques.

2. **Schedule** (de *Executor* a *Task*): L'*Executor* programa una tasca per a la seva execució. En aquest context, programar significa decidir quan i com s'executarà la tasca.

3. **Run** (de *Task* a *Executor*): La *task* està sent executada i li comunica a l'*executor* que es troba en estat de *running*.

4. **.await** (de *Task* a *Executor*): La *Task*, en trobarse un punt '.await' en el codi, cedeix voluntàriament el control de nou a l'*Executor*. Això acostuma a passar quan la tasca està esperant que es completi una operació asíncrona, com una lectura de fitxer o una petició de xarxa.

5. **Reschedule after await** (de *Executor* a *Task*): Després que una tasca hagi cedit el control a l'*Executor*, en algun moment la tornarà a programar per a la seva continuació de l'execució. Això passa una vegada que l'operació que s'estava esperant està a punt de progressar, com quan una lectura de fitxer està completa.

6. **Run to completion** (de *Task* a *Executor*): A l'executar-se la *task* se li comunica a l'*executor*.

7. **All tasks done** (de *Executor* a *Tokio Runtime*): un cop executades les tasques, l'*Executor* envia un missatge al *Tokio Runtime* amb aquesta informació.

Tauri IPC:

Per demostrar com funciona l'IPC de *Tauri*, aprofundim en el que s'ha explicat en l'apartat anterior, obviant la asincronia dels processos de Tokio Runtime i centrant-nos en la comunicació front-back. Fem servir un exemple semblant al que hem vist en el cas d' *Electron*.

1. **Calling invoke:** Al front-end, fem servir el mètode 'invoke' del mòdul '@tauri-apps/api/tauri' per cridar una funció de Rust. Imaginem que volem enviar un 'Hello World' que tenim guardat en una variable de tipus string al back-end de Rust. En aquest cas, passaríem per paràmetre el nom de la funció Rust que volíem cridar (per exemple, 'say_hello') com a string a 'invoke'.

```
const { invoke } = require('@tauri-apps/api/tauri')

function readDataFromRust() {
  invoke('send_hello_world')
    .then(message => {
      console.log(message)
    })
    .catch(error => console.error(error))
}
```

Figura 12: Crida a una funció Tauri

2. **Passing the Command to the Webview:** *Tauri* executa la interfície d'usuari com una aplicació web dins d'una webview, semblant a un navegador lleuger. Quan cridem 'invoke', *Tauri* envia un missatge a la webview amb el nom de la comanda Rust a executar, en aquest cas, 'say_hello'.

3. **Intercepting the Message:** La webview intercepta aquest missatge i el gestiona en conseqüència. Com que s'està invocant una comanda Rust, la webview reenvia el missatge al servidor Rust.

4. **Finding the Command:** El servidor Rust manté un registre de totes les comandes disponibles que han estat marcades amb l'atribut '[tauri::command]'. Utilitza el nom de la comanda proporcionat per la interfície d'usuari ('say_hello' en aquest cas) per trobar quina s'ha d'executar.

5. **Executing the Command:** una vegada trobada la funció 'say_hello', el servidor Rust l'executa. Si la comanda requereix qualsevol argument (en aquest cas, el nostre 'Hello World'), aquests hauran estat inclosos en el missatge original de la interfície d'usuari i per tant se li proporcionaran a la comanda.
   Cal resaltar que el *return* a rust pot ser implícit (l'última línia de la funció es considera com a línia de *return*, com en el cas de la Figura 11. La instrucció *.into()* que cridem sobre el *string* és característica de *Tauri* per indicar que aquella variable és la que s'ha d'enviar al *front-end*

```
#[tauri::command]
fn send_hello_world() -> String{
  "Hello, World!".into()
}
```

Figura 13: Funció de rust marcada com a comanda tauri

6. **Returning the Result:** El resultat de la comanda (que podria ser una confirmació que el 'Hello World' ha estat enviat amb èxit, per exemple) es serialitza a JSON i s'envia de tornada a la webview.

7. **Receiving the Result:** La webview rep la resposta JSON (la confirmació de l'èxit de la nostra comanda 'say_hello', en aquest cas) i la reenvia al codi *JavaScript* qui ha iniciat la crida.

8. **Promise Resolution:** El mètode 'invoke' en el codi de la interfície d'usuari inicialment retorna una *promise*, que ara es resol amb les dades rebudes de la comanda Rust. Si es produeix un error en qualsevol moment del procés, el resultat de la *promise* es rebutja i salta l'excepció de *JavaScript*.

Features orientats al desenvolupament multiplataforma

En ser *Tauri* un *framework* orientat a desenvolupar de manera única aplicacions d'escriptori que siguin *cross-platform*, no només és necessari la comunicació interprocés que hem explicat en els apartats anteriors, sinó que fan falta una sèrie de característiques que acabin de uniformitzar tot aquest entorn que s'està construint poc a poc.
La questió és també poder unificar en una sola manera de desenvolupar tot allò que per norma general requereix un tractament específic segons el sistema operatiu en el qual ens trobem. A continuació els *features* més importants que destaquen els propis desenvolupadors del seu *Framework*:

1. **Desktop Bundler:** Aquest *feature* permet que les aplicacions de *Tauri* tinguin un *bundle* conjunt per a tots els principals sistemes operatius, macOS, Windows i Linux. Això la converteix en una solució multiplataforma per als desenvolupadors, i elimina la necessitat de tenir un *bundle* específic per a cada plataforma.

2. **Self Updater:** Això permet que les aplicacions de *Tauri* puguin auto actualitzar-se, i significa que l'aplicació pot comprovar si hi ha actualitzacions i aplicar-les sense que l'usuari hagi de descarregar i instal.lar l'actualització manualment, proporcionant un procés d'actualització suau i no intrusiu.

3. **Core Plugin System:** El sistema de plugins permet als desenvolupadors construir *plugins* reutilitzables que poden ampliar la funcionalitat principal de *Tauri*. Això ofereix una gran flexibilitat, ja que els desenvolupadors poden crear noves funcionalitats o comportaments en forma de *plugins* que es poden reutilitzar en diferents aplicacions de *Tauri* i d'aquesta manera ajudar a augmentar la quantitat de recursos disponibles.

4. **Scoped Filesystem:** Aquesta és una característica de seguretat que assegura interaccions segures de fitxers. Garanteix que l'aplicació només pot interactuar amb fitxers dins del seu *scope* específic. D'aquesta manera s'impedeix qualsevol accés no intencionat o maliciós als fitxers del sistema. Aquest *feature* concret és molt util en cas que l'aplicació faci servir plugins *Tauri* que es connectin amb servidors externs. Com veurem més endavant, sí que es pot accedir al sistema d'arxius, però només amb el codi executat localment, no amb connexions externes sense permís.

5. **App Tray:** Aquesta *feature* proporciona la capacitat a *Tauri* d'aparèixer a la *task bar* de qualsevol sistema operatiu, i, per tant, establint l'últim punt de connexió de l'usuari amb l'aplicació i proporcionant-hi un accés ràpid i fàcil, és a dir, com en qualsevol altra aplicació.

6. **GitHub Action:** Aquesta *feature* permet la construcció automàtica del *binary* de *Tauri* per a macOS, Linux i Windows, utilitzant els sistemes *CI/CD* de GitHub. Això agilitza el procés de desenvolupament, fent-lo més eficient i menys propens a errors.

7. **Native Notifications:** *Tauri* ofereix notificacions per a tots els sistemes operatius fent servir una API. Aquesta funcionalitat permet la implementació de notificacions de forma relativament homogènia per a tots els sistemes.

8. **Sidecar:** Aquesta característica permet integrar i instrumentar altres binaris a l'aplicació. Permet que una aplicació *Tauri* es comuniqui amb altres processos, ampliant les seves capacitats més enllà d'aquelles proporcionades per les tecnologies web.

9. **App Storage:** *Tauri* proporciona una ubicació nativa per emmagatzemar dades d'usuari. Això facilita la gestió de la persistència de dades, ja que es

poden guardar i recuperar de manera fiable les dades de l'usuari, independentment de la plataforma en què s'executa la teva aplicació. Veurem més endavant que hi ha altres opcions per fer això, però aquesta és la recomanada pels desenvolupadors del *framework*

Estat actual i projeccions futures

Mentre *Electron* no està orientat a ser compatible amb més llenguatges, *Tauri* com a objectiu a llarg termini en té la intenció.
Per les raons exposades i per la gran adaptabilitat d'aquest objectiu, el *framework* amb més popularitat té possibilitats d'establir-se com a estàndard de la indústria per desenvolupar aplicacions d'escriptori.

És també destacable que *Tauri* setrà compatible amb el desenvolupament d'aplicacions per a *smartphones* i per *smartwatches*. Com a exemple tenim que a data de 12 de Juny del 2023 ja s'estan desenvolupant aplicacions en fase beta per a l'*apple watch*

Cal destacar la poca popularitat de *Tauri* envers el seu màxim competidor. A la Figura 14 podem observar el nombre de descàraregues de cada *framework* amb el sistema de gestió de paquets *npm*.

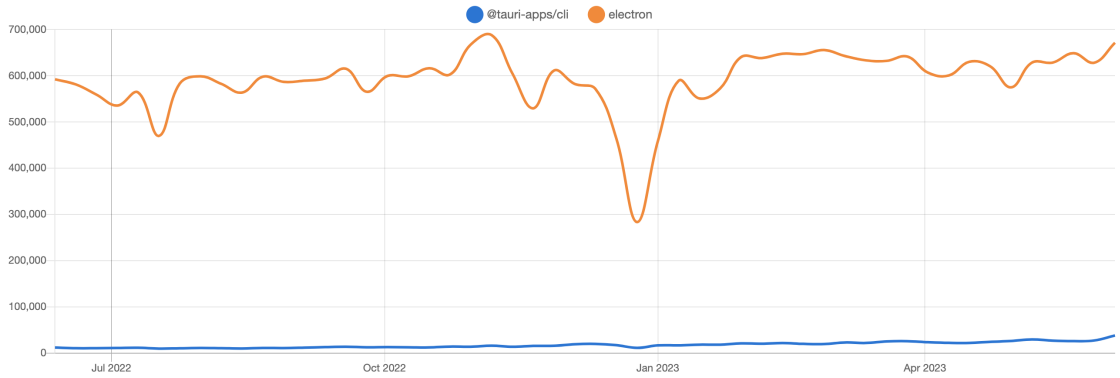

Figura 14: Comparativa de Tauri vs Electron en descàrreques l'ultim any

Els pròxims anys seràn fundamentals per al creixement de *Tauri* ja que com podem veure *Electron* segueix essent bastant més conegut. Tot i això aquest darrer *Framework* està creixent en popularitat i financiació.

5 Electron vs Tauri: Comparativa directa

Hem estudiat a fons el comportament dels dos *framework* en les seves respectives seccions. Un dels objectius d'aquest treball és el de demostrar perquè *Tauri* és el que hauria de ser l'alternativa a *Electron*, així com *Electron* ho era al les aplicacions d'escriptori tradicionals. Fem una comparativa més directa per tal d'aclarir les conclusions.

Feature	**Tauri**	**Electron**
Mida i Rendiment	Executables significativament més petits, la qual cosa resulta en temps d'inici més ràpids i un rendiment superior.	Executables més grans i ocupen molta memòria, la qual cosa porta a temps d'inici més lents i un rendiment inferior.
Seguretat	*Tauri* ofereix funcions de seguretat millorades de forma nativa i sense una gran implicació del desenvolupador.	*Electron* ofereix funcions de seguretat, però les aplicacions poden ser més susceptibles a vulnerabilitats de seguretat si no són configurades amb cura.
Compatibilitat cross-platform	Aplicacions resultants per a Windows, macOS i Linux a partir d'un sol codi base.	Aplicacions resultants per a Windows, macOS i Linux a partir d'un sol codi base.
Frameworks Web	Suporta qualsevol *framework* de frontend (React, Vue...) proporcionant més flexibilitat als desenvolupadors.	Es construeixen utilitzant HTML, CSS i *JavaScript*.
Self-updater i packaging	Funcionalitat de *self-update* incorporada i crea executables binaris més petits per a una distribució més fàcil.	Funcionalitat de *self-update* incorporada i crea executables binaris però els paquets resultants solen ser més grans.
Comunitat	*Tauri* és més nou i té una comunitat més petita però que creix ràpidament.	*Electron* ja disposa d'una comunitat gran i activa.
Adaptabilitat	Back-end en un llenguatge dedicat a aquesta funció.	Back-end programat amb un entorn d'un llenguatge ja existent.
Futur	Orientat a ser compatible amb més llenguatges.	No orientat a ser compatible amb més llenguatges.

Taula 1: Comparació de *Tauri* i *Electron*

Conclusions: Tauri vs Electron

Hem vist en els darrers apartats l'evolució de la programació d'interfícies d'usuari per a aplicacions d'escriptori. Hem vist les desactualitzades llibreries dels llenguatges clàssics, la primera aproximació i actual estàndard Electron, i una solució més novedosa i prometedora, *Tauri*.

Són observables les millores que representa cada nou paradigma sobre l'anterior i per tant la tendència del desenvolupament d'aquestes aplicacions hauria de ser aquesta, la de fer servir *frameworks* que renderitzin una webview i estiguin connectats amb un IPC al procés principal d'algun llenguatge.

Analitzant la quantiat d'interès, la voluntat d'interacció amb la comunitat i la maduresa dels dos *frameworks*, podem concloure que tant *Electron* com *Tauri* són opcions vàlides per al desenvolupament d'aplicacions d'escriptori amb interfícies d'usuari modernes.

Tot i la seva popularitat, *Electron* presenta certes limitacions. Encara que hagui estat utilitzat per a la creació d'aplicacions com Slack, Visual Studio Code i Discord, hi ha raons per considerar que no és prou bo. Tot i ser un estàndard per al desenvolupament d'aplicacions d'escriptori amb tecnologies web, *Electron* sovint pateix problemes de rendiment i eficiència, ja que requereix un consum elevat de recursos del sistema. A més, tot i la seva àmplia comunitat, la gestió dels recursos proporciona resultats poc eficients. Tot i que *Electron* pot oferir una estabilitat i rendiment satisfactoris en general, les seves limitacions i desavantatges fan que *Tauri* sigui una millor alternativa.

Tauri, amb la seva arquitectura, ofereix una solució més eficient en termes de gestió de recursos del sistema i velocitat d'execució de les aplicacions. Tot i que *Tauri* encara és un projecte relativament jove i en desenvolupament actiu, ja ha generat interès i entusiasme en la comunitat de desenvolupadors amb patrocinadors com **1Password** i **CloudFare**

En resum, el futur del desenvolupament d'aplicacions d'escriptori s'inclina cap a l'ús de *frameworks* basats en tecnologies web, i si a *Tauri* li arriba la popularitat que es mereix per les seves característiques, seria beneficiós per a la indústria.

6 CleanMood: Concepció de la idea

Amb els conceptes de *Tauri* estudiats podem procedir a explicar com ha estat estructurada i implementada l'aplicació *CleanMood* i ens servirà també com a *prove of concept* del que aquest *Framework* és capaç de fer.

Amb la intenció d'automatitzar algunes tasques, fa un parell d'anys vaig programar un petit *script* amb *bash* per esborrar tots els arxius zip acumulats a la carpeta de *downloads*, el pas següent em va portar a intentar aconseguir el mateix però podent triar el directori i el següent a executar un *script* que busqués per a tot el sistema d'arxius que comunament s'hi acomulessin i no fessin servei. Els arxius *.dmg* en són un bon exemple. A *macOs* en instal.lar una aplicació s'executa el arxiu instal.lador, és a dir, el *dmg*, i un cop ja instal.lada l'aplicació, ja no tenen utilitat. Tant és així, que moltes vegades a l'usuari se li dona la opció de suprimir-los un cop utilitzats, però no sempre.

Aquest és només un exemple de tots els altres casos que hi ha: directoris completament buits, arxius amb el nom modificat amb *(n)*, etc.

Fer totes les tasques d'administració i organització d'un ordinador personal pot consumir molt de temps a l'usuari i ser esgotador. Revisar un per un tots els arxius decidint si són prescindibles o no és una tasca poc agradable i feixuga.

És d'aquí d'on em va sorgir la idea de fer l'script i afegir-hi una interfície d'usuari agradable i senzilla d'utilitzar que per un costat permetés esborrar tots els arxius que amb una seguretat del 100% fossin prescindibles i al mateix temps donés a l'usuari la possibilitat d'analitzar un per un els arxius més 'dubtosos' d'una forma més eficient per tal de poder decidir si suprimir-los o no.

7 Estructura de l'aplicació

Dividim l'estructura del projecte en dues parts com si es tractés d'una aplicació web.
Per un costat tenim el *Front-End* que ha estat desenvolupat amb *React*, un *Framework* de *JavaScript* que permet la reutilització del que es coneixen com a components (trossos de codi) per tal d'aconseguir una renderització dinàmica del contingut a mostrar.

Per l'altre, tenim el *back-end* i, com s'ha explicat en apartats anteriors, de moment ha de ser amb *Rust*.

Per poder comunicar ambdues parts necessitarem el *Tauri IPC*

Els primers objectius s'han modificat i s'ha desenvolupat una versió de l'aplicació que ens permetrà fer una desmostració de les capacitats del *framework Tauri*.

L'objectiu és comprovar que es pot seguir tot el cicle de desenvolupament estàndard de les aplicacions d'escriptori aplicat a *Tauri* i, per tant, demostrar que és un substitut de les tecnologíes de desenvolupament d'interfícies de molta més garantia i qualitat. És per això que aquesta darrera part del treball esta centrada en aquest

procés de desenvolupament.

7.1 Funcionalitats a implementar

- Una interfície que es desmarqui a nivell estètic de les interficies tradicionals que tenen un *look-and-feel* desactualitzat.
- Opció de realitzar un escaneig a un directori d'un ordinador amb *macOs*.
- Donar la possibilitat a l'usuari d'esborrar els fitxers escanejats.
- Buscar sis tipus d'arxius concrets que acostumen a ser comuns a l'ordinador. També directoris buits

7.2 Front-End

En aquest apartat ens centrarem en el disseny, estructura i desenvolupament dels components *React*. Hem fet una petita introducció al funcionament de *React*. Aprofundim més en els seus principis i en el perquè és una eina de desenvolupament d'interfícies de molt alta qualitat.

React

1. **Components**: A *React*, la interfície d'usuari es divideix en unitats independents i reutilitzables conegudes com a components. Cada component té el seu propi estat i propietats *(state i props)* i pot controlar la seva pròpia renderització.
   Això proporciona un alt nivell de modularitat i facilita el manteniment i l'escalabilitat del codi. Un component pot contenir altres components, i es crea així una jerarquia que s'anomena *component tree*.
   Cada component es pot instanciar amb una sintaxi molt còmoda per al desenvolupador. Simplement es tractarà com un *tag* de html al qual, a més a més, se li podran passar paràmetres.

   ```
   <MyCustomComponent title={"MyComponent"}/>
   ```

2. **Single Page Application (SPA)**: React es fa servir principalment per construir aplicacions de pàgina única *(SPAs)*. En una SPA, tot el contingut de la pàgina es carrega inicialment i només es renderitzen les parts de la interfície d'usuari que son necessàries en cada moment.
   Això contrasta amb les aplicacions de múltiples pàgines, on cada pàgina requereix una sol·licitud de servidor completament nova. Les SPAs tendeixen a oferir una experiència d'usuari més suau i ràpida.
   En el nostre cas concret, això ens facilita molt el desenvolupament ja que podrem escollir o tenir només un punt de connexió entre *front-end* i *back-end*

que serà l'únic punt d'entrada de les dades a la interfície o tenir-ne múltiples en funció de les necessitats de l'aplicació. No ens caldrà que cap servidor renderitzi pàgines perquè ja estarà tot carregat, simplement s'haurà de fer dinàmic.

3. **Virtual DOM**: React fa ús del concepte de *Virtual DOM* per augmentar l'eficiència de les seves actualitzacions de la UI.
   Cada vegada que l'estat d'un component canvia, React crea una nova còpia del *Virtual DOM* i la compara amb la versió anterior, és a dir, compara l'estat actual de la UI amb l'estat que hauria de tenir després de cada actualització (variació en textos a mostrar, dades, components...)
   D'aquesta manera fent un les mínimes modificacions al *DOM*, *React* aconsegueix optimitzar el rendiment del renderitzat. Aquesta tècnica es coneix com a *reconciliation*.

4. **JSX**: JSX és un tipus de codi *JavaScript* fet servir per *React* per englobar *HTML* amb *JavaScript* i poder fer renderitzacions dinàmiques de *HTML*. En ser tant *HTML* com *XML marking-languages*, s'ha pres la primera inicial de *XML* per donar-li nom a JSX, però és només *HTML* preprocessat. React interpreta tot el codi escrit com a *JSX* i el converteix a *HTML* de forma dinàmica. D"aquesta manera podem generar *tags HTML* en funció de les de l'aplicació. Suposem que tenim una taula amb n entrades on n varia en funció de valors a la base de dades.
   Podríem fer una renderització dinàmica del contingut de la següent manera:

```
const fileData = Object.keys(scanResults).map((fileId, i) => {

    let currentFilePath = scanResults[fileId][0]
    let currentFileLocation = currentFilePath.substring(0, currentFilePath.lastIndexOf("/"))
    let currentFileName = currentFilePath.split("/").pop()

    return (
        <>
            <tr>
                <td className={`${i != numberOfFiles - 1 ? "rtd " : "rtd-lc"}`}>{currentFileLocation}</td>
                <td className={`${i != numberOfFiles - 1 ? "rtd " : "rtd-lc"}`}>{currentFileName}</td>
            </tr>
        </>
    )
})
```

Figura 15: Càlcul dinàmic del cada entrada d'una taula

En la Figura 15 podem veure un *code snippet* de l'aplicació que s'ha desenvolupat en aquest treball. Creem la constant *fileData* que fa referència a la informació obtinguda dels fitxers escanejats.
El més interessant és el *return*. Allà podem veure que sen's permet retornar el que sembla codi *HTML*, però realment és JSX, com hem dit, *HTML* preprocessat. Aquest return fa referència a una sola entrada de la taula, però si ens fixem en la primèra línia, podem veure que estem iterant un *array* amb la funció *map*.

El que estem aconseguint és que la variable *fileData* contingui n vegades el codi que trobem dins al *return*. A la Figura 16 podem veure com dins del que segueix sent *JSX*, però en un *scope* superior, hi ha una línia de codi que renderitza el contingut previament generat dinàmicament a la Figura 15.

```
<tbody className="results-table-body">
    {fileData}
</tbody>
```

Figura 16: Càlcul del nth-value de la taula

CleanMood Front-End

A mode de demostració de les capacitats del *framework Tauri* l'objectiu principal de l'aplicació és que l'usuari pugui escanejar un directori de l'ordinador i esborrar tots els fitxers trobats.

Les principals vistes i components de l'aplicació són els següents:

Scanner

En aquesta vista, l'usuari tindrà la opció de realitzar un escaneig ràpid sobre el sistema. Per poder realitzar la demostració, en prémer el botó de *Run Scan* a la secció *Quick Scan* es farà una cerca de sis tipus d'arxius concrets en un directori creat especialment per aquest propòsit.

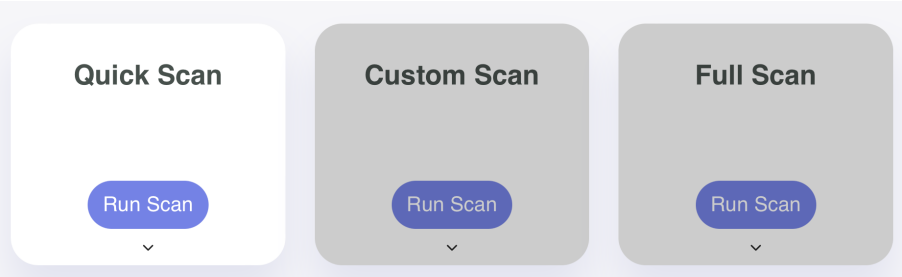

Figura 17: Vista de les opcions de scanner de la aplicació

Com podem veure a la Figura 17, tenim també dues altres opcions d'escaneig que estan reservades per a la versió "premium"de l'aplicació. Aquestes dues opcions s'inclouran dins de l'apartat de treballs futurs, però les funcions que tenen assignades són les següents:

- *Custom scan*: permetrà escanejar directoris concrets, buscar tipus d'arxius per extensió i la creació d'escaneigs recurrents i customitzats als quals l'usuari podrà accedir des de la barra superior del Mac.
- *Full scan*: permetrà a l'usuari buscar una àmplia quantitat d'arxius en tots els directoris on tingui intervenció directa, és a dir, qualsevol arxiu creat, modificat o susceptible d'haver estat creat per l'usuari serà subjecte de la cerca.

Results

A la secció de resultats, l'usuari disposarà de les seccions que veiem a la Figura 18.

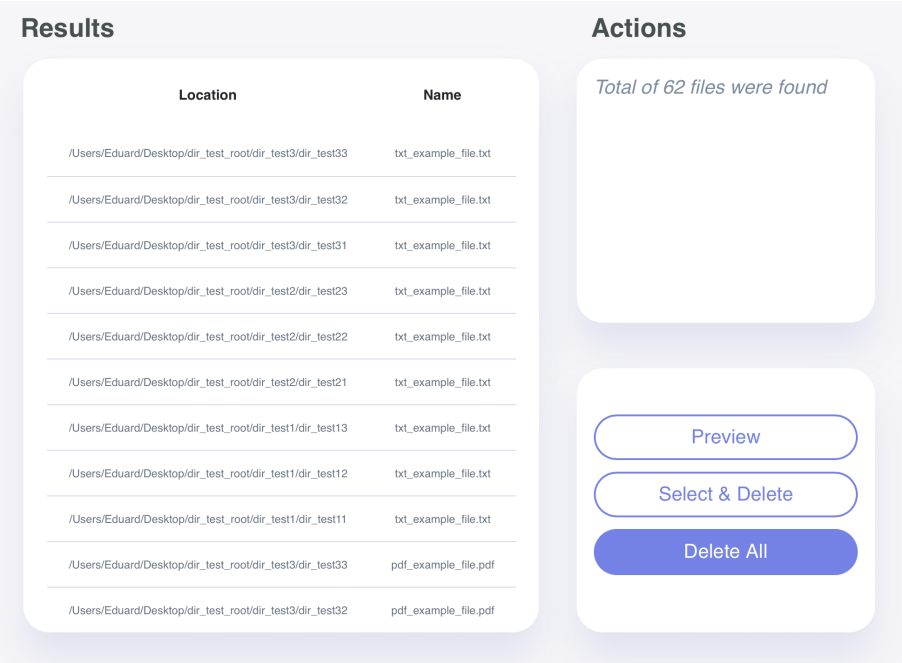

Figura 18: Vista dels resultats del Scanner

La secció de l'esquerra fa referència directament als resultats. Tots els arxius escanejats es mostraràn en aquesta taula perquè l'usuari en pugui coneixer la ruta i el seu nom.

A la secció de *Actions* podem veure dos *Cards*. La primera és informativa i en aquesta versió de l'aplicació mostra només la quantitat d'arxius trobats en un escaneig.

La segona *card* ens dona tres opcions.

1. Preview: Aquesta opció queda reservada per als usuaris ´premium' de l'aplicació. No s'ha desenvolupat i queda per a treballs futurs. La seva funcionalitat consistiria a permetre a l'usuari accedir a una nova vista en la qual podria previsualitzar un per un tots els arxius escanejats i seleccionar-los per tal d'esborrar-los després.

2. Select Delete: en el cas d'aquestà opció, es dona la mateixa situació que amb l'anterior. No s'ha desenvolupat i queda per a treballs futurs. La seva funcionalitat seria semblant a la de *Preview* amb la diferència que l'usuari no previsualitzaria els arxius. L'objectiu seria complir la mateixa funció que *Preview* però d'una forma més ràpida.

3. Delete All: Aquesta funcionalitat sí que s'ha implementat i queda per a la versió gratuïta de l'aplicació. El seu ús principal queda bastant clar: suprimir

tots els arxius escanejats. Realitzar aquesta acció és perillós, ja que es podrien perdre arxius que no es volguessin eliminar. És molt útil en casos en què l'usuari ha fet un *custom scan* d'una carpeta en concret i simplement vol repassar quins arxius s'han trobat a la taula de resultats per comprovar que tots son prescindibles per després procedir a esborrar-los tots.

La disponibilitat de les dades d'aquest apartat depèn si l'usuari ha realitzat un escaneig prèviament o no. En aquest segon escenari, l'usuari veurà una *cover screen* com la de la Figura 19 que el redirigirà a la vista de l' *Scanner*.

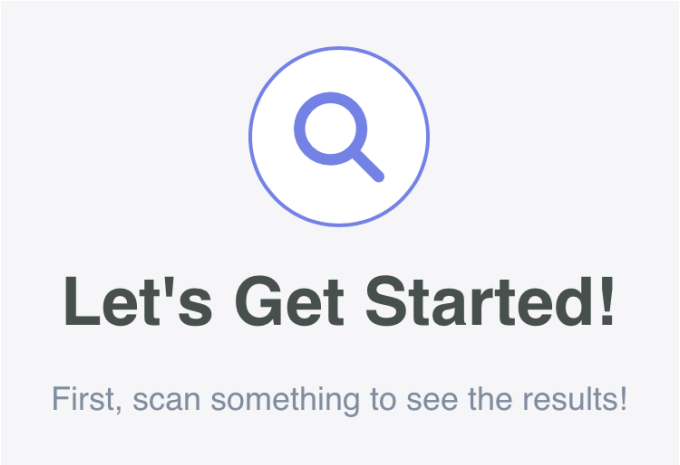

Figura 19: Cover Screen

Analytics

La vista de *Analytics* està pensada amb l'objectiu que l'usuari pugui veure quina és la seva tendència a l'hora d'acumular arxius. Per acnoseguir això la vista consta de cinc *cards* i les podem veure a la Figura 20.

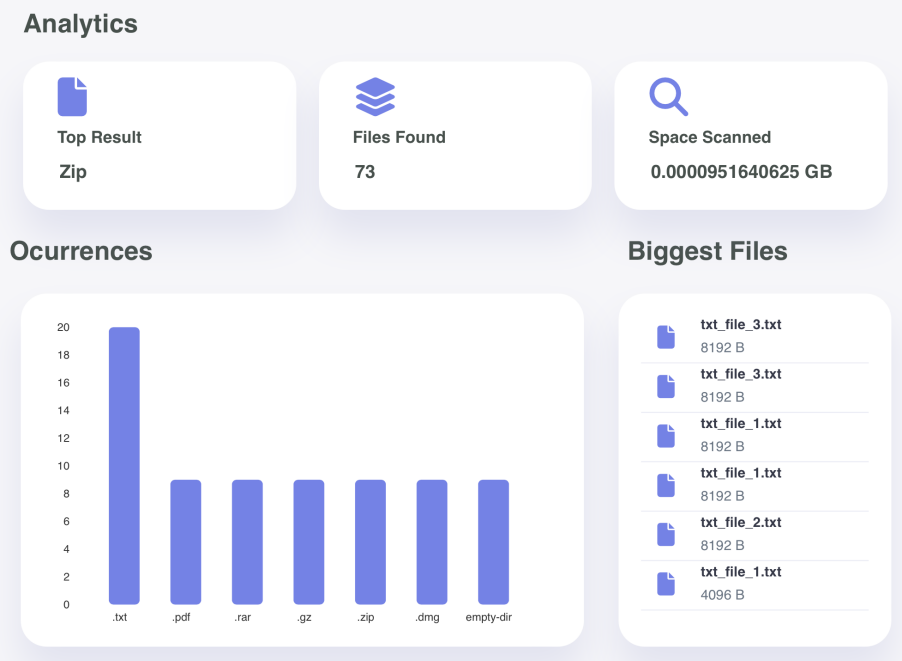

Figura 20: Vista de les estadístiques dels resultats del Scanner

1. Top Result: En aquesta primera *card* podem veure l'extensió més trobada en l'últim escaneig realitzat per l'usuari.

2. Files Found: En aquesta *card* podem veure la quantitat d'arxius trobats en un escaneig.

3. Space Scanned: El total d'espai que ocupen els arxius escanejats es mostra en GB en aquesta card.

4. Ocurrences: Aquí podem veure una gràfica de barres indicant la quantitat d'arxius trobats de cada extensió. Aquest gràfic és en si un component. A la Figura 24 es pot veure la relació *pare-fill* entre els components *Analytics* i *BarChart*.

5. Biggest Files: aquesta darrera *card* mostra a l'usuari els sis arxius que més espai ocupen, ordenats de major a menor. D'aquesta manera l'usuari pot alliberar una gran quantitat d'espai de forma ràpida només esborrant aquests arxius.

La disponibilitat de dades d'aquest apartat són dependents de dos factors. El primer és que s'hagui realitzat un escaneig en algun moment i el segon és que s'hagi accedit a la vista de resultats. En cas contrari l'usuari veurà una *cover screen* com la de la Figura 19, és a dir, veurà la mateixa *cover screen* que a l'apartat de *Results*.

Settings

Aquesta part de la vista queda per a treballs futurs. El seu ús principal seria el de permetre a l'usuari modificar paràmetres de permisivitat a l'hora d'accedir a segons quins directoris. Permetria costumitzar la secció de *full scan* que per defecte hagués accedit a tots els directoris on l'usuari acostuma a tenir interacció directa, però amb *settings* l'usuari podria especificar sobre quins altres directoris realitzar un escaneig, o sobre quins evitar-lo, i d'aquesta forma poder utilitzar el *full scan* com a escaneig general personalitzat.

En aquesta vista l'aplicació té un *paywall* indicant que no s'accedeix al seu ús sense una subscripció.

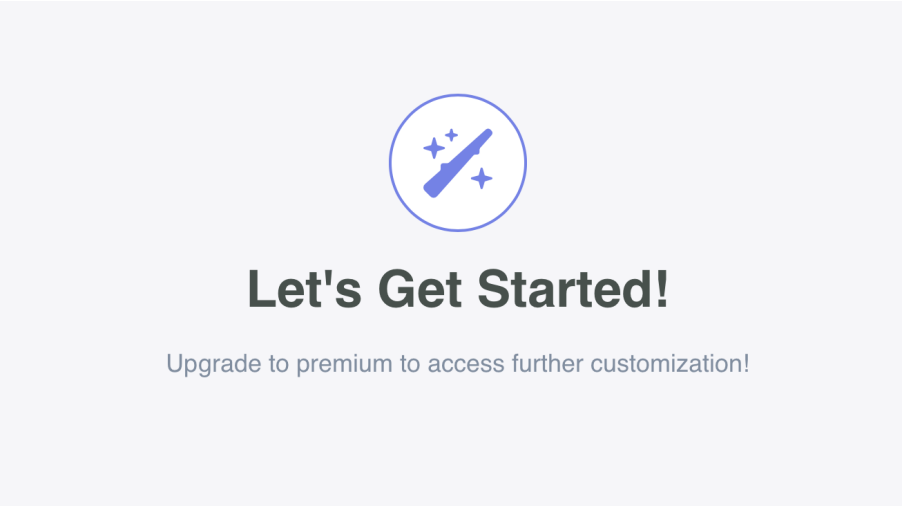

Figura 21: Paywall de l'apartat settings

About Us

En aquest apartat es mostra una vista que descriu el software i les seves funcionalitats i característiques. Es pretén donar un *look-and-feel* agradable on els components gràfics que hi intervenen tinguin *feedback*

Figura 22: Vista de la secció AboutUs

Side Navigation Bar

Per navegar per l'aplicació s'ha implementat una *SideNavBar* amb el mateix *look-and-feel* que la resta de l'aplicació. Així com a l'apartat de *AboutUs*, en aquest component s'ha donat importància al *feedback* que té l'usuari en interactuar-hi.

Un cop es selecciona un element de la barra lateral, mitjançant la transmisió de dades fill-pare que s'explicarà amb més profunditat al següent apartat, se li comunica al compoent *App.jsx* quina subsecció renderitzar.

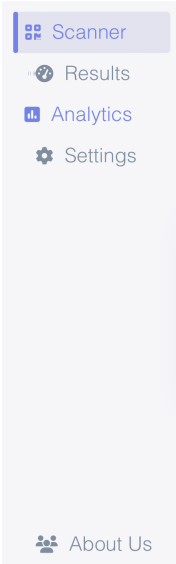

Figura 23: Barra lateral de la aplicació

CleanMood Component Tree

Hem vist les principals vistes de l'aplicació, aprofundim ara en el seu comportament com a components, en com es mouen, carreguen i modifiquen les dades.

A la Figura 24 podem veure les relacions entre tots els components del *Front-End* de l'aplicació.

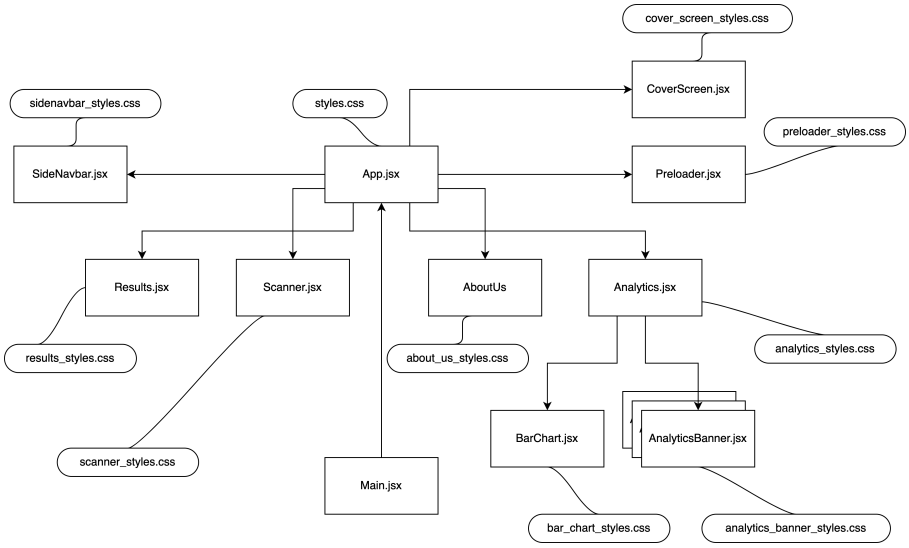

Figura 24: Diagrama de components del front-end de CleanMood

Podem veure com estan estructurats els components *React* del *Front-end*. Per entendre com funciona la transmisió de dades entre ells posarem un exemple.

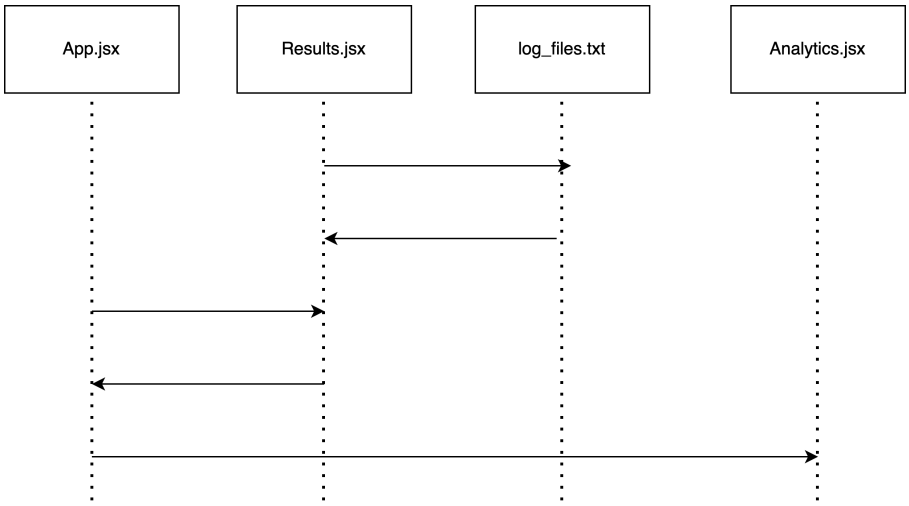

Figura 25: Diagrama de fluxe de dades entre Results i Analytics

Suposem que *Results.jsx* està llegint els resultats d'un escaneig relitzat previament. Aquesta lectura la farà dels arxius *log_files* d'on obtindrà en fromat *JSON* la informació dels *path* dels arxius i de l'espai que ocupen.

Aquesta informació ha d'estar a l'apartat d' *Analytics.jsx* però aquest component no fa cap crida a cap funció del *Back-end* i tampoc és fill de *Results.jsx*.

React no permet passar informació de fill a pare de forma convencional (per "props"), és a dir, passant arguments com si d'una funció es tractés. Per poder transmetre dades de fill a pare, hem d'assumir que totes les dades pertanyen al pare i crear en aquell component les variables d'estat i les corresponents funcions que les actualitzen. Fent això, el que podem fer és passar per paràmetre de pare a fill la propia funció que actualitza l'estat del pare per a que el fill la cridi quan sigui convenient.

En aquest projecte aquesta pràctica s'ha fet servir sovint. S'han creat les funcions *riseUp* que fan exactament el que acabem d'explicar, pujar dades de *Results.jsx* a *App.jsx* actualitzant l'estat d'aquest últim. D'aquesta manera, des de *App.jsx* ja podem enviar la informació al seu fill *Analytics.jsx*.

Repassant la Figura 25, veiem que *Results.jsx* llegeix les dades de *log_files*, veiem també que *App.jsx* realitza el procés d'actualització d'estat delegat a *Results.jsx* per obtenir les dades i després les transmet a *Analytics.jsx* per a que aquest component les mostri per pantalla.

7.3 Back-End

Totes les funcionalitats que impliquen crida a back-end, en el cas d'aquesta aplicació passaran pel main. Emmirallant-nos en el patró de disseny Model-Vista-Controlador *MVC*, podem assignar aquests rols a parts concretes del projecte.
El *Front-end* amb *JavaScript* i *React* és la vista, els mòduls de *Rust* són el model, i el *Main* actua com a controlador.
Si volem obtenir els resultats d'un escaneig d'un directori, el procediment a seguir implicarà:

1. Demanda de les dades des de la vista.
2. Execució del *tauri-command* associat.
3. Execució de comandes al model.
4. Retorn de les dades de model al main (controlador).
5. Retorn de les dades del main a la vista.

En versions anteriors de *Tauri*, al *package.json* s'havia d'especificar dins de la *allow list* quines funcions podien ser cridades des del *front-end*. En la versió actual només s'ha d'especificar al *main* i és al propi *main* on seran declarades i marcades com a *Tauri Command*.

Totes les funcions que realitzen operacions al back-end estan contingudes dins de mòduls i podem veure a la Figura 26 com estan estructurats aquests mòduls.

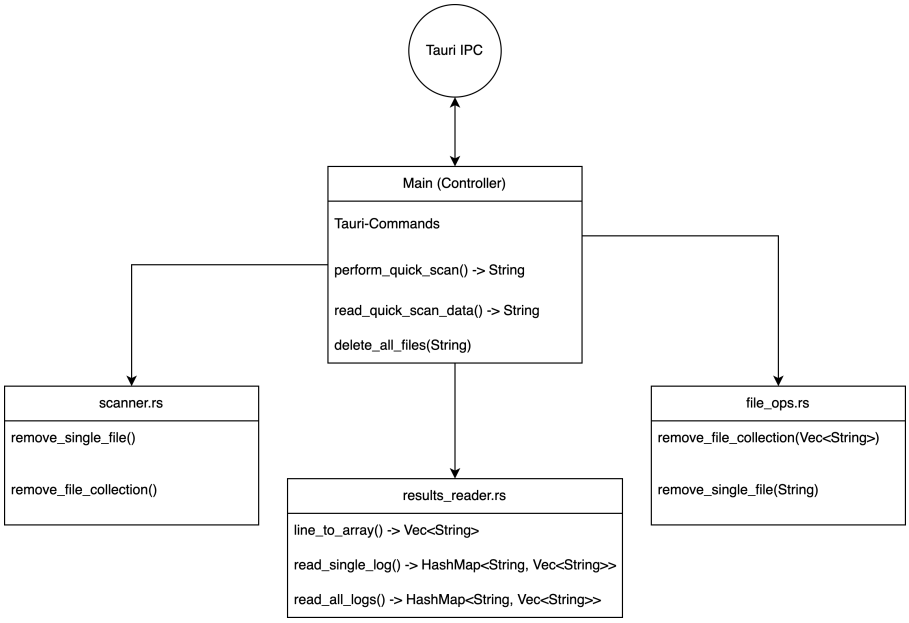

Figura 26: Diagrama de classes del back-end de CleanMood

Mòduls back-end

Per tal de coneixer una mica millor el funcionamenr del *back-end* aprofundim en la lògica de funcionament dels seus mòduls.

Scanner

Aquest mòdul és l'encarregat de gestionar tota la recerca d'arxius, d'organitzar les dades en el format apropiat i perquè puguin ser retornades al main per tal d'enviar-les posteriorment al *front-end*.

Per tal de demostrar que el procés de disseny i desenvolupament d'una aplicació no es veu limitat per culpa de *Tauri*, ens podem centrar en com s'ha fet la cerca d'arxius.

Tenim dues estratègies possibles per buscar els arxius demandats: podem fer servir eines integrades de bash per trobar arxius conctrets i d'aquesta manera mantenir-ne un control o podem fer servir un algorisme de recorregut de graphs per buscar exhaustivament el que vulguem.

Ambdues solucions tenen els seus avantatges i desavantatges. Si volem mantenir un control permanent sobre quins arxius tenim a l'ordinador i en quina jerarquia es troben, ens quedarem amb la segona opció. Si volem una cerca ràpida d'un arxiu concret, ens quedarem amb la primera.

En el cas d'aquesta aplicació, s'ha optat per implementar un algorisme *DFS* o *Depth-First-Search* per a buscar els arxius i d'aquesta manera demostrar que el *framework Tauri* no impedeix la modularitat del codi més enllà de la tasca que li correspon. *Electron* en aquest aspecte és més rígid.

L'algorisme s'ha implementat en un *script* de *bash* i per tal d'entende el funcionament del *scanner.rs* hem d'aprofundir en com es gestionen les dades en aquest *script*.

Depth-First-Search a Bash

El sistema de fitxers d'un sistema operatiu *Unix* està estructurat en forma d'arbre, i un arbre és un graph no cíclic. Per tant, l'algorisme per buscar exhaustivament nodes en aquest arbre ha de ser un algoritme de recorregut de graphs, és a dir, un *DFS* (ens podria servir també un *BFS*).

Serà el mòdul *scanner.rs* el que farà la crida al script *bash dfs_file_finder.sh*.

A la Figura 27 podem veure com es relaciona el mòdul *scanner.rs* amb l'execució de l'algorisme *DFS* i com a la vegada aquest escriu dades a fitxers d'on després hi llegirà un altre mòdul.

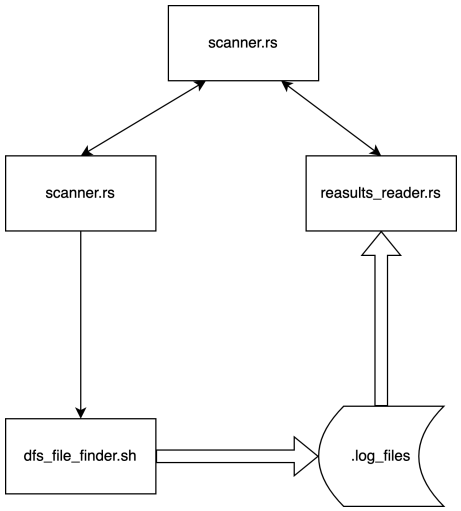

Figura 27: Gestió de crides i lectura/escriptura de dades a algorisme DFS

Com s'ha comentat en l'apartat de *Tauri*, el propi *Framework* ofereix un emmagatzament de dades canònic per poder tenir permanència de dades. Dit això, i per seguir demostrant la modularitat de *Tauri*, s'ha optat per una solució alternativa.

Aquesta implementació de l'algorisme busca tipus d'arxius concrets, com per exemple arxius *zip*, buscant i emmagatzemant el seu *path* absolut en format *string* en arxius *log* temàtics.

Els arxius *log* es troben dins d'una carpeta oculta anomenada *.log_files* situada al directori *root* del ordinador. Cada arxiu *log* contindrà els *path* de tots els arxius que comparteixin extensió.
L'arxiu *zip_log_file.txt* contindrà les dades dels *zip* trobats. Aquesta estructura es repeteix per a la resta d'extensions a buscar: *dmg, txt,* etc.

Results_reader

Com hem pogut veure en la Figura 27 i com podem deduir del seu nom, aquest mòdul s'encarregarà de llegir els *paths* dels arxius *log* que hauran estat prèviament escrits durant l'execució de l'algorisme *DFS*.

L'objectiu del *results_reader.rs* és també el d'estructurar les dades per tal de poder-les serialitzar i enviar al *front-end*

És per això que s'ha fet servir un *HashMap* per emmagatzemar la informació.

El *HashMap* es converteix a objecte i es serialitza per tal d'enviar-lo al *front-end*. Podem veure a la Figura 28, que dins de cada objecte tenim l'estructura *key:value* on la *key* és un identificador numèric i el *value* és un *array* que consta de dues posicions (incrementable amb implementacions futures). A la primera posició tenim el *path* de l'arxiu i a la segona la seva mida. D'aquesta manera podrem obtenir l'espai total escanejat i si és necessari l'espai que ocupa cada arxiu.

```
▼ Object
  ▶ O 0: ["/Users/Eduard/Desktop/dir_test_root/dir_test3/dir_test33/txt_file_1.txt", "4.0K"] (2)
  ▶ O 1: ["/Users/Eduard/Desktop/dir_test_root/dir_test3/dir_test33/txt_file_2.txt", "4.0K"] (2)
  ▶ O 2: ["/Users/Eduard/Desktop/dir_test_root/dir_test3/dir_test33/txt_file_3.txt", "4.0K"] (2)
  ▶ O 3: ["/Users/Eduard/Desktop/dir_test_root/dir_test3/dir_test32/txt_file_1.txt", "4.0K"] (2)
  ▶ O 4: ["/Users/Eduard/Desktop/dir_test_root/dir_test3/dir_test32/txt_file_2.txt", "4.0K"] (2)
```

Figura 28: Format de lectura de dades d'un escaneig

File_ops

Per tal de mantenir la estructura modular del *back-end*, s'ha implementat el *file_ops.rs*, que té com a objectiu rebre peticions del *front-end* (passant pel *main*) per realitzar les operacions sobre es fitxers escanejats.

Aquest mòdul llegeix dades del conjunt d'arxius *log* del directori *log_files* per executar les operacions d'esborrat d'arxius.

Agafant els *path* absoluts dels arxius a esborrar s'executa el mètode *remove_file_collection()* que a la vegada executa n cops el mètode *remove_single_file()* on n és el nombre d'arxius a esborrar.
L'esborrat dels arxius es realitza amb l'execució de la comanda *rm -f* en *bash* mitjançant la llibreria de rust per executar *bash*.

8 Funcionament global de la apliació

Una aplicació *Tauri*, com ja hem vist, consta de diferents processos i subprocessos, que correctament sincronitzats formen un tot.

Hem vist com funciona tant el *front-end* com el *back-end* i com es comuniquen entre ells.

Per demostrar el funcionament de l'aplicació de forma conjunta podem veure l'estructura de tota l'aplicació a la Figura 29.

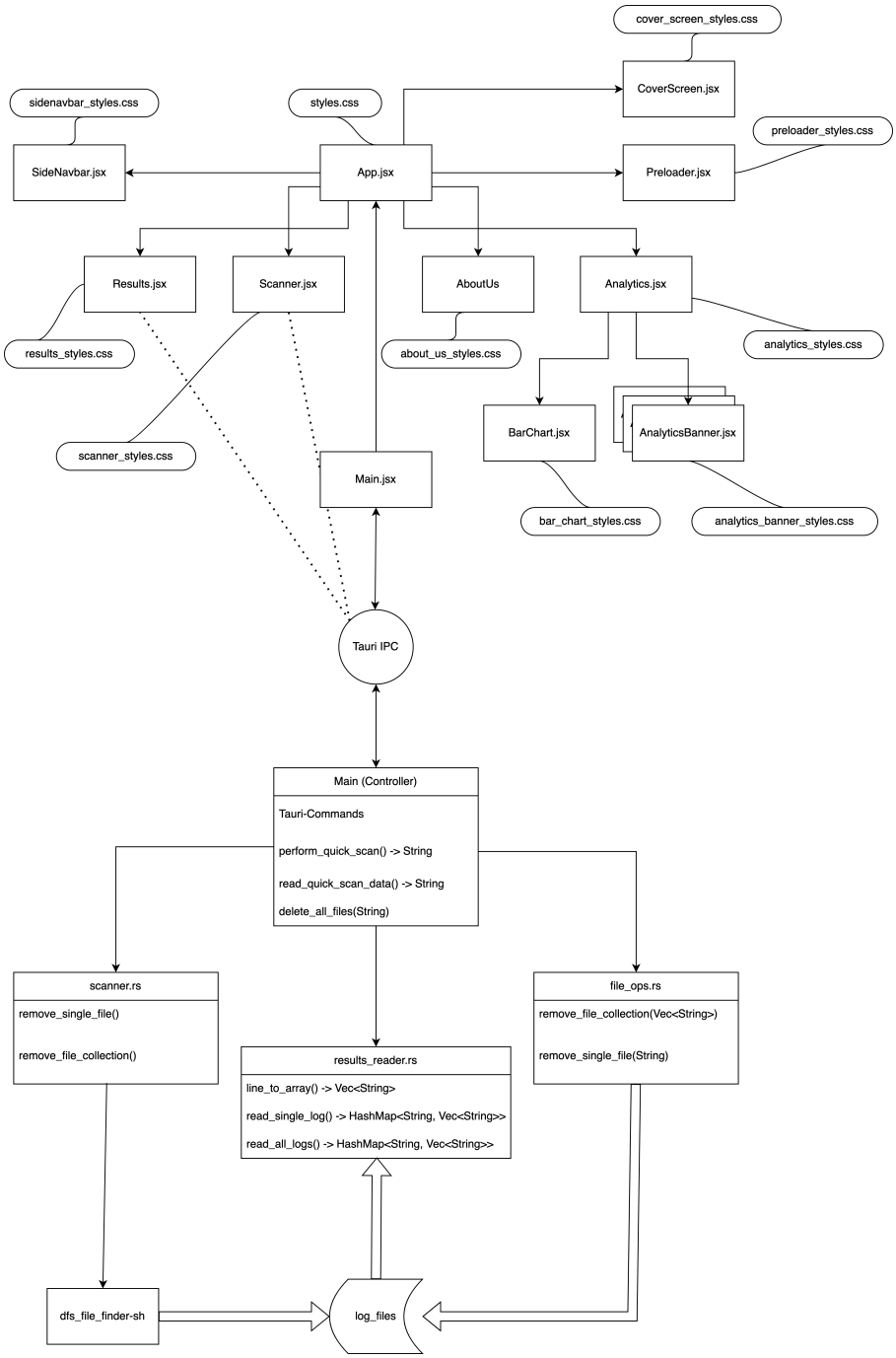

Figura 29: Estructura de mòduls, components i comunicació de tota la aplicació

Podem veure l'estructura de components del *front-end* i com es relacionen entre ells. Cada un dels components té associat un arxiu *css* per definir els seus estils. Per no complicar la visualització del diagrama, no s'han afegit les connexions entre l'arxiu *styles.css* i la resta de components. Aquest arxiu, no només esta associat al *App.jsx* ja que també hi podem trobar un conjunt de variables d'estil globals que

ens serveixen per al conjunt de l'aplicació, *primary-color*, *secondary-color*, *info-dark*, etc, que definieixen com han de ser els botons, *cards*, i *strings* per renderitzar.

En aquest diagrama, les connexions entre components indiquen pertinença, per exemple, el component *App.jsx* instància a *Results.jsx*, *Scanner.jsx*, etc.

En el cas concret d'aquests dos darrers components, ambdós mantenen connexions directes amb el *back-end* ja que un és l'encarregat de realitzar les cerces d'arxius i l'altre de llegir els resultats que aquest produeix, i totes aquestes connexions es fan sempre a través del *Tauri IPC*.

9 Funcionalitats futures

L'aplicació desenvolupada en aquest treball té una amplia quantitat de possibles *features* a implementar. Els objectius plantejats han estat reduïts amb la intenció de fer de l'aplicació una demostració de les capacitats de *Tauri*.
Anomenarem alguns dels *features* que en cas de continuar amb el projecte es podrien implementar i, per tant, d'aquesta manera completar apartats com el *custom scan* o el *full_scan* .

1. Increment del tipus d'arxius a buscar.
2. Possibilitat de cutomitzar la cerca d'arxius en funció d'extensió, directori, grup de directoris i/o grup d'extensions.
3. Oferir la possibilitat d'executar periòdicament accions customitzades, com per exemple esborrar el condingut de la paperera o esborrar correus d'spam en funció del remitent.
4. Buscar arxius potencialment maliciosos.
5. Filtrar per mida, antiguitat i ús de software instal.lat per tal d'esborrar-lo si fos necessari.

10 Conclusions

Després de realitzar una anàlisi profunda tant de les llibreries de desenvolupament d'interficies d'usuari de llenguatges clàssics com *Java* i *Python*, com del *framework Electron* i del *framework Tauri* es pot concloure:

1. Cada una de les aproximacions per desenvolupar interfícies i crear aplicacions d'escriptor és superior a l'anterior.

2. L'estandarització del desenvolupament de la part de la informàtica visible a l'usuari és positiva de cara a la masterització de la mateixa.

3. Desenvolupar aplicacions d'escriptori amb *Tauri* no té limitacions per implementar funcionalitats al back-end a causa de la modularitat del *framework*.

4. El procés de concepció d'una idea, especificació de requeriments, disseny d'aplicació, desenvolupament i distribució d'un producte és traslladable de llenguatges com *Java* i *Python* a l'entorn de desenvolupament que ofereix *Tauri*.

5. Les aplicacions d'escriptori tot i tenir menys popularitat tenen una quota de mercat fixa, el treball en local, segueix estant a l'ordre del dia i les aplicacions més populars ja estan essent desenvolupades amb *Electron*. Amb més finançament i popularitat *Tauri* podrà competir-hi.

6. El concepte d'utilitzar diferents processos per gestionar el *front-end* i el *back-end* d'una aplicació d'escriptori com si es tractés d'una *web application* resulta beneficiós per a la modularització de l'aplicació i el seu rendiment i ja s'ha convertit en l'estàndard de la indústria.

Referències

[1] Tauri: https://tauri.app/

[2] Electron: https://www.electronjs.org/es/

[3] Tkinter https://docs.python.org/es/3/library/tkinter.html

[4] React: https://es.react.dev/

[5] CleanMyMacX: https://cleanmymac.macpaw.com/

[6] Rust: https://www.rust-lang.org/es

[7] Code prettifyer https://carbon.now.sh/