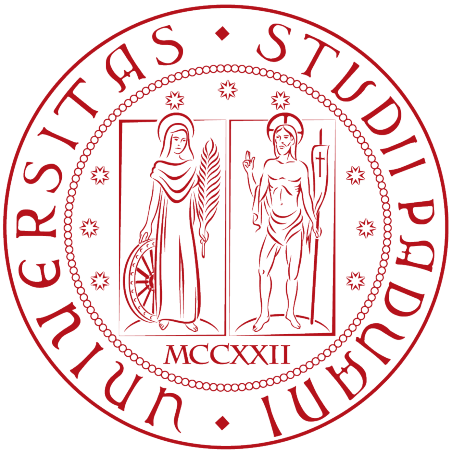
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Gestione sicura di chiavi crittografiche su dispositivi mobili

Tesi di Laurea

Relatore
Prof. Tullio Vardanega

Laureando
Gabriele Di Pietro
Matricola 2010000

Anno Accademico 2024–2025

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

"Life is really simple, but we insist on making it complicated"

— Confucius

Ringraziamenti

Padova, Dicembre 2025 Gabriele Di Pietro

Indice

Elenco delle figure

Elenco delle tabelle

Lista degli elementi di codice

* [1]
**List of Listings

Capitolo 1

Contesto aziendale

Il presente capitolo introduce l'azienda presso cui ho svolto l'attività di *stage*, **SyncLab S.r.l**, fornendo la base necessaria alla comprensione del progetto di tesi. Verranno descritti gli ambiti in cui l'azienda opera, gli strumenti e i processi di *Way of working* di cui ho avuto esperienza diretta e la sua propensione all'innovazione, elemento cruciale in relazione allo sviluppo di un'applicazione innovativa come quella illustrata nei futuri capitoli.

1.1 Presentazione azienda

L'azienda ospitante, *SyncLab S.r.l*, è stata fondata nel 2002 a Napoli e si è affermata fin da subito come una realtà innovativa attenta ai paradigmi della trasformazione digitale. Nel corso degli anni, la società ha intrapreso un significativo processo di espansione a livello nazionale, che l'ha portata ad aprire sei sedi operative distribuite sul territorio italiano.

Figura 1.1: Sedi operative dell'azienda - Fonte: synclab.it

Inizialmente nata come *software house*, l'azienda si è dedicata allo sviluppo di soluzioni *software* innovative, progettate *ex novo* in base alle opportunità e alle esigenze del mercato. In tale contesto, la società si distingue per la sua vocazione all'innovazione tecnologica e alla trasformazione digitale, realizzando prodotti e fornendo servizi in diversi settori strategici, tra cui quello sanitario, energetico, industriale, finanziario e logistico.

Nel tempo, *SyncLab S.r.l* ha ampliato il proprio raggio d'azione, assumendo un ruolo rilevante come *system integrator*. In questa veste l'azienda si occupa dell'ottimizzazione, integrazione e manutenzione di soluzioni *software* già esistenti per conto di clienti esterni, offrendo supporto tecnologico e consulenziale volto a a favorire l'adozione delle

più recenti innovazioni digitali. Questa evoluzione ha permesso all'azienda di diventare uno dei principali *system integrator* del panorama italiano nel settore dell'*Information and Communication Technology (ICT)*.
La duplice identità coniuga la creatività e la proattività di una *software house* con l'approccio orientato all'efficienza e al cliente tipico di un *system integrator*, e rappresenta uno dei principali punti di forza dell'azienda, che è dunque in grado di offrire soluzioni complete e personalizzate.
SyncLab S.r.l promuove attivamente la collaborazione interna, incoraggiando l'interazione non solo tra i membri della stessa sede, ma anche tra i colleghi di sedi diverse. La politica aziendale è volta a favorire un costante scambio di conoscenze e competenze, creando in tal modo un ambiente dinamico, in cui la crescita professionale di ogni individuo è il risultato del lavoro di squadra.

1.2 Rami aziendali e progetti

SyncLab S.r.l collabora con un ampio numero di clienti appartenenti a molteplici settori industriali e tecnologici. Nel corso degli anni l'azienda ha consolidato una presenza trasversale in diversi ambiti, tra cui *web, mobile, privacy*, sanitario, *blockchain* e trasporti, sviluppando soluzioni capaci di rispondere a esigenze eterogenee.

Figura 1.2: Alcuni prodotti offerti dall'azienda - Fonte: synclab.it

Alcuni dei *software* che l'azienda ha prodotto sono:

- **Sobereye** (ambito *web*): un'applicazione innovativa progettata per monitorare il rischio di deterioramento neuro-cognitivo dovuto a stanchezza, malori, alcol o stupefacenti all'interno dell'ambiente lavorativo.

- **SynClinic** (ambito sanitario): un sistema integrato che supporta la gestione completa dei processi clinici e amministrativi di ospedali, cliniche e case di cura, consentendo di organizzare e monitorare tutte le fasi del percorso di cura del paziente.

- **DPS 4.0** (ambito *web* e *privacy*): una piattaforma *web* che supporta i titolari, responsabili, *data protection officer* (DPO) nelle attività di conformità del Regolamento Generale Protezione Dati (GPDR) nel rispetto del principio di *accountability*.

- **Fast Ride** (ambito trasporti): una soluzione per la gestione di servizi di trasporto pubblico a chiamata, in contesti urbani ed extraurbani, pensata per integrare le linee tradizionali con un sistema flessibile, dinamico ed ecocompatibile.

1.3 *Way of working*

In questa sezione tratterò di alcune tecnologie di cui ho avuto esperienza diretta per lo sviluppo dell'applicazione.

1.3.1 Tecnologie interne

L'azienda utilizza una vasta gamma di tecnologie che includono linguaggi di programmazione e *framework* all'avanguardia. Per la mia esperienza e lo sviluppo di un applicazione mobile ho utilizzato:

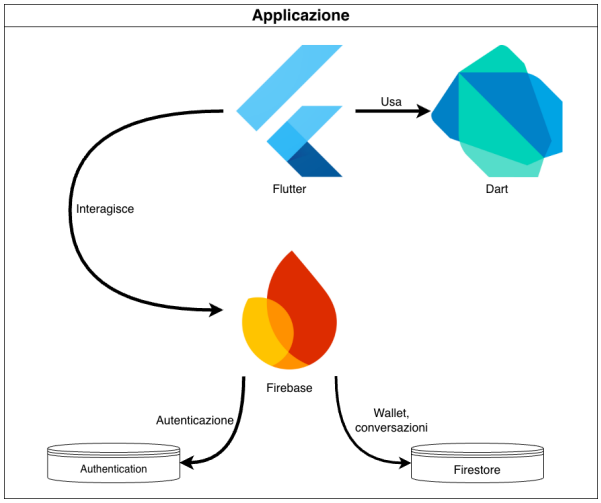


Figura 1.3: Tecnologie impiegate nella realizzazione dell'applicativo.

- **Dart**: linguaggio di programmazione orientato agli oggetti sviluppato da Google, noto per la sua versatilità. Offre numerosi vantaggi come:
  - lo sviluppo di un applicativo *cross-platform* che consente di creare applicazioni native per diversi sistemi da un'unica base di codice.
  - compilazione *Ahead-Of-Time (AOT)* che accelerano lo sviluppo.
  - supporto alla programmazione asincrona che riduce la gestione dei processi in *background*.
- **Flutter**: un *framework* basato su Dart per lo sviluppo di applicazioni multi-piattaforma, che offre vantaggi come:
  - ***Hot-reload***: permette di visualizzare immediatamente le modifiche al codice durante lo sviluppo senza dover aspettare una nuova compilazione.
  - **Compatibilità con *material***: mette a disposizione un ricco arsenale di *widget* per creare interfacce utente moderne e accattivanti, senza dover scrivere molte righe di codice.

- **Firebase**: un *database* sviluppato da Google e ben integrato con Flutter che offre una serie di funzionalità di *backend* pronte all'uso per la propria applicazione.
  - **Authentication**: servizio che offre i servizi di *backend* e librerie pronte all'uso per autenticare gli utenti nell'applicazione in molteplici modi.
  - **Firestore**: *database NoSQL* orientato ai documenti, che permette di archiviare i documenti in raccolte, le quali fungono da contenitori per organizzare i dati e facilitare le interrogazioni.

Per garantire continuità operativa e una efficace gestione dei processi collaborativi, è necessario disporre di strumenti in grado di coordinare e sincronizzare le attività del *team*. Di seguito sono elencati alcuni *software* progettati a questo scopo:

Figura 1.4: Strumenti di supporto.

- **Git**: un sistema di controllo versione distribuito che permette di tracciare in modo efficiente le modifiche ai *file* e di coordinare il lavoro tra più sviluppatori. Questo strumento consente ai membri del gruppo di collaborare in maniera strutturata, gestire eventuali conflitti, apportare modifiche senza rischiare di sovrascrivere il lavoro altrui, creare *branch* dedicati allo sviluppo di nuove funzionalità e recuperare agevolmente versioni precedenti dei *file*.
- **Android Studio**: è un ambiente di sviluppo integrato *(IDE)* gratuito, progettato per lo sviluppo di applicazioni **Android**. Questo *software* mette a disposizione degli sviluppatori diverse funzionalità. Ad esempio un sistema di dispositivi *Android* sia fisici, accopiabili tramite cavo o *Wi-Fi*, che virtuali. Questi ultimi lavorano tramite emulatore integrato che permette di emulare telefoni *smartphone* o *smartwatch*.
- **Xcode**: è un ambiente di sviluppo integrato *(IDE)*, sviluppato e mantenuto da Apple che contiene una *suite* di strumenti utili per lo sviluppo di applicazioni per sistemi proprietari Apple.
- **UMLet**: è uno strumento gratuito e *open source* che permette di creare diagrammi UML[1]. In particolare diagrammi dei casi d'uso, di sequenza e attività e di esportarli in diversi formati.

[1]UML: (Unified Modeling Language) è un linguaggio di modellazione e di specifica

1.3.2 *Smart working* e strumenti di comunicazione

L'azienda adotta un modello di lavoro prevalentemente da remoto: la presenza in sede è richiesta solitamente due volte a settimana, il lunedì e il venerdì, salvo specifiche esigenze. Questi momenti sono dedicati al confronto diretto sul lavoro svolto, ai progressi raggiunti e alle eventuali difficoltà emerse.

Nel mio caso, trattandosi di un progetto di *scouting* tecnologico, gli incontri con il responsabile designato sono stati più frequenti rispetto alla media. Inoltre, ho mantenuto un contatto costante attraverso un server Discord aziendale dedicato, dove aggiornavo regolarmente lo stato di avanzamento, segnalando nuove scoperte, sviluppi dell'applicazione o problemi riscontrati.

Per garantire una gestione efficace delle attività e tracciare correttamente l'evoluzione del progetto, ho utilizzato diversi strumenti digitali:

- **Google Calendar**: un calendario condiviso che permette la creazione e la modifica di eventi, specificandone durata e luogo. È stato utilizzato principalmente per organizzare le presenze in sede, pianificare riunioni e segnalare eventi interni, come ad esempio incontri sindacali.
- **GitHub Projects**: una sezione di GitHub dedicata alla gestione dei *ticket*, preferita in alternativa a **Trello** normalmente utilizzato dall'azienda. Questo strumento consente di creare *roadmap* e schede integrate con le *issue* dei *repository*, facilitando la pianificazione e il monitoraggio delle attività sia a livello individuale che di *team*.

Per la comunicazione interna durante il lavoro da remoto:

- **Discord**: utilizzato dai dipendenti per lo scambio di informazioni tramite *chat* testuali e vocali. La piattaforma funge anche da spazio per condividere aggiornamenti interni, materiali informativi e risorse formative.

Figura 1.5: Strumenti di comunicazione

1.4 Spirito di innovazione aziendale

L'azienda *SyncLab* si caratterizza per una forte propensione all'innovazione, elemento che guida in modo significativo le sue scelte strategiche e operative. Questa attitudine si manifesta attraverso un duplice orientamento:

1. **Miglioramento delle dinamiche interne**: l'azienda investe costantemente nell'ottimizzazione dei processi di gestione dei progetti e nel consolidamento delle relazioni con i clienti. Parallelamente, promuove la crescita professionale del *team* tramite aggiornamenti continui su metodologie e tecnologie emergenti, in linea con la propria identità di *system integrator*.

2. **Proposizione di soluzioni d'avanguardia**: L'impegno innovativo non si limita all'organizzazione interna, ma si estende alla progettazione di prodotti e servizi allineati alle prospettive future del mercato.*SyncLab* sviluppa soluzioni pensate non solo per rispondere alle esigenze attuali dei clienti, ma anche per anticipare *trend* tecnologici, generando valore nel medio-lungo periodo.

In questo contesto, orientato alla ricerca dell'avanguardia si è collocata la mia esperienza di *stage*. La vocazione aperta al cambiamento e la flessibilità dell'azienda sono state condizioni centrali che hanno reso possibile lo sviluppo di un progetto basato su Dart e Flutter, tecnologie ancora poco diffuse all'interno dell'organizzazione.

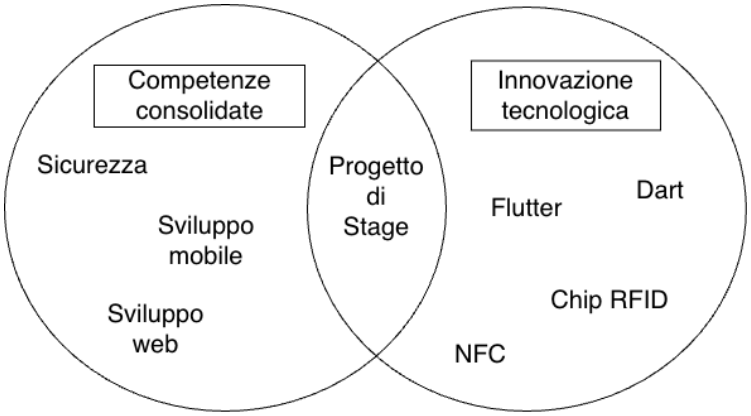


Figura 1.6: Progetto di *stage* come intersezione tra le aree di competenze consolidate di *SyncLab* e l'obiettivo di innovazione tecnologica dell'azienda.

Il progetto si è inserito coerentemente nella strategia aziendale, contribuendo su due fronti principali:

- **Scouting tecnologico**: L'utilizzo di Flutter per la realizzazione di un'applicazione, caratterizzata da requisiti in ambito di sicurezza (crittografia e NFC [2]), si colloca all'interno di una più ampia iniziativa di valutazione dell'adozione di *framework cross-platform* di nuova generazione per i futuri prodotti aziendali.
- **Adattamento di competenze consolidate**: sebbene *SyncLab* possieda già un panorama della sicurezza e dello sviluppo mobile (come descritto nel Paragrafo 1.2), il progetto ha richiesto la reinterpretazione di alcune conoscenze come la gestione dei *chip* RFID[3]/NFC e l'impiego di algoritmi crittografici all'interno di un contesto tecnologico completamente nuovo.

L'approccio fondato su posizioni all'avanguardia di *SyncLab* ha rappresentato quindi un motore concreto che ha reso possibile la realizzazione di una soluzione tecnologicamente avanzata. Il progetto di *stage* è stato un vero e proprio banco di prova per l'introduzione di nuove tecnologie, contribuendo alla strategia aziendale di ampliamento e aggiornamento continuo delle competenze interne.

[2]NFC: Near Field Comunication è una tecnologia di ricetrasmissione che fornisce connettività senza fili bidirezionale a distanza a corto raggio

[3]RFID: Identificazione a radiofrequenza è una tecnologia di riconoscimento e validazione e/o memorizzazione automatica di informazioni a distanza

Capitolo 2

Stage

2.1 Visione aziendale

2.1.1 Offerte aziendali

Per *SyncLab*, i programmi di *stage* rappresentano uno strumento strategico per favorire l'innovazione interna e sostenere l'evoluzione continua delle competenze aziendali. L'azienda considera infatti l'inserimento di stagisti come un'opportunità per acquisire nuove conoscenze e approfondire aree emergenti dell'*information technology*, integrando tali contributi nei progetti esistenti o cogliendo l'opportunità di esplorare tecnologie non ancora adottate. Lo *stage* diventa così un mezzo per cercare soluzioni e idee che altrimenti, a causa dell'impegno continuo richiesto dalle attività operative quotidiane, difficilmente verrebbero affrontate. Al tempo stesso, offre allo stagista un contesto professionale nel quale sviluppare nuove competenze e specializzarsi in ambiti di interesse. Questo modello è reso possibile da una rete consolidata di collaborazioni con università italiane ed estere, che garantisce un continuo scambio di conoscenze e talenti.

Figura 2.1: Collaborazioni dell'azienda - Fonte: synclab.it

I programmi di *stage* si articolano principalmente in tre aree:

- **Integrazione**: gli stagisti contribuiscono al perfezionamento di *software* già in uso, intervenendo su funzionalità specifiche. Tale attività consente di ampliare la comprensione delle potenzialità delle soluzioni aziendali e di migliorarne progressivamente le *performance*.
- **Analisi** e **ottimizzazione**: questa area prevede una valutazione approfondita delle soluzioni *software* esistenti, con l'obiettivo di individuare inefficienze, proporre miglioramenti e ottimizzare i processi. L'approccio critico e analitico adottato permette di valorizzare al massimo il patrimonio tecnologico dell'azienda.
- **Innovazione**: in questo ambito gli stagisti svolgono analisi teoriche e sperimentazioni su tecnologie emergenti, con *focus* specifici allineati agli obiettivi strategici aziendali. Questa attività valuta il potenziale innovativo delle nuove tendenze e identifica possibili applicazioni operative.

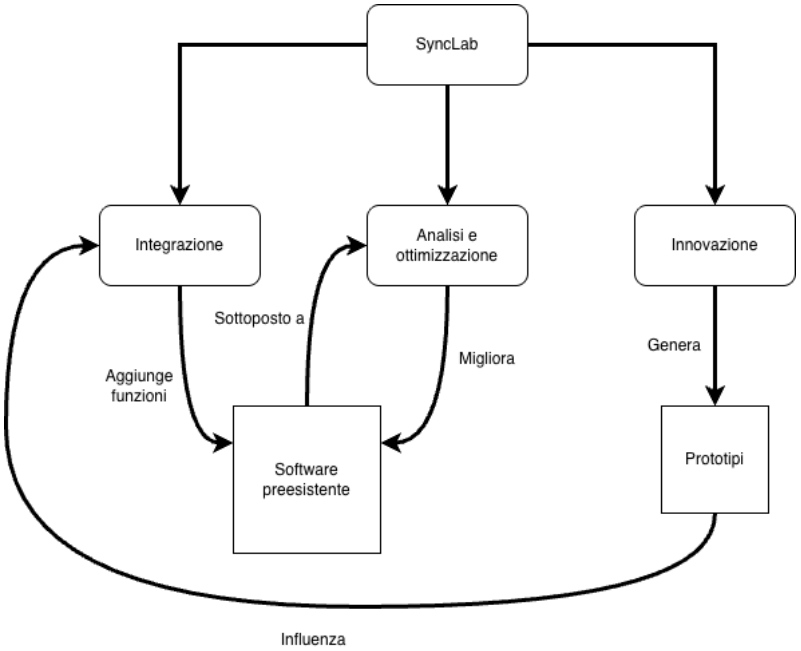


Figura 2.2: Contributo dei programmi di *stage* in SyncLab.

2.1.2 *Stage* in azienda

Gli *stage* presso *SyncLab* rappresentano un investimento strategico bidirezionale, che riflette la duplice identità aziendale, orientata alla crescita interna e all'avanguardia tecnologica. I programmi sono concepiti come un ciclo continuo di apprendimento, dove l'azienda trae valore dalle competenze degli stagisti e questi ultimi traggono esperienza dall'organizzazione. Questo approccio si manifesta attraverso due obiettivi fondamentali:

1. **Formazione e inserimento di risorse qualificate:** Il programma di *stage* funge da canale primario per identificare e valutare talenti. L'azienda offre un'immersione pratica agli studenti, consentendo loro di applicare le conoscenze accademiche a problemi reali. La società valuta in modo approfondito il potenziale dei candidati, e gli stagisti che mostrano valore e impegno ricevono offerte di assunzione come membri effettivi del *team*. Questo garantisce all'azienda una *pipeline* costante di nuove risorse già integrate nel proprio ecosistema.

2. **Acquisizione di nuove competenze:** Il programma di *stage* funge da banco di prova per l'innovazione continua, riducendo così i rischi associati all'adozione di nuove tecnologie. Attraverso questo meccanismo l'azienda rimane all'avanguardia e adatta le competenze consolidate a contesti tecnologici inediti.

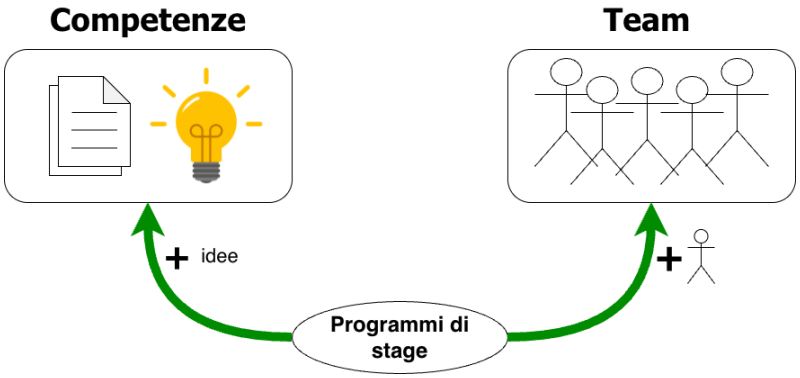


Figura 2.3: Investimento strategico degli *stage* per l'azienda

2.1.3 Ruolo del *tutor* aziendale

Durante il percorso ogni stagista è affidato a un *tutor*. Il *tutor* è una figura, professionale che opera nello stesso ambito in cui lo stagista svolge le proprie attività costituendo di fatto un punto di riferimento. Il suo ruolo ha tre funzioni principali:

- **Guida**: il *tutor* propone linee guida su come organizzare il lavoro e fornisce allo stagista materiali utili che possono aiutarlo a comprendere le tecnologie da utilizzare nel progetto.

- **Supporto**: il *tutor* offre una serie di suggerimenti o soluzioni possibili ai problemi incontrati, questo per scongiurare l'eventualità che il lavoro venga bloccato per lungo tempo e per garantire allo stagista un confronto con una figura preparata stimolando una soluzione concreta.

- **Supervisore**: il *tutor* offre *feedback* allo stagista nella valutazione del lavoro svolto per verificarne la qualità e supportarlo al miglioramento. Questa attività è molto utile in quanto migliorativa nei confronti dello stagista per formarlo verso un percorso professionale.

Nel mio caso i dialoghi con il *tutor* assegnato sono stati molto importanti in quanto senza il suo confronto, e i suoi *feedback*, avrei incontrato maggiori ostacoli, e conseguenti ritardi, nello sviluppo del progetto. Anche i suggerimenti sull'architettura di sistema, l'organizzazione delle cartelle e un congruo supporto all'analisi del problema sono state fondamentali per la progettazione dell'applicazione e per la comprensione dei vari componenti da utilizzare.

2.2 Motivazione dello *stage*

L'obiettivo dello *stage* è stato la valutazione di fattibilità e il potenziale impiego di nuove tecnologie nello sviluppo di prodotti futuri. Lo sviluppo nativo per **Android** e **iOS** richiede competenze eterogenee e la gestione di *codebase* separate, che possono comportare costi elevati di manutenzione e di sviluppo. Altra spinta motrice dello *stage* è stata la sperimentazione di una cifratura *end-to-end* che consente di rendere dei messaggi inseriti nell'applicazione illeggibili ad un intercettatore, implementando quindi un'applicazione mobile capace di generare e gestire dei *wallet* crittografici, comprensivi di coppie private e pubbliche. Per questo motivo, lo stage ha previsto lo studio e la valutazione del *framework Flutter* e linguaggio Dart, che consentono di sviluppare un applicazione mobile per diversi dispositivi tramite un'unica *codebase*.

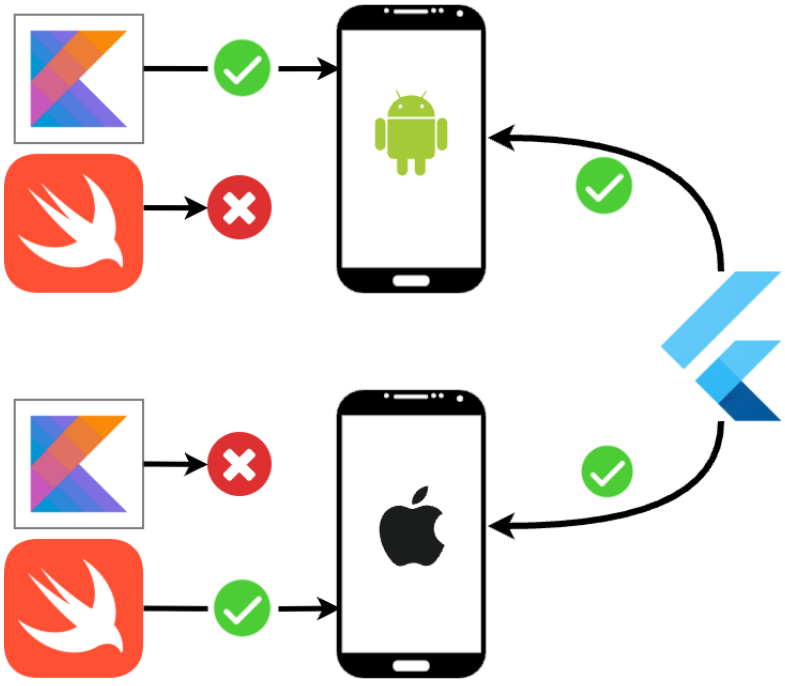


Figura 2.4: Confronto tra lo sviluppo nativo e l'approccio *cross-platform* con *Flutter*

2.3 Progetto proposto

Nel contesto attuale, i dispositivi mobili rappresentano uno strumento di comunicazione ampiamente diffuso e accessibile. Tuttavia, l'utilizzo di canali comuni espone gli utenti a potenziali rischi di intercettazione e compromissione della *privacy*. In questo scenario, la protezione delle comunicazioni assume un ruolo centrale e richiede soluzioni tecniche in grado di garantire riservatezza e integrità dei messaggi scambiati.
Per tale motivo, l'azienda ha proposto la progettazione e l'implementazione di un'applicazione in grado si generare e gestire *wallet* crittografici, comprensivi di coppie di chiavi private e pubbliche.
La realizzazione del progetto ha comportato l'identificazione e la risoluzione di alcune sfide progettuali e tecnologiche qui di seguito riportate.

Sviluppo multipiattaforma

L'applicazione deve essere sviluppata interamente tramite il *framework* Flutter, e deve poter girare su dispositivi diversi con architetture diverse, sia *Android* che *iOS* a partire da un unica base di codice in modo tale che sia mantenibile senza l'impiego di numerose risorse. Questo richiede uno studio approfondito del *framework* e del linguaggio di programmazione Dart, che potrebbe rappresentare un onere importante.

Algoritmi di crittografia

Sviluppare un'applicazione che permetta di verificare l'uso corretto degli algoritmi di crittografia, in modo tale che vengano impiegati correttamente in modo tale che i messaggi inseriti nell'applicazione vengano letti solo da interessati e non da esterni. Questo richiede uno stratagemma che consenta allo *stakeholders* di vedere come i messaggi vengano criptati e decriptati.

Gestione delle chiavi crittografiche

Il recupero della chiave privata usata per decifrare un messaggio rappresenta una grossa sfida in quanto l'applicazione deve poter generare più chiavi crittografiche e, soprattutto l'applicazione deve supportare il passaggio dati in caso di smarrimento, da un dispositivo a un altro. Quindi ci deve essere una base di dati che consenta il recupero di informazioni sul dispositivo, ma alcune di queste informazioni devono essere trasmesse in modo differente.

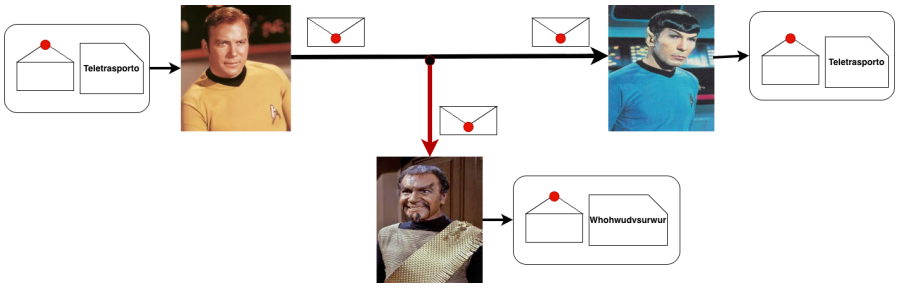


Figura 2.5: Esempio di comunicazione criptata

Integrazione con il lettore NFC

L'applicazione deve supportare l'utilizzo del lettore NFC presente sui dispositivi mobili in modo da associare i documenti con chip RFID validi alle chiavi salvate sul dispositivo. Quindi bisogna capire quali dati si possono estrapolare dai documenti tramite l'utilizzo di librerie Flutter/Dart e come poterle integrare nell'applicazione.

2.4 Obiettivi e vincoli

2.4.1 Obiettivi

Il progetto proposto è stato guidato da una serie di obiettivi volti a garantirne il successo e la produzione di risultati concreti. In particolare gli obiettivi specifici del progetto sono: lo **studio di fattibilità** e un successivo **sviluppo di una applicazione** da eseguire almeno sui dispositivi *Android*.

Studio di fattibilità: effettuare uno studio dettagliato e un'analisi approfondita sulla fattibilità tecnica dell'utilizzo di Flutter e delle sue librerie per la realizzazione di una applicazione funzionante. In particolare uno studio concentrato su:

- l'integrazione con il *server Firebase*;
- il funzionamento del lettore NFC;
- la generazione di una coppia di chiavi (pubblica e privata) crittografiche;

Questo per permettere di valutare le potenzialità e i limiti degli strumenti, in modo da individuare i migliori da utilizzare per lo sviluppo dell'applicazione.

Sviluppo di una applicazione: per verificare che i componenti applicativi funzionino bene tra di loro, e che permetta di associare i documenti scansionati con le coppie di chiavi generate direttamente dall'applicazione.

2.4.2 Vincoli

Nella realizzazione di un progetto è fondamentale considerare i vincoli ovvero delle restrizioni e limitazioni che devono essere considerate per garantire il successo del progetto. Questi ultimi infatti incidono sulle scelte architetturali da compiere durante la progettazione.

- **Vincoli tecnologici**:
  - **Utilizzo obbligatorio del *framework Flutter*:**
    L'intera applicazione deve essere sviluppata interamente tramite *Flutter*, con il conseguente utilizzo del linguaggio *Dart* per tutta l'implementazione della logica applicativa. Questo rappresenta un vincolo rilevante poiché rispetto ad altri linguaggi come *Java* la comunità di sviluppatori è più contenuta. Di conseguenza, la disponibilità di pacchetti e librerie potrebbe risultare limitata o acerba, specialmente per funzionalità specifiche.
  - **Utilizzo del piano gratuito di *Firebase*:**
    L'uso di *Firebase* presenta alcune limitazioni derivanti dal piano gratuito, che esclude funzionalità specifiche presenti solo nelle versioni a pagamento.

Queste restrizioni incidono sulla progettazione di alcune componenti e sulle scelte implementate.

- **Vincoli architettura di sicurezza**:
  - **Protezione della chiave privata:**
    L'architettura deve impedire la trasmissione *online* della chiave privata, garantendo però un meccanismo sicuro che consenta all'utente di recuperare il proprio *wallet* in caso di smarrimento o migrazione su un nuovo *smartphone*.
  - **Associazione delle chiavi ai documenti:**
    Ogni coppia di chiavi deve essere associata a un documento, e non a *tag NFC* generici.
  - **Unicità del *wallet* per documento e *account*:**
    Non deve essere consentita la creazione di più *wallet* associati allo stesso documento per un medesimo *account*. Preservando l'unicità della coppia di chiavi per ogni documento, si evita di generare più *wallet* per lo stesso documento, e di conseguenza compromettere la coerenza del sistema.

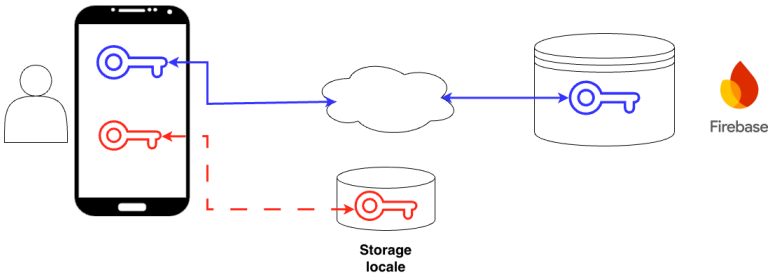


Figura 2.6: Schema che rappresenta l'architettura di sistema, la chiave blu corrisponde alla chiave pubblica, quella rossa invece corrisponde alla chiave privata

2.5 Scelta dello *stage*

2.5.1 Motivazioni della scelta

Ho conosciuto *Synclab* nel mese di aprile attraverso una comunicazione telefonica, ma prima di confermare la proposta aziendale ho valutato altre possibilità di *stage*. In seguito, durante il mese di giugno, ho deciso di ricontattare l'azienda, che mi ha presentato due proposte di progetto particolarmente interessanti: la prima riguardava lo sviluppo di un'applicazione con *backend* basato su Odoo ERP, mentre la seconda consisteva nel realizzare un'applicazione mobile sviluppata tramite il *framework Flutter*, dedicata alla generazione di *wallet* contenenti chiavi pubbliche e private.

Parallelamente, durante il periodo estivo avevo ricevuto ulteriori offerte di stage da altre aziende e stavo valutando alcune alternative per avviare un percorso lavorativo tempestivamente, nella consapevolezza che l'inizio effettivo dello stage sarebbe comunque avvenuto a settembre dopo la pausa estiva.

Alla fine ho scelto di intraprendere il mio percorso in *Synclab* per i seguenti motivi:

- **Possibilità di lavorare da remoto**, condizione per me fondamentale poiché, dovendomi spostare frequentemente e non disponendo di un mezzo di trasporto personale, avrei avuto difficoltà a raggiungere quotidianamente la sede aziendale.

- **Elevato livello di autonomia offerto dal progetto**, trattandosi di uno *scouting* tecnologico finalizzato alla valutazione dei punti di forza e delle criticità del framework Flutter. Ciò mi avrebbe permesso di affrontare un lavoro altamente esplorativo, con ampi margini decisionali.

2.5.2 Obiettivi personali prefissati

Nell'attuale contesto tecnologico, l'uso dei dispositivi mobili è divenuto parte integrante della vita quotidiana: la maggior parte delle persone possiede almeno uno *smartphone*, attraverso il quale comunica e accede a numerosi servizi. Tuttavia, molte applicazioni di messaggistica sono oggi controllate dalle *big tech* e sia l'hardware sia il *software* risultano sempre più lontane dalle diretta gestione da parte dell'utente.

L'introduzione della proposta di legge europea relativa al controllo delle *chat* e alla possibilità di analizzare i dati delle conversazioni ha ulteriormente alimentato in me domande sulla reale tutela dei dati personali e sensibili. Le chiavi private degli utenti, che idealmente dovrebbero rimanere sotto il loro esclusivo controllo, sono spesso memorizzate su *server* esterni per ragioni di praticità.

Da qui è nata la domanda che ha guidato parte della mia ricerca: esiste un modo per conservare le chiavi private in spazi realmente sicuri all'interno del dispositivo, garantendo una protezione efficace e impedendo l'accesso da parte di terzi?

Nello sviluppo dell'applicazione mi sono quindi posto i seguenti obiettivi:

- Progettare un'applicazione capace di gestire i dati privati degli utenti in modo distinto rispetto a quelli pubblici, incrementandone la sicurezza complessiva;
- Apprendere lo sviluppo di applicazioni per dispositivi mobili attraverso l'utilizzo del *framework* Flutter;
- Acquisire i principi di crittografia necessari a consentire una comunicazione sicura tra due dispositivi mobili.

Per quanto qui permesso, lo stage presso *Synclab* costituiva per me un'importante occasione per approfondire i concetti di sicurezza informatica affrontati durante il percorso universitario e applicarli in un contesto reale, concreto e di grande rilevanza come il settore delle applicazioni mobili.

Capitolo 3

Realizzazione dell'applicazione

Il presente capitolo introduce la realizzazione dell'applicazione *Key Wallet App* descrivendo una fase di pianificazione e di analisi per poi proseguire verso i problemi riscontrati durante lo sviluppo, la loro mitigazione e le scelte che sono state intraprese e si conclude con un resoconto su ciò che è stato realizzato.

3.1 Pianificazione del lavoro

Come introdotto nel capitolo 2 l'obiettivo del mio progetto era una valutazione di fattibilità e di potenziale impiego nuove tecnologie per l'azienda in nuovi prodotti. Pertanto mi serviva trovare un compromesso che mi offrisse nella fase di studio rigidità e nella fase di sviluppo e implementazione flessibilità. Quindi per la realizzazione del progetto ho scelto un approccio ibrido tra un modello sequenziale a cascata durante lo studio e la pianificazione, mentre nella realizzazione dell'applicazione ho scelto un modello agile.

Attività	**Settimane**								**Ore**
	1	2	3	4	5	6	7	8	
Ripasso costrutti di Java	X								5
Studio di Dart	X	X							30
Studio di Flutter		X	X						40
Studio algoritmi di criptazione		X		X		X			30
Analisi del problema	X								10
Progettazione della piattaforma				X					25
Sviluppo maschera di *login*			X						5
Sviluppo di un prototipo che genera chiavi			X						30
Sviluppo applicazione finale					X	X	X		100
Stesura finale della specifica tecnica							X	X	20
Live demo e presentazione finale								X	5
totale ore	300								

Tabella 3.1: Pianificazione del lavoro durante le 8 settimane

Durante le prime settimane di stage, è stata applicata una metodologia a cascata,

essenziale per stabilire le fondamenta del progetto. Durante la prima settimana ho lavorato a stretto contatto con il mio responsabile per definire gli obiettivi e la struttura del progetto. Dopodiché mi sono concentrato sullo studio del linguaggio Dart e del framework Flutter per poi spostarmi sulla libreria di pointycastle e gli algoritmi di crittografia durante la prime due settimane di stage. Dalla terza settimana fino alla 4 mi sono concentrato sullo sviluppo di piccoli prototipi (es. login finto, generazione di chiavi alla pressione di un tasto e lettura di tag nfc). Questi prototipi sono stati molto utili per effettuare piccoli test sulle funzioni nelle settimane successive e a trovare falle sul progetto immediatamente una tra tutte la lettura di tag nfc. Difatti questi ultimi non possiedono un identificativo fisso sulle carte d'identità elettroniche ma cambia ogni secondo.

Una volta completata la prototipazione, il progetto è transitato verso una metodologia agile, in particolare durante la realizzazione dell'applicazione finale. Lo sviluppo agile offre una gestione flessibile dei problemi tecnici, e la possibilità di concentrarsi in modo progressivo sull'implementazione dei singoli componenti del sistema, permettendomi di integrare funzionalità ad ogni iterazione. Questo significava che per le ultime fasi del progetto pianificavo, implementavo e testavo le funzionalità implementate. Permettendomi di reagire immediatamente ai problemi ritrovati, uno tra questi, l'uso di un algoritmo RSA senza padding.

L'adozione quindi dell'approccio ibrido ha garantito la copertura formativa e analitica iniziale, consentendo al contempo la gestione flessibile e sicura delle complessità tecniche emerse durante la loro implementazione.

3.2 Analisi dei requisiti

La fase di analisi svolta durante le prime settimane del percorso di *stage*, sono state cruciali per definire l'architettura applicativa e le tecnologie da utilizzare. Gli incontri con il *tutor* hanno rappresentato un fulcro importante per lo sviluppo e la definizione del progetto. Le diverse iterazioni hanno permesso di delineare le linee guida e velocizzare le scelte tecnologiche, ad esempio l'uso di librerie specifiche come `pointycastle` per la crittografia o `flutter_secure_storage` per la gestione locale della chiave privata. A seguito di queste indicazione la fase di analisi è stata focalizzata sullo studio di fattibilità delle tecnologie e sullo sviluppo di piccoli prototipi.

3.2.1 Requisiti funzionali

I requisiti funzionali definiscono le funzionalità e i servizi specifici che l'applicazione deve fornire per conseguire gli obiettivi del progetto. Ogni requisito funzionale è codificato in questo modo: **RF[Numero]**.

Codice	Descrizione
RF1	Generazione di una coppia di chiavi pubbliche e private
RF2	Generazione ed eliminazione di *wallet* contenenti le coppie di chiavi
RF3	Implementazione della parte di accesso e registrazione tramite appoggio di *database*
RF4	Implementazione di un meccanismo di storage sicuro che permetta agli utenti di salvare le chiavi private sul *Keystore/Keychain* del dispositivo
RF5	Riuscire a criptare un messaggio inserito nell'applicazione tramite la chiave pubblica di un altro utente
RF6	Riuscire a decifrare un messaggio tramite la propria chiave privata personale
RF7	Implementazione di un meccanismo di recupero *wallet*
RF8	Implementazione di un meccanismo di lettura di *tag NFC* per associare un documento a un *wallet*

Tabella 3.2: Requisiti funzionali

3.2.2 Requisiti di qualità

I requisiti di qualità stabiliscono degli *standard* secondo cui tali funzionalità del prodotto devono essere realizzate ed eseguite, assicurando un elevato livello di affidabilità e coerenza del sistema. Quest'ultimi sono stati definiti all'inizio del percorso di *stage* e sono stati misurati alla fine come criteri per la valutazione del progetto. Ogni requisito di qualità è codificato in questo modo: **RQ[Numero]**.

Codice	Descrizione
RQ1	Assicurare un'accuratezza nella generazione e crittografia delle chiavi superiore al 95%
RQ2	Mantenere il tempo di risposta per la generazione del *wallet* al di sotto dei 2 secondi
RQ3	Raggiungere una copertura di test automatici del 70% per le principali funzionalità

Tabella 3.3: Requisiti di qualità

Ognuno di questi requisiti è stato verificato attraverso lo sviluppo di test automatici.

3.2.3 Requisiti di vincolo

I requisiti di vincolo definiscono le limitazioni operative e tecnologiche cui il progetto ha dovuto attenersi, influenzando direttamente le scelte architetturali e l'ambito di sviluppo. Abbiamo già discusso dei vincoli del progetto nella sezione 2.4.2 Pertanto mi

limito ad elencarli in questa sezione. Ogni requisito di vincolo è codificato in questo modo: **RV[Numero]**.

Codice	Descrizione
RV1	L'applicazione finale deve essere sviluppata tramite il *framework* Flutter e il linguaggio Dart
RV2	L'applicazione finale deve appoggiarsi a Firebase come appoggio per una base di dati
RV3	Le chiavi private generate non devono essere trasmesse *online* ma devono essere cutodite in spazi sicuri del dispositivo come il *Keychain* e *Keystore*
RV4	Ogni documento deve essere associato a unico wallet

Tabella 3.4: Requisiti di vincolo

3.3 Implementazione

3.3.1 Il linguaggio Dart

Lo studio di questo linguaggio di programmazione ha rappresentato la prima sfida del progetto, in quanto sarebbe stata la base da cui partire per costruire l'intera applicazione. Personalmente non avevo mai visto ne provato questo linguaggio tuttavia la sua sintassi simile a Java e JavaScript hanno fortemente semplificato il suo apprendimento. Riporto di seguito le principali differenze e peculiarità che secondo me sono importanti da sottolineare in quanto caratteristiche fondamentali del linguaggio:

Dart è un linguaggio di programmazione orientato agli oggetti sviluppato da Google, è stato concepito come alternativa a JavaScript per lo sviluppo *web*. Sebbene questo obiettivo non abbia trovato ampia adozione, Dart si è affermato come linguaggio versatile ed efficace nello sviluppo di applicazioni multipiattaforma. Una delle sue caratteristiche è quella di essere compilato verso diversi *target*: ARM64 per i dispositivi mobili, JavaScript per il *web*, e architetture x86_64 per i sistemi operativi *desktop* quali Windows, macOS e Linux.

Type-safe

Dart è un linguaggio *type-safe*: ovvero ogni variabile mantiene una corrispondenza certa con il proprio tipo statico. Però allo stesso tempo offre un sistema di tipizzazione flessibile grazie alla *keyword* `dynamic`, che permette la verifica dei tipi a *run-time* quando necessario.

```
void main() {
  int temp = 10;
  temp = 20; // OK
  //temp = "ciao"; // ILLEGALE: string != int

  print(temp); // stampa 20

  dynamic temp1 = 30;
  print(temp1); // stampa 30
  temp1 = "ciao";
  print(temp1); // stampa ciao
}
```

Listing 1: Esempio di *type-safe*

null-safe

Il linguaggio integra un sistema di *null safety*: il valore `null` può essere associato a una variabile solo se esplicitamente dichiarato.

```
void main() {
  String nome = "Mario";
  print(nome); // stampa: Mario
  //nome = null; //ILLEGALE: null non può essere assegnato a String

  String? cognome = "Rossi";
  cognome = null;
  print(cognome); //Nessun problema stampa null
}
```

Listing 2: Esempio di *null safety*

Interfacce e classi astratte

In Dart sia le interfacce esplicite che le classi astratte sono dichiarate usando la *keyword* `abstract class`. La distinzione dei due concetti risiede nel modo in cui vengono utilizzate le classi derivate:

Un'interfaccia definisce un contratto: qualsiasi classe che la derivi deve utilizzare la *keyword* `implements`, impegnandosi pertanto a fornire una implementazione concreta di tutti i metodi dichiarati. Al contempo, una classe astratta può essere estesa tramite la *keyword* `extends` e può contenere sia i metodi astratti sia metodi già implementati.

```dart
abstract class ContrattoDiConnessione {
  void connetti();
  void disconetti();
}

class B{
    void bMethod() {}
}

abstract class BaseScreen {
  void logicaComune() {}
}

class GestoreServizio implements ContrattoDiConnessione, B {
  @override
  void connetti() { /* implementazione */ }
  @override
  void disconetti() { /* implementazione */ }
  @override
  void bMethod() {/* implementazione */}
}

class HomePage extends BaseScreen {
  void init() {
    super.logicaComune();
  }
}
```

Listing 3: Esempio di interfaccia e classe astratta

In Dart, ogni classe definita genera implicitamente un'interfaccia che include tutti i suoi metodi e proprietà pubbliche.

3.3.2 Il *framework* Flutter

La seconda sfida è stata lo studio del *framework* Flutter. Flutter non utilizza componenti di interfaccia nativi del sistema operativo ospite, ma al contrario sfrutta un suo motore grafico che disegna direttamente sulla schermata del dispositivo, garantendo un controllo totale sul *rendering* dell'interfaccia e garantendo prestazioni comparabili a quelle delle applicazioni native. Appoggiandosi sul linguaggio Dart, Flutter supporta funzioni specifiche che permettono ad un programmatore di velocizzare lo sviluppo. Una di queste funzioni è l'*hot-reload* che permette di visualizzare immediatamente le modifiche al codice senza dover aspettare ogni volta una nuova compilazione.

Per lavorare bene in Flutter è fondamentale comprendere che ogni elemento dell'interfaccia è rappresentato da un *widget*. Il *framework* adotta un approccio basato sulla composizione, tutto dagli elementi più semplici ai *layout* più complessi è costruito tramite *widget* annidati tra loro. Per fare un esempio banale una schermata tipica in Flutter è definita attraverso il *widget* `Scaffold` che fornisce la struttura di base dell'interfaccia. Lo `Scaffold` accetta parametri come `appBar`, `body` e `floatingActionButton`. Questi parametri verranno a loro volta definiti tramite altri *widget*. Bisogna tuttavia fare una distinzione importante tra i *widget* senza stato *(stateless)* e i *widget* con stato *(stateful)*.

Stateless widget

Uno *stateless widget* è un *widget* immutabile, ovvero un componente la cui configurazione non cambia durante l'intero ciclo di vita senza variare nel tempo.

```
class Titolo extends StatelessWidget {
  final String testo;

  const Titolo({super.key, required this.testo});

  @override
  Widget build(BuildContext context) {
    return Text(testo);
  }
}
```

Listing 4: *Stateless Widget*

Stateful widget

Uno *stateful widget* è un *widget* mutabile, in grado di mantenere e aggiornare un suo stato interno. Questo stato è gestito da una classe chiamata `State` che permette di modificare lo stato e aggiornare l'interfaccia. Quando lo stato cambia, Flutter chiama il metodo `setState()`, che a sua volta forza la ricostruzione del *widget*.

```dart
class Contatore extends StatefulWidget {
  const Contatore({super.key});

  @override
  State<Contatore> createState() => _ContatoreState();
}

class _ContatoreState extends State<Contatore> {
  int valore = 0;

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Valore: $valore'),
        ElevatedButton(
          onPressed: () {
            setState(() {
              valore++;
            });
          },
          child: Text('Incrementa'),
        ),
      ],
    );
  }
}
```

Listing 5: *Stateful Widget*

3.3.3 Architettura a tre livelli

L'applicazione sviluppata adotta il modello di architettura a tre livelli, una struttura che segmenta i componenti *software* in strati logicamente distinti, ciascuno con responsabilità differenti e ben definite.

Il primo livello di presentazione rappresenta un punto di interazione tra l'utente e il sistema. È composto da tutto ciò che definisce l'interfaccia grafica. Quindi le schermate e *widget* che le compongono. La sua funzione è quella di presentare i dati in chiaro e raccogliere l'*input* dell'utente.

Il secondo livello rappresenta la logica di *business*. Coordina le operazioni tra il livello di presentazione e il livello dei dati. Nel mio caso specifico è responsabile di operazioni fondamentali come la generazione delle chiavi crittografiche e l'esecuzione dei processi di cifratura e decifratura.

Il terzo livello è responsabile della persistenza e recupero dei dati. Definisce come queste vengono archiviate e rese disponibili agli altri livelli.

3.3.4 Il *design pattern provider*

Per gestire lo stato dei *widget* nell'applicazione realizzata è stato adottato il *design pattern provider* che si allinea perfettamente alle esigenze di una architettura a tre livelli in quanto:

1. Permette la separazione delle responsabilità (SoC[1]) eliminando la necessità di propagare le informazioni tramite la gerarchia dei *widget*.

2. Aumenta le *performance* consentendoci di costruire i *widget* strettamente necessari quando lo stato cambia, evitando i *rebuild* di intere porzioni dell'interfaccia.

3. Fornisce un meccanismo per accedere e aggiornare lo stato rendendo il codice più facile da mantenere, facilitando quindi l'introduzione di funzionalità o la modifica di quelle esistenti.

3.3.5 la tecnologia NFC

La tecnologia NFC permette a due dispositivi di connettersi, scambiarsi informazioni o attivare funzioni nel momento in cui si trovano a una distanza ravvicinata. Depositato da Charles Walton nel 1983 NFC nasce grazie all'intenzione di Sony e Philips di cooperare nella creazione di uno *standard* per la comunicazione a radiofrequenza nel 2002, e L'ISO[2] lo approva solo nel 2003 come *standard*. L'idea è quella di far integrare informazioni all'interno dei dispositivi mobili permettendo una facile iterazione tra documenti e telefoni.

Lettura dati nei *chip* RFID

Uno dei requisiti era l'implementazione di un meccanismo di riconoscimento dei documenti contenenti il *chip* RFID per associare le coppie di chiavi al ID del documento. In buona sostanza l'utente alla creazione del *wallet*, avrebbe dovuto scansionare il documento.

Per realizzare questo sono state analizzate due librerie Flutter che permettono la scansione e la scrittura attivando il sensore NFC presente sul telefono. La prima `nfc_manager` alla versione 4.1.1 è stata scartata in quanto da poco tempo è passata a una *major update* modificando tutti i nomi dei metodi e non viene aggiornata spesso, la seconda `flutter_nfc_kit` invece è stata la scelta definitiva in quanto include un supporto maggiore da parte degli sviluppatori e sta crescendo rapidamente.
Grazie alla documentazione fornita dagli sviluppatori eseguire un prototipo che permetta la scansione dei documenti è stato facile.

[1] Separation of Concerns
[2] International Standard Organization

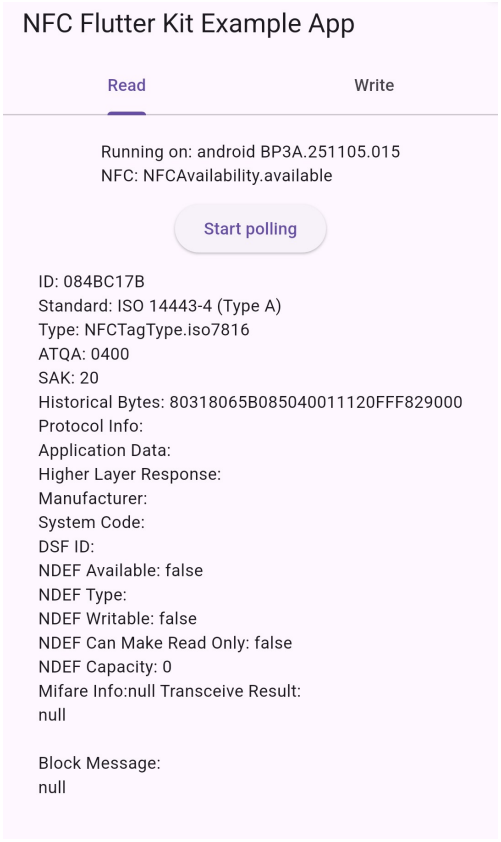


Figura 3.1: Applicazione prototipale per dimostrare la fattibilità della scansione NFC

La libreria ci permette di recuperare i campi di ID, lo *standard* ISO del *chip* RFID (ad esempio la carta d'identità elettronica utilizza uno *standard* ISO 14443), e gli *historical bytes*. Questi ultimi rappresentano una piccola sezione dati fornita dal *chip* che identificano in modo informativo il *chip*, il produttore o la versione.

Testando l'applicazione con una varietà di documenti e *tag* NFC, ho riscontrato differenze significative nella struttura dei dati restituiti. Ad esempio, la tessera del *tram* di Busitalia Veneto e quella di una palestra privata rispetto a documenti come la tessera sanitaria o il *badge* universitario non forniscono né includono i campi relativi agli *hystorical bytes*, evidenziando quindi come il comportamento e il livello di informazione dei *tag* NFC vari sensibilmente in base allo *standard* impiegato e al livello di sicurezza previsto dall'emittente.
Nel caso della CIE ho rilevato un comportamento peculiare: il campo identificativo (`id`) letto tramite NFC non rappresenta un identificatore univoco permanente ma bensì un valore temporaneo che cambia ad ogni scansione. Approfondendo la documentazione tecnica del *chip* - disponibile sul sito del ministero dell'Interno - emerge che la CIE adotta modelli di sicurezza avanzati come il protocollo BAC[3] o il protocollo PACE[4] che limitano severamente la quantità di informazioni accessibili senza autenticazione. In particolare il numero unico nazionale (`nun`) che si trova in alto a destra sul fronte

[3] aggiungi
[4] aggiungi

del documento.

Il *chip* della Carta d'Identità Digitale (CIE) è infatti progettato per memorizzare dati personali sensibili, proteggendole tramite una protocolli avanzati e non direttamente leggibili con una interrogazione NFC.
L'identificatore temporaneo e non persistente rientra nelle misure previste per ridurre i rischi legati al tracciamento e alla profilazione utente.

Questo rappresenta per il progetto un bel problema in quanto senza un identificativo del documento recuperabile tramite scansione è impossibile associare una singola chiave ad esso.

Mitigazione del problema

Per interagire con la Carta d'Identità Elettronica (CIE) è necessario utilizzare gli strumenti *software* ufficiali messi a disposizione dall'Istituto Poligrafico e Zecca dello Stato (IPZS). Senza la loro implementazione, un'applicazione di terze parti può accedere solo a informazioni superficiali come quelle indicate precedentemente, senza accedere ai dati personali o ai file protetti presenti sul documento.

Una prima soluzione valutata è stata quella di consultare la piattaforma dedicata agli sviluppatori che realizzano i software per la Pubblica Amministrazione (PA). Tra i progetti disponibili è presente anche il *kit* di sviluppo ufficiale per Android, reperibile al *repository* `cieid-android-sdk` . Questo consente di integrare nei sistemi Android l'autenticazione basata su CIE, ma è stato sviluppato in Kotlin ed è compatibile esclusivamente per le applicazioni native Android, entrando in conflitto con uno dei punti fondamentali dello sviluppo di applicazioni in Flutter ovvero lo sviluppo multipiattaforma.

A seguito di riunioni interne con il mio responsabile, si è scelto di non procedere su questa strada, sia per ragioni di tempo ma anche per evitare la gestione di dati sensibili, che sarebbero soggetti alle normative del GDPR[5] per l'azienda. Pertanto si è deciso di utilizzare come identificativo il valore degli *historical bytes*, insieme allo standard del *chip* NFC rilevato durante la scansione.

I documenti quali la CIE, la tessera sanitaria o le carte di credito adottano lo standard ISO/IEC 14443-4, che prevedono la presenza di *historical bytes*. Sebbene questi non rappresentino un identificatore univoco in senso assoluto, risulta estremamente improbabile che due *tag* diversi registrino lo stesso valore, soprattutto se appartenenti a categorie differenti. Per ridurre ancora tale probabilità, l'applicazione esegue la verifica di unicità solo tra i documenti registrati dall'utente e non su tutti quelli presenti nel *database* dell'applicazione, garantendo così un adeguato livello di affidabilità.

[5] inserire

```
Future<NFCTag?> fetchNfcData() async {
    try {
      if (defaultTargetPlatform == TargetPlatform.iOS) {
        await _nfcKitWrapper
          .setIosAlertMessage("Avvicina il dispositivo...");
      }
      NFCTag tag = await _nfcKitWrapper.poll();
      return tag;
    } catch (e) {
      await _nfcKitWrapper.finish();
      return null;
    }
}

Future<void> _scanNfcTag() async {
    if (_isScanning) return;
    setState(() => _isScanning = true);
    try {
      final nfcService = context.read<INfcService>();
      dynamic tagData =
        await nfcService.fetchNfcData();
      if (tagData != null && mounted) {
        setState(() {
          hBytes = tagData.historicalBytes
            ?.toString() ?? 'N/D';
          standard = tagData.standard
            ?.toString() ?? 'N/D';
          if (hBytes.isNotEmpty &&
              hBytes != 'N/D' &&
              standard.isNotEmpty) {
            ScaffoldMessenger.of(context)
              .showSnackBar(const SnackBar(
                content: Text(
                  "Documento scansionato!"),
                backgroundColor: Colors.green));
          }
        });
      }
    } catch (e) {
      if (mounted) {
        ScaffoldMessenger.of(context)
          .showSnackBar(SnackBar(
            content: Text("Errore: $e"),
            backgroundColor: Colors.red));
      }
    } finally {
      if (mounted) {
        setState(() => _isScanning = false);
      }
    }
}
```

Listing 6: Funzione che permette la scansione del tag nfc

3.3.6 Studio degli algoritmi di crittografia

In questa sezione si espongono e si analizzano i principali algoritmi crittografici, evidenziandone il funzionamento e discutendo i rischi associati al loro impiego. Per molti esempi e descrizione degli scenari utilizzerò gli attori Alice e Bob che rappresentano due utenti generici che vogliono conversare tra di loro.

Per analizzare la comunicazione digitale è importante garantire quattro aspetti:

1. **Confidenzialità**: assicurare che nessun osservatore non autorizzato sia in grado di leggere il messaggio durante la trasmissione.
2. **Integrità**: garantire che il messaggio non sia alterato da nessuno.
3. **Autenticazione**: verificare che il mittente sia chi dichiara di essere.
4. **Non ripudio**: impedire che il mittente possa negare di aver inviato il messaggio.

Algoritmi a chiave simmetrica

Gli algoritmi di crittografia simmetrica utilizzano un'unica chiave segreta condivisa tra i due attori ed è usata sia per cifrare sia per decifrare il messaggio.
Il funzionamento è descritto come segue:

1. Alice e Bob concordano o si scambiano in modo sicuro una chiave condivisa k;
2. Alice prende il messaggio in chiaro P e applica un algoritmo di cifratura simmetrica S, ottenendo il messaggio cifrato C.
3. Quando Bob riceve C, applica l'algoritmo di decifratura D, utilizzando k, ricostruendo il messaggio originale P.

Pertanto per adottare questo algoritmo Alice e Bob devono conoscere e adoperare la stessa chiave utilizzando un canale di comunicazione sicuro.

- Se la chiave venisse rubata, l'attaccante può fingersi il mittente originale.
- Non è possibile distribuire in modo sicuro la chiave privata a un partner remoto senza utilizzare un altro sistema di sicurezza.

Algoritmi a chiave asimmetrica

Gli algoritmi di crittografia asimmetrica superano le limitazioni logistiche degli algoritmi di crittografia simmetrici. Per ogni attore coinvolto nella comunicazione viene utilizzata una coppia di chiavi fondamentali per il processi di cifratura e decifratura.
Il funzionamento è descritto come segue.

1. Bob genera una coppia di chiavi (k^B_{pub}, K^B_{priv}) e rende k^B pubblico a tutti;
2. Alice ricerca la chiave pubblica di Bob e cifra il messaggio in chiaro P con la chiave pubblica di Bob k^B_{pub}, ottenendo un messaggio cifrato C.
3. Bob riceve il messaggio e lo decifra con utilizzando la propria chiave privata k^B_{priv} ottenendo P.

L'uso di questi algoritmi risolve il problema dello scambio della chiave presente negli algoritmi di crittografia simmetrica in quanto la condivisione della chiave pubblica sulla rete non compromette la sicurezza del sistema. Inoltre si garantisce l'autenticazione e l'integrità dato che è possibile verificare l'identità del mittente invertendo il processo di crittografia. Se il messaggio viene cifrato con la chiave privata del mittente e successivamente è possibile decifrarlo con la sua chiave pubblica corrispondente. Inoltre questo ci permette di verificare che il messaggio non sia stato alterato durante la comunicazione.

3.3.7 Realizzazione di una *chat end-to-end*

Uno degli obbiettivi del progetto riguarda la generazione di coppie di chiavi pubbliche e private e il loro utilizzo per cifrare e decifrare i messaggi inseriti nell'applicazione. Per soddisfare questo, ho deciso di realizzare una *chat end-to-end* che permetta lo scambio di messaggi tra 2 utenti della *chat* mediante protocollo RSA (Rivest-Shamir-Adleman).

Generazione sicura delle chiavi

Il processo inizia con la creazione di un generatore di numeri pseudocasuali basata sul generatore `FortunaRandom`, messo a disposizione da `pointycastle`, per assicurare una adeguata fonte di entropia tale da garantire che il programma non generi coppie di chiavi con lo stesso valore numerico.

```
SecureRandom getSecureRandom() {
    final secureRandom = FortunaRandom();
    final seed = Uint8List(32);
    final random = Random.secure();
    for (int i = 0; i < seed.length; i++) {
      seed[i] = random.nextInt(256);
    }
    secureRandom.seed(KeyParameter(seed));
    return secureRandom;
  }
```

Listing 7: Generatore di numeri casuali

Questo generatore viene passato al metodo `generateRSAkeyPair()` responsabile della creazione delle chiavi RSA. Il generatore produce quindi una coppia di chiavi lunga 256 *byte* che rappresenta la lunghezza standard delle moderne chiavi RSA.

```
AsymmetricKeyPair<PublicKey, PrivateKey> generateRSAkeyPair(SecureRandom secureRandom) {
    final keyGen = RSAKeyGenerator();
    keyGen.init(
      ParametersWithRandom(
        RSAKeyGeneratorParameters(BigInt.parse('65537'), 2048, 64),
        secureRandom,)
    );
    return keyGen.generateKeyPair();
}
```

Listing 8: Generazione della coppia di chiavi

Per rispettare i requisiti e permettere sia allo *stakeholder* di visualizzare il risultato ottenuto le chiavi sono mostrate in chiaro sull'applicazione ma in un contesto pratico questo non deve avvenire in quanto compromette la sicurezza del sistema.

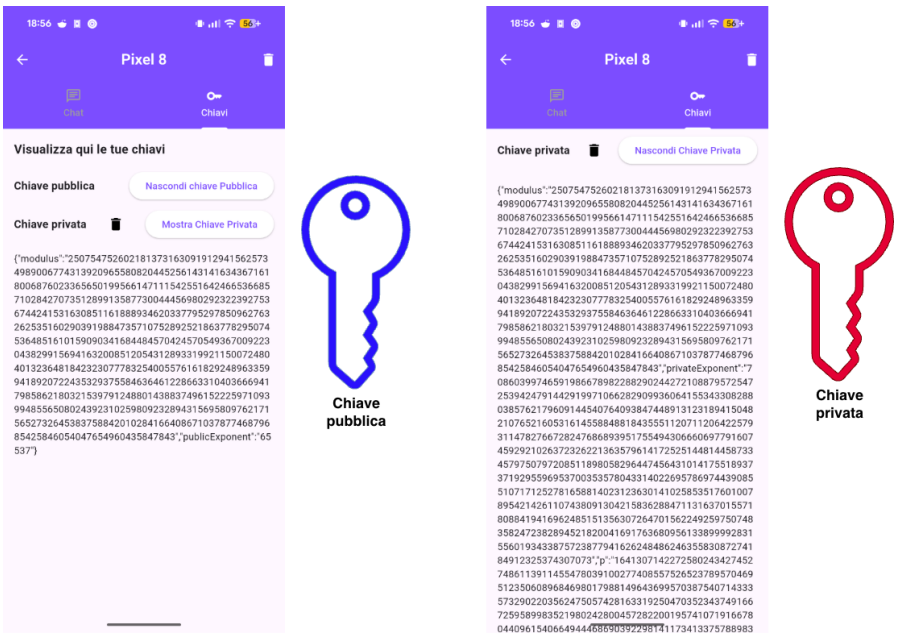


Figura 3.2: Risultato del processo di creazione delle chiavi

Codifica e decodifica del messaggio

Una volta generate le coppie di chiavi RSA è possibile utilizzarle per la realizzazione dei requisiti funzionali cinque e sei del progetto che riguardano la possibilità di cifrare e decifrare i messaggi inseriti nell'applicazione.
La cifratura è gestita dal metodo `rsaEncryptBase64()` che elabora il messaggio in chiaro tramite il motore crittografico `RSAEngine` sempre offerto dalla libreria `pointycastle` e dalla chiave pubblica del destinatario.

```
  Future<String?> rsaEncryptBase64(String plainText, RSAPublicKey publicKey) async {
  final engine = RSAEngine();
  engine.init(true, PublicKeyParameter<RSAPublicKey>(publicKey));

  try {
    final encrypted = engine.process(Uint8List.fromList(utf8.encode(plainText)));
    return base64Encode(encrypted);
  } catch (e) {
    return null;
  }
}
```

Listing 9: Codifica di un messaggio mediante l'uso delle chiave pubblica

Il messaggio cifrato restituito viene poi convertito in un formato *Base64* in modo tale che sia memorizzato e trasmesso al *database* in forma di stringa senza incorrere in problemi di formattazione. Come mostrato nell'immagine sottostante, il messaggio inserito risulta cifrato prima della memorizzazione.

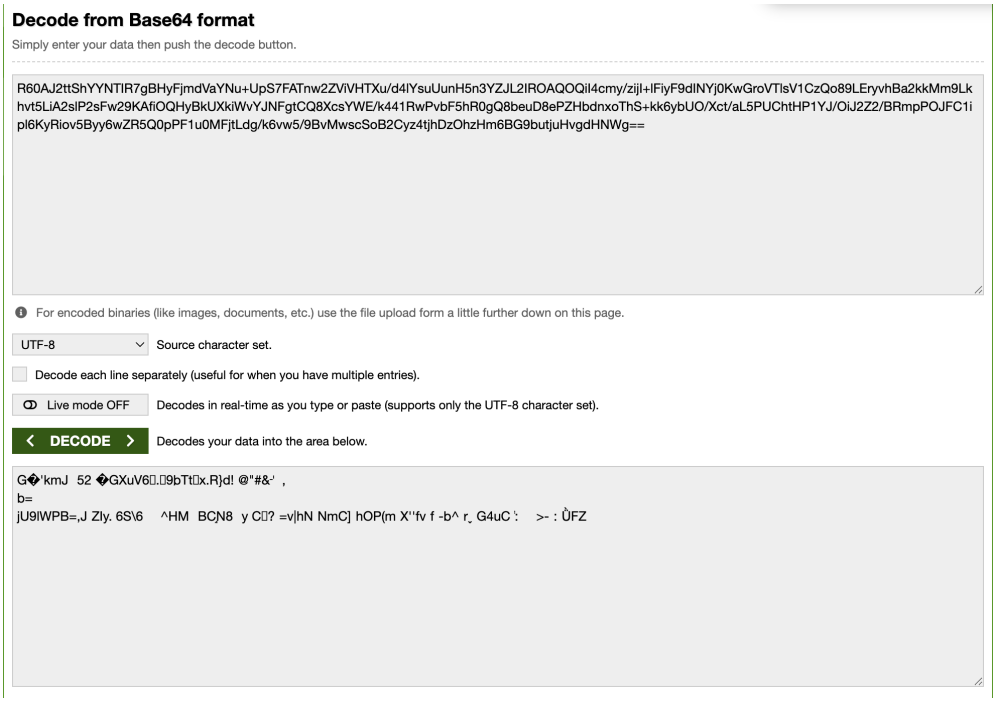


Figura 3.3: decodifica del testo cifrato in *base64* – Strumento: base64decode.org

La decifratura è affidata a `rsaDecryptBase64()`, che esegue l'operazione inversa decodificando il testo cifrato tramite l'uso della chiave privata del destinatario rendendo il testo di nuovo leggibile.

```
Future<String?> rsaDecryptBase64(String cipherText,RSAPrivateKey privateKey) async {
  final engine = RSAEngine();
  engine.init(false, PrivateKeyParameter<RSAPrivateKey>(privateKey));

  try {
    final decrypted = engine.process(base64Decode(cipherText));
    return utf8.decode(decrypted);
  } catch (e) {
    return "Messaggio non decodificabile";
  }
}
```

Listing 10: Decodifica di un messaggio mediante l'uso della chiave privata

Problema di questo algoritmo

L'implementazione dell'algoritmo mostrato presenta un problema specifico per la realizzazione di una *chat* tra due utenti. RSA consente la visione del messaggio in chiaro solo al destinatario (Bob) in quanto solo lui possiede la relativa chiave privata. Pertanto il mittente (Alice) dopo aver cifrato il testo con la chiave pubblica di Bob non ha più la possibilità di leggerlo.
È possibile procedere in due modi diversi per la risoluzione del problema:

- la prima soluzione consiste nel salvare localmente sul dispositivo del mittente una copia del messaggio in chiaro prima che esso venga cifrato e inviato al database. In questo modo l'interfaccia utente mostra la versione salvata localmente per i messaggi che invia il mittente.

- la seconda soluzione adottata nel progetto consiste nel inviare due copie dello stesso messaggio al *database*: la prima cifrata con la chiave pubblica del destinatario e la seconda cifrata con la chiave pubblica del mittente. In questo modo entrambi gli interlocutori possono recuperare la loro versione decifrabile dal *database* visualizzando i messaggi in chiaro.

Tuttavia questa scelta espone un rischio aggiuntivo in quanto un attaccante potrebbe individuare nel *database* due messaggi identici ma cifrati con chiavi differenti mettendo il sistema a rischio. Per ridurre questa possibilità ho introdotto lo schema di *padding* OAEP[6] all'algoritmo RSA che introduce casualità e rende impossibile ottenere due testi cifrati uguali dallo stesso messaggio.

[6]Inserire

```
Future<String?> rsaEncryptBase64(String plainText, RSAPublicKey publicKey) async {
  final engine = OAEPEncoding(RSAEngine());
  engine.init(true, PublicKeyParameter<RSAPublicKey>(publicKey));
  // ...
}

Future<String?> rsaDecryptBase64(String cipherText,RSAPrivateKey privateKey) async {
  final engine = OAEPEncoding(RSAEngine());
  engine.init(false, PrivateKeyParameter<RSAPrivateKey>(privateKey));
  // ...
}
```

Listing 11: Modifica agli algoritmi di codifica e decodifica aggiungendo OAEP

3.3.8 Gestione sicura delle chiavi pubbliche e private

La protezione delle chiavi crittografiche generate sul dispositivo dell'utente costituisce uno dei punti più delicati del progetto. L'applicazione pertanto è stata progettata per garantire il principio di controllo totale da parte dell'utente sulla chiave privata.

3.3.9 Separazione e archiviazione delle chiavi

Durante la creazione di un nuovo *wallet*, l'applicazione chiama la classe `WalletService` esegue la creazione del nuovo *wallet* generando una coppia di chiavi RSA chiamando la classe `CriptoUtils`, come scritto precedentemente.

```
final keyPair = cryptoUtils.generateRSAkeyPair(cryptoUtils.getSecureRandom());
final publicKeyObj = keyPair.publicKey as pointy.RSAPublicKey;
final privateKeyObj = keyPair.privateKey as pointy.RSAPrivateKey;
```

Listing 12: Generazione coppia di chiavi private in `wallet_service.dart`

A questo punto avviene le separazione delle due chiavi ricordando che la chiave privata deve essere salvata localmente sul dispositivo mentre quella pubblica deve essere trasferita insieme alle caratteristiche del *wallet* sul database.

// Inserire Immagine

La chiave pubblica viene inviata al *database* Firestore per essere associata al *wallet* dell'utente insieme ai dati di creazione quali userID, *email*, nome del *wallet*, i dati di scansione *historical bytes* e *standard*. Rendendola quindi disponibile sul *database* per le operazioni di comunicazione.

```
Map<String, dynamic> walletDataForFirestore = {
  'userId': userId,
  'email': email,
  'name': tempWallet.name,
  'hBytes': hBytes,
  'standard': standard,
  'color': colorString,
  'publicKey': tempWallet.publicKey,
  'createdAt': FieldValue.serverTimestamp(),
};

DocumentReference docRef = await _firestore.collection('wallets')
                                .add(walletDataForFirestore);
```

Listing 13: Invio delle informazioni pubbliche del *wallet* su Firebase

Per la chiave privata invece viene chiamata la classe di `SecureStorage` che utilizza la libreria di `flutter_secure_storage` per utilizzare il *Keychain* sui dispositivi iOS o il *KeyStore* per i dispositivi Android, proteggendo la chiave tramite crittografia a sistema operativo. La chiave è conservata solo per il tempo necessario al salvataggio locale sul dispositivo dopo l'operazione di scrittura il valore della chiave privata viene impostata su valore nullo assicurando che nessuna copia venga inutilmente salvata nella memoria temporanea del dispositivo.

```
await _secureStorage.writeSecureData(tempWallet.localKeyIdentifier,
                     tempWallet.transientRawPrivateKey!);
tempWallet.transientRawPrivateKey = null;
```

Listing 14: Salvataggio della chiave privata sul dispositivo

3.3.10 La classe `SecureStorage`

La classe `SecureStorage` è responsabile del salvataggio, lettura ed eliminazione della chiave privata sul dispositivo locale. Possiamo immaginare il funzionamento di `SecureStorage` come una mappa o un dizionario chiave valore, garantendo quindi che la chiave venga protetta in modo robusto.

```
class SecureStorage implements ISecureStorage {
  final FlutterSecureStorage storage;

  SecureStorage({FlutterSecureStorage? storage}) : storage = storage
                                    ?? const FlutterSecureStorage();

  @override
  Future<void> writeSecureData(String key, String value) async {
    await storage.write(key: key, value: value);
  }

  @override
  Future<String?> readSecureData(String key) async {
    final value = await storage.read(key: key);
    return value;
  }

  @override
  Future<void> deleteSecureData(String key) async {
    await storage.delete(key: key);
  }
}
```

Listing 15: La classe `SecureStorage`

Per accedere al dato salvato localmente è quindi necessario che alla creazione della coppia di chiavi venga anche generata una terza chiave identificativa più piccola che permetta il recupero della chiave privata dal KeyStore o dal Keychain

```
var uuid = const Uuid();
final newLocalKeyIdentifier = uuid.v4();
```

Listing 16: Creazione della chiave identificativa

Tramite il pacchetto `uuid` è possibile generare una chiave unica da associare alla chiave privata per permetterne il recupero. È importante sottolineare come questa chiave-identificatore possa essere memorizzata sul *database* in quanto non compromette la sicurezza del sistema.

3.3.11 Procedura di recupero chiave

Questa architettura delega all'utente la responsabilità esclusiva sulla propria chiave privata. In quanto non esiste alcun modo per rigenerarla partendo dai dati memorizzati nel *database*. In caso di smarrimento o trasferimento da un dispositivo a un altro l'applicazione permette una procedura di recupero. Tuttavia questo richiede che l'utente esporti manualmente la chiave. Al momento l'applicazione per motivi di tempo permette il recupero solo tramite inserimento di testo, tuttavia, la progettazione è stata eseguita considerando la possibilità di implementare un metodo alternativo di

salvataggio della chiave privata su un *tag* NFC singolo in modo tale che l'utente possa salvare in un dispositivo esterno allo *smartphone*

3.4 Risultati Raggiunti

(in questa sezione parlo dei risultati e raggiunti, delle funzioni implementate nell'applicazione come ad esempio la ricerca, la conversazione, e come le chiavi sono state associate ai documenti) ..

3.4.1 Risultati quantitativi

3.4.2 Risultati qualitativi

Bibliografia

Siti web consultati

Architettura a 3 livelli. URL: https://learn.microsoft.com/it-it/windows/win32/cossdk/using-a-three-tier-architecture-model.

Brevetto NFC. URL: https://patents.google.com/patent/US4384288A.

Design pattern provider. URL: https://pub.dev/packages/provider.

Documentazione Dart. URL: https://dart.dev/.

Documentazione Flutter. URL: https://flutter.dev/.

Flutter nfc kit. URL: https://pub.dev/packages/flutter_nfc_kit.

Flutter Secure Storage. URL: https://pub.dev/packages/flutter_secure_storage.

Integrazioni NFC nel contesto mobile. URL: https://webthesis.biblio.polito.it/31014/1/tesi.pdf.

Manuale Tecnico per gli erogatori di servizi pubblici e privati. URL: https://docs.italia.it/media/pdf/cie-manuale-tecnico-docs/bozza/cie-manuale-tecnico-docs.pdf.

Organizzazione open-source italiana per lo sviluppo di software governativi. URL: https://github.com/italia.

Pointycastle. URL: https://pub.dev/packages/pointycastle.

SDK entra con CIE. URL: https://github.com/italia/cieid-android-sdk.

Sito SyncLab. URL: https://www.synclab.it.

Specifiche chip carta d'identità elettronica italiana. URL: https://www.cartaidentita.interno.gov.it/downloads/2021/03/cie_3.0_-_specifiche_chip.pdf.