Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Desenvolupament d'una aplicació de gestió de fitxers col·laborativa

---

Autor:
Marc Piñeiro Selas

Tutors:
Josep Soler

MEMÒRIA

Convocatòria:
Setembre 2025

Departament :
INFORMÀTICA, MATEMÀTICA APLICADA I ESTADÍSTICA

Projecte:	Projecte Final de Grau
Document:	Memòria
Títol:	Desenvolupament d'una aplicació de gestió de fitxers col·laborativa
Autor:	Marc Piñeiro Selas
Data:	Setembre 2025

Estudi:
Grau en Enginyeria Informàtica
Universitat de Girona

Supervisor 1:
Josep Soler
Universitat de Girona
Email: josep.soler@udg.edu
Web: Link

Índex

1 Introducció, motivacions, propósits i objectius **1**
1.1 Context del projecte 1
1.2 Motivacions 1
1.3 Propósit del projecte 2
1.4 Objectius generals 2
1.5 Objectius específics 2
1.6 Situació inicial i públic objectiu 3

2 Estudi de viabilitat **4**
2.1 Viabilitat tècnica 4
2.2 Viabilitat econòmica 5
2.3 Viabilitat humana 5
2.4 Viabilitat legal 6
2.5 Conclusió 6

3 Metodologia seguida **7**
3.1 Marc teòric: voluntat d'un enfocament àgil 7
3.2 Seguiment i control 7
3.3 Definition of Done: l'únic criteri formal 7
3.4 Fases del projecte: del pla a la realitat 8
3.5 Desglossament i revisió de les unitats de treball 9
3.6 Feedback dels testers 9
3.7 Dificultats de gestió i conciliació 9
3.8 Gestió de riscos i lliçons apreses 9
3.9 Lliçons apreses 9

4 Planificació **10**
4.1 Pla original 10
4.2 Execució real i validacions 10
4.3 Pla vs. execució: la bretxa quantificada 13
4.4 Causes de les desviacions 13
4.5 Accions correctores aplicades 13
4.6 Estimació d'hores dedicades 13
4.7 Lliçons finals 14

5 Marc de treball i conceptes previs **15**
5.1 Introducció 15
5.2 Entorn de desenvolupament 15
5.3 Eines de desenvolupament 15
5.4 Frontend web 17
5.5 Backend i microserveis 22

5.6 Client d'escriptori . . . 27
5.7 Utilitats addicionals . . . 28
5.8 Versionat de dependències . . . 29

6 Requisits del sistema **30**
6.1 Introducció . . . 30
6.2 Requisits funcionals . . . 30
6.2.1 1. Gestió d'usuaris . . . 30
6.2.2 2. Gestió d'arxius . . . 30
6.2.3 3. Sistema de paperera . . . 31
6.2.4 4. Compartició d'arxius . . . 31
6.2.5 5. Sincronització en temps real . . . 31
6.2.6 6. Panell d'administració . . . 31
6.3 Requisits no funcionals . . . 31
6.3.1 1. Rendiment . . . 31
6.3.2 2. Seguretat . . . 32
6.3.3 3. Usabilitat . . . 32
6.3.4 4. Mantenibilitat . . . 32
6.3.5 5. Compatibilitat . . . 32
6.3.6 6. Escalabilitat . . . 32
6.3.7 7. Portabilitat . . . 33
6.4 Requisits de maquinari i programari . . . 33
6.5 Resum . . . 33

7 Estudis i decisions **34**
7.1 Introducció . . . 34
7.2 Pila Tecnològica del Backend . . . 34
7.2.1 Spring Boot . . . 35
7.2.2 Spring Data JPA . . . 36
7.2.3 PostgreSQL . . . 36
7.2.4 Ecosistema Spring Cloud . . . 37
Spring Cloud Gateway . . . 37
Eureka . . . 38
Spring Cloud OpenFeign . . . 38
7.2.5 Spring Security . . . 39
7.2.6 RabbitMQ . . . 39

7.3 Pila Tecnològica del Frontend Web 40
7.3.1 React 40
7.3.2 React Query (TanStack Query) 42
7.3.3 Zustand 42
7.4 Pila Tecnològica del Client d'Escriptori 42
7.4.1 Tauri i Svelte 42
7.5 Maquinari i infraestructura 43
7.6 Eines de disseny i estil (UI/UX) 44
7.7 Cadena d'eines i DevOps 44
7.8 Llicències i compliment legal 45
7.9 Traçabilitat global amb els requisits 45

8 Anàlisi i disseny del sistema **47**
8.1 Introducció 47
8.2 Anàlisi funcional 47
8.2.1 Diagrama de casos d'ús 47
8.2.2 Actors i mòduls 49
8.2.3 Casos d'ús principals 49
8.2.4 Diagrames d'activitat dels Casos d'Ús Principals 53
UC-01: Registrar-se 53
UC-02: Iniciar sessió 54
UC-08: Crear o pujar arxius 55
UC-10: Descarregar 55
UC-11: Enviar a la paperera 56
UC-12: Eliminar permanentment 57
UC-13: Compartir arxius 57
8.2.5 Matriu de traçabilitat entre requisits i casos d'ús 58
8.3 Arquitectura del sistema 60
8.3.1 Visió general i components 60
8.3.2 Disseny dels microserveis 62
UserAuthentication 62
UserManagement 63
FileManagement 63
FileAccessControl 65
FileSharing 66
Trash 67
SyncService 67
8.3.3 Patrons de disseny i principis arquitectònics 68
8.3.4 Justificació de l'arquitectura de microserveis 70
8.4 Disseny de les interfícies d'usuari 71
8.4.1 Client web 71
Autenticació 71
Escriptori principal 72
8.4.2 Client d'escriptori 78

8.5 Consideracions de seguretat . . . 82
8.5.1 Control d'accés basat en rols (RBAC) . . . 83
8.5.2 Mecanismes de protecció . . . 83
8.5.3 Justificació del compliment del RGPD . . . 84
8.6 Cobertura dels requisits no funcionals . . . 84
8.7 Conclusions . . . 85

9 Implementació i proves **86**
9.1 Visió general de la implementació . . . 86
9.2 Entorn de desenvolupament i desplegament . . . 87
9.3 Implementació del backend . . . 88
9.3.1 Gateway i filtre de JWT . . . 88
9.3.2 Gestió d'usuaris i autenticació . . . 91
UC-01: Registrar-se . . . 92
UC-02: Iniciar sessió . . . 94
UC-15: Canviar contrasenya . . . 95
UC-16 i UC-07: Eliminació de comptes d'usuari . . . 96
9.3.3 Gestió d'arxius (FileManagement) . . . 99
UC-08: Crear i pujar elements . . . 99
UC-09: Modificar elements . . . 103
UC-10: Descarregar elements . . . 109
9.3.4 Compartició (FileSharing) . . . 110
UC-13: Compartir un element . . . 111
UC-13A i UC-13B: Revocar accés i Deixar de compartir . . . 115
9.3.5 Papelera i eliminació asíncrona (TrashService) . . . 117
UC-11 i UC-17: Borrat lògic i restauració d'elements . . . 117
UC-12: Eliminació permanent i orquestració asíncrona . . . 118
9.3.6 Servei de sincronització (SyncService) . . . 121
Flux de connexió WebSocket . . . 121
Processament d'esdeveniments i actualització de l'estat . . . 123
Estratègies de difusió i punts clau . . . 124
9.3.7 Refactor del codi . . . 126
9.4 Implementació del client web . . . 127
9.4.1 Introducció . . . 127
Estructura de directoris del client web . . . 127
9.4.2 Disseny Visual i Components de la Interfície . . . 129
Autenticació . . . 129
Escriptori Principal i Navegació . . . 130
Interacció amb Elements . . . 132
Seccions Específiques . . . 135
9.4.3 Implementació per Casos d'Ús . . . 138
Gestió d'Usuaris . . . 138
Gestió d'Administració . . . 143
9.4.4 Gestió d'Arxius . . . 146
9.4.5 Compartició d'Arxius . . . 153
9.5 Implementació del client web . . . 159
9.5.1 Introducció . . . 159
Estructura de directoris del client web . . . 159
9.5.2 Disseny Visual i Components de la Interfície . . . 160
Autenticació . . . 161
Escriptori Principal i Navegació . . . 161

Interacció amb Elements . . . 164
Seccions Específiques . . . 167
9.5.3 Implementació per Casos d'Ús . . . 170
Gestió d'Usuaris . . . 170
Gestió d'Administració . . . 175
9.5.4 Gestió d'Arxius . . . 178
9.5.5 Compartició d'Arxius . . . 185
9.6 Implementació del client de escriptori . . . 191
9.6.1 Introducció . . . 191
9.6.2 Disseny Visual i Gestió de Finestres . . . 191
Flux Inicial: Configuració i Autenticació . . . 191
Interfície Principal i Menú del Sistema . . . 194
Finestra de Configuració . . . 196
9.6.3 Motor de Sincronització: El Nucli Reactiu . . . 198
Sincronització Local-a-Remot: Vigilant el Sistema d'Arxius . . . 199
Lògica de Comparació i Detecció de Canvis . . . 201
Sincronització Remot-a-Local: Escoltant el Servidor . . . 202
9.6.4 Proves d'integració manuals . . . 204
9.7 Implementació dels sistemes d'instal·lació i desplegament . . . 205
9.7.1 Desplegament mitjançant Docker Compose . . . 205
Arquitectura de desplegament . . . 205
Estructura de l'arxiu compose.yml . . . 206
Configuració de xarxa i volums . . . 209
9.7.2 Scripts d'instal·lació automatitzada . . . 209
Script per a Linux (setup.sh) . . . 209
Script per a Windows (setup.ps1) . . . 212
Procés de creació del superadministrador . . . 212

10 Implantació i resultats **213**
10.1 Introducció . . . 213
10.2 Autenticació i gestió d'usuaris . . . 213
10.3 Gestió d'arxius . . . 215
10.3.1 Pujada d'arxius . . . 215
10.3.2 Descàrrega d'arxius . . . 216
10.3.3 Creació de carpetes . . . 217
10.4 Operacions amb arxius . . . 219
10.4.1 Operacions de copiar, tallar i enganxar . . . 219
10.4.2 Moviment i redenominació . . . 220
10.4.3 Eliminació d'arxius . . . 221
10.5 Sistema de paperera . . . 222
10.6 Compartició d'arxius . . . 222
10.6.1 Creació de comparticions . . . 222
10.6.2 Gestió d'arxius compartits . . . 225
10.6.3 Modificació i revocació de comparticions . . . 225
10.7 Sincronització . . . 229
10.8 Funcionalitats addicionals . . . 230
10.8.1 Ordenació d'arxius . . . 230
10.9 Panell d'administració . . . 230
10.9.1 Gestió d'usuaris . . . 230
10.9.2 Eliminació d'usuaris . . . 232
10.10 Gestió de perfils . . . 233

10.11 Compliment dels objectius 235
10.12 Demostració de sincronització en temps real 235
10.13 Avaluació del Compliment Normatiu 236
10.13.1 Enfocament del compliment en un projecte de Codi Obert . . . 236
10.13.2 Mesures de compliment implementades al codi 236
10.13.3 Responsabilitats transferides a l'Administrador del Sistema . . 237
10.13.4 Mesures no aplicades i Treball a Futur (Capítol 12) 237
10.14 Conclusió 238

11 Conclusions **240**
11.1 Assoliment dels objectius 240
11.1.1 Valoració com a Treball de Fi de Grau 240
11.1.2 Valoració del producte desenvolupat 241
11.2 Desviacions de la planificació 241
11.3 Discussió crítica dels resultats 242
11.4 Conclusions finals i lliçons apreses 242

12 Treball futur i línies de millora **244**
12.1 Millores Crítiques de Seguretat i Compliment Normatiu (Prioritat Alta) 244
12.2 Completar Funcionalitats del Client d'Escriptori (Prioritat Alta) 245
12.3 Optimització del Rendiment i l'Escalabilitat (Prioritat Mitjana) 245
12.4 Millores en la Gestió i el Desplegament (Prioritat Mitjana) 246
12.5 Millores Funcionals i d'Usabilitat (Prioritat Baixa) 246
12.6 Reflexió sobre el Futur del Projecte 247

13 Manual d'instal·lació **248**
13.1 Requisits previs 248
13.2 Procés d'instal·lació automatitzada 248
13.2.1 Obtenció del codi font 249
13.2.2 Execució de l'script 249
13.3 Gestió del sistema 250
13.4 Accés a les aplicacions 251

Capítol 1

Introducció, motivacions, propósits i objectius

1.1 Context del projecte

En l'era digital actual, la gestió d'arxius personals i compartits s'ha convertit en una necessitat fonamental tant per a usuaris individuals com per a grups de treball, famílies o petites organitzacions. La creixent dependència de serveis d'emmagatzematge al núvol ha facilitat l'accés remot, la compartició i la sincronització de documents, però també ha generat noves problemàtiques relacionades amb el cost, la privacitat, la dependència tecnològica i la manca de control per part de l'usuari final.

Durant la meva experiència personal amb serveis com Google Drive, OneDrive o Mega, vaig identificar diverses limitacions que em van portar a plantejar aquest projecte. Aquestes solucions, encara que funcionals, imposen restriccions pel que fa a l'espai disponible, requereixen subscripcions per accedir a funcionalitats completes i no ofereixen un control total sobre les dades. Davant d'aquesta situació, vaig decidir desenvolupar una alternativa lliure, extensible i autogestionada.

La solució que proposo es basa en una arquitectura de microserveis, i està dissenyada per ser accessible tant des d'una interfície web com des d'una aplicació d'escriptori multiplataforma. Gràcies a l'ús de contenidors Docker, el desplegament del sistema resulta senzill fins i tot per a usuaris sense coneixements tècnics avançats en oferir també un conjunt d'scripts d'instal·lació automatitzats.

1.2 Motivacions

La meva motivació principal ha estat disposar d'una eina gratuïta, de codi obert i sense restriccions artificials d'ús, que ofereixi una funcionalitat comparable a la dels serveis comercials actuals, retornant el control de les dades a l'usuari final.

Durant el procés de desenvolupament d'aquest projecte, va mancar l'anàlisi de les eines lliures existents, ja que desconeixia la seva existència, un error per part meva que podria haver fet que replantegés el projecte des d'una direcció diferent, però la meva experiència amb serveis comercials va ser suficient per detectar mancances importants pel que fa a accessibilitat, privacitat i control. Aquest projecte neix del

desig de superar aquestes limitacions, desenvolupant una solució que pogués ser utilitzada per qualsevol, sense cap cost i amb plena autonomia sobre les seves dades.

A més, el caràcter obert del projecte permet fomentar la col·laboració, l'aprenentatge i la millora contínua per part de la comunitat, alineant-se amb els principis del programari lliure i la sobirania digital, oferint al final una solució lliure, distribuïda i extensible.

1.3 Propósit del projecte

El propòsit general d'aquest treball és desenvolupar una plataforma de gestió d'arxius al núvol que permeti la sincronització i compartició de fitxers entre múltiples usuaris, mitjançant una arquitectura distribuïda basada en microserveis. El sistema haurà d'estar preparat per al seu ús tant des d'una aplicació web com des d'una aplicació d'escriptori, adaptant-se així a diferents escenaris i dispositius.

He dissenyat aquesta plataforma amb la intenció que sigui autogestionada, reproduïble mitjançant contenidors i accessible per a usuaris sense coneixements tècnics, gràcies a la inclusió de scripts d'instal·lació automatitzats. El meu objectiu és facilitar a qualsevol persona la possibilitat de desplegar el seu propi núvol privat de forma gratuïta, segura i senzilla.

1.4 Objectius generals

Els principals objectius generals que em proposo assolir amb aquest projecte són:

- Dissenyar una arquitectura backend modular, basada en microserveis, que permeti una alta escalabilitat, mantenibilitat i separació de responsabilitats.
- Desenvolupar una interfície web moderna utilitzant React i Tailwind CSS, amb un disseny centrat en l'experiència d'usuari.
- Crear una aplicació d'escriptori multiplataforma utilitzant Tauri i Svelte, que proporcioni accés complet a les funcionalitats del sistema i sigui compatible amb diferents sistemes operatius a més d'utilitzar el mínim de recursos perquè pugui ser utilitzat en dispositius amb poca capacitat.
- Facilitar el desplegament i ús del sistema mitjançant contenidors Docker i scripts d'instal·lació que minimitzin la intervenció tècnica de l'usuari.
- Publicar tot el sistema sota llicència de codi obert (MIT).

1.5 Objectius específics

A més dels objectius generals, he definit una sèrie d'objectius tècnics concrets que permetran donar suport a les funcionalitats requerides per la plataforma:

- Implementar un sistema d'autenticació d'usuaris segur i extensible.
- Desenvolupar microserveis independents per a la gestió de fitxers, compartició entre usuaris, paperera de reciclatge i sincronització d'estats.

- Implementar un sistema de control de permisos que permeti definir els nivells d'accés als arxius compartits.
- Integrar mecanismes de sincronització en temps real mitjançant WebSockets, que assegurin l'actualització immediata de l'estat dels arxius entre tots els clients connectats.
- Incloure una interfície de paperera per a la recuperació d'arxius eliminats de forma accidental.
- Assegurar la portabilitat del sistema entre diferents plataformes mitjançant l'ús de contenidors Docker i fitxers de configuració estàndard (com `docker-compose.yml`).

1.6 Situació inicial i públic objectiu

Abans de començar el projecte, comptava amb una base sòlida en programació amb Java i Spring Boot, així com en l'ús de contenidors Docker. També tenia coneixements funcionals en desenvolupament web amb React i una comprensió teòrica del paradigma de microserveis, encara que sense experiència pràctica prèvia.

He dissenyat el sistema pensant en petits grups d'usuaris —com famílies o grups d'amics— que desitgin disposar d'una solució al núvol privada i de fàcil instal·lació. Per això, he desenvolupat scripts automatitzats que simplifiquen el procés de desplegament i configuració, eliminant la necessitat de coneixements tècnics avançats.

Finalment, la meva intenció inicial era continuar el desenvolupament del sistema després de la finalització del Treball de Fi de Grau, incorporant noves funcionalitats i millores, i consolidant-lo com una eina lliure, distribuïda i extensible per a la gestió d'arxius personals, però durant el transcurs del desenvolupament del projecte vaig descobrir tota una comunitat de desenvolupadors que estaven treballant en un projecte similar, la qual cosa va fer que em plantegi a futur primer investigar les capacitats dels projectes ja establerts (ja que són més madurs i tenen una comunitat activa al darrere) i finalment decidir si el meu projecte pot aportar una solució millorada o si ja estan cobertes aquestes necessitats.

Capítol 2

Estudi de viabilitat

2.1 Viabilitat tècnica

Per desenvolupar aquest projecte vaig optar per eines i llenguatges amb els quals ja comptava amb experiència prèvia o que oferien clars avantatges tècnics. En el backend vaig utilitzar Java amb Spring Boot a causa de la seva maduresa, extensibilitat i que ja havia treballat amb aquest entorn a nivell professional. Aquesta elecció em va permetre aplicar bones pràctiques de desenvolupament, integrant biblioteques com MapStruct o Spring Cloud per millorar la productivitat i la mantenibilitat del codi.

Per a la interfície web vaig seleccionar React amb Vite i Tailwind CSS. React em va permetre construir una aplicació reactiva, modular i reutilitzable, mentre que Vite va millorar notablement els temps d'arrencada i recàrrega en calent durant el desenvolupament. L'elecció de Tailwind CSS es va justificar per la seva eficiència en la definició d'estils directament des del codi dels components, sense necessitat d'arxius CSS separats.

Pel que fa al client d'escriptori, vaig decidir utilitzar Tauri amb Svelte. Tot i que inicialment vaig considerar reutilitzar la interfície de React, finalment vaig optar per Svelte pel seu menor consum de recursos. Svelte no inclou un runtime, la qual cosa genera binaris més lleugers i amb menor ús de memòria, especialment rellevant en entorns d'escriptori. Aquesta decisió es va basar en la documentació de Tauri i en el consell d'una persona del meu entorn amb una gran quantitat d'experiència en desenvolupament d'aplicacions Tauri a nivell personal.

L'arquitectura es recolza en microserveis, que s'orquestren mitjançant Docker i Docker Compose. El fitxer `compose.yml` defineix els següents serveis:

- **user-auth**: gestió d'autenticació i tokens.
- **user-management**: gestió de dades d'usuari.
- **file-access**: control d'accés a arxius.
- **file-sharing**: compartició d'arxius entre usuaris.
- **trash**: paperera de reciclatge.
- **file-manager**: pujada, descàrrega i metainformació de fitxers i carpetes.

- **sync**: gestió de sincronització en temps real.
- **gateway**: punt d'entrada unificat per al client web i d'escriptori.
- **eureka**: descobriment de serveis.
- **postgres**: base de dades.
- **rabbitmq**: sistema de missatgeria per a comunicació asíncrona.

Aquesta configuració permet aixecar tota la infraestructura amb una sola comanda, garantint que l'entorn de proves és reproduïble i estàndard en tots els desplegaments, a més de fer l'entorn agnòstic al sistema operatiu que l'ha de córrer, gràcies a les capacitats dels contenidors Docker. La base de dades PostgreSQL es comparteix entre serveis, però cadascun gestiona les seves pròpies taules, evitant dependències creuades. Aquesta va ser una decisió de disseny per seguir les bones pràctiques de disseny de microserveis. RabbitMQ permet una comunicació desacoblada entre serveis que requereixen notificació o processament en segon pla, com la sincronització o el buidatge de la paperera.

L'entorn de desenvolupament es va basar en VSCode com a IDE principal, aprofitant la seva extensió per a Java, el suport per a React i el terminal integrat. Les proves es van realitzar inicialment amb Postman i, més endavant, mitjançant una llista de proves funcionals mantinguda manualment (documentada al fitxer `tests.md`) per garantir la cobertura de totes les funcionalitats del sistema tant al backend com al frontend.

2.2 Viabilitat econòmica

El cost econòmic del projecte ha estat nul. Totes les eines utilitzades són de codi obert i gratuïtes. Java, React, Vite, Svelte, Tauri, Docker, Git, VSCode i Postman no requereixen llicències de pagament. A més, el projecte s'ha desenvolupat en un ordinador personal ja disponible, equipat amb un processador Intel i7 i 32 GB de RAM, de manera que no ha calgut adquirir maquinari addicional.

Tampoc es preveuen costos de manteniment, ja que les eines emprades són estables, àmpliament documentades i un estàndard de la indústria, fet que n'afavoreix el suport a llarg termini i la possibilitat d'incorporar col·laboradors externs.

2.3 Viabilitat humana

Aquest projecte s'ha desenvolupat de forma individual. Initially vaig planificar dedicar les tardes i els caps de setmana durant diversos mesos, amb una estimació d'unes 640 hores de treball en total. Tot i que aquesta planificació va resultar massa optimista i no es va seguir de manera exacta, considero que el temps real invertit ha estat raonable i que, si s'hagués mantingut, hauria estat suficient per finalitzar el projecte, com es detalla al Capítol 4.

El projecte també es va concebre com una oportunitat per aprendre noves tecnologies. Es va fixar l'objectiu explícit d'aprofundir en l'ús de React, experimentar amb

arquitectures basades en microserveis, treballar de prop amb tecnologies consolidades com RabbitMQ o Eureka i explorar eines fins aleshores desconegudes com Svelte, Rust i Tauri.

L'organització del treball va incloure l'ús de Git per al control de versions. El codi final està disponible en un repositori dedicat al projecte. El repositori original, utilitzat durant les primeres fases del desenvolupament, es va allotjar en un compte personal vinculat a altres projectes privats; per això es va decidir migrar-lo a un nou repositori exclusiu per a la seva difusió pública.

2.4 Viabilitat legal

Tot i que el projecte s'ha desenvolupat tenint en compte el marc legal vigent (RGPD i LOPDGDD), cal remarcar que es tracta d'una eina de codi obert destinada a ser desplegada i gestionada per tercers. Per tant, la responsabilitat última sobre el compliment de la normativa de protecció de dades recau en l'usuari o organització que decideixi utilitzar la plataforma en un entorn real.

Per facilitar aquest compliment, el projecte inclou plantilles base d'avís legal i política de privacitat (ubicades al directori /legal), que han de ser revisades i adaptades per qui desplegui la solució. Tot i això, en aquesta versió, el sistema no implementa totes les mesures tècniques avançades de seguretat (com el xifratge de dades en trànsit), per la qual cosa es recomana aplicar les mesures addicionals necessàries segons la normativa.

El projecte es publica sota llicència MIT, que en permet l'ús, la modificació i la redistribució sense restriccions, sempre que es conservi l'avís de llicència. La inclusió de les plantilles legals esmentades, juntament amb la llicència permissiva, busca oferir una base sòlida per a un ús responsable i legal de l'eina.

2.5 Conclusió

Considero que el projecte és tècnicament viable gràcies a l'experiència prèvia en les tecnologies seleccionades, l'ús d'eines madures i la modularitat de l'arquitectura. A més, el fet d'utilitzar exclusivament eines de codi obert elimina qualsevol barrera econòmica.

El temps dedicat al projecte, molt superior al previst, posa de manifest que la planificació inicial era massa ambiciosa. La magnitud de la feina va ser més gran de l'esperada, i cal admetre que, fins i tot amb una gestió del temps perfecta, l'abast del projecte probablement superava el que era realista finalitzar en el termini d'un TFG, com es detalla al Capítol 4.

Tot i que el sistema no disposa actualment d'eines de monitoratge ni de logs estructurats, aquest aspecte es contempla com una millora futura que s'abordarà al Capítol 12.

En conjunt, el projecte s'emmarca dins dels recursos disponibles per a un Treball Final de Grau. Les tecnologies emprades i la llicència oberta faciliten que la plataforma sigui adoptada, estesa i millorada fàcilment per altres desenvolupadors o usuaris interessats.

Capítol 3

Metodologia seguida

3.1 Marc teòric: voluntat d'un enfocament àgil

Des de l'inici del projecte, la meva intenció era aplicar una metodologia àgil inspirada en *Scrum*, adaptada a la realitat d'un únic desenvolupador. El plantejament teòric consistia a dividir el treball en sprints curts, amb planificació, desenvolupament i revisió regulars, per tal de poder avançar de manera incremental i adaptar-me als imprevistos.

A la realitat, aquesta estructura àgil va quedar principalment a la fase de planificació. A la pràctica, la temporalitat dels sprints no es va respectar i el desenvolupament es va veure fortament condicionat per la dificultat de compaginar el projecte amb la vida laboral i personal. Aquesta manca de continuïtat i de gestió del temps va ser un dels principals factors que van afectar el progrés, com s'analitza amb més detall al Capítol 4.

3.2 Seguiment i control

En lloc d'utilitzar eines digitals o sistemes formals de seguiment, el control de l'avanç es va fer mitjançant una simple llibreta d'objectius. En aquesta llibreta anotava les tasques a realitzar, afegia nous objectius a mesura que sorgien i anava marcant o tachant els que es completaven. No es va fer servir cap eina de gestió de projectes ni registre digital, fet que en retrospectiva considero un error, ja que va dificultar la visibilitat global del progrés i la priorització de tasques.

La revisió de les tasques es feia una vegada es donava com a completades, fent un test de la funcionalitat i si es completava es marcava com a completada. Era durant aquest procés que pasava per un període de reflexió sobre el funcionament per a contemplar si la implementació era la correcta o no i si necessitava d'una nova interpretació.

3.3 Definition of Done: l'únic criteri formal

Tot i la manca d'un sistema de seguiment estructurat, sí que vaig mantenir una *Definition of Done* clara per a cada tasca o funcionalitat. Aquest criteri va ser l'únic element formalment aplicat durant tot el projecte i em va ajudar a garantir un mínim de qualitat i coherència en el resultat final.

Definition of Done

Per evitar ambigüitats vaig definir aquests criteris de tancament:

- El projecte compilava sense errors i la imatge Docker es generava correctament.
- Totes les proves funcionals definides passaven sense incidències.
- La funcionalitat era accessible i usable des de la interfície client corresponent (web o escriptori).
- En cas de ser un dels blocs principals del desenvolupament, es feia una breu sessió exploratòria i cap dels testers trobava errors crítics.

Aquesta *Definition of Done* va ser clau per mantenir un estàndard de qualitat constant, tot i la manca d'altres mecanismes de control.

3.4 Fases del projecte: del pla a la realitat

Per estructurar el desenvolupament, inicialment vaig dividir el projecte en vuit fases, tal com es mostra a la Taula 3.1. Aquesta planificació pretenia agrupar els sprints i establir objectius tangibles per a cada etapa.

TAULA 3.1: Fases d'implementació i criteris de finalització

Fase	Criteri de finalització
1. Preparació i aprenentatge	Definició de requisits, entorn Docker operatiu i exploració inicial de Rust/Tauri.
2. Backend inicial	Microserveis d'autenticació, usuaris i gestió d'arxius desenvolupats.
3. Prototip web (MVP)	Client web funcional capaç de pujar i descarregar arxius.
4. Prototip escriptori (Tauri)	Client d'escriptori bàsic amb llistat d'arxius i revisió de l'API.
5. Paperera de reciclatge	Implementació de la funcionalitat de paperera.
6. Compartició i Sync	Incorporació de la compartició d'arxius i sincronització en temps real.
7. Accés a compartits (Tauri)	Client d'escriptori capaç d'accedir a arxius compartits.
8. Admin i desplegament	Panell d'administració creat, proves integrals realitzades i desplegament final.

Aquesta taula reflecteix el disseny original, però la realitat del desenvolupament va ser molt menys lineal. Tal com s'explica al Capítol 4, el projecte va patir desviacions importants respecte al calendari previst: pauses llargues, replanificacions i una execució fragmentada per la dificultat de mantenir la continuïtat. Les fases es van solapar, alguns objectius es van ajornar i d'altres es van modificar o descartar segons la disponibilitat i el feedback rebut.

3.5 Desglossament i revisió de les unitats de treball

Cada fase es traduïa en objectius concrets, però el desglossament i la revisió d'aquestes unitats de treball es feia de manera informal. No hi havia un procés sistemàtic d'anàlisi, desenvolupament i prova dins d'un mateix sprint, sinó que les tasques s'anaven adaptant i reescrivint a mesura que avançava el projecte, sempre anotades a la llibreta.

3.6 Feedback dels testers

El feedback dels testers va ser sempre informal i de paraula, sense cap registre formal. Els comentaris rebuts eren del tipus "m'agrada aquesta part de l'aplicació"o "podries fer que aquesta altra sigui diferent com X". Aquestes aportacions, tot i ser valuoses, no seguien cap procés estructurat ni quedaven documentades més enllà de la meva pròpia memòria o de notes puntuals.

3.7 Dificultats de gestió i conciliació

Un dels principals obstacles va ser la dificultat de compaginar el desenvolupament del projecte amb la vida laboral i personal. Aquesta manca de dedicació continuada, sumada a l'absència d'un sistema de gestió del temps i de seguiment formal, va provocar retards i una execució molt menys àgil del que s'havia previst. Aquesta qüestió s'analitza amb més profunditat al Capítol 4.

3.8 Gestió de riscos i lliçons apreses

Al llarg del projecte vaig identificar diversos riscos, però la manca d'eines de seguiment i la dificultat per mantenir la continuïtat van ser els més rellevants. La taula següent recull els principals riscos i les estratègies de resposta:

TAULA 3.2: Riscos principals i estratègia de mitigació

ID	Risc	Resposta / Mitigació
R1	Falta de temps a causa d'obligacions laborals i personals.	Reprioritzar tasques i posposar característiques no crítiques al pla de treball futur.
R2	Corba d'aprenentatge de Tauri i Rust.	Dedicar una fase específica de formació i buscar ajuda puntual d'un expert.
R3	Absència d'integració contínua, mètriques objectives i eines de seguiment.	Validació manual abans de cada *merge*; deixar la implantació de CI i eines de gestió com a millora futura.

3.9 Lliçons apreses

Les lliçons apreses apunten clarament a la necessitat d'adoptar eines de seguiment, definir millor la gestió del temps i establir fites intermèdies més rigoroses en futurs projectes. La flexibilitat de l'enfocament àgil va ser útil, però sense disciplina i eines adequades, la planificació es va veure superada per la realitat del dia a dia.

Capítol 4

Planificació

4.1 Pla original

Abans d'escriure la primera línia de codi vaig traçar un calendari ambiciós: acabar el projecte en **vuit mesos** aprofitant tardes i caps de setmana. La idea era avançar de baix a dalt: desenvolupar el backend, validar un prototip web funcional, sumar el client d'escriptori i, finalment, polir i provar tot el conjunt.

TAULA 4.1: Full de ruta inicial (Oct-2023 → Jun-2024)

Mes	Objectiu principal
Oct 2023	Definir requisits, preparar entorn Docker i explorar Rust / Tauri.
Nov-Des 2023	Desenvolupar microserveis d'autenticació, usuaris i operacions d'arxius.
Gen 2024	Construir un prototip web capaç de pujar i descarregar arxius.
Feb 2024	Prototipar el client Tauri i revisar l'API.
Mar 2024	Afegir la paperera.
Abr 2024	Incorporar la compartició d'arxius i la sincronització en temps real.
Mai 2024	Permetre que el client Tauri accedeixi als arxius compartits.
Jun 2024	Crear el panell d'administració, executar proves integrals i desplegar.

4.2 Execució real i validacions

La trajectòria real del projecte va ser força diferent del que s'havia planejat inicialment. El desenvolupament es va allargar fins al juliol de 2025, amb diversos períodes de pausa entremig. Durant aquest temps, vaig realitzar tres validacions principals amb usuaris beta testers: la primera després d'implementar l'administració de fitxers, la segona quan vaig completar la paperera, i l'última un cop finalitzada la sincronització.

Cal destacar que, després de la implementació de la compartició d'arxius, es va detectar un error d'arquitectura en la gestió dels serveis del backend. Aquest error feia que tant el servei de paperera com el de compartició depenguessin del servei de file manager com a punt de contacte, trencant així el principi d'acoblament feble (loose coupling) dels microserveis, que estableix que cada servei ha de ser independent i autònom. Això va obligar a fer un refactor important del backend, que va durar

aproximadament dues setmanes i mitja i es va solapar amb el desenvolupament del servei de compartició. Aquest refactor va requerir tornar a testejar tota l'aplicació per assegurar que no s'havien trencat funcionalitats ja implementades, i va endarrerir l'inici del desenvolupament de la sincronització, que es va començar just després. El detall tècnic d'aquest error i la solució adoptada es tracta amb més profunditat al capítol 8.

Tot això es pot veure reflectit al diagrama de Gantt de la Figura 4.1, on es mostren les diferents etapes del desenvolupament, els períodes d'aturada, les proves amb usuaris (en groc), les millores aplicades (en morat) i el període de refactor (en marró).

Finalment, cal destacar que una de les funcionalitats planificades, la visualització d'arxius compartits des del client d'escriptori, es va haver de descartar a causa de la complexitat tècnica i la falta de temps. Aquesta última etapa de desenvolupament del prototip de Tauri, a més, es va haver de realitzar en paral·lel amb la creació del panell d'administració per tal de complir amb els terminis.

Els beta-tests van durar entre un i dos dies i van ser decisius per polir usabilitat. Gràcies a ells vaig afegir la compartició múltiple d'arxius, accessos ràpids de teclat i components *Tremor Raw*, entre altres millores.

FIGURA 4.1: Cronograma real amb desenvolupaments, pauses, refactorització i validacions (Oct-2023 → Jul-2025)

4.3 Pla vs. execució: la bretxa quantificada

TAULA 4.2: Retards acumulats respecte al pla inicial

Fita completada	Pla	Real	Retard / comentari
Auth + Usuaris	Nov-Des 2023	Feb 2024	+2 mesos
Admin de fitxers (MVP)	Feb 2024	Mar 2024	+1 mes; optimització React + accessibilitat
Paperera	Abr 2024	Oct 2024	+6 mesos; pauses intermitents
Compartició	Mai 2024	Nov 2024	+6 mesos; validació + refactor
Accés a compartits (Tauri)	Jun 2024	No implementat	Descartat per complexitat i falta de temps
Refactor backend	—	Nov 2024	2,5 setmanes; revalidació completa
Sync	Mai 2024	Feb 2025	+9 mesos; inici després del refactor
Prototip Tauri	Mar 2024	Jun 2025	+15 mesos; desenvolupament en paral·lel amb panell d'admin
Admin + proves globals	Jun 2024	Jun 2025	+12 mesos
Redacció memòria	—	Jun-Jul 2025	no previst; 2 setmanes exclusives

4.4 Causes de les desviacions

1. **Discontinuïtat en la dedicació**. Les pauses prolongades van exigir un "peatge cognitiu"que estimo entre un 10 i un 15 % del temps total: reprendre context, re-configurar l'entorn i refrescar la lògica del codi.

2. **Corba d'aprenentatge subestimada**. Rust i Svelte van requerir més pràctica de la prevista; la integració amb Tauri es va desplaçar gairebé un any.

3. **Obligacions laborals i viatges**. Entre gener i octubre de 2024 vaig alternar desplaçaments a Granada i Alemanya, fragmentant els cicles de treball.

4.5 Accions correctores aplicades

- Vaig re-prioritzar el backlog: vaig desplaçar el prototip de Tauri al tram final, sacrificant la funcionalitat de visualitzar arxius compartits des d'aquest, i vaig deixar la sincronització de fitxers compartits a local per a futures versions.

- Vaig reservar un sprint-buffer al desembre 2024 per absorbir retards acumulats i estabilitzar el full de ruta.

4.6 Estimació d'hores dedicades

TAULA 4.3: Pla vs. realitat en esforç mensual aproximat

Mètrica	Pla	Real
Mitjana mensual	60 h	20 – 40 h
Mes pic	60 h	~45 h
Mes vall	60 h	0 h
Pèrdua per represes	—	10 – 15 % del temps total

4.7 Lliçons finals

Aquesta experiència em va ensenyar que la **continuïtat** pesa tant com la quantitat d'hores: cada pausa llarga afegeix un peatge cognitiu que penalitza el calendari. Igualment, sense **mètriques objectives** (Kanban, full d'hores, CI amb indicadors) és fàcil sobrestimar els avenços.

Per a futurs projectes aplicaré:

1. Un tauler Kanban públic.
2. Buffers explícits després de viatges o pics laborals que redueixin la incertesa.
3. Prototips inicials per validar corbes d'aprenentatge abans de planificar dates ambicioses.
4. Sessions de feedback programades des del primer dia, amb temps reservat per implementar millores.

En definitiva, tot i que els terminis inicials es van dilatar, el cronograma real mostra una evolució honesta, validada amb usuaris i enriquida amb lliçons que ja aplico en la meva pràctica professional.

Capítol 5

Marc de treball i conceptes previs

5.1 Introducció

Aquest Treball de Fi de Grau s'ha desenvolupat de manera independent, sense el suport d'una organització externa, amb l'objectiu d'aprofundir en la construcció de sistemes distribuïts moderns, així com un repte tècnic personal en afrontar un projecte de gran magnitud i complexitat amb múltiples llenguatges i tecnologies. El seguiment acadèmic ha anat a càrrec del professor **Josep Soler** (tutor del projecte).

El desenvolupament del client d'escriptori es va realitzar en col·laboració molt estreta amb **Jawad Adbellaoui**, qui va aportar la seva experiència en Tauri i Rust. Aquesta col·laboració va ser clau per assolir un resultat funcional i em va permetre aprendre de primera mà bones pràctiques en tecnologies emergents.

Al llarg del capítol es descriuen les eines i tecnologies que conformen la pila de desenvolupament. Cada una es presenta especificant el seu propòsit, la motivació de la seva elecció i el seu paper dins l'arquitectura global. Totes elles són de codi obert i d'ús gratuït, fet que reforça la sostenibilitat econòmica discutida al Capítol 2.

5.2 Entorn de desenvolupament

El treball s'ha dut a terme principalment en un portàtil amb **Ubuntu 22.04 LTS** (kernel 5.15) i l'editor **Visual Studio Code** com a IDE principal. La Taula 5.1 —inclosa al final d'aquest capítol— detalla les versions exactes dels llenguatges, marcs i eines utilitzades. Per a les proves inicials de l'API es va utilitzar **Postman**, tot i que el flux de validació va passar ràpidament a la pròpia aplicació web a mesura que el frontend va guanyar funcionalitat.

5.3 Eines de desenvolupament

Sistema operatiu

Ubuntu 22.04 LTS proporciona un entorn estable, actualitzat i àmpliament documentat. El seu gestor de paquets *apt* i la compatibilitat amb contenidors `docker-ce` van facilitar la instal·lació de dependències i la creació d'entorns reproduïbles.

Git i GitHub

Git és un sistema de control de versions distribuït creat per Linus Torvalds el 2005. La seva arquitectura descentralitzada permet treballar sense connexió, conservar l'-històric complet a cada clon i afavorir fluxos paral·lels mitjançant branques lleugeres. GitHub afegeix a Git funcionalitats col·laboratives (*pull requests*, revisions de codi i GitHub Actions) i ha servit d'eix per a la integració contínua i el desplegament automatitzat del projecte.

Visual Studio Code

Visual Studio Code (VS Code) és un editor multiplataforma, lleuger i extensible. Amb extensions com *Rust Analyzer*, *Spring Tools* i suport integrat per a TypeScript, ha permès una experiència de desenvolupament unificada per al *backend* en Java i els clients en React i Tauri/Svelte.

Node.js

Node.js és un entorn d'execució que permet fer servir JavaScript fora del navegador, principalment al servidor. Gràcies a Node.js, es poden crear aplicacions i scripts que s'executen directament a la màquina, com ara eines de desenvolupament, servidors web o processos d'automatització. En aquest projecte, Node.js s'utilitza per executar scripts que ajuden a preparar l'entorn i per donar suport a la construcció i execució del frontend web.

pnpm

pnpm (performant npm) és un gestor de paquets per a Node.js, dissenyat per ser ràpid i eficient en l'ús de l'espai en disc. A diferència de npm, que pot duplicar paquets, pnpm utilitza un magatzem de contingut adreçable on només es desa una versió de cada paquet. Després, crea enllaços simbòlics (symlinks) a la carpeta `node_modules` del projecte. Aquest enfocament no només redueix dràsticament l'espai en disc necessari, sinó que també accelera significativament els temps d'instal·lació. En aquest projecte, s'ha estandarditzat l'ús de pnpm per gestionar totes les dependències dels frontends (`ui-new` i `desktop`) i per executar scripts útils durant el desenvolupament i la construcció de les aplicacions, garantint consistència i eficiència.

Postman

Postman és una eina gràfica molt utilitzada per desenvolupar, provar i documentar APIs REST. Permet enviar peticions HTTP (GET, POST, PUT, DELETE, etc.) a un servidor, veure les respostes, gestionar col·leccions de peticions i validar el comportament dels endpoints de manera ràpida i visual. Això facilita la detecció d'errors, la comprovació de contractes i la col·laboració entre equips de backend i frontend.

En aquest projecte, Postman es va utilitzar durant les primeres iteracions per dissenyar i provar les peticions a l'API REST, assegurant que els endpoints funcionaven correctament i que les respostes complien l'estructura esperada. A mesura que el frontend va madurar i va guanyar funcionalitat, les proves es van traslladar a la pròpia interfície web, reduint la dependència de Postman i agilitzant el flux de validació.

Scripts d'instal·lació

Els scripts setup.sh (Bash) i setup.ps1 (PowerShell) automatitzen la preparació de l'entorn: descàrrega de dependències, construcció de contenidors Docker i configuració de variables. Així es garanteix la reproduïbilitat de l'entorn amb una sola comanda.

5.4 Frontend web

React

React (2013) és una biblioteca JavaScript de codi obert creada per Facebook per facilitar la construcció d'interfícies d'usuari (UI) complexes de manera eficient, escalable i mantenible. El seu model es basa en la composició de components: unitats independents i reutilitzables que encapsulen tant la lògica com la presentació, afavorint la reutilització, la testabilitat i la separació de responsabilitats.

Una de les innovacions clau de React és el *Virtual DOM*, una representació en memòria de l'estructura del DOM real. Quan l'estat d'un component canvia, React genera un nou arbre virtual i calcula la diferència respecte a l'anterior (*diffing algorithm*). Només les parts modificades s'actualitzen al DOM real mitjançant el procés de *reconciliació*, minimitzant operacions costoses i millorant el rendiment, especialment en aplicacions dinàmiques o amb moltes actualitzacions.

FIGURA 5.1: Procés de reconciliació a React: arbre virtual → diff → pegat sobre el DOM.

El flux de dades en React és unidireccional (de pares a fills), cosa que simplifica el raonament sobre l'estat i redueix la probabilitat d'efectes col·laterals. Les dades es passen als components fills mitjançant *props*, mentre que l'estat local es gestiona dins de cada component. Per compartir estat entre components no relacionats jeràrquicament, es pot utilitzar el context de React o gestors d'estat externs com Zustand.

Amb la introducció dels *hooks*, la gestió de l'estat i dels efectes col·laterals es pot fer de manera funcional, sense recórrer a classes. Hooks com useState, useEffect o

useContext permeten encapsular lògica reutilitzable i millorar la llegibilitat i mantenibilitat del codi.

React destaca també per la seva integració amb l'ecosistema JavaScript modern i una comunitat molt activa. Llibreries com TanStack React Query, Zustand, Radix UI o Vite cobreixen des de la gestió eficient de dades asíncrones fins a l'accessibilitat i l'optimització del frontend. Això permet abordar projectes professionals de qualsevol escala, des de prototips fins a sistemes grans i robustos.

A més, React facilita paradigmes avançats com la renderització al servidor (SSR), les aplicacions d'una sola pàgina (SPA) i la integració amb WebSockets per a temps real. La seva filosofia de considerar la UI com una funció de l'estat ($UI = f(state)$) simplifica la gestió de la complexitat i afavoreix la previsibilitat del comportament de l'aplicació. Tot plegat fa de React una eina idònia per a projectes que requereixen manteniment, escalabilitat i un alt grau d'interactivitat.

TypeScript

TypeScript (2012) amplia JavaScript amb tipatge estàtic i característiques orientades a objectes. El *transpiler* tsc prevé errors en temps de compilació i facilita l'autocompletat, millorant la robustesa entre client i servidor.

Vite

Vite és una eina que ajuda a posar en marxa i construir aplicacions web de manera molt ràpida i senzilla. Durant el desenvolupament, permet veure els canvis a l'instant cada vegada que es modifica el codi, sense haver d'esperar llargues càrregues. Això fa que programar sigui molt més àgil i còmode. Quan arriba el moment de publicar l'aplicació, Vite s'encarrega d'ordenar i optimitzar tots els fitxers perquè la web es carregui ràpidament als usuaris. En aquest projecte, Vite ha estat fonamental per poder desenvolupar i provar la interfície web de forma eficient i sense complicacions.

Tailwind CSS

Tailwind CSS és un framework de CSS utilitari que permet construir interfícies d'usuari modernes i responsives mitjançant l'ús de classes predefinides molt específiques (com p-4, text-lg, bg-blue-500, etc.). A diferència dels frameworks tradicionals basats en components o temes, Tailwind aposta per una filosofia ütility-first", on cada classe aplica un sol estil CSS, la qual cosa permet escriure el disseny directament a l'HTML o JSX de cada component.

Aquesta aproximació té diversos avantatges: elimina la necessitat d'escriure fulls d'estil personalitzats per a cada component, redueix la duplicació de codi i facilita la coherència visual a tota l'aplicació. A més, Tailwind facilita la creació de dissenys responsius i adaptatius gràcies a la seva sintaxi per a punts de trencament i variants d'estat (hover, focus, dark mode, etc.).

Un dels punts forts de Tailwind és el seu procés de "purge"o depuració: durant la construcció de l'aplicació, Tailwind analitza tot el codi font i elimina automàticament les classes que no s'utilitzen, generant un fitxer CSS final molt lleuger i optimitzat

per a producció. Això millora el rendiment de càrrega i l'experiència d'usuari, especialment en aplicacions grans.

En aquest projecte, Tailwind CSS ha permès desenvolupar una interfície moderna, coherent i altament personalitzable, accelerant el procés de maquetació i assegurant una bona experiència visual.

Radix UI

Radix UI és una llibreria de components d'interfície d'usuari per a React, centrada en l'accessibilitat i la composició. A diferència d'altres biblioteques, Radix UI proporciona components "headless", és a dir, sense estils visuals predefinits, de manera que el desenvolupador pot aplicar el seu propi disseny sense perdre funcionalitat ni accessibilitat. Tots els components compleixen les directrius ARIA i gestionen automàticament focus, navegació per teclat i altres requisits d'accessibilitat, la qual cosa garanteix que l'aplicació sigui usable per a tothom. A més, Radix UI facilita la creació de diàlegs, menús, popovers i altres elements complexos sense haver de preocupar-se pels detalls interns de gestió d'estat o focus. En aquest projecte, Radix UI s'ha utilitzat per implementar diàlegs i menús accessibles, assegurant una experiència d'usuari coherent i inclusiva.

FIGURA 5.2: Diàleg basat en Radix amb focus i navegació accessibles.

React Query

React Query (actualment TanStack Query) és una llibreria per a la gestió eficient de dades asíncrones en aplicacions React. La seva funció principal és facilitar la consulta, la memòria cache i la sincronització de dades provinents d'APIs o serveis externs. React Query permet definir "queries"declaratives que s'executen automàticament quan el component es renderitza, gestionant l'estat de càrrega, error i dades sense necessitat de codi addicional. A més, manté una memòria cache intel·ligent que evita peticions innecessàries i actualitza automàticament les dades quan detecta canvis, aplicant estratègies com "stale-while-revalidate". Això simplifica molt la gestió de dades remotes, millora el rendiment i redueix la complexitat del codi, especialment en aplicacions amb moltes consultes o dades en temps real. En aquest projecte, React Query ha estat clau per mantenir la UI sincronitzada amb el backend i gestionar l'estat de les peticions de manera robusta i eficient.

FIGURA 5.3: Circuit de dades a React Query: memòria cache → fetch → revalidate.

El funcionament intern de React Query es pot entendre millor amb l'ajuda de la imatge anterior. Al centre del sistema hi ha el **QueryClient**, que gestiona totes les opcions i la configuració global. Aquest client conté una **QueryCache**, que és la memòria cache on es desen totes les dades de les consultes (queries) realitzades a l'API o backend.

Quan un component React utilitza el hook useQuery, crea un **Query Observer** que s'encarrega de subscriure's a una consulta concreta dins la memòria cache. Si la dada ja existeix a la cache, el component la rep immediatament; si no, React Query fa la petició a l'API i desa la resposta a la cache per a futures consultes. Això permet que diversos components puguin compartir i reutilitzar dades sense repetir peticions innecessàries.

A més, la memòria cache es pot sincronitzar amb un emmagatzematge extern (com LocalStorage) mitjançant els **Persistors**, de manera que les dades es conserven encara que l'usuari recarregui la pàgina. Tot aquest sistema garanteix que l'estat de les dades a la UI estigui sempre actualitzat, gestionant automàticament la recàrrega, la invalidació i la revalidació de la informació segons sigui necessari.

Aquesta arquitectura fa que React Query sigui molt eficient i robust per a la gestió de dades asíncrones en aplicacions React modernes.

Zustand

Zustand és una llibreria lleugera per a la gestió d'estat global en aplicacions React. A diferència de solucions més pesades com Redux, ofereix una API molt minimalista basada en *hooks*: cada *store* és, de fet, un hook creat amb create(). Els components consumeixen l'estat cridant aquest hook i poden subscriure's només a la porció que necessiten, cosa que evita re–renderitzacions innecessàries i millora el rendiment. Internament, Zustand utilitza *proxies* per detectar canvis i notificar únicament els

components afectats, facilita la divisió en *slices* per escalar grans projectes i disposa de *middleware* per persistir, registrar o desfer canvis.

FIGURA 5.4: Diagrama de flux de Zustand en una aplicació React: els components llegeixen o actualitzen l'estat a través dels *hooks*, que deleguen l'operació al *store*; quan aquest actualitza l'estat, només notifica els components subscrits.

Com es veu al diagrama 5.4, el flux és extremadament directe:

1. **Components React** criden el *hook* del *store* per llegir l'estat o executar accions.
2. El ***hook*** retorna l'estat sol·licitat i exposa les accions definides dins el *store*.
3. Les **accions** invocades fan $\mathrm{set}()$ o $\mathrm{get}()$ per modificar l'estat.
4. El ***store*** actualitza l'estat i notifica només els components subscrits al tros de dades canviat, provocant el re–render corresponent.

Aquesta simplicitat permet separar la lògica d'estat dels components, millorar la testabilitat i mantenir el codi net.

En el projecte actual, Zustand s'utilitza per gestionar l'estat global relacionat amb la navegació d'arxius, la selecció múltiple i el context de visualització. Aquesta aproximació permet mantenir la UI sincronitzada i reactiva, simplificant la gestió de l'estat entre components i facilitant la implementació de funcionalitats avançades com la selecció múltiple, el porta-retalls i la navegació entre seccions.

@dnd-kit i Selecto

@dnd-kit és una llibreria modular que abstrau la complexitat de les interaccions d'arrossegar i deixar anar (drag & drop) mitjançant sensors que detecten esdeveniments del ratolí, tàctils i de teclat. La seva arquitectura basada en contexts i proveïdors permet implementar funcionalitats avançades com l'ordenació de llistes o el moviment entre contenidors de forma declarativa.

Per altra banda, Selecto ofereix una API per implementar selecció múltiple mitjançant un rectangle o llaç de selecció, similar al que trobem en aplicacions d'escriptori. Permet configurar la detecció de col·lisions, filtrar elements seleccionables i personalitzar l'aparença visual del llaç.

La combinació d'aquestes dues llibreries ha permès implementar una experiència d'usuari molt similar a la d'un explorador d'arxius natiu, on es poden seleccionar

múltiples elements arrossegant el ratolí i després moure'ls a altres carpetes mitjançant drag & drop, tot mantenint una implementació neta i mantenible.

WebSocket API

L'aplicació utilitza WebSocket, un protocol de comunicació que estableix una connexió bidireccional persistent entre el client web React i el servidor. A diferència del protocol HTTP tradicional, on el client ha de fer peticions explícites per obtenir dades, WebSocket manté un canal obert que permet al servidor enviar informació al client en qualsevol moment. Aquesta característica és especialment útil en la nostra interfície web per implementar funcionalitats en temps real, com la sincronització automàtica quan altres usuaris modifiquen arxius o la recepció instantània de notificacions del sistema.

5.5 Backend i microserveis

Java 17

Java és un llenguatge de programació orientat a objectes àmpliament utilitzat en el desenvolupament d'aplicacions empresarials, sistemes distribuïts i serveis web. La seva principal característica és la portabilitat: el codi Java es compila a bytecode, que pot ser executat en qualsevol plataforma que disposi d'una màquina virtual Java (JVM), independentment del sistema operatiu o el maquinari. Això facilita la creació d'aplicacions escalables i mantenibles, ja que el mateix codi pot desplegar-se en entorns molt diversos sense modificacions. Java ofereix un sistema de tipus fort, gestió automàtica de memòria mitjançant recollida d'escombraries (garbage collection) i una àmplia biblioteca estàndard per a operacions de xarxa, persistència, concurrència i seguretat. A més, la comunitat Java disposa d'un ecosistema ric en frameworks i eines que acceleren el desenvolupament i garanteixen la robustesa de les aplicacions. En el context d'aquest projecte, Java s'utilitza com a llenguatge principal per implementar els microserveis del backend, aprofitant la seva maduresa, estabilitat i suport a llarg termini.

Spring Boot

Spring Boot és un framework que simplifica la creació d'aplicacions Java basades en Spring, especialment microserveis. Proporciona una configuració automàtica intel·ligent i una estructura d'aplicació predefinida, permetent als desenvolupadors centrar-se en la lògica de negoci en lloc de la configuració manual. Un dels seus avantatges clau és la capacitat de generar microserveis autosuficients, que inclouen un servidor Tomcat embegut, eliminant la necessitat de desplegar arxius WAR en servidors externs. Els *starters* de Spring Boot encapsulen conjunts de dependències habituals, facilitant la integració de tecnologies com bases de dades, seguretat, missatgeria o REST. A més, ofereix eines per monitoritzar, provar i desplegar serveis de manera eficient. En aquest projecte, Spring Boot serveix de base per a cada microservei, assegurant una inicialització ràpida, una gestió coherent de dependències i una arquitectura modular i escalable.

Spring Data JPA

Spring Data JPA és un framework que facilita la persistència de dades en bases de dades relacionals mitjançant el patró ORM (Object-Relational Mapping). Permet definir entitats Java que es corresponen amb taules de la base de dades i proporciona una API per realitzar operacions CRUD (crear, llegir, actualitzar, eliminar) sense escriure SQL manualment. Una de les seves funcionalitats més potents és la generació automàtica de consultes JPQL a partir de la nomenclatura dels mètodes (per exemple, findByUsernameAndStatus), la qual cosa accelera el desenvolupament i redueix errors. A més, integra mecanismes per a la gestió de transaccions, la paginació, la ordenació i la validació d'entitats. En el projecte, Spring Data JPA s'utilitza per accedir i manipular les dades de cada microservei, assegurant una persistència robusta, coherent i fàcilment testejable.

Eureka

Eureka és el servei de descobriment que manté un registre viu i coherent de tots els microserveis desplegats. Cada instància, tan bon punt s'aixeca, es registra automàticament al servidor i, mitjançant petits *heartbeats* periòdics, confirma que continua disponible. Així, qualsevol altre servei —o bé el Gateway— pot consultar el registre per obtenir l'adreça més recent i balancejar les peticions sense codificar IPs fixes. Aquesta capacitat d'autodescobriment garanteix resiliència i escalabilitat en entorns on les instàncies poden aparèixer o desaparèixer dinàmicament durant pics de càrrega, actualitzacions o fallades.

FIGURA 5.5: Flux de descobriment amb Eureka: els microserveis *Mail* i *Address* es registren al servidor; quan el primer necessita localitzar el segon, consulta el registre i obté l'URL actual abans de fer la crida.

La figura 5.5 mostra aquest procés en acció. Quan el *Mail Service* s'inicialitza (quadrat de l'esquerra) envia la seva sol·licitud de registre al servidor Eureka, igual que fa l'*Address Service* (quadrat de la dreta). Un cop registrats, tots dos renoven la seva entrada de forma periòdica. Quan l'usuari demana enviar un correu, el *Mail Service*

no necessita conèixer prèviament on s'està executant l'*Address Service*; simplement interroga el registre, rep l'adreça actual i executa la petició HTTP. Si l'instància canvia de port o se'n desplega una altra, la propera consulta ja retornarà la nova ubicació sense cap intervenció manual. D'aquesta manera, Eureka actua com un directori telefònic dinàmic dels microserveis i assegura que la comunicació interna sigui fluida, flexible i tolerant a fallades: un pilar imprescindible en la nostra arquitectura distribuïda.

Spring Cloud Gateway

Spring Cloud Gateway és el punt d'entrada reactiu i altament eficient de la nostra arquitectura de microserveis. Actua com a *reverse proxy* basat en Netty: rep totes les peticions externes, avalua predicats i filtres declarats en el application.yml, i redirigeix cada sol·licitud al microservei pertinent. Aquesta porta única permet centralitzar l'autenticació amb JWT, aplicar *rate-limiting*, reescriure capçaleres o rutes i, alhora, protegir la xarxa interna de microserveis.

FIGURA 5.6: Flux de Spring Cloud Gateway: totes les peticions externes passen pel *Gateway*, que aplica filtres i distribueix el trànsit cap als microserveis; alhora, es coordina amb el *Config Server* per obtenir la configuració, amb *Service Discovery* per registrar i descobrir instàncies, i amb Zipkin per a la traça distribuïda.

Com il·lustra la figura 5.6, les sol·licituds de l'usuari arriben primer al bloc verd de *Spring Cloud Gateway*, on s'executen predicats (paths, mètodes, host, etc.) i filtres (autenticació, límit de peticions, logging). Si la petició compleix els criteris, el Gateway la deriva —amb equilibri de càrrega— al microservei corresponent, que prèviament s'ha registrat al servidor de descoberta. Mentrestant, la configuració dinàmica es recupera del *Config Server* i la traça de cada salt es diposita a Zipkin. D'aquesta manera, aconseguim un punt de control únic, reaccionant amb baixa latència fins i tot sota alta concurrència, i simplificant la seguretat i l'observabilitat de tot l'ecossistema.

PostgreSQL

PostgreSQL és un sistema de gestió de bases de dades relacional d'alt rendiment, de codi obert i reconegut per la seva robustesa, extensibilitat i compliment estricte dels principis ACID (Atomicitat, Consistència, Aïllament i Durabilitat). Ofereix suport avançat per a tipus de dades complexos, incloent-hi columnes JSONB, que és un tipus de dada natiu de PostgreSQL que permet emmagatzemar documents JSON de manera binària i optimitzada, per a l'emmagatzematge eficient de documents semi-estructurats, així com arrays, enums i tipus definits per l'usuari. Permet la creació de consultes SQL complexes, vistes, funcions i triggers, i implementa mecanismes de control de concurrència multiversió (MVCC) per garantir l'accés simultani sense bloquejos. PostgreSQL destaca també per la seva capacitat d'escalar tant verticalment com horitzontalment, la integració amb extensions (com PostGIS per a dades geoespacials) i el suport natiu per a transaccions distribuïdes. En aquest projecte, PostgreSQL s'executa dins d'un contenidor Docker amb volum persistent, assegurant la integritat i la disponibilitat de les dades fins i tot en cas de reinici o migració de l'entorn.

Spring AMQP i RabbitMQ

RabbitMQ

RabbitMQ és un sistema de missatgeria basat en el protocol AMQP 0-9-1 que introdueix un intermediari (*message broker*) entre productors i consumidors. Aquesta capa intermèdia permet que els microserveis es comuniquin de manera asíncrona i totalment desacoblada: cada missatge s'envia a un *exchange*, que el distribueix a una o més cues (*queues*) segons els *bindings* i les claus de *routing* definides. Gràcies a la persistència de missatges, les confirmacions i els reintents, el broker pot retenir els esdeveniments fins que el consumidor estigui disponible, absorbint pics de càrrega sense perdre informació i permetent escalar productors i consumidors de forma independent.

FIGURA 5.7: Esquema de RabbitMQ: un productor publica missatges a l'*exchange*, que els encamina cap a les cues vinculades mitjançant *bindings* i claus de *routing*; cada consumidor processa els missatges de la seva cua de manera independent.

Tal com es veu a la figura 5.7, el productor només necessita conèixer l'*exchange* al qual publicar; un cop el missatge arriba al broker, el ruteig cap a les cues es fa de forma declarativa i dinàmica. Els consumidors, al seu torn, extreuen missatges de les cues que els pertoquen i confirmen la recepció quan els han processat. Aquest patró

elimina dependències rígides entre serveis, facilita l'equilibri de càrrega i garanteix la fiabilitat fins i tot si algun component deixa de respondre momentàniament.

Spring AMQP

Spring AMQP és un conjunt de llibreries de l'ecosistema Spring que proporciona una abstracció d'alt nivell per treballar amb sistemes de missatgeria compatibles amb AMQP, com RabbitMQ. Simplifica la integració amb el broker mitjançant la configuració declarativa de cues, intercanvis i lligams, i ofereix una API orientada a missatges per enviar i rebre dades de manera tipada i segura. Spring AMQP gestiona automàticament la serialització i deserialització d'objectes Java, la gestió de connexions, la confirmació de missatges i la implementació de mecanismes de reintent en cas d'errors temporals. A més, permet definir *listeners* asíncrons que processen missatges de manera concurrent, millorant l'escalabilitat i la reactivitat del sistema. En aquest projecte, la combinació de RabbitMQ i Spring AMQP permet implementar fluxos d'esdeveniments robustos, desacoblar microserveis i garantir la fiabilitat en la comunicació interna.

Spring Cloud OpenFeign

Spring Cloud OpenFeign és una llibreria que permet crear clients REST declaratius a partir d'interfícies Java, simplificant la comunicació entre microserveis. Amb OpenFeign, només cal definir una interfície amb anotacions que descriuen els endpoints remots (`@GetMapping`, `@PostMapping`, etc.), i el framework genera automàticament la implementació que realitza les crides HTTP corresponents. Això elimina la necessitat de gestionar manualment la serialització, la construcció d'URLs o la gestió d'errors bàsics, i permet una integració més neta i tipada amb altres serveis.

OpenFeign suporta la injecció d'interceptors personalitzats, que permeten afegir capçaleres comunes (com `Authorization` amb JWT, `X-Request-Id`, etc.) a totes les peticions de manera centralitzada. També facilita la gestió de la signatura i validació de tokens JWT, la propagació de contextos de seguretat i la implementació de patrons de resiliència com *retry* o *circuit breaker* mitjançant integració amb Resilience4j. A més, permet configurar codificadors i decodificadors personalitzats per adaptar-se a formats de dades específics (JSON, XML, etc.), i ofereix suport per a la documentació automàtica dels clients.

En aquest projecte, OpenFeign s'utilitza per connectar microserveis interns de manera segura i eficient, garantint la coherència en la gestió de capçaleres, l'autenticació i la traçabilitat de les peticions. Aquesta aproximació declarativa redueix el codi repetitiu, millora la mantenibilitat i facilita l'escalabilitat de l'arquitectura.

Spring Security

Spring Security és el marc de seguretat de referència per a aplicacions Spring. Proporciona mecanismes robustos per a l'autenticació i l'autorització, permetent definir regles d'accés a nivell de rutes, mètodes o recursos. Al nostre projecte, Spring Security s'encarrega de validar cada petició que arriba al Gateway, assegurant que només els usuaris autenticats i amb els permisos adequats puguin accedir als microserveis interns. La configuració es realitza de manera declarativa, integrant filtres personalitzats per a la validació de tokens i la gestió de rols d'usuari.

JWT (JSON Web Token)

JWT és un estàndard per a la transmissió segura d'informació entre parts com a objectes JSON signats digitalment. En aquest sistema, cada usuari rep un token JWT després d'autenticar-se correctament. Aquest token, signat amb un algorisme de criptografia asimètrica (RSA), conté la informació essencial de l'usuari (identitat, rols, data d'expiració, etc.) i s'envia en cada petició posterior al Gateway. L'ús de JWT elimina la necessitat de mantenir estat de sessió al servidor, ja que tota la informació rellevant viatja dins del token, i permet escalar el sistema de manera eficient. A més, la signatura criptogràfica garanteix la integritat i autenticitat del token, reduint la latència i millorant la seguretat global de la plataforma.

5.6 Client d'escriptori

Tauri

Tauri és un framework modern per al desenvolupament d'aplicacions d'escriptori multiplataforma que aprofita tecnologies web (HTML, CSS, JavaScript/TypeScript) per a la interfície d'usuari, però genera binaris natius extremadament lleugers (<15 MB, en comparació amb els >100 MB d'Electron). Utilitza la WebView nativa del sistema operatiu (com WebView2 a Windows, WKWebView a macOS i WebKitGTK a Linux), la qual cosa redueix significativament el consum de memòria i recursos, i minimitza el temps de descàrrega i instal·lació. A més, Tauri proporciona una API segura per accedir a funcionalitats del sistema (fitxers, xarxa, notificacions, etc.) mitjançant un pont entre el codi web i el backend natiu escrit en Rust. Aquesta arquitectura permet mantenir una superfície d'atac reduïda, actualitzacions ràpides i una millor integració amb el sistema operatiu, tot garantint la seguretat i la privacitat de l'usuari.

Svelte

Svelte és un framework per al desenvolupament d'interfícies d'usuari que es distingeix d'altres solucions com React o Vue perquè trasllada la major part del processament al moment de la compilació. En comptes de fer servir un *virtual DOM* i gestionar les actualitzacions de manera reactiva durant l'execució, Svelte transforma els components en codi JavaScript imperatiu optimitzat que manipula directament el DOM. Aquesta aproximació permet que les aplicacions s'iniciïn més ràpidament i consumeixin menys recursos, aspectes especialment rellevants en aplicacions d'escriptori on la rapidesa i l'eficiència són fonamentals. Svelte facilita la creació de components reutilitzables, la gestió d'estat reactiu i la integració amb llibreries externes, tot mantenint una sintaxi senzilla i declarativa. En aquest projecte, Svelte s'ha utilitzat per construir la interfície del client d'escriptori,aprofitant la seva eficiència i la facilitat d'integració amb Tauri.

Rust

Rust és un llenguatge de programació de sistemes reconegut per la seva seguretat de memòria, absència de condicions de carrera i alt rendiment. El seu sistema de propietat i préstec (*ownership & borrowing*) evita errors comuns com accessos a memòria nul·la o dobles alliberaments sense necessitat de recollida d'escombraries (*garbage collector*). En el context de Tauri, Rust s'utilitza per implementar el backend natiu

de l'aplicació d'escriptori, gestionant operacions sensibles com l'accés a fitxers, la comunicació amb la xarxa, la criptografia i la integració amb APIs del sistema operatiu. Aquesta elecció garanteix que les operacions crítiques siguin ràpides, segures i fiables, i permet aprofitar l'ecosistema de llibreries de Rust per a funcionalitats avançades. A més, la interoperabilitat entre Rust i el frontend web de Tauri es realitza mitjançant canals segurs, assegurant la separació de privilegis i la protecció davant vulnerabilitats.

5.7 Utilitats addicionals

Docker

Docker és una plataforma de virtualització lleugera que empaqueta cada microservei amb totes les seves dependències dins d'un contenidor aïllat. En comptes de virtualitzar maquinari complet —com fan les màquines virtuals (VM)— Docker aprofita funcionalitats del nucli de Linux (*namespaces* i *cgroups*) per crear espais d'usuari independents que comparteixen el mateix nucli del sistema host. D'aquesta manera, cada contenidor arrenca en pocs segons, ocupa molt menys disc i memòria, i s'executa de forma idèntica en qualsevol entorn, eliminant el típic "works on my machine". El procés de construcció, definit al Dockerfile, garanteix traçabilitat i reproduïbilitat: la mateixa imatge es pot desplegar localment o en producció sense sorpreses.

FIGURA 5.8: Arquitectura basada en el nucli compartit: Docker Engine s'executa directament sobre el nucli de Linux i crea contenidors que comparteixen aquest nucli però disposen d'espais de noms i control de recursos propis; per contra, cada VM carrega un sistema operatiu complet sobre l'hipervisor.

La figura 5.8 mostra com el motor de Docker actua com una capa fina entre el sistema operatiu host i els contenidors. El nucli (en blau) roman únic per a totes les instàncies, mentre que cada contenidor (en taronja) inclou només la capa d'aplicacions. Aquesta construcció aprofita que totes les distribucions Linux comparteixen el mateix nucli, convertint Docker en una "pseudo màquina virtualültralleugera que pot crear-se i destruir-se ràpidament, fet que facilita tant l'escalat automàtic com els desplegaments sense temps mort. En resum, amb Docker garantim portabilitat, consistència i una gestió d'infraestructura tan declarativa com el codi que hi corre. Això

també dóna l'avantatge que es pot crear una configuració única perquè les aplicacions mai tinguin el problema d'estar en un entorn sense alguna configuració o eina que necessitin per funcionar, el que dóna la seguretat que sempre podran funcionar independentment de l'entorn on s'executin.

Docker Compose

Docker Compose és una eina que permet definir i gestionar aplicacions multi-contenidor mitjançant un únic fitxer de configuració YAML (compose.yml). En aquest fitxer es descriuen tots els serveis que formen part de l'arquitectura (per exemple, base de dades, backend, frontend, serveis de missatgeria, etc.), així com les xarxes virtuals i els volums persistents que utilitzen. Compose permet especificar variables d'entorn, dependències entre serveis, ordres d'inicialització i polítiques de reinici, facilitant la coordinació i l'orquestració de tot el sistema. Amb una sola comanda (docker compose up), es poden aixecar tots els contenidors de manera coherent, garantint que la infraestructura es desplega de forma consistent tant en entorns de desenvolupament com de producció. Això simplifica enormement la posada en marxa, les proves i el manteniment de l'ecosistema de microserveis.

5.8 Versionat de dependències

TAULA 5.1: Resum de versions efectives

Tecnologia	**Versió**
Java	17
Spring Boot	3.2.5
Spring Cloud Gateway	2023.0.1
Spring Security	6.2.4
PostgreSQL	14.18
RabbitMQ	4.1.2
React	18.2.0
TypeScript	5.2.2
Vite	5.0.0
Tailwind CSS	3.3.5
React Query	5.52.2
Zustand	5.0.4
Tauri	1.5
Svelte	5.0.0
Docker	27.0.3
Docker Compose	3.8

Capítol 6

Requisits del sistema

Aquest capítol descriu de forma detallada tots els requisits que ha de complir el sistema per garantir el seu correcte funcionament. Es classifiquen en requisits funcionals i no funcionals, i es descriuen els aspectes relacionats tant amb el programari com amb el maquinari necessari per al desenvolupament i l'execució.

6.1 Introducció

La plataforma neix amb la voluntat d'oferir una alternativa lliure, autoallotjada i extensible per a la gestió d'arxius al núvol. Prenent com a punt de partida les motivacions descrites al *Capítol 1*, s'emfatitza la necessitat de controlar completament l'emmagatzematge i la sincronització de dades sense dependre de proveïdors externs.

En síntesi, el sistema comprèn:

- Administració d'arxius i carpetes amb paperera de reciclatge.
- Compartició segura entre usuaris amb permisos granulars.
- Sincronització en temps real entre dispositius mitjançant WebSocket i notificacions d'esdeveniments.

6.2 Requisits funcionals

6.2.1 1. Gestió d'usuaris

- Registre de nous usuaris.
- Inici de sessió segur mitjançant autenticació amb JWT.
- Edició i actualització de perfils d'usuari.
- Control d'accés basat en rols: usuari estàndard i administrador.

6.2.2 2. Gestió d'arxius

- Pujar arxius i carpetes a l'espai d'usuari.

- Descarregar arxius de forma individual o en grup.
- Eliminar arxius (amb trasllat a la paperera).
- Crear i gestionar carpetes personals.
- Navegar per l'estructura de directoris.

6.2.3 3. Sistema de paperera

- Eliminació temporal d'arxius amb trasllat a la paperera.
- Restauració d'arxius des de la paperera.
- Eliminació permanent manual o automàtica.

6.2.4 4. Compartició d'arxius

- Definició de permisos d'accés: lectura, escriptura, etc.
- Visualització d'arxius compartits amb altres usuaris.
- Llistat i revocació de comparticions actives.

6.2.5 5. Sincronització en temps real

- Actualització automàtica de canvis entre clients.
- Notificacions d'activitats (pujades, canvis, eliminacions).
- Sincronització d'arxius entre diversos dispositius.

6.2.6 6. Panell d'administració

- Gestió d'usuaris: creació, edició i eliminació.
- Monitoratge del sistema i estat dels serveis.

6.3 Requisits no funcionals

6.3.1 1. Rendiment

- Temps de resposta inferior a 2 segons en operacions habituals. **(Prioritat mitjana)**
- Suport per a la manipulació d'arxius grans (1GB). **(Prioritat baixa)**
- Optimització de la transferència de dades entre client i servidor. **(Prioritat mitjana)**
- Escalabilitat horitzontal per suportar més càrrega. **(Prioritat alta)**

6.3.2 2. Seguretat

- Autenticació segura amb tokens JWT. **(Prioritat alta)**
- Protecció contra atacs comuns (XSS, CSRF, SQLi). **(Prioritat alta)**
- Validació estricta de dades d'entrada i sortida. **(Prioritat mitjana)**
- Control d'accés granular per arxiu i per usuari. **(Prioritat alta)**

6.3.3 3. Usabilitat

- Interfície intuïtiva i adaptada a dispositius de diverses resolucions. **(Prioritat alta)**
- Funcionalitat de *drag and drop* per facilitar el moviment d'arxius. **(Prioritat alta)**
- Missatges d'error clars i informatius. **(Prioritat alta)**
- Ajuda contextual i indicacions visuals per guiar l'usuari. **(Prioritat mitjana)**

6.3.4 4. Mantenibilitat

- Codi modular per facilitar el manteniment. **(Prioritat mitjana)**
- Arquitectura extensible per afegir noves funcionalitats. **(Prioritat alta)**
- Procés de desplegament fàcil i automatitzat. **(Prioritat mitjana)**
- Diagnòstic i traçabilitat d'errors amb eines de logging. **(Prioritat mitjana)**

6.3.5 5. Compatibilitat

- Suport per a tots els navegadors moderns (Chrome, Firefox, Edge). **(Prioritat alta)**
- Funcionament en sistemes Windows i Linux. **(Prioritat alta)**
- Adaptació a múltiples resolucions de pantalla. **(Prioritat alta)**
- Gestió de formats de fitxer comuns (PDF, DOCX, PNG, etc.). **(Prioritat mitjana)**

6.3.6 6. Escalabilitat

- Arquitectura distribuïda basada en microserveis. **(Prioritat alta)**
- Gestió eficient de recursos i instàncies. **(Prioritat mitjana)**
- Optimització de l'espai d'emmagatzematge. **(Prioritat baixa)**
- Capacitat per a múltiples usuaris actius simultàniament. **(Prioritat mitjana)**

6.3.7 7. Portabilitat

- Instal·lació senzilla a través de contenidors Docker. **(Prioritat alta)**
- Configuració mitjançant fitxers .env editables. **(Prioritat alta)**
- Dependències mínimes per executar cada component. **(Prioritat alta)**
- Documentació clara per a desenvolupadors i usuaris. **(Prioritat alta)**
- Guia d'instal·lació i configuració pas a pas. **(Prioritat alta)**

6.4 Requisits de maquinari i programari

Per tal d'executar la solució completa –format microserveis basats en Spring Boot, PostgreSQL i RabbitMQ al costat del frontend React i del client d'escriptori Tauri–, cal comptar amb els recursos següents:

- **Entorn servidor/desenvolupament (Docker Compose)**: un processador de com a mínim quatre nuclis (Intel Core i5 o equivalent), entre **8 GB i 16 GB** de memòria RAM i **5 GB** d'espai lliure per allotjar imatges i volums de dades. Es recomana un sistema operatiu de 64 bits amb Docker 20 + i Docker Compose 2 +.
- **Client d'escriptori (Tauri + Svelte)**: gràcies a la compilació a codi natiu, l'aplicació és molt lleugera i funciona sense problemes en equips de 64 bits amb **2 GB** de RAM i uns **200 MB** d'espai en disc. Compatible amb Windows, Linux i macOS.
- **Generació de clients** (web i escriptori): cal disposar de **Node.js 18+**, **Rust 1.73+** i Docker per orquestrar els serveis durant el desenvolupament.

6.5 Resum

Aquest capítol ha recollit tots els requisits –funcionals i no funcionals– que defineixen el producte. S'han desglossat les funcionalitats clau, els estàndards de seguretat i els objectius de rendiment. Els capítols posteriors de disseny i implementació demostren com s'han abordat aquests requisits, tot i que algun objectiu, com la gestió d'arxius compartits al client d'escriptori, es va haver de descartar per la seva complexitat i falta de temps.

Capítol 7

Estudis i decisions

7.1 Introducció

Aquest capítol explora el procés de reflexió i les decisions tècniques que han donat forma a l'arquitectura d'aquest projecte. Més que una simple llista d'eines, el que es presenta a continuació és un recorregut per les anàlisis i els criteris que van guiar l'elecció de cada component, sempre amb l'objectiu de materialitzar la visió central del projecte: crear una plataforma de gestió de fitxers autogestionada, segura i de codi obert, on l'usuari final mantingui el control total sobre les seves dades.

Per navegar aquest procés de decisió, em vaig basar en un conjunt de criteris clau:

- **Experiència prèvia:** Aprofitar un coneixement de base en certes tecnologies per accelerar les fases inicials del desenvolupament.
- **Rendiment i eficiència:** Prioritzar solucions lleugeres, un requisit crític per al client d'escriptori.
- **Cohesió de l'ecosistema:** Optar per conjunts d'eines dissenyades per integrar-se de manera fluida.
- **Capacitats tècniques específiques:** Seleccionar tecnologies que oferissin solucions a reptes concrets com la comunicació asíncrona o la gestió de dades.
- **Codi obert i compliment legal:** Escollir eines de codi obert, un factor que garanteix la sostenibilitat i la viabilitat econòmica del projecte a llarg termini i que facilitessin l'adhesió a normatives com el RGPD.

Aquestes directrius em van permetre dissenyar una arquitectura coherent i alineada amb la visió del projecte.

7.2 Pila Tecnològica del Backend

Per al backend, vaig decidir implementar una arquitectura de microserveis. Aquesta elecció es fonamenta en la necessitat de construir un sistema modular, escalable i fàcil de mantenir, on cada servei tingui una responsabilitat única. Vaig escollit l'ecosistema de Java i Spring per la seva coneguda maduresa, estabilitat i ampli suport en entorns empresarials.

7.2.1 Spring Boot

Spring Boot és un framework que simplifica la creació d'aplicacions Java autònomes i llestes per a producció. Facilita la configuració i el desplegament de microserveis gràcies a la seva filosofia de çonvenció sobre configuracióï a la inclusió de servidors web embeguts.

Abans de prendre una decisió, vaig analitzar alternatives modernes a Spring Boot altament considerades en el desenvolupament de microserveis per la seva eficiència [**baeldung-spring-alternatives**]. A continuació, es presenta una taula comparativa que resumeix els factors clau.

Framework	Avantatges (Pros)	Inconvenients (Contres)
Spring Boot	• Ecosistema molt madur i estable. • Gran comunitat i suport empresarial. • Experiència prèvia que garantia alta productivitat.	• Temps d'arrencada més lent. • Major consum de memòria en comparació amb les alternatives.
Quarkus	• Temps d'arrencada extremadament ràpids. • Baix consum de memòria (optimitzat per a contenidors). • Enfocament natiu a GraalVM.	• Corba d'aprenentatge en no tenir experiència prèvia. • Risc per al compliment dels terminis del projecte.
Micronaut	• Temps d'arrencada quasi instantanis. • Mínim ús de reflexió gràcies a la injecció de dependències en temps de compilació.	• També presentava una corba d'aprenentatge. • Ecosistema menys extens que el de Spring.

TAULA 7.1: Comparativa de frameworks Java per a microserveis.

Justificació de l'elecció Vaig basar la meva elecció principalment en l'experiència prèvia que ja tenia amb l'ecosistema Spring. Aquesta familiaritat em va permetre assolir una alta productivitat des de l'inici del projecte. Tot i que vaig considerar alternatives modernes com Quarkus o Micronaut, que prometen millors temps d'arrencada, la corba d'aprenentatge associada hauria suposat un risc per al compliment dels terminis. Spring Boot representava, doncs, la via més pragmàtica i segura per al meu cas.

7.2.2 Spring Data JPA

Spring Data JPA facilita la implementació de la capa de persistència de dades en aplicacions Spring, abstraient gran part del codi necessari per a les operacions de base de dades. La seva capacitat més destacada és la generació automàtica de consultes a partir de la signatura dels mètodes en una interfície de repositori.

Justificació de l'elecció Vaig adoptar Spring Data JPA per la seva integració nativa i sense fricció amb Spring Boot. L'objectiu era accelerar el desenvolupament de la capa de dades, i la seva capacitat per generar repositoris em permetia eliminar una gran quantitat de codi repetitiu en comparació amb l'ús de JPA estàndard o JDBC. A més, en fomentar l'ús de consultes parametritzades de manera inherent, proporciona una capa de seguretat fonamental en prevenir atacs de tipus injecció SQL, un requisit indispensable per garantir la integritat de les dades segons el RGPD.

7.2.3 PostgreSQL

PostgreSQL és un sistema de gestió de bases de dades relacional d'objectes, reconegut per la seva robustesa, extensibilitat i compliment estricte de l'estàndard SQL i les propietats ACID.

Per a la selecció del sistema de gestió de bases de dades, vaig comparar les opcions més consolidades del mercat [**dbengines-comparison**], tenint en compte tant requisits tècnics com factors pràctics.

Justificació de l'elecció Vaig escollir PostgreSQL no només per la seva reputació com a base de dades fiable i de codi obert, sinó també perquè és el sistema amb el qual tinc més experiència, la qual cosa em va permetre ser més productiu. A més, la seva coneguda robustesa en la gestió de rols i permisos s'alineava perfectament amb les exigències del RGPD per assegurar la confidencialitat i la integritat de les dades dels usuaris, sent aquest un factor clau en la decisió.

SGBD	Avantatges (Pros)	Inconvenients (Contres)
PostgreSQL	• Codi obert i gratuït. • Altament extensible i personalitzable. • Suport avançat per a tipus de dades complexos (JSONB, GIS, etc.). • Compliment estricte de l'estàndard SQL.	• Pot tenir una corba d'aprenentatge lleugerament superior a MySQL per a tasques bàsiques.
MySQL	• Molt popular i fàcil d'iniciar per a aplicacions senzilles. • Bon rendiment en escenaris de lectura intensiva. • Gran comunitat d'usuaris.	• Menys funcionalitats avançades que PostgreSQL. • Propietat d'Oracle, la qual cosa genera incertesa sobre el seu futur com a projecte totalment obert.
Oracle	• Solució empresarial molt potent i amb un ampli ventall de funcionalitats. • Suport tècnic professional garantit pel proveïdor.	• Cost de llicències extremadament elevat. • Propietari, la qual cosa genera una forta dependència (vendor lock-in). • Complexitat elevada en la seva administració.

TAULA 7.2: Comparativa de Sistemes de Gestió de Bases de Dades.

7.2.4 Ecosistema Spring Cloud

Una arquitectura de microserveis requereix un conjunt d'eines per gestionar la comunicació, el descobriment i l'enrutament de serveis. Vaig optar per la suite de Spring Cloud per garantir una **màxima cohesió i compatibilitat**, ja que les seves eines estan dissenyades per funcionar conjuntament de manera nativa.

Spring Cloud Gateway

Spring Cloud Gateway actua com la porta d'enllaç (API Gateway) del sistema. La seva funció principal és ser l'únic punt d'entrada per a totes les peticions externes. Aquesta centralització és clau per a la seguretat i l'organització de l'arquitectura.

Justificació i ús en el projecte En aquest projecte, el Gateway s'encarrega de:

- **Enrutament dinàmic:** Dirigir cada petició al microservei corresponent (gestió de fitxers, usuaris, etc.) basant-se en el camí de l'URL.
- **Seguretat centralitzada:** Integrar-se amb Spring Security per aplicar un filtre d'autenticació a cada petició. Abans que una sol·licitud arribi a un servei intern, el Gateway valida el token JWT, garantint que només els usuaris autenticats tinguin accés. Això simplifica la lògica dels microserveis, que no necessiten implementar aquesta validació individualment.

L'elecció del Gateway va ser fonamental per crear una barrera de seguretat robusta i mantenir l'ordre en un sistema distribuït.

spring Eureka Eureka

Eureka és un servidor de descobriment de serveis (Service Discovery). En un entorn de microserveis, les instàncies poden aparèixer i desaparèixer dinàmicament. Eureka soluciona el problema de com un servei pot trobar la ubicació de xarxa (IP i port) d'un altre.

Justificació i ús en el projecte Cada microservei es registra a Eureka en arrencar. Quan un servei necessita comunicar-se amb un altre, consulta a Eureka per obtenir la seva ubicació actualitzada. Això aporta:

- **Resiliència i escalabilitat:** El sistema pot escalar horitzontalment o sobreviure a la caiguda d'una instància sense necessitat de reconfiguració manual.
- **Desacoblament:** Els serveis no necessiten conèixer les adreces físiques dels altres, simplificant la configuració i el desplegament.

Spring Cloud OpenFeign

OpenFeign és un client REST declaratiu que simplifica la comunicació entre serveis. Permet definir la comunicació amb una API REST remota simplement creant una interfície de Java i anotant-la.

Justificació i ús en el projecte En lloc d'escriure manualment el codi per a peticions HTTP, OpenFeign ho automatitza. S'utilitza, per exemple, quan el servei de gestió de fitxers necessita dades del servei d'usuaris. Això ofereix:

- **Codi més net i llegible:** La lògica de la petició HTTP queda abstracta darrere d'una simple interfície, reduint el codi repetitiu.
- **Integració nativa:** Es connecta automàticament amb Eureka per resoldre els noms dels serveis i realitzar balanceig de càrrega.

L'ús d'OpenFeign va accelerar el desenvolupament i va millorar la mantenibilitat del codi de comunicació interna.

7.2.5 Spring Security

Spring Security és un framework potent i altament personalitzable que proporciona funcionalitats d'autenticació i control d'accés per a aplicacions Java.

Justificació de l'elecció La implementació d'un sistema d'autenticació robust era un requisit no funcional crític, directament lligat al compliment del RGPD. Vaig escollit Spring Security per la seva maduresa i la seva integració completa amb Spring Boot. La seva implementació en el servei UserAuthentication, juntament amb el filtre del gateway, em va permetre configurar un flux d'autenticació segur basat en JWT, amb emmagatzematge de contrasenyes mitjançant l'algorisme BCrypt, garantint que les dades d'accés dels usuaris estiguin protegides en tot moment.

7.2.6 RabbitMQ

RabbitMQ és un intermediari de missatges (message broker) que implementa el protocol AMQP, dissenyat per gestionar la comunicació asíncrona entre diferents components d'un sistema.

Per a la comunicació asíncrona, vaig avaluar dos dels intermediaris de missatges més populars [**cloudamqp-rabbitmq-kafka**].

Sistema	Avantatges (Pros)	Inconvenients (Contres)
RabbitMQ	• Configuració i gestió relativament senzilles. • Gran flexibilitat en l'enrutament de missatges. • Ideal per a patrons de desacoblament i tasques en segon pla.	• Menor rendiment que Kafka en escenaris de volum de dades massiu. • L'emmagatzematge no està dissenyat per a una retenció de dades a llarg termini per defecte.
Apache Kafka	• Rendiment extremadament alt i alta escalabilitat. • Sistema de log distribuït i persistent, ideal per a *event sourcing*.	• Configuració, gestió i operació notablement més complexes. • Corba d'aprenentatge més pronunciada.

TAULA 7.3: Comparativa d'Intermediaris de Missatges.

Justificació de l'elecció L'elecció d'un intermediari de missatges es va centrar en resoldre un repte específic: la comunicació asíncrona per millorar la resiliència. Vaig

escollit **RabbitMQ** per la seva **simplicitat en la configuració i gestió**. El seu ús, per exemple, en l'eliminació de dades en cascada, no només millora l'experiència d'usuari, sinó que també garanteix la fiabilitat en el compliment de les sol·licituds d'eliminació de dades sota el RGPD, assegurant que l'operació es completi de manera fiable fins i tot si un servei falla temporalment.

L'objectiu no era aplicar la comunicació asíncrona a tot el sistema, sinó utilitzar-la de manera estratègica. Concretament, es va implementar en processos com l'eliminació de dades en cascada entre microserveis. Aquest enfocament permet que la petició principal de l'usuari (p. ex., eliminar un fitxer) es completi ràpidament sense quedar bloquejada esperant la neteja de dades dependents. A més, proporciona un mecanisme de reintent transparent: si un servei consumidor falla temporalment, el sistema te un mecanisme de reintent asíncron que garanteix la consistència final de les dades sense que l'usuari ho percebi ni hagi d'intervenir. Per aquest cas d'ús, la facilitat d'implementació de RabbitMQ era ideal.

7.3 Pila Tecnològica del Frontend Web

Per al client web, necessitava una solució moderna que permetés construir una interfície d'usuari interactiva i eficient.

7.3.1 React

React és una biblioteca de JavaScript per construir interfícies d'usuari, basada en un model de components reutilitzables i un flux de dades unidireccional. El seu ús del *Virtual DOM* optimitza les actualitzacions de la interfície i millora el rendiment en aplicacions complexes.

A l'hora de seleccionar la tecnologia per al frontend, vaig comparar les biblioteques i frameworks més rellevants del mercat.

Justificació de l'elecció L'elecció de React, tot i basar-se en una familiaritat prèvia, no va ser una decisió superficial. Més enllà de l'experiència inicial, vaig valorar positivament la seva **flexibilitat** i el seu **enfocament en el rendiment**. L'arquitectura basada en components i l'ús del Virtual DOM s'alineaven amb el meu objectiu de construir una interfície eficient i modular. A diferència d'Angular, que imposa una estructura més rígida, React em proporcionava la llibertat de seleccionar les millors eines per a cada necessitat específica (com React Query per a l'estat del servidor i Zustand per a l'estat global). A més, el seu vast ecosistema de llibreries i el gran suport de la comunitat em donava la confiança necessària per afrontar els reptes del projecte, sabent que disposaria de solucions provades per la indústria.

Framework/Biblioteca	Avantatges (Pros)	Inconvenients (Contres)
React	• Arquitectura basada en components que fomenta la reutilització. • Alt rendiment gràcies al Virtual DOM. • Ecosistema de llibreries enorme i gran suport de la comunitat. • Flexibilitat per triar les eines complementàries (p. ex., gestió d'estat, enrutament).	• És una biblioteca, no un framework complet, la qual cosa requereix integrar altres solucions. • La llibertat d'elecció pot portar a una major complexitat en la configuració inicial.
Angular	• Framework complet i robust amb solucions integrades ("out-of-the-box"). • Basat en TypeScript, que millora la mantenibilitat del codi. • Fort suport empresarial per part de Google.	• Corba d'aprenentatge més pronunciada i major verbositat. • Menys flexible a causa de la seva naturalesa més rígida i opinionada.
Vue	• Corba d'aprenentatge molt suau i excel·lent documentació. • Bon rendiment i flexibilitat.	• Comunitat i ecosistema més petits en comparació amb React. • Menys presència en grans aplicacions empresarials.

TAULA 7.4: Comparativa de tecnologies per al Frontend Web.

7.3.2 React Query (TanStack Query)

React Query és una llibreria per a la gestió de l'estat del servidor (server state) en aplicacions React. S'encarrega de la consulta, la gestió de memòria cache i la sincronització de dades amb fonts externes com una API.

Justificació de l'elecció Per a la comunicació amb el backend, vaig voler evitar la gestió manual de l'estat de les dades amb *hooks* com useState i useEffect, ja que pot portar a codi complex i propens a errors. Vaig triar React Query amb la intenció de disposar d'una solució més robusta i declarativa. El meu objectiu en adoptar-la era simplificar la lògica de peticions asíncrones, millorar l'experiència d'usuari amb estratègies de memòria cache intel·ligents i, en definitiva, mantenir la interfície sincronitzada amb el backend de manera eficient.

7.3.3 Zustand

Zustand és una llibreria minimalista per a la gestió d'estat global (client state) en React, basada en una API senzilla de hooks.

Justificació de l'elecció Per l'estat global de la interfície (elements no dependents del servidor, com la selecció d'arxius), vaig avaluar diverses opcions. Vaig descartar Redux per considerar-lo excessivament complex per a les meves necessitats. Vaig escollit Zustand perquè buscava una solució lleugera, ràpida i amb una corba d'aprenentatge mínima. La seva simplicitat i el seu enfocament basat en *hooks* em semblaven ideals per gestionar l'estat global necessari sense afegir una sobrecàrrega de configuració ni afectar negativament el rendiment de l'aplicació.

7.4 Pila Tecnològica del Client d'Escriptori

Per al client d'escriptori, el requisit principal era aconseguir una aplicació multiplataforma que fos nativa en rendiment i consum de recursos.

7.4.1 Tauri i Svelte

FIGURA 7.1: Logo tauri.

FIGURA 7.2: Logo svelte.

Tauri és un framework que permet construir aplicacions d'escriptori utilitzant tecnologies web per a la interfície i un backend natiu escrit en Rust. Svelte, per la seva banda, és un compilador que transforma components d'interfície en codi JavaScript imperatiu altament optimitzat.

Justificació de l'elecció La meva decisió clau en aquest àmbit va ser **prioritzar el rendiment i la lleugeresa**. Vaig descartar l'alternativa més estesa, Electron, a causa del seu conegut alt consum de memòria i la gran mida dels binaris que genera. Vaig escollit **Tauri** perquè la seva arquitectura, basada en la *WebView* nativa del sistema operatiu, em permetia crear una aplicació molt més eficient i amb una mida final significativament menor.

Una de les capacitats clau implementades a Rust és la monitorització del sistema d'arxius local mitjançant la llibreria `notify`. Aquesta eina permet detectar en temps real qualsevol canvi que es produeixi a la carpeta sincronitzada (creació, modificació o eliminació de fitxers i directoris). Quan es detecta un esdeveniment, el backend de Rust ho comunica a la interfície i inicia el procés de sincronització amb el servidor, garantint així que l'estat local i remot es mantinguin sempre consistents, una funcionalitat essencial per a una experiència d'usuari fluida i fiable.

De manera complementària, per a la interfície d'aquesta aplicació d'escriptori, vaig optar per **Svelte** en lloc de React. Atès que Svelte és un compilador, genera un codi final molt més petit i eficient, sense la sobrecàrrega d'un DOM virtual en temps d'execució. Aquesta combinació de Tauri i Svelte em sembla la idònia per assolir el meu objectiu d'una aplicació d'escriptori ràpida, lleugera i amb una experiència d'usuari fluida.

La decisió per aixo va tenir un cost que no preveia en la corva d'aprenentage de rust, que va ser mes gran del que preveia. Si be encara ara crec que va ser una bona decisió, no puc assegurar que en cas de tornar a començar el projecte el tornaria a triar.

7.5 Maquinari i infraestructura

Durant el desenvolupament del projecte s'ha utilitzat un equip amb les especificacions següents:

- **CPU**: Intel Core i7-1185G7 (11a generació) amb 4 nuclis (8 fils) a 3.00 GHz.
- **Memòria RAM**: 32 GB.
- **Sistema operatiu**: Ubuntu 22.04.4 LTS (Linux).

Pel desplegament es recomana executar-la en qualsevol servidor o màquina virtual amb mínim 2 nuclis de CPU, 4 GB de RAM i 50 GB d'emmagatzematge, amb accés a internet i un *reverse proxy* per gestionar els certificats SSL. El reverse proxy no és una obligatorietat tècnica perquè el projecte funcioni, però és la solució estàndard i recomanada per garantir la seguretat. No s'ha afegit al projecte la configuració d'un reverse proxy perquè el temps disponible no permetia abordar la recerca necessària per desplegar-ne un de manera segura. Es planteja com una millora futura, ja que per complir amb la normativa de protecció de dades (RGPD), que exigeix garantir la

confidencialitat de les dades personals en trànsit, és imprescindible implementar el xifratge SSL/TLS. L'ús d'un reverse proxy és la via més eficient i robusta per assolir aquest objectiu.

Les versions específiques recomanades per als serveis de tercers són **PostgreSQL 16 o superior** i **RabbitMQ 3.12 o superior**, ja que són les versions estables més recents amb les quals s'ha provat el sistema durant el desenvolupament. L'ús de versions anteriors podria causar incompatibilitats o comportaments inesperats en alguns microserveis.

Els requisits detallats estan al capítol 6.

7.6 Eines de disseny i estil (UI/UX)

Per al disseny es va optar per Tailwind CSS per agilitzar el desenvolupament. Les llibreries complementàries utilitzades són:

- **Radix UI**: Biblioteca de components d'interfície sense estils, de baix nivell i accessibles, que serveix com a base per construir un sistema de disseny personalitzat.
- **Remix Icon i React Icons**: Col·leccions d'icones SVG de codi obert, fàcilment integrables en projectes React per millorar la usabilitat visual.
- **Tailwind Merge, clsx, Class Variance Authority**: Utilitats per gestionar i fusionar classes de Tailwind CSS de manera intel·ligent, evitant conflictes d'estils i simplificant la lògica de variants en components.
- **Dnd-kit/core**: Conjunt d'eines lleuger i modular per crear funcionalitats d'arrossegar i deixar anar ('drag-and-drop') accessibles i performants a React.
- **Selecto.js**: Llibreria per seleccionar elements mitjançant el ratolí o el tacte, utilitzada per implementar la selecció múltiple d'arxius i carpetes a la interfície web.

7.7 Cadena d'eines i DevOps

S'han utilitzat les eines següents per a la construcció i el desplegament:

- **Backend**: Maven 3.9.x (Maven Wrapper).
- **Frontend**: Node.js v20.14.0 amb pnpm.
- **Contenidors**: Docker Engine 27.0.3 i Docker Compose v2.28.1.

Per a la orquestració dels contenidors del projecte, s'utilitza un fitxer `docker-compose.yml`. Es va optar per aquesta solució per la seva simplicitat, ja que satisfà les necessitats del sistema amb una configuració senzilla que es pot executar en qualsevol ordinador amb un script, basant-se en el coneixement ja adquirit sobre la tecnologia.

Com a treball a futur, s'ha previst afegir fitxers de configuració de Helm que permetrien instal·lar el sistema en una infraestructura Kubernetes per a usuaris més avançats que ho puguin requerir, oferint més escalabilitat i robustesa.

El projecte és a GitHub, la qual cosa facilita aquesta futura integració. A més, en ser un repositori públic, facilita la col·laboració amb altres desenvolupadors i, tal com mostren les estadístiques, és l'eina més popular per a la gestió de projectes de codi obert [**kinsta˙github˙stats**].

7.8 Llicències i compliment legal

La llicència escollida és MIT per màxima flexibilitat:

Llicència	Ús comercial	Publicar derivats	Compatibilitat propietària
MIT (escollida)	Sí	No	Molt alta
Apache-2.0	Sí	Parcial	Alta
GPL-3.0	Sí	Obligatori	Baixa
LGPL-3.0	Sí	Parcial	Mitjana

TAULA 7.5: Comparativa abreujada de llicències

7.9 Traçabilitat global amb els requisits

La Taula 7.6 resumeix la traçabilitat entre els criteris de decisió, la tecnologia escollida i els requisits que cobreix cadascuna.

Criteri Inicial	Tecnologia Escollida	Requisit cobert
Experiència prèvia	Spring Boot, Maven, React	RF-1 a RF-10, RNF-Rendiment, RNF-Mantenibilitat
Rendiment i eficiència	Tauri, Svelte, Tailwind	RNF-Rendiment, RNF-Compatibilitat, RNF-Usabilitat
Cohesió ecosistema	Spring Cloud, Docker Compose	RNF-Escalabilitat, RNF-Portabilitat
Capacitats tècniques	RabbitMQ, PostgreSQL	RF-Gestió dades, RNF-Escalabilitat
Codi obert	GitHub, MIT License	Viabilitat econòmica, Legalitat

TAULA 7.6: Traçabilitat global criteris-tecnologia-requisits

Capítol 8

Anàlisi i disseny del sistema

8.1 Introducció

Aquest capítol presenta l'anàlisi funcional i estructural del sistema desenvolupat, així com les decisions arquitectòniques preses durant la seva fase de disseny. Es descriuen els components principals de l'arquitectura de microserveis, els fluxos de dades, els casos d'ús més rellevants i els esbossos inicials de la interfície d'usuari, tant pel client web com per al client d'escriptori.

8.2 Anàlisi funcional

8.2.1 Diagrama de casos d'ús

A continuació es mostra el diagrama general de casos d'ús, que resumeix la interacció dels actors principals (Usuari i Administrador) amb les funcionalitats clau del sistema.

FIGURA 8.1: Diagrama de casos d'ús

Tal com mostra el diagrama, el sistema té diferents tipus d'usuaris (actors). El punt de partida és el Client Tauri l'aplicació d'escriptori pensada per a una integració total amb el sistema de fitxers local. L'Usuari web hereta d'aquest les funcions bàsiques de gestió d'arxius, com pujar-ne, descarregar-ne o compartir-los, però operant des del navegador.

El que els diferencia és la sincronització. Tot i que tots dos clients en tenen, la seva funció és distinta. El client Tauri busca la rèplica de fitxers, és a dir, que una carpeta local sigui un mirall del que hi ha al servidor. En canvi, a l'aplicació web la sincronització s'utilitza per refrescar l'estat de la interfície. D'aquesta manera, si un altre usuari, o el mateix usuari en un altre instancia de l'aplicació, fa un canvi, aquest es reflecteix a la pantalla al moment, sense haver de recarregar la pàgina. Cal dir que, tot i que el disseny inicial preveia la sincronització d'arxius compartits a Tauri, aquesta funció no es va poder implementar per falta de temps, com ja s'explica al capítol 4.

Per sobre del usuari normal, tenim l'Administrador. Aquest rol, a part de fer el mateix que un usuari normal, té la capacitat de gestionar altres comptes d'usuari: pot consultar la llista, modificar-ne les dades o directament eliminar-los.

Finalment, tenim el Superadministrador. És el rol amb més privilegis, ja que fa tot el que fan els altres i, a més, pot gestionar els comptes dels propis administradors, i

tambe la capacitat de modificar les contrasenyes d'altres usuaris (independentment del seu rol) o modificar el rol d'un altre usuari. Això garanteix un control absolut sobre el sistema.

El disseny d'aquesta jerarquia respon a una filosofia de control centralitzat. La figura del Superadministrador es concep com un rol únic, amb autoritat sobre tot el sistema, que es crea exclusivament durant el procés d'instal·lació mitjançant scripts, no des de l'aplicació. Això garanteix un punt de control segur. A partir d'aquí, el Superadministrador pot delegar funcions nomenant altres Administradors, creant una estructura piramidal. No obstant això, les accions més sensibles —com modificar contrasenyes alienes o canviar rols— queden reservades per a ell. Aquesta concentració de poder en una única figura el converteix en el màxim responsable de la seguretat i la integritat de la plataforma. Al ser un rol generat durant el procés d'instal·lació, també s'asumeig que sera el gestor principal de la aplicacio (qui inicia el procès d'instal·lació) qui asumira aquest rol al ser tambe el major responsable del manteniment de les dades y la seguretat del sistema.

8.2.2 Actors i mòduls

Els principals actors identificats en el sistema, ordenats per jerarquia, són:

- **Client Tauri**: Representa l'usuari de l'aplicació d'escriptori. Té accés a totes les funcionalitats de gestió de fitxers i la capacitat de sincronització automàtica amb una carpeta local.
- **Usuari web**: Usuari que interactua amb el sistema a través de la interfície web. Pot gestionar els seus arxius, compartir-los i configurar el seu compte.
- **Administrador**: A més de les capacitats d'un usuari normal, pot gestionar altres usuaris, incloent la modificació de dades i l'eliminació de comptes.
- **Superadministrador**: El rol amb màxims privilegis. Té control total sobre la plataforma, incloent la gestió de comptes d'administradors i la capacitat de restablir qualsevol contrasenya.

El sistema es divideix funcionalment en els següents mòduls:

- Autenticació i gestió d'usuaris
- Gestió d'arxius i carpetes
- Control d'accés
- Compartició d'arxius
- Sincronització en temps real
- Gestió de la paperera

8.2.3 Casos d'ús principals

A continuació es descriuen diversos casos d'ús extrets de les fitxes funcionals, detallant la interacció entre els principals serveis implicats:

UC-01: Registrar-se

- **Descripció**: L'usuari crea un compte nou.
- **Actors**: Usuari.
- **Precondicions**: No haver iniciat sessió.
- **Postcondicions**: Compte creat i sessió iniciada.
- **Escenari principal**:
  1. El client envia una sol·licitud de registre al servei UserAuthentication a través del Gateway.
  2. El servei valida les dades rebudes:
     - Comprova que el format del nom d'usuari, contrasenya i email siguin correctes.
     - Verifica que el nom d'usuari no estigui ja en ús.
     - Consulta a UserManagement per assegurar que l'email no estigui registrat.
  3. Si les validacions són correctes, UserAuthentication guarda les credencials (nom d'usuari i contrasenya xifrada) i inicia la creació de dades en altres serveis:
     - Fa una crida aUserManagement per guardar la informació personal de l'usuari (email, nom i cognoms).
     - Crida aFileManagement per crear la carpeta arrel de l'usuari.
     - Finalment, envia un missatge asíncron a través de RabbitMQ aSyncService per generar l'estat inicial de sincronització (*snapshot*) amb la carpeta arrel.
  4. Un cop finalitzat el procés,UserAuthentication retorna els tokens d'accés i de refresc per iniciar la sessió automàticament.

UC-02: Iniciar sessió

- **Descripció**: L'usuari inicia sessió amb el seu nom i contrasenya.
- **Actors**: Usuari.
- **Precondicions**: Tenir un compte vàlid.
- **Postcondicions**: Sessió activa.
- **Escenari principal**:
  1. El client envia el nom d'usuari i la contrasenya al serveiUserAuthentication.

2. El servei busca l'usuari a la base de dades a partir del seu nom.
3. Si l'usuari existeix, compara la contrasenya rebuda amb la versió xifrada emmagatzemada.
4. Si les credencials són correctes, genera un nou token d'accés (JWT) de curta durada i un token de refresc de llarga durada.
5. Finalment, retorna els dos tokens al client per iniciar la sessió i mantenir-la activa.

UC-08: Pujar un arxiu

- **Descripció**: Pujada de fitxers o creació de carpetes.
- **Actors**: Usuari.
- **Precondicions**: Permís d'escriptura a la carpeta de destinació.
- **Postcondicions**: Nou element emmagatzemat i notificat.
- **Escenari principal**:
  1. El Gateway valida el token JWT i reenvia la petició a **FileManagement**.
  2. El servei consulta a **FileAccessControl** per verificar que l'usuari té permís d'escriptura (WRITE) a la carpeta de destinació.
  3. Si el permís és correcte, crea les metadades de l'arxiu (nom, mida, etc.) a la seva base de dades.
  4. Emmagatzema el contingut del fitxer al sistema d'arxius del servidor, utilitzant un ID únic com a nom.
  5. Sol·licita a **FileAccessControl** que assigni el permís de propietari (ADMIN) sobre el nou element a l'usuari que l'ha pujat.
  6. Finalment, envia un missatge asíncron a **SyncService** per notificar la creació i actualitzar els clients.

UC-10: Descarregar un arxiu

- **Descripció**: Obtenir el contingut d'un arxiu o carpeta.
- **Actors**: Usuari.
- **Precondicions**: Permís de lectura sobre l'element sol·licitat.
- **Postcondicions**: Arxiu descarregat pel client.
- **Escenari principal**:
  1. El Gateway valida el JWT i reenvia la petició a **FileManagement**.

2. Aquest consulta **FileAccessControl** per confirmar que l'usuari té permís de lectura (READ).

3. Si els permisos són correctes, recupera el fitxer del sistema d'emmagatzematge. Si és una carpeta, la comprimeix en format ZIP.

4. Retorna el contingut com un flux de dades (*stream*) perquè el client iniciï la descàrrega.

UC-11: Moure un arxiu a la paperera

- **Descripció**: Moure elements a la paperera.
- **Actors**: Usuari.
- **Precondicions**: Permís d'escriptura sobre l'element.
- **Postcondicions**: Element marcat com a eliminat i visible a la paperera.
- **Escenari principal**:

  1. El Gateway valida el JWT i envia la petició a **TrashService**.

  2. **TrashService** verifica a **FileAccessControl** que l'usuari té permís d'escriptura.

  3. Crida a **FileManagement** perquè marqui l'element i els seus descendents com a eliminats (sense esborrar-los físicament).

  4. Crea un registre a la seva pròpia base de dades (TrashRecord) per cada element mogut, emmagatzemant la data d'eliminació i de caducitat.

  5. **FileManagement** notifica a **SyncService** el canvi d'estat per actualitzar els clients.

UC-12: Eliminar definitivament

- **Descripció**: Esborrar definitivament un element de la paperera.
- **Actors**: Usuari.
- **Precondicions**: Element a la paperera i ser-ne el propietari.
- **Postcondicions**: Element eliminat de forma permanent.
- **Escenari principal**:

  1. El Gateway envia la petició a **TrashService**.

  2. El servei verifica que l'usuari és el propietari de l'element.

  3. Crida a **FileManagement** per esborrar l'arxiu físic i les seves metadades.

4. Canvia l'estat del TrashRecord a PENDING_DELETION i inicia un procés de purga asíncron.
5. A través de missatges per cua, ordena a **FileAccessControl** i **FileSharing** que eliminin totes les regles associades a l'element.
6. Un cop confirmat per tots els serveis, el TrashRecord s'esborra.
7. Es notifica a **SyncService** per eliminar les còpies locals de l'element.

UC-13: Compartir arxiu

- **Descripció**: Concedir o revocar accessos sobre elements a altres usuaris.
- **Actors**: Usuari.
- **Precondicions**: Permís de propietari o d'administrador sobre l'element.
- **Postcondicions**: Els usuaris seleccionats obtenen o perden l'accés indicat.
- **Escenari principal**:
  1. El Gateway valida el JWT i passa la petició a **FileSharing**.
  2. El servei verifica a **FileAccessControl** que el sol·licitant és el propietari (ADMIN) de l'element.
  3. Consulta a **UserManagement** per obtenir l'ID de l'usuari amb qui es vol compartir.
  4. Sol·licita a **FileAccessControl** que creï una nova regla d'accés (lectura o escriptura) per a l'usuari convidat sobre l'element i els seus descendents (si es una carpeta).
  5. Desa un registre de la compartició a la seva base de dades.
  6. Notifica a **SyncService** a través de RabbitMQ per propagar els canvis als clients implicats.

(*La resta de fitxes completes es poden consultar a l'Apèndix* **??**)

8.2.4 Diagrames d'activitat dels Casos d'Ús Principals

A continuació, es presenten els diagrames d'activitat per als casos d'ús més representatius del sistema. Cada diagrama il·lustra el flux de treball, les decisions i les interaccions entre serveis.

UC-01: Registrar-se

El flux comença quan l'usuari envia el formulari de registre. El Gateway reenvia la petició a UserAuthentication, que valida el format de les dades, comprova que el nom d'usuari no existeixi i consulta a UserManagement per assegurar que l'email tampoc estigui en ús. Si tot és correcte, orquestra la creació de l'usuari: guarda credencials, demana a UserManagement que creï el perfil, a FileManagement que

generi la carpeta arrel i notifica a SyncService via RabbitMQ. Finalment, retorna els tokens per iniciar la sessió.

FIGURA 8.2: Diagrama d'activitat per al cas d'ús UC-01: Registrar-se.

UC-02: Iniciar sessió

L'usuari envia les seves credencials, que el Gateway reenvia a UserAuthentication. El servei busca l'usuari i, si existeix, verifica la contrasenya. Si les credencials són correctes, genera i retorna un nou joc de tokens (accés i refresc) per activar la sessió del client.

FIGURA 8.3: Diagrama d'activitat per al cas d'ús UC-02: Iniciar sessió.

UC-08: Crear o pujar arxius

FileManagement rep la petició i consulta a FileAccessControl si l'usuari té permís d'escriptura. Si és així, crea les metadades, emmagatzema el fitxer (si escau), demana a FileAccessControl que assigni el permís de propietari i finalment notifica SyncService del canvi.

FIGURA 8.4: Diagrama d'activitat per al cas d'ús UC-08: Crear o pujar arxius.

UC-10: Descarregar

Després de verificar el permís de lectura a FileAccessControl, FileManagement recupera el fitxer del sistema d'emmagatzematge (o el comprimeix si és una carpeta) i el retorna al client com un flux de dades.

FIGURA 8.5: Diagrama d'activitat per al cas d'ús UC-10: Descarregar.

UC-11: Enviar a la paperera

TrashService rep la petició, verifica el permís d'escriptura a FileAccessControl i demana a FileManagement que marqui l'element com a eliminat. Finalment, crea un registre a la seva pròpia base de dades per gestionar la caducitat de l'element.

FIGURA 8.6: Diagrama d'activitat per al cas d'ús UC-11: Enviar a la paperera.

UC-12: Eliminar permanentment

Quan un usuari sol·licita l'eliminació permanent, TrashService verifica que n'és el propietari. Si ho és, inicia la saga d'eliminació enviant missatges a la cua perquè FileManagement, FileAccessControl i FileSharing purguin totes les dades associades de forma asíncrona.

FIGURA 8.7: Diagrama d'activitat per al cas d'ús UC-12: Eliminar permanentment.

UC-13: Compartir arxius

El servei FileSharing comprova que el sol·licitant és el propietari de l'element. Després, obté l'ID de l'usuari convidat de UserManagement i demana a FileAccessControl que creï la nova regla d'accés. Finalment, desa un registre de la compartició i notifica SyncService.

FIGURA 8.8: Diagrama d'activitat per al cas d'ús UC-13: Compartir arxius.

(*La resta de diagrames d'activitat detallats per a cada cas d'ús es poden consultar a l'Apèndix* **??**).

8.2.5 Matriu de traçabilitat entre requisits i casos d'ús

Per garantir que tots els requisits funcionals descrits al Capítol 6 estan coberts per la funcionalitat del sistema, es presenta a la Taula 8.1 una matriu de traçabilitat detallada. Aquesta relaciona cada requisit funcional amb els casos d'ús individuals (detallats a l'Apèndix **??**) que l'implementen.

TAULA 8.1: Matriu de traçabilitat detallada entre requisits funcionals i casos d'ús.

Requisit Funcional	UC-01	UC-02	UC-03	UC-04	UC-05	UC-06	UC-07	UC-08	UC-09A	UC-09B	UC-09C	UC-10	UC-11	UC-12	UC-13	UC-13A	UC-13B	UC-14	UC-15	UC-16	UC-17	UC-18	UC-19	UC-20	UC-21	UC-22
1. Gestió d'usuaris	X	X	X																X	X						
2. Gestió d'arxius								X	X	X	X	X	X		X	X	X				X					
3. Sistema de paperera													X	X							X					
4. Compartició d'arxius															X	X	X									
5. Sincronització en temps real																		X								X
6. Panell d'administració				X	X	X	X															X	X	X	X	

8.3 Arquitectura del sistema

8.3.1 Visió general i components

El sistema segueix una arquitectura de microserveis, dissenyada per garantir l'escalabilitat, la resiliència i la mantenibilitat. La figura 8.9 mostra una visió general d'aquesta arquitectura, els components principals de la qual es descriuen a continuació.

- **Gateway**: Actua com a punt d'entrada únic (*Single Point of Entry*) per a totes les sol·licituds dels clients. La seva funció és enrutar les peticions REST a l'API cap al microservei corresponent i gestionar les connexions WebSocket per a les actualitzacions en temps real, dirigint-les exclusivament al **SyncService**. Aquesta capa d'abstracció simplifica la comunicació des del client i centralitza la gestió de l'autenticació i el control d'accés inicial.

- **Core Services**: Constitueixen el nucli funcional de l'aplicació. Estan agrupats en dos paquets lògics:

  - **User & Auth**: Conté els serveis responsables de l'autenticació (UserAuthentication) i la gestió de les dades dels usuaris (UserManagement).

  - **File System**: Agrupa tots els serveis relacionats amb la gestió d'arxius, incloent la manipulació de metadades (FileManagement), el control d'accés (FileAccessControl), la compartició (FileSharing), la gestió de la paperera (Trash) i la sincronització en temps real (SyncService).

- **Components d'Infraestructura**:

  - **Eureka Server**: Actua com a registre de serveis. Cada microservei es registra a Eureka en iniciar-se, la qual cosa permet el descobriment dinàmic de serveis dins de la xarxa interna.

  - **PostgreSQL Database**: És el sistema de gestió de bases de dades relacional on tots els serveis principals persisteixen les seves dades. Encara que comparteixen la mateixa instància de base de dades, cada servei opera sobre el seu propi esquema per mantenir un acoblament baix.

- **Comunicació Asíncrona (RabbitMQ)**: Per a operacions que requereixen un alt grau de desacoblament o que són de llarga durada, s'utilitza una cua de missatges amb RabbitMQ. Els principals fluxos asíncrons són:

  - **Sincronització en Temps Real**: Serveis com FileManagement i FileSharing publiquen esdeveniments (p. ex., creació o modificació d'un arxiu). SyncService consumeix aquests esdeveniments per notificar els clients connectats via WebSocket.

  - **Eliminació d'Usuari (Fan-out)**: Quan s'inicia l'eliminació d'un usuari, es publica un únic missatge que és rebut per tots els serveis. Aquest patró (*fan-out*) assegura que cada servei pugui purgar de forma independent totes les dades associades a l'usuari eliminat.

FIGURA 8.9: Diagrama de components de l'arquitectura del backend.

– **Eliminació Permanent (Saga)**: El servei Trash orquestra l'eliminació definitiva d'un fitxer mitjançant una saga. Aquest patró s'utilitza perquè l'eliminació no és una acció atòmica, sinó una transacció distribuïda que ha d'executar-se de forma coordinada en diversos serveis. La saga, implementada mitjançant missatges, garanteix que l'operació es completi de forma resilient: Trash publica un missatge que és consumit pels serveis del paquet *File System* per garantir que es netegen totes les dades relacionades (metadades, permisos i comparticions) de forma eventualment consistent.

Aquesta estructura modular permet un desenvolupament i desplegament independents de cada component. A continuació, es detalla el disseny específic de cada microservei, incloent les seves responsabilitats, el seu diagrama de classes i l'esquema de la seva base de dades, per proporcionar una visió completa del seu funcionament intern.

8.3.2 Disseny dels microserveis

UserAuthentication

Aquest servei és el responsable de gestionar les credencials dels usuaris (nom d'usuari, contrasenya) i els seus rols. Centralitza els processos de registre, inici de sessió i validació de tokens JWT.

FIGURA 8.10: Diagrama de classes del servei UserAuthentication.

FIGURA 8.11: Diagrama de la base de dades del servei UserAuthentication.

Per optimitzar el rendiment de les consultes, s'ha creat un índex a la columna email de la taula user_info. Aquest índex és crucial per accelerar la comprovació d'existència de correus durant el registre d'usuaris i per a les cerques ràpides basades en l'adreça de correu electrònic.

UserManagement

Gestiona la informació personal dels usuaris, com el correu electrònic, el nom i els cognoms. Col·labora estretament amb UserAuthentication durant el registre i proporciona funcionalitats per a la cerca i gestió d'usuaris per part dels administradors.

FIGURA 8.12: Diagrama de classes del servei UserManagement.

FIGURA 8.13: Diagrama de la base de dades del servei UserManagement.

Per optimitzar el rendiment de les consultes, s'ha creat un índex a la columna email de la taula user_info. Aquest índex és crucial per accelerar la comprovació d'existència de correus durant el registre d'usuaris i per a les cerques ràpides basades en l'adreça de correu electrònic.

FileManagement

És el nucli del sistema de gestió de fitxers. S'encarrega de les metadades dels arxius i carpetes (nom, mida, dates) i de la seva ubicació física al servidor. Processa operacions com la creació, pujada, descàrrega, renombrat i moviment d'elements.

FIGURA 8.14: Diagrama de classes del servei FileManagement.

FIGURA 8.15: Diagrama de la base de dades del servei FileManagement.

Per garantir un rendiment òptim, s'han definit diversos índexs. A les taules file_entity i folder_entity, s'han indexat les columnes user_id i parent_id. L'índex sobre user_id accelera la càrrega inicial dels arxius d'un usuari, mentre que l'índex sobre parent_id és fonamental per llistar de forma ràpida el contingut d'una carpeta, una de les operacions més freqüents del sistema.

FileAccessControl

Aquest servei actua com a autoritat central per a la gestió de permisos. Emmagatzema les regles que defineixen quin usuari té quin tipus d'accés (lectura, escriptura, propietari) sobre cada fitxer o carpeta. És consultat per altres serveis abans de realitzar qualsevol operació crítica.

FIGURA 8.16: Diagrama de classes del servei FileAccessControl.

FIGURA 8.17: Diagrama de la base de dades del servei FileAccess-Control.

La taula access_rule està fortament indexada per les columnes user_id i element_id. Aquests índexs són essencials per a la funció principal del servei: verificar de manera quasi instantània si un usuari concret té permís sobre un element específic, una consulta que es realitza abans de la majoria d'operacions sobre fitxers.

FileSharing

Gestiona la lògica de compartició d'arxius entre usuaris. Emmagatzema registres de qui ha compartit què i amb qui, i col·labora amb FileAccessControl per aplicar els permisos corresponents als usuaris convidats.

FIGURA 8.18: Diagrama de classes del servei FileSharing.

FIGURA 8.19: Diagrama de la base de dades del servei FileSharing.

Per optimitzar les consultes relacionades amb la compartició, s'han creat índexs a les columnes user_id i element_id de la taula shared_access. Aquests permeten recuperar ràpidament tots els elements compartits amb un usuari o, inversament, tots els usuaris amb qui s'ha compartit un element.

Trash

Implementa la funcionalitat de la paperera de reciclatge. Quan un usuari elimina un element, aquest servei el marca com a "eliminatï emmagatzema un registre amb la data de caducitat. També orquestra el procés d'eliminació permanent (la saga descrita anteriorment).

FIGURA 8.20: Diagrama de classes del servei Trash.

FIGURA 8.21: Diagrama de la base de dades del servei Trash.

El rendiment de la paperera es millora amb índexs a les columnes user_id i element_id. El primer accelera la càrrega de la vista de la paperera per a un usuari, mentre que el segon permet localitzar de forma eficient el registre d'un element concret quan es vol restaurar o eliminar.

SyncService

És el responsable de les actualitzacions en temps real. Manté una connexió WebSocket amb els clients actius i consumeix esdeveniments de RabbitMQ per notificar canvis en l'estructura de fitxers, mantenint així les interfícies d'usuari sincronitzades.

FIGURA 8.22: Diagrama de classes del servei SyncService.

FIGURA 8.23: Diagrama de la base de dades del servei SyncService.

Per assegurar la rapidesa en les operacions de sincronització, s'han establert índexs clau. A la taula snapshot_entity, s'indexa user_id per localitzar ràpidament l'estat de sincronització d'un usuari. A la taula snapshot_element_entity, s'indexen snapshot_id per carregar tots els elements d'una sincronització i element_id per trobar un fitxer o carpeta específic dins de l'estructura de sincronització.

8.3.3 Patrons de disseny i principis arquitectònics

El desenvolupament del backend s'ha guiat per un conjunt de patrons de disseny i principis arquitectònics orientats a garantir un codi net, mantenible, escalable i desacoblat. L'ús del framework Spring facilita l'adopció d'aquests patrons, especialment a través del seu contenidor d'Inversió de Control (IoC) i la injecció de dependències.

- **Arquitectura per Capes**: Tots els microserveis segueixen una arquitectura de tres capes ben diferenciades (Controlador, Servei i Repositori). Aquesta separació de responsabilitats és fonamental:

  - La **Capa de Controlador** gestiona les peticions HTTP i actua com a façana de l'API.
  - La **Capa de Servei** conté la lògica de negoci del domini.
  - La **Capa de Repositori** abstrau l'accés a les dades.

  Aquesta estructura compleix el **Principi de Responsabilitat Única (SRP)** de SOLID, ja que cada capa té un propòsit clar i aïllat, la qual cosa millora la cohesió i redueix l'acoblament.

- **Injecció de Dependències i Inversió de Control (IoC)**: En lloc que els components creïn les seves pròpies dependències, el contenidor de Spring les "injecta" automàticament. Per exemple, un servei no crea la seva instància de repositori, sinó que la declara com una dependència. Això compleix el **Principi d'Inversió de Dependències (DIP)** de SOLID, ja que els mòduls d'alt nivell (serveis) depenen d'abstraccions (interfícies de repositori) en lloc d'implementacions concretes. El resultat és un sistema molt més modular i fàcil de provar, ja que les dependències es poden substituir per simulacres (*mocks*) en els tests unitaris.

- **Patró Repository**: Implementat a través de Spring Data JPA, aquest patró desacobla la lògica de negoci de la tecnologia de persistència de dades. Els serveis interactuen amb una interfície (p. ex., UserRepository) sense conèixer els detalls de la base de dades subjacents. Això permetria, per exemple, canviar de PostgreSQL a una altra base de dades SQL amb un impacte mínim en el codi de l'aplicació.

- **Data Transfer Object (DTO)**: El sistema utilitza objectes específics per a les peticions ('...Request') i respostes ('...Response') de l'API. Aquest patró desacobla el model de dades intern (entitats JPA) del model exposat públicament. Això no només és una bona pràctica per a la seguretat, evitant l'exposició excessiva de dades, sinó que també permet que l'API evolucioni de forma independent al model de la base de dades.

A més d'aquests principis generals, en el codi dels microserveis s'han implementat patrons de disseny específics per resoldre problemes concrets del domini:

- **Patró Composite**: En el servei FileManagement, el model de dades de carpetes (FolderEntity), que contenen una llista d'altres carpetes filles, implementa aquest patró de manera natural. Això permet tractar de manera uniforme i recursiva les estructures de directoris, simplificant operacions com moure o eliminar una carpeta amb tot el seu contingut. L'ús d'una ElementEntity associada a cada fitxer i carpeta permet, a més, que la resta del sistema pugui referenciar-los de manera polimòrfica a través d'un ID comú.

- **Patró State**: El servei Trash utilitza una implementació d'aquest patró. L'entitat TrashRecord té un camp d'estat (RecordStatus) que determina el seu comportament: un element només es pot restaurar si el seu estat és ACTIVE, pero no si està PENDING_DELETION.

- **Patró Soft Delete**: Implementat al servei TrashService, aquest patró evita l'eliminació física immediata dels arxius quan un usuari els elimina. En lloc

d'esborrar definitivament un element del sistema de fitxers, es marca com a "eliminat"mitjançant el mètode setElementDeletedState i es crea un registre TrashRecord que inclou la data d'eliminació i una data de caducitat. Aquesta aproximació millora significativament l'experiència d'usuari en permetre la recuperació d'eliminacions accidentals, i garanteix que les dades estiguin disponibles per a processos de neteja programats.

- **Patró Saga**: Utilitzat per orquestrar transaccions distribuïdes complexes que no poden ser atòmiques, especialment en l'eliminació permanent d'elements (UC-12) i l'eliminació completa d'usuaris (UC-07). El TrashService actua com a coordinador, enviant missatges asíncrons a través de RabbitMQ a múltiples microserveis (FileManagement, FileAccessControl, FileSharing) perquè cadascun executi la seva part de l'operació. Aquest patró és fonamental per garantir la consistència eventual en un sistema distribuït, ja que proporciona tolerància a fallades mitjançant mecanismes de reintent i confirmació, assegurant que totes les operacions es completin de manera fiable sense bloquejar el sistema.

- **Patró Factory Method (simplificat)**: A UserAuthentication, el mètode generateToken actua com una fàbrica. El client sol·licita un token, i el mètode, basant-se en un paràmetre, decideix si ha de crear un token d'accés (de curta durada) o un de refresc (de llarga durada), encapsulant-ne la lògica de creació.

- **Patró d'Agregació**: El disseny de les dades d'usuari segueix aquest principi. En lloc de tenir una entitat monolítica, la informació s'ha separat en dos microserveis:

  - UserAuthentication: Gestiona les dades crítiques de seguretat (credencials, rol).

  - UserManagement: Gestiona la informació personal (nom, correu electrònic).

  Aquesta agregació, connectada per un userId comú, permet tractar l'usuari com una unitat lògica alhora que es mantenen les seves dades desacoblades i segures.

8.3.4 Justificació de l'arquitectura de microserveis

L'elecció d'una arquitectura de microserveis per a aquest projecte no va ser una decisió trivial, sinó una resposta estratègica als requisits funcionals i no funcionals del sistema, com l'escalabilitat, la resiliència i la mantenibilitat a llarg termini. A continuació, es justifiquen els motius principals d'aquesta elecció enfront d'una alternativa monolítica:

- **Escalabilitat independent**: En una aplicació de gestió de fitxers, no tots els components tenen la mateixa càrrega de treball. Per exemple, el servei FileManagement (operacions de fitxers) i el SyncService (connexions WebSocket) són susceptibles de rebre una càrrega molt més alta que el UserManagement. L'arquitectura de microserveis permet escalar horitzontalment només aquells serveis que ho necessiten, optimitzant l'ús de recursos sense haver de replicar tota l'aplicació.

- **Mantenibilitat i Cohesió**: El sistema es descompon en serveis petits i cohesionats, cadascun amb una única responsabilitat ben definida (p. ex., autenticació,

gestió d'arxius, paperera). Aquesta separació facilita enormement el desenvolupament i el manteniment. Un desenvolupador pot treballar en el servei de compartició (FileSharing) sense necessitat de comprendre les complexitats internes del sistema de sincronització, la qual cosa redueix la càrrega cognitiva i accelera el cicle de desenvolupament.

- **Aïllament de fallades i resiliència**: En un sistema monolític, un error no controlat en una part del codi pot provocar la caiguda de tota l'aplicació. En canvi, amb microserveis, una fallada en un servei no crític (com podria ser el TrashService) no hauria d'afectar el funcionament de la resta del sistema, com l'autenticació o la gestió de fitxers. L'ús d'un registre de serveis com Eureka contribueix a aquesta resiliència, permetent que els serveis es descobreixin i es comuniquin de manera dinàmica fins i tot si algunes instàncies fallen.

- **Flexibilitat tecnològica**: Tot i que actualment tots els serveis estan desenvolupats amb Spring Boot, l'arquitectura de microserveis ofereix la llibertat d'adoptar diferents tecnologies per a diferents serveis en el futur. Si, per exemple, es descobrís que un servei de processament de fitxers requereix un llenguatge optimitzat per a computació intensiva, es podria desenvolupar i integrar sense afectar la resta de l'ecosistema tecnològic.

- **Desplegament independent i agilitat**: Cada microservei es pot desplegar de forma autònoma. Això significa que una actualització al servei d'usuaris (UserManagement) es pot llançar a producció sense necessitat de tornar a provar i desplegar tot el sistema. Aquest cicle de desplegament més ràpid i menys arriscat augmenta l'agilitat del projecte i facilita la integració contínua.

En conclusió, l'arquitectura de microserveis proporciona la base per a un sistema robust, flexible i preparat per créixer, alineant-se perfectament amb els objectius d'un servei al núvol modern i escalable.

8.4 Disseny de les interfícies d'usuari

8.4.1 Client web

El disseny de la interfície d'usuari per al client web s'ha centrat a oferir una experiència intuïtiva i funcional, semblant a la d'altres serveis d'emmagatzematge al núvol. A continuació es descriuen les principals vistes i components.

Autenticació

El primer contacte de l'usuari amb l'aplicació són els formularis d'autenticació. Es presenta una finestra per a l'inici de sessió (Figura 8.24) i una altra per al registre de nous usuaris (Figura 8.25), amb validacions clares per a cada camp.

FIGURA 8.24: Pantalla d'inici de sessió.

FIGURA 8.25: Pantalla de registre.

Escriptori principal

Un cop autenticat, l'usuari accedeix a l'escriptori principal (Figura 8.26), que és el nucli de l'aplicació. Aquesta vista es compon de diversos elements clau.

FIGURA 8.26: Escriptori principal del client web.

Navegació i accions principals A la part esquerra, un arbre de carpetes (Figura 8.27) permet navegar per l'estructura de directoris. En fer clic a una carpeta, s'accedeix al seu contingut i el seu estil canvia per reflectir que és la seleccionada. L'arbre inclou seccions per als arxius propis, els compartits i la paperera. La vista de contingut es pot filtrar per tipus d'element (fitxers o carpetes) mitjançant dos botons dedicats. Just a sobre d'aquest arbre, es troben els botons d'acció contextuals a la

carpeta actual (Figura 8.28), que permeten pujar arxius, pujar carpetes senceres o crear una nova carpeta. Aquestes opcions també estan disponibles a la secció de Çompartits amb mi", sempre que l'usuari tingui permisos d'escriptura a la carpeta seleccionada.

A la cantonada superior dreta (Figura 8.29), es troben les opcions globals: un menú per ordenar els fitxers (per nom, data, etc.), un botó per accedir al panell d'administració (només visible per a administradors), un altre per a la configuració del compte d'usuari, que obre un modal per modificar dades personals i canviar la contrasenya (Figura 8.30), i, finalment, el botó per tancar la sessió.

FIGURA 8.27: Detall de l'arbre de navegació, on s'indica la carpeta seleccionada i les diferents seccions.

FIGURA 8.28: Accions sobre la carpeta actual (pujar i crear).

FIGURA 8.29: Opcions globals (ordenació, configuració i logout).

FIGURA 8.30: Modal de configuració d'usuari.

Interacció amb elements Cada fitxer o carpeta disposa d'un menú contextual (Figura 8.31) que permet realitzar diverses operacions. Algunes d'aquestes accions es duen a terme a través de finestres modals dissenyades per a cada tasca específica:

- **Detalls**: Mostra informació rellevant de l'element (Figura 8.32).
- **Renombrar**: Permet canviar el nom de l'element (Figura 8.33).
- **Moure**: Facilita el trasllat d'elements a una altra carpeta (Figura 8.34).
- **Compartir**: Obre un modal on es pot buscar un usuari, assignar-li permisos i gestionar els accessos existents (Figura 8.35).

A més d'aquestes accions que es gestionen amb finestres modals, el menú contextual ofereix opcions d'acció directa com ara descarregar, copiar, tallar i eliminar (que mou l'element a la paperera).

FIGURA 8.31: Menú contextual d'accions sobre un element.

FIGURA 8.32: Modal amb els detalls d'un fitxer.

FIGURA 8.33: Modal per renombrar un element.

FIGURA 8.34: Modal per moure elements a una altra ubicació.

FIGURA 8.35: Modal de compartició d'arxius.

Gestió d'elements compartits Per als elements que es troben a la secció Çompartits amb mi", les opcions del menú contextual són dinàmiques i depenen dels permisos

que l'usuari tingui sobre l'element (Figura 8.36). Les opcions només es mostren si l'usuari té els privilegis necessaris:

- Per a tots els fitxers compartits, existeix l'opció per deixar de compartir, que elimina l'accés de l'usuari a l'element.
- Si es tenen permisos de **lectura**, es pot descarregar l'arxiu i consultar-ne els detalls.
- Si es tenen permisos d'**escriptura**, a més de les anteriors, s'afegeixen les opcions de renombrar, copiar, tallar i moure l'element.

FIGURA 8.36: Menú contextual per a un element compartit.

Paperera La vista de la paperera (Figura 8.37) mostra tots els elements eliminats i ofereix un menú contextual amb diverses opcions per a cada un:

- **Restaurar**: Retorna l'element a la seva ubicació original.
- **Eliminar definitivament**: Esborra l'element de forma permanent del sistema.
- **Detalls**: Mostra la informació de l'arxiu.

FIGURA 8.37: Opcions de la paperera.

Panell d'administració Finalment, els usuaris amb rol d'administrador tenen accés a un panell exclusiu (Figura 8.38) per gestionar els comptes d'usuari de la plataforma, on poden veure la llista d'usuaris, modificar-ne les dades o eliminar-los.

FIGURA 8.38: Panell d'administració d'usuaris.

8.4.2 Client d'escriptori

El disseny de l'aplicació d'escriptori amb Tauri s'ha centrat a oferir una experiència nativa i integrada amb el sistema operatiu, posant especial èmfasi en la sincronització automàtica de fitxers.

El primer cop que s'inicia l'aplicació, es presenta a l'usuari la finestra de configuració inicial (Figura 8.39). Aquí ha de definir dos paràmetres clau: l'endpoint del servidor

al qual es connectarà i la carpeta local que es mantindrà sincronitzada. Aquesta configuració és un pas previ indispensable per poder operar.

FIGURA 8.39: Configuració inicial del servidor i la carpeta de sincronització.

Un cop guardada la configuració, l'usuari ha d'iniciar sessió (Figura 8.40) per accedir al seu compte. L'aplicació desarà els tokens d'autenticació de manera segura, de manera que aquest pas només serà necessari la primera vegada o si la sessió caduca després d'un llarg període d'inactivitat.

FIGURA 8.40: Finestra d'inici de sessió del client Tauri.

Després d'autenticar-se, l'usuari accedeix a les funcionalitats principals de l'aplicació:

- **Finestra principal de sincronització**: Aquesta és la vista principal de l'aplicació (Figura 8.41), on l'usuari pot monitorar l'estat de la sincronització. Mostra

un historial de les pujades i baixades actives i recents. A la part superior, disposa de quatre botons per a accions ràpides:

- Obrir la carpeta de sincronització local (icona de carpeta).
- Accedir a l'aplicació web (icona de globus terraqüi).
- Forçar una sincronització manual (icona de dues fletxes circulars).
- Obrir el menú de configuració (icona d'engranatge).

FIGURA 8.41: Finestra principal de sincronització del client d'escriptori.

- **Menú de configuració**: L'usuari pot accedir a una finestra de configuració (Figura 8.42) per ajustar paràmetres de l'aplicació, com les dades de l'usuari o canviar la carpeta de sincronització.

FIGURA 8.42: Menú de configuració del client d'escriptori.

Finalment, l'aplicació s'integra a la safata del sistema operatiu mitjançant una icona (Figura 8.43), des de la qual es pot accedir ràpidament a la configuració, obrir la carpeta local, veure l'estat de la sincronització o tancar l'aplicació.

FIGURA 8.43: Menú contextual de la icona de Tauri a la safata del sistema.

8.5 Consideracions de seguretat

La seguretat ha estat un pilar fonamental en el disseny del sistema, abordant-se des de l'autenticació i autorització fins a la protecció de dades en trànsit i en repòs. A continuació, es detallen les principals mesures implementades.

8.5.1 Control d'accés basat en rols (RBAC)

El sistema implementa un model de control d'accés jeràrquic amb tres nivells de rols, gestionats pel servei UserAuthentication. Aquests rols defineixen les capacitats de cada usuari dins de l'aplicació, seguint el principi de mínim privilegi. A la Taula 8.2 es resumeixen els permisos associats a cada rol.

TAULA 8.2: Matriu de permisos per rol d'usuari.

Acció	Usuari (USER)	Administrador (ADMIN)	Superadministrador (SUPER_ADMIN)
Gestió del propi compte			
Actualitzar perfil i canviar contrasenya	✓	✓	✓
Eliminar el propi compte	✓	✓	✓
Gestió d'arxius			
Crear, llegir, actualitzar i esborrar (CRUD) arxius/carpetes propis	✓	✓	✓
Compartir arxius/carpetes amb altres usuaris	✓	✓	✓
Gestionar la paperera (restaurar/eliminar permanentment)	✓	✓	✓
Administració d'usuaris			
Llistar tots els usuaris		✓	✓
Actualitzar perfil d'usuaris amb rol USER		✓	✓
Eliminar usuaris amb rol USER		✓	✓
Administració avançada			
Actualitzar perfil d'usuaris amb rol ADMIN			✓
Eliminar usuaris amb rol ADMIN			✓
Canviar la contrasenya de qualsevol usuari			✓
Modificar el rol de qualsevol usuari			✓

A més d'aquests rols globals, el servei FileAccessControl gestiona permisos a nivell d'element individual (READ, WRITE, ADMIN), la qual cosa permet un control granular sobre qui pot accedir o modificar cada arxiu i carpeta.

8.5.2 Mecanismes de protecció

Per garantir la integritat i confidencialitat del sistema, s'han implementat diverses capes de seguretat:

- **Autenticació amb JSON Web Tokens (JWT)**: Totes les peticions a l'API protegida han d'incloure un token JWT. Aquest és generat pel servei UserAuthentication utilitzant un sistema de criptografia asimètrica (RSA), signant cada token amb una clau privada. El **Gateway**, posseïdor de la clau pública, actua com a primer filtre, validant la signatura i la caducitat del token abans de redirigir la petició. Aquest mètode garanteix que només el servei d'autenticació pot generar tokens vàlids. A més, el payload del token s'enriqueix amb el rol de l'usuari i un identificador de connexió únic (connectionId). Un cop validat el token, la combinació de l'identificador d'usuari (userId) i aquesta connectionId permet identificar de manera inequívoca cada sessió d'usuari, possibilitant una autorització primerenca i un control més granular.

- **Abstracció d'identificadors interns**: El sistema ha estat dissenyat per no exposar mai els identificadors interns de la base de dades (claus primàries, generalment UUIDs) a l'exterior. En la comunicació amb els clients, s'utilitzen identificadors públics com el nom d'usuari per a referenciar usuaris o un 'elementId' específic per a fitxers i carpetes. Aquesta capa d'abstracció impedeix que un atacant que pugui interceptar la comunicació obtingui informació sobre l'estructura interna del sistema o pugui endevinar fàcilment els identificadors per accedir a recursos aliens (atac de tipus *Insecure Direct Object Reference* o IDOR).

- **Validació de dades d'entrada**: Els serveis del backend apliquen validacions estrictes sobre les dades rebudes (p. ex., format de correu electrònic, complexitat de la contrasenya, tipus de dades esperats). Aquesta mesura és crucial per prevenir atacs d'injecció (com SQLi o XSS) i garantir la integritat de les dades.

- **Limitació de recursos**: S'estableix un límit en la mida màxima dels arxius que es poden pujar (100 MB), configurat a nivell del servei FileManagement. Aquesta mesura evita que un atacant pugui esgotar l'espai d'emmagatzematge del servidor. Tot i que no s'ha arribat a implementar un límit en la quantitat de peticions per segon (*rate limiting*), es considera una millora de seguretat crucial a desenvolupar en el futur per protegir el sistema contra atacs de denegació de servei (DoS) i de força bruta.

8.5.3 Justificació del compliment del RGPD

El disseny del sistema ha tingut en compte els requisits del Reglament General de Protecció de Dades (RGPD) des de la seva concepció:

- **Dret a l'oblit**: El sistema garanteix el dret a l'eliminació completa de les dades d'un usuari. Quan un usuari elimina el seu compte (o un administrador ho fa en nom seu), s'inicia una saga asíncrona que propaga l'ordre d'eliminació a tots els microserveis. Aquest procés assegura que totes les dades personals, arxius, permisos i registres associats siguin purgats de forma permanent de totes les bases de dades.

- **Seguretat i confidencialitat de les dades**: Les dades personals es tracten amb mesures de seguretat robustes. Les contrasenyes s'emmagatzemen a la base de dades utilitzant un algorisme de *hashing* fort, la qual cosa impedeix que puguin ser llegides fins i tot en cas d'un accés no autoritzat a la base de dades. L'accés a les dades en repòs està restringit per les credencials de cada microservei. A més, el sistema implementa una ofuscació per disseny per als arxius emmagatzemats: cada fitxer es desa al sistema d'arxius amb un identificador únic (UUID) com a nom, sense la seva extensió original, i tots junts en una única ubicació. Aquesta tècnica desacobla el contingut de les seves metadades (nom, propietari), fent que, en cas d'un accés no autoritzat al servidor, sigui extremadament difícil identificar, interpretar o associar els arxius amb usuaris concrets.

- **Principi de minimització de dades i accés granular**: El sistema està dissenyat per limitar l'accés a les dades només al personal autoritzat a través del sistema de rols. A més, la compartició d'arxius requereix una acció explícita per part del propietari, que pot assignar permisos de només lectura o d'escriptura, garantint que els usuaris només tinguin accés a la informació estrictament necessària.

8.6 Cobertura dels requisits no funcionals

El disseny del sistema, detallat al llarg d'aquest capítol, s'ha concebut per donar resposta directa als requisits no funcionals establerts al Capítol 6. A continuació, s'analitza com les decisions arquitectòniques i de disseny cobreixen cada una de les àrees clau.

- **Rendiment i Escalabilitat**: La decisió fonamental per satisfer aquests requisits és l'adopció d'una **arquitectura de microserveis**. Com s'ha justificat prèviament, aquesta permet l'**escalabilitat horitzontal** independent de cada servei, de manera que components amb alta demanda com FileManagement o SyncService es poden replicar per gestionar un major nombre d'usuaris i operacions simultànies. Per garantir temps de resposta ràpids, el disseny de la base de dades de cada microservei inclou **índexs estratègics** en les consultes més freqüents, com la cerca d'usuaris o la llista de fitxers d'una carpeta.

- **Seguretat**: Aquesta àrea es cobreix àmpliament a la secció de Çonsideracions de seguretat". El disseny compleix els requisits d'alta prioritat mitjançant:

  – Una **autenticació segura** amb tokens JWT signats amb criptografia asimètrica (RSA).

  – Un **control d'accés granular** a dos nivells: un control global basat en rols (RBAC) i un de més fi a nivell de fitxer i carpeta gestionat pel servei FileAccessControl.

  – La **protecció contra atacs comuns** com IDOR, mitjançant l'abstracció d'identificadors interns, i la validació estricta de totes les dades d'entrada per prevenir injeccions de codi.

- **Mantenibilitat**: L'arquitectura de microserveis promou per si mateixa un codi modular i desacoblat. A més, tal com s'ha detallat a la secció de "Patrons de disseny i principis arquitectònics", cada servei segueix una **arquitectura per capes** (controlador, servei, repositori) que aplica el Principi de Responsabilitat Única (SRP). L'ús extensiu de la **injecció de dependències** i patrons com el **Repository** o el **DTO** contribueix a un codi més net, extensible i fàcil de provar.

- **Portabilitat**: L'arquitectura dissenyada està pensada per a un desplegament amb **Docker**, complint el requisit d'instal·lació senzilla a través de contenidors. La comunicació entre serveis a través d'un registre com Eureka i una passarel·la API facilita l'orquestració en qualsevol entorn compatible.

- **Usabilitat i Compatibilitat**: El disseny contempla des del principi la creació de clients diferenciats (web i escriptori amb Tauri), la qual cosa garanteix la compatibilitat amb els principals navegadors i sistemes operatius. Els esbossos de les interfícies d'usuari defineixen una estructura de vistes lògica i funcional, orientada a una experiència d'usuari intuïtiva.

8.7 Conclusions

L'arquitectura modular basada en microserveis ha permès una separació clara de responsabilitats, facilitant el desenvolupament, la integració de clients diversos i el compliment de requisits com la seguretat o la sincronització. El disseny de la interfície s'ha alineat des del principi amb la funcionalitat tècnica, assegurant una experiència d'usuari fluida.

Capítol 9

Implementació i proves

9.1 Visió general de la implementació

Aquest capítol descriu com els conceptes teòrics, els requisits i les decisions de disseny exposades en capítols anteriors s'han materialitzat en una solució de programari funcional i robusta. L'objectiu aquí no és detallar cada línia de codi, sinó oferir una visió panoràmica de l'arquitectura final, les tecnologies emprades i els patrons de comunicació que garanteixen la cohesió del sistema. La implementació que presento és el resultat directe de l'estudi de viabilitat (Capítol 2), la metodologia de treball (Capítol 3) i, especialment, de les decisions tècniques justificades al Capítol 7, tot plegat per satisfer els requisits funcionals i no funcionals definits al Capítol 6.

L'arquitectura del sistema es fonamenta en un model de microserveis, una decisió presa per garantir la modularitat, l'escalabilitat i la mantenibilitat a llarg termini, tal com es va raonar al Capítol 7. El backend està format per un conjunt de serveis independents, cadascun amb una responsabilitat única, que es despleguen en contenidors Docker i s'orquestren mitjançant un fitxer 'compose.yml'. Aquesta estratègia compleix el requisit de portabilitat (RNF-7) i facilita enormement la posada en marxa de l'entorn. Els serveis principals són:

- **UserAuthentication i UserManagement:** S'encarreguen del registre, l'autenticació (mitjançant JWT) i la gestió de les dades dels usuaris.
- **FileManagement:** Constitueix el nucli de la gestió d'arxius, gestionant la pujada, la descàrrega, la creació de carpetes i l'estructura de directoris.
- **TrashService:** Implementa la funcionalitat de la paperera de reciclatge, permetent l'eliminació temporal i la restauració d'arxius.
- **FileSharing:** Gestiona la lògica per compartir arxius entre usuaris, incloent-hi el control de permisos.
- **SyncService:** Orquestra la sincronització en temps real entre clients mitjançant WebSockets.

Aquests serveis es recolzen en components d'infraestructura com **Spring Cloud Gateway**, que actua com a única porta d'entrada per a totes les peticions externes, centralitzant la seguretat i l'enrutament, i **Eureka**, que proporciona el descobriment de serveis per a una comunicació interna resilient. Al front, tenim un **client web** desenvolupat amb React i un **client d'escriptori** natiu construït amb Tauri i Svelte, ambdós

dissenyats per oferir una experiència d'usuari moderna i eficient.

La comunicació entre aquests components segueix dos patrons principals, una decisió clau explicada al Capítol 7. Per a les operacions que requereixen una resposta immediata per part de l'usuari (com iniciar sessió o pujar un arxiu), s'utilitza una comunicació síncrona mitjançant **peticions HTTP**. Aquestes peticions flueixen des del client, a través del Gateway, fins al microservei corresponent. En canvi, per a processos que poden executar-se en segon pla sense bloquejar l'usuari (com la neteja de dades en cascada després d'eliminar un compte), he implementat un sistema de **missatgeria asíncrona amb RabbitMQ**. Aquest enfocament millora la resiliència i l'experiència d'usuari, ja que garanteix que les tasques es completin de manera fiable fins i tot si un servei pateix una fallada temporal. Finalment, la sincronització en temps real es duu a terme a través d'una connexió **WebSocket** persistent, que permet al servidor notificar canvis a tots els clients connectats de manera instantània.

Els fluxos complets de cada funcionalitat, que demostren la interacció entre tots aquests components per satisfer els requisits del sistema, es troben detallats a l'Apèndix A (@AppendixA.tex, casos d'ús UC-01 a UC-22).

En lloc de presentar fragments de codi aïllats, considero més il·lustratiu referir-se a la taula de versions i dependències principals que ja vaig detallar a la Taula 5.1 del Capítol 5. Aquesta taula proporciona una fotografia precisa de la pila tecnològica final del projecte, on destaquen **Java 17** amb **Spring Boot 3** per al backend, **React 18** per al client web i **Tauri 1.5** amb **Rust** per al client d'escriptori. Aquestes eleccions són el resultat de l'anàlisi i les decisions preses al llarg de tot el desenvolupament.

En resum, la implementació del projecte tradueix una arquitectura distribuïda a un sistema tangible. El flux global es basa en una combinació estratègica de comunicació HTTP per a la interacció directa i missatgeria asíncrona per a la resiliència i el desacoblament, demostrant com l'aplicació pràctica de patrons de disseny moderns permet construir solucions complexes, escalables i robustes.

9.2 Entorn de desenvolupament i desplegament

Per garantir la coherència i la reproduïbilitat del sistema, un dels requisits clau del projecte (RNF-4, RNF-7), vaig prestar una atenció especial a la definició de l'entorn de desenvolupament i al procés de desplegament. La decisió fonamental va ser estructurar tot el projecte en un **monorepo**, una elecció estratègica per simplificar la gestió de dependències i assegurar que totes les peces del sistema (backend, frontend, client d'escriptori i documentació) evolucionessin de manera sincronitzada. Aquesta centralització facilita la traçabilitat dels canvis i redueix la complexitat inherent a la gestió de múltiples repositoris.

El cor de l'entorn de desenvolupament local és el fitxer docker-compose.yml, que actua com a orquestrador de tota l'arquitectura. A través d'aquest, s'inicien tant els microserveis del servidor com el client web, permetent aixecar l'ecosistema complet amb una única comanda: docker compose up. Com es va justificar al Capítol 7, l'ús de Docker i Docker Compose va ser una decisió presa per la seva simplicitat i per l'experiència prèvia. Aquest fitxer no només defineix cada servei, sinó també les seves relacions, les xarxes, els volums persistents i la configuració essencial.

```
1  file-manager:
```

```
    build: ./FileManagement
    container_name: filemanagement
    environment:
      - "SPRING_PROFILES_ACTIVE=docker"
    depends_on:
      eureka:
        condition: service_healthy
    volumes:
      - ./storage_data:/app/files
```

LISTING 9.1: Fragment del fitxer 'compose.yml' definint un servei

Com es pot observar en aquest fragment, cada servei es construeix a partir del seu propi directori (p. ex., ./FileManagement), se li assigna un perfil de Spring específic per a Docker (SPRING_PROFILES_ACTIVE=docker) i s'estableixen dependències explícites. En aquest cas, el servei file-manager no s'iniciarà fins que Eureka estigui saludable (service_healthy), garantint un ordre d'arrencada correcte i evitant errors en cascada. A més, es munta un volum local (./storage_data) per persistir els arxius pujats pels usuaris, una configuració crítica per al desenvolupament i les proves.

Per simplificar la posada en marxa a usuaris que no tinguin coneixements tècnics avançats, es va desenvolupar un script d'instal·lació (setup.sh), tal com es va mencionar al Capítol 5. Aquest permet una instal·lació ràpida i automatitzada de tots els passos necessaris: comprova les dependències del sistema (com Docker i Java), construeix les imatges de tots els contenidors i finalment invoca docker compose up per llançar la plataforma. Això redueix el procés a l'execució d'un sol fitxer, complint l'objectiu de fer el sistema accessible.

La configuració dels serveis es gestiona principalment a través de **variables d'entorn** i els fitxers de propietats de Spring Boot, que inclouen perfils per a diferents entorns (com dev per a desenvolupament local i docker per a l'execució dins de contenidors). Les variables d'entorn crítiques, com les credencials de la base de dades o les claus per a la signatura de tokens JWT, es defineixen directament al compose.yml o en fitxers .env per evitar exposar informació sensible al codi font, una pràctica de seguretat fonamental.

9.3 Implementació del backend

A continuació, s'analitza la implementació de cada bloc funcional del backend, posant èmfasi en com els microserveis materialitzen els requisits del sistema i les decisions d'arquitectura preses.

9.3.1 Gateway i filtre de JWT

El microservei Gateway, basat en Spring Cloud Gateway, és la porta d'entrada única per a totes les peticions externes, una peça clau en l'arquitectura de microserveis que vaig escollir, tal com vaig justificar al Capítol 7. La seva funció no es limita a ser un simple proxy; actua com un vigilant que centralitza la seguretat i organitza el trànsit cap als diferents serveis interns. Totes les rutes de l'aplicació estan definides de manera declarativa en una classe de configuració, GatewayConfig.java, mitjançant un bean de RouteLocator.

```
.route("user_login_post", r -¿ r.path("/users/auth/login")
        .and().method(HttpMethod.POST, HttpMethod.OPTIONS)
```

```
3          .uri("lb://USERAUTHENTICATION"))
4
5  .route("files˙root˙get", r -¿ r.path("/files/root")
6          .and().method(HttpMethod.GET, HttpMethod.OPTIONS)
7          .filters(f -¿ f.filter(jwtAuthenticationFilter))
8          .uri("lb://FileManagement"))
9
10 .route("admin˙users˙get", r -¿ r.path("/admin/users")
11         .and().method(HttpMethod.GET, HttpMethod.OPTIONS)
12         .filters(f -¿ f.filter(jwtAuthenticationFilter))
13         .uri("lb://UserManagement"))
```

LISTING 9.2: Exemples de definició de rutes a 'GatewayConfig'

Aquest enfocament permet una gestió centralitzada i clara de l'enrutament. Per exemple:

- La ruta user_login_post (**UC-02**) és pública; les peticions a /users/auth/login es redirigeixen directament al servei USERAUTHENTICATION sense cap filtre de seguretat.
- La ruta files_root_get, en canvi, aplica el filtre jwtAuthenticationFilter abans de redirigir la petició al servei FileManagement. Això assegura que només els usuaris autenticats puguin accedir als seus arxius.
- La ruta admin_users_get (**UC-05**) també passa pel filtre, que, com veurem, conté lògica específica per validar rols administratius.

El component més crític del Gateway és el filtre global JwtAuthenticationFilter. Aquest filtre intercepta cada petició (excepte les públiques) i orquestra la validació del token JWT. La seva implementació gestiona dos mètodes d'extracció del token: per a les peticions HTTP convencionals, l'extreu de la capçalera Authorization; per a la connexió WebSocket desde el client web, l'obté d'un paràmetre a la URL (?token=...). Aquesta dualitat és necessària a causa d'una limitació de l'API de WebSockets dels navegadors, que no permet afegir capçaleres personalitzades a la petició inicial d'establiment de la connexió, tal com es pot comprovar al fitxer websocket.ts.

```
1  @Component
2  public class JwtAuthenticationFilter implements GatewayFilter -
3      // ... injecció de WebClient i altres dependències ...
4
5      @Override
6      public Mono¡Void¿ filter(ServerWebExchange exchange,
       GatewayFilterChain chain) -
7          ServerHttpRequest request = exchange.getRequest();
8          HttpHeaders headers = request.getHeaders();
9          String authHeader = "Bearer ";
10         String p = request.getPath().toString();
11
12         if(p.toString().startsWith("/websocket/web")) -
13             authHeader = "Bearer ".concat(request.getQueryParams().
       getFirst("token"));
14         " else -
15
16             if (!headers.containsKey(HttpHeaders.AUTHORIZATION)) -
17                 exchange.getResponse().setStatusCode(HttpStatus.
       UNAUTHORIZED);
18                 return exchange.getResponse().setComplete();
19             "
```

```
20
21          authHeader = headers.getFirst(HttpHeaders.AUTHORIZATION);
22          if (authHeader == null -- !authHeader.startsWith("Bearer "))
 -
23              exchange.getResponse().setStatusCode(HttpStatus.
UNAUTHORIZED);
24              return exchange.getResponse().setComplete();
25          "
26      "
27
28      String accessToken = authHeader.substring(7);
29
30      return webClient.method(HttpMethod.POST)
31              .uri("/users/auth/check")
32              .header(HttpHeaders.AUTHORIZATION, "Bearer " +
accessToken)
33              .retrieve()
34              .onStatus(HttpStatusCode::isError, response -¿ -
35                  logger.error("Invalid token response from auth
service");
36                  return Mono.error(new InvalidTokenException());
37              ")
38              .bodyToMono(UserAuthResponse.class)
39              .flatMap(userAuthResponse -¿ -
40                  if (userAuthResponse.getId() == null --
userAuthResponse.getRole() == null) -
41                      logger.warn("User -" with role -" attempted to
access admin route -"", userAuthResponse.getUsername(),
userAuthResponse.getRole(), request.getPath());
42                      exchange.getResponse().setStatusCode(HttpStatus.
FORBIDDEN);
43                      return exchange.getResponse().setComplete();
44                  "
45                  if (request.getPath().toString().startsWith("/admin/
") && !("ADMIN".equals(userAuthResponse.getRole()) -- "SUPER'ADMIN".
equals(userAuthResponse.getRole()))) -
46                      logger.warn("User -" with role -" attempted to
access admin route -"", userAuthResponse.getUsername(),
userAuthResponse.getRole(), request.getPath());
47                      exchange.getResponse().setStatusCode(HttpStatus.
FORBIDDEN);
48                      return exchange.getResponse().setComplete();
49                  "
50
51                  ServerHttpRequest modifiedRequest = request.mutate()
52                          .header(X'USER'ID'HEADER, userAuthResponse.
getId().toString())
53                          .header(X'CONNECTION'ID'HEADER,
userAuthResponse.getConnectionId())
54                          .build();
55                  return chain.filter(exchange.mutate().request(
modifiedRequest).build());
56              ")
57              .onErrorResume(e -¿ -
58                  logger.error("Error during authentication process",
e);
59                  exchange.getResponse().setStatusCode(
60                          e instanceof InvalidTokenException ?
HttpStatus.UNAUTHORIZED : HttpStatus.INTERNAL'SERVER'ERROR
61                  );
62                  return exchange.getResponse().setComplete();
63              ");
64  "
```

```
65 ”
```

LISTING 9.3: Implementació del filtre ‘JwtAuthenticationFilter‘

Com es pot observar al codi, el filtre primer comprova si la ruta de la petició correspon a l'endpoint de WebSocket (/websocket/web). Si és així, extreu el token del paràmetre de la consulta. En cas contrari, busca el token a la capçalera Authorization, com és habitual en peticions REST.

Un cop extret el token, i en lloc de contenir la lògica de validació directament, el filtre delega aquesta responsabilitat al servei UserAuthentication fent una crida interna al seu endpoint /users/auth/check. Aquesta decisió de disseny manté el Gateway lleuger i respecta el principi de responsabilitat única. Si la resposta del servei d'autenticació és positiva, el filtre realitza dues accions clau:

1. **Validació de rols:** Comprova si la ruta requereix permisos d'administrador i si l'usuari els té. Si no, retorna un error **403 Forbidden**, indicant que l'usuari està autenticat però no autoritzat per a aquest recurs. Això és diferent d'un **401 Unauthorized**, que es retorna quan el token és invàlid o no es proporciona.

2. **Enriquiment de la petició:** Afegeix les capçaleres X-User-Id i X-Connection-Id a la petició abans de passar-la al microservei de destinació. Això permet que els serveis interns confiïn en aquestes capçaleres per identificar l'usuari sense necessitat de validar el token de nou, optimitzant el rendiment.

Tot i que aquest disseny és robust, soc conscient que la crida interna des del Gateway a UserAuthentication a cada petició protegida introdueix una latència addicional i podria convertir-se en un coll d'ampolla sota càrregues elevades. Una possible millora futura, discutida al Capítol 12, seria implementar un mecanisme de memòria cache al Gateway per als resultats de validació de tokens o compartir la clau pública de signatura perquè el Gateway pugui validar els tokens localment sense la crida de xarxa.

9.3.2 Gestió d'usuaris i autenticació

La gestió d'usuaris és una responsabilitat compartida entre dos microserveis diferents, UserAuthentication i UserManagement, que junts formen un patró d'agregació. Aquesta separació de responsabilitats és una decisió de disseny clau: UserAuthentication actua com l'arrel de l'agregat, gestionant exclusivament les dades crítiques per a la seguretat —com les credencials i els tokens—, mentre que UserManagement s'encarrega de les dades del perfil de l'usuari, com l'email o altres detalls personals.

L'arquitectura interna de tots dos serveis segueix el patró per capes de Spring Boot, amb una clara separació de responsabilitats:

- **Capa de controladors (**controllers**):** Exposa els endpoints REST que el client o altres serveis consumeixen.

- **Capa de serveis (**services**):** Conté la lògica de negoci principal, com la validació de dades i l'orquestració de processos.

- **Capa de repositoris (**repositories**):** Abstrau l'accés a la base de dades mitjançant Spring Data JPA.

Aquesta estructura modular facilita el manteniment i l'escalabilitat de cada microservei de manera independent.

Una decisió clau en la implementació de tots els controladors del sistema va ser la gestió d'errors. Per evitar embolcallar tota la lògica de negoci en blocs try-catch i per centralitzar la conversió d'errors interns a respostes HTTP, vaig optar per utilitzar les excepcions de Spring. La majoria d'errors controlats dins de la capa de serveis llancen una ResponseStatusException, una excepció especial que conté directament el codi d'estat HTTP i el missatge que es vol retornar. Als controladors, vaig implementar *handlers* d'excepcions amb l'anotació @ExceptionHandler. Aquests mètodes capturen automàticament les excepcions llançades, incloent-hi les ResponseStatusException, i les mapegen a una resposta ResponseEntity adequada. A més, es va incloure un handler genèric per a RuntimeException que captura qualsevol error no controlat, registra la traça per a depuració i retorna un error 500 Internal Server Error genèric. A continuació es mostra un exemple d'aquests gestors al UserController:

```
@ExceptionHandler(RuntimeException.class)
public ResponseEntity¡String¿ handleRuntimeException(RuntimeException e)
    –
    e.printStackTrace();
    return ResponseEntity.status(HttpStatus.INTERNAL'SERVER'ERROR)
                        .body(e.getMessage());
"

@ExceptionHandler(ResponseStatusException.class)
public ResponseEntity¡String¿ statusException(ResponseStatusException e)
    –
    e.printStackTrace();
    return ResponseEntity.status(e.getStatusCode()).body(e.getMessage()
    ;
"
```

LISTING 9.4: Exemple de gestors d'excepcions al UserController.

Aquest enfocament no només simplifica el codi, sinó que també assegura un maneig d'errors consistent i segur a tota l'API.

UC-01: Registrar-se

El cas d'ús de registre és un procés orquestrat per UserAuthentication que involucra múltiples serveis per assegurar la creació consistent de totes les dades d'un nou usuari.

FIGURA 9.1: Flux de dades per al registre d'un nou usuari (UC-01).

El flux comença quan el client envia una petició de registre a l'endpoint /users/auth/register del Gateway. Aquest, després de validar la ruta, la reenvia a UserAuthentication. El controlador AuthenticationController rep la petició i delega la lògica al servei UserService.

```
@PostMapping("/register")
public ResponseEntity¡?¿ registerUser(@RequestBody @Valid
    UserRegisterRequest userInfoRequest, HttpServletResponse response) -
    // ... validació de l'usuari i email ...
    try -
        userService.register(userInfoRequest);
        UserEntity user = userService.loadUserByUsername(userInfoRequest
    .getUsername());
        return addTokenHeaders(user);
    " catch (IllegalArgumentException e) -
        return ResponseEntity.badRequest().body(e.getMessage());
    "
"
```

LISTING 9.5: Endpoint de registre a 'AuthenticationController'

Dins del mètode userService.register, el servei realitza les següents accions en una única transacció:

1. **Validació:** Comprova que el nom d'usuari i l'email no estiguin ja en ús. Per a l'email, fa una crida a UserManagement a través d'un client Feign.

2. **Creació a UserAuthentication:** Si les dades són vàlides, es xifra la contrasenya amb BCrypt i es desa la nova UserEntity (amb nom d'usuari, contrasenya i rol USER) a la seva pròpia base de dades.

3. **Creació a UserManagement:** A continuació, crida a UserManagement per crear el perfil d'usuari associat, enviant l'ID, el nom i l'email. Això demostra el patró d'agregació, on UserAuthentication gestiona el cicle de vida de l'usuari complet.

4. **Creació de la carpeta arrel:** Fa una crida a FileManagement perquè creï la carpeta arrel per al nou usuari.

5. **Creació del snapshot inicial:** Finalment, publica un missatge a RabbitMQ que serà consumit per SyncService per generar la representació inicial de l'arbre de fitxers (*snapshot*) de l'usuari.

Si totes les operacions tenen èxit, el controlador genera els tokens JWT i els retorna al client, iniciant la seva sessió automàticament. L'ús d'excepcions permet un maneig d'errors clar, retornant un codi 400 Bad Request si alguna validació falla.

UC-02: Iniciar sessió

L'inici de sessió és un procés gestionat íntegrament per UserAuthentication, ja que és l'únic servei que té accés a les credencials dels usuaris.

FIGURA 9.2: Flux de dades per a l'inici de sessió (UC-02).

El client envia el nom d'usuari i la contrasenya, i el AuthenticationController utilitza el mètode checkCredentials del UserService per validar-los.

```
@PostMapping("/login")
public ResponseEntity¡?¿ loginUser(@RequestBody UserInfoRequest
    userInfoRequest) -
    if(!userService.checkCredentials(userInfoRequest.getUsername(),
    userInfoRequest.getPassword())) -
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("
    Invalid credentials");
    "
    UserEntity user = userService.loadUserByUsername(userInfoRequest.
    getUsername());
    return addTokenHeaders(user);
"
```

LISTING 9.6: Endpoint de login a 'AuthenticationController'

Si les credencials són correctes, es generen dos tipus de tokens JWT a través de la utilitat JwtTokenUtil: un token d'accés de curta durada (p. ex., 1 hora) i un de refresc de llarga durada (p. ex., 7 dies). La gestió dels secrets, com la clau privada RSA per signar els tokens, es fa carregant-la des d'un fitxer al *classpath* a l'inici de l'aplicació, evitant la seva exposició al codi font.

```
public String generateToken(UserEntity user, boolean isRefreshToken) -
    long validity = isRefreshToken ? REFRESH_TOKEN_VALIDITY :
    ACCESS_TOKEN_VALIDITY;
```

```
    Map¡String, Object¿ claims = new HashMap¡¿();
    claims.put("role", user.getRole());
    claims.put("connection-id", UUID.randomUUID().toString());
    claims.put("password-changed-at", String.valueOf(user.
    getLastPasswordChange().getTime()));
    return createToken(claims, user.getUsername(), validity);
"
```

LISTING 9.7: Generació de tokens a la classe 'JwtTokenUtil'

Els tokens contenen *claims* addicionals com el rol de l'usuari, que serà utilitzat pel Gateway per a l'autorització, i la marca de temps de l'últim canvi de contrasenya, clau per a la invalidació automàtica de sessions.

UC-15: Canviar contrasenya

Aquest cas d'ús permet a un usuari modificar la seva pròpia contrasenya i és gestionat per UserAuthentication.

FIGURA 9.3: Flux de dades per al canvi de contrasenya (UC-15).

El procés comença amb una petició PUT a /users/password. El servei UserService primer verifica que la contrasenya antiga proporcionada per l'usuari coincideix amb la que hi ha a la base de dades. Si és així, valida que la nova contrasenya compleixi els requisits de seguretat, la xifra i l'actualitza.

```
    public void changePassword(UUID userId, String oldPassword, String
    newPassword) -
        UserEntity user = userRepository.findById(userId).orElseThrow(()
     -¿ new ResponseStatusException(HttpStatus.NOTFOUND, "User not found
    "));
        if (!passwordEncoder.matches(oldPassword, user.getPassword())) -
```

```
            throw new ResponseStatusException(HttpStatus.UNAUTHORIZED, "Current password is incorrect");
        "

        validatePassword(newPassword, true);

        user.setPassword(passwordEncoder.encode(newPassword));
        user.setLastPasswordChange(new Date());
        userRepository.save(user);
"
```

LISTING 9.8: Implementació del canvi de contrasenya a UserService

L'aspecte més important d'aquesta operació és que, just després de canviar la contrasenya, el servei actualitza la marca de temps lastPasswordChange a l'entitat de l'usuari. Com s'ha vist al codi de JwtTokenUtil, aquesta marca de temps s'inclou com un *claim* a cada token JWT. El Gateway, en validar un token, comprova que aquest valor coincideixi amb el que hi ha a la base de dades. Per tant, un canvi de contrasenya invalida automàticament tots els tokens emesos anteriorment, tancant totes les sessions actives en altres dispositius i millorant significativament la seguretat del compte.

UC-16 i UC-07: Eliminació de comptes d'usuari

L'eliminació d'un compte és un procés crític que ha de garantir la purga completa de les dades de l'usuari a tot el sistema. A diferència d'altres operacions d'autenticació, el Gateway centralitza totes les peticions d'eliminació, tant les iniciades per un usuari sobre el seu propi compte com les realitzades per un administrador, i les redirigeix directament al microservei UserManagement. Aquest actua com a orquestrador de tot el procés.

FIGURA 9.4: Flux de dades centralitzat per a l'eliminació d'un compte (UC-16).

La configuració de rutes al GatewayConfig.java és explícita en aquest sentit, demostrant que UserManagement és el punt d'entrada únic per a aquesta operació.

```
// A GatewayConfig.java

// Ruta per a l'auto-eliminació de l'usuari
```

```
.route("user_delete", r -¿ r.path("/users")
        .and().method(HttpMethod.DELETE, HttpMethod.OPTIONS)
        .filters(f -¿ f.filter(jwtAuthenticationFilter))
        .uri("lb://UserManagement"))

// Ruta per a l'eliminació per part d'un administrador
.route("admin_user_delete", r -¿ r.path("/admin/users/-username"")
        .and().method(HttpMethod.DELETE, HttpMethod.OPTIONS)
        .filters(f -¿ f.filter(jwtAuthenticationFilter))
        .uri("lb://UserManagement"))
```

LISTING 9.9: Rutes d'eliminació a 'GatewayConfig.java'

Un cop la petició arriba a UserManagement, el controlador corresponent (per a usuaris o administradors) invoca el servei UserDeletionService. Aquest servei executa un procés híbrid en dues fases per garantir tant la rapidesa en el bloqueig com la resiliència en la neteja:

1. **Eliminació síncrona i immediata:** El servei fa crides directes via Feign per esborrar les dades més crítiques. Primer, contacta amb UserAuthentication per eliminar les credencials, la qual cosa impedeix que l'usuari pugui tornar a iniciar sessió a l'instant. Seguidament, elimina el perfil de l'usuari de la seva pròpia base de dades.

2. **Inici de la purga asíncrona:** Un cop bloquejat l'accés, el servei publica missatges a diferents cues de RabbitMQ. Aquests missatges són consumits per la resta de serveis per a una neteja completa.

```
public void startUserDeletion(UUID userId) -
    // ... comprovació per evitar duplicats ...

    // 1. Eliminació síncrona de les dades d'autenticació i perfil
    userAuthClient.deleteAccountInternal(userId);
    userService.deleteUserLocal(userId);

    // Es crea un registre per seguir el procés i marcar-lo com a
    iniciat
    UserDeletionProcess process = new UserDeletionProcess();
    // ... s'estableixen els estats inicials ...
    process.setUserId(userId);
    process.setFileManagementStatus("PENDING");
    // ...
    process.setUserManagementStatus("DONE");
    // ...
    userDeletionProcessRepository.save(process);

    // 2. S'envien missatges per a la purga asíncrona a tots els serveis
     implicats
    sender.sendDeleteCommandToFileManagement(userId);
    sender.sendDeleteCommandToFileSharing(userId);
    sender.sendDeleteCommandToFileAccessControl(userId);
    sender.sendDeleteCommandToTrash(userId);
    sender.sendDeleteCommandToSyncService(userId);
"
```

LISTING 9.10: Inici del procés d'eliminació a 'UserDeletionService'

Aquest disseny desacoblat garanteix la resiliència del procés. Si un servei està temporalment inoperatiu, la comanda d'eliminació roman a la cua de RabbitMQ i es

processarà quan el servei es recuperi. Tanmateix, el sistema no es limita a enviar les comandes, sinó que implementa un mecanisme d'orquestració i confirmació per garantir que la purga es completi de manera efectiva a tots els serveis.

Per aconseguir aquesta orquestració, el servei UserDeletionService utilitza l'entitat UserDeletionProcess. Aquesta entitat actua com una màquina d'estats que registra el progrés de l'eliminació per a cada microservei implicat. Quan s'inicia el procés, es crea un registre a la base de dades amb l'identificador de l'usuari i l'estat de cada servei marcat com a PENDING.

Cada microservei, un cop ha finalitzat la seva tasca de neteja, té la responsabilitat de notificar-ho a UserManagement. Ho fa publicant un missatge de confirmació a una cua específica de RabbitMQ. Un consumidor a UserManagement escolta aquesta cua i, en rebre una confirmació, actualitza l'estat del servei corresponent a DONE a l'entitat UserDeletionProcess.

Finalment, per garantir la màxima robustesa i evitar que processos d'eliminació quedin bloquejats indefinidament, s'ha implementat un sistema de tasques programades a través de la classe ScheduledTasks. Aquesta classe executa dues funcions de manera periòdica (diàriament a la 1:00 AM):

- **Reintent de tasques pendents (**retryPendingDeletions**):** La tasca cerca a la base de dades tots els UserDeletionProcess que encara tinguin algun servei en estat PENDING. Per a cadascun, torna a enviar la comanda d'eliminació a través de RabbitMQ al servei corresponent. Això assegura que, fins i tot en cas de fallada, el sistema intentarà activament completar la purga. Donat el eliminar un element que no existeix a la base de dades no perjudica la coherencia de les dades encara que el servei acabi fent multiples crides per eliminar les mateixes dades no es fa cap tipus de gestio de la casoistica i nomes s'espera la resposta.

- **Neteja de processos finalitzats (**cleanupCompletedDeletions**):** Un cop tots els serveis han confirmat l'eliminació i tots els estats d'un UserDeletionProcess són DONE, el registre de seguiment ja no és necessari. Aquesta tasca s'encarrega

de localitzar aquests registres completats i eliminar-los de la base de dades, mantenint el sistema net.

Aquest cicle de comanda, confirmació i reintent programat converteix UserManagement en un orquestrador resilient que assegura la integritat i la neteja completa de les dades de l'usuari a tot l'ecosistema de microserveis.

9.3.3 Gestió d'arxius (FileManagement)

El microservei FileManagement és el cor operacional del sistema, responsable de tota la lògica relacionada amb la creació, modificació, consulta i eliminació d'arxius i carpetes. La seva implementació materialitza un gran nombre de casos d'ús, com **UC-08** (Crear/Pujar), **UC-09** (Renomenar, Moure, Copiar) i **UC-10** (Descarregar), els detalls dels quals es poden consultar a l'Apèndix A.

L'arquitectura d'aquest servei, igual que la dels altres, segueix el patró per capes de Spring Boot. El controlador principal, ElementController, exposa una API REST que permet manipular elements de manera genèrica, sense distingir inicialment entre fitxers i carpetes. Aquest delega la lògica de negoci a un servei de façana, ElementService, que actua com a punt d'entrada polimòrfic. La persistència de les dades es gestiona a través de tres entitats JPA clau: FileEntity, FolderEntity i ElementEntity. El disseny és el següent:

- FileEntity i FolderEntity emmagatzemen totes les metadades específiques de cada tipus d'element (nom, mida, dates, etc.).
- Cada FileEntity i FolderEntity té una relació @OneToOne amb una ElementEntity.
- L'ElementEntity té un propòsit doble i fonamental: en primer lloc, proporciona un identificador únic i comú (elementId) que serveix per referenciar qualsevol element des d'altres parts del sistema; en segon lloc, conté un camp booleà, isFolder, que actua com a discriminador de tipus.

Aquesta estructura permet a ElementService funcionar de manera eficient: davant d'una petició sobre un elementId, primer consulta l'ElementEntity per determinar si es tracta d'un fitxer o una carpeta, i tot seguit delega l'operació al servei especialitzat corresponent: FileService o FolderService. Aquest disseny desacobla l'API externa del model de dades intern, permetent un tractament uniforme dels elements a la vegada que es manté una lògica de negoci ben separada i especialitzada.

UC-08: Crear i pujar elements

Aquest cas d'ús gestiona tant la creació de noves carpetes com la pujada de fitxers. Una decisió clau d'implementació va ser exposar dos endpoints diferents per a la pujada de fitxers, cadascun optimitzat per a un tipus de client diferent. Tots dos mètodes estan preparats no només per crear un fitxer nou des de zero, sinó també per actualitzar el contingut d'un fitxer existent si es proporciona el seu identificador. Tot i que la intenció a futur és unificar-los en una única solució basada en *streams* per a màxima eficiència, actualment això respon a les necessitats específiques del client web i del client d'escriptori.

FIGURA 9.5: Flux del cas d'us UC-08.

El flux, com es veu a la figura 9.5, sempre comença amb una validació de permisos d'escriptura contra el microservei FileAccessControl abans de procedir.

Endpoint Multipart per al client web. El client web utilitza l'endpoint estàndard i polivalent POST /files, que accepta dades en format multipart/form-data. Aquest mètode és el responsable de gestionar tota la creació d'elements des del web: no només la pujada de fitxers, sinó també la creació de noves carpetes. Com es pot veure al codi, una comprovació sobre la petició (request.isFolder()) bifurca el flux per invocar FolderService o FileService segons correspongui. Això permet enviar en una única petició tant les metadades (en un objecte JSON) com, si s'escau, el contingut del fitxer. L'endpoint també contempla la lògica per actualitzar un fitxer existent si rep un elementId, encara que aquesta funcionalitat actualment no és utilitzada pel client web.

```
1  @PostMapping(produces = MediaType.APPLICATION_JSON_VALUE)
2  public ResponseEntity¡?¿ createElement(
3          @RequestHeader("X-User-Id") UUID userId,
4          @RequestHeader("X-Connection-Id") UUID connectionId,
5          @RequestPart("request") CreateFileRequest request,
6          @RequestPart(value = "file", required = false) MultipartFile
       file
7  ) throws IOException -
8      // ... lògica per determinar si es crea una carpeta o un fitxer ...
9      fileAccessService.checkAccessElement(userId, request.
       getParentFolderId(), true, AccessType.WRITE);
10
11     if (request.isFolder()) -
12         FolderEntity folder = folderService.createFolder(/* ... */);
```

```
13          String selectedConnection = folder.getUserId().equals(userId) ?
    connectionId.toString() : null;
14          commandService.sendCreate(folder.getElementId(),
    selectedConnection, folder.getUserId().toString(), folder.getParent()
    , "", folder.getName(), "folder");
15          return ResponseEntity.status(HttpStatus.CREATED)
16                  .contentType(MediaType.APPLICATION_JSON)
17                  .body(folderMapper.map(folder));
18      } else {
19          FileEntity fileEntity;
20          if (request.getElementId() != null) {
21              fileEntity = fileService.getFolderByElementId(UUID.
    fromString(request.getElementId()), false);
22              fileEntity.setName(request.getName());
23              fileEntity.setMimeType(request.getContentType());
24              fileEntity.setSize(request.getSize());
25          } else {
26              fileEntity = fileService.createFile(
27                      request.getName(),
28                      request.getContentType(),
29                      request.getSize(),
30                      request.getParentFolderId(),
31                      userId,
32                      connectionId.toString()
33              );
34          }
35
36          try {
37              // Pas clau: guardar el fitxer físic
38              fileUtil.storeFile(fileEntity.getId(), file);
39          } catch (IOException e) {
40              // Rollback si falla l'escriptura a disc
41              fileService.deleteFile(fileEntity.getElementId());
42          }
43          // ... notificació a RabbitMQ ...
44          return new ResponseEntity<>(fileMapper.map(fileEntity),
    HttpStatus.CREATED);
45      }
46  }
```

LISTING 9.11: Endpoint per a la creació d'elements amb 'MultipartFile' a 'ElementController'

En aquest flux, el controlador delega la creació de l'element al servei corresponent. Si es tracta d'un fitxer, primer invoca FileService per crear les metadades de FileEntity a la base de dades i, un cop l'entitat existeix i té un ID intern, crida a fileUtil.storeFile per desar el contingut físic del MultipartFile al sistema d'arxius. Si es tracta d'una carpeta, simplement invoca el FolderService per crear l'entitat corresponent a la base de dades, sense cap interacció amb el sistema d'arxius.

Endpoint de Streaming per al client d'escriptori. L'endpoint POST /upload es va afegir específicament per permetre al client d'escriptori pujar fitxers mitjançant streaming. Aquest endpoint rep el contingut del fitxer directament com un application/octet-stream en el cos de la petició, mentre que les metadades es transmeten a través de les capçaleres HTTP. El Gateway reescriu la ruta a /files/upload internament. Actualment, només el client de Tauri fa ús d'aquest mecanisme de streaming i de la capacitat d'actualització de continguts, ja que no s'ha disposat de temps per adaptar el client web per a utilitzar streams en lloc de MultipartFile.

```
@PostMapping(value = "/upload", consumes = MediaType.
    APPLICATION_OCTET_STREAM_VALUE)
public ResponseEntity¡?¿ uploadFileStream(
        @RequestHeader("X-User-Id") UUID userId,
        @RequestHeader("X-Connection-Id") UUID connectionId,
        @RequestHeader("parentId") String parentId,
        @RequestHeader("fileName") String fileName,
        // ... altres capçaleres ...
        HttpServletRequest request
) throws IOException -
    String dfn = URLDecoder.decode(fileName, StandardCharsets.UTF_8.name
    ());
        FolderEntity parent = folderService.getFolderByElementId(UUID.
    fromString(parentId), false);

        if (!fileAccessService.checkAccessFolder(parent, userId,
    AccessType.WRITE)) -
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body("
    User does not have access to this resource");
        "

        FileEntity fileEntity;

        if (elementId != null && !elementId.isEmpty()) -
            FileEntity existingFile = fileService.getFileByElementId(
    UUID.fromString(elementId), false);
            if (existingFile != null) -
                if (!fileAccessService.checkAccessFile(userId,
    existingFile.getElementId(), AccessType.WRITE)) -
                    return ResponseEntity.status(HttpStatus.FORBIDDEN).
    body("User does not have access to this resource");
                "
                fileEntity = fileService.updateFileStream(existingFile,
    dfn, request.getContentType(), request.getContentLengthLong(),
    request.getInputStream());
                String selectedConnection = fileEntity.getUserId().
    equals(userId) ? connectionId.toString() : null;
                commandService.sendUpdate(fileEntity.getElementId(),
    selectedConnection, fileEntity.getUserId().toString(), fileEntity.
    getParent(), fileService.getHash(fileEntity.getId()), fileEntity.
    getName(), "file");
                return new ResponseEntity¡¿(fileMapper.map(fileEntity),
    HttpStatus.OK);
            "
        "

        fileEntity = fileService.createFileFromStream(
                dfn,
                request.getContentType(),
                request.getContentLengthLong(),
                parent.getElementId(),
                userId,
                request.getInputStream(),
                connectionId.toString()
        );
        String selectedConnection = fileEntity.getUserId().equals(userId
    ) ? connectionId.toString() : null;
        commandService.sendCreate(fileEntity.getElementId(),
    selectedConnection, fileEntity.getUserId().toString(), fileEntity.
    getParent(), fileService.getHash(fileEntity.getId()), fileEntity.
    getName(), "file");

```

```
44          return new ResponseEntity¡¿(fileMapper.map(fileEntity),
    HttpStatus.CREATED);
45 ”
```

LISTING 9.12: Endpoint per a la pujada de fitxers amb ‘InputStream‘ a ‘ElementController‘

Aquí, la lògica està més encapsulada. La crida a fileService.createFileFromStream orquestra internament tant la creació de les metadades com l'emmagatzematge del fitxer llegint directament de l'InputStream, la qual cosa evita haver de carregar el fitxer sencer a la memòria del servidor.

Estratègia d'emmagatzematge i ofuscació. Una peça clau del disseny és com la utilitat FileUtil gestiona l'emmagatzematge físic. Quan es crida storeFile, ja sigui amb un MultipartFile o un InputStream, aquesta no utilitza el nom original del fitxer per desar-lo al disc. En lloc d'això, utilitza l'identificador intern i únic de l'entitat, FileEntity.id (un UUID), com a nom del fitxer.

Aquesta tècnica proporciona dos avantatges importants:

1. **Prevenció de col·lisions:** Com que cada ID és un UUID, es garanteix que no hi haurà dos fitxers amb el mateix nom al sistema d'arxius, eliminant la necessitat de gestionar noms duplicats.

2. **Ofuscació i seguretat:** Es desacobla completament el nom que veu l'usuari (FileEntity.name) i l'identificador extern (elementId) del nom real del fitxer al servidor. Això afegeix una capa de seguretat, ja que algú amb accés al sistema d'arxius no podria deduir fàcilment a quin element o usuari pertany cada fitxer només mirant el seu nom.

Finalment, un cop l'element s'ha creat i guardat correctament, el servei corresponent publica un missatge a RabbitMQ per notificar a SyncService que s'ha produït un canvi, el qual s'encarregarà de propagar-lo als clients connectats.

UC-09: Modificar elements

La modificació d'elements inclou operacions com renomenar, moure i copiar, cadascuna amb les seves particularitats tècniques.

UC-09A: Renomenar un element El canvi de nom, igual que el moviment d'elements, es gestiona a través de l'endpoint genèric de modificació de metadades PUT /files/{elementId}. Per a una operació de canvi de nom, el cos de la petició (UpdateFileRequest) conté el nou nom que se li vol assignar a l'element.

Un aspecte fonamental del flux és la seguretat. Com mostra el diagrama, abans de realitzar qualsevol canvi, el controlador invoca el microservei FileAccessControl per verificar que l'usuari té permisos d'escriptura (WRITE) sobre l'element que intenta modificar. Si la validació falla, l'operació es denega immediatament amb un estat 403 Forbidden.

El mètode del controlador que gestiona aquesta lògica centralitza les modificacions de metadades.

```
@PutMapping("/-elementId"")
public ResponseEntity¡?¿ updateElement(@RequestHeader("X-User-Id") UUID
    userId, @RequestHeader("X-Connection-Id") UUID connectionId,
    @RequestBody UpdateFileRequest request, @PathVariable("elementId")
    UUID elementId) -
    elementService.elementNotDeleted(elementId);

    boolean isFolder = elementService.isFolder(elementId);
    // Pas de seguretat clau: validació de permisos
    fileAccessService.checkAccessElement(userId, elementId, isFolder,
    AccessType.WRITE);

    // La petició pot incloure un nou nom i/o un nou pare
    if(request.getParentId() != null) -
        fileAccessService.checkAccessElement(userId, UUID.fromString(
    request.getParentId()), true, AccessType.WRITE);
    "

    if (isFolder) -
        FolderEntity folder = folderService.updateFolderMetadata(
    elementId, request.getName(), UUID.fromString(request.getParentId()))
    ;
        commandService.sendUpdate(folder.getElementId(), connectionId.
    toString(), userId.toString(), folder.getParent(), "", folder.getName
    (), "folder");
    " else -
        FileEntity file = fileService.updateFile(elementId, request.
    getName(), UUID.fromString(request.getParentId()));
        commandService.sendUpdate(file.getElementId(), connectionId.
    toString(), userId.toString(), file.getParent(), fileService.getHash(
    file.getId()), file.getName(), "file");
    "
    return ResponseEntity.accepted().build();
"
```

LISTING 9.13: Endpoint per a l'actualització de metadades a 'ElementController'

Un cop validats els permisos, el controlador determina si l'element és un fitxer o una carpeta i delega l'operació al servei corresponent (FileService o FolderService). Aquests serveis contenen la lògica de negoci, com la validació per evitar noms duplicats a la mateixa carpeta, i executen l'actualització a la base de dades dins d'una

transacció per garantir l'atomicitat. Finalment, es publica un missatge a RabbitMQ per notificar el canvi a SyncService i, conseqüentment, a la resta de clients connectats.

UC-09B: Moure un element La funcionalitat de moure un element actualment es pot invocar a través de dos endpoints diferents, una situació resultant de l'evolució del disseny. Inicialment, existia un endpoint dedicat, PUT /files/{elementId}/move/{folderId}. No obstant això, per optimitzar el funcionament del client d'escriptori, que necessitava poder canviar el nom i la ubicació en una única operació, donat que es possible utilitzar linies de comanda que facin tots dos canvis a la vegada, es va ampliar la funcionalitat de l'endpoint genèric PUT /files/{elementId} (explicat a la secció anterior) perquè també acceptés un canvi del parentId. Aquesta segona aproximació va resultar ser més lògica i robusta, ja que centralitza tota la lògica d'actualització de metadades en un sol punt.

Com a treball futur, es contempla modificar el client web perquè utilitzi l'endpoint d'actualització genèric i poder així eliminar l'endpoint de moviment específic. A causa de la falta de temps, aquest canvi no s'ha realitzat, i el client web continua utilitzant l'endpoint original move. A continuació, es detalla el flux d'aquest mètode.

El flux comença amb una doble validació de permisos a través de FileAccessControl: el sistema verifica que l'usuari té permís d'escriptura tant sobre l'element que vol moure com sobre la carpeta de destinació. Aquesta doble comprovació és crucial per a la seguretat.

```
@PutMapping("/-elementId"/move/-folderId"")
public ResponseEntity¡?¿ moveFolder(@RequestHeader("X-User-Id") UUID
    userId, @RequestHeader("X-Connection-Id") UUID connectionId,
    @PathVariable("elementId") UUID elementId, @PathVariable("folderId")
    UUID folderId) -
    elementService.elementNotDeleted(elementId);

    boolean isFolder = elementService.isFolder(elementId);
    // 1. Validar permís sobre l'element a moure
    fileAccessService.checkAccessElement(userId, elementId, isFolder,
    AccessType.WRITE);
    // 2. Validar permís sobre la carpeta de destí
    fileAccessService.checkAccessElement(userId, folderId, true,
    AccessType.WRITE);

```

```
11     if (isFolder) -
12         // 3. Delegació al servei amb la lògica de negoci
13         folderService.moveFile(elementId, folderId, userId);
14         commandService.sendUpdate(elementId, connectionId.toString(),
       userId.toString(), folderService.getFolderByElementId(folderId, false
       ), "", "", "folder");
15     " else -
16         fileService.moveFile(elementId, folderId, userId);
17         FileEntity fileEntity = fileService.getFile(elementId);
18         commandService.sendUpdate(elementId, connectionId.toString(),
       userId.toString(), folderService.getFolderByElementId(folderId, false
       ), fileService.getHash(fileEntity.getId()), fileEntity.getName(), "
       file");
19     "
20
21     return ResponseEntity.accepted().build();
22 "
```

LISTING 9.14: Endpoint específic per moure un element a 'ElementController'

Un cop superada la validació de permisos, el controlador delega l'operació al servei corresponent. En el cas de les carpetes, un dels reptes tècnics més importants és evitar les dependències circulars (no es pot moure una carpeta dins d'ella mateixa o una de les seves subcarpetes). Per solucionar-ho, abans de confirmar l'operació a la base de dades, el FolderService executa un mètode de validació específic.

```
1 private void validateNoCircularDependency(FolderEntity elementToMove,
      FolderEntity targetParent) -
2     if (elementToMove.getId().equals(targetParent.getId())) -
3         throw new ResponseStatusException(HttpStatus.BAD'REQUEST, "
      Cannot move a folder to itself");
4     "
5
6     // S'obté una llista plana amb els IDs de totes les subcarpetes
      descendents
7     List¡UUID¿ allDescendantIds = getAllDescendantFolderIds(
      elementToMove);
8
9     // Si la carpeta de destí és una de les descendents, es llança una
      excepció
10    if (allDescendantIds.contains(targetParent.getElementId())) -
11        throw new ResponseStatusException(HttpStatus.BAD'REQUEST, "
      Cannot move a folder to one of its own subfolders");
12    "
13 "
14
15 // Mètode auxiliar per obtenir tots els descendents de manera recursiva
16 private List¡UUID¿ getAllDescendantFolderIds(FolderEntity folder) -
17     List¡UUID¿ descendantIds = new ArrayList¡¿();
18     getAllDescendantFolderIdsRecursive(folder, descendantIds);
19     return descendantIds;
20 "
21
22 private void getAllDescendantFolderIdsRecursive(FolderEntity folder,
      List¡UUID¿ accumulator) -
23     for (FolderEntity child : folder.getChildren()) -
24         accumulator.add(child.getElementId());
25         getAllDescendantFolderIdsRecursive(child, accumulator);
26     "
27 "
```

LISTING 9.15: Validació de dependències circulars a 'FolderService'

Primer, fa una comprovació ràpida per veure si l'origen i el destí són la mateixa carpeta. Si no ho són, el mètode getAllDescendantFolderIds recorre de manera recursiva tota l'estructura de subcarpetes de l'element que es vol moure (elementToMove) i en construeix una llista plana amb tots els seus identificadors. Finalment, comprova si l'identificador de la carpeta de destí (targetParent) es troba en aquesta llista. Si és així, significa que s'està intentant fer un moviment il·legal, i es llança una excepció que atura la transacció, evitant que el sistema d'arxius entri en un estat inconsistent. Un cop superada aquesta validació, es procedeix amb el canvi i es notifica a través de RabbitMQ.

UC-09C: Copiar un element La còpia d'un element és una operació de creació que implica duplicar un element existent i tot el seu contingut, si n'hi ha, en una nova ubicació. El procés es gestiona a través de l'endpoint POST /files/{elementId}/copy/{newParentId}.

FIGURA 9.6: Flux de dades per copiar un element, destacat la doble validació de permisos i la naturalesa recursiva del procés.

La seguretat en aquesta operació és clau i requereix una doble validació de permisos: l'usuari ha de tenir permís de **lectura** (READ) sobre l'element original que vol copiar, i permís d'**escriptura** (WRITE) a la carpeta de destinació.

```
@PostMapping("/-elementId"/copy/-newParentId"")
public ResponseEntity¡?¿ copyElement(
        @RequestHeader("X-User-Id") UUID userId,
        @RequestHeader("X-Connection-Id") UUID connectionId,
        @PathVariable("elementId") UUID elementId,
        @PathVariable("newParentId") UUID newParentId
) throws IOException -
    elementService.elementNotDeleted(elementId);

    boolean isFolder = elementService.isFolder(elementId);
    // Doble validació de permisos
    fileAccessService.checkAccessElement(userId, elementId, isFolder,
    AccessType.READ);
    fileAccessService.checkAccessElement(userId, newParentId, true,
    AccessType.WRITE);

```

```
    if (isFolder) -
        folderService.copyFolder(elementId, newParentId, userId,
    connectionId.toString());
        // ... notificació a RabbitMQ ...
    " else -
        fileService.copyFile(elementId, newParentId, userId);
        // ... notificació a RabbitMQ ...
    "

    return ResponseEntity.accepted().build();
"
```

LISTING 9.16: Endpoint per a la còpia d'elements a 'ElementController'

La implementació varia significativament depenent de si l'element és un fitxer o una carpeta:

- **Còpia d'un fitxer:** El FileService crea una nova entitat FileEntity amb un nou elementId i la desa a la base de dades. A continuació, invoca el mètode fileUtil.copyFile(originalId, newId), que realitza la còpia física dels bytes del fitxer original al nou fitxer en el sistema d'emmagatzematge, utilitzant els identificadors interns.

- **Còpia d'una carpeta:** Aquesta és l'operació més complexa, ja que requereix una còpia en profunditat (*deep copy*). El FolderService implementa un mètode recursiu que realitza les següents accions:

  1. Crea una nova carpeta (la còpia) a la ubicació de destí.

  2. Itèra sobre tots els fitxers de la carpeta original. Per a cada un, crida el mètode fileService.copyFile per duplicar-lo dins de la nova carpeta.

  3. Itèra sobre totes les subcarpetes de la carpeta original. Per a cada una, es crida a si mateix de manera recursiva, passant la subcarpeta original i la nova carpeta com a nou pare.

```
private FolderEntity copyFolderRecursive(FolderEntity folderToCopy,
    FolderEntity newParent, UUID userId, String connectionId) throws
    IOException -
    // 1. Es crea la nova carpeta en el destí
    FolderEntity newFolder = createFolder(folderToCopy.getName(),
    newParent, userId, false, connectionId);

    // 2. Es copien tots els fitxers de dins
    for (FileEntity fileToCopy : folderToCopy.getFiles()) -
        if(fileToCopy.getDeleted()) continue;
        fileService.copyFile(fileToCopy.getElementId(), newFolder.
    getElementId(), userId);
    "

    // 3. Es crida recursivament per a cada subcarpeta
    for (FolderEntity subFolderToCopy : folderToCopy.getChildren()) -
        if(subFolderToCopy.getDeleted()) continue;
        copyFolderRecursive(subFolderToCopy, newFolder, userId,
    connectionId);
    "

    return newFolder;
```

```
"
```

LISTING 9.17: Lògica de còpia recursiva a 'FolderService'

Aquest procés garanteix que tota l'estructura de directoris es dupliqui de manera íntegra. Igual que en les altres operacions, un cop finalitzada la còpia, s'envien els missatges corresponents a RabbitMQ per notificar la creació dels nous elements.

UC-10: Descarregar elements

La descàrrega d'elements es fa mitjançant l'endpoint GET /files/{elementId}/download. Aquest endpoint permet descarregar tant fitxers com carpetes (retornades com fitxers zip amb tot el contingut de la carpeta). La implementació utilitza streaming, de manera que les dades es llegeixen i s'envien al client directament des del disc, sense carregar tot el contingut a la memòria del servidor.

El procés de descàrrega segueix uns passos ben definits:

1. **Validació de permisos:** Com sempre, el primer pas és una crida a FileAccessControl per assegurar que l'usuari té permisos de lectura (READ) sobre l'element que vol descarregar.

2. **Bifurcació de la lògica:** Es comprova si l'element és un fitxer o una carpeta per determinar com s'ha de gestionar la descàrrega.

3. **Construcció de la resposta:** El controlador construeix una resposta HTTP del tipus ResponseEntity¡Resource¿, establint les capçaleres necessàries (Content-Type: application/octet-stream i Content-Disposition: attachment) perquè el navegador iniciï una descàrrega.

```
@GetMapping(value = "/-elementId"/download", produces = MediaType.
    APPLICATION_OCTET_STREAM_VALUE)
public ResponseEntity¡?¿ serveFile(@RequestHeader("X-User-Id") UUID
    userId, @PathVariable("elementId") UUID elementId) -
    Resource resource;
    String filename;

```

```
6     boolean isFolder = elementService.isFolder(elementId);
7     fileAccessService.checkAccessElement(userId, elementId, isFolder,
      AccessType.READ);
8
9     if (isFolder) -
10        FolderEntity folder = folderService.getFolderByElementId(
      elementId, false);
11        resource = folderService.downloadFolder(elementId);
12        filename = folder.getName() + ".zip";
13    " else -
14        FileEntity fileEntity = fileService.getFile(elementId);
15        resource = fileUtil.loadAsResource(fileEntity.getId());
16        filename = fileEntity.getName();
17    "
18
19    // ... construcció de la resposta amb capçaleres ...
20
21    return response.body(resource);
22 "
```

LISTING 9.18: Endpoint per a la descàrrega d'elements a 'ElementController'

La gestió del recurs varia segons el tipus d'element:

- **Si és un fitxer**, el controlador invoca fileUtil.loadAsResource, que obté un InputStream del fitxer emmagatzemat al disc (identificat pel seu ID intern) i l'embolcalla en un objecte Resource sense carregar-lo completament a la memòria.

- **Si és una carpeta**, la tasca és més complexa. El FolderService s'encarrega de crear un fitxer ZIP "al vol". Recorre l'estructura de la carpeta, afegeix cada fitxer i subcarpeta a un ZipOutputStream que escriu directament al flux de sortida de la resposta HTTP. D'aquesta manera, es poden descarregar carpetes de mida considerable sense esgotar la memòria del servidor.

Aquesta implementació basada en *streaming* és un exemple clar d'optimització, ja que proporciona una solució eficient i escalable que funciona de manera transparent per a qualsevol client capaç de processar una resposta HTTP estàndard.

9.3.4 Compartició (FileSharing)

El microservei FileSharing gestiona la lògica de compartir arxius i carpetes entre usuaris, una de les funcionalitats centrals del sistema. La seva implementació materialitza els casos d'ús **UC-13** (Compartir), **UC-13A** (Revocar accés) i **UC-13B** (Deixar de seguir un element compartit). Una de les fortaleses del disseny del sistema és com la funcionalitat de crear un element dins d'una carpeta compartida s'integra de manera natural en el flux de creació ja existent, gràcies a l'abstracció que proporciona el servei de control d'accés.

Com s'ha vist a la secció anterior (UC-08), quan un usuari crea un fitxer o una carpeta, el servei corresponent a FileManagement comprova si l'ID de l'usuari que fa la petició és el mateix que el del propietari de la carpeta pare. Si no coincideixen, el sistema interpreta que l'acció es realitza dins d'una carpeta compartida. En aquest punt, a més de crear l'element, fa una crida interna a FileSharing per registrar la

compartició i assignar els permisos corresponents: d'escriptura (WRITE) per al creador i d'administrador (ADMIN) per a l'amo original. Aquest comportament es pot observar al següent fragment de codi.

```
// ... dins del metode de creacio de fitxers ...
if(userId.equals(folder.getUserId())) -
    // Si l'usuari es el propietari de la carpeta, s'assigna ADMIN sobre
     el nou fitxer
    fileAccessService.addFileAccess(fileEntity.getElementId(), userId,
    AccessType.ADMIN);
" else -
    // Si no es el propietari, es gestiona com un element en una carpeta
     compartida
    UserDTO userDetails = userManagementClient.get(userId);
    fileShareClient.shareFileInternal(userId, connectionId, new
    SharedRequest(file.getElementId(), folder.getElementId(), userDetails
    .getUsername(), AccessType.WRITE));
    fileAccessService.addFileAccess(fileEntity.getElementId(), userId,
    AccessType.WRITE);
    fileAccessService.addFileAccess(fileEntity.getElementId(), folder.
    getUserId(), AccessType.ADMIN);
"
```

LISTING 9.19: Gestió de permisos en crear un fitxer a 'FileService'

UC-13: Compartir un element

El procés per compartir un element amb un altre usuari s'inicia a través de l'endpoint POST /share. El servei orquestra una sèrie de validacions i operacions per garantir la integritat i la seguretat.

La lògica, encapsulada a FileSharingService, segueix aquests passos:

1. **Validació de permisos de l'emissor:** Es fa una crida a FileAccessControl per comprovar que l'usuari que inicia la compartició té permisos d'ADMIN sobre

l'element. Això és una mesura de seguretat per asegurar que nomes el propietari del element el pot compartir.

2. **Obtenció de l'usuari receptor:** Es contacta amb UserManagement per obtenir l'ID de l'usuari amb qui es vol compartir, a partir del seu nom d'usuari (les ids dels usuaris no surten mai del sistema, tota funcio que es favi amb un usuari que no sigui el que crida i es pot identificar amb el JWT es fa em el nom d'usuari que també l'identifica.).

3. **Creació de la regla d'accés principal:** Es crida a FileAccessControl per crear la nova regla d'accés que atorga al receptor el permís especificat sobre l'element.

4. **Creació del registre de compartició:** Es crea i desa una entitat SharedAccess, que serveix per identificar quins elements són l'arrel d'una compartició per a un usuari determinat. A més, s'assigna un valor booleà (root) a aquest registre. Aquest indicador és clau: si un element es comparteix directament, es marca com a arrel (root = true). Aquesta distinció és fonamental perquè el client web pugui construir la vista Çompartit amb mi"de manera eficient, mostrant només els elements arrel al primer nivell. La navegació posterior dins d'aquests elements ja funciona de manera estàndard, sense necessitar un tractament especial.

```
boolean isRootShared;
if (parent != null) -
    // Si l'element te un pare, es comprova si el pare ja estava compartit amb l'usuari
    Optional¡SharedAccess¿ parentSharedAccess = sharedAccessRepository.findByElementIdAndUserId(parent.getId(), userId);
    isRootShared = !parentSharedAccess.isPresent();
" else -
    // Si no te pare, es una comparticio arrel per definicio
    isRootShared = true;
"
sharedAccess.setRoot(isRootShared);
```

LISTING 9.20: Lògica per determinar si una compartició és arrel a 'FileSharingService'

5. **Propagació de permisos (recursiva):** Si l'element compartit és una carpeta, el servei obté la llista de tots els seus descendents i crida a FileAccessControl per crear les regles d'accés corresponents per a cada un, assegurant que el permís es propagui a tota l'estructura.

6. **Notificació asíncrona:** Finalment, es publica un missatge a RabbitMQ per notificar a SyncService que l'element ha estat compartit.

```
@Transactional
public void shareElement(UUID userId, ShareRequest shareRequest, String connectionId) -
    // 1. Validació de permisos
    AccessResponse requesterAccess = fileAccessControlClient.getFileAccess(
        shareRequest.getElementId(), userId);
    if (requesterAccess == null —— requesterAccess.getAccessType() != AccessType.ADMIN.ordinal()) -
        throw new ResponseStatusException(HttpStatus.FORBIDDEN, "Admin permission required...");
```

```
8       "
9
10      // 2. Obtenció del receptor
11      UUID shareWithUserId = userManagementClient.getUserId(shareRequest.
        getUser());
12
13      // 3. Creació de la regla d'accés
14      AccessRequest accessRequest = new AccessRequest(shareWithUserId,
        shareRequest.getElementId(), shareRequest.getAccessType().ordinal());
15      fileAccessControlClient.createAccess(accessRequest);
16
17      // 4. Creació del registre SharedAccess
18      SharedAccess sharedAccess = new SharedAccess();
19      // ... s'estableixen les dades ...
20
21      // 5. Propagació recursiva de permisos
22      try -
23          List ¡UUID¿ ids = fileManagementClient.getChildren(shareRequest.
        getElementId());
24          updateChildren(ids, shareWithUserId, shareRequest.getAccessType
        ().ordinal());
25      " catch (FeignException.NotFound e) -"
26
27      sharedAccessRepository.save(sharedAccess);
28
29      // 6. Notificació
30      commandService.sendShared(shareRequest.getElementId(), connectionId,
         userId.toString(), parentIdString);
31  "
```

LISTING 9.21: Lògica principal per compartir un element a 'FileSharingService'

Perquè el client web pugui mostrar aquesta llista d'elements compartits, s'ha creat un endpoint específic, GET /share/root. Aquest retorna una carpeta virtual que conté tots els elements marcats com a root = true per a l'usuari que fa la petició. El flux de crides per a aquesta operació, demostra la naturalesa col·laborativa de l'arquitectura: el servei FileSharing actua com a orquestrador, consultant la seva pròpia base de dades i comunicant-se amb FileManagement, UserManagement i FileAccessControl per enriquir la informació abans de retornar-la, tal com es mostra al següent diagrama.

FIGURA 9.7: Diagrama de flux per obtenir la carpeta virtual d'elements compartits.

El controlador exposa l'endpoint, que simplement delega la feina al servei.

Aquest endpoint ara mateix no esta optimitzat per a dades complexes, ja que es fa una crida a FileManagement per a cada element compartit, i després es fa una crida a UserManagement per a cada element compartit. Aixo es pot millorar en futur, per exemple, fent una sola crida a FileManagement per a tots els elements compartits i després iterar sobre els resultats per enriquir-los amb les dades del propietari real de cada element (obtingudes de UserManagement) i el nivell de permís de l'usuari actual (des de FileAccessControl) i/o amb la implementació d'una cache de dades a nivell de servei.

```
1 @GetMapping("/share/root")
2 public ResponseEntity¡FolderInfo¿ getSharedWithUserRoot(@RequestHeader("
    X-User-Id") UUID userId) -
3     FolderInfo sharedRoot = fileSharingService.getSharedWithUserRoot(
    userId);
4     return ResponseEntity.ok(sharedRoot);
5 "
```

LISTING 9.22: Endpoint per obtenir la carpeta arrel compartida a 'FileSharingController'

La implementació al servei recull tots els registres de SharedAccess d'un usuari, filtra aquells que són arrel, i fa una crida a FileManagement per obtenir l'estructura de dades completa d'aquests elements. Posteriorment, itera sobre els resultats per enriquir-los amb les dades del propietari real de cada element (obtingudes de UserManagement) i el nivell de permís de l'usuari actual (des de FileAccessControl). El resultat és un objecte FolderInfo que representa la carpeta Çompartit amb mi", llest per ser renderitzat pel client.

```
1 public FolderInfo getSharedWithUserRoot(UUID userId) -
2     // 1. Obtenir tots els registres d'elements compartits directament
    amb l'usuari
3     List¡SharedAccess¿ sharedAccesses = sharedAccessRepository.
    findByUserIdAndRoot(userId, true);
4
```

```
5       // ... (s'obtenen també els no arrel per a informació addicional
        posterior) ...
6
7       // 2. Extreure els IDs dels elements arrel
8       List ¡UUID¿ elementIds = sharedAccesses.stream()
9           .map(SharedAccess::getElementId)
10          .collect(Collectors.toList());
11
12      if (elementIds.isEmpty()) -
13          return new FolderInfo(); // Retorna una carpeta buida si no hi
        ha res compartit
14      "
15
16      try -
17          // 3. Demanar a FileManagement l'estructura d'aquests elements
18          FolderInfo folderInfo = fileManagementClient.getFolderStructure(
        elementIds, false);
19          // 4. Afegir detalls de la compartició (propietari, etc.)
20          setDetails(folderInfo, map);
21          return folderInfo;
22      " catch (Exception e) -
23          e.printStackTrace();
24          return new FolderInfo();
25      "
26  "
```

LISTING 9.23: Construcció de la carpeta virtual d'elements compartits a 'FileSharingService'

UC-13A i UC-13B: Revocar accés i Deixar de compartir

Tant la revocació de l'accés per part del propietari (**UC-13A**) com l'acció de deixar de seguir un element per part del receptor (**UC-13B**) es gestionen a través del mateix endpoint: DELETE /share/{elementId}/user/{username}. La implementació distingeix entre els dos casos d'ús basant-se en qui fa la petició.

FIGURA 9.8: Diagrama de flux per revocar o deixar de seguir un element (UC-13A/B).

El servei revokeShare implementa la següent lògica:

1. **Obtenció de l'usuari a eliminar:** Es crida a UserManagement per obtenir l'ID de l'usuari que perdrà l'accés.

2. **Validació de permisos (condicional):** Si l'usuari que fa la petició (requesterId) no és el mateix que l'usuari a qui se li revoca l'accés (userIdToRemove), el sistema assumeix que és el propietari qui fa l'acció (UC-13A) i, per tant, valida que tingui permisos d'ADMIN. Si els dos IDs són iguals, significa que l'usuari està abandonant la compartició (UC-13B), i aquesta comprovació de permisos s'omet.

3. **Eliminació de regles i registres:** Es crida a FileAccessControl per eliminar la regla d'accés principal i, de manera recursiva, les de tots els fills. Paral·lelament, s'elimina el registre corresponent de la taula SharedAccess.

4. **Notificació asíncrona:** Finalment, es publica un missatge a RabbitMQ per notificar a SyncService que la compartició s'ha eliminat.

```
@Transactional
public void revokeShare(UUID requesterId, UUID elementId, String
    usernameToRemove, String connectionId) -
    // 1. Obtenció de l'usuari
    UUID userIdToRemove = userManagementClient.getUserId(
    usernameToRemove);

    // 2. Validació condicional de permisos
    if (!requesterId.equals(userIdToRemove)) -
        AccessResponse requesterAccess = fileAccessControlClient.
    getFileAccess(elementId, requesterId);
        if (requesterAccess == null -- requesterAccess.getAccessType()
    != 3) - // 3 = ADMIN
            throw new ResponseStatusException(HttpStatus.FORBIDDEN, "
    Admin permission required...");
        "
    "

    // 3. Eliminació de regles i registres
    fileAccessControlClient.deleteAccess(elementId, userIdToRemove);
    sharedAccessRepository.deleteByElementIdAndUserId(elementId,
    userIdToRemove);

    try -
        List ¡UUID¿ children = fileManagementClient.getChildren(elementId
    );
        revokeChildrenAccess(children, userIdToRemove);
    " catch (FeignException.NotFound e) -"

    // 4. Notificació
    commandService.sendUnshared(elementId, connectionId, requesterId.
    toString());
"
```

LISTING 9.24: Lògica per revocar un accés a 'FileSharingService'

Aquest disseny dual en un sol mètode és eficient i reutilitza la lògica per a dos casos d'ús que, tot i ser conceptualment diferents per a l'usuari, tècnicament són molt similars.

9.3.5 Papelera i eliminació asíncrona (TrashService)

El microservei TrashService implementa la funcionalitat de la paperera de reciclatge. La seva responsabilitat no es limita a gestionar un estat temporal per als fitxers eliminats, sinó que també orquestra el procés d'eliminació permanent de dades, una operació crítica per a la integritat del sistema. La seva implementació dona servei als casos d'ús **UC-11** (Enviar a la paperera), **UC-17** (Restaurar) i **UC-12** (Eliminar permanentment).

UC-11 i UC-17: Borrat lògic i restauració d'elements

Una decisió fonamental del disseny és que enviar un element a la paperera no n'implica l'eliminació física immediata. En lloc d'això, es realitza un **borrat lògic** (*soft delete*). Aquest procés es gestiona a través de l'endpoint DELETE /files/{elementId}, que el Gateway redirigeix al TrashService.

Com mostra el diagrama, el servei primer valida els permisos i després orquestra una operació en dues parts:

1. Fa una crida síncrona a FileManagement perquè marqui l'element i tots els seus descendents com a "eliminats". Aquests continuen existint a la base de dades i al disc, però amb un indicador que els oculta de les vistes normals.

2. Per a cada element mogut, crea un registre a la seva pròpia base de dades, l'entitat TrashRecord. Aquest registre emmagatzema qui l'ha eliminat, la data i, crucialment, una data de caducitat (30 dies per defecte), a més d'un flag per saber si era l'element arrel de l'operació.

```
1 public void moveToTrash(UUID userId, UUID connectionId, UUID elementId)
    -
2     try -
3         AccessResponse accessResponse = fileAccessControlClient.
    getFileAccess(elementId, userId);
4         if (accessResponse.getAccessType() ¡= AccessType.ADMIN.ordinal()
    ) -
5             throw new ResponseStatusException(HttpStatus.FORBIDDEN, "
    User does not have WRITE access to the element.");
```

```
6           "
7       " catch (Exception e) -
8           throw new ResponseStatusException(HttpStatus.FORBIDDEN, "
        Permission check failed for the element");
9       "
10
11      SetDeletedRequest request = new SetDeletedRequest(true);
12      SetDeletedResponse response;
13      try -
14          response = fileManagementClient.setElementDeletedState(userId,
        connectionId, elementId, request);
15          if (response == null -- response.getElementIds() == null --
        response.getElementIds().isEmpty()) -
16              throw new ResponseStatusException(HttpStatus.
        INTERNAL_SERVER_ERROR, "Failed to move element to trash");
17          "
18      " catch (Exception e) -
19          throw new ResponseStatusException(HttpStatus.
        INTERNAL_SERVER_ERROR, "Error communicating with file management
        service");
20      "
21
22      List¡UUID¿ affectedIds = response.getElementIds();
23      for (UUID affectedId : affectedIds) -
24          boolean isRoot = affectedId.equals(elementId);
25          updateOrCreateRecord(userId, affectedId, isRoot);
26      "
27  "
```

LISTING 9.25: Mètode per moure un element a la paperera a 'TrashService'

La restauració d'un element (**UC-17**), invocada amb PUT /trash/{elementId}/restore, simplement reverteix aquest procés: es torna a cridar FileManagement per marcar els elements com a no eliminats (deleted=false) i s'esborren els registres corresponents de la taula TrashRecord.

UC-12: Eliminació permanent i orquestració asíncrona

L'eliminació permanent o **borrat físic** (*hard delete*) és una operació molt més complexa i dissenyada per ser resilient i completament traçable. Pot ser iniciada per l'usuari des de la paperera o automàticament per una tasca programada quan un element caduca. En ambdós casos, el procés comença canviant l'estat del TrashRecord a PENDING_DELETION i orquestra una sèrie de crides síncrones i asíncrones amb un sistema de confirmació per garantir que cap dada residual quedi al sistema.

El flux complet es pot dividir en tres fases:

1. **Fase 1: Eliminació síncrona del contingut.** El primer pas, i l'únic síncron, és una crida a FileManagement per eliminar de manera irreversible el fitxer del disc i les seves metadades principals. Un cop confirmat, TrashService actualitza el TrashRecord corresponent, canviant el seu estat a PENDING_DELETION i marcant el flag manager a true, indicant que aquesta part del procés s'ha completat.

```
public void deletePermanently(UUID userId, UUID elementId) -
    TrashRecord trashRecord = trashRecordRepository.findByElementId(elementId)
            .orElseThrow(() -¿ new ResponseStatusException(HttpStatus.NOT'FOUND, "Element not found...")); 

    if (!trashRecord.getUserId().equals(userId)) -
        throw new ResponseStatusException(HttpStatus.FORBIDDEN, "User does not have permission...");
    "
    try -
        // Crida síncrona per eliminar el fitxer físic
        SetDeletedResponse response = fileManagementClient.deleteElementPermanently(elementId);

        // Canvi d'estat per iniciar la purga asíncrona
        List¡TrashRecord¿ trashRecords = trashRecordRepository.findByUserIdAndElementIdIn(userId, response.getElementIds());
        trashRecords.forEach(record -¿ -
            record.setStatus(RecordStatus.PENDING'DELETION);
            record.setManager(true); // Marca la part de FileManagement com a feta
            record.setAccess(false);
            record.setSharing(false);
        ");
        trashRecordRepository.saveAll(trashRecords);

    " catch (Exception e) - /* ... */ "
"
```

LISTING 9.26: Inici de l'eliminació permanent a 'TrashService'

2. **Fase 2: Neteja asíncrona amb reintent i confirmació.** Aquí entra en joc el mecanisme de tolerància a fallades gestionat per la tasca programada processPendingDeletions. Aquesta s'executa periòdicament, cerca tots els registres en estat PENDING_DELETION i, per a cada flag de confirmació que estigui a false (access o sharing), reenvia el missatge de neteja corresponent a la cua de RabbitMQ.

```
public void processPendingDeletions() -
    List¡TrashRecord¿ pendingRecords = trashRecordRepository.
    findByStatus(RecordStatus.PENDING'DELETION);

    for (TrashRecord record : pendingRecords) -
        // ... Lògica per notificar a SyncService quan tot ha acabat ...

        // Si la neteja d'accessos no s'ha confirmat, es reenvia la
    comanda
        if(!record.isAccess()) -
            sender.removeAccess(record.getUserId(), record.getElementId
    ());
        "

        // Si la neteja de comparticions no s'ha confirmat, es reenvia
        if(!record.isSharing()) -
            sender.removeSharing(record.getUserId(), record.getElementId
    ());
        "
    "
"
```

LISTING 9.27: Lògica de processament de pendents a 'TrashService'

Cada servei consumidor (FileAccessControl, FileSharing), després de processar la seva cua i eliminar les dades, envia un missatge de confirmació a la cua CONFIRM_DELETE. El TrashService escolta aquesta cua i, en rebre un missatge, actualitza el flag booleà corresponent del TrashRecord a true.

```
@RabbitListener(queues = RabbitConfig.CONFIRM'DELETE, messageConverter =
    "jackson2JsonMessageConverter")
public void confirmDeleteAccess(Map¡String, Object¿ payload) -
    UUID elementId = UUID.fromString((String) payload.get("elementId"));
    UUID userId = UUID.fromString((String) payload.get("userId"));
    String service = (String) payload.get("service");
    trashService.confirm(elementId, userId, service);
"
```

LISTING 9.28: Receptor de confirmacions a 'TrashService'

3. **Fase 3: Neteja final del registre.** Un cop tots els serveis han confirmat la seva part de la neteja, el TrashRecord tindrà tots els seus flags (manager, access, sharing) a true. En aquest punt, el registre ja ha complert la seva funció. Una altra tasca programada, clean(), s'executa diàriament i esborra de la base de dades tots aquells TrashRecord que ja estiguin completament processats, mantenint el sistema net i eficient.

Aquest disseny basat en una màquina d'estats per registre, reintents automàtics i un cicle de confirmació explícit, garanteix una eliminació de dades robusta, resilient i auditable a tot l'ecosistema de microserveis.

9.3.6 Servei de sincronització (SyncService)

El microservei SyncService és el component final de l'arquitectura del backend i el responsable d'una de les funcionalitats més complexes i importants del sistema: mantenir tots els clients sincronitzats en temps real (**UC-14**). La seva principal responsabilitat és gestionar les connexions WebSocket, processar els esdeveniments de canvi publicats per altres serveis i difondre les actualitzacions als clients corresponents.

L'arquitectura del servei es basa en diversos components clau:

- **Gestor de WebSockets:** Un WebSocketHandler que gestiona el cicle de vida de les connexions dels clients (web i escriptori).
- **Consumidor de RabbitMQ:** Un *listener* que se subscriu a la cua d'esdeveniments on altres microserveis (com FileManagement o FileSharing) publiquen canvis.
- **Servei de Snapshots:** Un servei que manté a la base de dades una representació de l'arbre de fitxers de cada usuari (un "snapshot") i el reconstrueix cada cop que es produeix un canvi.

Flux de connexió WebSocket

El procés de connexió s'inicia quan un client vol establir una comunicació en temps real. Com es va explicar a la secció del Gateway, aquest valida el token JWT i enriqueix la petició injectant l'ID de l'usuari i un ID de connexió únic a les capçaleres abans de redirigir la connexió a SyncService.

Dins de SyncService, la configuració de Spring WebSocket s'encarrega de registrar els *handlers* i un interceptor clau per a aquest procés.

```
1 @Configuration
2 @EnableWebSocket
```

```
public class WebSocketConfig implements WebSocketConfigurer -
    @Override
    public void registerWebSocketHandlers(@NonNull
    WebSocketHandlerRegistry registry) -
        RawWebSocketHandler handler = rawWebSocketHandler();

        // Endpoint per al client d'escriptori
        registry.addHandler(handler, "/websocket")
                .setAllowedOrigins("*")
                .addInterceptors(new UserIdHandshakeInterceptor());

        // Endpoint per al client web
        registry.addHandler(handler, "/websocket/web")
                .setAllowedOrigins("*")
                .addInterceptors(new UserIdHandshakeInterceptor());
    "

    @Bean
    public RawWebSocketHandler rawWebSocketHandler() -
        return new RawWebSocketHandler();
    "
"
```

LISTING 9.29: Configuració del WebSocket a 'SyncService'

L'interceptor UserIdHandshakeInterceptor s'executa abans de l'establiment de la connexió i extreu l'ID de l'usuari i de la connexió de les capçaleres HTTP, injectant-los als atributs de la sessió WebSocket. Aquesta és una peça fonamental que permet associar cada connexió a un usuari concret.

```
@Override
public boolean beforeHandshake(ServerHttpRequest request,
    ServerHttpResponse response,
                                WebSocketHandler wsHandler, Map¡String,
    Object¿ attributes) throws Exception -
    try -
        String userId = getHeaderValue(request, "X-User-Id");
        String connectionId = getHeaderValue(request, "X-Connection-Id")
    ;

        if (userId != null && connectionId != null) -
            attributes.put("userId", userId);
            attributes.put("connectionId", connectionId);
            return true;
    " else -
            return false;
        "
    " catch (Exception e) - return false; "
"
```

LISTING 9.30: Interceptor per extreure IDs d'usuari a 'UserIdHandshakeInterceptor'

Un cop la connexió arriba al RawWebSocketHandler, aquest guarda la sessió en un mapa en memòria, diferenciant si prové del client web o del d'escriptori. Aquesta distinció és clau, ja que l'estratègia de notificació varia: per a les connexions d'escriptori, en establir-se la connexió, s'envia immediatament un *snapshot* complet de l'arbre de fitxers de l'usuari.

Un altre aspecte fonamental del disseny és l'ús del connectionId. Aquest identificador únic, generat a cada inici de sessió i inclòs al token JWT, es propaga a totes les peticions i missatges asíncrons que s'originen en una connexió de client específica. La seva finalitat és evitar bucles de notificació infinits. Quan un client realitza una acció (p. ex., crear un fitxer), el canvi es propaga a través de RabbitMQ fins al SyncService, que el reenvia a tots els clients de l'usuari. No obstant això, el client que va originar l'acció no ha de rebre aquesta notificació, ja que ja ha actualitzat la seva pròpia vista localment.

El RawWebSocketHandler implementa aquesta lògica de manera explícita: abans d'enviar una notificació a les sessions d'un usuari, comprova si l'identificador de la sessió coincideix amb el connectionId que ve amb el missatge de l'esdeveniment i, si és així, l'omet. Això garanteix que un usuari amb múltiples sessions (p. ex., el client web i el d'escriptori oberts alhora) rebi les actualitzacions a totes les seves pantalles, excepte a la que va iniciar el canvi.

```
1  public void sendSnapshot(String userId, String message, String
       connectionId) throws IOException -
2      Map¡String, WebSocketSession¿ sessions = snapshotSessions.get(userId
       );
3      if (sessions != null) -
4          for(String connection : sessions.keySet()) -
5              // Es comprova que no sigui la connexió que va originar el
       canvi
6              if(!connection.equals(connectionId)) -
7                  if(sessions.get(connection) != null && sessions.get(
       connection).isOpen()) -
8                      sendMessage(sessions.get(connection), message);
9                  "
10             "
11         "
12     "
13 "
```

LISTING 9.31: Exclusió de la connexió origen a 'RawWebSocketHandler'

Processament d'esdeveniments i actualització de l'estat

El nucli de la sincronització esta en el processament dels esdeveniments. Quan el consumidor de RabbitMQ rep un missatge (p. ex., "s'ha creat un fitxer"), invoca el SnapshotService per actualitzar la representació de l'arbre de fitxers a la base de dades.

```
1  @RabbitListener(queues = RabbitConfig.COMMAND'QUEUE, messageConverter =
       "jackson2JsonMessageConverter")
2  public void processCommand(Map¡String, Object¿ payload) -
3      try -
4          CommandRabbit command = new CommandRabbit(/* ... mapping from
       payload ... */);
5
6          // La lògica de (des)compartir no afecta l'estructura de l'
       snapshot directament
7          if(!command.action().equals("unshared-element") && !command.
       action().equals("shared-element")) -
8              SnapshotEntity snapshot = snapshotService.processCommand(
       command);
9              // Envia el snapshot complet al client d'escriptori
```

```
10                 websocketService.sendSnapshot(snapshot, command.userId(),
    command.connectionId());
11             ”
12             // Envia una comanda simple al client web
13             websocketService.sendWebCommand(command.action(), command.userId
    (), command.connectionId(), command.elementId(), command.parentId());
14         ” catch (Exception e) –
15             e.printStackTrace();
16         ”
17 ”
```

LISTING 9.32: Listener de RabbitMQ per processar comandes a ‘Receiver‘

El mètode snapshotService.processCommand és el cor de la lògica d’estat. Rep la comanda i modifica les entitats SnapshotEntity i SnapshotElementEntity per reflectir el canvi. Un dels aspectes més importants és el recàlcul de hashes: cada cop que un element canvia, es recalcula el seu hash i es propaga el canvi cap amunt, actualitzant el hash de totes les carpetes pare fins a l’arrel. Això permet que el client d’escriptori detecti un canvi en qualsevol part de l’arbre simplement comparant el hash de la carpeta arrel.

```
1  private void recalculateHash(SnapshotElementEntity element) –
2      if (element != null && ”folder”.equals(element.getType())) –
3          String newHash = calculateFolderHash(element);
4          element.setHash(newHash);
5          snapshotElementRepository.save(element);
6
7          if (element.getParent() != null) –
8              // Crida recursiva cap a l’arrel
9              recalculateHash(element.getParent());
10         ”
11     ”
12 ”
13
14 private String calculateFolderHash(SnapshotElementEntity folder) –
15     if (folder.getContent() == null —— folder.getContent().isEmpty()) –
16         return DigestUtils.sha256Hex(”);
17     ”
18     StringBuilder elementHash = new StringBuilder();
19     // Ordenar per nom garanteix un hash consistent
20     for (SnapshotElementEntity content : folder.getContent().stream().
    sorted(Comparator.comparing(SnapshotElementEntity::getName)).toList()
    ) –
21         if (content.getHash() != null) –
22             elementHash.append(content.getHash());
23         ”
24     ”
25     return DigestUtils.sha256Hex(elementHash.toString());
26 ”
```

LISTING 9.33: Recàlcul recursiu de hashes a ‘SnapshotService‘

Estratègies de difusió i punts clau

Un cop el *snapshot* s’ha actualitzat, el WebSocketService s’encarrega de difondre el canvi. La lògica de difusió és diferent segons el tipus de client:

- **Per al client web:** S’envia un esdeveniment simple, com updated_tree, que indica al client que ha de tornar a demanar les dades de la carpeta afectada.

Aquesta estratègia és lleugera i eficient per a un client que només visualitza una part de l'arbre de fitxers a la vegada.

- **Per al client d'escriptori:** S'envia el *snapshot* complet actualitzat. El client compara el nou hash de l'arrel amb el que tenia localment; si són diferents, substitueix tot el seu estat local pel nou snapshot. Això garanteix una consistència total amb el servidor.

Aquesta doble estratègia es pot veure clarament reflectida a la implementació del WebSocketService, que conté mètodes especialitzats per a cada tipus de client.

```
@Transactional
public void sendSnapshot(SnapshotEntity snapshot, String userId, String connectionId) -
    try -
        SnapshotElementEntity root = snapshot.getElements().stream().filter(element -¿ element.getParent() == null).findFirst().orElse(null);
        String snapshotMessage = createMessage("SNAPSHOT", "snapshot˙update", convertElementToMap(root));

        webSocketHandler.sendSnapshot(userId, snapshotMessage, connectionId);
    " catch (Exception e) -
        e.printStackTrace();
    "
"
```

LISTING 9.34: Enviament del snapshot complet al client d'escriptori a 'WebSocketService'

Aquest mètode, 'sendSnapshot', s'invoca quan el destinatari és el client d'escriptori. Construeix un missatge de tipus SNAPSHOT que conté l'arbre de fitxers complet serialitzat i utilitza el webSocketHandler per enviar-lo exclusivament a les connexions d'escriptori de l'usuari.

```
public void sendWebCommand(String action, String userId, String connectionId, String elementId, String parentId) -
    try -
        if(parentId == null —— parentId.isEmpty()) -
            parentId = fileManagementClient.getParentId(UUID.fromString(elementId)).toString();
        "

        String message = createMessage("COMMAND", "updated˙tree", Map.of("elementId", elementId, "parentId", parentId, "section", getSection(action)));

        webSocketHandler.sendWebCommand(userId, message, connectionId);

        List¡SharedInfo¿ users = fileShareClient.getUsersShared(UUID.fromString(elementId));

        message = createMessage("COMMAND", "updated˙tree", Map.of("elementId", elementId, "parentId", parentId, "section", "shared"));
        for(SharedInfo user : users) -
            UUID id = userManagementClient.getUserId(user.getUsername());
            webSocketHandler.sendWebCommand(id.toString(), message, "");
        "
    " catch (Exception e) -
```

```
19            e.printStackTrace();
20        "
21    "
```

LISTING 9.35: Enviament d'una comanda de refresc al client web a 'WebSocketService'

En canvi, el mètode 'sendWebCommand' està dissenyat per al client web. En lloc d'enviar tot l'estat, crea un missatge de tipus COMMAND molt més lleuger. Aquest missatge conté únicament els identificadors necessaris perquè el client sàpiga quina part de la seva vista ha d'actualitzar, fent una nova petició per obtenir només les dades que han canviat. A més, aquest mètode s'encarrega de notificar a altres usuaris si el canvi afecta un element compartit.

Per garantir la robustesa, he implementat mecanismes per gestionar la pèrdua de connexions (*idle timeouts*) i una estratègia de reconnexió simple per part dels clients. El sistema es basa en la **consistència eventual**: tot i que poden existir petites latències, es garanteix que tots els clients acabaran rebent tots els canvis. La gestió de l'estat de les connexions en memòria és un punt crític; en un escenari de producció a gran escala, aquest estat s'hauria d'externalitzar a un magatzem compartit com Redis per permetre l'escalat horitzontal del propi SyncService.

9.3.7 Refactor del codi

Durant el desenvolupament del projecte, vaig arribar a un punt on vaig haver de replantejar-me seriosament l'arquitectura del sistema i fer una refactorització important. Al principi, quan estava dissenyant les funcionalitats per compartir arxius i gestionar la paperera de reciclatge, vaig cometre un error en el disseny. Totes les peticions relacionades amb aquestes funcionalitats passaven pel Gateway i anaven directament al servei de FileManagement.

En aquesta primera versió, les entitats de FileManagement tenien camps booleans i dates per controlar si un fitxer estava compartit o a la paperera, la data en què es va compartir o eliminar, i fins i tot quan s'havia de purgar definitivament. Tot i que aquest enfocament funcionava, no seguia el principi de separació de responsabilitats que és clau en una arquitectura de microserveis. En realitat, FileManagement estava fent de servei central, mentre que FileSharing i TrashService només feien tasques secundàries, en lloc de ser serveis independents.

Quan em vaig adonar d'aquest error, vaig haver de parar el desenvolupament de noves funcionalitats per fer una refactorització gran. Això va implicar redissenyar com es movien les dades per assegurar que cada servei fos responsable només del seu àmbit. Ara, les peticions per compartir un fitxer van directament a FileSharing, que gestiona els seus propis registres, i les operacions de la paperera són només cosa de TrashService. Aquest canvi va requerir modificar rutes al Gateway, reescriure controladors, moure la lògica de negoci entre serveis i ajustar les crides internes. Tot i que aquesta refactorització va ser una inversió de temps considerable per corregir l'error inicial, era absolutament necessària per alinear la implementació amb els principis arquitectònics del projecte, assegurant una solució final més robusta, fàcil de mantenir i realment distribuïda.

Una de les avantatges d'aquest disseny és que, malgrat l'error en la implementació al servidor, com totes les crides passen pel Gateway abans d'arribar al servei corresponent, la implementació del client web (en aquell moment l'únic implementat) no va necessitar pràcticament cap modificació. Això es deu al fet que la interfície externa de l'API es va mantenir pràcticament idèntica, tant pel que fa al cos JSON que s'enviava al servidor com a l'endpoint que es consumia. Aquesta consistència va permetre que el client web continués funcionant sense interrupcions, malgrat els canvis significatius en l'arquitectura del servidor.

9.4 Implementació del client web

9.4.1 Introducció

El client web és una *Single Page Application* (SPA) desenvolupada amb les tecnologies més modernes de l'ecosistema de JavaScript, com són React, Vite i TypeScript. La interfície d'usuari s'ha construït utilitzant el framework CSS **Tailwind CSS** per a un disseny àgil i personalitzable, juntament amb un conjunt de components reutilitzables i accessibles de **Radix UI**, que han estat encapsulats i estilitzats al directori ui-new/src/components. La gestió de l'estat global de l'aplicació es realitza mitjançant **Zustand**, una solució lleugera i potent, mentre que les operacions asíncrones, el cacheig de dades del servidor i les actualitzacions optimistes es gestionen amb la llibreria **TanStack React Query**, garantint una experiència d'usuari fluida i reactiva.

L'arquitectura del codi font segueix un enfocament modular i organitzat, separant clarament les diferents responsabilitats: la lògica de negoci s'encapsula en *hooks* reutilitzables, l'estat global en *stores* de Zustand, les crides a l'API del backend s'abstrauen en serveis, i els components d'interfície són purs i centrats exclusivament en la representació visual.

Estructura de directoris del client web

El projecte del client web, ubicat a ui-new/, presenta una estructura de directoris clara i modular que separa les diferents responsabilitats de l'aplicació. A continuació, es detallen els directoris principals i el seu contingut:

- src/components/: Aquest directori conté tots els components reutilitzables de React que conformen la interfície d'usuari. Alguns dels components més rellevants són:
  - **Components de Tremor**: S'han integrat diversos components de la llibreria de components de React de codi obert Tremor [**tremor**], adaptats per a aquest projecte. Aquests inclouen Accordion, Card, Dialog, Drawer, Input, Popover, Toast, i Tooltip. Aquests components proporcionen una base sòlida i accessible per a la construcció d'interfícies d'usuari complexes.
  - Button: Un botó personalitzat amb diferents variants visuals (primari, secundari, fantasma, etc.).
  - CreateNewFolderDialog: Un diàleg modal per a la creació de noves carpetes.

  - File: Component que representa un fitxer o carpeta a la interfície, amb la seva icona i nom. Inclou subcomponents com RenameDialog i ShareDialog per a la gestió d'aquestes accions.
  - Tabs i TabsSubHeader: Components per a la creació de navegació per pestanyes.
  - Toaster: Component que gestiona la cua i visualització de notificacions (toasts).
- src/pages/: Conté els components que representen les pàgines completes de l'aplicació, com ara:
  - AdminDashboard: El panell d'administració d'usuaris.
  - FileManager: La pàgina principal de gestió de fitxers.
  - Login i SignUp: Les pàgines d'inici de sessió i registre.
- src/hooks/: Aquest directori és fonamental per a la lògica de negoci, ja que conté els *hooks* personalitzats de React que encapsulen la majoria de la complexitat:
  - useAuth: Gestiona l'estat d'autenticació de l'usuari, incloent l'inici de sessió, el tancament de sessió i el registre.
  - useFileOperations: Conté tota la lògica per a les operacions de fitxers (crear, moure, copiar, eliminar, etc.), utilitzant TanStack Query per a les actualitzacions optimistes.
  - useFileSelection: Gestiona la selecció de fitxers i el porta-retalls (copiar/-tallar).
  - useShareManager: Encapsula la lògica per a compartir fitxers i gestionar els permisos.
  - useValidation: Proporciona funcions per a la validació de formularis.
- src/layouts/: Conté els components d'estructura de la pàgina, com AppLayout, que defineix la disposició general amb la barra lateral i la capçalera.
- src/lib/: Un directori per a utilitats i configuracions generals:
  - api.ts: Configura Axios i defineix els serveis per a interactuar amb l'API del backend (authService, fileService, etc.).
  - utils.ts: Funcions d'utilitat genèriques.
  - websocket.ts: Gestiona la connexió WebSocket per a les actualitzacions en temps real.
- src/store/: Conté els *stores* de Zustand per a la gestió de l'estat global, com ara fileStore (estat dels fitxers) i fileSelectionStore (estat de la selecció de fitxers).

- src/types/: Defineix els tipus de TypeScript utilitzats a tota l'aplicació, garantint la consistència i la seguretat del tipat.

9.4.2 Disseny Visual i Components de la Interfície

A continuació, es presenta una descripció visual detallada de la implementació final del client web. Aquesta secció té un doble objectiu: d'una banda, il·lustrar com els esbossos conceptuals presentats al **Capítol 8 (secció 8.5.1)** s'han materialitzat en una interfície funcional i interactiva; de l'altra, justificar les decisions de disseny preses durant la implementació que divergeixen o milloren les propostes inicials, sempre amb l'objectiu de perfeccionar l'experiència d'usuari.

Autenticació

El flux d'autenticació és el primer punt de contacte de l'usuari amb la plataforma. El disseny s'ha centrat en la claredat i la simplicitat, evitant distraccions per facilitar un accés ràpid i segur.

FIGURA 9.9: Pantalla d'inici de sessió.

FIGURA 9.10: Pantalla de registre.

Com es pot observar, la implementació final de les pantalles d'inici de sessió (**Figura 9.46**) i registre (**Figura 9.47**) és una traducció fidel dels esbossos conceptuals presentats a les **Figures 8.21 i 8.22** del Capítol 8. El disseny final és minimalista i funcional, utilitzant un component de Card per enmarcar els formularis i millorar-ne la llegibilitat. Una millora funcional no detallada en la fase de disseny inicial és el sistema de validació de dades. A més de la validació de la lògica de negoci que es produeix en el moment de l'enviament del formulari, s'ha incorporat una validació prèvia al client. Com es pot apreciar a la **Figura 9.48**. Això millora l'experiència d'usuari i redueix la càrrega del servidor.

FIGURA 9.11: Exemple de validació nativa del navegador en el formulari de registre.

Escriptori Principal i Navegació

L'escriptori principal és el nucli de l'aplicació, on es centralitzen totes les funcionalitats de gestió d'arxius. A diferència del boceto general de la **Figura 8.23**, la implementació final opta per un *layout* més net i dinàmic, on els components s'organitzen de manera fluida per adaptar-se a diferents mides de pantalla.

FIGURA 9.12: Detall de l'arbre de navegació.

L'arbre de navegació lateral (Figura 9.49) és una implementació fidel i funcional del concepte descrit a la **Figura 8.25**. Permet a l'usuari desplaçar-se de manera intuïtiva entre els seus arxius, la secció de Çompartits amb miï la "Paperera", distingint visualment la carpeta seleccionada.

FIGURA 9.13: Diàleg del sistema per a la selecció d'arxius.

FIGURA 9.14: Notificació emergent o toast.

La pujada de fitxers es realitza mitjançant el diàleg natiu del sistema operatiu (Figura 9.50, esquerra), oferint a l'usuari una experiència familiar i intuïtiva per seleccionar arxius i carpetes. Com a millora clau per al *feedback* a l'usuari, s'han incorporat notificacions emergents o *toasts* (Figura 9.51, dreta). Aquestes s'utilitzen a tota l'aplicació per comunicar l'èxit o el fracàs de les operacions, proporcionant una resposta visual immediata.

FIGURA 9.15: Modal de configuració del perfil d'usuari.

FIGURA 9.16: Vista per a un usuari estàndard.

FIGURA 9.17: Modal de configuració del perfil, amb vistes diferenciades per rol.

Pel que fa a les accions globals, la **Figura 9.54** mostra el modal de configuració del perfil, que correspon al disseny de la **Figura 8.28**. La implementació, però, és més avançada i dinàmica. El contingut s'adapta al rol de l'usuari per motius de seguretat. Com es pot veure, la vista per a un usuari estàndard (dreta) inclou un botó per eliminar el seu propi compte. En canvi, a la vista per a un usuari amb rol SUPER_ADMIN (esquerra), aquest botó no hi és. Aquesta és una restricció deliberada per impedir que l'administrador principal es pugui eliminar a si mateix, la qual cosa deixaria el sistema sense un compte amb control total.

FIGURA 9.18: Diàleg modal per a la creació d'una nova carpeta.

La creació de noves carpetes, una de les "accions sobre la carpeta actual"esmentades a la **Figura 8.26**, es gestiona mitjançant un diàleg modal simple, com es mostra a la **Figura 9.55**. Aquesta elecció de disseny minimitza la interrupció del flux de treball

de l'usuari, ja que el diàleg apareix superposat al gestor de fitxers i requereix una acció directa (crear o cancel·lar) per continuar.

FIGURA 9.19: Ordenar.

FIGURA 9.20: Panell admin.

FIGURA 9.21: Configuració.

FIGURA 9.22: Tancar sessió.

FIGURA 9.23: Controls globals de l'aplicació.

Els controls per a les "opcions globals"de la **Figura 8.27** s'han implementat a la cantonada superior dreta de la interfície. Com es pot apreciar a la **Figura 9.60**, el disseny final agrupa de manera lògica les opcions per ordenar el contingut, accedir al panell d'administració, a la configuració de l'usuari i per tancar la sessió, utilitzant icones minimalistes per mantenir una estètica neta i organitzada.

Interacció amb Elements

La interacció amb arxius i carpetes s'ha dissenyat per ser rica i contextual. El menú d'accions, que s'activa tant amb el clic dret com a través d'un botó dedicat, és una implementació directa del concepte de la **Figura 8.29** i ofereix totes les operacions rellevants per a l'element seleccionat.

FIGURA 9.24: Clic dret.

FIGURA 9.25: Botó d'opcions.

FIGURA 9.26: Opcions a l'arrel.

FIGURA 9.27: Menú contextual dinàmic.

La **Figura 9.64** mostra com el menú contextual apareix tant en fer clic dret sobre un element o les opcions que es mostren tant al fer click dert pero a cap element, com en prémer el botó d'opcions. Aquest menú és dinàmic i adapta les seves opcions al context, com per exemple a la seccio dels fitxer del propietari (*root*), on permet crear nous elements.

FIGURA 9.28: Selecció múltiple.

FIGURA 9.29: Feedback visual de "Tallar".

FIGURA 9.30: Panell de detalls.

La implementació final inclou millores significatives d'usabilitat. S'ha implementat la selecció múltiple (Figura 9.65) per realitzar operacions en lot, i es proporciona un *feedback* visual clar per a accions com "Tallar", on l'element es mostra semitransparent (Figura 9.66). Per a la visualització de detalls, es va decidir substituir el modal proposat a la **Figura 8.30** per un panell lateral o *Drawer* (Figura 9.67). Aquesta decisió millora l'experiència en permetre a l'usuari consultar la informació sense perdre el context de la vista d'arxius. A més, com a resultat d'una millora proposada per un *beta tester*, aquest panell mostra també la llista d'usuaris amb qui s'ha compartit un fitxer, oferint l'opció de modificar el seu nivell d'accés o deixar de compartir-lo. Això permet gestionar la compartició d'un element de forma més àgil, sense haver de navegar a la secció de fitxers compartits.

FIGURA 9.31: Diàleg per canviar el nom d'un element.

El diàleg per canviar el nom d'un element (**Figura 9.68**) és una implementació directa i funcional del concepte mostrat a la **Figura 8.31**, proporcionant una manera ràpida i senzilla de realitzar aquesta operació comuna.

FIGURA 9.32: Vista inicial.

FIGURA 9.33: Vista amb carpeta desplegada.

FIGURA 9.34: Modal per moure elements.

El diàleg per moure elements (**Figura 9.71**) és una evolució significativa del seu esbós inicial (**Figura 8.32**). La implementació final és molt més avançada, ja que presenta un arbre de carpetes completament navegable. Com es pot apreciar, l'usuari pot desplegar les subcarpetes (dreta) per seleccionar la ubicació de destí de manera precisa, una millora substancial d'usabilitat respecte a un simple camp de text. Aquest dialeg es utilitzable tant per els fitxers o carpetes dintre de la seccio *root* com per la seccio de *Compartits amb mi* on es poden moure els elements dintre de l'arbre de carpetes compartides amb l'usuari (desde la primera carpeta a l'arrel dels compartits fins l'ultim fill), aquesta limitacio es va posar per motius de seguretar per evitar que

es moguin fitxers entre carpetes d'usuaris diferents sense permis explicit, de forma nativa a l'aplicacio.

FIGURA 9.35: Vista principal.

FIGURA 9.36: Cerca d'usuaris.

FIGURA 9.37: Modal de compartició d'arxius.

Finalment, el diàleg per compartir arxius (**Figura 9.74**) és substancialment més complet que el seu concepte a la **Figura 8.33**. La versió implementada permet buscar usuaris pel seu nom (dreta), assignar permisos específics de lectura o escriptura, i mostra una llista clara dels usuaris que ja tenen accés, amb opcions per modificar o revocar els seus permisos directament des de la mateixa interfície, oferint una gestió de compartició molt més potent, ja que permet compartir llistes d'elements d'una sola accio, evitant que l'usuari hagi de compartir un per un.

Seccions Específiques

Les seccions de la Paperera, els Elements Compartits i el Panell d'Administració disposen de vistes i opcions contextuals pròpies.

FIGURA 9.38: Menú contextual a la secció Çompartit amb mi".

Elements Compartits El menú contextual per a un element a la secció Çompartit amb mi"(**Figura 9.75**) és un exemple de la naturalesa dinàmica de la interfície. Com

es va planejar a la **Figura 8.34**, les opcions disponibles ("Descargar", "Dejar de seguir", etc.) canvien en funció dels permisos (lectura o escriptura) que l'usuari tingui sobre l'element, oferint només les accions permeses.

Paperera La vista de la paperera (**Figura 9.76**) implementa fidelment les opcions conceptualitzades a la **Figura 8.36**, oferint les accions de "Restaurarï "Eliminar definitivament"per a cada element.

FIGURA 9.39: Opcions contextuals de la paperera.

Panell d'Administració El panell d'administració (**Figura 9.77**) representa una de les millores més significatives respecte al seu disseny inicial (**Figura 8.37**). En lloc d'una simple llista, s'ha implementat una taula de dades completa. A més, la interfície és dinàmica i conscient dels permisos.

FIGURA 9.40: Panell d'administració d'usuaris.

FIGURA 9.41: Diàleg per a l'edició d'un usuari des del panell d'administració (vista SUPER_ADMIN).

L'edició d'usuaris és una de les funcionalitats clau del panell. En seleccionar l'opció d'editar, s'obre un diàleg modal que permet modificar les dades de l'usuari, tal com il·lustra la **Figura 9.78**. La implementació estableix una clara jerarquia de permisos: mentre que un administrador estàndard (ADMIN) pot actualitzar informació bàsica, només un superadministrador (SUPER_ADMIN) té l'autoritat per canviar rols o restablir contrasenyes. Aquesta segregació de privilegis, que reserva les accions més crítiques al rol més alt, és una mesura de seguretat fonamental. La decisió d'atorgar aquests poders al SUPER_ADMIN es fonamenta en la seva funció com a responsable últim del sistema. Tot i que aquesta centralització de permisos representa un risc de seguretat calculat, es considera una capacitat indispensable per a la gestió d'incidències crítiques, com ara la recuperació de l'accés per a un usuari que hagi perdut la contrasenya. Aquesta responsabilitat és coherent amb el seu rol com a responsable de la integritat de les dades i del compliment de la normativa vigent.

FIGURA 9.42: Botó d'eliminació actiu.

FIGURA 9.43: Botó d'eliminació desactivat.

FIGURA 9.44: Comportament dinàmic del botó d'eliminació.

Com es pot veure, el botó "Delete"és interactiu quan es tenen permisos sobre l'usuari (esquerra), però es mostra desactivat i no és funcional quan s'intenta eliminar un compte protegit, com el del propi SUPER_ADMIN (dreta). Aquesta restricció visual i funcional és una mesura de seguretat clau per garantir la integritat del sistema.

FIGURA 9.45: Diàleg de confirmació per a l'eliminació d'un usuari.

Finalment, abans de qualsevol eliminació, es mostra un diàleg de confirmació per prevenir accions accidentals (**Figura 9.82**), una mesura de seguretat afegida durant la implementació que no estava als esbossos inicials.

9.4.3 Implementació per Casos d'Ús

A continuació, es detalla com l'arquitectura del client web i els seus components es combinen per donar resposta als principals casos d'ús del sistema. Cada secció descriu el flux d'implementació, els fitxers clau implicats i fragments de codi rellevants que il·lustren les decisions tècniques més importants.

Gestió d'Usuaris

UC-01: Registre d'Usuari & UC-02: Inici de Sessió La funcionalitat d'autenticació constitueix el punt d'entrada a l'aplicació. Per a un usuari no autenticat, el component principal App.tsx actua com a gestor de l'enrutament, presentant un interruptor que permet alternar entre les vistes de login (/login) i registre (/sign-up). Aquestes vistes, implementades a ui-new/src/pages/Login/index.tsx i ui-new/src/pages/SignUp/index.tsx respectivament, es construeixen amb components reutilitzables com Card per a l'estructura, i Input i Button per als formularis.

Quan l'usuari interactua amb el formulari, la funció submit del component corresponent orquestra el procés. Abans d'enviar cap dada, es realitza una validació completa dels camps mitjançant la lògica encapsulada al hook ui-new/src/hooks/useValidation.ts. Només si la validació és exitosa, s'invoca la lògica de negoci principal.

El cervell de l'autenticació resideix al hook ui-new/src/hooks/userAuth.tsx. Les seves funcions login i register són les responsables de gestionar tot el flux. Aquestes funcions no interactuen directament amb la xarxa, sinó que deleguen la comunicació al servei authService, definit a ui-new/src/lib/api.ts, que s'encarrega de construir i enviar la petició HTTP al backend.

Després d'una resposta exitosa del servidor, el hook userAuth executa una seqüència d'accions crucials per actualitzar l'estat del client: desa els tokens d'accés i de refresc al localStorage, obté les dades completes del perfil de l'usuari, actualitza l'estat global de l'aplicació i, finalment, estableix la connexió WebSocket per a la sincronització en temps real.

```
const login = useCallback(async (username: string, password: string):
    Promise¡boolean¿ =¿ -
  try -
    if(!isLogin) -
      const response = await authService.login(username, password);

      localStorage.setItem("accessToken", response.accessToken);
      localStorage.setItem("refreshToken", response.refreshToken);

      const userData = await userService.getCurrentUser();
      localStorage.setItem("username", userData.username);

      setUser(userData);
      setIsLogin(true);
      websocketService.connect();
    "
    return true;
  " catch (error) -
    // ... maneig d'errors
    return false;
  "
", [notifications, isLogin]);
```

LISTING 9.36: Fragment de la funció 'login' al hook 'userAuth.tsx'

Un cop l'usuari està autenticat, per mantenir la sessió activa de manera eficient, s'utilitza un mecanisme arquitectònic clau: un interceptor d'Axios, configurat a ui-new/src/lib/api.ts. Aquest interceptor s'activa automàticament quan una petició a l'API falla amb un codi d'estat 401 (No autoritzat). En aquest moment, intenta renovar el token d'accés utilitzant el token de refresc. Si la renovació té èxit, la petició original es reintenta amb el nou token, un procés completament imperceptible per a l'usuari.

```
api.interceptors.response.use(
    (response) =¿ response,
    async (error) =¿ -
        const originalRequest = error.config;
        const authRoutes = ['/users/auth/login', '/users/auth/keep-alive
    ', '/users/auth/register'];
        const refreshToken = localStorage.getItem('refreshToken');

        if (error.response?.status === 401 && !originalRequest.˙retry &&
     !authRoutes.includes(originalRequest.url) && refreshToken) -
            originalRequest.˙retry = true;

            try -
                const response = await api.post('/users/auth/keep-alive'
    , null, -
                    headers: -
                        'X-Refresh-Token': refreshToken
                    "
                ");

                const newAccessToken = response.headers['authorization'
    ]?.split(' ')[1];
                const newRefreshToken = response.headers['x-refresh-
    token'];

                if (newAccessToken && newRefreshToken) -
                    localStorage.setItem('accessToken', newAccessToken);
                    localStorage.setItem('refreshToken', newRefreshToken
    );

```

```
                    originalRequest.headers.Authorization = `Bearer ${newAccessToken}`;
                    return api(originalRequest);
                }
            } catch (refreshError) {
                localStorage.removeItem('accessToken');
                localStorage.removeItem('refreshToken');
                localStorage.removeItem('username');
                window.location.href = '/login';
                return Promise.reject(refreshError);
            }
        }

        return Promise.reject(error);
    }
);
```

LISTING 9.37: Interceptor d'Axios per a la renovació automàtica de tokens a api.ts

He de destacar que l'estratègia actual d'emmagatzematge de tokens al localStorage va ser una solució temporal. Inicialment, vaig intentar implementar un sistema basat en cookies HttpOnly per al transport dels tokens. Aquesta tècnica no només és una pràctica de seguretat estàndard que protegeix contra atacs de Cross-Site Scripting (XSS) en fer que els tokens siguin inaccessibles des del JavaScript del client [**bitsrc˙tokens˙vs˙cookies**], sinó que també s'alinea amb les exigències de la normativa de protecció de dades com el RGPD. Segons aquesta regulació, els identificadors en línia com els tokens es consideren dades personals i requereixen mesures tècniques per garantir-ne la seguretat [**gdpr˙cookies**].

No obstant això, vaig trobar dificultats tècniques en la comunicació amb el servidor. Per tal de no bloquejar el desenvolupament, vaig optar per l'emmagatzematge local com a solució intermèdia, amb la intenció de refactoritzar-la en un futur. La implementació d'un flux d'autenticació robust basat en cookies no només és una millor pràctica de seguretat altament recomanada [**auth0˙token˙best˙practices**], sinó també un pas necessari per a un compliment més estricte de la normativa. A causa de les limitacions de temps, vaig haver de deixar aquesta tasca registrada com una millora futura, tal com es detalla al Capítol 12.

UC-03: Tancament de Sessió El procés de tancament de sessió és una mesura de seguretat fonamental, dissenyada per ser directa i robusta. Assegura que la sessió de l'usuari finalitzi de forma definitiva mitjançant la invalidació dels tokens d'accés al client, prevenint així l'accés no autoritzat al compte si el dispositiu queda desatès i protegint contra el segrest de sessió. El flux comença a la interfície d'usuari, concretament quan l'usuari fa clic sobre el botó de logout situat al component ui-new/src/layouts/AppLayout/Header/index.tsx. Aquesta integració directa, on l'esdeveniment onClick invoca directament la funció logoutEndpoint, simplifica el component visual, que només s'ha de preocupar de renderitzar el botó i connectar-lo a l'acció corresponent.

La lògica central resideix al hook ui-new/src/hooks/userAuth.tsx, que proporciona la funció logoutEndpoint. Aquesta funció és la responsable de garantir una sortida neta i segura. Primer intenta notificar al backend mitjançant authService.logout. Independentment del resultat d'aquesta crida, ja que s'asumeix que el servidor rep la petició i fa les accions necessaries per el logout, el bloc finally s'encarrega d'eliminar

tota la informació de sessió del localStorage. Finalment, reinicia l'estat global de l'aplicació, desconnecta la sessió WebSocket activa i redirigeix forçosament l'usuari a la pàgina d'inici de sessió.

```
import - useAuth " from "../../../hooks/userAuth"

const Header = (-setAdminOpen": -setAdminOpen: (open: boolean) =¿ void")
    =¿ -
    const - logoutEndpoint, user " = useAuth();

    return (
        ¡div id="39" className="flex justify-end gap-2"¿
            ¡Tooltip side="left" content="Logout" triggerAsChild=-true"¿
                ¡Button variant="primary" className="p-2" onClick=-
    logoutEndpoint"¿
                    ¡RiLogoutBoxRLine className="h-4 w-4" /¿
                ¡/Button¿
            ¡/Tooltip¿
            -/* ... */"
        ¡/div¿
    )
"
```

LISTING 9.38: Fragment del component 'Header' amb el botó de logout

```
const logoutEndpoint = useCallback(async () =¿ -
  try -
    await authService.logout();
  " catch -
    // S'ignora l'error per assegurar que el logout en el client sempre
    ocorri
  " finally -
    localStorage.removeItem("accessToken");
    localStorage.removeItem("refreshToken");
    localStorage.removeItem("userId");
    localStorage.removeItem("username");

    setUser(null);
    setIsLogin(false);
    websocketService.disconnect();
    window.location.href = "/login";
  "
", []);
```

LISTING 9.39: Funció 'logoutEndpoint' al hook 'userAuth.tsx'

UC-04: Modificar Perfil & UC-05: Eliminar Compte La gestió del perfil d'usuari, que va des de la modificació de dades personals fins a l'eliminació permanent del compte, es consolida en una única narrativa de flux de dades per garantir una experiència d'usuari coherent i eficient. El procés s'inicia quan l'usuari accedeix a la configuració del seu perfil, una acció que renderitza el component intel·ligent ui-new/src/components/UserSettings/index.tsx. Aquest component actua com el nucli central de la funcionalitat, encapsulant tota la lògica necessària. En ser muntat, invoca immediatament el mètode userService.getCurrentUser(), definit a ui-new/src/lib/api.ts, per carregar les dades actuals de l'usuari i popular el formulari.

Un aspecte clau d'aquest component és la seva estratègia de validació en temps real, dissenyada per oferir feedback instantani sense sobrecarregar el sistema. Mentre que la validació de formats bàsics per a la majoria de camps s'implementa a

través del hook ui-new/src/hooks/useValidation.ts, la comprovació de la disponibilitat de camps únics com el nom d'usuari i el correu electrònic requereix una solució més sofisticada. Per evitar una gran quantitat de peticions a l'API amb cada pulsació de tecla, s'utilitza el hook ui-new/src/hooks/useDebounce.ts. Aquest retrasa l'execució de les funcions de validació (userService.checkUsernameExists i userService.checkEmailExists) fins que l'usuari ha deixat d'escriure durant un breu període. Aquesta tècnica millora dràsticament el rendiment i l'experiència d'usuari, evitant validacions innecessàries i oferint un indicador d'estat només quan és rellevant.

```
// ...
const debouncedUsername = useDebounce(userData.username, 500);
const debouncedEmail = useDebounce(userData.email, 500);

useEffect(() =¿ -
    const checkUsernameAvailability = async () =¿ -
        // No validar si el valor no ha canviat del original
        if (!initialUserData —— debouncedUsername === initialUserData.
    username) -
            // ...
            return;
        "
        // ...
        try -
            const isTaken = await userService.checkUsernameExists(
    debouncedUsername);
            // Actualitzar estat de la validació
        " catch -
            // Gestionar error de la comprovació
        "
    ";
    if (debouncedUsername && isOpen) checkUsernameAvailability();
", [debouncedUsername, initialUserData, errors.username, isOpen]);
// ... (codi similar per a la validació de l'email)
```

LISTING 9.40: Ús de 'useDebounce' per a la validació a 'UserSettings/index.tsx'

Nota: La implementació del hook useDebounce *es va adaptar de l'article Ïmplementing a Debounce Hook in React"[***debounce˙hook˙react****]. Aquesta millora va ser suggerida durant les sessions de proves beta per un tester amb experiència professional en React per tal d'optimitzar la interacció de l'usuari.*

Un cop les dades són validades, el component gestiona les dues accions principals. La funció handleUpdateProfile orquestra l'actualització del perfil, cridant userService.updateUserProfile i, si l'usuari ha introduït una nova contrasenya, userService.changePassword. Per altra banda, la funció handleDeleteAccount gestiona l'eliminació del compte. Per seguretat, primer mostra un diàleg de confirmació que requereix que l'usuari escrigui el seu nom d'usuari. Un cop confirmat, invoca userService.deleteAccount. Un detall crucial és que, després d'una eliminació exitosa, crida la funció logoutEndpoint del hook useAuth per netejar completament la sessió del client i redirigir-lo a la pàgina d'inici de sessió.

```
const handleUpdateProfile = async () =¿ -
    // ... (validació final)
    await userService.updateUserProfile(userData);
    if (newPassword && oldPassword) -
        await userService.changePassword(oldPassword, newPassword);
    "
```

```
    notifications.success("Profile updated");
    // ...
";

const handleDeleteAccount = async () =¿ -
    if (deleteConfirmation !== userData.username) -
        // ...
        return;
    "
    try -
        await userService.deleteAccount();
        notifications.success("Account deleted");
        logoutEndpoint(); // Funció del hook useAuth
    " catch (error) -
        notifications.error("Delete failed");
    "
";
```

LISTING 9.41: Orquestració de les crides a l'API a 'UserSettings/index.tsx'

Finalment, per a cada operació, ja sigui d'èxit o de fracàs, s'utilitzen notificacions tipus "toast"per proporcionar un feedback clar i immediat a l'usuari, tancant així el cicle d'interacció. El servei userService a api.ts abstrau les crides a l'API, proporcionant mètodes que mapegen directament als endpoints del backend per a una gestió de perfil clara i mantenible.

```
export const userService = -
    // ...
    updateUserProfile: async (userData: - /* ... */ ") =¿ -
        const response = await api.put('/users/profile', userData);
        return response.data;
    ",

    changePassword: async (oldPassword: string, newPassword: string) =¿
    -
        await api.put('/users/password', - oldPassword, newPassword ");
    ",

    deleteAccount: async () =¿ -
        await api.delete('/users');
    "
";
```

LISTING 9.42: Mètodes del 'userService' a 'lib/api.ts'

Gestió d'Administració

UC-06, UC-07, UC-15, UC-16: Panell d'Administració El panell d'administració representa una funcionalitat crítica per a la gestió d'usuaris del sistema, accessible exclusivament per a usuaris amb privilegis elevats.

L'accés al panell s'inicia a través d'un botó situat a la capçalera de l'aplicació, implementat a ui-new/src/layouts/AppLayout/Header/index.tsx. Aquest component utilitza el hook useAuth per verificar els permisos de l'usuari i renderitza el botó d'accés només si l'usuari té el rol ADMIN o SUPER_ADMIN:

```
-(user?.role === 'ADMIN' -- user?.role === 'SUPER'ADMIN') && (
    ¡Tooltip side="left" content="Admin Panel" triggerAsChild=-true"¿
```

```
3          ¡Button variant="primary" className="p-2" onClick=-() =¿
    setAdminOpen(true)"¿
4              ¡RiAdminLine className="h-4 w-4" /¿
5          ¡/Button¿
6      ¡/Tooltip¿
7  )"
```

LISTING 9.43: Renderitzat condicional del botó d'administració

Quan s'activa el botó, el component AppLayout actua com a orquestrador, gestionant la transició entre la vista principal de l'aplicació i el panell d'administració. Aquest canvi es realitza mitjançant un estat local adminOpen que renderitza condicionalment el component AdminDashboard en lloc de la interfície principal:

```
1      if(adminOpen) -
2      return (
3        ¡div className="flex h-screen bg-gray-100 relative"¿
4          ¡button
5            className="absolute top-6 right-8 z-50 px-4 py-2 bg-indigo-600
     text-white rounded-lg shadow hover:bg-indigo-700 transition-colors
    font-semibold"
6            onClick=-() =¿ setAdminOpen(false)"
7          ¿
8            Volver
9          ¡/button¿
10         ¡main className="flex-1 overflow-auto"¿
11           ¡AdminDashboard  /¿
12         ¡/main¿
13       ¡/div¿
14     );
15   "
```

LISTING 9.44: Renderitzat condicional del panell d'administració a 'AppLayout'

El component AdminDashboard serveix com a punt central per a la gestió d'usuaris. En el seu muntatge, utilitza l'adminService per carregar la llista d'usuaris del sistema. La visualització d'aquesta informació es realitza mitjançant una taula interactiva que permet als administradors realitzar operacions de modificació i eliminació d'usuaris.

La gestió d'usuaris es realitza a través de dos diàlegs especialitzats: EditUserDialog i DeleteUserDialog. El primer és particularment complex, ja que implementa validacions en temps real i restriccions basades en rols. Per exemple, la capacitat de modificar el rol d'un usuari està estrictament controlada:

```
1      -isSuperAdmin && editForm.role !== 'SUPER_ADMIN' &&(
2          ¡¿
3        ¡div className="space-y-2"¿
4          ¡label className="text-sm font-medium text-gray-300"¿Role¡/label
    ¿
5          ¡select
6            className="w-full rounded-md border border-input bg-background
     px-3 py-2"
7            value=-editForm.role"
8            onChange=-e =¿ setEditForm(prev =¿ (- ...prev, role: e.target.
    value as 'USER' — 'ADMIN' — 'SUPER_ADMIN' "))"
9            disabled=-!currentUser —— currentUser && currentUser.role !==
    'SUPER_ADMIN' —— (currentUser.role === 'ADMIN' && user.role === '
    ADMIN')"
10         ¿
```

```
            ¡option value="USER"¿User¡/option¿
            ¡option value="ADMIN"¿Administrator¡/option¿
          ¡/select¿
        ¡/div¿
          ¡/¿
        )"
```

LISTING 9.45: Control d'accés per a la modificació de rols

Aquest fragment de codi il·lustra el control d'accés per a la modificació de rols. La lògica implementada assegura que només un usuari amb el rol de SUPER_ADMIN pugui canviar el rol d'un altre usuari. A més, s'estableix una restricció clau: el SUPER_ADMIN no pot modificar el seu propi rol. Aquesta decisió de disseny, al igual que impedir que es pugui eliminar el seu propi compte, té com a objectiu garantir que sempre hi hagi un, i només un, superadministrador com a màxim responsable de l'aplicació i les seves dades. La condició isSuperAdmin && editForm.role !== 'SUPER_ADMIN' del codi és la que materialitza aquesta regla, mostrant el selector de rol únicament quan es compleixen aquestes condicions.

L'execució de les operacions d'administració es gestiona a través de funcions específiques al component AdminDashboard. Aquestes funcions no només realitzen les operacions sol·licitades sinó que també gestionen la retroalimentació a l'usuari i l'actualització de l'estat de l'aplicació:

```
    const handleSaveEdit = async (updatedUser: Partial¡AdminUser¿) =¿ -
        if (!selectedUser) return;
        try -
            await adminService.updateUser(selectedUser.username,
    updatedUser);

            notifications.success("User updated");
            setLoaded(false);
            await loadUsers();
            setIsEditDialogOpen(false);
        " catch -
            notifications.error("Update failed");
        "
    ";

    const handleConfirmDelete = async () =¿ -
        if (!selectedUser) return;
        try -
            await adminService.deleteUser(selectedUser.username);
            if(selectedUser.username === user?.username) -
                logoutEndpoint();
            "

            notifications.success("User deleted");
            setLoaded(false);
            await loadUsers();
            setIsDeleteDialogOpen(false);
        " catch -
            notifications.error("Delete failed");
        "
    ";
```

LISTING 9.46: Gestió d'operacions administratives

Un aspecte notable de la implementació és el tractament especial del cas d'autoeliminació (**UC-07**). Quan un administrador elimina el seu propi compte, el sistema

detecta aquesta situació i executa automàticament el procés de tancament de sessió mitjançant la funció logoutEndpoint.

Tota la comunicació amb el servidor es realitza a través de l'adminService, definit a ui-new/src/lib/api.ts, que proporciona una capa d'abstracció per a totes les operacions administratives. Aquest servei gestiona les crides a l'API REST del backend, mantenint la coherència i la seguretat en totes les operacions d'administració.

Com es pot veure, el botó "Delete"és interactiu quan es tenen permisos sobre l'usuari (**Figura 9.79**), però es mostra desactivat i no és funcional quan s'intenta eliminar un compte protegit, com el del propi SUPER_ADMIN (**Figura 9.80**). Aquesta restricció visual i funcional és una mesura de seguretat clau per garantir la integritat del sistema.

Finalment, abans de qualsevol eliminació, es mostra un diàleg de confirmació per prevenir accions accidentals (**Figura 9.82**), una mesura de seguretat afegida durant la implementació que no estava als esbossos inicials.

9.4.4 Gestió d'Arxius

El gestor d'arxius, implementat principalment al component ui-new/src/pages/FileManager/index.tsx, constitueix el nucli funcional de l'aplicació web.

UC-08: Crear o Pujar Arxius i Carpetes Aquesta funcionalitat permet als usuaris afegir nou contingut al seu espai de treball, ja sigui creant carpetes o pujant arxius i estructures de carpetes completes des del seu dispositiu.

La interacció de l'usuari s'inicia en el component ui-new/src/components/AddButton/index.tsx, que presenta un botó que, en ser premut, desplega un Popover amb tres opcions: "New folder", "Upload file" i "Upload folder". Aquest component implementa una tècnica comuna per a la pujada d'arxius: conté dos elements ¡input type="file» ocults que s'activen per codi mitjançant referències (useRef) quan l'usuari selecciona les opcions corresponents. Per a la pujada de carpetes, l'atribut no estàndard webkitdirectory és clau, ja que instrueix el navegador perquè permeti la selecció de carpetes completes.

L'opció "New folder" obre el diàleg modal ui-new/src/components/CreateNewFolderDialog/index.tsx, mentre que les opcions de pujada activen directament els respectius inputs ocults. Aquests components de la UI actuen com a mers disparadors: tota la lògica de negoci complexa resideix en el hook ui-new/src/hooks/useFileOperations.ts, que centralitza les operacions d'arxius. Les accions de l'usuari en la UI invoquen les funcions createFolder, uploadFile i uploadFolder proporcionades per aquest hook.

```
1  const folderInputRef = useRef¡HTMLInputElement¿(null);
2
3      const handleFolderUploadLocal = async (event: React.ChangeEvent¡
       HTMLInputElement¿) =¿ -
4      const files = event.target.files;
5          if (!files —— files.length ==== 0) -
6              return;
7          "
8          try -
9          uploadFolder(- files: Array.from(files), parentId:
       currentDirectory.id!.toString() ");
10         " catch (error) -
```

```
11                console.error('[AddButton] Error uploading folder:', error);
12        "
13            event.target.value = '';
14    ";
15
16        // JSX del component
17    ¡input
18        type="file"
19        ref=-folderInputRef"
20            className="hidden"
21            webkitdirectory=""
22            directory=""
23            multiple=-false"
24            onChange=-handleFolderUploadLocal"
25        /¿
26        ¡Button
27            variant="ghost"
28            className="w-full justify-start gap-2"
29            onClick=-() =¿ -
30                folderInputRef.current?.click();
31            ""
32        ¿
33            ¡RiFolderUploadLine className="h-4 w-4" /¿
34            Upload folder
35        ¡/Button¿
```

LISTING 9.47: Activació de la pujada de carpetes a 'AddButton/index.tsx'

El hook useFileOperations.ts constitueix una peça clau de l'arquitectura del client, utilitzant el hook useMutation de TanStack Query per a totes les operacions d'escriptura. Això permet una gestió d'estat molt avançada, incloent-hi les **actualitzacions optimistes**, que milloren dràsticament la percepció de velocitat de l'usuari. El flux d'una actualització optimista segueix un patró consistent: l'onMutate s'executa abans de la crida a l'API, actualitzant la memòria cache local de TanStack Query de manera immediata i fent que la interfície reflecteixi el canvi a l'instant. Es guarda una còpia de l'estat previ per poder-lo revertir en cas d'error. Si la crida a l'API falla, l'onError reverteix l'actualització optimista restaurant l'estat previ. Si la crida té èxit, l'onSuccess invalida la memòria cache per assegurar que les dades locals se sincronitzin amb les dades definitives provinents del servidor.

```
1  const createFolderMutation = useMutation(-
2      mutationFn: async (- name, parentId ": - name: string, parentId:
       string ") =¿ -
3        return fileService.createFolder(name, parentId);
4      ",
5      onMutate: async (- name, parentId ") =¿ -
6        // 1. Cancelar queries per evitar que sobreescriguin l'
       actualitzacio optimista
7        await queryClient.cancelQueries(- queryKey: [QUERY'KEYS.
       CURRENT'DIRECTORY, parentId] ");
8        await queryClient.cancelQueries(- queryKey: [QUERY'KEYS.
       FOLDER'STRUCTURE] ");
9
10       // 2. Guardar l'estat previ
11       const previousDirectory = queryClient.getQueryData([QUERY'KEYS.
       CURRENT'DIRECTORY, parentId]) as FileItem;
12       const previousStructure = queryClient.getQueryData([QUERY'KEYS.
       FOLDER'STRUCTURE]);
13
14       // 3. Crear el nou objecte i actualitzar la UI de forma optimista
```

```
15        const newFolder: FileItem = -
16          id: `temp-$-Date.now()"`,
17          name: generateUniqueName(name, previousDirectory.subfolders?.map
    ((f: FileItem) =¿ f.name) —— [], name, true),
18          type: 'folder',
19          parent: parentId,
20          subfolders: []
21        ";
22
23        const directoryCopy = JSON.parse(JSON.stringify(previousDirectory)
    );
24        directoryCopy.subfolders = [...(directoryCopy.subfolders —— []),
    newFolder];
25        queryClient.setQueryData([QUERY'KEYS.CURRENT'DIRECTORY, parentId],
     directoryCopy);
26
27        return - previousDirectory, previousStructure ";
28      ",
29      onError: (err, variables, context) =¿ -
30        // Revertir en cas d'error
31        if (context?.previousDirectory) -
32          queryClient.setQueryData([QUERY'KEYS.CURRENT'DIRECTORY,
    variables.parentId], context.previousDirectory);
33        "
34        if (context?.previousStructure) -
35          queryClient.setQueryData([QUERY'KEYS.FOLDER'STRUCTURE], context.
    previousStructure);
36        "
37        notifications.error('Error creating folder');
38      ",
39      onSuccess: () =¿ -
40        notifications.success('Folder created');
41        refreshAll(); // Sincronitzar amb el servidor
42      "
43  ");
```

LISTING 9.48: Implementacio d'una mutacio optimista a 'useFileOperations.ts'

Les funcions de mutació dins del hook són les que finalment comuniquen amb el backend a través dels mètodes corresponents del fileService (definits a ui-new/src/lib/api.ts), que s'encarreguen de la comunicació HTTP. La pujada de carpetes presenta una complexitat addicional: la funcionalitat uploadFolderMutation ha de processar la llista de fitxers (FileList) que proporciona el navegador, reconstruir l'estructura de directoris en el client i enviar-la de forma recursiva al backend per recrear-la.

Aquesta implementació recursiva, si bé és robusta per a gestionar estructures de carpetes complexes, introdueix un important coll d'ampolla de rendiment. Cada fitxer dins de la carpeta es tradueix en una petició HTTP independent, la qual cosa pot generar una càrrega excessiva al backend en pujar directoris amb un gran volum d'elements. L'API actual no suporta operacions de pujada per lots (*batch uploads*), una limitació tècnica que impedeix una solució més eficient. L'optimització d'aquest flux s'ha identificat com un treball futur, tal com s'exposa al Capítol 12.

UC-09A/B/C: Operacions amb Arxius (Renomenar, Moure, Copiar) Aquest conjunt de funcionalitats cobreix les manipulacions més habituals sobre els elements del gestor d'arxius, com canviar el nom, moure'ls de lloc o duplicar-los.

El cervell que orquestra aquestes operacions resideix al ui-new/src/store/fileSelectionStore.ts, un store de Zustand que actua com a nucli centralitzat de l'estat. Aquest store gestiona un registre dels fitxers que l'usuari té seleccionats (selectedFiles) i, a més, actua com un porta-retalls virtual, emmagatzemant els fitxers copiats o tallats (clipboard) juntament amb l'estat de l'operació (isCut).

Per alimentar aquesta llista de fitxers seleccionats, la interfície ofereix múltiples mètodes d'interacció. El hook ui-new/src/hooks/useSelecto.ts, que integra la llibreria Selecto.js, permet a l'usuari dibuixar un quadre de selecció amb el ratolí per seleccionar diversos elements de forma intuïtiva. A més, s'han implementat dreceres de teclat per a una navegació eficient: el hook ui-new/src/hooks/useFileSelection.ts exposa la funció selectArrowKeys, que permet expandir o moure la selecció utilitzant les tecles de fletxa, mentre que ui-new/src/hooks/useFileShortcuts.ts gestiona la drecera Ctrl+A per seleccionar tots els elements del directori. Tots aquests mecanismes convergeixen en la modificació de l'estat selectedFiles dins del fileSelectionStore. Aquesta arquitectura desacobla la lògica de selecció de la d'acció, permetent que la resta de funcionalitats operin sobre la selecció actual sense necessitat de conèixer com s'ha generat.

Per determinar quins elements són seleccionables, el hook useSelecto.ts configura la llibreria Selecto.js perquè apunti a qualsevol element que tingui la classe CSS .file-item. Cada component File assigna aquesta classe al seu element principal, un component Card, fent-lo així un objectiu vàlid per a la selecció.

Aquesta connexió permet que l'estil de cada fitxer reaccioni dinàmicament a l'estat de la selecció. Per exemple, quan un fitxer és seleccionat o marcat per a ser tallat, el component File actualitza les seves classes CSS amb Tailwind per reflectir visualment aquest canvi. Aquesta reactivitat s'aconsegueix mitjançant hooks personalitzats que se subscriuen a l'store de Zustand.

```
1  const File = (- file ": FileProps) =¿ -
2    // ...
3    const isItemSelected = useIsItemSelected(file);
4    const isCutFile = useIsCutFile(file);
5    // ...
6    return (
7      ¡Card
8        className=-cx(
9          "file-item flex flex-col items-center justify-center p-2 rounded
     -lg cursor-pointer",
10         isItemSelected ? "bg-blue-100 border-blue-300" : "hover:bg-gray
     -100",
11         isCutFile && "opacity-50"
12       )"
13       // ...
14     ¿
15       -/* ... contingut del fitxer ... */"
16     ¡/Card¿
17   );
18 ";
```

LISTING 9.49: Estils dinàmics del component 'File' a 'index.tsx'

```
1  export function useIsItemSelected(file: FileItem) -
2    return useFileSelectionStore((state) =¿ state.selectedFilesIds.some(f
     =¿ f === file.id));
3  "
4
```

```
function isCutFile(file: FileItem, state: FileSelectionState) -
  const res = Array.from(state.clipboard).filter(f =¿ f.id === file.id);
  return res.length ¿ 0 && state.isCut;
"

export function useIsCutFile(file: FileItem) -
  return useFileSelectionStore((state) =¿ isCutFile(file, state));
"
```

LISTING 9.50: Implementació dels hooks 'useIsItemSelected' i 'useIsCutFile' a 'fileSelectionStore.ts'

El fragment de codi anterior mostra com els booleans isItemSelected i isCutFile s'utilitzen per aplicar classes condicionalment. Aquests booleans no són estats locals del component, sinó el resultat de dos hooks personalitzats, useIsItemSelected i useIsCutFile. Aquests hooks implementen un patró de subscripció selectiva a l'store de Zustand (fileSelectionStore).

Aquest mecanisme és clau per a l'eficiència de la interfície. Quan l'usuari selecciona un fitxer, l'estat selectedFiles de l'store canvia. Zustand notifica només aquells components que estan subscrits a aquesta part específica de l'estat. En aquest cas, només els components File afectats (el que s'acaba de seleccionar i el que s'acaba de deseleccionar) rebran el nou valor del hook isItemSelected i es tornaran a renderitzar per actualitzar el seu estil. La resta de fitxers a la graella no es veuen afectats i no es renderitzen de nou, evitant així càlculs innecessaris i mantenint una experiència d'usuari fluida, fins i tot en directoris amb centenars d'elements.

Les accions de l'usuari s'inicien des de diversos punts d'entrada que utilitzen aquest estat centralitzat. El menú contextual (ui-new/src/components/FileManagerContextMenu/index.tsx) proporciona les opcions visuals. Paral·lelament, el hook ui-new/src/hooks/useFileShortcuts.ts afegeix un *listener* global per a dreceres com Ctrl+C o Ctrl+X, que criden les funcions copyFiles o cutFiles de l'store. Finalment, el component ui-new/src/pages/FileManager/index.tsx utilitza dnd-kit per a la funcionalitat d'arrossegar i deixar anar. Cal destacar que, tot i que la lògica permet operacions amb múltiples fitxers mitjançant arrossegar i deixar anar, la representació visual actualment només mostra un únic element sent arrossegat. Aquesta és una millora pendent que s'exposa al Capítol 12.

Independentment del disparador utilitzat, l'execució final de la lògica de negoci es delega sempre a les mutacions definides al hook ui-new/src/hooks/useFileOperations.ts. Per al **renombrament (UC-09A)**, l'opció del menú contextual obre el diàleg ui-new/src/components/Fil En confirmar el nou nom, una funció de callback al FileManager/index.tsx (handleRename) invoca la mutació updateItem del hook. Per al **moviment (UC-09B)**, tant l'acció d'arrossegar i deixar anar com la confirmació des del diàleg de selecció de carpeta acaben cridant a la mutació moveItem. Per a les operacions de **copiar i enganxar (UC-09C)**, la drecera Ctrl+V al useFileShortcuts.ts llegeix el contingut del fileSelectionStore i invoca la mutació pasteFiles. Aquesta única mutació gestiona tant la còpia (fileService.copyElement) com el tallat (fileService.moveElement), basant-se en el valor del booleà isCut de l'store.

```
const handleDragEnd = useCallback(async (event: DragEndEvent) =¿ -
    const - active, over " = event;
    setActiveDragItem(null);

    if (!over —— active.id === over.id) -
        return;
```

```
7         "
8       if(!active.data.current —— !over.data.current) -
9         return;
10      "
11
12      const draggedItem = active.data.current.file as FileItem;
13      const dropTarget = over.data.current.file as FileItem;
14      if (dropTarget && dropTarget.id) -
15          try -
16              if (draggedItem.id !== dropTarget.id && draggedItem.id !==
    dropTarget.parent && dropTarget.type === 'folder') -
17                  const itemsToMove = selectedFiles.length ¿ 0 &&
    selectedFiles.some(f =¿ f === draggedItem)
18                  ? selectedFiles
19                  : [draggedItem];
20              await fileOperations.moveItem(-
21                  items: itemsToMove,
22                  toFolderId: dropTarget.id.toString(),
23                      fromFolderId: fileOperations.currentDirectory.id ——
    'root'
24              ");
25              notifications.success('Items moved successfully');
26              "
27          " catch -
28              notifications.error('Could not move the item');
29          "
30      "
31  ", [selectedFiles, fileOperations, notifications]);
```

LISTING 9.51: Gestió del Drag & Drop a 'FileManager/index.tsx'

```
1  useEffect(() =¿ -
2    const handleKeyDown = async (event: KeyboardEvent) =¿ -
3      if (event.ctrlKey —— event.metaKey) -
4        switch (event.key.toLowerCase()) -
5          case 'c':
6            event.preventDefault();
7            if(isInRoot —— (isInShared && currentDirectory.shared &&
    currentDirectory.accessLevel !== 'READ')) -
8            copyFiles(currentDirectory.id!.toString());
9            " else -
10             notifications.error("You don't have permission to copy files
    ");
11           "
12           break;
13
14         case 'x':
15           event.preventDefault();
16           if(isInRoot —— (isInShared && currentDirectory.shared &&
    currentDirectory.accessLevel !== 'read')) -
17           cutFiles(currentDirectory.id!.toString());
18           " else -
19             notifications.error("You don't have permission to cut files
    ");
20           "
21           break;
22
23         case 'v':
24           event.preventDefault();
25           if(isInRoot —— (isInShared && currentDirectory.accessLevel !==
     'read')) -
26           const items = Array.from(clipboard);
27           if(items.length ¿ 0) -
```

```
28                pasteFiles(-items: items, targetFolderId: currentDirectory
    .id!.toString(), prevParentId: clipboardParentId, isCut: isCut");
29              "
30              if(isCut) -
31                setClipboardFiles(new Set(), undefined);
32              "
33            " else -
34              notifications.error("You don't have permission to paste
    files");
35            "
36            break;
37        "
38      "
39    ";
40
41    window.addEventListener('keydown', handleKeyDown);
42    return () =¿ window.removeEventListener('keydown', handleKeyDown);
43  ", [selectedFiles, currentDirectory, clipboard, isCut, clipboardParentId
    , /* ... */]);
```

LISTING 9.52: Gestió de dreceres de teclat a 'useFileShortcuts.ts'

Totes aquestes mutacions (updateItem, moveItem, pasteFiles) segueixen el patró d'**actualització optimista** implementat amb TanStack Query, proporcionant una resposta visual immediata a l'usuari mentre la comunicació amb el backend es processa en segon pla.

UC-10: Eliminar Arxiu (Moure a la Paperera) Quan eliminem un arxiu des de la vista principal del gestor, no l'estem esborrant per sempre, sinó que el movem a la paperera de reciclatge, com si fos un pas intermedi.

Podem començar a eliminar un arxiu de dues maneres: fent clic amb el botó dret per obrir el menú contextual a ui-new/src/components/FileManagerContextMenu/index.tsx o simplement prement la tecla Delete. El hook ui-new/src/hooks/useFileShortcuts.ts s'encarrega de capturar quan premem Delete i, igual que el menú contextual, crida la funció deleteItem del hook ui-new/src/hooks/useFileOperations.ts. Això fa que, independentment de com decidim eliminar l'arxiu, l'experiència sigui la mateixa.

El que fa que aquesta funcionalitat sigui interessant és la lògica dual de la mutació deleteItemMutation. El que passa quan eliminem un arxiu depèn totalment de la secció on estem, informació que ens proporciona el ui-new/src/store/fileContextStore.ts. Aquest store ens diu on som (root, shared o trash), i això permet que la mateixa acció (prémer Delete o fer clic a "Eliminar") faci coses diferents segons el context.

```
1 else if (event.key === 'Delete') -
2   event.preventDefault();
3   if (fileContext.section === 'shared') -
4     revokeAccess(-items: selectedFiles")
5   " else -
6     deleteItem(-items: selectedFiles, section: fileContext.section");
7   "
8   clearSelection();
9 "
```

LISTING 9.53: Gestió de la tecla Delete a 'useFileShortcuts.ts'

Dins de la funció mutationFn de la mutació deleteItemMutation, un simple condicional decideix què fer. Si estem a la secció 'root', es crida a fileService.deleteElement, que mou l'element a la paperera al backend. Però si estem a la secció 'trash', es crida

a trashService.deleteItem, que esborra l'element de manera permanent. Aquest disseny ens permet utilitzar la mateixa interfície d'usuari per a dues operacions molt diferents, fent que la lògica sigui més senzilla als components de la interfície mentre mantenim la complexitat al nivell de servei.

```
const deleteItemMutation = useMutation(-
  mutationFn: async (- items, section ": - items: FileItem[], section: '
    root' — 'trash' — 'shared' ") =¿ -
    try -
      if(section === 'root') -
        return Promise.all(items.map(item =¿ fileService.deleteElement(
    item.id!.toString())));
      " else if(section === 'trash') -
        return Promise.all(items.map(item =¿ trashService.deleteItem(
    item.id!.toString())));
      " else -
        notifications.error('Cannot delete items from this section');
        return;
      "
    " catch (error) -
      console.error('Error deleting items:', error);
      notifications.error('Error deleting items');
      return;
    "
  ",
  onSuccess: (', variables) =¿ -
    notifications.success(
      variables.section === 'root'
        ? 'Items moved to trash correctly'
        : 'Items deleted permanently'
    );
    refreshAll();
  "
  // ... onMutate i onError per a l'actualització optimista
");
```

LISTING 9.54: Lògica dual de la mutació d'eliminació a 'useFileOperations.ts'

Igual que les altres mutacions del sistema, deleteItemMutation utilitza el patró d'actualització optimista de TanStack Query. Quan eliminem un element, aquest desapareix de la interfície immediatament gràcies a l'onMutate, que actualitza la cache local, mentre que l'operació real es fa en segon pla. Les notificacions d'èxit ens donen un feedback específic segons el tipus d'operació: Ïtems moved to trash correctly"per al moviment a la paperera o Ïtems deleted permanently"per a l'esborrat definitiu.

9.4.5 Compartició d'Arxius

UC-13, UC-13A, UC-13B: Compartir, Deixar de Compartir i Veure Compartits Aquesta funcionalitat permet als usuaris compartir els seus arxius i carpetes amb altres, gestionar els permisos i veure els elements que altres han compartit amb ells.

La navegació entre les diferents seccions del gestor d'arxius (root, shared, trash) es gestiona a través d'un canvi de context en lloc d'un canvi de ruta. L'usuari inicia aquesta navegació fent clic a les icones corresponents a la barra lateral, implementada al component ui-new/src/components/FileDirectorySidebar/index.tsx. Aquesta acció invoca la funció handleNavigateToSection, que actualitza l'estat global a través del ui-new/src/store/fileContextStore.ts, establint la secció activa. Aquest canvi de

context fa que el hook useFileOperations obtingui les dades del directori arrel corresponent a la secció seleccionada, poblant la vista principal amb els fitxers i carpetes adequats.

```
const handleNavigateToSection = useCallback(async (section: 'root' — '
    shared' — 'trash', e: React.MouseEvent) =¿ -
  e.preventDefault();
  e.stopPropagation();

  try -
    let folder: FileItem;
    switch (section) -
      case 'root':
        folder = await fileService.getRootFolder();
        break;
      case 'shared':
        folder = await sharingService.getSharedRootFolder();
        folder.id = "shared";
        break;
      case 'trash':
        folder = await trashService.getTrashRootFolder();
        folder.id = "trash";
        break;
    "
    await setCurrentDirectory(folder);
    fileContext.setSection(section);
  " catch -
    notifications.error('Loading error');
  "
", [fileContext, setCurrentDirectory, notifications]);
```

LISTING 9.55: Navegació entre seccions a 'FileDirectorySidebar/index.tsx'

Un cop dins d'una secció, l'usuari pot navegar per l'arbre de directoris fent clic als elements de la barra lateral (ui-new/src/components/FileDirectorySidebarItem/index.tsx) o fent doble clic sobre una carpeta a la vista principal (ui-new/src/components/File/index.tsx). Ambdues accions invoquen la funció setCurrentDirectory del hook useFileOperations, que carrega el contingut de la carpeta seleccionada i actualitza la vista.

La funció setCurrentDirectory és fonamental per a la gestió de l'estat de la navegació. Aquesta funció actualitza manualment la memòria cache de TanStack Query mitjançant queryClient.setQueryData, establint la carpeta seleccionada com el nou directori actual. Aquest procés actualitza l'estat global a Zustand a través de setStoreCurrentDirectory i persisteix l'ID del directori a la sessionStorage per mantenir l'estat entre recàrregues de la pàgina.

Aquesta acció desencadena l'execució de la query currentDirectoryQuery, que utilitza el hook useQuery per obtenir les dades completes del directori des del backend. La clau d'aquesta query (queryKey) és un array que inclou l'identificador del directori actual, assegurant que cada directori tingui la seva pròpia entrada a la memòria cache. La configuració staleTime de 5 minuts indica a TanStack Query que consideri les dades a la memòria cache com a "fresques"durant aquest període, evitant crides innecessàries a l'API si l'usuari torna a un directori visitat recentment.

```
const setCurrentDirectory = async (folder: FileItem) =¿ -
  if (folder.id) -
    const newId = folder.id.toString();
    setStoreCurrentDirectory(folder);
```

```
    safeSessionStorage.setItem(STORAGE_KEYS.CURRENT_DIRECTORY_ID, newId);
  `

  await queryClient.setQueryData([QUERY_KEYS.CURRENT_DIRECTORY, folder.id], folder);
  setStoreCurrentDirectory(folder);
`;

const currentDirectoryQuery = useQuery(-
  queryKey: [QUERY_KEYS.CURRENT_DIRECTORY, currentDirectory.id],
  queryFn: () =¿ -
    if(section === 'trash') -
      return fileService.getFolderById(currentDirectory.id, true);
    ` else -
      return fileService.getFolderById(currentDirectory.id, false);
    `
  `,
  staleTime: 1000 * 60 * 5,
  enabled: !!localStorage.getItem('accessToken') && !!currentDirectory.id,
`);
```

LISTING 9.56: Gestió del directori actual amb 'useQuery' a 'useFileOperations.ts'

Aquest mecanisme de memòria cache és essencial per a una experiència de navegació fluida. Quan l'usuari navega a un directori ja visitat, les dades es carreguen instantàniament des de la memòria cache, eliminant la latència de la xarxa. La dada només es torna a demanar al servidor si ha passat el staleTime o si la memòria cache s'invalida manualment després d'una operació (com crear o eliminar un fitxer), assegurant un equilibri òptim entre rendiment i consistència de les dades.

```
const handleDoubleClick = async () =¿ -
  if (file.type === 'folder') -
    fileOperations.setCurrentDirectory(file);
  ` else -
    // ... lògica per descarregar fitxers
  `
`;
```

LISTING 9.57: Navegació per doble clic a 'File/index.tsx'

L'acció de compartir s'inicia des del menú contextual, que obre el component ui-new/src/components/File/S el centre neuràlgic de la gestió de permisos. Cal destacar que aquesta funcionalitat només està disponible per a fitxers i carpetes propis de l'usuari, és a dir, quan es troba a la secció root. En obrir-se el diàleg, aquest utilitza el hook ui-new/src/hooks/useShareManager.ts per carregar l'estat actual dels permisos per a tots els fitxers seleccionats, una informació que es desa per a una comparació posterior.

Una de les optimitzacions clau d'aquesta funcionalitat rau en la lògica de la funció share, passada com a callback onShare des de FileManager/index.tsx. En lloc d'enviar simplement l'estat final dels permisos, aquesta funció realitza una operació de "diferència"entre l'estat original (guardat en obrir el diàleg) i l'estat final (quan l'usuari fa clic a "Share"). El codi itera sobre l'estat final per identificar nous permisos o modificacions, i sobre l'estat original per identificar permisos revocats. Això genera una llista precisa de les accions necessàries (shareWithUser, handleUpdateAccess, handleUnshareToUser), minimitzant el nombre de crides a l'API.

```
const share = useCallback(async (userAccess: -username: string, fileId:
    string, accessType: 'READ' — 'WRITE' "[], originalSharedAccess: Map¡
    string, -username: string, fileId: string, accessType: 'READ' — '
    WRITE' "[]¿) =¿ -
  const promises = [];
  if(originalSharedAccess.size ¿ 0) -
    for (const access of userAccess) -
      if(!originalSharedAccess.has(access.username) —— !
    originalSharedAccess.get(access.username)!.some(a =¿ a.fileId ===
    access.fileId)) -
        promises.push(shareManager.shareWithUser(access.fileId, access.
    username, access.accessType));
      " else if(originalSharedAccess.get(access.username)!.some(a =¿ a.
    fileId === access.fileId && a.accessType !== access.accessType)) -
        promises.push(shareManager.handleUpdateAccess(access.fileId,
    access.username, access.accessType));
      "
    "
    for (const user of originalSharedAccess.keys()) -
      if(!userAccess.some(a =¿ a.username === user)) -
        originalSharedAccess.get(user)!.forEach(a =¿ -
          promises.push(shareManager.handleUnshareToUser(a.fileId, user)
    );
        ");
      " else -
        originalSharedAccess.get(user)!.forEach(a =¿ -
          if(!userAccess.some(access =¿ access.username === user &&
    access.fileId === a.fileId)) -
            promises.push(shareManager.handleUnshareToUser(a.fileId,
    user));
          "
        ");
      "
    "
    await Promise.all(promises);
  " else -
    for (const access of userAccess) -
      promises.push(shareManager.shareWithUser(access.fileId, access.
    username, access.accessType));
    "
    await Promise.all(promises);
  "
", [shareManager]);
```

LISTING 9.58: Lògica de comparació de permisos a 'FileManager/index.tsx'

Aquesta optimització va ser una millora directa basada en el feedback d'un *beta tester*, que va assenyalar que compartir múltiples fitxers amb múltiples usuaris era un procés lent i afarragós. La funció de comparació utilitza el hook useShareManager com a intermediari, que conté funcions més granulars que són les que finalment criden als mètodes corresponents del sharingService.

```
const shareWithUser = useCallback(async (fileId: string, username:
    string, accessType: 'READ' — 'WRITE' = 'READ') =¿ -
  if (!username) -
    console.error("Please enter a username");
    return false;
  "
  try -
    await sharingService.shareFile(fileId, username, accessType);
    notifications.success('Item shared successfully');
```

```
    console.log(`File ${fileId} shared with ${username} with access type
     ${accessType}`);
    return true;
  } catch (error) {
    console.error('Error sharing file:', error);
    notifications.error('Could not share item');
    return false;
  }
}, []);
```

LISTING 9.59: Funció per compartir amb un usuari a 'useShareManager.ts'

A més de mostrar els fitxers compartits, la secció shared ofereix opcions contextuals diferents. El menú contextual ui-new/src/components/FileManagerContextMenu/index.tsx adapta les seves opcions a la secció activa, mostrant l'opció "Stop seeing"quan l'usuari es troba a la secció de compartits. Aquesta acció invoca la funció revokeAccess del hook useFileOperations, que s'encarrega de revocar el permís de l'usuari sobre el fitxer seleccionat, eliminant-lo de la seva vista de compartits.

Aquesta adaptabilitat s'aconsegueix mitjançant una sèrie de renderitzats condicionals dins del component, que comproven la secció activa (isInRoot, isInTrash, isInShared) i si s'ha seleccionat un fitxer per mostrar només les accions pertinents.

```
const handleUnshare = useCallback(async () => {
  if (!file) return;

  await revokeAccess({items: selectedFiles});
  onClose();
}, [file, selectedFiles, onClose, revokeAccess]);
// ... //
<div className="min-w-[220px] p-1">
  {/* Opcions quan NO hi ha cap fitxer seleccionat (Crear, Pujar,
    Enganxar...) */}
  {!file && (isInRoot || (isInShared && currentDirectory.accessLevel ===
     'WRITE')) ? (
    <>
      <div onClick={() => setIsCreateFolderOpen(true)}>
        <RiFolderLine />
        <span>New folder</span>
      </div>
      {/* ... Altres opcions de creació i enganxar ... */}
    </>
  ) : (
    <>
      {/* Opcions quan SÍ hi ha un fitxer seleccionat */}

      {/* Opcions per a la secció ROOT */}
      {isInRoot && (
        <>
          <div onClick={handleCopy}><span>Copy</span></div>
          <div onClick={handleCut}><span>Cut</span></div>
          <div onClick={() => setOpenShareDialog(true)}><span>Share</
    span></div>
          <div onClick={handleDelete}><span>Delete</span></div>
        </>
      )}

      {/* Opcions per a la secció TRASH */}
      {isInTrash && (
        <>
```

```
35            ¡div onClick=-handleRestore"¿¡span¿Restore¡/span¿¡/div¿
36            ¡div onClick=-handleDelete"¿¡span¿Delete permanently¡/span¿¡/
    div¿
37          ¡/¿
38        )"
39
40        -/* Opcions per a la secció SHARED (renderitzades per una funció
    auxiliar) */"
41        -isInShared && renderSharedOptions()"
42
43        -/* Opció comuna per veure detalls si només hi ha un fitxer
    seleccionat */"
44        -selectedFiles.length === 1 && (
45          ¡div onClick=-handleDetails"¿¡span¿Details¡/span¿¡/div¿
46        )"
47      ¡/¿
48    )"
49  ¡/div¿
```

LISTING 9.60: Renderitzat condicional de les opcions del menú a 'FileManagerContextMenu/index.tsx'

Cal destacar que, de manera similar a l'operació de pujada de carpetes, el procés de compartir múltiples fitxers amb múltiples usuaris no està optimitzat per a grans volums de dades. Tot i que és menys probable que un usuari modifiqui milers de permisos manualment a través de la interfície, l'arquitectura actual realitza una crida a l'API per a cada modificació individual de permís. L'optimització d'aquest flux mitjançant operacions per lots (*batch operations*) es considera una de les millores futures, tal com es detalla al Capítol 12.

9.5 Implementació del client web

9.5.1 Introducció

El client web és una *Single Page Application* (SPA) desenvolupada amb les tecnologies més modernes de l'ecosistema de JavaScript, com són React, Vite i TypeScript. La interfície d'usuari s'ha construït utilitzant el framework CSS **Tailwind CSS** per a un disseny àgil i personalitzable, juntament amb un conjunt de components reutilitzables i accessibles de **Radix UI**, que han estat encapsulats i estilitzats al directori ui-new/src/components. La gestió de l'estat global de l'aplicació es realitza mitjançant **Zustand**, una solució lleugera i potent, mentre que les operacions asíncrones, el cacheig de dades del servidor i les actualitzacions optimistes es gestionen amb la llibreria **TanStack React Query**, garantint una experiència d'usuari fluida i reactiva.

L'arquitectura del codi font segueix un enfocament modular i organitzat, separant clarament les diferents responsabilitats: la lògica de negoci s'encapsula en *hooks* reutilitzables, l'estat global en *stores* de Zustand, les crides a l'API del backend s'abstrauen en serveis, i els components d'interfície són purs i centrats exclusivament en la representació visual.

Estructura de directoris del client web

El projecte del client web, ubicat a ui-new/, presenta una estructura de directoris clara i modular que separa les diferents responsabilitats de l'aplicació. A continuació, es detallen els directoris principals i el seu contingut:

- src/components/: Aquest directori conté tots els components reutilitzables de React que conformen la interfície d'usuari. Alguns dels components més rellevants són:
  - **Components de Tremor**: S'han integrat diversos components de la llibreria de components de React de codi obert Tremor [**tremor**], adaptats per a aquest projecte. Aquests inclouen Accordion, Card, Dialog, Drawer, Input, Popover, Toast, i Tooltip. Aquests components proporcionen una base sòlida i accessible per a la construcció d'interfícies d'usuari complexes.
  - Button: Un botó personalitzat amb diferents variants visuals (primari, secundari, fantasma, etc.).
  - CreateNewFolderDialog: Un diàleg modal per a la creació de noves carpetes.
  - File: Component que representa un fitxer o carpeta a la interfície, amb la seva icona i nom. Inclou subcomponents com RenameDialog i ShareDialog per a la gestió d'aquestes accions.
  - Tabs i TabsSubHeader: Components per a la creació de navegació per pestanyes.
  - Toaster: Component que gestiona la cua i visualització de notificacions (toasts).

- src/pages/: Conté els components que representen les pàgines completes de l'aplicació, com ara:
  - AdminDashboard: El panell d'administració d'usuaris.
  - FileManager: La pàgina principal de gestió de fitxers.
  - Login i SignUp: Les pàgines d'inici de sessió i registre.
- src/hooks/: Aquest directori és fonamental per a la lògica de negoci, ja que conté els *hooks* personalitzats de React que encapsulen la majoria de la complexitat:
  - useAuth: Gestiona l'estat d'autenticació de l'usuari, incloent l'inici de sessió, el tancament de sessió i el registre.
  - useFileOperations: Conté tota la lògica per a les operacions de fitxers (crear, moure, copiar, eliminar, etc.), utilitzant TanStack Query per a les actualitzacions optimistes.
  - useFileSelection: Gestiona la selecció de fitxers i el porta-retalls (copiar/-tallar).
  - useShareManager: Encapsula la lògica per a compartir fitxers i gestionar els permisos.
  - useValidation: Proporciona funcions per a la validació de formularis.
- src/layouts/: Conté els components d'estructura de la pàgina, com AppLayout, que defineix la disposició general amb la barra lateral i la capçalera.
- src/lib/: Un directori per a utilitats i configuracions generals:
  - api.ts: Configura Axios i defineix els serveis per a interactuar amb l'API del backend (authService, fileService, etc.).
  - utils.ts: Funcions d'utilitat genèriques.
  - websocket.ts: Gestiona la connexió WebSocket per a les actualitzacions en temps real.
- src/store/: Conté els *stores* de Zustand per a la gestió de l'estat global, com ara fileStore (estat dels fitxers) i fileSelectionStore (estat de la selecció de fitxers).
- src/types/: Defineix els tipus de TypeScript utilitzats a tota l'aplicació, garantint la consistència i la seguretat del tipat.

9.5.2 Disseny Visual i Components de la Interfície

A continuació, es presenta una descripció visual detallada de la implementació final del client web. Aquesta secció té un doble objectiu: d'una banda, il·lustrar com els esbossos conceptuals presentats al **Capítol 8 (secció 8.5.1)** s'han materialitzat en una interfície funcional i interactiva; de l'altra, justificar les decisions de disseny preses

durant la implementació que divergeixen o milloren les propostes inicials, sempre amb l'objectiu de perfeccionar l'experiència d'usuari.

Autenticació

El flux d'autenticació és el primer punt de contacte de l'usuari amb la plataforma. El disseny s'ha centrat en la claredat i la simplicitat, evitant distraccions per facilitar un accés ràpid i segur.

FIGURA 9.46: Pantalla d'inici de sessió.

FIGURA 9.47: Pantalla de registre.

Com es pot observar, la implementació final de les pantalles d'inici de sessió (**Figura 9.46**) i registre (**Figura 9.47**) és una traducció fidel dels esbossos conceptuals presentats a les **Figures 8.21 i 8.22** del Capítol 8. El disseny final és minimalista i funcional, utilitzant un component de Card per enmarcar els formularis i millorar-ne la llegibilitat. Una millora funcional no detallada en la fase de disseny inicial és el sistema de validació de dades. A més de la validació de la lògica de negoci que es produeix en el moment de l'enviament del formulari, s'ha incorporat una validació prèvia al client. Com es pot apreciar a la **Figura 9.48**. Això millora l'experiència d'usuari i redueix la càrrega del servidor.

FIGURA 9.48: Exemple de validació nativa del navegador en el formulari de registre.

Escriptori Principal i Navegació

L'escriptori principal és el nucli de l'aplicació, on es centralitzen totes les funcionalitats de gestió d'arxius. A diferència del boceto general de la **Figura 8.23**, la implementació final opta per un *layout* més net i dinàmic, on els components s'organitzen de manera fluida per adaptar-se a diferents mides de pantalla.

FIGURA 9.49: Detall de l'arbre de navegació.

L'arbre de navegació lateral (Figura 9.49) és una implementació fidel i funcional del concepte descrit a la **Figura 8.25**. Permet a l'usuari desplaçar-se de manera intuïtiva entre els seus arxius, la secció de Çompartits amb miï la "Paperera", distingint visualment la carpeta seleccionada.

FIGURA 9.50: Diàleg del sistema per a la selecció d'arxius.

FIGURA 9.51: Notificació emergent o toast.

La pujada de fitxers es realitza mitjançant el diàleg natiu del sistema operatiu (Figura 9.50, esquerra), oferint a l'usuari una experiència familiar i intuïtiva per seleccionar arxius i carpetes. Com a millora clau per al *feedback* a l'usuari, s'han incorporat notificacions emergents o *toasts* (Figura 9.51, dreta). Aquestes s'utilitzen a tota l'aplicació per comunicar l'èxit o el fracàs de les operacions, proporcionant una resposta visual immediata.

FIGURA 9.52: Modal de configuració del perfil d'usuari.

FIGURA 9.53: Vista per a un usuari estàndard.

FIGURA 9.54: Modal de configuració del perfil, amb vistes diferenciades per rol.

Pel que fa a les accions globals, la **Figura 9.54** mostra el modal de configuració del perfil, que correspon al disseny de la **Figura 8.28**. La implementació, però, és més avançada i dinàmica. El contingut s'adapta al rol de l'usuari per motius de seguretat. Com es pot veure, la vista per a un usuari estàndard (dreta) inclou un botó per eliminar el seu propi compte. En canvi, a la vista per a un usuari amb rol SUPER_ADMIN (esquerra), aquest botó no hi és. Aquesta és una restricció deliberada per impedir que l'administrador principal es pugui eliminar a si mateix, la qual cosa deixaria el sistema sense un compte amb control total.

FIGURA 9.55: Diàleg modal per a la creació d'una nova carpeta.

La creació de noves carpetes, una de les "accions sobre la carpeta actual"esmentades a la **Figura 8.26**, es gestiona mitjançant un diàleg modal simple, com es mostra a la **Figura 9.55**. Aquesta elecció de disseny minimitza la interrupció del flux de treball

de l'usuari, ja que el diàleg apareix superposat al gestor de fitxers i requereix una acció directa (crear o cancel·lar) per continuar.

FIGURA 9.56: Ordenar.

FIGURA 9.57: Panell admin.

FIGURA 9.58: Configuració.

FIGURA 9.59: Tancar sessió.

FIGURA 9.60: Controls globals de l'aplicació.

Els controls per a les "opcions globals"de la **Figura 8.27** s'han implementat a la cantonada superior dreta de la interfície. Com es pot apreciar a la **Figura 9.60**, el disseny final agrupa de manera lògica les opcions per ordenar el contingut, accedir al panell d'administració, a la configuració de l'usuari i per tancar la sessió, utilitzant icones minimalistes per mantenir una estètica neta i organitzada.

Interacció amb Elements

La interacció amb arxius i carpetes s'ha dissenyat per ser rica i contextual. El menú d'accions, que s'activa tant amb el clic dret com a través d'un botó dedicat, és una implementació directa del concepte de la **Figura 8.29** i ofereix totes les operacions rellevants per a l'element seleccionat.

FIGURA 9.61: Clic dret.

FIGURA 9.62: Botó d'opcions.

FIGURA 9.63: Opcions a l'arrel.

FIGURA 9.64: Menú contextual dinàmic.

La **Figura 9.64** mostra com el menú contextual apareix tant en fer clic dret sobre un element o les opcions que es mostren tant al fer click dert pero a cap element, com en prémer el botó d'opcions. Aquest menú és dinàmic i adapta les seves opcions al context, com per exemple a la seccio dels fitxer del propietari (*root*), on permet crear nous elements.

FIGURA 9.65: Selecció múltiple.

FIGURA 9.66: Feedback visual de "Tallar".

FIGURA 9.67: Panell de detalls.

La implementació final inclou millores significatives d'usabilitat. S'ha implementat la selecció múltiple (Figura 9.65) per realitzar operacions en lot, i es proporciona un *feedback* visual clar per a accions com "Tallar", on l'element es mostra semitransparent (Figura 9.66). Per a la visualització de detalls, es va decidir substituir el modal proposat a la **Figura 8.30** per un panell lateral o *Drawer* (Figura 9.67). Aquesta decisió millora l'experiència en permetre a l'usuari consultar la informació sense perdre el context de la vista d'arxius. A més, com a resultat d'una millora proposada per un *beta tester*, aquest panell mostra també la llista d'usuaris amb qui s'ha compartit un fitxer, oferint l'opció de modificar el seu nivell d'accés o deixar de compartir-lo. Això permet gestionar la compartició d'un element de forma més àgil, sense haver de navegar a la secció de fitxers compartits.

FIGURA 9.68: Diàleg per canviar el nom d'un element.

El diàleg per canviar el nom d'un element (**Figura 9.68**) és una implementació directa i funcional del concepte mostrat a la **Figura 8.31**, proporcionant una manera ràpida i senzilla de realitzar aquesta operació comuna.

FIGURA 9.69: Vista inicial.

FIGURA 9.70: Vista amb carpeta desplegada.

FIGURA 9.71: Modal per moure elements.

El diàleg per moure elements (**Figura 9.71**) és una evolució significativa del seu esbós inicial (**Figura 8.32**). La implementació final és molt més avançada, ja que presenta un arbre de carpetes completament navegable. Com es pot apreciar, l'usuari pot desplegar les subcarpetes (dreta) per seleccionar la ubicació de destí de manera precisa, una millora substancial d'usabilitat respecte a un simple camp de text. Aquest dialeg es utilitzable tant per els fitxers o carpetes dintre de la seccio *root* com per la seccio de *Compartits amb mi* on es poden moure els elements dintre de l'arbre de carpetes compartides amb l'usuari (desde la primera carpeta a l'arrel dels compartits fins l'ultim fill), aquesta limitacio es va posar per motius de seguretar per evitar que

es moguin fitxers entre carpetes d'usuaris diferents sense permis explicit, de forma nativa a l'aplicacio.

FIGURA 9.72: Vista principal.

FIGURA 9.73: Cerca d'usuaris.

FIGURA 9.74: Modal de compartició d'arxius.

Finalment, el diàleg per compartir arxius (**Figura 9.74**) és substancialment més complet que el seu concepte a la **Figura 8.33**. La versió implementada permet buscar usuaris pel seu nom (dreta), assignar permisos específics de lectura o escriptura, i mostra una llista clara dels usuaris que ja tenen accés, amb opcions per modificar o revocar els seus permisos directament des de la mateixa interfície, oferint una gestió de compartició molt més potent, ja que permet compartir llistes d'elements d'una sola accio, evitant que l'usuari hagi de compartir un per un.

Seccions Específiques

Les seccions de la Paperera, els Elements Compartits i el Panell d'Administració disposen de vistes i opcions contextuals pròpies.

FIGURA 9.75: Menú contextual a la secció Çompartit amb mi".

Elements Compartits El menú contextual per a un element a la secció Çompartit amb mi"(**Figura 9.75**) és un exemple de la naturalesa dinàmica de la interfície. Com

es va planejar a la **Figura 8.34**, les opcions disponibles ("Descargar", "Dejar de seguir", etc.) canvien en funció dels permisos (lectura o escriptura) que l'usuari tingui sobre l'element, oferint només les accions permeses.

Paperera La vista de la paperera (**Figura 9.76**) implementa fidelment les opcions conceptualitzades a la **Figura 8.36**, oferint les accions de "Restaurarï "Eliminar definitivament"per a cada element.

FIGURA 9.76: Opcions contextuals de la paperera.

Panell d'Administració El panell d'administració (**Figura 9.77**) representa una de les millores més significatives respecte al seu disseny inicial (**Figura 8.37**). En lloc d'una simple llista, s'ha implementat una taula de dades completa. A més, la interfície és dinàmica i conscient dels permisos.

FIGURA 9.77: Panell d'administració d'usuaris.

FIGURA 9.78: Diàleg per a l'edició d'un usuari des del panell d'administració (vista SUPER_ADMIN).

L'edició d'usuaris és una de les funcionalitats clau del panell. En seleccionar l'opció d'editar, s'obre un diàleg modal que permet modificar les dades de l'usuari, tal com il·lustra la **Figura 9.78**. La implementació estableix una clara jerarquia de permisos: mentre que un administrador estàndard (ADMIN) pot actualitzar informació bàsica, només un superadministrador (SUPER_ADMIN) té l'autoritat per canviar rols o restablir contrasenyes. Aquesta segregació de privilegis, que reserva les accions més crítiques al rol més alt, és una mesura de seguretat fonamental. La decisió d'atorgar aquests poders al SUPER_ADMIN es fonamenta en la seva funció com a responsable últim del sistema. Tot i que aquesta centralització de permisos representa un risc de seguretat calculat, es considera una capacitat indispensable per a la gestió d'incidències crítiques, com ara la recuperació de l'accés per a un usuari que hagi perdut la contrasenya. Aquesta responsabilitat és coherent amb el seu rol com a responsable de la integritat de les dades i del compliment de la normativa vigent.

FIGURA 9.79: Botó d'eliminació actiu.

FIGURA 9.80: Botó d'eliminació desactivat.

FIGURA 9.81: Comportament dinàmic del botó d'eliminació.

Com es pot veure, el botó "Delete"és interactiu quan es tenen permisos sobre l'usuari (esquerra), però es mostra desactivat i no és funcional quan s'intenta eliminar un compte protegit, com el del propi SUPER_ADMIN (dreta). Aquesta restricció visual i funcional és una mesura de seguretat clau per garantir la integritat del sistema.

FIGURA 9.82: Diàleg de confirmació per a l'eliminació d'un usuari.

Finalment, abans de qualsevol eliminació, es mostra un diàleg de confirmació per prevenir accions accidentals (**Figura 9.82**), una mesura de seguretat afegida durant la implementació que no estava als esbossos inicials.

9.5.3 Implementació per Casos d'Ús

A continuació, es detalla com l'arquitectura del client web i els seus components es combinen per donar resposta als principals casos d'ús del sistema. Cada secció descriu el flux d'implementació, els fitxers clau implicats i fragments de codi rellevants que il·lustren les decisions tècniques més importants.

Gestió d'Usuaris

UC-01: Registre d'Usuari & UC-02: Inici de Sessió La funcionalitat d'autenticació constitueix el punt d'entrada a l'aplicació. Per a un usuari no autenticat, el component principal App.tsx actua com a gestor de l'enrutament, presentant un interruptor que permet alternar entre les vistes de login (/login) i registre (/sign-up). Aquestes vistes, implementades a ui-new/src/pages/Login/index.tsx i ui-new/src/pages/SignUp/index.tsx respectivament, es construeixen amb components reutilitzables com Card per a l'estructura, i Input i Button per als formularis.

Quan l'usuari interactua amb el formulari, la funció submit del component corresponent orquestra el procés. Abans d'enviar cap dada, es realitza una validació completa dels camps mitjançant la lògica encapsulada al hook ui-new/src/hooks/useValidation.ts. Només si la validació és exitosa, s'invoca la lògica de negoci principal.

El cervell de l'autenticació resideix al hook ui-new/src/hooks/userAuth.tsx. Les seves funcions login i register són les responsables de gestionar tot el flux. Aquestes funcions no interactuen directament amb la xarxa, sinó que deleguen la comunicació al servei authService, definit a ui-new/src/lib/api.ts, que s'encarrega de construir i enviar la petició HTTP al backend.

Després d'una resposta exitosa del servidor, el hook userAuth executa una seqüència d'accions crucials per actualitzar l'estat del client: desa els tokens d'accés i de refresc al localStorage, obté les dades completes del perfil de l'usuari, actualitza l'estat global de l'aplicació i, finalment, estableix la connexió WebSocket per a la sincronització en temps real.

```
const login = useCallback(async (username: string, password: string):
    Promise¡boolean¿ =¿ -
  try -
    if(!isLogin) -
      const response = await authService.login(username, password);

      localStorage.setItem("accessToken", response.accessToken);
      localStorage.setItem("refreshToken", response.refreshToken);

      const userData = await userService.getCurrentUser();
      localStorage.setItem("username", userData.username);

      setUser(userData);
      setIsLogin(true);
      websocketService.connect();
    "
    return true;
  " catch (error) -
    // ... maneig d'errors
    return false;
  "
", [notifications, isLogin]);
```

LISTING 9.61: Fragment de la funció 'login' al hook 'userAuth.tsx'

Un cop l'usuari està autenticat, per mantenir la sessió activa de manera eficient, s'utilitza un mecanisme arquitectònic clau: un interceptor d'Axios, configurat a ui-new/src/lib/api.ts. Aquest interceptor s'activa automàticament quan una petició a l'API falla amb un codi d'estat 401 (No autoritzat). En aquest moment, intenta renovar el token d'accés utilitzant el token de refresc. Si la renovació té èxit, la petició original es reintenta amb el nou token, un procés completament imperceptible per a l'usuari.

```
api.interceptors.response.use(
    (response) =¿ response,
    async (error) =¿ -
        const originalRequest = error.config;
        const authRoutes = ['/users/auth/login', '/users/auth/keep-alive
    ', '/users/auth/register'];
        const refreshToken = localStorage.getItem('refreshToken');

        if (error.response?.status === 401 && !originalRequest.˙retry &&
     !authRoutes.includes(originalRequest.url) && refreshToken) -
            originalRequest.˙retry = true;

            try -
                const response = await api.post('/users/auth/keep-alive'
    , null, -
                    headers: -
                        'X-Refresh-Token': refreshToken
                    "
                ");

                const newAccessToken = response.headers['authorization'
    ]?.split(' ')[1];
                const newRefreshToken = response.headers['x-refresh-
    token'];

                if (newAccessToken && newRefreshToken) -
                    localStorage.setItem('accessToken', newAccessToken);
                    localStorage.setItem('refreshToken', newRefreshToken
    );

```

```
                    originalRequest.headers.Authorization = `Bearer ${newAccessToken}`;
                    return api(originalRequest);
                }
            } catch (refreshError) {
                localStorage.removeItem('accessToken');
                localStorage.removeItem('refreshToken');
                localStorage.removeItem('username');
                window.location.href = '/login';
                return Promise.reject(refreshError);
            }
        }

        return Promise.reject(error);
    }
);
```

LISTING 9.62: Interceptor d'Axios per a la renovació automàtica de tokens a api.ts

He de destacar que l'estratègia actual d'emmagatzematge de tokens al localStorage va ser una solució temporal. Inicialment, vaig intentar implementar un sistema basat en cookies HttpOnly per al transport dels tokens. Aquesta tècnica no només és una pràctica de seguretat estàndard que protegeix contra atacs de Cross-Site Scripting (XSS) en fer que els tokens siguin inaccessibles des del JavaScript del client [**bitsrc˙tokens˙vs˙cookies**], sinó que també s'alinea amb les exigències de la normativa de protecció de dades com el RGPD. Segons aquesta regulació, els identificadors en línia com els tokens es consideren dades personals i requereixen mesures tècniques per garantir-ne la seguretat [**gdpr˙cookies**].

No obstant això, vaig trobar dificultats tècniques en la comunicació amb el servidor. Per tal de no bloquejar el desenvolupament, vaig optar per l'emmagatzematge local com a solució intermèdia, amb la intenció de refactoritzar-la en un futur. La implementació d'un flux d'autenticació robust basat en cookies no només és una millor pràctica de seguretat altament recomanada [**auth0˙token˙best˙practices**], sinó també un pas necessari per a un compliment més estricte de la normativa. A causa de les limitacions de temps, vaig haver de deixar aquesta tasca registrada com una millora futura, tal com es detalla al Capítol 12.

UC-03: Tancament de Sessió El procés de tancament de sessió és una mesura de seguretat fonamental, dissenyada per ser directa i robusta. Assegura que la sessió de l'usuari finalitzi de forma definitiva mitjançant la invalidació dels tokens d'accés al client, prevenint així l'accés no autoritzat al compte si el dispositiu queda desatès i protegint contra el segrest de sessió. El flux comença a la interfície d'usuari, concretament quan l'usuari fa clic sobre el botó de logout situat al component ui-new/src/layouts/AppLayout/Header/index.tsx. Aquesta integració directa, on l'esdeveniment onClick invoca directament la funció logoutEndpoint, simplifica el component visual, que només s'ha de preocupar de renderitzar el botó i connectar-lo a l'acció corresponent.

La lògica central resideix al hook ui-new/src/hooks/userAuth.tsx, que proporciona la funció logoutEndpoint. Aquesta funció és la responsable de garantir una sortida neta i segura. Primer intenta notificar al backend mitjançant authService.logout. Independentment del resultat d'aquesta crida, ja que s'asumeix que el servidor rep la petició i fa les accions necessaries per el logout, el bloc finally s'encarrega d'eliminar

tota la informació de sessió del localStorage. Finalment, reinicia l'estat global de l'aplicació, desconnecta la sessió WebSocket activa i redirigeix forçosament l'usuari a la pàgina d'inici de sessió.

```
import - useAuth " from "../../../hooks/userAuth"

const Header = (-setAdminOpen": -setAdminOpen: (open: boolean) =¿ void")
    =¿ -
    const - logoutEndpoint, user " = useAuth();

    return (
        ¡div id="39" className="flex justify-end gap-2"¿
            ¡Tooltip side="left" content="Logout" triggerAsChild=-true"¿
                ¡Button variant="primary" className="p-2" onClick=-
    logoutEndpoint"¿
                    ¡RiLogoutBoxRLine className="h-4 w-4" /¿
                ¡/Button¿
            ¡/Tooltip¿
            -/* ... */"
        ¡/div¿
    )
"
```

LISTING 9.63: Fragment del component 'Header' amb el botó de logout

```
const logoutEndpoint = useCallback(async () =¿ -
  try -
    await authService.logout();
  " catch -
    // S'ignora l'error per assegurar que el logout en el client sempre
    ocorri
  " finally -
    localStorage.removeItem("accessToken");
    localStorage.removeItem("refreshToken");
    localStorage.removeItem("userId");
    localStorage.removeItem("username");

    setUser(null);
    setIsLogin(false);
    websocketService.disconnect();
    window.location.href = "/login";
  "
", []);
```

LISTING 9.64: Funció 'logoutEndpoint' al hook 'userAuth.tsx'

UC-04: Modificar Perfil & UC-05: Eliminar Compte La gestió del perfil d'usuari, que va des de la modificació de dades personals fins a l'eliminació permanent del compte, es consolida en una única narrativa de flux de dades per garantir una experiència d'usuari coherent i eficient. El procés s'inicia quan l'usuari accedeix a la configuració del seu perfil, una acció que renderitza el component intel·ligent ui-new/src/components/UserSettings/index.tsx. Aquest component actua com el nucli central de la funcionalitat, encapsulant tota la lògica necessària. En ser muntat, invoca immediatament el mètode userService.getCurrentUser(), definit a ui-new/src/lib/api.ts, per carregar les dades actuals de l'usuari i popular el formulari.

Un aspecte clau d'aquest component és la seva estratègia de validació en temps real, dissenyada per oferir feedback instantani sense sobrecarregar el sistema. Mentre que la validació de formats bàsics per a la majoria de camps s'implementa a

través del hook ui-new/src/hooks/useValidation.ts, la comprovació de la disponibilitat de camps únics com el nom d'usuari i el correu electrònic requereix una solució més sofisticada. Per evitar una gran quantitat de peticions a l'API amb cada pulsació de tecla, s'utilitza el hook ui-new/src/hooks/useDebounce.ts. Aquest retrasa l'execució de les funcions de validació (userService.checkUsernameExists i userService.checkEmailExists) fins que l'usuari ha deixat d'escriure durant un breu període. Aquesta tècnica millora dràsticament el rendiment i l'experiència d'usuari, evitant validacions innecessàries i oferint un indicador d'estat només quan és rellevant.

```
// ...
const debouncedUsername = useDebounce(userData.username, 500);
const debouncedEmail = useDebounce(userData.email, 500);

useEffect(() =¿ -
    const checkUsernameAvailability = async () =¿ -
        // No validar si el valor no ha canviat del original
        if (!initialUserData —— debouncedUsername === initialUserData.
    username) -
            // ...
            return;
        "
        // ...
        try -
            const isTaken = await userService.checkUsernameExists(
    debouncedUsername);
            // Actualitzar estat de la validació
        " catch -
            // Gestionar error de la comprovació
        "
    ";
    if (debouncedUsername && isOpen) checkUsernameAvailability();
", [debouncedUsername, initialUserData, errors.username, isOpen]);
// ... (codi similar per a la validació de l'email)
```

LISTING 9.65: Ús de 'useDebounce' per a la validació a 'UserSettings/index.tsx'

Nota: La implementació del hook useDebounce *es va adaptar de l'article Ïmplementing a Debounce Hook in React"[***debounce˙hook˙react****]. Aquesta millora va ser suggerida durant les sessions de proves beta per un tester amb experiència professional en React per tal d'optimitzar la interacció de l'usuari.*

Un cop les dades són validades, el component gestiona les dues accions principals. La funció handleUpdateProfile orquestra l'actualització del perfil, cridant userService.updateUserProfile i, si l'usuari ha introduït una nova contrasenya, userService.changePassword. Per altra banda, la funció handleDeleteAccount gestiona l'eliminació del compte. Per seguretat, primer mostra un diàleg de confirmació que requereix que l'usuari escrigui el seu nom d'usuari. Un cop confirmat, invoca userService.deleteAccount. Un detall crucial és que, després d'una eliminació exitosa, crida la funció logoutEndpoint del hook useAuth per netejar completament la sessió del client i redirigir-lo a la pàgina d'inici de sessió.

```
const handleUpdateProfile = async () =¿ -
    // ... (validació final)
    await userService.updateUserProfile(userData);
    if (newPassword && oldPassword) -
        await userService.changePassword(oldPassword, newPassword);
    "
```

```
    notifications.success("Profile updated");
    // ...
";

const handleDeleteAccount = async () =¿ -
    if (deleteConfirmation !== userData.username) -
        // ...
        return;
    "
    try -
        await userService.deleteAccount();
        notifications.success("Account deleted");
        logoutEndpoint(); // Funció del hook useAuth
    " catch (error) -
        notifications.error("Delete failed");
    "
";
```

LISTING 9.66: Orquestració de les crides a l'API a 'UserSettings/index.tsx'

Finalment, per a cada operació, ja sigui d'èxit o de fracàs, s'utilitzen notificacions tipus "toast"per proporcionar un feedback clar i immediat a l'usuari, tancant així el cicle d'interacció. El servei userService a api.ts abstrau les crides a l'API, proporcionant mètodes que mapegen directament als endpoints del backend per a una gestió de perfil clara i mantenible.

```
export const userService = -
    // ...
    updateUserProfile: async (userData: - /* ... */ ") =¿ -
        const response = await api.put('/users/profile', userData);
        return response.data;
    ",

    changePassword: async (oldPassword: string, newPassword: string) =¿
    -
        await api.put('/users/password', - oldPassword, newPassword ");
    ",

    deleteAccount: async () =¿ -
        await api.delete('/users');
    "
";
```

LISTING 9.67: Mètodes del 'userService' a 'lib/api.ts'

Gestió d'Administració

UC-06, UC-07, UC-15, UC-16: Panell d'Administració El panell d'administració representa una funcionalitat crítica per a la gestió d'usuaris del sistema, accessible exclusivament per a usuaris amb privilegis elevats.

L'accés al panell s'inicia a través d'un botó situat a la capçalera de l'aplicació, implementat a ui-new/src/layouts/AppLayout/Header/index.tsx. Aquest component utilitza el hook useAuth per verificar els permisos de l'usuari i renderitza el botó d'accés només si l'usuari té el rol ADMIN o SUPER_ADMIN:

```
-(user?.role === 'ADMIN' -- user?.role === 'SUPER'ADMIN') && (
    ¡Tooltip side="left" content="Admin Panel" triggerAsChild=-true"¿
```

```
3          ¡Button variant="primary" className="p-2" onClick=-() =¿
    setAdminOpen(true)"¿
4              ¡RiAdminLine className="h-4 w-4" /¿
5          ¡/Button¿
6      ¡/Tooltip¿
7  )"
```

LISTING 9.68: Renderitzat condicional del botó d'administració

Quan s'activa el botó, el component AppLayout actua com a orquestrador, gestionant la transició entre la vista principal de l'aplicació i el panell d'administració. Aquest canvi es realitza mitjançant un estat local adminOpen que renderitza condicionalment el component AdminDashboard en lloc de la interfície principal:

```
1      if(adminOpen) -
2      return (
3        ¡div className="flex h-screen bg-gray-100 relative"¿
4          ¡button
5            className="absolute top-6 right-8 z-50 px-4 py-2 bg-indigo-600
     text-white rounded-lg shadow hover:bg-indigo-700 transition-colors
    font-semibold"
6            onClick=-() =¿ setAdminOpen(false)"
7          ¿
8            Volver
9          ¡/button¿
10         ¡main className="flex-1 overflow-auto"¿
11           ¡AdminDashboard /¿
12         ¡/main¿
13       ¡/div¿
14     );
15   "
```

LISTING 9.69: Renderitzat condicional del panell d'administració a 'AppLayout'

El component AdminDashboard serveix com a punt central per a la gestió d'usuaris. En el seu muntatge, utilitza l'adminService per carregar la llista d'usuaris del sistema. La visualització d'aquesta informació es realitza mitjançant una taula interactiva que permet als administradors realitzar operacions de modificació i eliminació d'usuaris.

La gestió d'usuaris es realitza a través de dos diàlegs especialitzats: EditUserDialog i DeleteUserDialog. El primer és particularment complex, ja que implementa validacions en temps real i restriccions basades en rols. Per exemple, la capacitat de modificar el rol d'un usuari està estrictament controlada:

```
1      -isSuperAdmin && editForm.role !== 'SUPER'ADMIN' &&(
2          ¡¿
3        ¡div className="space-y-2"¿
4          ¡label className="text-sm font-medium text-gray-300"¿Role¡/label
    ¿
5          ¡select
6            className="w-full rounded-md border border-input bg-background
     px-3 py-2"
7            value=-editForm.role"
8            onChange=-e =¿ setEditForm(prev =¿ (- ...prev, role: e.target.
    value as 'USER' — 'ADMIN' — 'SUPER'ADMIN' "))"
9            disabled=-!currentUser —— currentUser && currentUser.role !==
    'SUPER'ADMIN' —— (currentUser.role === 'ADMIN' && user.role === '
    ADMIN')"
10         ¿
```

```
11            ¡option value="USER"¿User¡/option¿
12            ¡option value="ADMIN"¿Administrator¡/option¿
13          ¡/select¿
14        ¡/div¿
15          ¡/¿
16        )"
```

LISTING 9.70: Control d'accés per a la modificació de rols

Aquest fragment de codi il·lustra el control d'accés per a la modificació de rols. La lògica implementada assegura que només un usuari amb el rol de SUPER_ADMIN pugui canviar el rol d'un altre usuari. A més, s'estableix una restricció clau: el SUPER_ADMIN no pot modificar el seu propi rol. Aquesta decisió de disseny, al igual que impedir que es pugui eliminar el seu propi compte, té com a objectiu garantir que sempre hi hagi un, i només un, superadministrador com a màxim responsable de l'aplicació i les seves dades. La condició isSuperAdmin && editForm.role !== 'SUPER_ADMIN' del codi és la que materialitza aquesta regla, mostrant el selector de rol únicament quan es compleixen aquestes condicions.

L'execució de les operacions d'administració es gestiona a través de funcions específiques al component AdminDashboard. Aquestes funcions no només realitzen les operacions sol·licitades sinó que també gestionen la retroalimentació a l'usuari i l'actualització de l'estat de l'aplicació:

```
1     const handleSaveEdit = async (updatedUser: Partial¡AdminUser¿) =¿ -
2         if (!selectedUser) return;
3         try -
4             await adminService.updateUser(selectedUser.username,
    updatedUser);
5
6             notifications.success("User updated");
7             setLoaded(false);
8             await loadUsers();
9             setIsEditDialogOpen(false);
10        " catch -
11            notifications.error("Update failed");
12        "
13    ";
14
15    const handleConfirmDelete = async () =¿ -
16        if (!selectedUser) return;
17        try -
18            await adminService.deleteUser(selectedUser.username);
19            if(selectedUser.username === user?.username) -
20                logoutEndpoint();
21            "
22
23            notifications.success("User deleted");
24            setLoaded(false);
25            await loadUsers();
26            setIsDeleteDialogOpen(false);
27        " catch -
28            notifications.error("Delete failed");
29        "
30    ";
```

LISTING 9.71: Gestió d'operacions administratives

Un aspecte notable de la implementació és el tractament especial del cas d'autoeliminació (**UC-07**). Quan un administrador elimina el seu propi compte, el sistema

detecta aquesta situació i executa automàticament el procés de tancament de sessió mitjançant la funció logoutEndpoint.

Tota la comunicació amb el servidor es realitza a través de l'adminService, definit a ui-new/src/lib/api.ts, que proporciona una capa d'abstracció per a totes les operacions administratives. Aquest servei gestiona les crides a l'API REST del backend, mantenint la coherència i la seguretat en totes les operacions d'administració.

Com es pot veure, el botó "Delete"és interactiu quan es tenen permisos sobre l'usuari (**Figura 9.79**), però es mostra desactivat i no és funcional quan s'intenta eliminar un compte protegit, com el del propi SUPER_ADMIN (**Figura 9.80**). Aquesta restricció visual i funcional és una mesura de seguretat clau per garantir la integritat del sistema.

Finalment, abans de qualsevol eliminació, es mostra un diàleg de confirmació per prevenir accions accidentals (**Figura 9.82**), una mesura de seguretat afegida durant la implementació que no estava als esbossos inicials.

9.5.4 Gestió d'Arxius

El gestor d'arxius, implementat principalment al component ui-new/src/pages/FileManager/index.tsx, constitueix el nucli funcional de l'aplicació web.

UC-08: Crear o Pujar Arxius i Carpetes Aquesta funcionalitat permet als usuaris afegir nou contingut al seu espai de treball, ja sigui creant carpetes o pujant arxius i estructures de carpetes completes des del seu dispositiu.

La interacció de l'usuari s'inicia en el component ui-new/src/components/AddButton/index.tsx, que presenta un botó que, en ser premut, desplega un Popover amb tres opcions: "New folder", "Upload file" i "Upload folder". Aquest component implementa una tècnica comuna per a la pujada d'arxius: conté dos elements ¡input type="file» ocults que s'activen per codi mitjançant referències (useRef) quan l'usuari selecciona les opcions corresponents. Per a la pujada de carpetes, l'atribut no estàndard webkitdirectory és clau, ja que instrueix el navegador perquè permeti la selecció de carpetes completes.

L'opció "New folder" obre el diàleg modal ui-new/src/components/CreateNewFolderDialog/index.tsx, mentre que les opcions de pujada activen directament els respectius inputs ocults. Aquests components de la UI actuen com a mers disparadors: tota la lògica de negoci complexa resideix en el hook ui-new/src/hooks/useFileOperations.ts, que centralitza les operacions d'arxius. Les accions de l'usuari en la UI invoquen les funcions createFolder, uploadFile i uploadFolder proporcionades per aquest hook.

```
1  const folderInputRef = useRef¡HTMLInputElement¿(null);
2
3      const handleFolderUploadLocal = async (event: React.ChangeEvent¡
       HTMLInputElement¿) =¿ -
4      const files = event.target.files;
5          if (!files —— files.length ==== 0) -
6              return;
7          "
8          try -
9          uploadFolder(- files: Array.from(files), parentId:
       currentDirectory.id!.toString() ");
10         " catch (error) -
```

```
11                console.error('[AddButton] Error uploading folder:', error);
12    "
13            event.target.value = '';
14 ";
15
16    // JSX del component
17 ¡input
18    type="file"
19    ref=-folderInputRef"
20        className="hidden"
21        webkitdirectory=""
22        directory=""
23        multiple=-false"
24        onChange=-handleFolderUploadLocal"
25    /¿
26    ¡Button
27        variant="ghost"
28        className="w-full justify-start gap-2"
29        onClick=-() =¿ -
30            folderInputRef.current?.click();
31        ""
32    ¿
33        ¡RiFolderUploadLine className="h-4 w-4" /¿
34        Upload folder
35    ¡/Button¿
```

LISTING 9.72: Activació de la pujada de carpetes a 'AddButton/index.tsx'

El hook useFileOperations.ts constitueix una peça clau de l'arquitectura del client, utilitzant el hook useMutation de TanStack Query per a totes les operacions d'escriptura. Això permet una gestió d'estat molt avançada, incloent-hi les **actualitzacions optimistes**, que milloren dràsticament la percepció de velocitat de l'usuari. El flux d'una actualització optimista segueix un patró consistent: l'onMutate s'executa abans de la crida a l'API, actualitzant la memòria cache local de TanStack Query de manera immediata i fent que la interfície reflecteixi el canvi a l'instant. Es guarda una còpia de l'estat previ per poder-lo revertir en cas d'error. Si la crida a l'API falla, l'onError reverteix l'actualització optimista restaurant l'estat previ. Si la crida té èxit, l'onSuccess invalida la memòria cache per assegurar que les dades locals se sincronitzin amb les dades definitives provinents del servidor.

```
1  const createFolderMutation = useMutation(-
2      mutationFn: async (- name, parentId ": - name: string, parentId:
       string ") =¿ -
3        return fileService.createFolder(name, parentId);
4      ",
5      onMutate: async (- name, parentId ") =¿ -
6        // 1. Cancelar queries per evitar que sobreescriguin l'
       actualització optimista
7        await queryClient.cancelQueries(- queryKey: [QUERY'KEYS.
       CURRENT'DIRECTORY, parentId] ");
8        await queryClient.cancelQueries(- queryKey: [QUERY'KEYS.
       FOLDER'STRUCTURE] ");
9
10       // 2. Guardar l'estat previ
11       const previousDirectory = queryClient.getQueryData([QUERY'KEYS.
       CURRENT'DIRECTORY, parentId]) as FileItem;
12       const previousStructure = queryClient.getQueryData([QUERY'KEYS.
       FOLDER'STRUCTURE]);
13
14       // 3. Crear el nou objecte i actualitzar la UI de forma optimista
```

```
15        const newFolder: FileItem = -
16          id: `temp-$-Date.now()"`,
17          name: generateUniqueName(name, previousDirectory.subfolders?.map
    ((f: FileItem) =¿ f.name) —— [], name, true),
18          type: 'folder',
19          parent: parentId,
20          subfolders: []
21        ";
22
23        const directoryCopy = JSON.parse(JSON.stringify(previousDirectory)
    );
24        directoryCopy.subfolders = [...(directoryCopy.subfolders —— []),
    newFolder];
25        queryClient.setQueryData([QUERY'KEYS.CURRENT'DIRECTORY, parentId],
     directoryCopy);
26
27        return - previousDirectory, previousStructure ";
28      ",
29      onError: (err, variables, context) =¿ -
30        // Revertir en cas d'error
31        if (context?.previousDirectory) -
32          queryClient.setQueryData([QUERY'KEYS.CURRENT'DIRECTORY,
    variables.parentId], context.previousDirectory);
33        "
34        if (context?.previousStructure) -
35          queryClient.setQueryData([QUERY'KEYS.FOLDER'STRUCTURE], context.
    previousStructure);
36        "
37        notifications.error('Error creating folder');
38      ",
39      onSuccess: () =¿ -
40        notifications.success('Folder created');
41        refreshAll(); // Sincronitzar amb el servidor
42      "
43  ");
```

LISTING 9.73: Implementació d'una mutació optimista a 'useFileOperations.ts'

Les funcions de mutació dins del hook són les que finalment comuniquen amb el backend a través dels mètodes corresponents del fileService (definits a ui-new/src/lib/api.ts), que s'encarreguen de la comunicació HTTP. La pujada de carpetes presenta una complexitat addicional: la funcionalitat uploadFolderMutation ha de processar la llista de fitxers (FileList) que proporciona el navegador, reconstruir l'estructura de directoris en el client i enviar-la de forma recursiva al backend per recrear-la.

Aquesta implementació recursiva, si bé és robusta per a gestionar estructures de carpetes complexes, introdueix un important coll d'ampolla de rendiment. Cada fitxer dins de la carpeta es tradueix en una petició HTTP independent, la qual cosa pot generar una càrrega excessiva al backend en pujar directoris amb un gran volum d'elements. L'API actual no suporta operacions de pujada per lots (*batch uploads*), una limitació tècnica que impedeix una solució més eficient. L'optimització d'aquest flux s'ha identificat com un treball futur, tal com s'exposa al Capítol 12.

UC-09A/B/C: Operacions amb Arxius (Renomenar, Moure, Copiar) Aquest conjunt de funcionalitats cobreix les manipulacions més habituals sobre els elements del gestor d'arxius, com canviar el nom, moure'ls de lloc o duplicar-los.

El cervell que orquestra aquestes operacions resideix al ui-new/src/store/fileSelectionStore.ts, un store de Zustand que actua com a nucli centralitzat de l'estat. Aquest store gestiona un registre dels fitxers que l'usuari té seleccionats (selectedFiles) i, a més, actua com un porta-retalls virtual, emmagatzemant els fitxers copiats o tallats (clipboard) juntament amb l'estat de l'operació (isCut).

Per alimentar aquesta llista de fitxers seleccionats, la interfície ofereix múltiples mètodes d'interacció. El hook ui-new/src/hooks/useSelecto.ts, que integra la llibreria Selecto.js, permet a l'usuari dibuixar un quadre de selecció amb el ratolí per seleccionar diversos elements de forma intuïtiva. A més, s'han implementat dreceres de teclat per a una navegació eficient: el hook ui-new/src/hooks/useFileSelection.ts exposa la funció selectArrowKeys, que permet expandir o moure la selecció utilitzant les tecles de fletxa, mentre que ui-new/src/hooks/useFileShortcuts.ts gestiona la drecera Ctrl+A per seleccionar tots els elements del directori. Tots aquests mecanismes convergeixen en la modificació de l'estat selectedFiles dins del fileSelectionStore. Aquesta arquitectura desacobla la lògica de selecció de la d'acció, permetent que la resta de funcionalitats operin sobre la selecció actual sense necessitat de conèixer com s'ha generat.

Per determinar quins elements són seleccionables, el hook useSelecto.ts configura la llibreria Selecto.js perquè apunti a qualsevol element que tingui la classe CSS .file-item. Cada component File assigna aquesta classe al seu element principal, un component Card, fent-lo així un objectiu vàlid per a la selecció.

Aquesta connexió permet que l'estil de cada fitxer reaccioni dinàmicament a l'estat de la selecció. Per exemple, quan un fitxer és seleccionat o marcat per a ser tallat, el component File actualitza les seves classes CSS amb Tailwind per reflectir visualment aquest canvi. Aquesta reactivitat s'aconsegueix mitjançant hooks personalitzats que se subscriuen a l'store de Zustand.

```
1  const File = (- file ": FileProps) =¿ -
2    // ...
3    const isItemSelected = useIsItemSelected(file);
4    const isCutFile = useIsCutFile(file);
5    // ...
6    return (
7      ¡Card
8        className=-cx(
9          "file-item flex flex-col items-center justify-center p-2 rounded
     -lg cursor-pointer",
10         isItemSelected ? "bg-blue-100 border-blue-300" : "hover:bg-gray
     -100",
11         isCutFile && "opacity-50"
12       )"
13       // ...
14     ¿
15       -/* ... contingut del fitxer ... */"
16     ¡/Card¿
17   );
18 ";
```

LISTING 9.74: Estils dinàmics del component 'File' a 'index.tsx'

```
1  export function useIsItemSelected(file: FileItem) -
2    return useFileSelectionStore((state) =¿ state.selectedFilesIds.some(f
     =¿ f === file.id));
3  "
4
```

```
function isCutFile(file: FileItem, state: FileSelectionState) -
  const res = Array.from(state.clipboard).filter(f =¿ f.id === file.id);
  return res.length ¿ 0 && state.isCut;
"

export function useIsCutFile(file: FileItem) -
  return useFileSelectionStore((state) =¿ isCutFile(file, state));
"
```

LISTING 9.75: Implementació dels hooks 'useIsItemSelected' i 'useIsCutFile' a 'fileSelectionStore.ts'

El fragment de codi anterior mostra com els booleans isItemSelected i isCutFile s'utilitzen per aplicar classes condicionalment. Aquests booleans no són estats locals del component, sinó el resultat de dos hooks personalitzats, useIsItemSelected i useIsCutFile. Aquests hooks implementen un patró de subscripció selectiva a l'store de Zustand (fileSelectionStore).

Aquest mecanisme és clau per a l'eficiència de la interfície. Quan l'usuari selecciona un fitxer, l'estat selectedFiles de l'store canvia. Zustand notifica només aquells components que estan subscrits a aquesta part específica de l'estat. En aquest cas, només els components File afectats (el que s'acaba de seleccionar i el que s'acaba de deseleccionar) rebran el nou valor del hook isItemSelected i es tornaran a renderitzar per actualitzar el seu estil. La resta de fitxers a la graella no es veuen afectats i no es renderitzen de nou, evitant així càlculs innecessaris i mantenint una experiència d'usuari fluida, fins i tot en directoris amb centenars d'elements.

Les accions de l'usuari s'inicien des de diversos punts d'entrada que utilitzen aquest estat centralitzat. El menú contextual (ui-new/src/components/FileManagerContextMenu/index.tsx) proporciona les opcions visuals. Paral·lelament, el hook ui-new/src/hooks/useFileShortcuts.ts afegeix un *listener* global per a dreceres com Ctrl+C o Ctrl+X, que criden les funcions copyFiles o cutFiles de l'store. Finalment, el component ui-new/src/pages/FileManager/index.tsx utilitza dnd-kit per a la funcionalitat d'arrossegar i deixar anar. Cal destacar que, tot i que la lògica permet operacions amb múltiples fitxers mitjançant arrossegar i deixar anar, la representació visual actualment només mostra un únic element sent arrossegat. Aquesta és una millora pendent que s'exposa al Capítol 12.

Independentment del disparador utilitzat, l'execució final de la lògica de negoci es delega sempre a les mutacions definides al hook ui-new/src/hooks/useFileOperations.ts. Per al **renombrament (UC-09A)**, l'opció del menú contextual obre el diàleg ui-new/src/components/Fil En confirmar el nou nom, una funció de callback al FileManager/index.tsx (handleRename) invoca la mutació updateItem del hook. Per al **moviment (UC-09B)**, tant l'acció d'arrossegar i deixar anar com la confirmació des del diàleg de selecció de carpeta acaben cridant a la mutació moveItem. Per a les operacions de **copiar i enganxar (UC-09C)**, la drecera Ctrl+V al useFileShortcuts.ts llegeix el contingut del fileSelectionStore i invoca la mutació pasteFiles. Aquesta única mutació gestiona tant la còpia (fileService.copyElement) com el tallat (fileService.moveElement), basant-se en el valor del booleà isCut de l'store.

```
const handleDragEnd = useCallback(async (event: DragEndEvent) =¿ -
    const - active, over " = event;
    setActiveDragItem(null);

    if (!over —— active.id === over.id) -
        return;
```

```
7         "
8       if(!active.data.current -- !over.data.current) -
9         return;
10      "
11
12      const draggedItem = active.data.current.file as FileItem;
13      const dropTarget = over.data.current.file as FileItem;
14      if (dropTarget && dropTarget.id) -
15          try -
16              if (draggedItem.id !== dropTarget.id && draggedItem.id !==
    dropTarget.parent && dropTarget.type === 'folder') -
17                  const itemsToMove = selectedFiles.length ¿ 0 &&
    selectedFiles.some(f =¿ f === draggedItem)
18                  ? selectedFiles
19                  : [draggedItem];
20              await fileOperations.moveItem(-
21                  items: itemsToMove,
22                  toFolderId: dropTarget.id.toString(),
23                      fromFolderId: fileOperations.currentDirectory.id --
    'root'
24              ");
25              notifications.success('Items moved successfully');
26              "
27          " catch -
28              notifications.error('Could not move the item');
29          "
30      "
31  ", [selectedFiles, fileOperations, notifications]);
```

LISTING 9.76: Gestió del Drag & Drop a 'FileManager/index.tsx'

```
1  useEffect(() =¿ -
2    const handleKeyDown = async (event: KeyboardEvent) =¿ -
3      if (event.ctrlKey -- event.metaKey) -
4        switch (event.key.toLowerCase()) -
5          case 'c':
6            event.preventDefault();
7            if(isInRoot -- (isInShared && currentDirectory.shared &&
    currentDirectory.accessLevel !== 'READ')) -
8            copyFiles(currentDirectory.id!.toString());
9            " else -
10             notifications.error("You don't have permission to copy files
    ");
11           "
12           break;
13
14         case 'x':
15           event.preventDefault();
16           if(isInRoot -- (isInShared && currentDirectory.shared &&
    currentDirectory.accessLevel !== 'read')) -
17           cutFiles(currentDirectory.id!.toString());
18           " else -
19             notifications.error("You don't have permission to cut files
    ");
20           "
21           break;
22
23         case 'v':
24           event.preventDefault();
25           if(isInRoot -- (isInShared && currentDirectory.accessLevel !==
     'read')) -
26           const items = Array.from(clipboard);
27           if(items.length ¿ 0) -
```

```
28                pasteFiles(-items: items, targetFolderId: currentDirectory
    .id!.toString(), prevParentId: clipboardParentId, isCut: isCut");
29              "
30              if(isCut) -
31                setClipboardFiles(new Set(), undefined);
32              "
33            " else -
34              notifications.error("You don't have permission to paste
    files");
35            "
36            break;
37        "
38      "
39    ";
40
41    window.addEventListener('keydown', handleKeyDown);
42    return () =¿ window.removeEventListener('keydown', handleKeyDown);
43  ", [selectedFiles, currentDirectory, clipboard, isCut, clipboardParentId
    , /* ... */]);
```

LISTING 9.77: Gestió de dreceres de teclat a 'useFileShortcuts.ts'

Totes aquestes mutacions (updateItem, moveItem, pasteFiles) segueixen el patró d'**actualització optimista** implementat amb TanStack Query, proporcionant una resposta visual immediata a l'usuari mentre la comunicació amb el backend es processa en segon pla.

UC-10: Eliminar Arxiu (Moure a la Paperera) Quan eliminem un arxiu des de la vista principal del gestor, no l'estem esborrant per sempre, sinó que el movem a la paperera de reciclatge, com si fos un pas intermedi.

Podem començar a eliminar un arxiu de dues maneres: fent clic amb el botó dret per obrir el menú contextual a ui-new/src/components/FileManagerContextMenu/index.tsx o simplement prement la tecla Delete. El hook ui-new/src/hooks/useFileShortcuts.ts s'encarrega de capturar quan premem Delete i, igual que el menú contextual, crida la funció deleteItem del hook ui-new/src/hooks/useFileOperations.ts. Això fa que, independentment de com decidim eliminar l'arxiu, l'experiència sigui la mateixa.

El que fa que aquesta funcionalitat sigui interessant és la lògica dual de la mutació deleteItemMutation. El que passa quan eliminem un arxiu depèn totalment de la secció on estem, informació que ens proporciona el ui-new/src/store/fileContextStore.ts. Aquest store ens diu on som (root, shared o trash), i això permet que la mateixa acció (prémer Delete o fer clic a "Eliminar") faci coses diferents segons el context.

```
1 " else if (event.key === 'Delete') -
2   event.preventDefault();
3   if (fileContext.section === 'shared') -
4     revokeAccess(-items: selectedFiles")
5   " else -
6     deleteItem(-items: selectedFiles, section: fileContext.section");
7   "
8   clearSelection();
9 "
```

LISTING 9.78: Gestió de la tecla Delete a 'useFileShortcuts.ts'

Dins de la funció mutationFn de la mutació deleteItemMutation, un simple condicional decideix què fer. Si estem a la secció 'root', es crida a fileService.deleteElement, que mou l'element a la paperera al backend. Però si estem a la secció 'trash', es crida

a trashService.deleteItem, que esborra l'element de manera permanent. Aquest disseny ens permet utilitzar la mateixa interfície d'usuari per a dues operacions molt diferents, fent que la lògica sigui més senzilla als components de la interfície mentre mantenim la complexitat al nivell de servei.

```
const deleteItemMutation = useMutation(-
  mutationFn: async (- items, section ": - items: FileItem[], section: '
    root' — 'trash' — 'shared' ") =¿ -
    try -
      if(section === 'root') -
        return Promise.all(items.map(item =¿ fileService.deleteElement(
    item.id!.toString())));
      " else if(section === 'trash') -
        return Promise.all(items.map(item =¿ trashService.deleteItem(
    item.id!.toString())));
      " else -
        notifications.error('Cannot delete items from this section');
        return;
      "
    " catch (error) -
      console.error('Error deleting items:', error);
      notifications.error('Error deleting items');
      return;
    "
  ",
  onSuccess: (', variables) =¿ -
    notifications.success(
      variables.section === 'root'
        ? 'Items moved to trash correctly'
        : 'Items deleted permanently'
    );
    refreshAll();
  "
  // ... onMutate i onError per a l'actualització optimista
");
```

LISTING 9.79: Lògica dual de la mutació d'eliminació a 'useFileOperations.ts'

Igual que les altres mutacions del sistema, deleteItemMutation utilitza el patró d'actualització optimista de TanStack Query. Quan eliminem un element, aquest desapareix de la interfície immediatament gràcies a l'onMutate, que actualitza la cache local, mentre que l'operació real es fa en segon pla. Les notificacions d'èxit ens donen un feedback específic segons el tipus d'operació: Ïtems moved to trash correctly"per al moviment a la paperera o Ïtems deleted permanently"per a l'esborrat definitiu.

9.5.5 Compartició d'Arxius

UC-13, UC-13A, UC-13B: Compartir, Deixar de Compartir i Veure Compartits Aquesta funcionalitat permet als usuaris compartir els seus arxius i carpetes amb altres, gestionar els permisos i veure els elements que altres han compartit amb ells.

La navegació entre les diferents seccions del gestor d'arxius (root, shared, trash) es gestiona a través d'un canvi de context en lloc d'un canvi de ruta. L'usuari inicia aquesta navegació fent clic a les icones corresponents a la barra lateral, implementada al component ui-new/src/components/FileDirectorySidebar/index.tsx. Aquesta acció invoca la funció handleNavigateToSection, que actualitza l'estat global a través del ui-new/src/store/fileContextStore.ts, establint la secció activa. Aquest canvi de

context fa que el hook useFileOperations obtingui les dades del directori arrel corresponent a la secció seleccionada, poblant la vista principal amb els fitxers i carpetes adequats.

```
const handleNavigateToSection = useCallback(async (section: 'root' — '
    shared' — 'trash', e: React.MouseEvent) =¿ -
  e.preventDefault();
  e.stopPropagation();

  try -
    let folder: FileItem;
    switch (section) -
      case 'root':
        folder = await fileService.getRootFolder();
        break;
      case 'shared':
        folder = await sharingService.getSharedRootFolder();
        folder.id = "shared";
        break;
      case 'trash':
        folder = await trashService.getTrashRootFolder();
        folder.id = "trash";
        break;
    "
    await setCurrentDirectory(folder);
    fileContext.setSection(section);
  " catch -
    notifications.error('Loading error');
  "
", [fileContext, setCurrentDirectory, notifications]);
```

LISTING 9.80: Navegació entre seccions a 'FileDirectorySidebar/index.tsx'

Un cop dins d'una secció, l'usuari pot navegar per l'arbre de directoris fent clic als elements de la barra lateral (ui-new/src/components/FileDirectorySidebarItem/index.tsx) o fent doble clic sobre una carpeta a la vista principal (ui-new/src/components/File/index.tsx). Ambdues accions invoquen la funció setCurrentDirectory del hook useFileOperations, que carrega el contingut de la carpeta seleccionada i actualitza la vista.

La funció setCurrentDirectory és fonamental per a la gestió de l'estat de la navegació. Aquesta funció actualitza manualment la memòria cache de TanStack Query mitjançant queryClient.setQueryData, establint la carpeta seleccionada com el nou directori actual. Aquest procés actualitza l'estat global a Zustand a través de setStoreCurrentDirectory i persisteix l'ID del directori a la sessionStorage per mantenir l'estat entre recàrregues de la pàgina.

Aquesta acció desencadena l'execució de la query currentDirectoryQuery, que utilitza el hook useQuery per obtenir les dades completes del directori des del backend. La clau d'aquesta query (queryKey) és un array que inclou l'identificador del directori actual, assegurant que cada directori tingui la seva pròpia entrada a la memòria cache. La configuració staleTime de 5 minuts indica a TanStack Query que consideri les dades a la memòria cache com a "fresques"durant aquest període, evitant crides innecessàries a l'API si l'usuari torna a un directori visitat recentment.

```
const setCurrentDirectory = async (folder: FileItem) =¿ -
  if (folder.id) -
    const newId = folder.id.toString();
    setStoreCurrentDirectory(folder);
```

```
    safeSessionStorage.setItem(STORAGE_KEYS.CURRENT_DIRECTORY_ID, newId);
  "

  await queryClient.setQueryData([QUERY_KEYS.CURRENT_DIRECTORY, folder.id], folder);
  setStoreCurrentDirectory(folder);
";

const currentDirectoryQuery = useQuery(-
  queryKey: [QUERY_KEYS.CURRENT_DIRECTORY, currentDirectory.id],
  queryFn: () =¿ -
    if(section === 'trash') -
      return fileService.getFolderById(currentDirectory.id, true);
    " else -
      return fileService.getFolderById(currentDirectory.id, false);
    "
  ",
  staleTime: 1000 * 60 * 5,
  enabled: !!localStorage.getItem('accessToken') && !!currentDirectory.id,
");
```

LISTING 9.81: Gestió del directori actual amb 'useQuery' a 'useFileOperations.ts'

Aquest mecanisme de memòria cache és essencial per a una experiència de navegació fluida. Quan l'usuari navega a un directori ja visitat, les dades es carreguen instantàniament des de la memòria cache, eliminant la latència de la xarxa. La dada només es torna a demanar al servidor si ha passat el staleTime o si la memòria cache s'invalida manualment després d'una operació (com crear o eliminar un fitxer), assegurant un equilibri òptim entre rendiment i consistència de les dades.

```
const handleDoubleClick = async () =¿ -
  if (file.type === 'folder') -
    fileOperations.setCurrentDirectory(file);
  " else -
    // ... lògica per descarregar fitxers
  "
";
```

LISTING 9.82: Navegació per doble clic a 'File/index.tsx'

L'acció de compartir s'inicia des del menú contextual, que obre el component ui-new/src/components/File/S el centre neuràlgic de la gestió de permisos. Cal destacar que aquesta funcionalitat només està disponible per a fitxers i carpetes propis de l'usuari, és a dir, quan es troba a la secció root. En obrir-se el diàleg, aquest utilitza el hook ui-new/src/hooks/useShareManager.ts per carregar l'estat actual dels permisos per a tots els fitxers seleccionats, una informació que es desa per a una comparació posterior.

Una de les optimitzacions clau d'aquesta funcionalitat rau en la lògica de la funció share, passada com a callback onShare des de FileManager/index.tsx. En lloc d'enviar simplement l'estat final dels permisos, aquesta funció realitza una operació de "diferència"entre l'estat original (guardat en obrir el diàleg) i l'estat final (quan l'usuari fa clic a "Share"). El codi itera sobre l'estat final per identificar nous permisos o modificacions, i sobre l'estat original per identificar permisos revocats. Això genera una llista precisa de les accions necessàries (shareWithUser, handleUpdateAccess, handleUnshareToUser), minimitzant el nombre de crides a l'API.

```
const share = useCallback(async (userAccess: -username: string, fileId:
    string, accessType: 'READ' — 'WRITE' "[], originalSharedAccess: Map¡
    string, -username: string, fileId: string, accessType: 'READ' — '
    WRITE' "[]¿) =¿ -
  const promises = [];
  if(originalSharedAccess.size ¿ 0) -
    for (const access of userAccess) -
      if(!originalSharedAccess.has(access.username) —— !
    originalSharedAccess.get(access.username)!.some(a =¿ a.fileId ===
    access.fileId)) -
        promises.push(shareManager.shareWithUser(access.fileId, access.
    username, access.accessType));
      " else if(originalSharedAccess.get(access.username)!.some(a =¿ a.
    fileId === access.fileId && a.accessType !== access.accessType)) -
        promises.push(shareManager.handleUpdateAccess(access.fileId,
    access.username, access.accessType));
      "
    "
    for (const user of originalSharedAccess.keys()) -
      if(!userAccess.some(a =¿ a.username === user)) -
        originalSharedAccess.get(user)!.forEach(a =¿ -
          promises.push(shareManager.handleUnshareToUser(a.fileId, user)
    );
        ");
      " else -
        originalSharedAccess.get(user)!.forEach(a =¿ -
          if(!userAccess.some(access =¿ access.username === user &&
    access.fileId === a.fileId)) -
            promises.push(shareManager.handleUnshareToUser(a.fileId,
    user));
          "
        ");
      "
    "
    await Promise.all(promises);
  " else -
    for (const access of userAccess) -
      promises.push(shareManager.shareWithUser(access.fileId, access.
    username, access.accessType));
    "
    await Promise.all(promises);
  "
", [shareManager]);
```

LISTING 9.83: Lògica de comparació de permisos a 'FileManager/index.tsx'

Aquesta optimització va ser una millora directa basada en el feedback d'un *beta tester*, que va assenyalar que compartir múltiples fitxers amb múltiples usuaris era un procés lent i afarragós. La funció de comparació utilitza el hook useShareManager com a intermediari, que conté funcions més granulars que són les que finalment criden als mètodes corresponents del sharingService.

```
const shareWithUser = useCallback(async (fileId: string, username:
    string, accessType: 'READ' — 'WRITE' = 'READ') =¿ -
  if (!username) -
    console.error("Please enter a username");
    return false;
  "
  try -
    await sharingService.shareFile(fileId, username, accessType);
    notifications.success('Item shared successfully');
```

```
    console.log(`File ${fileId} shared with ${username} with access type
     ${accessType}`);
    return true;
  } catch (error) {
    console.error('Error sharing file:', error);
    notifications.error('Could not share item');
    return false;
  }
}, []);
```

LISTING 9.84: Funció per compartir amb un usuari a 'useShareManager.ts'

A més de mostrar els fitxers compartits, la secció shared ofereix opcions contextuals diferents. El menú contextual ui-new/src/components/FileManagerContextMenu/index.tsx adapta les seves opcions a la secció activa, mostrant l'opció "Stop seeing"quan l'usuari es troba a la secció de compartits. Aquesta acció invoca la funció revokeAccess del hook useFileOperations, que s'encarrega de revocar el permís de l'usuari sobre el fitxer seleccionat, eliminant-lo de la seva vista de compartits.

Aquesta adaptabilitat s'aconsegueix mitjançant una sèrie de renderitzats condicionals dins del component, que comproven la secció activa (isInRoot, isInTrash, isInShared) i si s'ha seleccionat un fitxer per mostrar només les accions pertinents.

```
const handleUnshare = useCallback(async () => {
  if (!file) return;

  await revokeAccess({items: selectedFiles});
  onClose();
}, [file, selectedFiles, onClose, revokeAccess]);
// ... //
<div className="min-w-[220px] p-1">
  {/* Opcions quan NO hi ha cap fitxer seleccionat (Crear, Pujar,
    Enganxar...) */}
  {!file && (isInRoot || (isInShared && currentDirectory.accessLevel ===
     'WRITE')) ? (
    <>
      <div onClick={() => setIsCreateFolderOpen(true)}>
        <RiFolderLine />
        <span>New folder</span>
      </div>
      {/* ... Altres opcions de creació i enganxar ... */}
    </>
  ) : (
    <>
      {/* Opcions quan SÍ hi ha un fitxer seleccionat */}

      {/* Opcions per a la secció ROOT */}
      {isInRoot && (
        <>
          <div onClick={handleCopy}><span>Copy</span></div>
          <div onClick={handleCut}><span>Cut</span></div>
          <div onClick={() => setOpenShareDialog(true)}><span>Share</
    span></div>
          <div onClick={handleDelete}><span>Delete</span></div>
        </>
      )}

      {/* Opcions per a la secció TRASH */}
      {isInTrash && (
        <>
```

```
35            ¡div onClick=-handleRestore"¿¡span¿Restore¡/span¿¡/div¿
36            ¡div onClick=-handleDelete"¿¡span¿Delete permanently¡/span¿¡/
    div¿
37          ¡/¿
38        )"
39
40        -/* Opcions per a la secció SHARED (renderitzades per una funció
    auxiliar) */"
41        -isInShared && renderSharedOptions()"
42
43        -/* Opció comuna per veure detalls si només hi ha un fitxer
    seleccionat */"
44        -selectedFiles.length === 1 && (
45          ¡div onClick=-handleDetails"¿¡span¿Details¡/span¿¡/div¿
46        )"
47      ¡/¿
48    )"
49  ¡/div¿
```

LISTING 9.85: Renderitzat condicional de les opcions del menú a 'FileManagerContextMenu/index.tsx'

Cal destacar que, de manera similar a l'operació de pujada de carpetes, el procés de compartir múltiples fitxers amb múltiples usuaris no està optimitzat per a grans volums de dades. Tot i que és menys probable que un usuari modifiqui milers de permisos manualment a través de la interfície, l'arquitectura actual realitza una crida a l'API per a cada modificació individual de permís. L'optimització d'aquest flux mitjançant operacions per lots (*batch operations*) es considera una de les millores futures, tal com es detalla al Capítol 12.

9.6 Implementació del client de escriptori

9.6.1 Introducció

El client d'escriptori s'ha dissenyat com un component de servei, el propòsit principal del qual és la sincronització automàtica i contínua de fitxers entre l'entorn local de l'usuari i el servidor central. A diferència del client web, orientat a la interacció directa, aquesta aplicació opera de manera predominant en segon pla, minimitzant la intervenció de l'usuari i funcionant com un agent de sincronització desatès que s'integra a la safata del sistema.

L'arquitectura es basa en el framework **Tauri**, una elecció tecnològica que permet una clara separació entre el nucli de lògica de negoci i la interfície d'usuari. El backend de l'aplicació s'ha implementat íntegrament en **Rust** per aprofitar-ne el rendiment, la seguretat en la gestió de memòria i les capacitats de concurrència, aspectes crítics per a operacions intensives com la monitorització del sistema d'arxius i la gestió de processos de transferència de dades. Per a la interfície d'usuari, s'ha optat per **Svelte 5**, un compilador que genera codi JavaScript altament optimitzat, resultant en una aplicació lleugera i amb uns requisits de recursos mínims. Les vistes es presenten a l'usuari únicament en situacions necessàries, com la configuració inicial o la consulta de l'estat.

El sistema resultant és una aplicació híbrida que executa la lògica de sincronització com un procés natiu mentre ofereix una experiència de configuració i monitorització a través d'una interfície web renderitzada localment. Aquest capítol analitza el disseny d'aquesta solució, detallant l'estructura de les seves finestres i el funcionament intern del motor de sincronització basat en esdeveniments.

9.6.2 Disseny Visual i Gestió de Finestres

L'aplicació d'escriptori està dissenyada per guiar l'usuari a través d'un flux lògic, presentant diferents finestres segons l'estat de configuració i autenticació. La gestió d'aquestes finestres es coordina des del mòdul windows.rs, que s'encarrega d'orquestrar quina vista mostrar en cada moment, assegurant una transició fluida i contextual.

Flux Inicial: Configuració i Autenticació

Tal com vaig esbossar en la fase de disseny, el primer contacte de l'usuari amb l'aplicació és un procés de configuració guiat. En executar-la per primera vegada, el sistema comprova si ja està configurada. Si no ho està, es presenta la finestra de configuració inicial.

FIGURA 9.83: Finestra de configuració inicial.

FIGURA 9.84: Diàleg natiu per a la selecció de carpeta.

Aquesta finestra (**Figura 9.83**) és una implementació fidel del concepte presentat a la **Figura 8.40**. El seu propòsit és recollir les dues dades essencials per al funcionament: l'URL del servidor i la carpeta local a sincronitzar. Per a la selecció de la carpeta, es va seguir la intenció del disseny original d'invocar el diàleg natiu del sistema operatiu (**Figura 9.84**), la qual cosa ofereix una experiència més familiar i intuïtiva per a l'usuari.

Un cop l'usuari omple i envia el formulari, s'invoca el comandament save_initial_config. Aquesta funció de Rust primer valida que el servidor sigui accessible i que la carpeta seleccionada existeixi. Si les comprovacions són correctes, desa les dades en un fitxer config.json, tanca la finestra de configuració i obre la finestra de login per continuar amb el flux.

```
#[tauri::command]
async fn save_initial_config(
    app: AppHandle,
    folder_path: String,
    server_url: String,
) -¿ Result¡(), HashMap¡String, String¿¿ -
    // Validar que el servidor és accessible
    let resp = client
        .get(format!("-server_url"/actuator/health"))
        .send()
        .await;

    // Validar que la carpeta existeix
    let folder_exists = Path::new(&folder_path).exists();

    if resp.is_err() —— !folder_exists -
        // Retornar errors a la interfície si alguna validació falla
        return Err(error_map);
    "

    // Desar la configuració al fitxer JSON
    config.folder_path = folder_path;
    config.server_url = server_url;
    config.is_configured = true;
```

```
25      std::fs::write(config_path, serde_json::to_string_pretty(&config).
        unwrap()).unwrap();
26
27      // Tancar finestra actual i obrir la de login
28      window.close().unwrap();
29      windows::open_login_window(app.clone());
30      Ok(())
31  "
```

LISTING 9.86: Comandament per desar la configuració inicial a main.rs

És important mencionar una consideració tècnica sobre la validació del servidor. Com es pot veure al codi, actualment es fa una petició a l'endpoint '/actuator/-health' per comprovar si el servidor està actiu. Tot i que funcional, aquest és un endpoint estàndard de Spring Boot. Això introdueix un petit risc: si l'usuari, per error, apunta a una URL on hi ha una altra aplicació Spring Boot funcionant, el client podria interpretar-ho com una connexió vàlida. Per a solucionar això, queda plantejat com a treball futur la creació d'un endpoint personalitzat i únic al servidor, per exemple '/api/v1/ping', que retorni una cadena de text específica. D'aquesta manera, el client podria verificar no només que hi ha un servei actiu, sinó que es tracta, inequívocament, del servidor correcte del projecte.

FIGURA 9.85: Finestra d'inici de sessió del client d'escriptori.

Un cop desada aquesta configuració, l'aplicació presenta la finestra d'inici de sessió (**Figura 9.85**), que correspon directament amb l'esbós de la **Figura 8.41**. Igual que la seva contrapart web, el disseny és minimalista i funcional, centrat exclusivament a facilitar l'autenticació de l'usuari. El codi d'aquesta funcionalitat envia les credencials al servidor i, si la resposta és correcta, extreu els tokens JWT de les capçaleres de la resposta, els desa a la configuració local i inicia el motor de sincronització.

```
1  #[tauri::command]
```

```
2 async fn login(app: AppHandle, username: String, password: String) -¿
    Result¡(), String¿ -
3     let server = CONFIG.lock().unwrap().server˙url.to˙owned();
4     let client = Client::new();
5     let resp = client
6         .post(format!("-server"/users/auth/login"))
7         .json(&json!(-"username": username, "password": password"))
8         .send()
9         .await
10        .unwrap();
11
12    if !resp.status().is˙success() -
13        return Err("Username or password incorrect".to˙string());
14    "
15
16    // Extreure tokens de les capçaleres de la resposta
17    let token = resp.headers().get("authorization").unwrap().to˙str().
    unwrap();
18    let refresh˙token = resp.headers().get("x-refresh-token").unwrap().
    to˙str().unwrap();
19
20    // Desar els tokens i dades d'usuari a la configuració
21    let mut config = CONFIG.lock().unwrap();
22    config.username.replace(username);
23    config.token.replace(Token - value: token.to˙string(), .. ");
24    config.refresh˙token.replace(Token - value: refresh˙token.to˙string
    (), .. ");
25
26    // Iniciar serveis en segon pla i obrir la finestra principal
27    login˙window.close().unwrap();
28    tokio::spawn(token::watch˙tokens(app.clone()));
29    windows::open˙main˙window(app);
30    Ok(())
31 "
```

LISTING 9.87: Comandament per a l'inici de sessió a main.rs

Interfície Principal i Menú del Sistema

Un cop l'usuari està autenticat, el sistema fa la transició a l'estat operacional normal: les finestres de configuració es tanquen i s'inicia la finestra principal (**Figura 9.86**), alhora que apareix la icona de l'aplicació a la safata del sistema.

FIGURA 9.86: Finestra principal de transferències.

La implementació d'aquesta finestra principal és una traducció directa del disseny conceptual mostrat a la **Figura 8.42**. Es va mantenir la disposició original, ja que complia eficaçment el seu propòsit de ser un monitor d'activitat clar i concís. La interfície mostra dues llistes per a les transferències (en curs i completades) i inclou a la capçalera els quatre botons d'accés ràpid que s'havien previst. La idea original de que aquesta finestra es tanqués automàticament en perdre el focus es va mantenir. No obstant això, durant les proves vaig descobrir un bug documentat a la llibreria Tauri per a entorns GNOME a Linux [**tauri˙focus˙bug**]. Aquest error impedeix que la finestra obtingui el focus automàticament en mostrar-se, la qual cosa obliga a l'usuari a fer un clic inicial per poder interactuar-hi i perquè, posteriorment, el mecanisme de tancament en perdre el focus funcioni com s'esperava.

FIGURA 9.87: Menú de la safata del sistema.

Paral·lelament, el menú de la safata del sistema (**Figura 9.87**) també es va implementar seguint fidelment el disseny de la **Figura 8.44**. Aquest menú proporciona un punt d'accés persistent a les funcionalitats essencials, assegurant que l'aplicació sigui controlable fins i tot amb la finestra principal tancada.

Finestra de Configuració

L'aplicació ofereix una finestra de configuració detallada, accessible tant des de la finestra principal com des del menú de la safata del sistema.

FIGURA 9.88: Finestra de configuració amb les seves tres pestanyes.

Aquesta finestra (**Figura 9.88**), implementada a config/+page.svelte, utilitza un sistema de pestanyes per organitzar les diferents opcions:

- **User Authentication**: Permet a l'usuari veure el seu nom d'usuari i tancar la sessió actual.
- **Folders**: Mostra la ruta de la carpeta local sincronitzada i permet canviar-la.
- **Connection**: Mostra l'URL del servidor i permet modificar-la.

La implementació final de la finestra de configuració (**Figura 9.88**) segueix l'estructura de pestanyes que ja s'havia esbossat al disseny original (**Figura 8.43**), que separava les opcions en grups lògics.

La principal decisió de disseny que vaig prendre durant la implementació es troba a la pestanya d'autenticació. En lloc de crear una funcionalitat per canviar les credencials directament des d'aquí, vaig optar per una solució més simple i segura: un botó de l·logout". Aquesta decisió força a l'usuari a passar de nou per la finestra de login principal per canviar de compte, la qual cosa reutilitza un flux de treball ja validat i redueix la complexitat.

La comunicació entre la interfície de Svelte i el nucli de Rust es realitza a través de la funció invoke de l'API de Tauri, que permet cridar a funcions de Rust exposades com a comandaments. Per exemple, quan l'usuari modifica l'URL del servidor i prem el botó "Save", s'executa el següent codi del component de Svelte.

```
¡script lang="ts"¿
  import - config " from '$lib/store.svelte';
  import - update˙config " from '$lib/utils';
  let error = $state('');
¡/script¿

¡p class="mb-3 ..." ¿Server URL¡/p¿
¡div class="w-ful flex gap-2"¿
  ¡input
    bind:value=-config.server˙url"
    class="..."
    placeholder="Server URL ex. http://127.0.0.1:8080"
  /¿
  ¡button
    onclick=-() =¿ -
      error = '';
      update˙config().catch((e) =¿ (error = e.server));
    ""
    class="..."
  ¿
    Save
  ¡/button¿
¡/div¿
```

LISTING 9.88: Exemple de crida al backend des de Svelte a Connection.svelte

En aquest fragment, l'esdeveniment onclick del botó crida a la funció update_config(). Aquesta funció, definida a lib/utils.ts, és una abstracció que internament utilitza invoke('update_config', { config, restart: true }). Aquesta crida executa la funció update_config de Rust, passant-li l'objecte de configuració actualitzat des de la interfície perquè sigui validat i desat de forma persistent.

Cal destacar que aquesta finestra representa una primera versió. Per limitacions de temps, no vaig poder implementar totes les opcions de configuració avançada

que tenia previstes per donar a l'usuari un control més granular. Algunes de les funcionalitats que van quedar pendents són, per exemple, la capacitat de limitar el nombre de pujades i baixades concurrents, establir límits d'ample de banda per a la sincronització, o configurar quines notificacions es volen rebre, o un sistema de prioritats per decidir si es queden els canvis desde servidor o local en cas de conflicte.

Aquestes millores, juntament amb la funcionalitat de sincronitzar els fitxers de la secció Çompartits amb mi"—una característica clau que tampoc es va poder completar—, queden registrades com a treballs futurs per a properes versions de l'aplicació, tal com s'explicara en el **Capítol 12**.

9.6.3 Motor de Sincronització: El Nucli Reactiu

El cor del client d'escriptori és el seu motor de sincronització, implementat al mòdul synchronizer de Rust. Aquest component opera en segon pla i la seva responsabilitat és mantenir la carpeta local i el seu equivalent al núvol en perfecte sincronia. Per aconseguir-ho, el seu disseny es basa en un model de doble vigilància: monitoritza activament el sistema d'arxius local i, simultàniament, escolta els canvis provinents del servidor a través d'una connexió WebSocket.

El punt de partida del procés és la funció start, que s'executa en un fil de fons quan l'aplicació s'inicia. Aquesta funció orquestra la posada en marxa de dues tasques asíncrones principals: la que gestiona la connexió WebSocket i la que vigila el sistema de fitxers local.

La primera tasca que s'inicia és la connexió amb el servidor, encapsulada dins d'un bucle per garantir la reconnexió automàtica en cas de pèrdua de connexió.

```
1  let socket_task = async move -
2      loop -
3          let config = CONFIG.lock().unwrap().clone();
4          let token = config.token.as_ref().unwrap().value.clone();
5
6          let socket_url = format!(
7              "-"/websocket",
8              config.server_url.replace("http://", "ws://")
9          );
10         let uri: Uri = socket_url.parse().unwrap();
11
12         let request = ClientRequestBuilder::new(uri).with_header("
       authorization", &token);
13         match connect_async(request).await -
14             Ok((mut socket, _response)) =¿ -
15                 println!("Connected to server");
16                 *IS_CONNECTED.lock().unwrap() = true;
17                 app.emit_all("is_connected", true).unwrap();
18                 while let Some(msg) = socket.next().await -
19                     // ... Processament de missatges
20                 "
21             "
22             Err(e) =¿ -
23                 // ... Gestionar error i reintentar connexio
24                 tokio::time::sleep(std::time::Duration::from_secs(5)).
       await;
25             "
26         "
27     "
```

```
";
```

LISTING 9.89: Establiment de la connexió WebSocket a synchronizer.rs

Un cop connectat, el motor necessita un punt de referència per saber quin és l'estat inicial del directori local. Per a això, utilitza un fitxer anomenat tree.json, que actua com una fotografia o *snapshot* de l'estructura de fitxers local de l'última execució. Si el fitxer no existeix, el motor el crea recorrent tota la carpeta de sincronització i calculant els hashes de cada fitxer. Aquesta estructura d'arbre es carrega en memòria i servirà com a base per a totes les comparacions futures.

Sincronització Local-a-Remot: Vigilant el Sistema d'Arxius

La sincronització des del client cap al servidor s'inicia gràcies a la vigilància constant del sistema d'arxius. Per a aquesta tasca, es va utilitzar la llibreria notify, que monitoritza de forma eficient qualsevol canvi a la carpeta seleccionada.

Quan es produeix un canvi (creació, modificació, eliminació), notify genera un esdeveniment que és capturat per la funció handle_event. Aquesta funció actualitza l'arbre en memòria i, per evitar una allau de peticions al servidor, passa la lògica de comparació a un *debouncer*. Aquest component agrupa tots els canvis que ocorren en un breu interval i llança una única acció quan detecta que l'activitat s'ha aturat.

Un cop el *debouncer* activa l'acció, es desencadena el nucli de la lògica de comparació:

1. **Recuperació de l'estat previ**: Es llegeix l'estat de l'última sincronització correcta des del fitxer tree.json.

2. **Detecció de diferències**: L'arbre en memòria (que representa l'estat actual) es compara amb l'arbre llegit de tree.json mitjançant la funció fstree::diff_trees. Aquesta funció recorre recursivament tots dos arbres i, comparant noms i hashes de contingut, genera una llista precisa de canvis.

3. **Detecció de canvis de nom**: Es processa la llista de canvis amb fstree::detect_renames. Si un fitxer eliminat i un d'afegit tenen el mateix hash, s'infereix que ha estat un canvi de nom, optimitzant l'operació.

4. **Execució d'accions**: Finalment, es recorre la llista de canvis i s'invoca la funció corresponent del mòdul api.rs. Per il·lustrar com es gestionen aquestes operacions de xarxa, s'utilitzarà com a exemple la funció de descàrrega, ja que mostra clarament el procés de *streaming* i el càlcul de progrés. Per a garantir la integritat de les dades i evitar fitxers corruptes, la descàrrega es realitza primer a un directori temporal. Un cop la transferència s'ha completat correctament, el fitxer es mou a la seva ubicació final.

```
pub async fn download(
    app: tauri::AppHandle,
    root_path: &PathBuf,
    path: String,
    id: Arc¡Mutex¡Option¡String¿¿¿,
    hash: String,
    local_tree: Arc¡Mutex¡fstree::Node¿¿,
) -
    // ...
```

```
10      // Es crea un fitxer temporal per a la descàrrega
11      let temp_file_path = temp_dir.join(temp_name);
12      // ...
13      let response = client.get(format!("-server"/files/-id"/download"))
14          // ...
15          .send().await;
16      // ...
17      let total_size = resp.content_length().unwrap_or(0);
18      let mut file = fs::File::create(&temp_file_path).unwrap();
19      let mut downloaded: u64 = 0;
20      let mut stream = resp.bytes_stream();
21
22      while let Some(chunk_result) = stream.next().await -
23          let chunk = chunk_result.unwrap();
24          file.write_all(&chunk).unwrap();
25          downloaded += chunk.len() as u64;
26
27          let progress = (downloaded as f64 / total_size as f64) * 100.0;
28
29          // Emet un esdeveniment amb el progrés per a la UI
30          if let Some(ref window) = window -
31              window.emit("transfer", &transfer).unwrap();
32          "
33      "
34
35      // Un cop descarregat, es mou el fitxer a la seva destinació final
36      let _ = fs::copy(&temp_file_path, &destination);
37      let _ = fs::remove_file(&temp_file_path);
38
39      // S'actualitza l'arbre local amb el nou node
40      if let Ok(Some(node)) = fstree::build_node(&root_path, &destination)
        .map(Some) -
41          local_tree.lock().unwrap().add_node(node).unwrap();
42          fstree::save_tree(&local_tree.lock().unwrap(), "tree.json").
        unwrap();
43      "
44      // ...
45  "
```

LISTING 9.90: Gestió de la descàrrega amb progrés a api.rs

Com es pot observar en el fragment de codi anterior, la comunicació entre el nucli de Rust i la interfície d'usuari és un dels punts forts de Tauri i és clau per a l'experiència en temps real. Dins del bucle de descàrrega, un cop calculat el progrés, la línia 'window.emit("transfer", &transfer).unwrap()' és la que fa tota la màgia. Aquesta funció emet un esdeveniment anomenat '"transfer"' des del backend. El frontend, que està construït amb Svelte, té un 'listener' configurat específicament per a aquest esdeveniment. Quan el frontend rep l'esdeveniment '"transfer"', també rep la càrrega útil associada —en aquest cas, l'objecte 'transfer' que conté el percentatge de progrés actualitzat. Això provoca una actualització reactiva a la interfície, i la barra de progrés es redibuixa a la pantalla de l'usuari de manera instantània. Aquest patró d'esdeveniments és el que permet que el backend, que gestiona tota la lògica pesada, pugui notificar a la vista de manera eficient sense necessitat d'un acoblament directe.

Després de processar tots els canvis, el nou arbre en memòria es desa al fitxer tree.json, establint així el nou punt de partida per a la propera detecció de canvis.

Lògica de Comparació i Detecció de Canvis

El cor del motor de sincronització resideix en les funcions diff_trees i detect_renames del mòdul fstree.rs. Aquestes funcions són les responsables d'identificar de manera eficient les diferències entre l'estat local i el remot.

La funció diff_trees implementa un algorisme de comparació recursiva. Com es pot veure al codi, navega simultàniament per dos arbres (l'antic i el nou) i compara els nodes a cada nivell.

```
pub fn diff_trees(
    path: &str,
    node_1: Option¡&Node¿, // Arbre antic (o local)
    node_2: Option¡&Node¿, // Arbre nou (o remot)
    changes: &mut Vec¡Change¿,
) -
    match (node_1, node_2) -
        (Some(node_1), Some(node_2)) =¿ -
            // Si els hashes són diferents i és un fitxer, es marca com a modificat
            if node_1.hash != node_2.hash && node_1.node_type == NodeType::File -
                changes.push(Change -
                    id: node_2.id.clone(),
                    path: path.to_string(),
                    change_type: ChangeType::Modified,
                    // ... altres camps
                ");
            "

            // Si són carpetes, es comparen els seus continguts
            if let (Some(old_children), Some(new_children)) = (&node_1.content, &node_2.content) -
                // Es crea un conjunt amb totes les claus (noms de fitxer) dels dos directoris
                let all_keys: std::collections::BTreeSet¡'¿ =
                    old_children.keys().chain(new_children.keys()).collect();

                // Es recorren totes les claus i es fa una crida recursiva
                for key in all_keys -
                    let old_child = old_children.get(key);
                    let new_child = new_children.get(key);
                    let full_path = Path::new(path).join(key).to_str().unwrap().to_string();
                    diff_trees(&full_path, old_child, new_child, changes);
                "
            "
        "
        (None, Some(new)) =¿ -
            // El node només existeix al nou arbre -¿ s'ha afegit
            changes.push(Change - change_type: ChangeType::Added, path: path.to_string(), ..Default::default() ");
        "
        (Some(old), None) =¿ -
            // El node només existeix a l'arbre antic -¿ s'ha eliminat
            changes.push(Change - change_type: ChangeType::Deleted, path: path.to_string(), ..Default::default() ");
        "
        (None, None) =¿ -" // No hi ha canvis
```

```
    "
"
```

LISTING 9.91: Lògica de la funció diff_trees a fstree.rs

Un cop diff_trees ha generat la llista inicial de canvis (addicions i eliminacions), aquesta es passa a detect_renames. Aquesta funció implementa una optimització clau: en lloc de tractar un canvi de nom com una eliminació i una addició, busca coincidències basades en el contingut.

```
pub fn detect_renames(mut changes: Vec¡Change¿) -¿ Vec¡Change¿ -
    // ... separa els canvis en llistes 'added' i 'deleted'

    while let Some(add) = added.pop() -
        if let Some(add_hash) = &add.hash -
            // Busca a la llista d'eliminats un element amb el mateix hash
            if let Some(pos) = deleted.iter().position(—del— -
                del.node_type == add.node_type && del.hash == Some(add_hash.clone())
            ") -
                // Si es troba, es tracta d'un canvi de nom
                let del = deleted.remove(pos);
                final_changes.push(Change -
                    change_type: ChangeType::Renamed - from: del.path ",
                    // ...
                ");
                continue;
            "
        "
        // Si no hi ha coincidència, és una addició real
        final_changes.push(add);
    "
    // ... afegeix les eliminacions reals restants
    final_changes
"
```

LISTING 9.92: Fragment clau de detect_renames a fstree.rs

Sincronització Remot-a-Local: Escoltant el Servidor

Per a la sincronització en sentit contrari, el motor es basa en els missatges rebuts a través de la connexió WebSocket. El servidor envia un missatge de tipus snapshot cada cop que es produeix un canvi en els fitxers de l'usuari des d'una altra font. Aquest *snapshot* conté l'estructura de fitxers actualitzada del servidor. El missatge és processat per la funció handle_msg.

```
async fn handle_msg(
    local_tree: &Arc¡Mutex¡fstree::Node¿¿,
    root_path: &PathBuf,
    app: &tauri::AppHandle,
    msg: tungstenite::Message,
) -
    let text = msg.to_string();
    println!("Received new tree");
    let mut changes: Vec¡fstree::Change¿ = Vec::new();
    let resp: SocketResponse = serde_json::from_str(&text).unwrap();
    let mut remote_tree = resp.data;
    -
        let local = local_tree.lock().unwrap();
```

```
14          fstree::diff_trees("", Some(&local), Some(&remote_tree), &mut
    changes);
15      "
16      let mut futures = vec![];
17      let changes = fstree::detect_renames(changes);
18
19      for change in changes -
20          match change.change_type -
21              fstree::ChangeType::Added =¿ -
22                  if let fstree::NodeType::File = change.node_type -
23                      // S'afegeix una nova tasca de descàrrega al vector
    de futures
24                      futures.push(api::download(
25                          app.app_handle().clone(),
26                          root_path,
27                          change.path,
28                          change.id,
29                          change.hash.unwrap(),
30                          local_tree.clone(),
31                      ));
32                  " else -
33                      // ... Crear carpeta localment
34                  "
35                  // ...
36              ",
37              // ... Gestionar altres tipus de canvis (Deleted, Renamed,
    Modified)
38          "
39      "
40      // S'executen totes les tasques de descàrrega en paralel
41      join_all(futures).await;
42      // ... Actualitzar l'arbre local si no hi ha conflictes
43  "
```

LISTING 9.93: Processament de missatges WebSocket a synchronizer.rs

El flux de processament és simètric a l'anterior:

1. Es deserialitza el JSON del missatge per obtenir l'arbre de fitxers remot (remote_tree).

2. Es torna a utilitzar la funció fstree::diff_trees, però aquest cop per comparar l'arbre local actual amb l'arbre remot rebut.

3. La llista de canvis resultant indica les accions que el client ha de prendre. Per exemple, un canvi de tipus Added significa que un fitxer existeix al servidor però no en local, per la qual cosa s'ha de descarregar mitjançant la funció api::download, el funcionament de la qual ja s'ha detallat.

4. S'executen totes les operacions de forma asíncrona amb join_all per maximitzar el rendiment. Per aconseguir-ho, en lloc d'executar cada operació de descàrrega de manera seqüencial (una darrere l'altra), el sistema primer recorre tots els canvis detectats i crea una tasca asíncrona (*future*) per a cada fitxer que s'ha de descarregar. Totes aquestes tasques s'emmagatzemen en un vector. Finalment, s'utilitza la funció join_all, que executa totes aquestes tasques de manera concurrent. El programa esperarà en aquest punt fins que l'última de les descàrregues hagi finalitzat. Aquest enfocament millora dràsticament el rendiment, especialment quan s'han de sincronitzar múltiples fitxers, ja que el

temps total de l'operació depèn del fitxer més lent, no de la suma de tots ells.

```
    // ...
    let mut futures = vec![];
    let changes = fstree::detect_renames(changes);

    for change in changes -
        match change.change_type -
            fstree::ChangeType::Added =¿ -
                if let fstree::NodeType::File = change.node_type -
                    // S'afegeix una nova tasca de descàrrega al vector
    de futures
                    futures.push(api::download(
                        app.app_handle().clone(),
                        root_path,
                        // ... arguments de la funcio
                    ));
                "
                // ...
            ",
            // ...
        "
    "
    // S'executen totes les tasques de descàrrega en paralel
    join_all(futures).await;
```

LISTING 9.94: Execució concurrent de descàrregues amb join_all a synchronizer.rs

Aquesta arquitectura de doble canal, combinant la vigilància proactiva local amb l'escolta passiva de canvis remots, crea un sistema de sincronització robust. Tanmateix, cal destacar que la gestió de conflictes actual és bàsica: si un canvi es produeix al mateix temps en local i en remot, el darrer a ser processat sobreescriurà l'altre. Com ja he esmentat, la implementació d'un sistema de resolució de conflictes més sofisticat, que permeti a l'usuari decidir quina versió conservar, és una de les millores clau plantejades com a treball futur.

9.6.4 Proves d'integració manuals

Donada la naturalesa iterativa del desenvolupament, l'estratègia de proves es va centrar en la validació funcional contínua a través de casos d'ús definits. Per a sistematitzar aquest procés, des de les primeres fases d'implementació del client web, vaig mantenir un document de seguiment, tests/test.md, que funcionava com una llista de verificació (*checklist*). Aquest fitxer recollia cadascuna de les funcionalitats implementades, des del registre d'usuaris fins a les operacions complexes com la compartició d'arxius o la sincronització en temps real. Cada vegada que s'introduïa un canvi significatiu o es completava un nou component, es realitzava una ronda de proves manuals seguint els punts d'aquesta llista per verificar que el comportament esperat es mantenia.

No obstant això, he d'admetre les limitacions inherents a aquest enfocament. La dependència exclusiva de proves manuales, sense el suport d'un sistema de tests automatitzats, va demostrar ser un punt feble en el cicle de desenvolupament. En diverses ocasions, canvis introduïts en una part del sistema van provocar regressions —errades en funcionalitats prèviament estables— que no van ser detectades

immediatament. Aquests errors només sortien a la llum durant les "rondes"de proves més exhaustives, realitzades abans de consolidar una nova versió, la qual cosa generava un sobrecost de temps en la depuració.

Encara que es va fer un esforç per cobrir tots els casos d'ús y funcionalitats de l'aplicació documentats a la llista de verificació, la falta d'un marc de proves més rigorós implica que no es pot garantir al 100% la cobertura de tots els possibles casos límit (*edge cases*). La validació es basava en l'execució dels fluxos de treball principals, però la complexitat de les interaccions, especialment en un sistema distribuït, deixa oberta la possibilitat que existeixin escenaris no contemplats que puguin generar comportaments inesperats.

Aquesta experiència subratlla una lliçó apresa fonamental: la necessitat d'integrar proves automatitzades des de l'inici del projecte. La creació de tests unitaris i d'integració hauria proporcionat una xarxa de seguretat, permetent identificar regressions de forma instantània i garantint una major robustesa del codi. Per tant, la implementació d'un pla d'automatització de proves queda establerta com una de les principals prioritats per al treball futur, tal com es detallarà més endavant en el **Capítol 12**.

9.7 Implementació dels sistemes d'instal·lació i desplegament

Per a facilitar la implantació del sistema per part d'usuaris finals, es van desenvolupar un conjunt d'eines i scripts destinats a automatitzar el procés de desplegament. Aquesta secció detalla la implementació de la infraestructura basada en Docker Compose, que orquestra tots els microserveis, bases de dades i components auxiliars. L'arxiu principal server/compose.yml defineix tota la infraestructura necessària per a un desplegament complet, mentre que server/test.yaml proporciona una configuració mínima per a proves i desenvolupament. Aquest últim fitxer conté només els contenidors mínims necessaris (PostgreSQL i RabbitMQ) descomentats, facilitant que futurs contribuïdors puguin aixecar un entorn de proves local ràpidament sense llançar tot el sistema, i depurar els serveis individualment des del seu IDE. En cas de voler-se provar nomes un o un numero reduit de microserveis es poden descomntar aquests i engegar en mode debug el microserveis necessaris, ja que s'expossen utilitzant els mateixos ports que en mode debug, facilitant les proves i utilitzant menys recursos.

9.7.1 Desplegament mitjançant Docker Compose

Arquitectura de desplegament

El sistema es desplega mitjançant Docker Compose, que orquestra el conjunt complet de microserveis, bases de dades i components auxiliars. L'arxiu principal server/compose.yml defineix tota la infraestructura necessària per a un desplegament complet, mentre que server/test.yaml proporciona una configuració mínima per a proves i desenvolupament. Aquest últim fitxer conté només els contenidors mínims necessaris (PostgreSQL i RabbitMQ) descomentats, facilitant que futurs contribuïdors puguin aixecar un entorn de proves local ràpidament sense llançar tot el sistema, i depurar els serveis individualment des del seu IDE. En cas de voler-se provar nomes un o un numero reduit de microserveis es poden descomntar aquests i engegar en mode debug el microserveis necessaris, ja que s'expossen utilitzant els mateixos ports que en mode debug, facilitant les proves i utilitzant menys recursos.

L'elecció de Docker Compose enfront de solucions més complexes com Kubernetes es justifica per la senzillesa de desplegament i la idoneïtat per a instal·lacions de mida petita i mitjana, que constitueixen el públic objectiu principal del projecte. Com a treball futur, es planteja la creació d'un conjunt de fitxers Helm que permetin una configuració senzilla a Kubernetes. No obstant això, donat el temps disponible i el públic objectiu principal —usuaris que busquen una solució de gestió de fitxers al núvol sense coneixements tècnics avançats—, es va prioritzar el desenvolupament de l'aplicació i un mètode de desplegament simple. Kubernetes representa una solució de gestió més avançada, orientada a entorns de major complexitat que els previstos inicialment per al projecte.

Estructura de l'arxiu compose.yml

L'arxiu server/compose.yml defineix els serveis següents:

Serveis d'infraestructura A la base de l'ecosistema es troben els serveis que proporcionen la persistència de dades i la comunicació asíncrona.

```yaml
postgres:
  image: postgres:latest
  ports:
    - "5432:5432"
  environment:
    POSTGRES_USER: ${POSTGRES_USER:-admin}
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-admin}
    POSTGRES_DB: ${POSTGRES_DB:-mydb}
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql

rabbitmq:
  image: rabbitmq:management
  ports:
    - "5672:5672"
    - "15672:15672"
  environment:
    RABBITMQ_DEFAULT_USER: ${RABBITMQ_DEFAULT_USER:-admin}
    RABBITMQ_DEFAULT_PASS: ${RABBITMQ_DEFAULT_PASS:-admin}
```

LISTING 9.95: Serveis d'infraestructura a compose.yml

Com es mostra al llistat 9.95, la configuració és flexible:

- **postgres**: Base de dades PostgreSQL. El fitxer compose.yml permet configurar el nom d'usuari (POSTGRES_USER), la contrasenya (POSTGRES_PASSWORD) i el nom de la base de dades (POSTGRES_DB) mitjançant variables d'entorn externes, amb valors per defecte admin, admin i mydb respectivament.

- **rabbitmq**: Broker de missatgeria. De manera similar, les credencials d'accés (RABBITMQ_DEFAULT_USER, RABBITMQ_DEFAULT_PASS) són configurables des de l'exterior, amb admin/admin com a valors per defecte.

- **pgadmin**: Interfície web per a l'administració de PostgreSQL, les credencials de la qual (PGADMIN_DEFAULT_EMAIL, PGADMIN_DEFAULT_PASSWORD) també són configurables amb valors per defecte. Aquest servei no es mostra al fragment per brevetat.

Actualment, els ports dels serveis són fixos. La possibilitat de configurar-los es contempla com una línia de treball futur, tal com es detalla al capítol 12.

Registre i enrutament Aquests serveis són el nucli de la comunicació dins de l'arquitectura de microserveis. El seu funcionament es defineix de la següent manera al fitxer compose.yml:

```
eureka:
  build: ./Eureka
  container_name: eureka
  ports:
    - "8761:8761"
  environment:
    - "SPRING_PROFILES_ACTIVE=docker"
  healthcheck:
    test: ["CMD", "wget", "-q", "-O", "-", "http://localhost:8761/actuator/health"]
    interval: 30s
    timeout: 10s
    retries: 5
    start_period: 40s

gateway:
  container_name: gateway
  build: ./Gateway
  ports:
    - "8762:8762"
  environment:
    - "SPRING_PROFILES_ACTIVE=docker"
  depends_on:
    eureka:
      condition: service_healthy
```

LISTING 9.96: Definició d'Eureka i Gateway a compose.yml

Com es pot observar al llistat 9.96:

- **eureka**: Servei de descobriment que permet als microserveis registrar-se i localitzar-se mútuament. S'exposa al port 8761 i inclou un healthcheck que verifica la disponibilitat abans de permetre que altres serveis s'hi connectin. Aquest mecanisme és crucial per garantir una arrencada estable del sistema.

- **gateway**: Punt d'entrada únic que exposa l'API REST al port 8762 i gestiona l'enrutament cap als microserveis apropiats. La directiva depends_on assegura que no s'iniciarà fins que el servei Eureka estigui saludable (service_healthy). També proporciona l'endpoint WebSocket per a la sincronització en temps real.

Microserveis de negoci El cor de la lògica de l'aplicació resideix en un conjunt de microserveis que s'encarreguen de funcions específiques. Tots comparteixen una configuració similar, com es pot veure en l'exemple del servei file-manager:

```
file-manager:
  build: ./FileManagement
  container_name: fiulemanagement
  environment:
    - "SPRING_PROFILES_ACTIVE=docker"
  depends_on:
    eureka:
      condition: service_healthy
```

```
    volumes:
      - ./storage_data:/app/files
```

LISTING 9.97: Exemple de microservei de negoci a compose.yml

La configuració, exemplificada en el llistat 9.97, és anàloga per a tots els serveis de negoci:

- **build**: Especifica el directori on es troba el Dockerfile per construir la imatge del servei (p. ex., ./FileManagement).
- **environment**: Activa el perfil docker de Spring Boot, que carrega la configuració específica per a l'entorn de contenidors (connexió a Eureka, RabbitMQ i PostgreSQL).
- **depends_on**: Garanteix que cada microservei només s'iniciarà quan Eureka estigui operatiu.
- **volumes**: En el cas de file-manager, s'utilitza per mapejar el directori local ./storage_data amb el directori /app/files dins del contenidor, assegurant la persistència dels fitxers pujats.

Els serveis inclosos en aquesta categoria són: **user-auth**, **user-management**, **file-access**, **file-sharing**, **trash**, **file-manager** i **sync**.

Interfície d'usuari El component final del sistema és la interfície web, que es defineix de la següent manera:

```
  ui-new:
    build:
      context: ../ui-new
      dockerfile: Dockerfile
      args:
        VITE_API_URL: http://gateway:8762
        VITE_WS_URL: ws://gateway:8762/websocket/web
    container_name: ui-new
    ports:
      - "8080:80"
    depends_on:
      gateway:
        condition: service_started
```

LISTING 9.98: Definició de la UI a compose.yml

Del llistat 9.98 es desprèn que:

- El servei **ui-new** es construeix a partir del directori ../ui-new.
- Mitjançant args, es passen les variables d'entorn VITE_API_URL i VITE_WS_URL al procés de construcció de Vite. Aquestes variables configuren la connexió amb el gateway utilitzant el nom de servei gateway, que és resolt per la xarxa interna de Docker.
- El servei exposa el port intern 80 al port 8080 del host.
- La directiva depends_on amb la condició service_started assegura que la interfície no s'iniciarà fins que el gateway estigui disponible.

Tots els microserveis utilitzen el perfil docker de Spring Boot i depenen que Eureka estigui funcionant abans d'iniciar-se, assegurant una arrencada ordenada del sistema.

Configuració de xarxa i volums

Per garantir la persistència de les dades més enllà del cicle de vida dels contenidors, s'utilitzen volums de Docker, que desacoblen les dades del contenidor mitjançant un mapeig entre un directori de l'amfitrió i un d'intern. Aquesta arquitectura és fonamental per assegurar la integritat i durabilitat de la informació, ja que fitxers crítics romanen intactes al sistema amfitrió encara que els contenidors siguin eliminats o recreats. En aquest projecte, el volum postgres_data garanteix la persistència de la base de dades, mentre que el directori ./storage_data emmagatzema els arxius dels usuaris.

Addicionalment, Docker Compose crea una xarxa interna que permet la comunicació entre serveis utilitzant els seus noms com a *hostnames*.

Serà responsabilitat de l'administrador del sistema assegurar que les dades persistides no es conservin més enllà de l'ús previst de l'aplicació.

9.7.2 Scripts d'instal·lació automatitzada

Per simplificar el desplegament, s'han desenvolupat scripts d'instal·lació que automatitzen tot el procés, des de la descàrrega de dependències fins a la creació de l'usuari administrador inicial.

Script per a Linux (setup.sh)

L'script setup.sh proporciona les funcionalitats següents:

- -i: Instal·lació completa que inclou Docker, Node.js, Rust, compilació del codi (backend, web i aplicació d'escriptori), arrencada del sistema i creació obligatòria del superadministrador.
- -u: Inicia el sistema sense reinstal·lar les dependències.
- -d: Atura tots els serveis.
- -r: Elimina completament el sistema i totes les dades.
- -b: Actualitza el sistema recompilant els serveis, incloent-hi la regeneració dels instal·ladors de l'aplicació d'escriptori.
- -t: Configura el sistema amb dades de prova (només vàlid amb -i). Utilitza el fitxer init.sql que conté usuaris, carpetes, fitxers i comparticions preconfigurats per facilitar les proves de l'aplicació.

Dependències i construcció automàtica L'script s'encarrega automàticament d'instal·lar totes les dependències necessàries:

- **Docker**: Per a l'orquestració dels microserveis.

- **Node.js i pnpm**: Per a la construcció de la interfície web i l'aplicació d'escriptori.
- **Rust**: Per a la construcció del backend natiu de l'aplicació Tauri.

Durant el procés de construcció, l'script executa seqüencialment:

1. Compilació dels microserveis Java mitjançant Maven (./mvnw clean install)
2. Instal·lació de dependències del client web (npm install)
3. Construcció completa de l'aplicació Tauri (pnpm tauri build)

La construcció de l'aplicació Tauri genera automàticament els instal·ladors natius per a la plataforma actual i els mou a la carpeta installers/ a l'arrel del projecte per facilitar l'accés. Els formats generats inclouen .deb i .AppImage per a Linux, .msi per a Windows, i .dmg per a macOS. En cas d'error durant la construcció, l'script continua amb la resta del procés però mostra un missatge d'advertència.

Compilació multiplataforma Per defecte, Tauri només genera instal·ladors per al sistema operatiu on s'executa la construcció. No obstant això, els scripts intenten configurar suport de compilació creuada quan és possible:

- **Linux**: Plataforma més flexible per a la compilació creuada. Pot generar instal·ladors per a Windows amb les eines adequades instal·lades.
- **Windows**: Suport limitat per a compilació creuada. Genera principalment instal·ladors natius de Windows.
- **macOS**: Pot generar instal·ladors per a diferents arquitectures de Mac (Intel i Apple Silicon) però no per a altres sistemes operatius.

Per obtenir instal·ladors per a totes les plataformes, es recomana:

1. Executar la construcció en un sistema Linux amb eines de compilació creuada.
2. Executar manualment en cada plataforma objectiu o amb màquines virtuals.

Els scripts mostren informació sobre la plataforma actual i les limitacions de compilació creuada per orientar l'usuari.

Com a treball futur, es preveu automatitzar aquest procés mitjançant GitHub Actions. Aquesta solució utilitzaria *runners* per a cada sistema operatiu (Linux, Windows i macOS), permetent generar instal·ladors natius i signats digitalment per a totes les plataformes de manera fiable. Els binaris generats no s'emmagatzemarien al repositori de Git, ja que no és una pràctica recomanada per a fitxers grans, sinó que es publicarien a través de GitHub Releases, facilitant-ne la distribució als usuaris finals.

Configuració de credencials L'script permet personalitzar les credencials dels serveis d'infraestructura mitjançant paràmetres de línia d'ordres, que només es poden utilitzar amb l'opció -i:

- -up ¡usuari¿: Nom d'usuari de PostgreSQL

- -pp ¡contrasenya¿: Contrasenya de PostgreSQL
- -dp ¡base_dades¿: Nom de la base de dades PostgreSQL
- -ur ¡usuari¿: Nom d'usuari de RabbitMQ
- -pr ¡contrasenya¿: Contrasenya de RabbitMQ
- -ea ¡email¿: Email de pgAdmin
- -pa ¡contrasenya¿: Contrasenya de pgAdmin

Exemple d'ús:

```
./setup.sh -i -up myuser -pp mypass -dp mydb
```

Mode de proves amb dades preconfigurades L'opció -t permet iniciar el sistema amb un conjunt de dades de prova predefinides, facilitant la validació de funcionalitats sense necessitat de crear manualment usuaris, carpetes i fitxers. Quan s'utilitza aquesta opció:

- Es substitueix el fitxer d'inicialització de base de dades buit (init-creation.sql) pel fitxer init.sql, que conté dades de mostra.
- El sistema es configura automàticament amb usuaris de prova, carpetes, fitxers i relacions de compartició preestablertes.
- Es continua amb la creació del superadministrador de forma normal, afegint-se als usuaris de prova existents.

Aquest mode és especialment útil per a desenvolupadors, proves d'integració o demonstrations de l'aplicació, ja que proporciona un entorn preonfigurat que permet provar immediatament totes les funcionalitats sense haver de crear manualment el contingut necessari.

Exemple d'ús amb dades de prova:

```
./setup.sh -i -t -up myuser -pp mypass -dp mydb
```

Persistència de la configuració Quan s'utilitzen paràmetres personalitzats durant la instal·lació, l'script genera automàticament un fitxer .env a l'arrel del projecte que conté totes les credencials configurades. Aquest fitxer:

- Es carrega automàticament en futures execucions dels scripts (operacions -u, -b).
- Garanteix que les credencials personalitzades es mantinguin consistents entre reinicis.

Si no es proporcionen paràmetres personalitzats, el sistema utilitza els valors per defecte (admin/admin per als serveis, mydb per a la base de dades i admin@admin.com/admin per a pgAdmin).

En cas de utilitzar alguna d'aquestes opcions per generar credencials personalitzades es mostra un avís per pantalla indicant la importància del fitxer que es genera i que no s'ha d'eliminar.

Script per a Windows (setup.ps1)

L'script PowerShell setup.ps1 replica exactament la funcionalitat de l'script de Linux, incloent-hi:

- Instal·lació automàtica de dependències (Docker Desktop, Node.js, Rust) mitjançant gestors de paquets disponibles
- Construcció completa de tots els components del sistema
- Generació dels instal·ladors de l'aplicació d'escriptori per a Windows (.msi i .exe)
- Opcions de configuració de credencials idèntiques
- Generació i càrrega automàtica del fitxer .env
- Suport per al mode de dades de prova amb l'opció -t

L'script utilitza winget com a gestor de paquets preferit, amb chocolatey com a alternativa, i proporciona instruccions manuals si cap dels dos està disponible.

Procés de creació del superadministrador

La creació del superadministrador s'ha integrat com a part obligatòria del procés d'instal·lació inicial (opció -i). Aquesta decisió de disseny garanteix que sempre existeixi almenys un administrador amb permisos complets per gestionar el sistema des del primer moment. El procés utilitza les credencials de PostgreSQL configurades (ja siguin personalitzades o per defecte) per connectar-se a la base de dades i crear els registres necessaris.

Existeix la possibilitat de que un usuari mes avançat pugui crear mes superadministradors directament a la base de dades, però no es contempla com a part de l'instal·lació automatica ja que aixo incompleix el disseny inicial de l'aplicació, però tampoc s'han fet comprovacions ni limitacions en el codi per evitar-ho, es deixa com a responsabilitat de l'administrador del sistema, que es qui tindra les credencials d'instal·lació, la gestió d'aquesta casoistica concreta.

Capítol 10

Implantació i resultats

10.1 Introducció

Aquest capítol presenta la demostració visual de les funcionalitats implementades al sistema de gestió d'arxius al núvol desenvolupat. Mitjançant captures de pantalla reals del sistema en funcionament, es valida el compliment dels requisits funcionals establerts i es comprova que totes les funcionalitats principals operen correctament.

Les captures mostrades han estat obtingudes durant proves reals del sistema, demostrant que la plataforma ofereix una alternativa funcional als serveis comercials d'emmagatzematge al núvol tal com s'havia plantejat en els objectius inicials.

10.2 Autenticació i gestió d'usuaris

El sistema permet el registre de nous usuaris i l'autenticació segura tant des del client web com des de l'aplicació d'escriptori.

FIGURA 10.1: Registre de nou usuari

FIGURA 10.2: Confirmació d'èxit del registre

FIGURA 10.3: Inici de sessió web

FIGURA 10.4: Autenticació exitosa

FIGURA 10.5: Autenticació en l'aplicació d'escriptori

10.3 Gestió d'arxius

10.3.1 Pujada d'arxius

El sistema permet pujar arxius de manera individual amb confirmació d'èxit.

FIGURA 10.6: Pujada d'arxius

FIGURA 10.7: Confirmació de pujada exitosa

10.3.2 Descàrrega d'arxius

Els usuaris poden descarregar arxius individuals des de la interfície web.

FIGURA 10.8: Opció de descàrrega d'arxiu

FIGURA 10.9: Descàrrega completada amb èxit

10.3.3 Creació de carpetes

Es pot crear noves carpetes tant des del client web com des de l'aplicació d'escriptori.

FIGURA 10.10: Creació de nova carpeta - Client web

FIGURA 10.11: Diàleg de creació de carpeta

FIGURA 10.12: Carpeta creada correctament

FIGURA 10.13: Creació de carpeta - Client d'escriptori

FIGURA 10.14: Carpeta creada des del client d'escriptori

FIGURA 10.15: Verificació de la creació de carpeta

10.4 Operacions amb arxius

10.4.1 Operacions de copiar, tallar i enganxar

El sistema suporta les operacions estàndard de porta-retalls per gestionar arxius.

FIGURA 10.16: Operació de tallar arxiu

FIGURA 10.17: Enganxar arxiu tallat

FIGURA 10.18: Enganxar arxiu copiat

10.4.2 Moviment i redenominació

Els arxius es poden moure entre ubicacions i canviar de nom.

FIGURA 10.19: Moviment d'arxiu

FIGURA 10.20: Arxiu mogut correctament

FIGURA 10.21: Redenominació d'arxiu

10.4.3 Eliminació d'arxius

Els arxius s'eliminen amb confirmació i es mouen a la paperera.

FIGURA 10.22: Eliminació d'arxiu

FIGURA 10.23: Arxiu eliminat correctament

10.5 Sistema de paperera

La paperera permet recuperar arxius eliminats accidentalment.

FIGURA 10.24: Restauració d'arxiu des de la paperera

FIGURA 10.25: Arxiu restaurat correctament

10.6 Compartició d'arxius

10.6.1 Creació de comparticions

Els usuaris poden compartir arxius amb altres usuaris del sistema.

FIGURA 10.26: Diàleg de compartició d'arxius

FIGURA 10.27: Arxiu compartit correctament

10.6.2 Gestió d'arxius compartits

El sistema mostra els arxius compartits amb indicadors específics.

FIGURA 10.28: Visualització d'arxiu compartit

FIGURA 10.29: Llistat d'arxius compartits

10.6.3 Modificació i revocació de comparticions

Es poden modificar els permisos i revocar l'accés a arxius compartits.

FIGURA 10.30: Modificació de permisos de compartició

FIGURA 10.31: Revocació d'accés a arxiu compartit

FIGURA 10.32: Accés revocat correctament

FIGURA 10.33: Arxiu eliminat de la llista de compartits

10.7 Sincronització

L'aplicació d'escriptori proporciona sincronització automàtica amb el núvol.

FIGURA 10.34: Interfície de sincronització

FIGURA 10.35: Sincronització completada correctament

10.8 Funcionalitats addicionals

10.8.1 Ordenació d'arxius

El sistema ofereix diferents opcions d'ordenació per organitzar la visualització.

FIGURA 10.36: Opcions d'ordenació

FIGURA 10.37: Arxius ordenats per criteris seleccionats

10.9 Panell d'administració

10.9.1 Gestió d'usuaris

Els administradors poden gestionar els usuaris del sistema.

FIGURA 10.38: Panell d'administració principal

FIGURA 10.39: Actualització d'informació d'usuari

FIGURA 10.40: Usuari actualitzat correctament

10.9.2 Eliminació d'usuaris

El sistema permet eliminar usuaris amb un procés de confirmació.

FIGURA 10.41: Eliminació d'usuari - Pas 1

FIGURA 10.42: Eliminació d'usuari - Confirmació

FIGURA 10.43: Usuari eliminat correctament

10.10 Gestió de perfils

Els usuaris poden actualitzar la seva informació personal i canviar contrasenyes.

FIGURA 10.44: Actualització de perfil d'usuari

FIGURA 10.45: Perfil actualitzat correctament

FIGURA 10.46: Contrasenya canviada correctament

FIGURA 10.47: Validació d'errors en formularis

FIGURA 10.48: Accés a configuració després del registre

10.11 Compliment dels objectius

Les captures presentades demostren que s'han implementat satisfactòriament tots els requisits funcionals principals definits al projecte:

- Autenticació i gestió d'usuaris completa
- Operacions bàsiques de gestió d'arxius (pujada, descàrrega, eliminació)
- Sistema de paperera funcional
- Compartició d'arxius amb control de permisos
- Sincronització entre client web i d'escriptori
- Panell d'administració operatiu
- Gestió de perfils d'usuari

El sistema ha assolit un 95% dels objectius establerts, proporcionant una alternativa funcional i completa als serveis comercials d'emmagatzematge al núvol.

10.12 Demostració de sincronització en temps real

Per completar la demostració de les funcionalitats del sistema, s'ha creat un vídeo que mostra la sincronització en temps real entre múltiples sessions d'usuari i diferents clients (web i escriptori). Aquest vídeo, disponible al repositori GitHub del projecte, demostra:

- Sincronització automàtica de canvis entre sessions web simultànies
- Propagació immediata d'operacions d'arxius entre usuaris
- Actualització en temps real de comparticions i permisos
- Sincronització bidireccional entre el client d'escriptori i el núvol

- Notificacions instantànies de canvis realitzats per altres usuaris

Aquest vídeo complementa les captures estàtiques presentades en aquest capítol, oferint una visió completa del funcionament del sistema en condicions reals d'ús col·laboratiu.

10.13 Avaluació del Compliment Normatiu

10.13.1 Enfocament del compliment en un projecte de Codi Obert

Aquest projecte es distribueix com una eina de programari de codi obert (Open Source). Això implica una distinció clau en les responsabilitats legals: el codi font ofereix les bases i mecanismes per a un funcionament respectuós amb la normativa, però la responsabilitat final del compliment recau en la persona o entitat (l' "operador"o "administrador") que desplega la plataforma per al seu ús.

A continuació, s'analitza el grau de compliment des d'aquesta doble perspectiva: les funcionalitats que el projecte incorpora de base i les responsabilitats que l'administrador ha d'assumir.

10.13.2 Mesures de compliment implementades al codi

El sistema ha estat dissenyat amb una base sòlida per facilitar un desplegament legalment respectuós, aplicant els següents principis i mesures:

- **Base legal del tractament (RGPD):** El funcionament del programari es justifica sota la base legal de l'execució d'un contracte (art. 6.1.b del RGPD). Quan un usuari es registra, accepta uns termes d'ús per rebre el servei d'emmagatzematge, i el tractament de les seves dades (nom, email, fitxers) és estrictament necessari per a aquesta finalitat. No es requereix un consentiment addicional per a les funcionalitats bàsiques.

- **Minimització de dades:** El disseny inicial de la plataforma contempla la recollida de dades d'usuari considerades estàndard: nom, cognoms, correu electrònic, nom d'usuari i una contrasenya. Si bé el nom i els cognoms no són estrictament indispensables per al funcionament del servei, la seva inclusió es va basar en la pràctica comuna per a aplicacions d'aquest tipus. Aquesta decisió, que s'allunya d'una aplicació estricta del principi de minimització, es tracta com un punt de millora a la secció de treballs a futur.

- **Protecció de credencials:** Les contrasenyes dels usuaris es protegeixen a la base de dades utilitzant l'algorisme de hash segur BCrypt, que impedeix que puguin ser llegides directament, fins i tot per l'administrador del sistema.

- **Control d'accés lògic:** L'arquitectura garanteix que, per disseny, un usuari només pugui accedir i gestionar els seus propis fitxers i dades, evitant accessos no autoritzats entre usuaris.

- **Drets de l'usuari implementats:** La interfície d'usuari permet exercir directament diversos drets fonamentals del RGPD:

- **Dret d'accés i rectificació:** L'usuari pot veure i modificar les seves dades de perfil en qualsevol moment.
- **Dret de portabilitat:** L'usuari té la capacitat de descarregar tots els seus fitxers de forma senzilla.
- **Dret de supressió ("dret a l'oblit"):** El sistema inclou una funcionalitat per eliminar el compte d'usuari. Aquesta acció esborra de forma immediata i permanent totes les dades associades a l'usuari de la base de dades del sistema.

10.13.3 Responsabilitats transferides a l'Administrador del Sistema

Degut a la seva naturalesa de codi obert, certes obligacions legals depenen directament de la configuració i gestió que realitzi l'administrador que desplega la plataforma. El projecte facilita aquesta tasca proporcionant plantilles i documentació.

- **Identificació del prestador del servei (LSSICE):** El programari, per si mateix, no té un "propietari"que presti el servei. És l'administrador qui ho fa. Per això, el projecte inclou a la carpeta 'legal/' una plantilla d'Avís Legal ('AVIS_LEGAL.md') i Política de Privacitat ('PRIVACITAT.md'). L'administrador té l'obligació d'omplir aquests documents amb les seves dades reals (nom, NIF, contacte, etc.) i fer-los accessibles des de la plataforma per complir amb la LSSICE i el RGPD.
- **Documentació de compliment (RGPD):** De la mateixa manera, el projecte ofereix una plantilla per al Registre d'Activitats de Tractament a 'docs/rgpd-autoevaluacio.md'. És responsabilitat de l'administrador completar aquest document i realitzar les avaluacions d'impacte que siguin necessàries segons l'ús que se li doni a la plataforma.
- **Gestió de còpies de seguretat:** El programari no realitza còpies de seguretat de forma automàtica. La responsabilitat de configurar i executar una política de backups (i de la seva correcta protecció, com el xifratge) recau completament en l'administrador, que haurà de definir la freqüència i el mètode de retenció segons les seves necessitats i obligacions legals.
- **Gestió de drets específics:** Peticions legals concretes, com el "dret a l'oposició"per una situació particular de l'usuari, han de ser gestionades per l'administrador del servei, ja que la plataforma no pot automatitzar la valoració jurídica de cada cas. La via de contacte per a aquestes peticions ha d'estar definida a la Política de Privacitat.

10.13.4 Mesures no aplicades i Treball a Futur (Capítol 12)

Durant el desenvolupament s'han prioritzat funcionalitats essencials, deixant certes mesures de seguretat i compliment tècnic com a treball a futur. Aquestes millores són crucials abans de considerar un desplegament en un entorn de producció real i formaran part de les línies de treball futures descrites al capítol 12.

- **Revisió del principi de minimització de dades:** Atesa la integració d'aquests camps al codi, s'analitzarà a futur la possibilitat d'eliminar la recollida del nom i els cognoms de l'usuari durant el registre. Això implicaria modificar l'aplicació per operar únicament amb un nom d'usuari (o l'email) i la contrasenya, la

qual cosa alinearia el projecte de forma més estricta amb el principi de minimització de dades.

- **Xifratge de les comunicacions (TLS/HTTPS):** Actualment, la comunicació entre el client (navegador o aplicació d'escriptori) i el servidor, així com la comunicació interna entre microserveis, no està xifrada. Implementar HTTPS és un requisit de seguretat absolutament prioritari i fonamental abans de qualsevol ús real.

- **Gestió segura de tokens d'autenticació:** Els tokens JWT es transmeten a través de les capçaleres de les peticions. Una millora de seguretat futura seria transferirlos a galetes (cookies) amb els atributs 'HttpOnly' i 'Secure' per mitigar riscos d'atacs tipus XSS.

- **Sistema de registres (Logging) centralitzat:** Durant les fases inicials del desenvolupament, es va implementar un sistema bàsic de registres (logs) directament a cada microservei. No obstant això, es va fer evident que aquest enfocament era inadequat per a una arquitectura distribuïda. La naturalesa dels microserveis, on una única acció de l'usuari pot generar múltiples transaccions internes entre diferents serveis, provocava la creació de registres fragmentats i desconnectats, fent extremadament difícil seguir el rastre d'una operació completa, especialment amb múltiples usuaris actius simultàniament.

  Davant la ineficàcia i la manca de cobertura adequada d'aquests registres, es va prendre la decisió estratègica d'eliminar completament la funcionalitat de logging existent. La solució correcta, que es planteja com un treball a futur clau al capítol 12, consisteix en la implementació d'un servei de logging centralitzat. Aquest servei s'encarregaria d'agregar, ordenar i correlacionar els registres de tots els microserveis, proporcionant una traçabilitat clara i entenedora de les operacions i facilitant la detecció d'errors i la monitorització de seguretat. Conseqüentment, en l'estat actual, la plataforma no genera ni emmagatzema registres d'activitat.

10.14 Conclusió

La implantació del sistema de gestió d'arxius al núvol ha resultat exitosa, assolint la gran majoria dels objectius funcionals plantejats inicialment. La plataforma desenvolupada ofereix una alternativa operativa per a la gestió de fitxers, amb les funcionalitats essencials demostrades: autenticació, gestió d'arxius propis, paperera de reciclatge i la sincronització d'aquests amb el client d'escriptori. El mòdul per compartir arxius entre usuaris és plenament funcional a la interfície web, però la seva gestió des del client d'escriptori no es va poder implementar per falta de temps, quedant com una línia de treball a futur.

Pel que fa al compliment normatiu, el projecte estableix una base sòlida. S'han integrat al nucli del programari principis fonamentals del RGPD, com la minimització de dades (amb les consideracions ja esmentades), la protecció de contrasenyes i la garantia dels drets d'accés, rectificació, portabilitat i supressió dels usuaris. Així mateix, es facilita a l'administrador la tasca de compliment amb la LSSICE mitjançant la provisió de plantilles legals.

No obstant això, és crucial destacar que, en el seu estat actual, el sistema no està preparat per a un desplegament en un entorn de producció real. La manca de mesures de seguretat crítiques, com el xifratge de comunicacions (HTTPS) i un sistema de registres centralitzat, representa un incompliment significatiu de les obligacions de seguretat exigides per la normativa de protecció de dades. Aquestes mancances són reconegudes i seran l'eix principal del treball a futur descrit al capítol 12.

En resum, el projecte constitueix una prova de concepte robusta i una base excel·lent sobre la qual construir. S'han complert els objectius de desenvolupar una plataforma funcional i s'ha traçat un camí clar per assolir el compliment normatiu i la seguretat necessaris per al seu ús en un entorn real.

La disponibilitat del vídeo de demostració de sincronització al repositori GitHub complementa aquesta documentació, oferint una visió dinàmica del funcionament del sistema en temps real.

Capítol 11

Conclusions

En aquest capítol final, es realitza una anàlisi retrospectiva del projecte, avaluant el grau de compliment dels objectius establerts, tant des de la perspectiva acadèmica del Treball de Fi de Grau com des de la funcionalitat de l'aplicació desenvolupada. S'analitzaran les desviacions respecte a la planificació inicial, se'n discutiran les causes i es farà una reflexió crítica sobre els resultats obtinguts i les lliçons apreses durant el procés.

11.1 Assoliment dels objectius

A l'hora de valorar l'èxit del projecte, és imprescindible diferenciar entre els objectius acadèmics inherents a un Treball de Fi de Grau (TFG) i els requisits funcionals de l'aplicació com a producte de programari.

11.1.1 Valoració com a Treball de Fi de Grau

Des d'una perspectiva acadèmica, el projecte ha complert de manera satisfactòria els objectius d'aprenentatge que em vaig marcar. La finalitat principal d'un TFG és aplicar i consolidar els coneixements adquirits durant la carrera, i en aquest sentit, el desenvolupament d'aquesta plataforma ha representat un repte significatiu que m'ha permès:

- **Dissenyar i implementar un sistema distribuït complex:** He pogut materialitzar els conceptes teòrics d'una arquitectura de microserveis, enfrontant-me a reptes reals com la comunicació entre serveis (síncrona i asíncrona), el descobriment, la seguretat centralitzada en un *gateway* i la gestió de dades distribuïdes.

- **Adquirir experiència pràctica amb una pila tecnològica moderna:** El projecte m'ha permès treballar amb un conjunt d'eines estàndard a la indústria, des de l'ecosistema de Spring Boot per al backend fins a React per al frontend web. Cal destacar l'aprenentatge en tecnologies més noves com Tauri i Rust, que, tot i la seva corba d'aprenentatge, han estat clau per assolir els objectius de rendiment del client d'escriptori.

- **Gestionar un projecte de gran escala de forma autònoma:** M'he enfrontat al cicle de vida complet del desenvolupament de programari, des de la definició de requisits i la planificació fins a la implementació, les proves manuals i el

desplegament amb Docker, la qual cosa ha estat una experiència formativa molt valuosa.

Com a exercici acadèmic, considero que el projecte ha servit com un camp de proves excel·lent per consolidar la meva formació com a enginyer de programari.

11.1.2 Valoració del producte desenvolupat

Si avaluem l'aplicació respecte als requisits funcionals definits al Capítol 6, el resultat és parcialment positiu. S'ha aconseguit una plataforma funcional que implementa el nucli de les característiques planificades: els usuaris poden registrar-se, gestionar els seus fitxers, utilitzar la paperera, compartir arxius (a la versió web) i sincronitzar la seva carpeta principal amb el client d'escriptori. El panell d'administració és operatiu i permet una gestió bàsica d'usuaris.

No obstant això, cal ser transparent sobre les mancances. La funcionalitat més notable que va quedar pendent va ser la **sincronització dels fitxers compartits al client d'escriptori**. Aquesta era una característica complexa que va ser descartada per les limitacions de temps. A més, com s'ha reconegut al llarg de la memòria, el sistema en el seu estat actual **no està preparat per a un entorn de producció real**. L'absència de mesures de seguretat crítiques com el xifratge de les comunicacions (HTTPS) o un sistema de registres centralitzat són obstacles insalvables per al seu ús més enllà d'una prova de concepte.

Malgrat que s'ha implementat una part considerable del projecte, no estic del tot satisfet amb el resultat final, especialment després del procés de redacció d'aquesta memòria. Aquest exercici de documentació m'ha fet plenament conscient de la gran quantitat de treball que queda pendent per poder presentar el projecte com una eina robusta i segura, preparada per a l'ús d'usuaris externs.

11.2 Desviacions de la planificació

Com es va detallar al Capítol 4, el projecte va patir desviacions molt significatives respecte al calendari inicial. La planificació original de vuit mesos va resultar ser excessivament optimista, i el desenvolupament es va allargar considerablement. Les causes principals d'aquesta desviació van ser:

1. **Discontinuïtat en la dedicació:** La dificultat per compaginar el projecte amb obligacions laborals i personals va generar llargues pauses. Cada represa implicava un "peatge cognitiu"per recuperar el context, la qual cosa va alentir el progrés de manera considerable.

2. **Subestimació de la corba d'aprenentatge:** La decisió d'utilitzar tecnologies noves per a mi, com Tauri i Rust, va ser enriquidora però va requerir un temps d'aprenentatge molt superior al previst, retardant el desenvolupament del client d'escriptori.

3. **Reptes tècnics imprevistos:** El descobriment d'un error de disseny en l'arquitectura inicial del backend, que centralitzava massa responsabilitats al servei FileManagement, va obligar a una refactorització important que va consumir un temps valuós no planificat.

4. **Manca d'eines de gestió formal:** La gestió de tasques mitjançant una llibreta, en lloc d'una eina digital com un tauler Kanban, va dificultar la priorització i la visibilitat de l'estat real del projecte.

11.3 Discussió crítica dels resultats

Amb la perspectiva que dona haver finalitzat el projecte, és interessant reflexionar sobre algunes de les decisions tècniques clau. L'elecció d'una **arquitectura de microserveis** va ser, sens dubte, una decisió correcta des del punt de vista de l'aprenentatge. Em va forçar a pensar en termes de sistemes distribuïts i a aplicar principis de disseny de programari robustos. No obstant això, per a un únic desenvolupador, aquesta arquitectura introdueix una sobrecàrrega de gestió (múltiples serveis, comunicació, desplegament).

La tria de **Tauri i Rust** per al client d'escriptori il·lustra perfectament el compromís entre l'objectiu de rendiment i el cost de desenvolupament. El resultat final és una aplicació nativa, lleugera i eficient, molt superior en consum de recursos a una alternativa basada en Electron. Tanmateix, el preu a pagar va ser una corba d'aprenentatge molt pronunciada que va ser una de les principals causes dels retards.

Finalment, cal contextualitzar aquest projecte en el panorama de solucions existents. Com vaig admetre al Capítol 1, en iniciar el desenvolupament coneixia l'existència d'alternatives de codi obert madures com **Nextcloud** o **Seafile**. Aquestes plataformes ofereixen un conjunt de funcionalitats molt més ampli i tenen una comunitat consolidada. Aquest projecte no pretén competir directament amb elles en funcionalitats, sinó explorar una aproximació arquitectònica diferent. El seu principal valor diferencial rau en la seva arquitectura moderna basada en microserveis, que pot resultar més familiar, mantenible o extensible per a desenvolupadors que treballin amb aquest paradigma, oferint una base diferent per construir solucions personalitzades.

11.4 Conclusions finals i lliçons apreses

En conclusió, aquest Treball de Fi de Grau ha culminat amb la creació d'un sistema de programari complex i funcional que compleix la majoria dels seus objectius i serveix com una excel·lent prova de concepte. Tot i no estar llest per a producció, estableix una base arquitectònica sòlida sobre la qual es pot continuar construint.

Més enllà del producte final, el valor més gran del projecte ha estat el propi procés. Les lliçons apreses són nombroses i aplicables a la meva carrera professional:

- La **continuïtat és tan important com la quantitat d'hores** invertides. Les interrupcions tenen un cost ocult en la productivitat.
- Una **planificació realista** ha de tenir en compte no només les tasques, sinó també les corbes d'aprenentatge i els imprevistos.
- Les **eines de gestió de projectes i mètriques objectives** no són burocràcia, sinó instruments essencials per mantenir el rumb.

- La integració de **proves automatitzades des de l'inici** és indispensable per garantir la qualitat i evitar regressions en projectes de certa mida.

En definitiva, el viatge ha estat tan o més important que la destinació, proporcionant-me una experiència pràctica inavaluable en la construcció de sistemes de programari moderns.

Capítol 12

Treball futur i línies de millora

Aquest capítol final recull i organitza les diverses línies de treball que han quedat pendents, així com les possibles millores identificades al llarg del desenvolupament i la redacció d'aquesta memòria. Aquestes propostes no només busquen completar funcionalitats planificades que van ser descartades per limitacions de temps, sinó també enfortir l'arquitectura, la seguretat i la usabilitat del sistema per apropar-lo a un estat apte per a un entorn de producció real.

Les millores s'han agrupat en blocs temàtics i s'han prioritzat segons la seva importància crítica per a la seguretat, la funcionalitat bàsica i la robustesa general de la plataforma.

12.1 Millores Crítiques de Seguretat i Compliment Normatiu (Prioritat Alta)

Aquest bloc agrupa les tasques més urgents, indispensables per garantir la seguretat de les dades i un compliment normatiu més estricte, especialment amb el RGPD.

- **Implementació de Xifratge en Trànsit (HTTPS/TLS):** Com es va reconèixer als capítols 10 i 7, l'absència de xifratge en les comunicacions és la mancança de seguretat més crítica del sistema. La propera iteració ha d'implementar de manera prioritària la configuració d'un *reverse proxy* (com Nginx o Traefik) que gestioni els certificats SSL/TLS, assegurant que tota la comunicació entre els clients i el Gateway, així com la comunicació interna entre microserveis, estigui xifrada.

- **Gestió Segura de Tokens Mitjançant Cookies HttpOnly:** L'estratègia actual d'emmagatzemar els tokens JWT al `localStorage` del navegador és vulnerable a atacs XSS. És prioritari refactoritzar el sistema d'autenticació per transportar els tokens en cookies amb els atributs `HttpOnly` i `Secure`, fent-los inaccessibles des del JavaScript del client i millorant substancialment la seguretat de la sessió.

- **Sistema de Registres (Logging) Centralitzat:** La manca actual d'un sistema de logs estructurat impedeix una traçabilitat adequada de les operacions i la detecció d'anomalies. Per solucionar-ho, es proposa el disseny i la implementació d'un nou microservei dedicat exclusivament a la gestió de registres. Aquest servei s'encarregaria de rebre esdeveniments de log de la resta de microserveis,

per després agregar-los, correlacionar-los i persistir-los en un format estructurat que permeti consultar l'historial d'operacions de manera unificada.

12.2 Completar Funcionalitats del Client d'Escriptori (Prioritat Alta)

El client d'escriptori és una peça clau del projecte, i completar la seva funcionalitat és essencial per assolir els objectius inicials.

- **Sincronització dels Fitxers Compartits:** Aquesta és la funcionalitat més important que va quedar pendent (Capítols 4 i 11). La propera versió del client d'escriptori ha d'implementar la lògica necessària per detectar i sincronitzar els fitxers i carpetes de la secció Çompartits amb mi", permetent un accés transparent a aquests recursos des del sistema d'arxius local. La implementació actual del motor de sincronització ja proporciona una base sòlida i avançada des de la qual partir, fet que hauria de facilitar l'addició d'aquesta funcionalitat.

- **Sistema Avançat de Resolució de Conflictes:** El motor de sincronització actual gestiona els conflictes de manera bàsica, sobreescrivint la versió més antiga. S'ha de dissenyar i implementar un sistema més sofisticat que, en cas de conflicte (modificació simultània en local i remot), notifiqui a l'usuari i li permeti escollir quina versió conservar, crear una còpia o intentar fusionar els canvis.

12.3 Optimització del Rendiment i l'Escalabilitat (Prioritat Mitjana)

Aquestes millores busquen optimitzar l'eficiència del sistema, especialment en operacions que involucren un gran volum de dades o peticions.

- **Implementació d'Operacions per Lots (Batch Operations):** Actualment, operacions com la pujada de carpetes o la compartició de múltiples fitxers generen una petició a l'API per cada element individual. S'han de dissenyar i implementar nous endpoints al backend que acceptin operacions per lots, reduint dràsticament el nombre de crides a la xarxa i millorant el rendiment percebut per l'usuari.

- **Unificació dels Endpoints de Pujada:** Per simplificar el codi i millorar l'eficiència, l'endpoint de pujada de fitxers s'hauria d'unificar en una única solució basada en *streaming*, eliminant la duplicitat actual entre el mètode `multipart/form-data` (web) i `application/octet-stream` (escriptori).

- **Optimització de Consultes Complexes:** La càrrega de la vista Çompartit amb mi"actualment genera múltiples crides a altres serveis per enriquir les dades. Aquest procés s'ha d'optimitzar, ja sigui mitjançant una única crida que agregui les dades al backend o implementant un sistema de memòria cache per a les dades d'usuari i permisos, reduint la latència.

- **Externalització de l'Estat de WebSockets:** Per permetre l'escalat horitzontal del `SyncService`, l'estat de les connexions WebSocket (actualment en memòria) s'hauria d'externalitzar a un magatzem de dades compartit.

12.4 Millores en la Gestió i el Desplegament (Prioritat Mitjana)

Aquestes propostes se centren a millorar el cicle de vida del desenvolupament, les proves i el desplegament de la plataforma.

- **Implementació de Proves Automatitzades:** La manca de tests automatitzats és un dels principals punts febles del projecte actual (Capítol 9). És crucial desenvolupar una estratègia de proves completa que inclogui tests unitaris, d'integració i E2E (End-to-End) per garantir la qualitat del codi, detectar regressions de manera primerenca i facilitar futures refactoritzacions.

- **Integració i Desplegament Continus (CI/CD):** S'ha de configurar un pipeline de CI/CD utilitzant eines com GitHub Actions. Aquest pipeline hauria d'automatitzar l'execució de les proves, la construcció de les imatges de Docker i la generació dels instal·ladors multiplataforma del client d'escriptori, publicant-los a GitHub Releases per a una distribució senzilla.

- **Suport per a Kubernetes:** Per a entorns de producció més exigents, es preveu la creació de fitxers de configuració de Helm que facilitin el desplegament, l'escalat i la gestió del sistema en un clúster de Kubernetes.

- **Flexibilització de la Configuració:** S'ha de permetre la configuració dels ports dels serveis a través de variables d'entorn, oferint més flexibilitat en el desplegament.

12.5 Millores Funcionals i d'Usabilitat (Prioritat Baixa)

Finalment, aquest bloc recull un conjunt de millores que, tot i no ser crítiques, enriquirien l'experiència d'usuari i afegirien valor a la plataforma.

- **Revisió del Principi de Minimització de Dades:** Com es va mencionar al Capítol 10, s'analitzarà la possibilitat d'eliminar la recollida del nom i els cognoms de l'usuari durant el registre per alinear el projecte de forma més estricta amb el principi de minimització de dades del RGPD.

- **Feedback Visual en Operacions d'Arrossegar i Deixar Anar:** Millorar la representació visual quan s'arrosseguen múltiples elements, mostrant una pila o un comptador en lloc d'un únic element.

- **Configuració Avançada del Client d'Escriptori:** Afegir opcions per limitar l'ample de banda, configurar el nombre de transferències concurrents i personalitzar les notificacions.

- **Endpoint de Validació del Servidor:** Crear un endpoint específic (p. ex., `/api/v1/ping`) per a una validació més robusta del servidor per part del client d'escriptori.

12.6 Reflexió sobre el Futur del Projecte

Com vaig reflexionar al Capítol 1, abans d'embarcar-se en la implementació d'aquestes millores, seria prudent realitzar una anàlisi exhaustiva de les alternatives de codi obert existents, com Nextcloud o Seafile. Aquesta anàlisi permetria determinar si el projecte, un cop madurat, pot oferir un valor diferencial clar —ja sigui per la seva arquitectura, el seu rendiment o la seva filosofia— o si els esforços de la comunitat estarien millor invertits contribuint a projectes ja consolidats.

En conclusió, tot i que el sistema actual és una prova de concepte funcional, el camí per convertir-lo en una solució de producció robusta, segura i completa és llarg. Les línies de treball futur aquí exposades tracen un full de ruta clar per a aquesta evolució, abordant des de les mancances més crítiques fins a les millores que enriquirien l'experiència final de l'usuari.

Capítol 13

Manual d'instal·lació

Aquest capítol final serveix com a guia pràctica per a la instal·lació i posada en marxa de tot el sistema en un entorn local. Lluny de ser un manual exhaustiu, pretén descriure els passos essencials que qualsevol usuari tècnic —o un company valent— hauria de seguir per replicar l'entorn de desenvolupament. Per simplificar al màxim aquest procés, s'han creat dos scripts d'automatització, un per a sistemes basats en Unix ('setup.sh') i un altre per a Windows ('setup.ps1'), que encapsulen tota la complexitat de la compilació, configuració i desplegament dels diferents components que conformen l'arquitectura descrita al Capítol 8.

13.1 Requisits previs

Abans d'executar l'script d'instal·lació, és altament recomanable assegurar-se de tenir instal·lades les següents dependències a l'equip. Tot i que els scripts intentaran instal·lar-les automàticament si no les detecten, una instal·lació manual prèvia pot evitar possibles problemes de permisos o de configuració de l'entorn, especialment en sistemes Windows.

- **Git**: Per clonar el repositori del projecte.
- **Docker i Docker Compose**: Indispensable per orquestrar i executar l'ecosistema de microserveis en contenidors aïllats.
- **Java (JDK 17 o superior)**: Necessari per compilar els microserveis del backend desenvolupats amb Spring Boot.
- **Node.js (v18 o superior) i pnpm**: Per construir la interfície web desenvolupada amb Svelte. L'script instal·larà 'pnpm' globalment si no el troba.
- **Rust i Cargo**: Requerits per compilar l'aplicació d'escriptori desenvolupada amb Tauri.

13.2 Procés d'instal·lació automatitzada

El procés d'instal·lació s'ha dissenyat per ser el més senzill possible, realitzant-se gairebé íntegrament amb una única comanda.

13.2.1 Obtenció del codi font

El primer pas és descarregar el codi font del projecte clonant el repositori amb Git i accedir a la carpeta arrel:

```
git clone ¡URL˙DEL˙REPOSITORI¿
cd ¡NOM˙DE˙LA˙CARPETA˙DEL˙PROJECTE¿
```

13.2.2 Execució de l'script

Un cop a la carpeta arrel, s'ha d'executar l'script corresponent al sistema operatiu amb l'indicador d'instal·lació (-i). És important executar-lo amb privilegis d'administrador, ja que instal·larà dependències i gestionarà contenidors de Docker.

Per a Linux/macOS:

```
sudo bash setup.sh -i
```

Per a Windows (des d'una terminal PowerShell com a Administrador):

```
powershell -ExecutionPolicy Bypass -File .""setup.ps1 -i
```

Mode de dades de prova: Per facilitar les proves de l'aplicació, es pot utilitzar l'opció -t juntament amb -i per iniciar el sistema amb dades preconfigurades (usuaris, carpetes, fitxers i comparticions de mostra):

```
# Linux/macOS
sudo bash setup.sh -i -t

# Windows
powershell -ExecutionPolicy Bypass -File .""setup.ps1 -i -t
```

Aquesta comanda iniciarà un procés completament automatitzat que realitzarà les següents accions:

1. **Configuració de credencials**: L'script utilitzarà un conjunt de credencials per defecte per als serveis de base de dades (PostgreSQL) i missatgeria (RabbitMQ). Opcionalment, es poden personalitzar passant paràmetres addicionals a la comanda. Totes les credencials, ja siguin les predeterminades o les personalitzades, es desaran en un fitxer '.env' a l'arrel del projecte. És de vital importància no eliminar aquest fitxer, ja que garanteix que les mateixes credencials es reutilitzin en futures execucions.

2. **Instal·lació de dependències**: Comprovarà si les eines necessàries (Docker, Node, Rust) estan instal·lades i, si no ho estan, intentarà instal·lar-les.

3. **Compilació dels projectes**:

   - Compilarà tots els microserveis del backend (Java) amb Maven.

   - Construirà l'aplicació web (Svelte).

- Compilarà l'aplicació d'escriptori (Tauri) i generarà els instal·ladors natius per al sistema operatiu actual, desant-los a la carpeta '/installers' a l'arrel del projecte.

4. **Aixecament dels serveis**: Utilitzarà 'docker-compose' per aixecar tota l'arquitectura de serveis del backend en contenidors.

5. **Creació del Superadministrador**: Un cop els serveis estiguin actius, l'script sol·licitarà per terminal la creació d'un usuari i contrasenya per al primer compte de Superadministrador, que tindrà control total sobre la plataforma.

Quan s'utilitza l'opció -t, a més dels passos anteriors, el sistema es configura amb dades de prova predefinides que inclouen usuaris de mostra, carpetes, fitxers i relacions de compartició, permetent provar immediatament totes les funcionalitats sense haver de crear manualment el contingut.

Nota sobre les dades de prova: El mode de proves carrega un conjunt d'usuaris fictius amb carpetes, fitxers i configuracions de compartició preestablertes. Això permet avaluar funcionalitats com la gestió d'arxius, el control d'accés, la compartició entre usuaris i la sincronització sense necessitat de configurar manualment un entorn de proves. Els usuaris de prova creats tenen diferents nivells de permisos per permetre provar tots els aspectes del sistema de control d'accés.

Un cop finalitzat el procés, tot el sistema estarà completament operatiu.

13.3 Gestió del sistema

Els mateixos scripts permeten realitzar altres tasques de gestió sobre l'entorn desplegat.

- **Iniciar el sistema**: Si el sistema està aturat, es pot tornar a iniciar amb:

  ```
  # Linux/macOS
  sudo bash setup.sh -u
  # Windows
  powershell -File .""setup.ps1 -u
  ```

- **Aturar el sistema**: Per aturar tots els contenidors:

  ```
  # Linux/macOS
  sudo bash setup.sh -d
  # Windows
  powershell -File .""setup.ps1 -d
  ```

- **Actualitzar el sistema**: Per reconstruir les imatges dels serveis després de canvis en el codi i reiniciar els contenidors:

  ```
  # Linux/macOS
  sudo bash setup.sh -b
  # Windows
  powershell -File .""setup.ps1 -b
  ```

- **Eliminar totes les dades**: Aquesta és una operació destructiva que atura el sistema i elimina permanentment totes les dades (volums de Docker, fitxers emmagatzemats). S'ha d'utilitzar amb precaució.

  ```
  # Linux/macOS
  sudo bash setup.sh -r
  # Windows
  powershell -File .""setup.ps1 -r
  ```

13.4 Accés a les aplicacions

Un cop el sistema està en funcionament, es pot accedir als diferents components:

- **Client Web**: La interfície web principal estarà disponible a través del navegador a l'adreça http://localhost:8080.

- **Client d'Escriptori**: S'ha d'anar a la carpeta '/installers' i executar l'instal·lador corresponent al sistema operatiu. Un cop instal·lat, es podrà iniciar l'aplicació de forma nativa.

- **Eines d'administració del backend**:

  - **RabbitMQ Management**: Per monitorar les cues de missatges, disponible a http://localhost:15672.

  - **Eureka Server**: Per visualitzar l'estat i el registre de tots els microserveis, disponible a http://localhost:8761.

  Les credencials per accedir a aquestes eines són les que es van definir durant la instal·lació i que es troben al fitxer '.env'.