POLITECNICO DI TORINO

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA

Tesi di Laurea Magistrale

Implementazione di un'architettura client ibrida per la sincronizzazione automatica di file su un cloud proprietario

Relatori
Prof. Giovanni MALNATI
Prof. Daniele APILETTI

Candidato
Luca ROMEO

Ottobre 2025

Abstract

In un'epoca in cui sempre più aziende si trovano a dover affrontare la sfida della migrazione e dell'adattamento dei propri servizi verso il cloud, l'adozione di politiche sicure ed efficienti per la gestione e sincronizzazione dei dati richiede l'utilizzo di strumenti sempre più affidabili ed efficienti.

Tuttavia, il coordinamento e la condivisione di file negli ambienti distribuiti rappresenta una sfida complessa, data l'ampia scelta delle piattaforme disponibili unita alla necessità di garantire consistenza, sicurezza e scalabilità. Le piattaforme cloud più diffuse in questo ambito, come Google Drive o Dropbox, offrono servizi managed di qualità, ma peccano di flessibilità e non sempre si adattano a contesti in cui è richiesta un'integrazione profonda con applicativi specifici.

Il caso d'uso preso in considerazione nell'ambito di questa tesi si colloca all'interno del progetto Geedi dell'azienda Edilclima S.r.l., che tra i suoi obiettivi prevedeva la realizzazione di un logbook digitale per l'archiviazione e la condivisione dei risultati prodotti dai software aziendali. L'esigenza era quella di fornire ai clienti un'infrastruttura proprietaria in grado di virtualizzare un file system personale, garantendo al tempo stesso solidità, sicurezza e integrazione con l'ecosistema applicativo esistente.

Con il seguente lavoro di tesi si è voluto dimostrare come un'architettura client ibrida per la sincronizzazione automatica di file su un'infrastruttura cloud proprietaria fosse una strada più che praticabile, in grado di rispondere a specifiche esigenze di integrazione, controllo e scalabilità.

Partendo dall'analisi di librerie e soluzioni esistenti, è stata condotta un'analisi iniziale sui CRDT (Conflict-free Replicated Data Types), presi come paradigma utile per comprendere i modelli di sincronizzazione più evoluti in ambienti distribuiti e le strategie implementative più moderne per la risoluzione dei conflitti.

La soluzione realizzata adotta un SDK modulare che funge da nucleo tecnico del progetto, gestendo la comunicazione, tramite API REST, con un backend proprietario sviluppato in Spring Boot e supportato da un database MongoDB.

Il client, realizzato in .NET, integra un SQLite locale come storage per le informazioni di sincronizzazione, assieme ad un File System Watcher per il rilevamento in tempo reale delle modifiche ai file. La sicurezza è garantita dall'autenticazione secondo lo standard OAuth2 tramite Keycloak e dalla cifratura dello storage locale. Tutti i componenti server-side sono stati distribuiti all'interno di un cluster Kubernetes, assicurando portabilità e scalabilità. La scelta di sviluppare l'applicazione in .NET ha permesso di sfruttare a pieno la potenza degli ambienti nativi, fornendo al tempo stesso una user experience web-based fluida e interattiva con l'ausilio di componenti dedicati come WebView2.

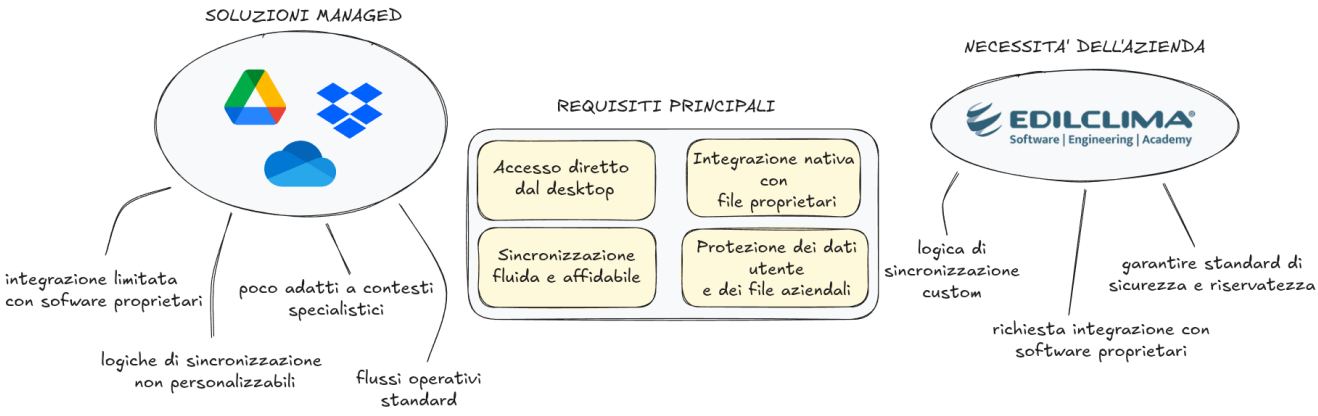
Il prototipo realizzato apre la strada a future evoluzioni del sistema verso una piattaforma cloud integrata, flessibile e scalabile all'interno dell'intero ecosistema software di Edilclima.

Sommario

La crescente diffusione di soluzioni cloud sta trasformando in maniera profonda il modo in cui le aziende gestiscono e distribuiscono i propri servizi digitali. Sempre più realtà si trovano ad affrontare la sfida di migrare parte della propria infrastruttura su piattaforme distribuite, dovendo garantire al tempo stesso sicurezza, efficienza e continuità del servizio. In questo scenario, la gestione e la sincronizzazione dei dati rappresenta uno dei nodi più critici e intricati: l'elevato numero di strumenti disponibili sul mercato, unito alla necessità di performance alte dal punto di vista delle prestazioni e della flessibilità, rende complesso individuare soluzioni capaci di integrarsi a dovere con applicativi specialistici.

Contesto e Requisiti

Le piattaforme più diffuse, come Google Drive, Dropbox o OneDrive, offrono servizi managed affidabili e ampiamente utilizzati, ma presentano limiti significativi quando si tratta di integrarli con software proprietari o di garantire un controllo granulare sulle logiche di sincronizzazione e sull'archiviazione dei dati, peccano di personalizzazione.



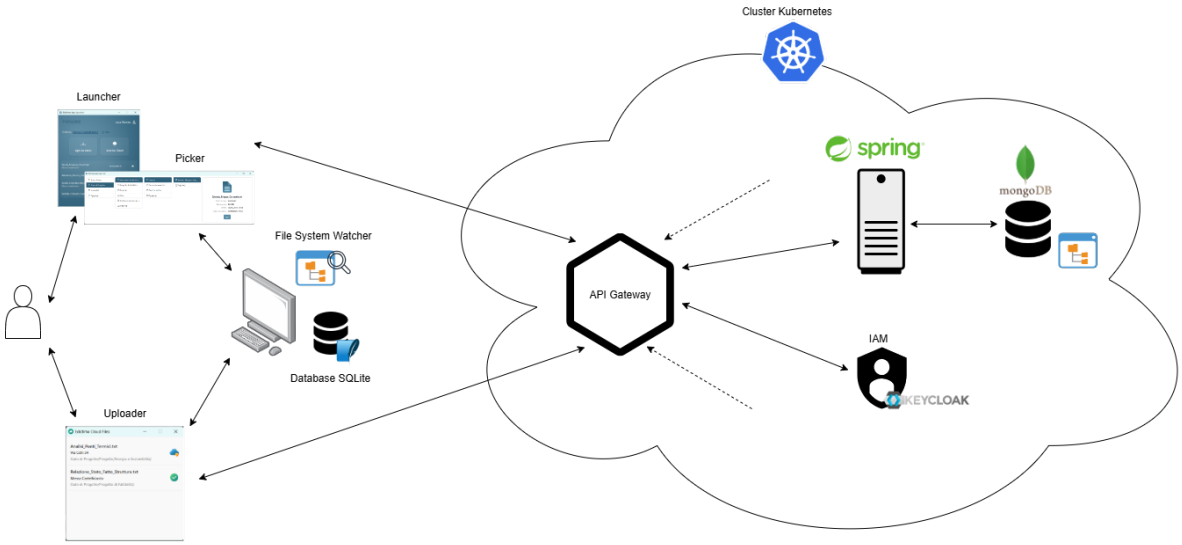
Da questa esigenza nasce questo lavoro di tesi, sviluppato nell'ambito del progetto **Geedi** dell'azienda **Edilclima S.r.l.**. Il progetto mirava a realizzare un logbook digitale per la raccolta e la condivisione dei risultati prodotti dai software tecnici dell'azienda, rivolti al settore dell'edilizia, della progettazione energetica e impiantistica. La necessità di disporre di una piattaforma proprietaria, in grado di virtualizzare un file system personale e di garantire solidità e integrazione con l'ecosistema esistente, ha costituito il punto di partenza per l'attività di ricerca e sviluppo di questa tesi.

L'elaborato si apre con una panoramica del problema della sincronizzazione dei file in ambienti distribuiti, analizzando lo stato dell'arte delle principali librerie e soluzioni disponibili. In particolare, viene approfondito il ruolo dei **CRDT (Conflict-free Replicated Data Types)** come paradigma teorico utile a comprendere i modelli più avanzati

di gestione della consistenza e le strategie più recenti di risoluzione dei conflitti. Questo inquadramento consente di individuare le criticità e le opportunità che caratterizzano le soluzioni distribuite, e di delineare le basi concettuali su cui poggia l'architettura proposta.

Il Design

La parte centrale della tesi descrive in dettaglio l'architettura sviluppata, che si fonda su un approccio **ibrido client-server**. Il cuore tecnico del sistema è rappresentato da un **SDK modulare**, progettato per gestire le operazioni di comunicazione con un backend proprietario attraverso **API REST**. Il backend è stato realizzato in **Spring Boot** e supportato da un database **MongoDB**, mentre il client è stato sviluppato in .NET. Quest'ultimo integra un **SQLite locale** per la memorizzazione delle informazioni di sincronizzazione e sfrutta un **File System Watcher** per rilevare in tempo reale eventuali modifiche ai file.



Dal punto di vista della sicurezza, l'infrastruttura implementa meccanismi di autenticazione basati sullo standard **OAuth2**, attraverso l'Identity and Access Management **Keycloak**, e garantisce la cifratura dei dati conservati in locale. La scelta tecnologica di .NET ha reso possibile sfruttare le potenzialità degli ambienti nativi, mantenendo al tempo stesso una user experience moderna e fluida tramite un'interfaccia web-based realizzata tramite **WebView2**.

Sul piano dell'infrastruttura, tutti i componenti server-side sono stati distribuiti in un cluster **Kubernetes** di proprietà di Edilclima, assicurando portabilità, scalabilità e resilienza del sistema. Questa impostazione ha reso possibile ottenere una piattaforma non solo funzionale al caso d'uso specifico del progetto Geedi, ma anche estendibile ad altri scenari applicativi dell'azienda.

Conclusioni

Il lavoro si conclude con la presentazione del **prototipo sviluppato**, che dimostra la fattibilità di una soluzione proprietaria per la sincronizzazione e la condivisione dei file in un contesto professionale. Il sistema realizzato non solo risponde alle esigenze di integrazione, controllo e sicurezza espresse dal committente, ma apre anche prospettive per ulteriori sviluppi. Tra questi, l'estensione del modello verso un ecosistema cloud integrato

e trasversale, capace di supportare in maniera unificata i diversi software dell'azienda, rappresenta la linea di evoluzione più promettente.

Con questa tesi si intende dimostrare come l'implementazione di un'architettura client ibrida, supportata da tecnologie moderne e distribuita su un'infrastruttura cloud proprietaria, possa costituire una soluzione efficace e scalabile alle sfide poste dalla sincronizzazione dei file in ambienti distribuiti, ponendo le basi per future innovazioni.

Indice

Elenco delle figure

Glossario

Keycloak Un server open-source per la gestione dell'identità e degli accessi (Identity and Access Management), sviluppato da Red Hat. Supporta autenticazione, autorizzazione, single sign-on (SSO) e integrazione con protocolli come OAuth2 e OpenID Connect.
Documentazione ufficiale: `https://www.keycloak.org/`.

MongoDB Un database NoSQL orientato ai documenti che memorizza i dati in formato JSON-like (BSON). Offre elevata scalabilità, flessibilità e supporta operazioni distribuite per applicazioni moderne.
Sito ufficiale: `https://www.mongodb.com/`.

OAuth 2.0 Un protocollo standard di autorizzazione che consente alle applicazioni di ottenere accesso limitato alle risorse di un altro servizio senza condividere le credenziali dell'utente. È ampiamente utilizzato per la protezione delle API e l'autenticazione basata su token.
Documentazione ufficiale: `https://datatracker.ietf.org/doc/html/rfc6749/`.

IAM Identity and Access Management, un insieme di processi e tecnologie per la gestione delle identità digitali e dei privilegi di accesso alle risorse, garantendo sicurezza e conformità nelle architetture distribuite..

JSON JavaScript Object Notation, un formato di interscambio dati leggero basato su testo, ampiamente utilizzato per la comunicazione tra client e server in applicazioni web e API REST.
Documentazione ufficiale: `https://www.json.org/`.

Spring Boot Un framework Java che semplifica lo sviluppo di applicazioni standalone e basate su microservizi. Riduce la necessità di configurazioni manuali e supporta l'integrazione con database, sicurezza e deployment cloud-native.
Documentazione ufficiale: `https://spring.io/projects/spring-boot`.

JWT JSON Web Token, uno standard aperto per la creazione di token di accesso firmati digitalmente. È comunemente utilizzato per autenticazione e autorizzazione in applicazioni web e API.
Specifica ufficiale: `https://datatracker.ietf.org/doc/html/rfc7519`.

Kotlin Un linguaggio di programmazione moderno, conciso e sicuro, sviluppato da JetBrains. Compatibile con la JVM, viene utilizzato per applicazioni server-side, Android e multiplatform.
Documentazione ufficiale: `https://kotlinlang.org/`.

Kubernetes Una piattaforma open-source per l'orchestrazione di container che automatizza la distribuzione, la scalabilità e la gestione di applicazioni containerizzate in cluster.
Sito ufficiale: `https://kubernetes.io/`.

CRDT Conflict-free Replicated Data Types, una famiglia di strutture dati distribuite che garantiscono la consistenza eventuale tra repliche senza necessità di coordinamento esplicito, permettendo la risoluzione automatica dei conflitti.
Introduzione: `https://crdt.tech/`.

SDK Software Development Kit, un insieme di strumenti, librerie e documentazione che consentono agli sviluppatori di creare applicazioni integrate con specifiche piattaforme o servizi..

REST API Representational State Transfer, un'architettura per la progettazione di servizi web basati su HTTP. Le API REST permettono la comunicazione tra client e server tramite operazioni standard come GET, POST, PUT e DELETE.
Documentazione introduttiva: `https://restfulapi.net/`.

.NET Un framework di sviluppo creato da Microsoft che consente la creazione di applicazioni multipiattaforma. Supporta diversi linguaggi di programmazione e include librerie per lo sviluppo desktop, web e cloud.
Sito ufficiale: `https://dotnet.microsoft.com/`.

SQLite Un database relazionale leggero e integrato, che memorizza i dati in un singolo file locale. È spesso utilizzato in applicazioni desktop e mobili per la gestione di dati persistenti senza la necessità di un server dedicato.
Sito ufficiale: `https://www.sqlite.org/`.

File System Watcher Un componente software che monitora i cambiamenti nel file system in tempo reale, come modifiche, creazioni o eliminazioni di file e directory, e genera eventi in risposta..

WebView2 Un controllo di Microsoft basato su Chromium che consente di incorporare contenuti web all'interno di applicazioni desktop .NET o C++. Facilita lo sviluppo di interfacce web-based integrate in ambienti nativi.
Documentazione ufficiale: `https://learn.microsoft.com/en-us/microsoft-edge/webview2/`.

Capitolo 1

Introduzione

Essendo capofila e coordinatrice dei partner coinvolti all'interno del progetto **GEEDI** (Gestione Energetica degli Edifici attraverso processi di Data analysis e building Information modeling, bando finanziato dall'Unione Europea con i fondi del PNRR), Edilclima aveva un ruolo centrale nella creazione di una piattaforma di servizi cloud che offrisse supporto nel processo di pianificazione e riqualificazione energetica nel settore degli edifici, al fine di promuoverne lo sviluppo sostenibile. Dato l'impiego di strumenti tecnici specifici, e collezionando e archiviando file di diversa natura aggiornabili nel tempo, uno dei punti cardine dell'intero progetto era lo sviluppo di un vero e proprio registro digitale legato ai singoli progetti ed edifici, in modo da avere uno strumento sempre attivo e accessibile a supporto di tutto il parco clienti che Edilclima deve gestire.

Questa volontà è del tutto condivisibile vista la criticità che rappresenta la gestione dei file di progetto all'interno del lavoro tecnico-professionale in ambito edilizio. In particolare, la sincronizzazione, condivisione e versioning dei file generati dai software tecnici si presta poco a servizi e soluzioni cloud più diffuse come Google Drive o Dropbox che, seppur molto comodi per l'utente generico, rendono meno in contesti dove viene richiesta un'integrazione trasparente con ambienti desktop e software proprietari specializzati, c'era la necessità di offrire un'esperienza di condivisione realmente integrata, sicura e coerente con le specificità dei loro prodotti.

Sebbene sia stata realizzata con successo una versione web del digital logbook, l'azienda voleva esplorare le possibilità di espandere il concetto relativo al collezionamento e alla sincronizzazione dei file utente generati dai loro applicativi in forma più trasversale, fornendo una piattaforma desktop che potesse integrarsi in maniera fluida con tutta la suite di applicativi che Edilclima ad oggi offre.

Con il seguente lavoro di tesi si vuole presentare e dimostrare l'efficacia di un'architettura client ibrida per la sincronizzazione automatica di file utente su un cloud proprietario, tramite un'applicazione desktop in grado di gestire in maniera asincrona operazioni di download e upload in background su una qualsiasi macchina Windows. Come base modulare e tecnica di tutto il progetto, è stato realizzato una libreria SDK, pensata per fungere da base riutilizzabile per future integrazioni e per facilitare l'approccio al paradigma cloud anche in altri contesti applicativi.

Il prototipo realizzato è stato testato con successo su file reali, anche complessi, generati dal software EC700 di proprietà della stessa Edilclima, e si pone come primo passo verso la realizzazione di un file system personale e virtualizzato per l'intero bacino di utenza Edilclima.

La struttura del seguente lavoro di tesi seguirà uno schema ben definito, cominciando

dall'analisi dello stato dell'arte di soluzioni alternative che sono state vagliate inizialmente e i criteri di valutazioni adottati, passando poi ad una descrizione più tecnica dell'architettura dell'intero sistema, con particolare enfasi sulla composizione dello SDK. Verrà poi analizzata più in dettaglio l'implementazione dei due attori principali, il launcher e l'uploader in background, per terminare poi riassumendo i risultati ottenuti e proponendo possibili sviluppi futuri del progetto.

1.1 Stato dell'arte

Prima di approfondire la struttura di questo progetto, è necessario fare una breve panoramica sullo stato della tecnologia per quello che riguarda la gestione della consistenza dei dati in sistemi distribuiti.

Storicamente in questo campo si è sempre preferito garantire una consistenza forte, garantendo che tutte le repliche concordino su uno stesso ordine globale di operazioni, privilegiando la correttezza e la coerenza delle informazioni, ma introducendo costi elevati in termini di latenza e scalabilità, rendendo questi approcci meno adatti a scenari con un numero elevato di nodi o ad ambienti che richiedevano bassa latenza.

Parallelamente, negli ultimi anni si sono fatti strada modelli di replicazione così detti multi-leader o leaderless, adottati da database come Cassandra o Riak, che si concentrano più sulla disponibilità del sistema e la sua capacità di mantenere i propri servizi attivi anche in presenza di problematiche legate alle partizioni di rete, guasti che rischiano di creare incoerenza tra le varie "isole" composte dalle repliche sparse per la rete.

Questo approccio, pur aumentando il throughput complessivo del sistema e riducendo i colli di bottiglia dovuti alla centralizzazione, introduce una sfida fondamentale: la gestione dei **conflitti di aggiornamento**. Un conflitto si verifica quando due o più repliche aggiornano simultaneamente lo stesso dato senza avere visibilità immediata delle modifiche altrui. La conseguenza è che, senza un meccanismo di coordinamento, i nodi possono divergere, arrivando a mantenere copie incoerenti dello stesso oggetto.

Le strategie principali per risolverli variano molto in termini di complessità ed efficacia; qui vengono citate le quattro più riconosciute a livello attuale:

- **Lock e WebSocket**: una porzione di dato viene bloccata quando un utente la modifica. Un approccio del genere riduce drasticamente la collaborazione, sebbene varianti più avanzate utilizzino anche forme di lock temporanei; la concorrenza reale non viene lontanamente presa in considerazione.
- **Last Write Wins (LWW)**: la replica conserva l'aggiornamento più recente utilizzando un timestamp. Questo approccio manca di affidabilità in quanto, soprattutto in un ambiente distribuito, la sincronizzazione temporale è praticamente impossibile da ottenere, rischiando quindi di scartare aggiornamenti validi.
- **Siblings/Versioning**: vengono mantenute dal sistema più versioni dello stesso dato, delegando all'utente finale o allo sviluppatore il compito di risolvere i conflitti manualmente. Qui si evita la perdita di dati ma aumenta la complessità, compromettendo anche la user experience.
- **Merge automatico**: il sistema stesso conosce la logica di risoluzione dei conflitti e ha l'obiettivo di unificare gli aggiornamenti concorrenti utilizzando complesse funzioni di merge.

Ed è proprio in questo contesto che emergono i **Conflict-free Replicated Data Types (CRDTs)**, una proposta solida dal punto di vista teorico capace di gestire automaticamente i conflitti e garantire la convergenza dello stato del sistema anche in assenza di coordinamento centralizzato, affidandosi ai vari peer e nodi che compongono la rete, già validato da diversi sistemi reali.

1.2 CRDT: Conflict-free Replicated Data Types

Un *CRDT* è una struttura dati replicabile in più nodi, capace di garantire convergenza e coerenza **eventuale**, ossia assicurando che tutte le repliche raggiungano lo stesso stato senza la necessità di un coordinamento centralizzato e senza perdita di informazioni, anche in presenza di aggiornamenti concorrenti.

La caratteristica che distingue i CRDT dalle altre tipologie di strutture dati è la loro capacità di fondere in maniera **deterministica** più copie divergenti dello stesso dato, indipendentemente dall'ordine delle operazioni eseguite o dal ritardo con cui vengono ricevute.

Questo è reso possibile dal fatto che queste strutture dati hanno solide basi teoriche e matematiche, fondandosi sulla definizione di un **ordine parziale** sullo spazio degli stati. È necessario che, per qualunque insieme di stati, esista un **Least Upper Bound (LUB)**, ossia un elemento che rappresenti il minimo stato maggiore o uguale a tutti quelli considerati.

La funzione di merge di un CRDT non è altro che il calcolo di questo LUB:

$$\text{merge}(a, b) = \text{lub}\{a, b\}$$

Grazie a questa definizione, l'operazione di merge possiede le seguenti proprietà algebriche fondamentali:

- **Idempotenza**: applicare più volte lo stesso merge non altera il risultato.
- **Commutatività**: l'ordine del merge è irrilevante, ovvero

$$\text{merge}(a, b) = \text{merge}(b, a)$$

- **Associatività**: la fusione può essere applicata a gruppi di stati in qualunque sequenza, ovvero

$$\text{merge}(a, \text{merge}(b, c)) = \text{merge}(\text{merge}(a, b), c)$$

Un aspetto cruciale nei sistemi distribuiti è garantire l'**idempotenza** delle operazioni, dato che un aggiornamento può essere ritrasmesso più volte. Per evitare duplicazioni, è necessario associare a ciascuna operazione un identificatore unico. Le due soluzioni più diffuse sono:

- **UUID univoco**: semplice da generare, ma introduce overhead di gestione.
- **Version vector**: ogni update è associato al nodo che l'ha prodotto e al numero progressivo delle sue operazioni, distribuendo il controllo ai nodi senza necessità di coordinamento centralizzato.

Un ulteriore aspetto fondamentale è la gestione del **contesto causale**, ossia l'informazione su quale stato è stato letto prima della modifica. Questo permette di evitare la perdita involontaria di aggiornamenti concorrenti. Il meccanismo più comune è quello dei

vector clocks, che tracciano la conoscenza di ciascun nodo e consentono di stabilire se un'operazione è nuova, duplicata o già incorporata nello stato corrente.

Grazie a queste proprietà, i CRDT non necessitano di meccanismi di coordinamento globale che tipicamente riducono la scalabilità e aumentano i ritardi: *ogni replica può evolvere localmente*, è sufficiente che le informazioni sugli aggiornamenti vengano scambiate periodicamente affinché lo stato finale converga in maniera univoca in tutti i nodi. Per questo motivo i CRDT sono particolarmente adatti a protocolli come il **Gossip Protocol**, ampiamente usato in sistemi distribuiti.

1.2.1 Tipologie di CRDT ed esempi applicativi

In letteratura si individuano due grandi categorie di CRDT:

1. **CvRDT (Convergent o State-based):** basati sullo scambio completo dello stato. Ogni nodo invia agli altri la versione aggiornata della propria copia del dato; la funzione di merge garantisce la convergenza. Esempio: *G-Set (Grow-only Set)*, dove il merge corrisponde all'unione insiemistica.
2. **CmRDT (Commutative o Operation-based):** basati sull'invio delle sole operazioni elementari propagate ai vari peer. Ogni operazione deve raggiungere *eventualmente* tutte le repliche. L'ordinamento globale non è richiesto; duplicati gestiti tramite idempotenza.

Una variante recente è rappresentata dai **delta-based CRDTs**, che trasmettono soltanto la differenza incrementale dello stato, riducendo il traffico di rete senza sacrificare la resilienza.

Esempi concreti includono:

- *G-Counter* (contatore solo incrementale) e *PN-Counter* (incremento e decremento)
- *2P-Set* (Two-Phase Set), che permette aggiunta e rimozione di elementi
- *Text CRDT*, utilizzati nell'editing collaborativo, rappresentando documenti come sequenze di caratteri ordinati tramite identificatori univoci

1.2.2 Operational Transform

Una soluzione storicamente popolare, soprattutto nell'editing collaborativo, è l'*Operational Transform*, un algoritmo che permette di trasformare le operazioni concorrenti in modo corretto sul documento condiviso, ma richiede un server centralizzato per ordinare e applicare le trasformazioni. Sebbene presenti limiti in termini di scalabilità e resilienza, ha reso possibile la collaborazione in tempo reale, ad esempio in Google Docs.

1.2.3 Campi di applicazione

I CRDT trovano applicazione in numerosi scenari:

- **Collaborative editing** (Google Docs, Figma, VS Code Live Share, Office 365)
- **Sistemi di messaggistica**, dove è necessario preservare l'ordine dei messaggi
- **Database distribuiti leaderless** come Riak o Redis
- **Calendari condivisi** e applicazioni multi-utente in tempo reale
- **Editing offline/online**, dove gli aggiornamenti locali si integrano automaticamente

Nei sistemi di collaborative editing, i CRDT testuali mantengono la corretta sequenza dei caratteri senza coordinamento centralizzato. Sfide pratiche includono l'**interleaving**, quando due utenti scrivono contemporaneamente nella stessa posizione.

Alcune considerazioni pratiche:

- La progettazione di strutture per dati complessi (file di testo o binari) può introdurre overhead significativi
- L'integrazione in sistemi esistenti richiede ripensamento architetturale
- Ideali per dati strutturati (set, contatori, liste), ma complesso su file binari modificati in parallelo

1.2.4 Riflessioni finali

I CRDT rappresentano lo **stato dell'arte** nella gestione dei conflitti in sistemi distribuiti ad alta disponibilità. Consentono convergenza automatica senza coordinamento centralizzato, garantendo che nessun aggiornamento venga perso.

Tuttavia, l'adozione pratica dipende dal contesto:

- In **collaborative editing** o **database distribuiti**, i CRDT sono ideali
- In scenari di sincronizzazione di file arbitrari, politiche più semplici possono risultare più pratiche

Nel contesto di questa tesi, i CRDT offrono una soluzione teorica elegante per le repliche multi-leader, ma poco adatta al contesto cloud di Edilclima. Si è quindi scelto un approccio pragmatico basato su **Last Write Wins** combinato con **versionamento dei file**, più semplice ed efficace per il prototipo.

Capitolo 2

Architettura del sistema

Prima di approfondire la composizione architetturale del progetto, verrà fatta una panoramica sulle opzioni di design che sono state vagliate in principio, una rassegna sulle idee e considerazioni che hanno portato alla versione attuale dell'applicazione.

2.1 Considerazioni iniziali sul design architetturale

L'idea iniziale era quella di fornire ad Edilclima un sistema in cui ogni utente avesse a disposizione una cartella che potesse sincronizzarsi con una qualche sorgente remota, in modo che le applicazioni Edilclima potessero lavorare con i file su disco senza soluzione di continuità, garantendo ad ogni utente che avesse effettuato l'accesso a questa cartella di poter lavorare con l'ultima versione di un determinato file.

Sono nate considerazioni differenti su come affrontare questo problema, basate principalmente su tre approcci che ora andremo a raccontare.

CBFS Connect™

Callback Filesystem™ Connect viene presentata come la leading SDK per creare e gestire virtual filesystem per qualsiasi tipo di storage.

Permette di presentare qualsiasi sorgente dati come un file system completo, che si tratti di file locali o remoti, contenuti generati dinamicamente, archiviazioni cloud, record di database. Soluzione completamente managed, permette di esporre i propri dati come fossero all'interno di un driver fisico, con all'interno cartelle e file, semplificando la visualizzazione e l'interazione da parte degli utenti, offrendo ad altre applicazioni la possibilità di accedervi e manipolarli tramite API standard per i file.

Offre un approccio più sicuro e flessibile allo sviluppo del kernel, consentendo una maggiore sicurezza e stabilità, pur offrendo i vantaggi dell'accesso al file system a livello kernel. Isolando il codice user mode da quello kernel, CBFS Connect riduce i rischi di vulnerabilità che possono essere sfruttata dagli attaccanti. Inoltre, dato che il codice user mode non esegue codice direttamente nel kernel, è meno probabile causare crash del sistema o altre issues di stabilità.

Dalle informazioni fornite direttamente dal team di CBFS Connect inoltre, emerge come il prodotto sia pensato per una grande varietà di back-end. Non è vincolato ad un protocollo specifico, è infatti possibile collegarsi a servizi via REST API, a condizione che il back-end supporti almeno in parte le operazioni di lettura e scrittura random. In aggiunta:

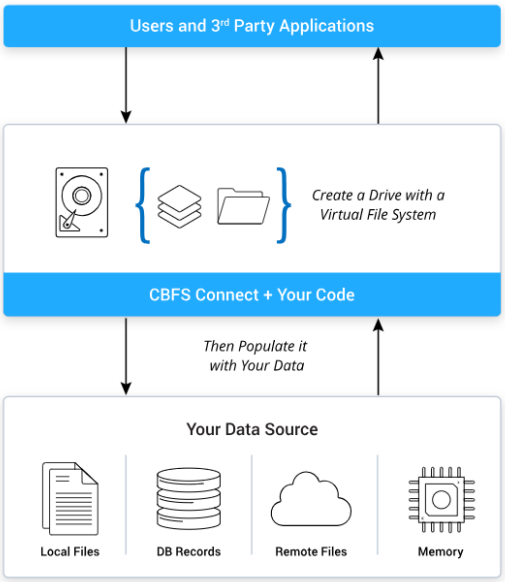


Figura 2.1: CBFS worflow

- qualora il sistema remoto permetta solo il trasferimento di file completi, il software resta comunque utilizzabile, ma viene necessario prevedere un meccanismo di caching lato client.
- per quanto riguarda la gestione della multiutenza invece, il file system virtuale può essere esposto globalmente o localmente per singolo utente (consentendo quindi la creazione di file system dedicati con contenuti differenziati per ciascun account)

Con oltre 20 anni di esperienza e la fiducia di centinaia aizende tech leader nel settore, si è ritagliato un perimetro di spessore all'interno del settore tech, i costi delle licenze .NET sono di 2,499$annuiperlaDeveloperSubscriptione$4,999 annui per la distribuzione a 100 macchine.

WinFsp™

WinFsp è un progetto open-source, il cui codice sorgente è hostato su github, che permette di creare e gestire file system personalizzati in modalità utente, senza la complessità tipica dello sviluppo kernel. E' possibile sviluppare file system che si comportano come normali unità Windows accessibili da qualsiasi applicazione, ma i dati possono in realtà risiedere su risorse remote.

In questo modo è possibile creare una piattaforma cloud completamente interna, con controllo totale sui dati e sulle policy di sicurezza. Grazie al supporto nativo delle API di Windows, gli utenti o i clienti finali interagiscono con quella che a loro sembra una nuova unità di rete, senza aggiungere complessità tramite software dedicati. Include inoltre supporto nativo per la creazione di file system personalizzati in C, C++ e .NET, in modo da adottare l'ambiente che meglio si adatta alle proprie esigenze.

I benchmark dimostrano che le prestazioni sono paragonabili o superiori a quelle dei file system nativi Windows (NTFS), garantendo affidabilità anche in scenari intensivi. Dalla wiki del repository ufficiale è possibile visionare i test effettuati confrontando due file system di riferimento WinFsp (MEMFS e NTPTFS) con NTFS in diversi scenari.

- MEMFS è un file system in-memory
- NTPTFS inoltra tutte le richieste di file system a un NTFS sottostante.

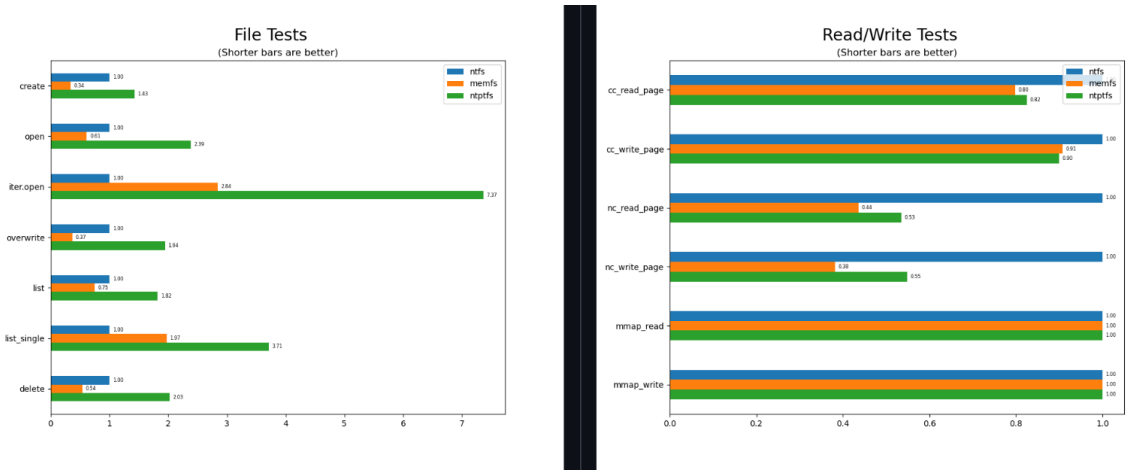


Figura 2.2: Confronto prestazioni MEMFS vs. NTPTFS

- i risultati mostrati sul grafico **File Tests** sintetizzano le prestazioni delle operazioni sul namespace dei percorsi, come creazione/eliminazione/apertura file
- il grafico **Read/Write Tests** mostra invece le prestazioni dell'IO sui file

WinFsp rappresenta una piattaforma robusta, flessibile e sicura per le aziende che desiderano costruire un servizio di cloud personale per i propri clienti. Grazie alla semplicità di sviluppo, all'elevata compatibilità con Windows e alle prestazioni garantite, costituisce una valida alternativa ai servizi cloud commerciali, offrendo al contempo pieno controllo e personalizzazione.

Come detto in precedenza, è disponibile come software open source con licenza GPLv3, per le aziende e professionisti che desiderano utilizzarlo in progetti senza vincoli è disponibile una licenza commerciale. Per soluzioni destinate a team fino a 10 sviluppatori, il costo è di 6000*unatantum*, *mentreperindividuisingoliilcostoèdi*2400.

Custom Solution

Nonostante i vantaggi offerti da soluzioni come CBFS Connect e WinFsp per la virtualizzazione di un file system condiviso, Edilclima ha manifestato l'interesse ad esplorare la realizzazione di un sistema completamente custom. Una piattaforma pienamente personalizzabile che potesse adattarsi in maniera flessibile alle esigenze specifiche dell'azienda, consentendo un'integrazione nativa coi software Edilclima già in uso.

Lo schema iniziale prevedeva una soluzione di questo tipo:

- **una desktop app** che permettesse all'utente di interfacciarsi con una cartella locale condivisa e sincronizzata con il cloud
- un cache web-server **Nginx** che contenesse i digest del file tree di riferimento, contattato preventivamente dall'utente per verificare se fosse cambiato il digest di riferimento su un determinato file

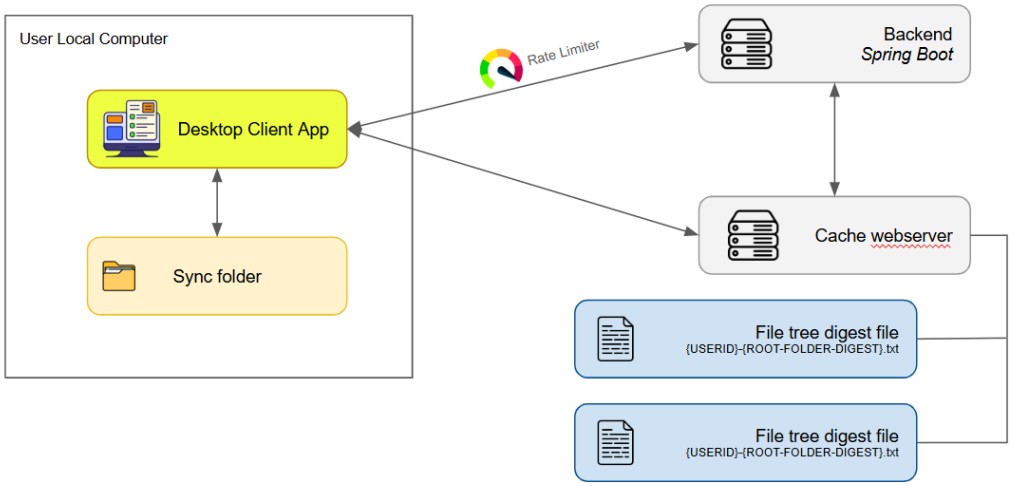


Figura 2.3: Design Iniziale

  - se il digest non era cambiato, l'utente aveva già in locale l'ultima versione del file
  - se invece era diverso, veniva interrogato il backend per ottenere la versione più aggiornata
  - con questo approccio si poteva limitare il traffico di rete sul server principale
- un back-end contattato per la gestione di upload e download
  - oltre a queste operazioni, si preoccupava anche di aggiornare i digest dei vari file tree relativi agli utenti qualora ci fossero state delle modifiche

Sebbene il prototipo iniziale venne sviluppato in Kotlin per avere un feedback immediato dall'azienda, si è poi deciso di proseguire migrando verso un'implementazione in **.NET core**, per poter sfruttare a pieno l'integrazione del framework con l'ambiente Windows, piattaforma che Edilclima prediligeva per lo sviluppo. Il back-end venne sviluppato anch'esso in maniera custom all'inizio, utilizzando il framework **Spring Boot** e scritto in **Kotlin**.

Opzioni per la sincronizzazione dei file tra applicativi Edilclima e Cloud

In principio, sono state considerate diverse modalità di integrazione tra i software nativi Edilclima e un sistema di archiviazioni/sincronizzazione in cloud. Le opzioni principali erano tre, ognuna con i suoi pro e contro

Full Offline vs. Full Online Approach

Full Offline: in questo modello, la sincronizzazione è gestita interamente da un service in background che si occupa di mantenere coerenti le versioni locali e cloud, in maniera analoga a quanto avvviene con i client di sincronizzazione tradizionali e più utilizzati (come Dropbox o OneDrive).

- features aggiuntive permetterebbero la sincronizzazione su una cartella selezionata manualmente dall'utente (interoperabile con l'Esplora Risorse di Windows), oppure su una cartella nascosta (diventa necessario un front-end dedicato)
- dando la possibilità all'utente di scegliere quali progetti o edifici sincronizzare si ottimizzerebbe il consumo di spazio sul disco locale
- è però necessario prestare attenzione in caso di ripristino da un backup locale, potrebbe sovrascrivere la versione più aggiornata sul cloud
- in più la possibilità di avere una folder condivisa su un disco di rete rende difficile gestire completamente gli hedge case, soprattutto per quanto riguarda i conflitti

Full Online: i file risiedono completamente nel cloud e l'accesso avviene tramite opzioni integrate direttamente nei software Edilclima attraverso comandi come **"Apri da Cloud.."**, per questa opzione è stato preso come esempio il software **Audacity** per la gestione dei file cloud.

- non occupa spazio sul disco locale e permette di avere sempre la versione più aggiornata
- ma si è totalmente dipendenti dalla connessioni, senza connessione il lavoro si blocca e c'è il rischio di corruzione del file in lavorazione

Questo approccio in principio sembrava poter essere il più percorribile, è stata quindi realizzata una demo molto basica per permettere di visualizzare il funzionamento che una soluzione del genere avrebbe avuto, integrata all'interno dei software Edilclima.

First Demo, Full Online Approach

La prima demo presentata aveva l'obiettivo di presentare come sarebbe stato il funzionamento di un'integrazione nativa all'interno dei software Edilclima:

- un menù molto snello con le opzioni per aprire file (da disco o dal cloud), salvare o resettare lo stato dell'applicazione
- un editor di testo, che veniva riempito col testo dei file di cui l'utente effettuava il download e che lo stesso utente poteva editare (cliccando su "Save", iniziava l'upload della versione aggiornata nel cloud)
- un footer che conteneva il path cloud o locale, con la presenza di un'icona a seconda che il file fosse remoto o all'interno del disco dell'utente

Premendo sull'opzione Open from Cloud, veniva aperto un secondo form, con la rappresentazione dell'alberatura delle directory relativa ad un edificio specifico, con la possibilità di scaricare un file all'interno di un file tree rappresentato in figura X

Il contenuto del file veniva quindi scaricato e mostrato all'utente direttamente dentro l'editor, non c'era un salvataggio locale sulla macchina dell'utente

- in questo modo si evitava di dover utilizzare il disco locale dell'utente risparmiando spazio e risorse
- ma in caso di perdita di connessione si potevano avere incongruenze e conflitti, la cui risoluzione portava ad un incremento di complessità di gestione non indifferente

Figura 2.4: Prima implementazione Launcher

La problematica più significativa rimaneva però il dover mettere le mani all'interno di tutta la gamma di software Edilclima, che essendo prodotti legacy risultavano difficili da modificare, per questo, si è pensato ad altre strategie, ma il sentore comune faceva intendere che ci si stava avvicinando verso qualcosa di effettivamente percorribile, restavano da limare e definire alcuni dettagli.

Online Hybrid, a file cloud solution

Combinare la sincronizzazione di una cartella nascosta e la presenza di un comando dedicato nel software ("File > Salva su Cloud"), oppure permettere il caricamento automatico dei file sul cloud ma effettuando il download, con eventuale aggiornamento, manualmente. Anche con queste soluzioni viene richiesto un controllo accurato dei conflitti in fase di upload, assieme alla modifica del codice interna ai software proprietari di Edilclima, scelta non considerata vantaggiosa.

L'obiettivo era quello di riuscire a creare un applicativo indipendente che interagisse coi loro software, non modificandoli dall'interno, visto anche il gran numero di applicativi che Edilclima gestisce, risultava poco pratico mettere le mani ad ognuno.

Il flusso operativo che l'applicazione avrebbe dovuto seguire era però chiaro e ben definito:

- l'applicazione doveva richiedere all'avvio la lista dei file al server remoto
- l'utente sceglie il file da aprire
- il file viene scaricato in una cartella temporanea e nascosta
- al salvataggio dopo le modifiche utente, la versione aggiornata viene inviata al server

Dopo una serie di confronti e grazie ai riscontri ottenuti dall'azienda rispetto alla demo presentata in precedenza, si è deciso di perseguire una direzione **Online Hybrid**:

- una desktop app che effettua il download di file selezionati dall'utente in una cartella nascosta e che ne permette l'editing

- un processo di caricamento in background che andasse a simulare in windows service, che si occupa di inviare le modifiche salvate dall'utente al server remoto, avendo un approccio robusto verso la perdita di connettività

In seguito ad alcuni confronti sul come organizzare una soluzione di questo tipo dal punto di vista del design, è stato possibile definire l'architettura che il sistema avrebbe dovuto presentare inizialmente, identificando dei blocchi logici che, comunicando in maniera coerente e sincronizzata, avrebbero reso possibile un'implementazione fedele alle richieste e ai requisiti che Edilclima aveva posto.

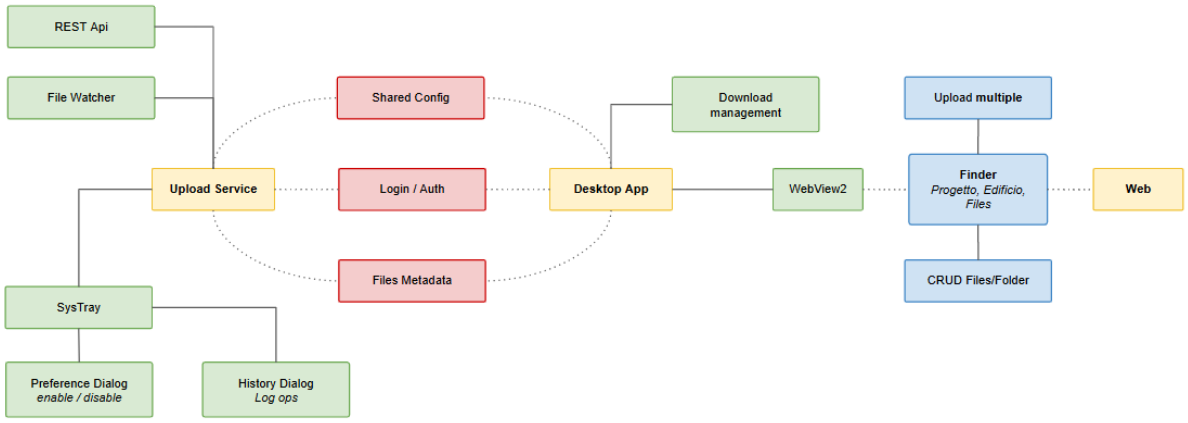


Figura 2.5: Architettura originale - Online Hybrid

Upload Service

L'uploader si sarebbe dovuto occupare del caricamento in background degli aggiornamenti e delle modifiche che gli utenti eseguivano sui file condivisi sul cloud.

Per accedere alle informazioni relative ai metadati dei file e per le richieste di upload e download, il service si appoggia ad un backend Spring Boot contattato tramite REST API, i cui endpoint principalmente utilizzati permettevano di interrogare la base dati per ottenere:

- le informazioni riguardanti progetti ed edifici
- la struttura del file tree che un determinato edificio ha con annessi i metadati relativi
- attributi per garantire un corretto versionamento
- il download o l'upload in chunk dei byte di un qualsiasi file
- l'accesso, previa fornitura di credenziali tramite l'IAM aziendale, alle API autenticate

Nel capitolo 3 ci sarà una descrizione più dettagliata delle REST API utilizzate dall'applicazione. Essendo che la totalità degli endpoint forniti dal backend era in origine destinato alla versione web del digital logbook, è stato molto utile potersi appoggiare agli stessi endpoint per poter fornire una certa coerenza tra le informazioni presenti sul web e all'interno dell'ambiente desktop.

Al centro della logica di funzionamento dell'uploader tuttavia, è presente un File System Watcher, agganciato ad una cartella temporanea e nascosta al momento dell'installazione,

che permetta il rilevamento immediato di tutta una serie di eventi che occorrono all'interno della folder specificata (creazione/eliminazione/apertura di file).

Sicuramente il fatto che la finalità di questa applicazione sarebbe dovuto essere un ambiente Windows, ha reso naturale la scelta di C# come linguaggio principale di programmazione e .NET come framework, potendone sfruttare la compatibilità con le API native di del sistema operativo, soprattutto per quanto riguarda l'accesso e la gestione del file system.

Sebbene in principio le idee sul come implementarne l'interfaccia grafica viravano verso l'implementazione di un windows service che lavorasse in background, si è poi preferito realizzare un'applicazione **Windows Form** che una volta eseguita fosse reperibile attraverso la **System Tray** di Windows, dando un'impressione Dropbox-like di programma secondario che venisse eseguito in background, con una propria interfaccia consultabile e che si aggiornasse a seconda delle azioni che l'utente eseguiva sui file condivisi.

Un fattore importante che l'uploader in sè doveva garantire era il corretto funzionamento in mancanza di connessione, avvisando l'utente sull'impossibilità di completare l'upload, riprendendolo però in maniera anche trasparente all'utente stesso quando la connessione ritorna. Il risultato che si è riuscito ad ottenere in questo caso è un **attesa passiva, non attiva**, quindi l'upload che viene fermato in caso di errori di connessione al momento non riparte senza l'azione dell'utente. Meccanismi di polling per rendere l'applicazione il più reattiva possibile sono stati implementati, ma per questa particolare funzionalità si è preferita una gestione più conservativa.

Veniva anche richiesta l'implementazione di un sistema di **logging** affidabile, che potesse fornire indicazioni precise e puntuali sulla sequenza di eventi che si susseguivano all'interno dell'applicazione (lato uploader in maniera più stringente, ma è stato utile implementarlo anche per la desktop app).

Desktop App

La desktop app doveva essere sicuramente la componente con cui l'utente avrebbe dovuto interagire di più, era quindi importante che l'interfaccia fosse chiara e diretta, che fornisse indicazioni nette e coincise sulle operazioni possibili senza overcomplicare inutilmente il flusso generico che l'utente avrebbe dovuto seguire per accedere e modificare i file all'interno dell'ambiente condiviso.

Per questo si è scelto di sfruttare un componente .NET come **WebView2**, che rendesse possibile iniettare pagine web statiche programmate in HTML e Javascript, fornendo un'alternativa alle estetiche più classiche a cui le normali applicazioni Windows Form si sono sempre affidate, dando una variante più moderna e accattivante all'intera user experience.

Oltre all'interfaccia iniziale e di presentazione della desktop app, il componente doveva poter fornire anche la possibilità di navigare all'interno del file tree dell'edificio di riferimento, per questo già dal principio era chiaro come fosse necessario un secondo form in cui mostrare un **file picker**, anche questo web-based sfruttando la WebView, che listasse in maniera organizzata i vari file, offrendo in maniera semplice e immediata la possibilità di scaricare o aggiungere determinati file all'interno dell'albero.

Arrivati a questa considerazione però, si è deciso di valutare il livello di convenienza che presentava il riutilizzare un componente React che mostrava questa alberatura, già pronto e presente all'interno della versione web del digital logbook. Questo tentativo è risultato vano per via della complessità che avrebbe aggiunto all'intera soluzione, risultava inutilmente complesso tentare di adattare l'Api Gateway su cui la versione web

si appoggiava tramite la scrittura di un plugin in codice Lua, necessario per autorizzare i token autenticati agganciati alle richieste che sarebbero partite dal browser embeddato all'interno della webview. Si è quindi deciso di realizzare un'interfaccia custom sempre web-based per mantenere la coerenza stilistica scelta nel form principale della Desktop App, che rimandasse in termini di struttura ciò che era presente nella versione web ma che avesse una personalità propria.

E' stato valutato anche l'utilizzo di soluzioni come Electron (espandere).

Verso una libreria comune

Sicuramente era fondamentale potersi affidare ad una serie di logiche comuni ad entrambe le componenti, queste necessità sono state divise principalmente in tre macro-aree, che man mano che lo sviluppo prendeva forma, hanno poi portato allo sviluppo di un SDK comune da cui uploader e launcher attingono:

- configurazione comune rispetto ai meccanismi di accesso alle informazioni remote fornite dal backend
- logica di autenticazione e gestione delle informazioni necessarie per effettuare richieste autenticate
- gestione e conseguente aggiornamento dei metadati relativi ai file, per comunicare indiciazioni precise e corrette al backend ma anche per fornire informazioni puntuali e coerenti all'utente

Più avanti verrà approfondita la composizione dell'SDK, ma la motivazione principale che ha portato alla sua creazione risiede nella forte presenza di logiche comuni a cui entrambe le componenti presto o tardi dovevano sfruttare e utilizzare.

Comunicazione tra .NET e Javascript

Determinante fu anche il capire come passare informazioni tra Javascript e .NET, in quanto le azioni che l'utente eseguiva sull'interfaccia web della desktop app dovevano poter essere gestite più a basso livello: ogni operazione che l'utente realizzava poteva aprire un altro form, richiedere il risultato di un'interrogazione direttamente alla base dati o eseguire un download in una posizione specifica della macchina locale su cui il programma veniva eseguito.

Determinante è stato l'utilizzo del metodo **postMessage()**, che permette l'abilitazione di una comunicazione cross-origin attraverso la gestione di eventi che vengono scatenati e catturati da entrambe le parti, attivando un canale di comunicazione basato su comuni event listener, essenziale per garantire una sequenza coerente del flusso di esecuzione e gestione degli eventi, permettendo la creazione di un vero e proprio **bridge** affidabile e veloce tra i due framework principali sui l'applicazione si appoggia.

2.2 Architettura e design attuale del sistema di cloud sync

I continui riscontri e feedback ottenuti sui requisiti e vincoli che l'applicazione doveva poter fornire, hanno portato alla costruzione di numerosi componenti che nel complesso

formano l'infrastruttura completa. Passeremo ora ad analizzare più nel dettaglio il modello nella sua versione più reccente, facendo una descrizione dei componenti in gioco e di come viene gestita la comunicazione tra di essi, attraverso l'esposizione di come l'SDK è stata costruita con l'obiettivo di renderlo modulo centrale e riutilizzabile per future implementazioni e architetture cloud-oriented.

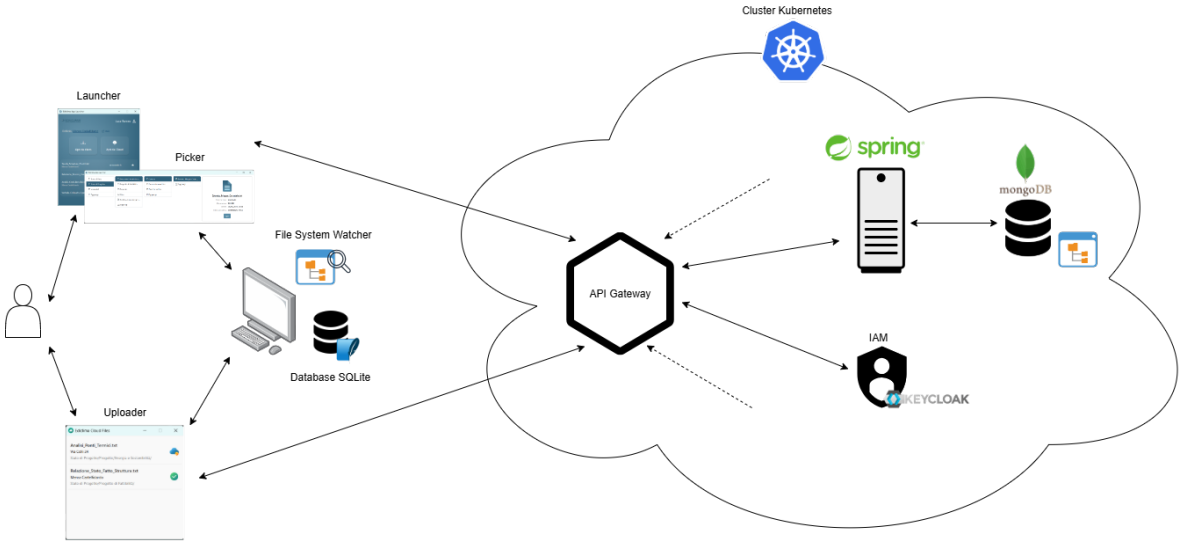


Figura 2.6: Architettura finale

Ogni componente all'interno del flusso operativo ha un compito definito e ben specifico, è essenziale che rimanga coerenza all'interno dello stato che l'utente crea utilizzando l'applicazione e che tutte le parti in gioco siano coordinate a dovere. Verranno ora passate in rassegna le parti in gioco, mantenendo però un approccio più funzionale e descrittivo, riservando l'analisi tecnica al capitolo successivo.

2.3 I componenti

Ora passeremo ad analizzare l'architettura dell'intero sistema, descrivendo i componenti in gioco, la comunicazione tra di essi e la composizione dello SDK come modulo centrale e riutilizzabile per architetture cloud-oriented.

2.4 Il Launcher

Il primo punto di contatto tra l'utente e il sistema è rappresentato dal launcher, l'applicazione desktop Windows Form, sviluppata in .NET 8.

Come accennato in precedenza, la user interface è stata realizzata principalmente usando linguaggi e framework destinati all'ambiente web (HTML, CSS, Javascript), che attraverso l'utilizzo del componente .NET WebView2 è stato possibile iniettare all'interno di un normale form destinato ad applicazione desktop più classiche.

WebView2 è un componente Microsoft che permette l'integrazione di Chromium, un motore di rendering basato su Microsoft Edge, all'interno delle applicazioni desktop .NET. La seguente scelta è stata strategica per diversi motivi:

- Permette di sviluppare interfacce utente moderne con tecnologie web più flessibili rispetto a WinForms o WPF puro.
- Consente di mantenere una certa coerenza stilistica tra Launcher e Uploader
- Facilita l'integrazione di elementi dinamici utilizzando Javascript, come l'utilizzo del drag and drop o l'utilizzo di Bootstrap per presentare interfacce moderne e accattivanti

L'utilizzo di un componente come la WebView ha un'importanza rilevante in quanto permette la creazione di un bridge tra Javascript e .NET basato su messaggi ed eventi, attraverso l'utilizzo della primitiva **postMessage**: attraverso questa funzione, è possibile infatti creare un canale di comunicazione per eventi o messaggi in entrambe le direzioni, in modo da poter reagire lato .NET quando l'utente interagisce con la pagina web, e al contrario scaturire aggiornamenti nell'interfaccia utente quando più a basso livello vengono scatenati degli eventi da notificare all'utente. Più dettagli a riguardo verranno forniti nel capitolo 3 riservato all'implementazione, ma il core di questa feature consiste nel continuo serializzamento di messaggi JSON e inviati al contesto .NET e Javascript, permettendo di aggiornare la UI web integrata.

Al primo avvio del **Launcher**, l'interfaccia fornisce solamente la possibilità di aprire ed editare file presenti sulla macchina locale dell'utente, mostrando una lista dei file più recentemente utilizzati dall'utente stesso. Un canvas laterale permette di aprire la sezione utente, in cui è presente il link per collegarsi all'IAM aziendale e iniziare la procedura di login.

Una volta autenticato con successo, l'interfaccia cambia e diverse informazioni e funzionalità aggiuntive vengono mostrate, basandosi direttamente sul contenuto del payload autenticato restituito dall'IAM.

Innanzitutto viene resa disponibile la possibilità di cambiare edificio, sebbene ne venga selezionato uno di default, per esplorare la lista degli edifici associati ai progetti di cui l'utente fa parte, sia come proprietario che come semplice spettatore. Ogni entry della lista che viene mostrata contiene un riferimento al progetto di cui fa parte.

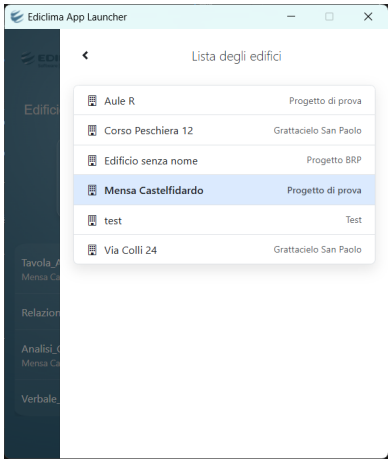


Figura 2.7: Sezione edifici

Viene mostrato il nome dell'utente loggato di fianco all'icona che permette di aprire il canvas relativo alla user section, che include informazioni quali l'email, l'username, il nome completo, e la quota attualmente utilizzata in termini di spazio utilizzato.

Assieme al bottone per il logout, il footer è dedicato a mostrare la versione corrente e il link al digital logbook.

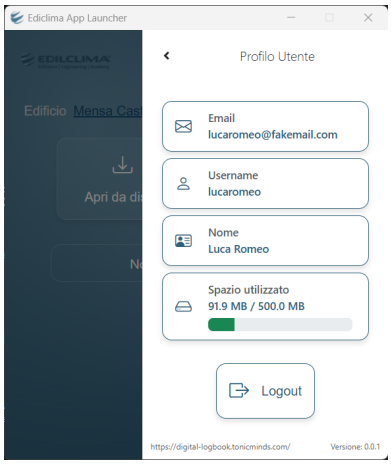


Figura 2.8: Sezione utente

Di fianco alla labela che permette di modificare l'edificio, è presente un link che rimanda alla versione web del digital logbook aprendo il browser predefinito, ma essendo che il reindirizzamento avverrà su un browser diverso rispetto a quello su cui girano le pagine statiche che mostrano l'ui del launcher, all'utente verrà richiesto di nuovo di inserire le credenziali per accedere al digital logbook. La situazione speculare invece, permette di evitare il doppio login: se quindi l'utente si dovesse loggare prima sul digital logbook web, la procedura di login che parte dall'applicazione usufruirebbe del token resituito al primo login, evitando di immettere una seconda volta le credenziali.

Oltre a ciò, l'Uploader mantiene una sezione Recent Files che consente di riaprire velocemente documenti e file usati di recenti, siano essi locali o cloud.

Accanto alla card con cui è possibile aprire l'Esplora Risorse del proprio sistema operativo, compare una seconda card con cui è possibile collegarsi al backend di Edilclima, interrogandolo per ricevere la lista di file e cartelle presenti nell'edificio correntemente selezionato.

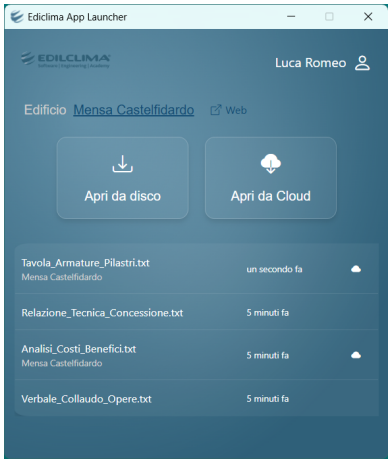


Figura 2.9: Interfaccia principale Launcher

Lo scheletro e l'organizzazione delle cartelle relative ai file dedicati ad un edificio specifico rimane la stessa per la totalità degli edifici, dovendo gestire pressoché sempre la

stessa categoria di file indipendentemente dall'edificio selezionato. Una volta aperto, il **picker** permette di effettuare il download di un file dal cloud per la modifica locale, e l'upload di un nuovo file attraverso la funzionalità di drag and drop nell'area dedicata al trascinamento.

L'utente ha poi la possibilità di esplorare queste cartelle, ricevendo informazioni relative ai metadati di file e cartelle grazie ad una preview laterale posizionata sulla destra. Il percorso all'interno del file tree di un file specifico viene evidenziato per aumentare la percezione di dove l'utente si trova all'interno dell'intera struttura.

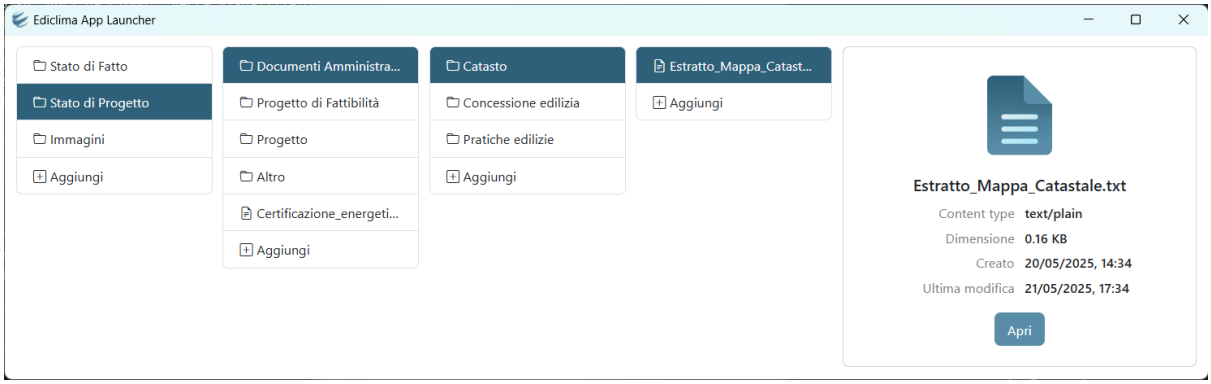


Figura 2.10: Interfaccia file picker

Il modo in cui questi metadati vengono presentati all'interno della preview rimanda al come sono organizzate le informazioni all'interno dell'Esplora Risorse di macOS, dando un taglio moderno e innovativo al come i dati vengono presentati e resi chiari in maniera elegante.

Viene anche data la possibilità di aggiungere nuovi file attraverso il trascinamento di questi ultimi all'interno di una sezione specifica della preview, previo selezionamento dell'opzione **Aggiungi**.

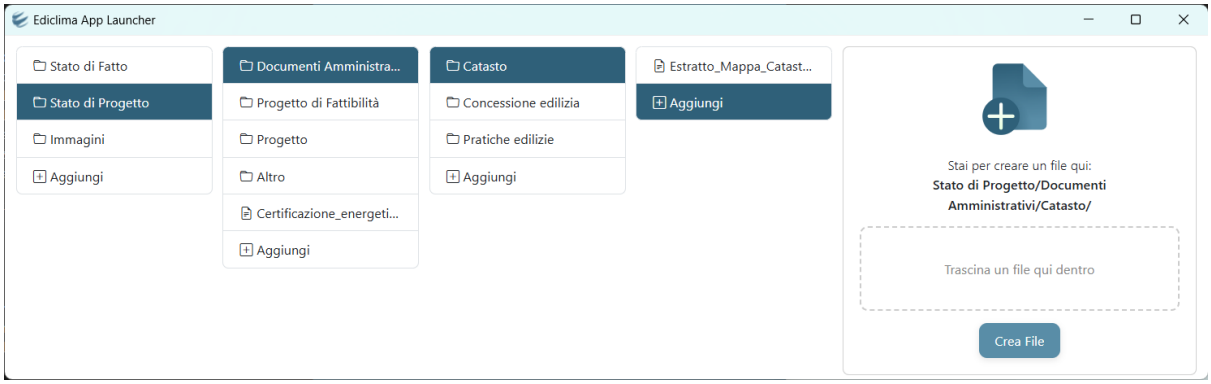


Figura 2.11: Preview di creazione file

2.5 L'uploader

L'uploader dal canto suo, è responsabile della gestione operativa della sincronizzazione dei contenuti locali con il cloud, mostrando la lista dei file aperti o aggiunti, assieme al percorso e all'edificio relativo.

E' possibile accederci, una volta lanciato l'eseguibile, attraverso un'icona che compare nella System Tray di Windows, aumentando l'impressione nei confronti dell'utente che il processo è in esecuzione in backgroung, capace di svolgere le proprie attività senza azioni dirette dell'utente stesso.

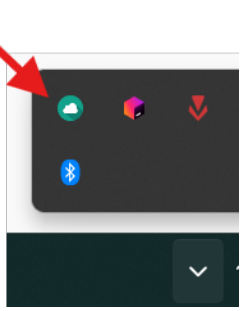


Figura 2.12: System Tray icon

Per quanto riguarda la user interface invece, è stato deciso che il posizionamento fisso nella sezione in basso a destra dello schermo fosse una scelta efficace, che ricorda il design di programmi come ToolBox di Jetbrains.

Anche questo componente, come il Launcher, è stato realizzato sfruttando il componente WebView per iniettare codice web-based. La struttura delle informazioni è semplice e chiara, presentando una entry per ogni file scaricato o caricato, fornendo il nome, l'edificio e il percorso relativo ad esso all'interno del file tree gerarchico dello stesso.

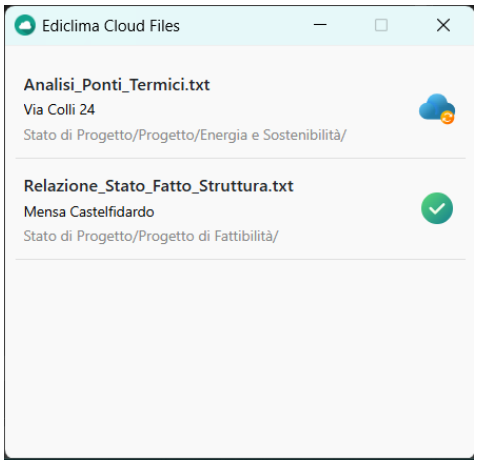


Figura 2.13: Uploader Main interface

Lo stato di sincronizzazione di ciascun file è rappresentato da un'icona differente, in modo da fornire un'indicazione immediata.

- **Synced**: la versione del file è allineata tra locale e cloud.
- **Syncing**: l'Uploader sta inviando i chunk del file modificato in background.
- **Not Synced**: sincronizzazione sospesa (Uploader non attivo al momento della modifica o attesa per upload in coda).

Figura 2.14: File Synced

Figura 2.15: File Not Synced

Figura 2.16: Syncing

postMessage(): il bridge tra JS e .NET

La sincronizzazione dei file in particolare, assieme a tutto ciò che concerne il mantenimento di una coerenza solida tra le informazioni mostrate e le operazioni rese disponibili all'utente, richiedeva che venisse presa in seria considerazione un metodo affidabile per scambiare informazioni tra ciò che l'utente faceva all'interno della WebView e la logica applicativa gestita dalla parte .NET dell'applicazione, in modo da poter sfruttare al massimo le possibilità di integrazione nativa che il framework offriva.

Ed è qui che entra in gioco il metodo **postMessage()**, in grado di permettere la definizione di eventi da rilevare e scatenare in base al comportamento dell'applicazione e in base alle azioni intraprese dall'utente stesso. Questo ha permesso la definizione di tutta una serie di canali bidirezionali tra l'ambiente nativo e quello web, in grado di poter gestire con semplicità la comunicazione tra di loro.

Alcuni degli eventi principali che sono stati modellati utilizzando questa logica sono:

- la conferma di avvenuta autenticazione, permettendo da una parte di salvare le informazioni relative all'utente autenticato lato .NET, e dall'altra di aggiornare le informazioni sull'interfaccia utente aggiungendo i dati relativi all'identità stessa dell'utente loggato
- l'avvenuto download e conseguente scrittura su disco dei file scaricati, assieme all'aggiornamento della status bar di caricamento in fase di upload, permettendo l'aggiornamento nella visualizzazione dell'uploader in termini di stato di sincronizzazione
- la possibilità di implementare meccanismi di polling, inviando messaggi periodici al backend per avere informazioni sull'eventuale aggiornamento della quota utilizzata dall'utente
- aggiornare la lista dei file recenti ogni qual volta l'utente scarichi un file nuovo

Queste e tutta una serie di altri eventi vengono gestiti pressoché nella stessa maniera, ovvero attraverso la serializzazione e deserializzazione di strutture JSON apposite, con all'interno discriminanti per distinguere la tipologia di messaggio, e quindi di evento, ricevuto.

Sono state utilizzate diverse declinazioni del metodo a seconda dei casi, ma questo aspetto verrà approfondito nel capitolo dedicato all'implementazione.

2.6 File System Watcher e SQLite locale cifrato

Una delle sfide principali dell'architettura è la sincronizzazione in tempo reale tra le due parti in gioco — Launcher e Uploader — e il backend cloud. Per ottenere questo risultato

è stato necessario implementare un sistema di monitoraggio continuo delle modifiche locali, in modo che ogni evento di creazione, modifica o eliminazione potesse essere intercettato e gestito il prima possibile.

TemporaryFileManager

Una delle classi principali del Launcher è **TemporaryFileManager**, il cui compito è quello di, una volta scaricato un file dal cloud Edilclima, scriverlo sul disco locale dell'utente loggato. Quando un utente seleziona un documento dal picker e ne richiede il download, il file viene scritto in una cartella nascosta e temporanea controllata dal sistema. Ci sono due vantaggi significativi in questo approccio:

- **Isolamento**: permette di attuare una politica di isolamento, i file sincronizzati non si "mischiano" con quelli realmente locali che l'utente apre con l'applicazione.
- **Controllo centralizzato**: permette che ogni modifica che avviene in questa cartella sia legata strettamente all'applicazione, fornendo un controllo centralizzato.

FileSystemWatcher

Per rilevare modifiche allo stato della cartella temporanea viene utilizzata la classe **FileSystemWatcher** del framework .NET, che osserva in background una o più directory e genera eventi al verificarsi di:

- creazione di file,
- modifica di file esistenti,
- cancellazione di file,
- rinominazione di file o cartelle.

Si è dimostrato fondamentale in questo contesto, dato che quando viene scaricato un nuovo file nella cartella il Watcher dell'Uploader intercetta l'evento di creazione, vengono generati eventi di modifica ogni volta che l'utente modifica un file aperto, permette anche di gestire eventuali cancellazioni (ad ora gestite cancellando il contenuto della cartella quando l'utente esegue il logout).

Questa scelta architetturale, di tipo *event-driven*, evita l'appesantimento dovuto al polling continuo, riducendo il consumo di risorse e migliorando la reattività.

La reazione agli eventi permette di aggiornare l'UI di Launcher e Uploader in maniera coerente, adeguando la visualizzazione dello stato di sincronizzazione, o modificando le informazioni contenute all'interno del database locale SQLite aggiungendo o aggiornando metadati. Qualsiasi aggiornamento viene riflesso anche nell'UI del Launcher, ad esempio aggiungendo una entry nella sezione dei file recenti, oppure aggiornando la quota utilizzata dall'utente in caso di file aggiunti al cloud Edilclima.

Questo è uno dei casi in cui è stato necessario aggiungere un meccanismo di polling per via del costo delle alternative, ma verrà approfondita meglio questa scelta nel terzo capitolo

SQLite locale cifrato

Il database **SQLite** integrato mantiene informazioni di stato sulla sincronizzazione, metadati, file recenti e dati essenziali per l'autenticazione.

Sebbene in principio venne valutato anche LiteDB come alternativa, SQLite risultava essere più affidabile sotto diversi aspetti, dall'accesso concorrente fino all'efficienza presentata nel manipolare query e dati integrato dentro l'ambiente .NET.

Ci sono state diverse motivazioni che hanno portato a scegliere SQLite, innanzitutto la portabilità che offre, permettendo di avere solamente un file da gestire. il fatto che non richieda installazioni o configurazioni particolari, che permetta di gestire un db pronto all'uso e la velocità dei tempi di lettura e scrittura per dataset piccoli come quelli che l'applicazione richiede sono stati anche fattori importanti.

Parte delle informazioni contenute al suo interno, come i token di Keycloak, sono cifrate, utilizzando una libreria di .NET chiamata **DataProtectionScope** che permette di cifrare utilizzando AES e legando la chiave simmetrica al thread pool utente. In questo modo, anche informazioni sensibili non possono essere lette anche in caso di accesso fisico al database.

Sincronizzazione programmata e coda di upload

Un aspetto fondamentale dell'Uploader è la gestione della **coda di upload**. Quando viene rilevata una modifica, il file non viene inviato immediatamente, ma inserito in una coda programmata:

In questo modo, vengono evitati upload multipli consecutivi se un file viene salvato ripetutamente in breve tempo, inoltre vengono rispettate le priorità e le risorse disponibili, garantendo che ogni operazione di upload sia completata senza errori prima di passare alla prossima.

In questo capitolo ci si limita alla spiegazione del concetto; la struttura dati e gli algoritmi di scheduling verranno analizzati in dettaglio nel Capitolo 3.

2.7 L' SDK, il cuore tecnico e modulare del progetto

Come già accennato in precedenza, il funzionamento corretto delle due parti in gioco dipende esclusivamente dalla capacità dell'intera applicazione nel mantenere uno stato coerente tra la versione locale e quella cloud dei file, coerenza garantita dall'SDK comune che centralizza:

- gestione autenticazione e token
- aggiornamento metadati e stato file
- invio e ricezione dei comandi di sincronizzazione

Questa scelta ha permesso di mantenere due eseguibili distinti ma con una base comune, dando una maggiore modularità e riutilizzabilità del codice, ma anche la riduzione del carico cognitivo lato utente, dato che ogni applicazione ha uno scopo chiaro e limitato.

Inizialmente, il sistema era composto solamente dai due programmi principali: Launcher e Uploader. Entrambi contenevano la propria logica interna per gestire autenticazione, comunicazione con il backend e manipolazione dello storage locale.

Tuttavia, l'aggiunta di nuove funzionalità ha fatto emergere una criticità evidente: gran parte del codice era duplicato, la gestione separate di operazioni uguali aumentavano il rischio di incoerenze, errori e bug difficili da rintracciare.

Per questo motivo, c'è stato bisogno di un'analisi architetturale più approfondita, evidenziando poi fattori determinanti alla creazione di una libreria condivisa tra le due parti in gioco, su tutti:

- la centralizzazione della logica condivisa permetteva di garantire comportamenti coerenti tra i due componenti
- la possibilità di trattare le feature come moduli separati, da agganciare ad entrambe o solo ad una delle due applicazioni
- l'evitare modifiche in parallelo alle due basi di codice indipendenti

Funzionalità principali integrate nello SDK

Alla lunga queste necessità hanno reso indispensabile lo sviluppo di uno SDK condiviso, che racchiude gran parte della logica operativa comune, tra le sue feature principali si distinguono:

- l'autenticazione, e tutta la logica che permette all'applicazione di interagire con l'Identity and Access Management aziendale basato su Keycloak, compresa la gestione, il rinnovo automatico e la memorizzazione sicura e cifrata dei token di accesso nello storage locale
- la gestione dello storage locale, permettendo di definire services, repositories e dtos utili per creare, leggere e aggiornare metadati relativi ai file sincronizzati, sfruttando SQLite come motore di persistenza
- la comunicazione con il backend Springboot, implementando le chiamate attraverso l'utilizzo di REST API fornite dal backend stesso, con gestione centralizzata di errori, retry e logging
- la condivisione di funzioni e metodi per notificare cambiamenti di stato, aggiornare le UI di client e uploader coordinandone le operazioni, l'utilizzo di classi di utilità per la definizione dei path condivisi in maniera trasversale

Vantaggi introdotti

L'adozione dello SDK ha portato benefici e vantaggi su più livelli:

- le operazioni comuni tra Launcher e Uploader producono risultati identici grazie alla centralizzazione della logica
- l'ottimizzazione di routine e funzioni comuni ha ridotto i tempi di esecuzione, specialmente nella gestione della cache locale
- eventuali modifiche o correzioni vengono applicate in un unico punto evitano interventi duplicati
- essendo un modulo autonomo a tutti gli effetti, può essere incluso in nuovi progetti desktop evitando di riscrivere la logica di base
- la code base di Launcher e Uploader è stata alleggerita considerevolmente, in modo da potersi concentrare solo sulla logica specifica del componente

Il ruolo che lo SDK ha assunto con il crescere delle funzionalità, ha sviluppato un'idea avvincente che va oltre al progetto stesso. Essendo diventato una sorta di controller delle operazioni che stanno alla base della comunicazione tra i componenti principali dell'architettura, lo SDK governa e orchestra a tutti gli effetti le interazioni tra i vari elementi dell'intero ecosistema, garantendo che ogni operazione segua procedure uniformi, mediando e dirigendo le richieste provenienti da più componenti, offrendo un'interfaccia di alto livello che nasconde complessità interne.

Questa sua natura ha reso possibile l'idea di una piattaforma modulare a tutti gli effetti, in cui lo SDK non è legato a questo spcifico progetto, ma ha la capacità di staccarsi da esso e fornire la base per applicazioni future con requisiti simili, garantendo una gestione sicura dello storage locale, autenticazione con un IAM consolidato, comunicazioni via REST API con backend proprietari.

In un ambiente cloud-oriented, lo SDK sviluppato si pone come pezzo fondamentale dell'intera infrastruttura, indipendentemente dal contesto applicativo.

2.8 Back-End: Kotlin e Spring Boot

Punto di collegamento fondamentale tra l'applicazione desktop è l'infrastruttura cloud proprietaria di Edilclima, il server ha la funzione di ricevere, validare ed elaborare le richieste provenienti dal client e dall'uploader, fornendo dati relativi al download e all'upload dei file, assieme ad informazioni sui progetti e gli edifici presenti, fonte primaria di dati per garantire che lo stato dello storage locale dell'utente e quello remoto risultino allineati.

Tutti i flussi di sincronizzazione e le operazioni di upload e download dei file passano attraverso una serie di endpoint REST consolidati, fungendo da interfaccia verso lo storage cloud. Le informazioni lato server relative al versioning permettono di mantenere la coerenza necessaria per poi dare indicazioni precise durante l'upload, evitando di sovrascrivere o compromettere le versioni precedenti.

L'intero backend è stato sviluppato utilizzando Kotlin come linguaggio principale, ed eseguito all'interno del framework Spring Boot. Confrontandomi con gli sviluppatori è stato possibile comprendere perché si è deciso di adottare queste tecnologie:

- Kotlin è stato scelto per la sintassi moderna e meno verbosa rispetto a Java, mantenendo comunque interoperabilità completa, rendendo il codice più leggibile e manutentibile
- Spring Boot dal canto sua ha rappresentato una scelta naturale per la costruzione di un'applicazione web robusta e modulare come il digital logbook, grazie alla sua capacità nel gestire automaticamente configurazioni di componenti e fornendo un valido sistema di iniezione delle dipendenze. Unito alla sua capacità di integrarsi in modo trasparente con database e sistemi di sicurezza ha velocizzato lo sviluppo in maniera considerevole

La totalità degli endpoint esposti dal server è utilizzata dal web digital logbook, ma essendo che molti di questi permettevano di eseguire gran parte delle operazioni che all'interno dell'applicazione erano richieste, ad un certo punto dello sviluppo è risultato naturale migrare dal backend custom che era stato costruito per le varie demo a quello ufficiale di cui il logbook si serviva. Segue una breve lista degli endpoint principali di cui il cloud sync si serve:

- **GET /api/projects**: richiesta per ottenere la lista completa dei progetti e degli edifici associati, contiene la lista degli utenti inclusi nello sviluppo di un determinato progetto o edificio, assieme a diversi metadati relativi ai permessi di ciascuno ed informazioni temporali sulla creazione e all'ultima modifica.

- **GET /api/file/tree**: l'endpoint restituisce la struttura gerarchica di directory e file associati a un determinato edificio, attraverso il passaggio del parametro **buildingId**. La response si compone di una rappresentazione ad albero (attraverso il campo children del JSON contenuto nella response stessa) che include sia le cartelle che i file all'interno, arricchita da metadati utili all'applicazione per eseguire richieste più specifiche in seguito. Tra le varie informazioni restituite si distinguono l'identificativo e il percorso di ciascun file, i permessi disponibili e i dettagli di tracciamento principali, relativi alla creazione ed ultima modifica degli utenti coinvolti. Questo endpoint è la base per la navigazione e corretta sincronizzazione dei contenuti digitali memorizzati nel backend, fornendo una visione chiara e gerarchica della struttura documentale legata ad un edificio specifico.

- **GET /api/file/download**: questo endpoint permette il download di un file a partire dal suo identificativo univoco (**fileId**), unico parametro obbligatorio. E' poi possibile specificare opzionalmente il parametro **versionNumber** per scaricare una versione precedent, in alternativa verrà restituita la versione più recente. La risposta contiene il contenuto binario del file che verrà poi gestito a livello client con la scrittura su disco all'interno della folder nascosta. In questo modo viene garantito l'accesso controllato ai documenti archiviati nel sistema, mantenendo comunque la possibilità di recuperare versioni meno recenti in caso di necessità.

- **GET /api/user/quota**: restituisce le informazioni relative allo spazio di archiviazioni disponibile, rispetto a quello utilizzato, dell'utente autenticato. Non richiede parametri espliciti in quanto i dati necessari vengono ricavati direttamente dal token JWT inviato con la richiesta. La risposta include l'identificativo dell'utente, la quota attualmente utilizzata e la quota massima. Questo endpoint è utilizzato principalmente dal launcher, come base per il meccanismo di polling periodico per l'aggiornamento della visualizzazione della quota utente, migliorando la reattività dell'interfaccia.

- **POST /api/file/fileId/version**: consente di prenotare la creazione di una nuova versione per un file già esistente, notificando al backend l'intenzione di procedere con l'upload di un documento aggiornato. Richiede che vengano specificati l'id del file, l'id dell'edificio e la size aggiornata. In risposta il server restituisce i metadati necessari al proseguimento della procedura di upload, come il numero aggiornato della versione, la dimensione dei chunk previsti e la quantità degli stessi.

- **POST /api/file**: permette la creazione e il caricamento di un nuovo file, previa prenotazione della versione. I parametri per la richiesta sono gli stessi della precedente API, con l'aggiunta del path relativo alla root dell'albero delle directory dell'edificio corrente. Questo endpoint viene chiamato in squeenza a quello di prenotazione, completando così il flusso di caricamento e sincronizzazione.

2.9 Keycloak e l'utilità dell'offline token

La gestione dell'autenticazione e dell'autorizzazione rappresenta un punto critico in qualsiasi sistema distribuito, soprattutto quando si tratta di garantire la sicurezza e la persistenza delle sessioni in applicazioni che operano in background. Nel contesto del progetto di cloud sync, questo ruolo è stato affidato a **Keycloak**, una piattaforma open source di Identity and Access Management (IAM) che offre una gamma completa di funzionalità per la gestione centralizzata di utenti, ruoli e policy di accesso.

Keycloak fornisce strumenti efficaci per implementare flussi di autenticazione e autorizzazione standardizzati, supportando protocolli come **OpenID Connect (OIDC) e SAML 2.0**. È progettato per ridurre al minimo la complessità lato client e server, permettendo di delegare interamente a un sistema esterno la gestione di login, logout, federazione di identità e recupero delle informazioni di profilo.

Tra le sue caratteristiche principali si possono trovare:

- gestione centralizzata di utenti, ruoli e gruppi
- supporto per più metodi di autenticazione (password, identity provider esterni, autenticazione a due fattori)
- generazione e validazione di token JWT per l'accesso alle API
- gestione delle sessioni e meccanismi di refresh per garantire continuità operativa

Fattore che contraddistingue Keycloak come Identiy Access Manager è sicuramente la gestione dei token di accesso che Keycloak stesso fornisce una volta che l'autenticazione è andata a buon fine:

- **access token**: questo è il token cifrato che il client aggancia alle REST API per autenticarle, garantendo in questo modo in maniera univoca l'identità di chi sta inviando queste richieste
- **refresh token**: utilizzato nel caso in cui l'access token scada, serve per ottenere nuovi access token, ha una scadenza più longeva rispetto al suo predecessore
- **id token**: contiene informazioni particolari sull'utente che si è appena loggato, come l'email, l'username e la password all'interno del realm Keycloak

Tutti i token vengono forniti in formato JWT, facilmente gestibile e manipolabile attraverso librerie e framework .NET.

Nel contesto della piattaforma cloud Edilclima, Keycloak è stato configurato per:

- **Autenticazione utente tramite OIDC**: ogni richiesta alle API back-end viene accompagnata da un token generato in seguito a login, che certifica l'identità dell'utente.
- **Gestione delle sessioni**: Keycloak si occupa di monitorare la durata delle sessioni, invalidarle se necessario e consentire il rinnovo tramite token.

L'offline token

Una caratteristica particolarmente rilevante per questo progetto è l'uso dell'offline token.

A differenza di un normale refresh token, l'offline token ha una durata molto più lunga (è possibile anche non mettere una scadenza specifica) e non è legato a una sessione attiva dell'utente. Può quindi essere utilizzato anche quando l'utente non ha un'interfaccia aperta o non ha effettuato recentemente un login manuale.

Inoltre resiste al riavvio della macchina o del servizio, mantenendo la possibilità di autenticare il client senza richiedere nuovamente le credenziali, e riduce la necessità di interruzioni del flusso di lavoro per richieste di autenticazione ripetute.

E' stato quindi utilizzato come refresh token, dal momento in cui una richiesta al backend restituisse un 401 Unauthorized, è stata implementata una logica di retry automatico utilizzando non l'access token ma bensì l'offline token.

Nel caso dell'applicazione desktop, questo approccio permette di effettuare il login una sola volta e non dover più immettere le credenziali tra una sessione e l'altra, oppute permette al modulo di upload in background di continuare a sincronizzare file in cloud in modo trasparente per l'utente, anche a distanza di giorni dall'ultima sessione attiva, dando all'utente una sensazione di connessione senza termine.

Integrazione con lo SDK

Lato client, la gestione dei token di Keycloak è uno dei tre punti fondamentali per la robustezza dell'SDK, occupandosi di memorizzare in modo sicuro i token stessi in maniera cifrata dentro il database locale SQLite e gestendo il retry automatico delle chiamate al backend in caso di 401 senza interazione manuale. In questo modo, il resto dell'applicazione non ha bisogno di conoscere i dettagli del meccanismo di autenticazione, potendo limitarsi a chiamare le API protette con un token sempre valido.

Questo modello di gestione delle credenziali garantisce una continuità di servizio indispensabile per un sistema di sincronizzazione automatica, riducendo al minimo le frizioni con l'utente e mantenendo elevati standard di sicurezza.

Il cluster Kubernetes

Kubernetes è una piattaforma open-source per l'orchestrazione di container ampiamente riconosciuta ed utilizzata nella gestione di applicazioni distribuite. Il suo scopo principale è quello di automatizzare il deployment, la scalabilità e il monitoraggio dei vari microservizi che il cluster offre, rendendo le applicazioni più resistenti ad eventuali guasti e indipendenti dall'infrastruttura che li gestisce. Grazie alle sue caratteristiche quindi, consente di gestire ambienti complessi garantendo al tempo stesso portabilità, isolamento e un livello di astrazione elevato.

Nel contesto di questo progetto, tutti i componenti server-side sono stati distribuiti all'interno di un cluster Kubernetes, ospita in particolare due microservizi centrali su cui il client si basa: il backend Spring Boot e l'Identity Access Management fornito da Keycloak. In questo modo si sono poste in partenza le basi per gestire il bilanciamento del carico, la gestione dei guasti e la scalabilità automatica, riducendo la complessità legata al deploy.

Dal punto di vista applicativo, l'interazione con Kubernetes è completamente trasparente. Il client .NET si limita a comunicare attraverso le API REST esposte dai microservizi, senza doversi preoccupare della logica sottostante che cura le repliche e l'allocazione delle risorse, gestita interamente dal cluster. Così facendo i servizi vengono trattati come entità

autonome, l'accesso alle risorse esposte avviene tramite endpoint noti e ben documentati, semplificando l'integrazione e l'evoluzione del sistema.

Pur non essendo stato oggetto di sviluppo diretto, l'adozione di Kubernetes ha permesso di concentrare l'attività di sviluppo sulle funzionalità applicative e sul design del client, avendo come infrastruttura e orchestratore un servizio robusto, scalabile e facilmente ampliabile.

Capitolo 3

Implementazione

Fino ad ora sono state analizzate le possibili soluzioni tecnologiche per la virtualizzazione di un file system e per la sincronizzazione dei file, ripercorrendo dal principio tutte le scelte di design che sono state effettuate e che hanno aiutato a dare una direzione precisa verso l'implementazione concreta del client desktop Edilclima.

Non ci sarà una rassegna dettagliata ed esaustiva del codice sorgente, ma verranno prese in considerazione le meccaniche interne e le logiche di integrazione che hanno caratterizzato meglio lo sviluppo.

Fin da subito è stato chiaro come la progettazione del client dovesse comprendere un insieme modulare di componenti, capaci di comunicare in maniera affidabile tra di loro, con delle responsabilità ben definite.

E' importante quindi la scomposizione corretta e precisa di questi moduli dividendo l'analisi per funzionalità, raccontare come ogni modulo giochi una parte importante nel disegno generale. Il filo conduttore del capitolo riguarderà quindi l'esaminazione di come questi moduli sono stati realizzati e orchestrati, dando la giusta importanza alle motivazioni delle scelte progettuali più importanti, alle tecniche implementative adottate, vantaggi e limiti riscontrati.

3.1 Panoramica architetturale

L'architettura del client sviluppato si articola quindi in tre moduli principali:

- il **Launcher** è il punto di ingresso, responsabile di autenticare l'utente permettendo la navigazione all'interno dei file tree associati ai vari edifici. Attore principale nella presentazione dei dati e nell'interazione con l'utente, espone le proprie funzionalità tramite un'interfaccia grafica basata su WebView2.
- l' **Uploader** si presenta come un servizio in background che si occupa della sincronizzazione con il cloud, gestendo in maniera coerente i conflitti e le code di caricamento, per effettuare l'upload in maniera robusta e non invasiva per l'utente.
- la **Common Library** è stata definita nella sua interezza in un secondo momento, quando la quantità di codice riutilizzabile e che era necessaria in entrambe le componenti cominciava a diventare rilevante. Il suo utilizzo ha garantito coerenza e la possibilità di raccogliere funzionalità comuni, alleggerendo in maniera significativa il client desktop.

Questa suddivisione in moduli risponde da un lato ad un'esigenza organizzativa, ma riflette anche l'esigenza di avere un architettura orientata alla manutenibilità, estendibilità e riuso del codice. Nello specifico, principi che rimandano a modelli come **MVVM**(ModelView-View-Model) o **MVC** sono stati utilizzati per separare il codice tra **repository** (incaricati di gestire la persistenza dei dati sul database interno SQLite), **servizi** (contenenti la business logic) e **interfaccia grafica** (quindi tutto ciò che riguarda la presentazione e l'interazione con l'utente), consentendo di poter avere una chiara distinzione in termini di responsabilità operative.

In questo modo sono stati ridotti al minimo i rischi di accoppiamento eccessivo del codice, semplificando molto lo sviluppo e favorendo l'evoluzione del sistema in maniera più naturale ed efficace.

Un altro aspetto importante è la modularità stessa: ogni componente è stato progettato per essere indipendente. Launcher e Uploader si parlano in maniera asincrona e nessun messaggio che da un fronte deve arrivare all'altro viene perso, ogni evento viene memorizzato e sfruttato poi nel momento in cui il modulo specifico si attiva, favorendo l'assenza di incomprensioni o incongruenze tra le due viste principali.

Ora ciascun modulo verrà descritto più nel dettaglio, con attenzione particolare alle funzionalità principali implementate, le classi che le forniscono e le scelte progettuali che sono state effettuate in questo senso.

3.2 Modulo Client

La responsabilità principale di questo modulo risiede nella sua capacità di fornire un'esperienza d'uso coerente, integrando la gestione locale dei file con alcune logiche applicative di più alto livello. E' il punto di contatto tra l'intero ecosistema software sottostante e l'utente, che opera quotidianamente sui suoi progetti.

La divisione più pragmatica del modulo riflette una separazione atta a suddividere il codice per gestire l'interfaccia utente, la logica applicativa e la gestione dei dati persistenti.

3.2.1 Integrazione di WebView2 con risorse locali

Il client si compone di due Windows Form che formano l'interfaccia grafica del modulo: il LauncherForm, punto di accesso iniziale, e il PickerForm, che permette la navigazione dei file da gestire in base all'edificio selezionato.

All'interno della cartella assets sono presenti numerosi file SVG, HTML, CSS e Javascript, che vengono caricati localmente e resi disponibili attraverso il componente .NET WebView2, in modo da integrare contenuti o codice web-based moderni senza dipendere totalmente da risorse esterne, riducendo anche i tempi di caricamento.

Questo è stato possibile utilizzando il metodo **SetVirtualHostNameToFolderMapping** , che permette di sfruttare il meccanismo di inizializzazione della WebView agganciandola a risorse locali, rappresentato nella figura sottostante:

Dopo aver garantito che il motore di rendering è disponibile tramite **EnsureCoreWebView2Async**, viene mappata la cartella assets ad un dominio virtuale permettendo il caricamento di file statici da disco locale come se provenissero da un server remoto, utilizzando un host virtuale fittizio (https://ec.static.files in questo caso). In questo modo è possibile combinare i vantaggi delle applicazioni web alla sicurezza e reattività data dall'ambiente desktop.

```
await webView.EnsureCoreWebView2Async(null);

// Mappa la cartella "assets" come un dominio virtuale
string assetsPath = Path.Combine(AppContext.BaseDirectory, "assets");
webView.CoreWebView2.SetVirtualHostNameToFolderMapping(
    "ec.static.files", assetsPath, CoreWebView2HostResourceAccessKind.Allow);

// Carica il file HTML usando l'host virtuale
webView.Source = new Uri("https://ec.static.files/home.html");

webView.CoreWebView2.WebMessageReceived += WebView_WebMessageReceived;
```

L'evento **WebMessageReceived** abilita inoltre il dialogo bidirezionale tra la WebView e il codice C#, fungendo come base per il bridge utile a scambiare comandi e dati in tempo reale tra i due ambienti. Consiste principalmente nell'agganciare il metodo con una funzione privata nel codice del Launcher che poi discrimina quale evento in particolare si è verificato.

3.2.2 Gestione del ciclo di autenticazione con Keycloak

La gestione completa dell'autenticazione e dell'autorizzazione è integrata all'interno del client tramite l'SDK, ma il Launcher utilizza diverse funzionalità offerte dalla libreria comune per avviare e coordinare il flusso di login/logout, preoccupandosi di integrare le informazioni relative all'utente a seguito di una procedura di autenticazione terminata con successo.

Il flusso tipico si compone di diversi step:

1. **Avvio** - lanciato l'eseguibile il client tenta il refresh dell'access token utilizzando l'offline token salvato localmente, così da evitare la richiesta delle credenziali se la sessione è ancora valida.

2. **Login** - se il refresh fallisce, la classe **AuthManager** gestisce il reindirizzamento verso la finestra di autenticazione e lo scambio sicuro con Keycloak, salvando poi all'interno della classe **AuthSession** i token restituiti assieme all'username. Questo comportamento è in realtà demandato alla libreria comune che contiene la logica per il **Device Authorization Flow**, AuthManager viene usata dal Launcher come interfaccia per lanciare il flusso di autenticazione.

3. **Gestione Token** - una volta ottenuti, i token vengono salvati criptati con l'ausilio della classe **TokenService** all'interno del database locale SQLite, oltre che all'interno della classe **AuthSession**, usata per accedere in maniera più immediata ai token stessi

4. **Chiamate al backend** - il server viene interrogato utilizzando il metodo **SendWithAutoRefreshAsync**, che intercetta eventuali errori di scadenza del token e tenta automaticamente il rinnovo tramite il token offline, garantendo continuità nel servizio e una sessione prolungata.

5. **Logout** - alla disconnessione dell'utente tutti i token precedentemente salvati vengono invalidati e rimossi, notificando l'UI e terminando la sessione utente con il metodo **Clear** della classe AuthSession

3.2.3 Polling semi-realtime per la quota utente

Altra funzionalità chiave del modulo è il polling semi-realtime della quota utente tramite il metodo **QuotaPollingAsync**.

La logica principale sfrutta un ciclo asincrono combinato con un **CancellationTokenSource** per effettuare in maniera sicura e programmata l'esecuzione periodica delle richieste al server, interrompendo il meccanismo di polling in caso di logout o chiusura dell'applicazione.

A intervalli regolari quindi, il client richiede al servizio remoto le informazioni aggiornate sulla quota disponibile per l'utente. Per evitare aggiornamenti ridondanti il sistema confronta il valore restituito con quello della quota correntemente salvata e visualizzata, in modo da aggiornare la WebView solo se vi è effettivamente una variazione.

3.2.4 Bridge tra WebView2 e codice .NET

Una delle caratteristiche principali è il meccanismo di comunicazione instaurato tra WebView2 e codice .NET, utilizzando il metodo **CoreWebView2.WebMessageReceived**. In questo modo è possibile ricevere ed inviare messaggi JSON tra lo strato Javascript della WebView, instaurando un vero e proprio bridge bidirezionale tra il front-end web e la logica applicativa in C#.

Il metodo che sta al centro della gestione dei messaggi è **WebViewWebMessageReceived**, composto essenzialmente da uno switch che associa a ciascuna azione la corrispondente logica implementata lato .NET, questo offre un duplice vantaggio: da un lato garantisce scalabilità ed estensibilità, dato che nuove azioni possono essere aggiunte semplicemente definendo un nuovo case, dall'altro migliora la leggibilità del codice.

Il risultato è quello di avere una comunicazione fluida basata sugli eventi, mantenendo distinte le reponsabilità: JavaScript gestisce l'interazione utente e la dinamica della UI, mentre C# si occupa delle operazioni di sistema come apertura file, autenticazione, logging e sincronizzazione con il cloud. Il pattern così pensato e realizzato permette di sfruttare la modernità di un front-end web senza rinunciare alla potenza delle API native di Windows.

3.2.5 Gestione dei file recenti

Il Launcher presenta la possibilità di recuperare in maniera rapida i file aperti di recente, senza doverli ricercare manualmente nel file system locale o nel cloud.

Il servizio a cui si appoggia è fornito dalle classi RecentFilesService e RecentFilesRepository, che assieme lavorano sulla persistenza locale tramite database SQLite dei riferimenti ai file aperti dall'utente, tra le altre cose. L'uso di Data Transfer Object per modellarli garantisce una rappresentazione coerente e facilmente serializzabile delle informazioni di ciascun file.

A livello di UI, è stato scelto di sfruttare la libreria Humanizer per trasformare i timestamp gressi in frasi più user-frendly (come ad esempio "aperto 5 minuti fa"), così che la cronologia di apertura risultasse più facilmente interpretabile dall'utente finale.

L'aggiornamento di questa lista avviene in maniera dinamica ad ogni apertura di un nuovo file, o di un file già presente ma il cui ultimo accesso è più lontano nel tempo, attraverso l'utilizzo del metodo **PostWebMessageAsString**, che invia alla WebView un payload JSON contenente i nuovi dati. Non appena il frontend riceve questo genere di messaggi, viene aggiornata istantaneamente l'interfaccia senza necessità di ricaricare la

finestra, in modo da mantenere una user interface reattiva e allineata in tempo reale con le operazioni svolte a più basso livello.

3.2.6 Separazione tra orchestrazione e logica di business

Come già accennato in precedenza, all'interno del modulo client c'è la chiara separazione tra i concetti di orchestrazione delle interazioni utente e la logica di business vera e propria. Il form principale non implementa direttamente la logica ma si limita a gestire l'interazione con la user interface e ad instradare le richieste ai services dedicati.

Il form agisce come un vero e proprio controller, ricevendo tutta una serie di input (ad esempio un messaggio proveniente dalla WebView o l'interazione con una finestra di dialogo) e richiamando i metodi esposti dai vari services relativi alla gestione dei file, ai loro metadati e alla sincronizzazione con l'uploader.

C'è quindi una chiara e netta separazione dei compiti, garantendo maggiore leggibilità del codice, miglioramento in termini di testabilità e riusabilità della logica. In questo senso il form diventa un coordinatore per l'ordine in cui le operazioni vanno eseguite, delegando alle classi specializzate per gestirle.

3.2.7 Robustezza e logging

Particolare importanza è stata data anche al rilevamento e gestione degli errori, man mano che il progetto cresceva in dimensione e requisiti diventava fondamentale racchiudere il codice in blocchi per la gestione delle eccezioni, in modo da avere un riscontro chiaro in caso di errori e per non compromettere la stabilità complessiva. Vengono in questo modo evitati crash improvvisi.

Oltre a questo aspetto, è stato introdotto un sistema di loggin centralizzato basato sulla classe custom AppLogger, registrando gli eventi significativi lato client e archiviandoli in file specifici, così da avere report affidabili da consultare per supportare le attività di debug in fase di sviluppo e agevolando la diagnostica per mantenere il sistema stabile.

3.2.8 La struttura del modulo Launcher

Fino ad ora sono state approfondite le funzionalità principali del Launcher in quanto controller dell'applicazione, con particolare attenzione agli aspetti legati all'interfaccia grafica e alle loro responsabilità di orchestrazione e integrazione con la Web View, alla gestione dell'autenticazione e al bridge .NET.

Per completare questa disamina è utile avere un'idea più chiara dell'organizzazione complessiva del modulo, osservando più da vicino le componenti che ne formano l'ossatura. Il progetto è strutturato in più directory, ciascuna con delle responsabilità precise all'interno dell'architettura e del design: dalle risorse statiche, ai repository come interfaccia con il database, attraverso la logica applicativa implementata nei services e le utilità trasversali.

La suddivisione in sottosistemi riflette un chiaro principio di separazione delle responsabilità, evidenziando una chiara distinzione tra presentazione, logica di business e persistenza.

Gli asset statici

Un primo aspetto rilevante che emerge dall'analisi dei file statici del Launcher e del Picker è la netta separazione tra presentazione e logica di orchestrazione. L'HTML è supportato

da fogli di stile CSS e da framework stilistici per migliorare l'esperienza utente, supporta unicamente la resa grafica e la definizione dei componenti dell'interfaccia. Il codice Javascript, dal canto suo, svolge più un ruolo di **adattatore**, intercetta le interazioni utente come clic, selezione e apertura di pannelli traducendole in messaggi inviati al backend .NET tramite **windows.chrom.webview.postMessage**.

```
document.getElementById("logo").addEventListener("click", () => {
    console.log("Logo clicked!");
    window.chrome.webview.postMessage({ action: "open-browser" });
});

document.getElementById("keycloakLogin").addEventListener("click", () => {
    document.getElementById("login-overlay").style.display = "flex";
    window.chrome.webview.postMessage({ action: "login" });
})

document.getElementById("keycloakLogout").addEventListener("click", () => {
    window.chrome.webview.postMessage({ action: "logout" });
})
```

Figura 3.1: Esempio di utilizzo della postMessage

Dalla figura X si nota bene come l'obiettivo principale sia quello di modellare le azioni dell'utente per scatenare l'invio di messaggi attraverso il bridge tra .NET e Javascript.

Questa scelta progettuale consente di evitare un numero considerevole di duplicazioni applicative lato client, mantenendo l'interfaccia leggera e demandando le elaborazioni al livello centrale.

```
document.querySelectorAll('.action-card').forEach(item => {
    item.addEventListener('click', () => {
        event.preventDefault();

        if (item.id === "navOpen") {
            window.chrome.webview.postMessage({ action: "open-file-dialog" });
        } else if (item.id === "navOpenCloud") {
            window.chrome.webview.postMessage({
                action: "open-cloud-form",
                content: currentBuilding,
                buildingName: currentBuildingName
            });
        } else {
            console.log("Unknown action:", item.id);
        }
    });
});
```

Figura 3.2: Esempio di payload del messaggio

Il payload del messaggio è tipicamente un oggetto JSON contenente un campo action,

che ne descrive a più alto livello il significato applicativo che .NET dovrà analizzare, assieme ad una serie opzionale di parametri di contesto, come l'identificativo dell'edificio nel caso si debba aprire il picker visualizzando i file associati.

Complementare al canale di invio dei comandi verso il backend è la gestione dei messaggi in direzione opposta, realizzata attraverso l'evento :

```
window.chrome.webview.addEventListener("message", ...)
```

In questo caso è il lato opposto del bridge, quello .NET, che a seguito di elaborazioni relative al cambiamento dello stato applicativo, notifica il frontend inviando un messaggio serializzato anche qui in formato JSON. All'interno del listener Javascript, il messaggio viene decodificato ed elaborato attraverso uno schema basato sul campo action, che permette di discriminare tra le diverse possibilità di interazione: aggiornamento della quota utente, popolamento della lista di edifici, visualizzazione dei file recenti, conferma di login/logout, notifiche relative a caricamenti o download riusciti.

In questo modo è possibile aggiornare dinamicamente l'interfaccia senza refresh completi della pagina, il frontend si limita a tradure lo stato ricevuto in modifiche dell'albero del DOM, garantendo così reattività. Un aspetto cruciale per quanto riguarda questo aspetto è la gestione dello stato applicativo tramite il listener stesso, assieme ad un insieme di variabili globali che riflettono il contesto corrente: dall'edificio selezionato, ai file aperti di recente, fino alla modalità operativa (locale o cloud). Non è uno stato che viene mantenuto in modo persistente sul lato client, ma è costantemente aggiornato dal backend .NET attraverso il meccanismo di messaggistica della WebView.

Un'azione tipica è la login-confirm, che provoca l'aggiornamento simultaneo di più sezioni dell'interfaccia, con la popolazione degli edifici e dei file recenti, aggiornando il pannello utente con i propri dati e la quota disponibile.

La struttura ad albero dell'interfaccia del File Picker

Il file picker presenta una visualizzazione gerarchica dei file e delle directory, modellata come un **albero ad N livelli**. La struttura ad albero riflette il formato in cui il backend invia i dati, file e cartelle vengono visualizzati in modo dinamico con elementi di supporto a strutture di directory più nidificate.

Ogni nodo dell'albero viene costruito in Javascript usando come base il JSON del body della response all'API apposita, le directory possono contenere ulteriori sotto-directory o file, generando colonne aggiuntive all'interno dell'interfaccia man mano che l'utente esplora la gerarchia. La visualizzazione dei dettagli è separata in una scheda dedicata che compare sulla destra dell'interfaccia del picker, mostrando tra le varie informazioni il path relativo corrente per ogni percorso calcolando ricorsivamente la gerarchia, in questo modo viene facilitata la tracciabilità dei percorsi anche grazie al cambiamento nello stile dei pannelli, tramite l'utilizzo di colori più marcati quando questi ultimi vengono selezionati.

Alla base della serializzazione e traduzione si trova il DTO BackendNode, che modella in maniera uniforme sia i nodi cartella sia i nodi file, ci sono alcune proprietà comuni ad entrambe:

- tipologia (file o cartella)
- nome
- id

- lista dei children (una lista che contiene i riferimenti ai nodi figli, se esistenti)

altre proprietà specifiche per i file, opzionali:

- dimensione
- numero di versione
- informazioni temporali sulla data di creazione e di ultima modifica
- il mime type

Ovviamente un nodo può contenere altri nodi children solamente se è di tipo directory, ma è abbastanza chiaro che se il primo gruppo di informazioni permette di capire dinamicamente come gestire la comparsa o scomparsa dei vari pannelli, il secondo gruppo sono dettagli utili per alimentare la preview situata alla destra della schermata del picker.

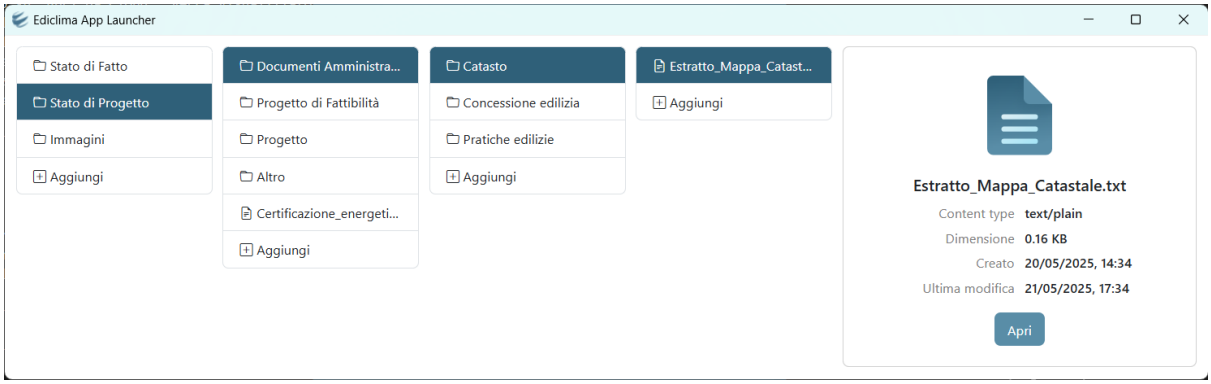


Figura 3.3: Interfaccia finale del Picker

In particolare, la funzione **generateFileStructure** itera sui vari nodi children a partire dalla root directory, richiamando ricorsivamente la funzione **appendChildren**, che crea l'elemento grafico corrispondente al nodo e gestendo la nidificazione aggiungendo uno colonna successiva se necessario.

Funzioni dedicate si occupano di evidenziare il nodo attivo rimuovendo eventuali colonne figlie non più pertinenti, garantendo una navigazione pulita e coerente con il percorso corrente.

Ogni azione utente viene tradotti in un messaggio inviato tramite l'utilizzo di postMessage al backend .NET, che centralizza la logica applicativa riguardante:

- aperture dei file con l'evento **download-from-cloud**
- caricamento dei nuovi file con l'evento **new-file**
- la navigazione all'interno delle directory con l'evento **load-cloud-files**

Ultimo punto importante da mettere in risalto è la possibilità di trascinare un file all'interno dell'interfaccia per caricarlo, utilizzando la drop zone. Il file viene convertito in Base64 e preparato per essere inviato al backend.

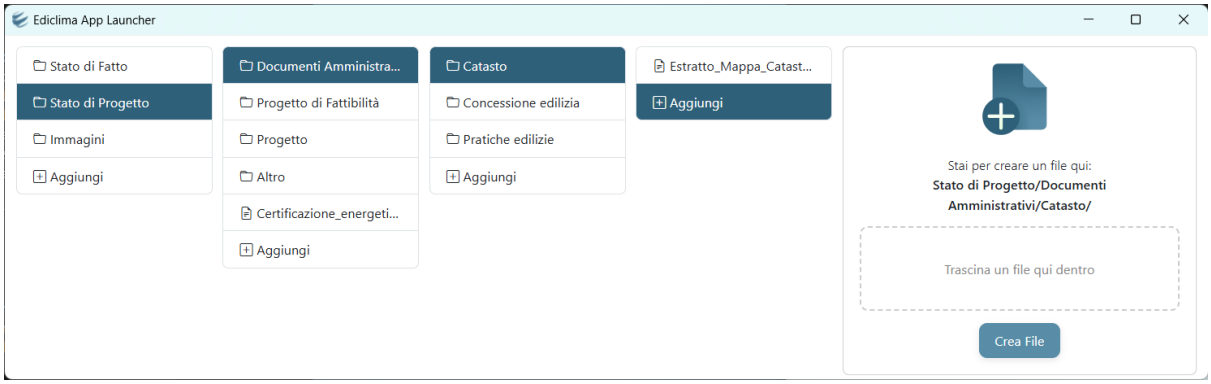


Figura 3.4: Aggiunta file tramite Drag and Drop

DTOs e Repositories

Questa sezione si concentra su ciò che è implementato a livello di logica applicativa e persistenza del modulo, sono classi fondamentali per l'accesso al backend e le regole applicative, definiscono ad un livello più basso la struttura e la composizione di come la comunicazione tra backend e database permette di orchestrare il flusso applicativo, dal punto di vista del client.

Questa componente dell'architettura è costruita attorno a due responsabilità principali: la gestione dei dati in transito tra frontend e backend, e la persistenza delle informazioni necessarie al funzionamento locale, con particolare attenzione sui dati utili al mantenimento della coerenza e della sincronizzazione. In questo quadro, sia i DTOs che i Repositories giocano un ruolo complementare, dato che i primi stabiliscono un canale di comunicazione standardizzato, mentre i secondi assicurano che l'accesso al database sia strutturato ma astratto allo stesso tempo, fornendo al tempo stesso un modo sicuro e affidabile per esaminare la base dati.

DTOS (Data Transfer Objects)

I Data Transfer Objects sono classi prive di logica applicativa, progettate per rappresentare in maniera chiara i dati scambiati tra i diversi strati del sistema. In questo modo si ottiene un vantaggio architetturale importante, in quanto il formato dei messaggi è standardizzato, ogni evento o comando trasmesso tramite la WebView è rappresentato da un oggetto JSON ben riconosciuto e comune a qualunque componente che ne abbia bisogno. Vengono ridotti anche gli errori di interpretazione, essendo i campi tipizzati ci sono meno probabilità che i dati attesi siano inconsistenti rispetto a quelli ricevuti.

Il progetto utilizza diversi DTOs per modellare le entità centrali che l'applicazione deve gestire, alcuni di questi sono:

- il **BackendNode**, citato in precedenza, modella le risposte del server per rappresentare la struttura ad albero delle directory e dei file associati ad un edificio.
- **UserInfo**, contiene i dati legati al profilo utente e allo stato di autenticazione.
- **JSMessage**, contiene la struttura dei messaggi e delle comunicazioni che avvengono attraverso il bridge tra Javascript e .NET, come il tipo di azione che l'utente ha eseguito nella WebView, assieme a qualche dettaglio dipendente dal contesto in cui l'utente si trova.

Repositories

I Repositories sono la controparte che gestisce la persistenza vera e propria. L'applicazione infatti, mantiene uno storage locale in un database nascosto all'interno della directory di lavoro, utilizzato principalmente per memorizzare informazioni che devono sopravvivere tra una sessione e l'altra, comprese quelle legate allo stato di sincronizzazione con il cloud.

Tra le funzioni di maggior importanza, i repositories si occupano di fornire un'interfaccia per accedere e modificare i dati, nascondere i dettagli implementativi del database centralizzando le query e la logica di persistenza. Incapsulano le principali operazioni CRUD da eseguire sulle varie entità del database SQLite, fungendo da interfaccia tra i servizi applicativi e la persistenza.

Sono tre i repository principali all'interno del modulo Launcher:

- **SyncRepository**: si occupa di gestire le operazioni sulla tabella SyncInfo, contenente lo stato di sincronizzazione dei file locale rispetto alla loro controparte cloud. La tabella è costruita per avere un vincolo di unicità sul nome del file, aggiungendo e aggiornando il campo LastHash ogni qualvolta che delle modifiche vengono effettuate sul file stesso. In questo modo si evita di dover gestire duplicati, mantenendo sempre un riferimento aggiornato allo stato del file. L'hash funge da impronta digitale del contenuto, evitando confronti byte-to-byte vengono rilevate rapidamente le modifiche.
- **RecentFilesRepository**: gestisce la cronologia dei file aperti dall'utente, contiene metadati associati ai file aperti di recente, fornendo una lettura immediata all'applicazione per mostrare nell'interfaccia utente del Launcher una lista dei file acceduti di recente, ordinata per l'attributo LastOpened. In questo modo non sono necessarie chiamate al server, la logica di gestione dei recent files è interamente gestita dal client.
- **MetadataRepository**: sicuramente il più ricco di operazioni, le informazioni manipolate da questo repository costituiscono il registro locale principale, da cui i servizi applicativi attingono per ricostruire la mappa dei file disponibili e determinare quali elementi devono essere sincronizzati con il cloud.

Usandoli assieme, è possibile mantenere un equilibrio tra efficienza locale e consistenza con il backend, in modo da garantire che l'applicazione sia robusta e immediata in termini di risposte, il loro utilizzo in combinazione con i DTO favorisce l'estendibilità del sistema e si inserisce in un approccio più ampio basato sul concetto di Separation of Concerns:

- i DTOs standardizzano il formato dei dati in ingresso e in uscita senza inglobare logica.
- i Repositories isolano il livello di persistenza.
- i Services, di cui si parlerà tra poco, si occupano delle regole applicative.

Services

Il ruolo principale dei services è quello di coordinare i casi d'uso, trasformando operazioni elementari sui dati in logiche applicative più complesse e significative per l'utente, così da incapsulare regole e validazioni, orchestrare i repository e le operazioni di sistema, offrendo interfacce chiare ai livelli superiori. Viene così separata la logica di business da quella di persistenza, mantenendo l'applicazione più robusta e modulare.

Analizziamo ora più in dettaglio il ruolo dei services utilizzati dal modulo Launcher.

FileService

E' un service trasversale, che si occupa di gestire tutte le operazioni più a basso livello sui file stessi.

- si assicura che la cartella di destinazione esista prima di scriverci un file all'interno
- sovrascrive eventuali versioni precedenti e salva in locale i contenuti in formato binario
- utilizzando l'algoritmo SHA-256, genera un hash univoco del contenuto del file, fondamentale per i meccanismi di sincronizzazione
- invoca il file system per aprire i file con la loro applicazione predefinita, fornendo all'utente un'esperienza di utilizzo immediata

Pur non interagendo direttamente con i repository, questo service fornisce funzionalità base necessarie ad altri servizi.

MetadataService

Incapsula la logica legata ai metadati dei file sincronizzati, appoggiandosi al **MetadataRepository** per gestire l'entità **Metadata**.

Esegue validazioni sui dati di input, impedendo ad esempio l'inserimento di record con FileName uguale o BuildingId mancanti, garantendo così un alto livello di consistenza dei dati.

Grazie a questo servizio, il frontend non deve conoscere i dettagli della persistenza o le logiche di validazione, può semplicemente invocare i metodi.

RecentFilesService

E' dedicato alla gestione dello storico dei file aperti all'utente, si interfaccia con il RecentFilesRepository eseguendo controlli specifici distinguendo la provenienza di un file, dando una visualizzazione chiara e distintiva di file cloud o locali. Alimenta direttamente il pannello dei file recenti del frontend.

SyncService

Gestice la logica di sincronizzazione e aggiornamento delle informazioni locali di coordinamento rispetto al cloud. Si appoggia al SyncRepository e all'entità SyncInfo, registrando o aggiornando lo stato di sincronizzazione di un file. E' strettamente legato al FileService per la generazione degli hash, creando una catena logica che permette di monitorare l'integrità e l'allineamento tra file locali e cloud.

E' ormai chiara la volontà di seguire i principi di un'architettura ben stratificata:

- i repository gestiscono la persistenza a basso livello
- i services aggiungono la logica applicativa, le validazioni e il coordinamento
- la WebView invoca i services senza doversi preoccupare dei dettagli sottostanti

In questo modo si hanno numerosi vantaggi, dalla manutenibilità agevolata in quanto cambiamenti eventuali a livello di database richiedono modifiche solo nei repository, alla possibilità di riutilizzare i services in vari punti dell'applicazione evitando la duplicazione di codice.

Utils

Per concludere il discorso sul Launcher, è necessario parlare degli strumenti trasversali per la gestione dei file temporanei, dei percorsi e delle configurazioni comuni che il modulo utilizza. Non sono componenti che rappresentano direttamente la logica di business, ma permettono al resto dell'applicazione di operare in maniera coordinata ed efficiente.

App.config

Centralizza i parametri di configurazioni utilizzati dal Launcher stesso, permette di modificare configurazioni del sistema senza dover ricompilare facilitando il deployment in ambienti diversi (test o produzione). In combinazione con la classe **AppPaths**, contenuta nella libreria comune, garantisce uniformità nella gestione dei percorsi.

TemporaryFileManager

Incapsula la logica relativa alla gestione dei file temporanei scaricati o creati dal Launcher, si colloca a metà strada tra i services e l'interfaccia utente, coordinando diverse operazioni:

- **Creazione dei file temporanei**: garantisce l'esistenza della directory temporanea, scrive i contenuti su disco, sovrascrivendo eventuali versioni precedenti. Calcola l'hash SHA-256, aggiorna i metadati e lo stato di sincronizzazione tramite i service relativi ai metadati e al servizio di sync.
- **Integrazione con i DTO**: utilizzo DTO specifici per standardizzare le richieste provenienti dal backend o dal frontend, semplificando la creazione dei file sia in caso di download dal cloud sia in caso di nuovi file locali
- **Gestione unificata dei file recenti**: attraverso i suoi metodi, aggiorna lo storico dei file aperti, distinguendo tra quelli locali e quelli cloud, assicurando coerenza nell'interfaccia e mantenendo uno storico sempre aggiornato e coordinato.

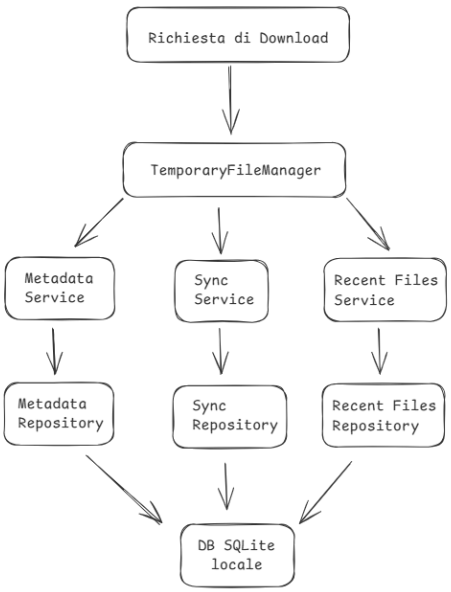


Figura 3.5: Flusso di comunicazione Service - Repo

Non accede mai al database, ma delega la persistenza a service e repository, mantenendo un livello di coordinamento alto.

La loro funzione non è marginale, senza questi componenti, la logica applicativa sarebbe sparsa tra più classi e la coordinazione sarebbe difficilemente mantenibile. In questo senso gli utils si configurano come veri e propri facilitatori dell'intera architettura.

3.3 Modulo Uploader

Il modulo Uploader costituisce il centro del meccanismo di sincronizzazione con il cloud, con la funzionalità di agire in background avendo un impatto minimo sull'esperienza utente.

E' configurato come servizio ibridio: da un lato presenta un UI minimale modellata, come per il modulo Client, dall'intreccio dell'ambiente nativo con le tecnologie web più moderne, dall'altra opera come processo in background integrandosi con la system tray di Windows, avvicinandosi al paradigma seguito da applicazioni di file sync moderne come Dropbox o OneDrive, fornendo una soluzione che mantenga allo stesso tempo visibilità e controllo sullo stato delle operazioni.

Il suo funzionamento si fonda su tre pilastri principali:

- il monitoraggio continuo delle cartelle locali tramite un watcher dedicato
- la gestione della coda di caricamento attraverso l'UploadManager
- l'aggiornamento in tempo reale dell'interfaccia utente tramite un approccio event-driven

A questo si aggiungono poi la gestione sicura dei token di autenticazione, permettendo al modulo di operare in modo autonomo senza richiedere interazioni frequenti con l'utente.

3.3.1 Integrazione con WebView2 e gestione UI minimale

Diversamente da quanto fatto nel modulo client, in cui l'integrazione con la WebView2 è stata sfruttata per costruire una vera e propria interfaccia con funzioni di navigazione e interazione diretta con i file, l'Uploader ha un approccio più minimale, dove l'interfaccia grafica si limita a monitorare lo stato degli upload fornendo indicazioni chiare sullo stato di sincronizzazione.

Anche in questo caso la gestione della WebView è demandata ad una classe dedicata chiamata WebViewManager, ponte tra la logica applicativa e l'aggiornamento dei contenuti web, e il flusso di comunicazione rimane di tipo event-driven: ogni evento rilevato dal watcher, ogni aggiornamento sullo stato della coda di caricamento viene tradotto in un aggiornamento della WebView, in modo che l'utente abbia sempre un feedback reale senza interagire mai direttamente con l'Uploader stesso.

Si differenzia quindi dall'utilizzo che ne è stato fatto nel modulo Client, fornendo una sorta di pannello informativo integrato con la logica di sincronizzazione, atto a fornire informazioni all'utente, senza che quest'ultimo ci debba interagire.

3.3.2 Monitoraggio dei file locali

Al centro del monitoraggio dei file da sincronizzare con il cloud aziendale c'è il Watcher, che osserva e rileva gli eventi all'interno della cartella nascosta utilizzata dal sistema.

Non si è scelto come approccio uno basato su cicli di polling periodici in quanto un approccio event-driven risultava più naturale da adottare: ogni volta che un file viene creato, modificato, rinominato o eliminato all'interno della cartella monitorata, si genera un evento che propaga l'informazione agli altri moduli interessati.

Gli eventi principali vengono raccolti con l'ausilio di due componenti importanti ma con ruoli diversi:

- l'UploadManager si occupa di gestire la coda dei file in attesa di caricamento verso il cloud
- il WebViewManager si occupa invece di aggiornare l'interfaccia utente per fornire un feedback immediato all'utente sullo stato corrente di sincronizzazione

Logica evidente nella fase di inizializzazione del form dove gli handler dei vari eventi vengono collegati in maniera esplicita

```
watcher.OnFileStatusListUpdated += webViewManager.UpdateView;
watcher.CleanUploader += webViewManager.CleanUploader;
watcher.InitializeUploader += webViewManager.InitializeUploader;
uploadMananger.UploaderService.SendUpdatedFile += webViewManager.UpdateUploadProgress;
uploadMananger.PendingUpload += webViewManager.UpdateView;
```

Lo snippet sopra fornito rende chiaro quanto l'infrastruttura sia fortemente basata su delegati ed eventi. Il Watcher emette notifiche ogni volta che rileva un cambiamento, mentre il WebViewManager e l'UploaderManager traducono queste notifiche in aggiornamenti nell'interfaccia e azioni di sincronizzazione.

In questo modo l'applicazione evita di consumare risorse inutilmente in controlli ripetitivi e l'andamento delle operazioni viene mostrato in tempo reale all'utente.

3.3.3 Gestione degli upload in background

Come già anticipato in precedenza, la fase di caricmaneto dei file verso il cloud è centralizzata all'interno della classe UploadManager, orchestratore principale per le operazioni di sincronizzazione. L'obiettivo principale di questo componente è garantire che il processo di upload sia affidabile, scalabile e il più trasparente possibile per l'utente.

Ogni nuovo file scaricato dall'utente viene rilevato dal Watcher e aggiunto in una lista di upload pendenti, che l'UploadManager processo in modo sequenziale. In questo modo è possibile mantenere l'ordine e la consistenza delle operazioni grazie all'ausilio di una struttura FIFO per i file in attesa, riducendo al minimo il rischio di conflitti o duplicazioni, anche in caso di variazioni rapide all'interno della cartella monitorata.

Dei due eventi esposti nella precedente figura, il primo (SendUpdatedFile) notifica il progresso di ogni singolo file, in modo da aggiornare l'interfaccia con informazioni puntuali e precise, con l'aggiunta di una nuova entry in caso di file scaricati per la prima volta, o l'aggiornamento delle percentuali di completamento in caso di upload, o ancora tramite l'avviso di non caricamento in caso di assenza di connessione o di attesa perché un altro caricamento è in corso, utilizzando l'evento PendingUpload.

Sono inoltre presenti anche metodi per la pulizia e la reinizializzaione dell'ambiente di upload, in modo che il sistema ritorni sempre ad uno stato coerente evitando la perdita di dati o informazioni, garantendo la completa continuità operativa senza l'ausilio di un intervento manuale.

L'insieme di queste dinamiche ha permesso lo sviluppo di un upload robusto e resiliente, in modo da operare come servizio in bacground in grado di garantire la sincronizzazione anche in scenari complessi.

3.3.4 Integrazione con la system tray

Per rendere più reale la sensazione di un service in background, nel momento di inizializzazione la finestra principale viene nascosta, eslcusa dalla taskbar delle applicazioni e resa non ridimensionabile, così da non interferire con il normale utilizzo del sistema da parte dell'utente. La finestra è accessabile selezionando una NotifyIcon posizionata nella system tray di Windows, così da mostrare la finestra di monitoraggio degli upload seguendo un pattern ampiamente consolidato in software di sincronizzazione di questo tipo, come Dropbox o OneDrive. In questo modo l'Uploader rimane attivo ma non risulta invasivo per l'utente finale, la possibilità però di aprire l'applicazione tramite un semplice clic sull'icona garantisce un buon compromesso tra discrezione e accessibilità.

3.3.5 Gestione sicura delle credenziali

Altro aspetto cruciale riguarda la gestione delle credenziali necessarie per comunicare in maniera autenticata e affidabile con il back-end proprietario. Seppur centrale all'interno dell'ultimo modulo che più avanti verrà analizzato (l'SDK), è giusto esaminare come questa funzionalità viene trattata all'interno del modulo Uploader.

Durante la fase di avvio, l'Uploader recupera i token di accesso e di aggiornamento, se disponibili, e li utilizza per inizializzare la sessione tramite la classe AuthSession. In questo modo il service opera in maniera completamente autonoma nelle sue operazioni, mantenendo al tempo stesso elevati standard di sicurezza.

Anche in questo caso l'utilizzo del refresh token consente di garantire continuità nelle operazioni di upload anche in caso di scadenza del token di accesso, in modo che vengano ridotte al minimo eventuali interruzioni del servizio. Il risultato della separazione tra client e uploader, unita alla centralizzazione della logica di autenticazione all'interno della common library, permette di bilanciare in maniera adeguata semplicità d'uso e protezione delle informazioni.

3.3.6 La struttura del modulo Uploader

E' utile soffermarsi sull'organizzazione interna del modulo dopo aver descritto il suo funzionamento come servizio di sincronizzazione in background. A differenza del Launcher, l'Uploader è pensato per avere una struttura tale da sostenere processi continuativi e automatizzati, con un impatto minimo sull'esperienza utente. Anche qui viene seguito un principio di separazione dei compiti, in cui a ciascun pacchetto è assegnato un ruolo preciso: gli asset contengono le risorse statiche, i services racchiudono la logica applicativa e contengono le classi responsabili dell'orchestrazione e del monitoraggio della sincronizzazione in background dei file, oltre a gestire la coda di caricamento.

Questa organizzazione permette la gestione di un sistema leggibile e facilmente mantenibile, in cui le dipendenze tra i vari componenti sono chiare e ben definite. Ora verranno analizzati più nel dettaglio le singole directory del modulo, sottolineando come queste contribuiscano attivamente alla realizzazione di un servizio di sincronizzazione affidabile e resiliente che opera in background.

Asset statici

All'interno del modulo, la folder dedicata agli asset statici si occupa dell'interfaccia utente ed è responsabile di una parte relativa all'interazione, al cui interno è presente il codice Javascript per regolamentare ed orchestrare il dialogo tra .NET e il frontend incorporato nella WebView2.

La logica di comunicazione si basa su un meccanismo di comunicazione asincrono tra il contesto web e il codice C#:

1. in risposta all'evento DOMContentLoaded, lo script invia un messaggio al backend utilizzando **window.chrome.webview.postMessage(action: 'uploader-ready')**;
2. questa chiamata segnala che il form è pronto per ricevere e visualizzare gli aggiornamenti relativi ai file in upload
3. da quel momento in poi diventa un listener, in attesa di messaggi provenienti dal backend che portano con sè informazioni relative allo stato di un file, dei caricamenti ed eventuali errori di sincronizzazione.

Gestione degli eventi e aggiornamento dinamico dell'interfaccia

Ogni messaggio proveniente dal backend contiene un campo action che determina il tipo di operazione da eseguire, l'evento principale utilizzato è

```
window.chrome.webview.addEventListener('message', event => { ... });
```

Gli eventi gestiti sono principalmente i seguenti:

- **initialize-uploader** → inizializza la view con la lista dei file già presenti nella coda di caricamento
- **update-view** → aggiorna o aggiunge una entry nella lista
- **update-upload-progress** → aggiorna la visualizzazione della barra di caricamento per mostrare la percentuale di caricamento dei chunk
- **clean-uploader** → ripulisce la view in caso di logout

Questa suddivisione permette di modellare in tempo reale l'interfaccia, senza necessità di refresh manuali e con una risposta agli eventi efficace e pronta, basandosi su casi d'uso specifici e funzioni dedicate.

```
  window.chrome.webview.addEventListener('message', event => {
switch (event.data.action) {
      case "update-view":
          let file = event.data.content;
          if(!checkExisting(file.Id))
              addRow(file);
          else
              updateRow(file);
          break;
```

```
        case "update-upload-progress":
            const id = event.data.content.file.Id;
            const percentage = event.data.content.percentage;
            updatePercentage(id, percentage);
            break;
        case "clean-uploader":
          ...
            cleanUploader()
            break;
        case "initialize-uploader":
          ...
            initializeUploader(event.data.content);
            break;
        }
    });
```

La funzione addRow(file) ad esempio, crea dinamicamente gli elementi necessari per visualizzare le proprietà principali del file, come il nome, l'edificio, il percorso all'interno della root directory. La più importante è forse l'icona di stato, che comunica all'utente se il file specifico è:

- in attesa di sincronizzazione (**TO_SYNC**)
- in corso di sincronizzazione (**SYNCHRONIZING**)
- sincronizzato correttamente (**SYNCED**)
- in errore (**ERROR**)

Questi stati sono definiti all'interno di un oggetto Status con la funzione di enumeratore:

```
const Status = {
    TO\_SYNC: 0,
    SYNCED: 1,
    SYNCHRONIZING: 2,
    ERROR: 3
};
```

Principi di design

L'intero script è costruito secondo un principio di modularità logica e reattività visiva: ogni funzione svolge un ruolo ben preciso, mantenendo separata la logica di business dalla presentazione. La comunicazione tra i due lati del ponte rimane asincrona e bidirezionale, favorendo un'interazione fluida da parte dell'utente. Dal punto di vista architetturale si tratta di una mini Single Page Application incapsulata del modulo, che aggiorna solo le porzioni di interfaccia necessarie, riducendo al minimo il consumo di risorse e migliorando la reattività percepita.

E' un componente funzionale del sistema di sincronizzazione che, attraverso un dialogo costante con il backend, gestisce in tempo reale lo stato dei file, fornisce feedback visivi all'utente e garantisce una rappresentazione coerente dell'avanzamento dei processi in background. L'interfaccia diventa a tutti gli effetti una dashboard di monitoraggio attivo, in grado di notificare l'utente in modo intuitivo e immediato, mantenendo un'architettura logica pulita e facilmente estendibile.

DTO e Services

Il flusso di informazioni scambiate con la base dati locale e le API remote viene modellato attraverso l'utilizzo di specifici DTO e Services, elementi centrali per mantenere una separazione chiara tra i livelli.

Come già accennato per il modulo Launcher, fungono da formattazione standard per i dati scambiati tra le componenti del sistema, includono entità che descrivono sia lo stato dei file che le informazioni necessarie per la gestione delle sincronizzazioni.

I DTO **FileStatus** e **FileStatusType** rappresentano le due unità logiche base dell'intero processo di upload, un file e lo stato del suo ciclo di vita: il primo contiene le sue proprietà necessarie per tracciare l'avanzamento della sincronizzazione, mentre il secondo codifica lo stato attuale in uno dei quattro stati fondamentali in cui un file si può trovare: **TO_SYNC, SYNCHRONIZING, SYNCED, ERROR**.

Il **SyncInfoDTO** dal canto suo consente di verificare se un file è stato modificato rispetto all'ultima sincronizzazione, ottimizzando così le operazioni di upload ed evitando caricamenti inutili. Altri DTO definiscono le informazioni necessarie per inizializzare un upload (come il numero e la dimensione dei chunk), altri notificano lo stato di avanzamento. L'insieme di queste strutture dati permette al sistema di aggiornare dinamicamente lo stato di upload e di gestire le informazioni da scambiare per le fasi di download e upload.

Infine, il **TokenDTO** rappresenta la struttura dedicata alla gestione dei token di autenticazioni scambiati con l'IAM, integrando il flusso di sicurezza con il TokenService condiviso con l'SDK.

Logica applicativa e gestione della persistenza

I servizi costituiscono il cuore operativo del modulo, ogni service incapsula un insieme ben definito di operazioni, isolando comunque le operazioni di accesso ai dati a quelle di comunicazione esterna al modulo.

Il **MetadataService** mantiene la consistenza tra le versioni dei file locali e remoti, le operazioni principali includono:

- **GetMetadataByFileName()**, che recupera i metadati di un file in base al nome e alla sua natura (nuovo o già sincronizzato);
- **UpdateFileIdAndVersion()**, che aggiorna gli identificativi remoti e il numero di versione dopo la creazione di un nuovo file sul server;
- **UpdateHashNewFilesByFileName()**, utilizzato per aggiornare l'hash dei nuovi file che non hanno ancora una versione.

Il **SyncService** dal canto suo gestisce le informazioni relative all'ultimo stato di sincronizzazione, restituendo l'ultimo hash sincronizzato o aggiornandolo dopo un'upload riuscito.

Ma è l'UploaderService a contenere la logica più complessa, è responsabile della gestione vera e propria del caricamento dei file, coordina i processi di rete, la logica di autenticazione e gli aggiornamenti locali.

Si distinguono, al suo interno, tre fasi principali del flusso operativo:

1. il punto di ingresso principale verifica la presenza del token di accesso, recupera i metadati del file e stabilisce se si tratti di un file nuovo o di una nuova versione di un file esistente, invocando poi gli endpoint appropriati.

2. dopodiché suddivide il file in chunk secondo la dimensione specificata dal server e li carica uno per uno, monitorando lo stato di avanzamento e aggiornando l'interfaccia

3. al termine del caricamento, il file viene marcato come Synced e viene aggiornato lo stato del sistema

Anche questo punto evidenzia come l'approccio seguito favorisce una comunicazione asincrona relativa agli eventi che, se gestiti correttamente, permettono di mantenere il sistema reattivo anche in presenza di upload di grandi dimensioni, garantendo la scalabilità e la resilienza del processo.

La gestione concorrente della coda di upload

La classe UploadManager rappresenta uno dei componenti chiave nella gestione del ciclo di sincronizzazione dei file: mentre i service si occupano della logica di business e della comunicazione con il database o con il server, questa classe funge da coordinatore dei processi di upload, gestendo in maniera thread-safe la coda dei file da sincronizzare. Si assicura che i file vengano caricati uno alla volta, rispettando l'ordine di inserimento nella coda senza creare conflitti.

Al centro di questa classe vi è la definizione di una coda concorrente di oggetti di tipo FileStatus (**ConcurrentQueue<FileStatus>**), una struttura dati adatta a gestire in sicurezza l'accesso da più thread senza la necessità di blocchi espliciti su ogni operazione. Sebbene al momento non venga supportato l'upload mutliplo simultaneo, è stata scelta per la sua predisposizione ad essere adattata più avanti, dato che diversi thread o processi potrebbero richiedere contemporaneamente il caricamento di nuovi file, e l'utilizzo della coda concorrente permette di accodare in modo sicuro ciascun file in attesa di elaborazione.

A supporto di questa logica, il manager mantiene un flag booleano, **isUploading**, e un lock per garantire che non vengano avviati più cicli di upload nello stesso momento. Questo approccio a metà tra concorrenza per la coda e sincronizzazione esplicita tramite il lock, consente di bilanciare efficienza e sicurezza, evitando race condition.

Interazione con l'UploaderService

Il manager non agisce da solo, esso delega la logica effettiva di caricamento al servizio UploaderService, che esegue la sincronizzazione vera e propria con il server. Il collegamento tra queste due classi è possibile attraverso la gestione di un sistema di eventi e callback, sottoscrivendo l'evento UploadCompleted esposto dal service:

```
uploaderService.UploadCompleted += OnUploadCompleted;
```

Quando un file viene sincronizzato con successo, viene invocato **OnUploadCompleted()** che a sua volta scatena l'evento **PendingUpload**, notificando al sistema che il file è stato correttamente processato. Questo e altre tipologie di eventi possono essere intercettati da altre componenti come l'interfaccia grafica o il servizio di log, per aggiornare in tempo reale lo stato dei caricamenti.

Così facendo il manager non gestisce solo il flusso di lavoro, ma funge da entità media per la comunicazione tra la logica di upload e il livello superiore dell'applicazione.

Gestione della coda di caricamento

Attraverso l'utilizzo di **ScheduleUploadFile()**, ogni volta che un nuovo file deve essere sincronizzato, viene accodato nella coda tramite il metodo **Enqueue**, passando come parametro la struttura di tipo FileStatus che rappresenta il file.

Successivamente viene chiamato **StartUpload()**, al cui interno viene eseguito un ciclo che processa tutti i file all'interno della coda:

```
while (fsq.TryDequeue(out FileStatus file))
{
    file.Status = FileStatusType.Synchronizing;
    if(!file.new_file)
        PendingUpload?.Invoke(file);
    await uploaderService.SyncFile(file);
}
```

Durante questo ciclo:

- ogni file viene contrassegnato come Synchronizing
- viene notificato il suo stato di avanzamento
- viene avviato l'upload effettivo chiamato **SyncFile**, metodo esposto dall'UploaderService

Quando la coda si svuota, il flag isUploading viene riportato a false, segnalando che il sistema è pronto per accettare nuovi file. Questo approccio permette di processare i caricamenti in modo sequenziale ma asincrono.

La logica adottata permette di gestire con flessibilità la prioritizzazione dei file da caricare, la ripresa automatica dopo un riavvio del servizio, l'integrazione con un sistema di logging centralizzato, separando la gestione della coda dalla logica di upload.

L'UploaderManager svolge quindi un ruolo importante all'interno dell'architettura, unendo la logica di persistenza con quella operativa, le sue responsabilità possono essere così riassunte:

- gestione della coda di upload in maniera concorrente e thread-safe
- orchestrazione del ciclo di sincronizzazione garantendo un caricamento alla volta in maniera ordinata rispetto all'ordine di arrivo
- comunicazione e notifica degli eventi di avanzamento e completamento verso l'esterno
- sincronizzazione dello stato dei file con il resto del sistema

L'uso combinato di strutture dati thread-safe, lock mirati e task asincroni rende il sistema efficiente, ma anche facilmente estendibile con nuove funzionalità. E' sicuramente l'elemento centrale in termini di coordinamento nel modulo Uploader, orchestrando il lavoro dei servizi sottostanti e garantendo la continuità del flusso di sincronizzazione in background.

Monitoraggio in tempo reale del file system

Passiamo ora ad analizzare il ponte tra file system locale e la logica di sincronizzazione dell'applicazione: il Watcher.

Ha il compito di monitorare in maniera continua la cartella locale in cui vengono scaricati i file remoti, intercettando in tempo reale le modifiche e reagendo in maniera coerente, aggiornando lo stato interno del sistema e, quando necessario, dare l'avvio alla procedura di sincronizzazione.

La classe si fonda sul componente .NET **FileSystemWatcher**, che consente di monitorare in modo asincrono una directory per rilevare eventi di modifica. Nel suo costruttore, la classe Watcher inizializza un'istanza di questo componente facendolo puntare alla cartella gestita dal programma, il cui path è memorizzato all'interno delle proprietà statiche esposte dall'SDK, definendo i tipi di eventi da osservare:

```
watcher = new FileSystemWatcher(AppPaths.FilesFolder);
watcher.NotifyFilter = NotifyFilters.CreationTime |
            NotifyFilters.FileName | NotifyFilters.LastWrite;
```

In questo modo vengono filtrati gli eventi, reagendo solamente a quelli realmente significativi, come la creazione o l'editing, e registrandoli attraverso degli handler specifici, così da eseguire le routine corrette quando questi eventi si verificano. Una volta registrati, il componente viene attivato settando a true la proprietà **EnableRaisingEvents**: da questo momento ogni modifica nella cartella osservata viene catturata e segnalata quando accade, consentendo al sistema di reagire dinamicamento.

La classe mantiene inoltre una lista di oggetti FileStatus, che rappresenta lo stato corrente dei file monitorati, assieme ad una mappa **lastCreated**, utile come meccanismo di **debouncing** per evitare notifiche duplicate dovute alla scrittura su disco, che anche in caso di creazione di un singolo file genera più eventi: questo comportamento è dovuto al fatto che, in base alla dimensione, è possibile che un file occupi più blocchi su disco, e che quindi, le scritture siano molteplici.

Scansione iniziale e ricostruzione dello stato locale

All'avvio, il Watcher esegue una scansione completa della cartella dei file locali, ricostruendo lo stato iniziale dei file e capendo quali siano già sincronizzati e quali invece devono essere aggiornati.

- per ogni file nella cartella, viene generato un hash SHA256, confrontato poi con la versione salvata all'interno del database locale
- a seconda del risultato, viene asseganto lo stato appropriato:
- viene poi notificata la lista al sistema, permettendo al resto del modulo di allinearsi allo stato reale dei file

La logica reattiva del Watcher si basa su una serie di callback dedicate, ciascuna delle quali gestisce un tipo specifico di evento.

L'evento **OnChanged** viene sollevato quando un file esistente viene modificato, controllando che non si tratti di un evvento duplicato tramite il debouncing citato in precedenza, e procedendo al confronto dell'hash attuale con quello registrato:

```
if (hash != file.Hash)
{
    file.Hash = hash;
    file.Status = FileStatusType.To\_Sync;
    OnFileStatusListUpdated?.Invoke(file);
    um.ScheduleUploadFile(file);
    syncService.UpdateLastHashByFileName(file.FileName, file.Hash);
}
```

In questo modo ogni modifica locale viene immediatamente identificata e inserita nella coda di upload, assicurando che il server venga aggiornato al più presto con la versione più recente del file.

L'**OnCreated** invece, viene richiamato quando un nuovo file compare nella cartella monitorata, quindi quando viene effettuato un download. La gestione è più complessa, dato che il sistema deve distinguere tra un file effettivamente nuovo o uno rigenerato da un processo esterno (i file generati da software come EC700 eliminano e riscrivono i file anziché modificarli).

Il metodo interroga il service di metadati quindi per ottenere le informazioni necessarie, e a seconda che questi dati vengano trovati o meno, il file viene classificato di conseguenza:

- se è nuovo, lo stato è To_Sync e viene programmato l'upload
- se è già esistente, lo stato è Synced

Controllo dell'integrità

E' interessante approfondire la generazione dell'hash

```
using (FileStream fs = new FileStream(path, FileMode.Open,
FileAccess.Read, FileShare.ReadWrite))
using (SHA256 sha256 = SHA256.Create())
{
    byte[] hash = sha256.ComputeHash(fs);
}
```

L'opzione **FileShare.ReadWrite** consente di leggere il file anche quando è in uso da altri processi, essendo che alcuni software (come EC700) possono mantenere i file aperti per operazioni di salvataggio prolungate. Questo meccanismo risulterà particolarmente vantaggioso nella misura in cui il sistema continui ad espandersi e migliorare dal punto di vista del consumo delle risorse, in quanto sarà possibile avviare processi secondari che provino a leggere in maniera esclusiva il file: dal momento in cui la lettura esclusiva non è possibile, significa che l'utente sta ancora editando il file specifico e quindi l'upload può essere posticipato a quando la lettura verrà eseguita correttamente, permettendo di avviare l'upload quando l'utente ha concluso le sue operazioni e ha chiuso il programma di editing.

Il Watcher si dimostra quindi in grado di assicurare una sincronizzazione continua, affidabile e automatizzata, rappresenta una sorta di **sentinella** del modulo Uploader, un componente discreto ma essenziale, che traduce gli eventi che occorrono nel file system in azioni concrete per il sistema di sincronizzazione.

Ponte di comunicazione tra logica applicativa e interfaccia WebView2

Ultima classe delle utils che ha un'importanza specifica è il **WebViewManager**, responsabile della gestione della comunicazione tra l'ambiente nativo e i contenuti web. Svolge un ruolo di mediazione tra la logica applicativa e il livello di presentazione, in modo da comunicare in tempo reale con l'interfaccia utente integrata nella WebView2. Pur operando come un servizio in background, l'Uploader offre una UI minimale in grado di visualizzare lo stato corrente della sincronizzazione, offrendo feedback immediati all'utente.

Oltre a caricare gli assett statici, traduce gli eventi provenienti dal backend in aggiornamenti dinamici sul frontend, l'obiettivo di questa duplice responsabilità è chiaro: isolare la logica di render in un modulo dedicato e garantire aggiornamenti reattivi allo stesso tempo, utilizzando un canale standardizzato e sicuro.

Gestione delle risorse ed inizializzazione

Attraverso il metodo **EnsureCoreWebView2Async** ci si assicura che l'ambiente WebView2 sia pronto all'uso, successivamente viene mappato un dominio virtuale sulla cartella che contiene gli asset locali tramite la funzione **SetVirtualHostNameToFolderMapping**. L'approccio è identico a quello utilizzato per l'UI del launcher, vengono caricati i file statici come se provenissero da una risorsa remota. In questo modo l'interfaccia viene caricata con un URL virtuale, inoltre viene evitato l'utilizzo di percorsi assoluti, semplificando il deployment e garantendo compatibilità con i moderni standard di sicurezza. Infine, viene registrato l'handler per gestire gli eventi provenienti dal front-end con:

```
webView.CoreWebView2.WebMessageReceived += WebMessageReceived;
```

In base al tipo di evento verranno eseguite operazioni diverse, ma l'importanza di questo metodo è che rende la comunicazione **bidirezionale**: non solo il backend invia aggiornamenti all'UI, ma la UI stessa può innescare azioni interne all'Uploader.

Comunicazione event-driven

Il WebViewManager contiene per la maggiorparte metodi pubblici di aggiornamento, ognuno dei quali rappresenta un messaggio strutturato inviato al front-end sotto forma di JSON, ma il modello utilizzato rimane lo stesso: costruire un oggetto contenente due proprietà chiamate **action** e **content**, per poi serializzarlo in formato JSON.

Ciascun messaggio viene poi interpretato dallo script Javascript dell'Uploader, aggiornando dinamicamente il DOM. In questo modo si mantiene un'interfaccia reattiva e coerente con lo stato del servizio, senza introdurre dipendenze dirette tra il codice C# e la logica di frontend.

Tutti i metodi convergono nel metodo privato PostJsonMessage(), che si distingue perché affronta una problematica importante: **garantisce sempre che l'accesso alla WebView2 avvenga sempre nel thread dell'interfaccia grafica**, evitando errori a runtime o legati all'accesso concorrente, così che il messaggio venga sempre inviato in modo sicuro.

Integrazione del WebView Manager con il resto del sistema

Questa classe non agisce in isolamento, ma rappresenta l'ultimo tassello del flusso di sincronizzazione. Con il WebViewManager si chiude il ciclo operativo dell'Uploader: ciò

che il Watcher rileva e l'UploaderManager elabora, viene reso visibile e notificato all'utente grazie a questo ultimo componente.

Si tratta di un elemento chiave nella realizzazione di un servizio che, pur agendo in background, riesce a fornire feedback costante, trasparente e in tempo reale sullo stato di sincronizzazione.

3.4 Common Library

Il modulo Common Library contiene il codice dell'SDK, è il modulo che garantisce la condivisione di funzionalità trasversali tra Launcher e Uploader.

In questo modo viene garantita coerenza e riutilizzo dei dettagli implementativi e delle logiche funzionali adottate. L'obiettivo era quello di centralizzare la logica comune e offrire un punto di riferimento unico per operazioni fondamentali come l'autenticazione, la comunicazione con le API di back-end e la gestione del logging.

L'intera struttura è composta da un insieme di moduli più specializzati, ciascuno con un preciso ambito di responsabilità:

- **API**: contiene le classi che modellano gli endpoint principali che Launcher e Uploader utilizzano, assieme ad altre più generiche come QuotaService, che implementa la funzionalità di polling periodico per il monitoraggio semi-realtime della quota utilizzata dall'utente, evitando di implementare meccanismi più pesanti e complessi come i WebSocket.
- DEV AUTH FLOW: racchiude al suo interno la logica di autenticazione con Keycloak, gestenndo l'intero flusso di scambio delle credenziali e la persistenza dei token resituiti dall'IAM aziendale. In questo modo viene isolata la parte più sensibile del sistema e resa modulare, così da garantire che sia il Launcher che l'Uploader possano autenticarsi e operare in sicurezza.
- REPOS - SERVICES: contengono la definizione dei componenti dedicati alla gestione dei token e delle richieste verso il back-end. All'interno è presente anche la definizione della classe AuthReqExecutor che implementa il meccanismo di retry automatico sfruttando l'offline token. In questo modo è la libreria ad essere responsabile del rinnovo automatico dell'access token quando scade, non è richiesto un intervento manuale dell'utente, è la libreria che lo rinnova in automatico.
- UTILS: costanti globali e utility comuni, tra cui la classe AppPaths che centralizza la gestione dei percorsi di file e cartelle, assieme alla configurazione del sistema di logging centrale.

La Common Library funge così da collante tra i due moduli principali, fornendo gli strumenti essenziali per autenticare, comunicare e monitorare ciò che accade all'interno dell'applicazione, assicurando al tempo stesso manutenibilità e scalabilità.

3.4.1 Astrazione degli endpoint di back-end

L'obiettivo è quello di centralizzare le chiamate al back-end fornendo un'interfaccia semplice e diretta che Launcher e Uploader possano utilizzare senza doversi preoccupare troppo dei dettagli implementativi delle richieste HTTP. In questo modo si migliora la manutenibilità del codice e il codice duplicato viene ridotto drasticamente.

La classe LauncherService fornisce i metodi per recuperare progetti ed edifici disponibili per l'utente autenticato, navigare il file tree di un edificio e scaricare i documenti dal cloud. L'aspetto centrale è però dato dall'utilizzo del metodo AuthenticatedRequestExecutor.SendWithAutoRefreshAsync, che incapsula al suo interno la logica di retry automatico in caso di scadenza del token.

```
public static async Task<byte[]> DownloadFile(HttpClient client, string fileId)
{
    var content = new MultipartFormDataContent();
    content.Add(new StringContent(fileId), "fileId");

    var response = await AuthenticatedRequestExecutor.SendWithAutoRefreshAsync(
        client,
        requestFactory: () =>
        {
            var request = new HttpRequestMessage(HttpMethod.Get, AppPaths.RemoteUri + "file/download");
            request.Content = content;
            return request;
        },
        tryRefreshTokenAsync: AuthManager.TryRefreshAccessTokenAsync
    );

    response.EnsureSuccessStatusCode();
    return await response.Content.ReadAsByteArrayAsync();
}
```

Figura 3.6: Struttura base delle chiamate Http

L'uso di una request factory permette di costruire la richiesta HTTP in maniera differita e modulare, in quanto invece di istanziare l'oggetto HttpRequestMessage direttamente all'interno della logica di download, lo si delega ad una funzione che viene invocata al momento opportuno. Così il codice risulta più flessibile e facilmente estendibile dato che lo stesso meccanismo può essere riutilizzato in altri contesti variando solo i parametri di costruzione della request stessa.

In secondo luogo, l'integrazione con il **SendWithAutoRefreshAsync** consente di intercettare in automatico i casi in cui il token di accesso è scaduto, evitando di dover scrivere codice duplicato per ogni chiamata API. Il retry viene applicato in maniera trasparente, senza che il servizio chiamante se ne debba preoccupare.

In sintesi, la combinazione di factory methods e retry automatico migliora la qualità generale del codice, riflettendo anche la volontà di inserire un patter architetturale volto a separare in maniera netta le responsabilità per quanto riguarda la comunicazione con il back-end.

La classe **UploaderServiceApi**, invece, implementa gli endpoint dedicati all'inizializzazione dei flussi di upload, distinguendo i casi in cui l'utente stia cercando di caricare un nuovo file (**PostNewFile**) oppure di registrare una nuova versione di un file già esistente (**PostVersion**).

In entrambi i casi, l'output non è l'upload del file stesso, ma la creazione di un identificativo e di una sessione di caricamento che saranno poi utilizzati per iniziare l'upload in chunk. Sono quindi chiamate di "prenotazione" che non trasferiscono ancora dati binari, ma consentono al server di generare un contesto, in modo che il caricamento dei blocchi possa avvenire in modo sicuro. Entrambi i metodi ricevono e utilizzano le informazioni riguardanti i metadati dei file inclusi all'interno di un payload JSON, per poi delegare l'esecuzione della richiesta a **SendWithAutoRefreshAsync** che, come nel caso precedente, gestisce automaticamente il refresh del token in caso di scadenza.

Questa classe quindi si preoccupa di preparare il sistema a ricevere nuovi contenuti, rispettando le regole di versioning e consistenza richieste dall'applicazione.

La classe **QuotaService** dal canto suo, ha il compito di interrogare il back-end per recuperare informazioni puntuali sullo spazio disponibile e sul consumo della quota associata all'utente loggato. Il metodo principale, GetUserQuota, effettua una richiesta HTTP verso l'endpoint dedicato (**/user/quota**), restituendo i dati in formato JSON.

Anche in questo caso il retry automatico del token risulta fondamentale per simulare una sessione di aggiornamento real-time, monitorando in maniera periodica tramite polling la quota utente. Questa scelta permette di raggiungere un compromesso importante, pur non utilizzando tecniche più complesse come WebSocket e SignalR l'approccio basato su polling risulta sufficiente a garantire reattività, notificando l'utente nel caso sia necessario liberare risorse o acquisire ulteriore spazio.

Considerazioni generali

L'elemento più rilevante nella progettazione di queste API è la standardizzazione dell'accesso al back-end, tutte le classi seguono lo stesso patter implementativo basato su dei principi che all'interno delle classi si ritrovano sempre:

- l'uso di un HttpClient condiviso riduce overhead e duplicazione codice
- i FactoryMethods permettono di incapsulare in maniera efficiente i parametri comuni delle varie chiamate
- il retry automatico permette di garantire robustezza e continuità del servizio

3.4.2 Device Authorization Flow

Il seguente modulo costituisce il cuore della logica di autenticazione dell'applicazione, realizzato secondo il protocollo **OAuth2 Device Authorization Grant con PKCE (Proof Key for Code Exchange)**.

Questa modalità si presta molto bene in scenari in cui il client non dispone di una UI tradizionale per inserire in maniera diretta le credenziali, come avviene spesso per applicazioni desktop che demandano la fase di login al browser.

L'obiettivo principale di questo modulo è fornire un flusso di autenticazione sicuro e trasparente per l'utente, in modo che login, refresh e logout vengano effettuati senza richiedere ulteriori interventi manuali, affidandosi ad un IAM riconosciuto e affidabile come Keycloak.

La classe **AuthManager** centralizza le logiche utilizzate per le principali operazioni di autenticazione:

- il metodo **LoginWithDevFlowAsync** per il Login con Device Flow e PKCE
- il metodo **TryRefreshAccessTokenAsync** per il refresh automatico dell'access token
- il Logout, che invalida i token forniti dall'Identity Provider e rimuove eventuali dati temporanei dal sistema locale

Soprattutto all'interno di questo modulo diventa importante separare le responsabilità. L'AuthManager non gestisce direttamente il salvataggio dei token, ma si appoggia ad

un'interfaccia creata ad hoc (ITokenService) che espone i metodi e le chiamate da effettuare per salvarli e gestirli in maniera corretta.

Tuttavia, è bene specificare come il flow di login si articoli in due fasi ben distinte: **inizializzazione e polling**.

```
public static async Task LoginWithDevFlowAsync(Func<Task> postLoginCallback)
{
    var pkceUtils = new PkceUtils();

    var codeVerifier = pkceUtils.generateCodeVerifier();
    var codeChallenge = pkceUtils.generateCodeChallenge(codeVerifier);

    var json = await DeviceFlowAuth.SendChallengeRequest(AppPaths.AuthUrl, codeChallenge);

    AuthSession.Clear();

    var parsedJson = JsonSerializer.Deserialize<JsonObject>(json)
                     ?? throw new NullReferenceException("JSON di risposta nullo");

    var deviceCode = parsedJson["device_code"]?.Deserialize<string>()
                    ?? throw new NullReferenceException("device_code nullo");
    var verificationUri = parsedJson["verification_uri_complete"]?.Deserialize<string>()
                          ?? throw new NullReferenceException("verification_uri nullo");
    var delay = parsedJson["interval"]?.Deserialize<int>() ?? 5;
    var expiresIn = parsedJson["expires_in"]?.Deserialize<int>() ?? 300;

    OpenBrowser(verificationUri);
```

Figura 3.7: Inizializzazione

In questa prima figura è possibile analizzare la parte di codice che si occupa di avviare il processo di autenticazione. In particolare, tramite l'ausilio della classe PkceUtils, vengono creati un **code verifier** e il corrispondente **code challenge**, cruciali per garantire che la richiesta sia legata crittograficamente al client e non possa essere riutilizzata da terzi.

Tramite il metodo **SendChallengeRequest**, viene inviata la challenge all'endpoint di autorizzazione di Keycloak, che risponde con un payload contenente i dati essenziali per proseguire con il flow di autenticazione:

- **device code**, identificatore univoco della richiesta di autorizzazione
- **verification uri complete**, il link a cui l'utente verrà reindirizzato per autenticarsi
- **interval** e **expires in**, che definiscono la frequenza e la durata del polling utilizzato per verificare se l'utente ha inserito o meno le credenziali

Viene comunque demandata a Keycloak la gestione e presentazione dell'interfaccia utente per l'inserimento delle credenziali, così da rinforzare lo stile di design secondo cui l'applicazione si deve limitare ad orchestrare il flusso, ma rimanga esterna rispetto alla gestione delle password e all'issue dei token di autenticazione.

In questo secondo snippet di codice invece viene mostrato il ciclo di polling che rappresenta la seconda fase del device flow. L'applicazione effettua richieste periodiche, tramite il metodo **SendVerifyRequest**, all'endpoint dedicato al rilascio dei token, in attesa che l'utente completi l'autenticazione nel browser.

Quando il server restituisce HTTP 200 (OK), la validazione può ritenersi completata con successo e, deserializzando dal payload JSON ricevuto, riceve l'access e l'offline

```
for (var i = 0; i < expiresIn / delay; i++)
{
    await Task.Delay(delay * 1000);
    var (response, code) = await DeviceFlowAuth.SendVerifyRequest(
        AppPaths.TokenUrl, codeVerifier, deviceCode);

    if (code == HttpStatusCode.OK)
    {
        var tokenJson = JsonSerializer.Deserialize<JsonObject>(response)
                        ?? throw new NullReferenceException();

        var access_token = tokenJson["access_token"]?.Deserialize<string>()
                           ?? throw new NullReferenceException("access_token nullo");
        var refresh_token = tokenJson["refresh_token"]?.Deserialize<string>()
                            ?? throw new NullReferenceException("refresh_token nullo");

        var handler = new JwtSecurityTokenHandler();
        var jwtToken = handler.ReadJwtToken(access_token);
        var username = jwtToken.Claims.First(c => c.Type == "preferred_username").Value;

        AuthSession.SetSession(access_token, refresh_token, username);
        _tokenService.Save(access_token, "access");
        _tokenService.Save(refresh_token, "offline");

        // Callback per eseguire poi la loadUserInfo
        if (postLoginCallback != null)
            await postLoginCallback();

        break;
    }
}
```

Figura 3.8: Polling

token, che vengono poi analizzati utilizzando **JwtSecurityTokenHandler** per estrarre il nome utente, salvando sia in memoria che sullo storage persistente le informazioni utili al mantenimento della sessione. La callback che viene eseguita alla fine del metodo è pensata per svolgere operazioni di inizializzazione post-login, come il caricamento delle informazioni utente dal back-end.

Per come il flow è strutturato, si mantiene l'applicazione in uno stato reattivo non bloccante, dove l'unico intervento richiesto è la conferma del login all'interno del browser.

La classe **AuthSession**, dal canto suo, rappresenta la parte più leggera ma cruciale del modulo, dato che mantiene in memoria i token correnti e lo username dell'utente autenticato, espone inoltre una proprietà IsAuthenticated che permette al resto dell'applicazione di verificare lo stato della sessione in maniera semplice e veloce. Si limita quindi a rendere disponibile lo stato corrente.

Infine, la classe PkceUtils permette di garantire la sicurezza del device flow tramite l'utilizzo del meccanismo **Proof Key for Code Exchange**. Si occupa di generare il code verifier casuale e criptato, derivando da esso un code challenge ottenuto applicando SHA256 come algoritmo di crittografia e Base64Url come codifica. In questo modo si impedisce che eventuali intercettazioni possano riutilizzare i dati scambiati, impersonando l'utente, dato che anche ottenuta la challenge, risalire al code verifier risulta proibitivo computazionalmente.

Nel complesso, l'implementazione del flusso di autenticazione OAuth2 con PKCE risulta completa e robusta, rendendo il sistema scalabile e sicuro, minimizzando i rischi associati

alla gestione di credenziali sensibili, e con un'equa divisione delle responsabilità.

3.4.3 Repository e Service

La struttura del modulo common lib si compone anche di una sezione costruita secondo il pattern repository-service, in modo da separare la logica di persistenza dei dati da quella più di business ed orchestrazione.

Le classe definite all'interno del repository rappresentano il livello di accesso ai dati e si occupa principalmente della lettura, scrittura e cifratura dei token nel database locale SQLite. Le varie interfaccie definiscono le operazioni principali come l'inizializzazione delle tabelle, il salvataggio dei token, la verifica della loro esistenza unito al recupero e alla cancellazione degli stessi.

Ci sono alcune funzionalità chiave che è bene sottolineare:

- i token vengono cifrati tramite ProtectedData.Protect prima di essere memorizzati come BLOB in SQLite, garantendo che possano essere decifrati solo dal pool di thread lanciati dall'utente corrente, servendonosi di DataProtectionScope.CurrentUser
- le query di inserimento utilizzano il costrutto ON CONFLICT(Type) DO UPDATE, in modo da poter aggiornare il token esistente evitando duplicazioni
- il metodo GetToken legge il BLOB e lo decifra utilizzando ProtectedData.Unprotect, restituendo il token all'applicazione

Il livello di astrazione implementato tra il service e il repository corrispondente permette un livello di astrazione tale da poter modificare la logica di persistenza senza impattare il codice che la utilizza.

Ultima tra queste classi è la classe statica AuthenticatedRequestExecutor, che permette di gestire automaticamente:

1. l'impostazione dell'header HTTP Authorization con il token corrente
2. il retry automatico in caso di risposta HTTP 401, invocando il metodo per aggiornare l'access token utilizzando l'offline token
3. l'aggiornamento delle informazioni relative alla sessione

Il pattern utilizzato permette di poter salvare in maniera sicura e affidabile i dati sensibili grazie alla cifratura, separare in maniera netta la logica di persistenza da quella applicativa e gestire in maniera automatica e centralizzata i token e le chiamate al back-end.

3.4.4 Sicurezza e cifratura dei token con ProtectedData

La gestione dei token e la loro protezione a riposo (il loro salvataggio locale) è sicuramente un aspetto fondamentale che è stato curato all'interno dell'applicazione.

La classe ProtectedData del namespace System.Security.Cryptography rappresenta un wrapper semplificato per le API di crittorgrafia di windows. In particolare, il metodo Protect consente di trasformare una stringa di testo in un array di byte cifrato, legato al contesto di sicurezza dell'utente corrente. I token, una volta salvati nel DB SQLite,

risultano illeggibili all'esterno e non possono essere decifrati neppure da altri utenti all'interno dello stesso sistema operativo.

Per recuperare il token, si utilizza il metodo inverso Unprotect, che ricostruisce la stringa originale solo se la Protect è stata eseguita sotto lo stesso profilo utente Windows che l'ha generata.

L'utilizzo di questa classe permette di non implementare manualmente algoritmi di crittografia e gestione delle chiavi, è una protezione garantita a livello di sistema operativo. Inoltre i token memorizzati non possono essere condivisi tra utenti diversi, riducendo i rischi in ambienti condivisi o con accesso remoto. Unito al fatto che a livello SQLite vengono salvati solo i BLOB binari cifrati, si è ritenuto che questa scelta garantisse un livello di sicurezza adeguato.

3.4.5 Centralizzazione dei percorsi principali e logging

Sono stati raggruppati componenti di supporto sia al Launcher sia all'Uploader, racchiusi nelle classi AppPaths e AppLogger.

La prima fornisce un punto unico di definizione per le costanti globali di configurazione, come i percorsi delle cartelle temporanee, del database locale e gli endpoint remoti utilizzati durante l'autenticazione o la sincronizzazione.

La seconda invece implementa un sistema di logging basato su Serilog, configurato per scrivere log sia in console sia su file con la possibilità di personalizzare il rolling (che in questo caso è mensile). Questo permette di avere una tracciabilità uniforme tra i diversi moduli, facilità le attività di debugging e fornendo uno storico fedele delle operazioni eseguite, utile per rilevare anomalie o comportamenti inattesi.

La divisione in questi due componenti permette da un lato la gestione centralizzata delle risorse statiche principali che l'applicazione utilizza, dall'altro la registrazione e il monitoring sistematico delle attività, entrambi punti cruciali per il caso d'uso che è stato preso in considerazione.

Capitolo 4

Conclusioni e Sviluppi Futuri

Il lavoro svolto per la progettazione e lo sviluppo del modulo di sincronizzazione e gestione dei file Edilclima nasce da una duplice prospettiva: da un lato, la necessità di integrare in maniera efficiente il lavoro dei clienti con l'infrastruttura cloud dell'azienda, dall'altro la volontà di costruire un applicativo solido ed estendibile all'intera suite di applicativi che Edilclima offre, che rimanga conforme a determinati principi di scalabilità e sicurezza.

Il progetto al momento è articolato quindi in due componenti principali - il Launcher e l'Uploader - che operano in maniera complementare: il primo si occupa dell'interfaccia utente e dell'interazione diretta con i file, mentre il secondo lavora in background, garantendo la sincronizzazione automatica tra ambiente cloud e repository locale.

Questa architettura ha permesso di raggiungere un equilibrio efficace tra usabilità e automazione, se i capitoli precedenti sono serviti ad analizzarla dal punto di vista tecnico e infrastrutturale, il capitolo finale analizzerà due aspetti:

- i riscontri ottenuti da Edilclima durante l'ultima presentazione del prototipo attuale, che hanno fornito un utile riscontro sull'efficacia e sulla maturità della soluzione proposta
- le prospettive di sviluppo futuro che delineano l'evoluzione dell'applicazione
- le considerazioni finali relative al progetto, con una sintesi degli obiettivi raggiunti e del valore introdotto nell'ecosistema Edilclima.

L'obiettivo del capitolo è quello di fornire una visione complessiva e consapevole del progetto, fungendo da punto di partenza per successive iterazioni e miglioramenti. L'applicazione rappresenta un passaggio significativo nel percorso di digitalizzazione dei processi aziendali, in quanto consente una più efficace integrazione dei servizi e dei dati all'interno dell'infrastruttura cloud di Edilclima.

4.1 Riscontri e valutazioni

Terminato lo sviluppo del prototipo, l'intero ambiente è stato testato direttamente con alcuni esponenti e responsabili del reparto tecnico di Edilclima, testandolo in un contesto reale e con file effettivamente utilizzati dai loro applicativi. La dimostrazione si è svolta utilizzando la versione più aggiornata dell'applicazione, permettendo di verificarne il corretto comportamento utilizzando file proprietari del software EC700.

Durante la sessione di test, l'applicazione ha confermato la stabilità e l'affidabilità delle principali funzionalità previste. La sincronizzazione dei file ha operato correttamente in entrambi i sensi, dimostrando che il sistema di monitoraggio in background, basato sul **FileSystemWatcher**, e la coda di upload gestita dall'**UploadManager** funzionano in modo coerente e coordinato.

Sono state inoltre analizzate nel dettaglio alcune componenti tecniche, come il meccanismo di autenticazione basato su Keycloak con l'utilizzo dell'offline token, e la gestione dell'interfaccia utente integrata tramite WebView2, ed entrambi gli aspetti hanno suscitato interesse.

Uno dei punti più discussi è stata la **gestione dello storage** e l'**ottimizzazione della sincronizzazione** dei file EC700. E' emersa infatti una problematica legata alla natura del formato ZIP utilizzato da questi file: anche in assenza di modifiche sostanziali, ogni salvataggio ricostruisce quasi integralmente l'archivio, alterando la maggior parte dei byte. Questo comportamento rende particolarmente complesso determinare se un file sia effettivamente cambiato e, di conseguenza, ottimizzare la fase di upload per evitare trasferimenti inutili di dati.

Sono state discusse possibili soluzioni, come l'analisi differenziale del contenuto interno degli ZIP, o l'adozione di algoritmi in grado di rilevare modifiche a livello di blocchi di byte, è stato sottolineato come un eventuale approccio incrementale, pur più complesso, potrebbe portare benefici significativi in termini di riduzione della banda utilizzata e dei tempi di sincronizzazione, soprattutto in contesti con connessioni limitate o grandi volumi di dati.

Nel corso della discussione, sono stati toccati anche temi di tipo gestionale e commerciale, come la possibilità di introdurre **diversi livelli di servizio** (ad esempio, una versione gratuita con limiti di spazio e una premium con funzionalità estese) e l'eventuale uso dei **realm roles di Keycloak** per differenziare quote e permessi in base al tipo di utente o al piano di abbonamento. Sono state inoltre proposte ulteriori estensioni funzionali, tra cui l'arricchimento del file picker con funzioni di rinomina, cancellazione e creazione di nuove cartelle, nonché la visualizzazione e il recupero delle **versioni precedenti dei file**, a supporto di una gestione più evoluta della cronologia dei progetti.

Nel complesso, il riscontro ottenuto è stato ampiamente positivo. Il referente tecnico di Edilclima ha sottolineato come il livello di maturità raggiunto dal progetto vada già oltre una semplice **proof of concept**, configurandosi come un prototipo completo e stabile, capace di dimostrare concretamente la validità dell'architettura proposta. L'interesse manifestato verso il codice sorgente e la volontà di esplorare un'integrazione più profonda con l'infrastruttura aziendale rappresentano segnali incoraggianti in vista delle future evoluzioni del sistema. Pur restando alcune sfide aperte - in particolare la generalizzazione del rilevamento delle modifiche su file di natura diversa - il prototipo ha dimostrato la solidità dell'approccio concettuale e l'efficacia della logica di sincronizzazione automatica su cui si fonda l'intera architettura.

4.2 Sviluppi Futuri

L'attuale versione del sistema ha dimostrato di essere una base solida per la gestione automatica della sincronizzazione dei file, ma al tempo stesso ha evidenziato numerose potenzialità di evoluzione. Le prospettive di sviluppo futuro mirano a rendere l'applicazione più efficiente, generalizzabile e integrata all'interno dell'ecosistema Edilclima, con un'attenzione particolare alla scalabilità e all'ottimizzazione delle risorse. In questa

sezione vengono delineate le principali direttrici di sviluppo identificate a seguito delle discussioni con il team tecnico aziendale.

Analisi differenziale e delta sync dei file

Uno degli sviluppi più rilevanti riguarda la possibilità di introdurre un sistema di sincronizzazione differenziale ("delta sync"), in grado di inviare al server soltanto le parti effettivamente modificate di un file, piuttosto che l'intero contenuto.

Tale approccio permetterebbe di ridurre in modo drastico la quantità di dati trasferiti e, di conseguenza, il tempo necessario per completare un'operazione di upload, con vantaggi evidenti in termini di efficienza e sostenibilità della rete.

Nel caso specifico dei file EC700, che sono archivi compressi in formato ZIP, la sfida consiste nel fatto che anche una minima modifica interna ne altera completamente la struttura binaria. Per affrontare questa criticità, è stato ipotizzato un meccanismo di analisi del contenuto interno dell'archivio:

- lo ZIP verrebbe "spacchettato" temporaneamente;
- ciascun file contenuto verrebbe confrontato con la versione precedente;
- solo i file effettivamente modificati verrebbero ricompressi e inviati al server.

Un approccio di questo tipo, se ben generalizzato, potrebbe essere esteso progressivamente a tutti i software Edilclima, analizzando la struttura dei formati specifici di ciascun applicativo. Ciò aprirebbe la strada a una logica di delta sync adattiva, capace di ottimizzare le risorse in modo trasversale rispetto alla tipologia di progetto o di file gestito.

Verso un cloud personale unificato

Un secondo obiettivo di medio periodo è la trasformazione dell'applicazione da un sistema di sincronizzazione legato ai singoli progetti o edifici a un cloud personale per gli utenti Edilclima.

In questa nuova visione, ogni utente disporrebbe di un proprio spazio dedicato, indipendente dalla struttura rigida basata su "Stato di Fatto", "Stato di Progetto" e simili. Il sistema diventerebbe così una piattaforma trasversale, accessibile da tutti i software Edilclima e integrata con l'infrastruttura cloud aziendale.

Questa evoluzione permetterebbe di ottenere un duplice risultato:

- un'esperienza utente più coerente e centralizzata, in cui i file di progetto non dipendono più dal singolo applicativo, ma appartengono all'utente e al suo profilo cloud;
- una maggiore modularità del sistema, che diventerebbe una componente comune a tutti i prodotti, favorendo l'unificazione delle logiche di storage, autenticazione e sincronizzazione.

Un'implementazione di questo tipo potrebbe anche consentire a Edilclima di introdurre diversi livelli di servizio (gratuito, premium, enterprise), per assegnare quote di storage e funzionalità avanzate in base al piano utente.

Ottimizzazione dei controlli e sincronizzazione condizionata

Un'altra direzione di sviluppo riguarda l'ottimizzazione dei controlli preliminari alla sincronizzazione.

Attualmente, l'Uploader rileva le modifiche a un file sulla base dell'hash calcolato, ma non verifica se il file sia ancora aperto o in uso da parte del programma principale. In alcuni casi, ciò può comportare tentativi di upload mentre il file è ancora in scrittura.

Una possibile evoluzione consiste nell'introdurre un processo secondario di validazione, che tenti l'apertura del file in sola lettura prima di avviare la sincronizzazione.

Se il file risulta bloccato, il sistema attenderà che l'utente perda il focus sull'applicativo di editing, avviando la sincronizzazione solo al termine effettivo della sessione di lavoro.

Questo accorgimento ridurrebbe i conflitti di accesso e migliorerebbe la stabilità complessiva del servizio, garantendo che vengano inviati al server solo i file realmente "a riposo".

Gestione intelligente dello storage locale

Per ottimizzare ulteriormente le risorse e mantenere un footprint leggero sul dispositivo dell'utente, si prevede l'introduzione di politiche di retention automatica dei file locali.

I file scaricati e sincronizzati verrebbero conservati soltanto per un periodo di tempo definito (ad esempio 7 o 15 giorni), trascorso il quale verrebbero rimossi automaticamente, mantenendo comunque la possibilità di riscaricarli dal cloud in qualsiasi momento.

Un tale approccio consentirebbe di:

- liberare spazio sui dispositivi degli utenti senza intervento manuale;
- migliorare le performance generali del sistema;
- ridurre il rischio di disallineamenti tra storage locale e remoto.

Introduzione del protocollo TUS per upload resilienti

Infine, un'evoluzione di grande rilievo tecnico riguarda l'adozione del protocollo TUS per la gestione dei caricamenti di file in modo resumable e affidabile.

TUS è uno standard aperto basato su HTTP che consente di riprendere un upload interrotto in qualsiasi momento, senza dover ricominciare da zero.

Questo lo rende particolarmente adatto a scenari come quello Edilclima, dove gli utenti possono trovarsi a lavorare con connessioni instabili o file di grandi dimensioni (come gli archivi EC700 o le relazioni PDF).

L'integrazione del protocollo TUS permetterebbe di ottenere:

- maggiore affidabilità nelle operazioni di upload anche in caso di interruzioni di rete;
- compatibilità multi-piattaforma e scalabilità lato server;
- una base tecnologica condivisa e standardizzata, riducendo la complessità delle implementazioni personalizzate.

L'obiettivo finale è quello di rendere le operazioni di upload realmente "resilienti", capaci di adattarsi automaticamente alle condizioni di rete, mantenendo la coerenza dei dati e migliorando la percezione di stabilità da parte dell'utente.

4.3 Considerazioni finali sul prototipo

Il sistema di sincronizzazione cloud di Edilclima nasce da un'esigenza chiara: semplificare e automatizzare la gestione dei file di progetto generati dalla loro suite di applicativi, riducendo al minimo l'intervento manuale dell'utente integrando l'applicazione nella maniera più naturale possibile con l'infrastruttura cloud dell'azienda. Dato che in origine la gestione di questi file avveniva tramite uno scambio manuale o attraverso procedure non uniformi, le versioni duplicate e il disallineamento tra gli utenti erano scenari frequenti: serviva una soluzione capace di offrire continuità operativa tra l'ambiente cloud e quello locale.

Da queste considerazioni ha preso forma un ecosistema composto da più moduli interconnessi, ciascuno con un suo ruolo specifico, definendo una struttura modulare e ben definita in grado di realizzare un sistema complesso, mantenendo chiara la logica di comunicazione tra i vari componenti. L'insieme dei moduli interagisce in modo fluido grazie ad una logica event-driven, ma è progettato in modo che ogni componente abbia una propria autonomia: Launcher e Uploader svolgono le loro funzioni indipendentemente l'uno dall'altro, e sono in grado di tornare attivi e sincronizzarsi sullo stato corretto.

L'utente finale non deve preoccuparsi del funzionamento interno, ma può comunque monitorare in tempo reale lo stato delle operazioni attraverso le interfacce dei due componenti costruite con WebView2, che uniscono l'efficienza del codice nativo .NET alla flessibilità e modernità delle tecnologie web. Un altro risultato significativo è rappresentato dall'introduzione di una sincronizzazione elastica e flessibile, basata su meccanismi di coda e gestione concorrente delle operazioni. Grazie all'UploaderManager e al Watcher, il sistema è pienamente in grado di:

- rilevare automaticamente le modifiche locali ai file
- inserirle in una coda di upload gestita in modo thread-safe
- eseguire gli invii al cloud in maniera sequenziale e sicura
- aggiornare lo stato dell'interfaccia senza bloccare il flusso principale dell'applicazione

Vista la finalità di utilizzo in background dell'uploader, questo modello di gestione asincrona della sincronizzazione garantisce robustezza e affidabilità.

Sicurezza e integrazione

Uno degli obiettivi raggiunti degni di nota riguarda sicuramente l'integrazione con i sistemi esistenti all'interno dell'infrastruttura Edilclima, essendo che non è stata pensata come un elemento isolato, l'applicazione risulta essere un'estensione coerente del cloud Edilclima, capace di integrare con le API e con i servizi di autenticazione che l'azienda utilizza.

L'integrazione con Keycloak soprattutto, per la gestione delle sessioni e dei token di accesso, ha permesso di garantire un livello di sicurezza elevato senza compromettere la fluidità dell'esperienza utente. L'utilizzo del refresh token consente di mantenere attive le proprie sessioni anche dopo lunghi periodi di inattività, rendendo il servizio completamente autonomo. In questo modo, grazie alla logica di retry automatico delle richieste, entrambe le componenti possono continuare le loro operazioni anche in assenza di interazione diretta, rispettando al tempo stesso le politiche di sicurezza aziendale.

Il progetto rappresenta un passo concreto nel percorso di digitalizzazione che sta portando l'azienda a migrare progressivamente da un ambiente completamente desktop a un ecosistema integrato in cloud, permettendo di:

- uniformare la gestione dei file di progetto tra i vari applicativi che l'azienda offre
- eliminare la necessità di scambio manuale
- ridurre il rischio di inconsistenze o versioni obsolete
- offrire agli utenti una sincronizzazione e condivisione dei file intuitiva e trasparente

Tutto ciò si traduce in un aumento dell'efficienza operativa, una riduzione dei tempi di lavoro e una maggiore collaborazione tra gli utenti. Pur trovandosi ancora in una fase prototipale, l'applicazione dimostra già oggi il potenziale per evolversi verso estensioni e funzionalità più avanzate, che verranno discusse nelle sezioni successive.

Considerazioni personali

Dal punto di vista dello sviluppo, questo progetto ha rappresentato sicuramente un'esperienza significativa di progettazione software complessa, visto il bisogno di conciliare componenti eterogenei (tecnologie web e .NET, sistemi di file watching, API cloud, autenticazione, il tutto in un ambiente distribuito e con una struttura già ben definita) in un unico prodotto coerente e funzionale.

La sfida principale è stata trovare un equilibrio tra robustezza architetturale e semplicità d'uso, assicurando al tempo stesso una comunicazione chiara tra i diversi livelli del sistema. Il risultato finale è un'applicazione stabile, moderna e progettata per durare, costruita su una base solida che potrà essere facilmente ampliata nel tempo.

Dal punto di vista metodologico, il lavoro ha permesso di approfondire temi centrali dello sviluppo moderno, come la progettazione orientata agli eventi, la gestione asincrona dei processi e la definizione di interfacce modulari tra componenti. Soprattutto, ha dimostrato come una buona architettura software possa tradursi in valore reale per l'azienda, fungendo da ponte tra innovazione tecnologica e miglioramento dei processi interni.

In conclusione, il progetto non rappresenta soltanto la realizzazione di un'applicazione di sincronizzazione, ma un passo strategico nella costruzione di un'infrastruttura cloud integrata per Edilclima. È una base solida da cui poter evolvere verso nuovi servizi e modelli di collaborazione, mantenendo al centro la stessa filosofia: unire semplicità, automazione e integrazione per rendere più fluido e naturale il lavoro dell'utente.