UNIVERSITÀ
DI TORINO

Università Degli Studi di Torino

Corso di laurea in Informatica

Enhance the security in banking applications: Dynamic Logo with Kotlin Multiplatform

Relatore:
Prof. Ferruccio Damiani

Candidato:
Valerio Gozzellino
matricola: 995974

Anno accademico 2023/2024

Contents

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Abstract

Questa tesi presenta lo sviluppo di un SDK in Kotlin Multiplatform per applicazioni Android e iOS, con l'obiettivo di implementare un sistema di sicurezza basato su un Logo Dinamico con QR Code criptato.
Inizialmente, lo studio si è concentrato sull'apprendimento di Kotlin Multiplatform e Jetpack Compose Multiplatform, strumenti che facilitano la creazione di interfacce utente e logica condivisa tra piattaforme diverse.
Attraverso lo sviluppo di un prototipo, è stata testata l'efficacia di questi strumenti nel gestire processi crittografici complessi, come la generazione e la scansione di QR Code criptati, in ambienti Android e iOS.
Il prototipo ha evidenziato le sfide legate alla completa condivisione della logica a causa delle specifiche hardware dei dispositivi, portando a riflessioni critiche sulle limitazioni attuali delle tecnologie multipiattaforma.

Infine, l'esperienza acquisita è stata utilizzata per l'implementazione dell'SDK, che incorpora le migliori pratiche di design e programmazione scoperte durante lo stage e all'università. Questo lavoro mira a contribuire alla comprensione della portabilità e dell'efficienza delle soluzioni di sicurezza distribuite su dispositivi mobili diversi.

1 Introduzione

1.1 Collaborazione

Questa tesi è stata sviluppata in stretta collaborazione con **IrishCube Reply**, un'azienda di spicco nel settore della consulenza informatica, riconosciuta per la sua eccellenza nell'innovazione tecnologica.
L'obiettivo del progetto è stato l'implementazione di un avanzato **SDK** (Software Development Kit), progettato per integrare un innovativo sistema di sicurezza basato su un Logo Dinamico. Questo metodo rivoluzionario è stato concepito per ottimizzare la gestione degli accessi e delle transazioni all'interno di applicazioni bancarie, garantendo un livello di protezione senza precedenti.
La collaborazione con IrishCube Reply ha offerto l'opportunità di sfruttare competenze all'avanguardia e risorse tecnologiche di alto livello, contribuendo in modo significativo alla realizzazione di un progetto di grande rilevanza per il settore bancario.

1.2 La rivoluzione del Multiplatform

Da sempre, una delle migliori pratiche per uno sviluppatore è evitare la duplicazione del codice, puntando a rendere il software il più efficiente possibile.
Nel contesto dello sviluppo di applicazioni e siti Web, l'adozione di linguaggi multipiattaforma è diventata essenziale per eliminare la ridondanza e ottimizzare tempi e risorse. In questo scenario, **Kotlin Multiplatform** e **(Jetpack) Compose Multiplatform** emergono come soluzioni all'avanguardia.
Questo nuovo paradigma sta rivoluzionando il panorama tecnologico, consentendo alle aziende di rispondere più rapidamente alle esigenze del mercato e di ridurre i costi di sviluppo, senza compromettere la qualità del prodotto finale.

1.3 Obiettivo

All'interno delle applicazioni bancarie, vi è una crescente necessità di adottare metodi di sicurezza avanzati per prevenire e proteggere i clienti da frodi e attacchi mirati. Uno di questi metodi è il **Logo Dinamico**, un processo attraverso il quale la scansione di un logo, che varia nel tempo, garantisce la sicurezza dei clienti, autorizzando un unico e specifico processo.
Questa tesi si concentra sull'approfondimento delle competenze legate agli algoritmi di creazione e validazione di un **Logo Dinamico** sviluppato in **Kotlin Multiplatform**.

L'obiettivo principale è quello di realizzare un **SDK** (Software Development Kit) con tecnologia Kotlin Multiplatform, capace di:

- **Generare** un Logo Dinamico a partire da un set di dati da "condividere".
- **Scansionare** e interpretare il set di dati a partire da un logo dinamico generato dalla libreria su un dispositivo diverso.

1.4 Traguardi raggiunti

Il progetto è stato sviluppato in soli due mesi, partendo dalle basi di **Kotlin**, dei **QR Code** e del concetto di **SDK**, per poi arrivare a utilizzare tecnologie avanzate e complesse, che hanno consentito lo sviluppo di una logica di business altrettanto sofisticata.

In breve tempo, è stato possibile sviluppare diverse applicazioni: una per richiedere le previsioni meteo da API esterne, un'altra per scattare foto, oltre ad applicazioni per la generazione e la scansione dei QR Code. Questo processo ha permesso di comprendere a fondo l'importanza delle dipendenze di un progetto, dell'organizzazione e modularizzazione delle cartelle, e dell'uso dei pattern di programmazione. Il progetto ha anche permesso di approfondire e gestire i vari aspetti meno visibili che stanno dietro lo sviluppo di un'applicazione.

Al termine del percorso è stato sviluppato un SDK in grado di fornire funzionalità per la generazione e la scansione dei QR Code. Inoltre, sono state effettuate valutazioni che hanno portato alla conclusione che, nonostante Kotlin Multiplatform sia ancora in fase di sviluppo, è stato possibile per questo progetto condividere gran parte del codice, accelerando le fasi di implementazioni della buisness logic.

2 Requisiti

Il progetto di questa tesi si focalizza sulla realizzazione di un **SDK** in grado di offrire un sistema di sicurezza basato su un logo dinamico composto da QR Code generati da una chiave crittografata.
Questo SDK costituisce le fondamenta di un sistema che potrebbe essere integrato in progetti futuri per garantire la sicurezza nelle transazioni o nelle comunicazioni. Di seguito vengono dettagliati i requisiti implementativi necessari per il successo del progetto.

2.1 Requisiti Funzionali

2.1.1 Funzionalità Principali

L'SDK implementato deve fornire un sistema di sicurezza basato su un logo dinamico, il quale è composto da una serie di QR Code generati utilizzando una chiave crittografata.
Il logo dinamico viene visualizzato su un dispositivo e deve essere scannerizzato da un altro dispositivo, il quale si occuperà di decriptare la chiave e ricomporla correttamente per convalidare il processo.

2.1.2 Interazione Utente

L'utente è coinvolto attivamente in due fasi critiche:

- **Generazione del logo dinamico:** L'utente deve fornire una chiave che servirà per generare i QR Code che compongono il logo.
- **Scannerizzazione del logo:** L'utente deve scansionare correttamente i QR Code generati per permettere al sistema di decriptare la chiave e validare l'operazione.

2.1.3 Gestione degli Errori

Gli errori vengono gestiti principalmente nella fase di scannerizzazione. Se un QR Code che contiene un segmento della chiave non viene scannerizzato correttamente, è probabile che venga visualizzato nuovamente nel corso della fase di scannerizzazione per essere rielaborato. In altri casi, la perdita di un singolo QR Code è tollerabile, e il sistema deve gestire queste situazioni senza compromettere la sicurezza o l'integrità del processo.

2.1.4 Prestazioni

Le prestazioni del sistema devono essere elevate, sia nella fase di generazione dei QR Code sia durante la scannerizzazione.
Questo requisito funzionale è stato soddisfatto implementando l'uso dei **Thread**[1], permettendo un'esecuzione parallela delle operazioni e garantendo una reattività ottimale.

2.1.5 Sicurezza

La sicurezza è stata garantita attraverso la crittografia della chiave utilizzata per generare i QR Code, la dinamicità del logo (che varia nel tempo) e la temporizzazione della validità della chiave stessa.
Questo approccio garantisce che il logo dinamico sia valido solo per un periodo di tempo limitato, riducendo così il rischio di intercettazione e manipolazione.

2.2 Requisiti Non Funzionali

2.2.1 Usabilità

L'**SDK** deve essere facile da integrare nei progetti esistenti e la sua interfaccia deve essere intuitiva per gli sviluppatori che lo utilizzeranno. Inoltre, l'interazione utente deve essere semplice e diretta, garantendo un'esperienza fluida durante l'utilizzo del sistema di sicurezza.

2.3 Requisiti Tecnici

2.3.1 Ambiente di Sviluppo

L'ambiente di sviluppo scelto è Android Studio, utilizzando Kotlin Multiplatform per garantire che l'**SDK** possa essere implementato su più piattaforme, in particolare su Android e iOS.
Questo ambiente offre gli strumenti necessari per gestire un progetto complesso e multipiattaforma come questo. In oltre, per l'esecuzione del sistema su dispositivi iOS, è stato utilizzato Xcode, si tratta di un ambiente che utilizza come linguaggio di programmazione **Swift**, specifico per lo sviluppo iOS.

[1]Un thread è una suddivisione di un processo in due o più istanze o sottoprocessi che vengono eseguiti concorrentemente

2.3.2 Compatibilità delle Piattaforme

L'SDK generato è compatibile con le piattaforme iOS e Android, garantendo un ampio spettro di utilizzo.
Nonostante il focus iniziale sia su queste due piattaforme, il progetto è strutturato in modo tale da permettere sviluppi futuri per altre piattaforme, ampliando così le potenzialità d'uso dell'SDK.

2.3.3 Struttura del Codice

Il codice è stato progettato per essere modulare e pulito, rispettando i principi di buona programmazione e facilitando la manutenzione e l'estensione futura.
La business logic è stata separata dalla UI per garantire che l'SDK possa essere facilmente integrato e adattato a diversi contesti applicativi.

2.4 Requisiti di Test

2.4.1 Tipi di Test

Sono stati condotti vari test, inclusi test con chiavi nulle e test di stress[2], per verificare la robustezza del sistema.
Questi test sono stati fondamentali per assicurare che il sistema risponda correttamente anche in condizioni non ottimali, garantendo così la sicurezza e l'affidabilità.
Inoltre, sono stati condotti benchmark con un maggiore focus sulle prestazioni dei dispositivi, analizzati eventuali errori che potevano emergere simulando un comportamento di leggera distrazione da parte dell'utente nella fase di scannerizzazione del logo.

2.5 Vincoli e Limitazioni

2.5.1 Tempo e Risorse

A causa di vincoli di tempo, non è stato possibile sviluppare un logo graficamente sofisticato utilizzando tecnologie avanzate come la *steganografia*.

[2]Il test di stress mette alla prova la quantità di sforzo che il sistema può sopportare prima di guastarsi

Tuttavia, questo rimane un potenziale sviluppo futuro, che potrebbe migliorare ulteriormente la sicurezza e l'estetica del sistema.

3 Kotlin : Dalle Basi al Multiplatform

3.0.1 Introduzione e Storia di Kotlin

Kotlin è un linguaggio di programmazione moderno sviluppato da **JetBrains**, una società nota per i suoi strumenti di sviluppo come **IDE (Integrated Development Environment)** e numerosi linguaggi di programmazione.

Lo sviluppo di Kotlin è iniziato nel 2010, con il linguaggio che è stato annunciato pubblicamente nel luglio 2011. Il primo compilatore di Kotlin è stato rilasciato nel febbraio 2012 come open source, e la versione 1.0 stabile è stata resa disponibile nel febbraio 2016. [14]

3.0.2 Interoperabilità con Java

Uno degli aspetti più distintivi di Kotlin è la sua **interoperabilità** completa con **Java**. Questo significa che è possibile chiamare codice Java da Kotlin e viceversa, senza particolari configurazioni o complessità.
Kotlin compila in **bytecode Java**, permettendo ai due linguaggi di convivere nello stesso progetto. Questa interoperabilità rende Kotlin una scelta ideale per sviluppatori Java che vogliono adottare un linguaggio moderno senza riscrivere tutto il codice esistente.

Codice Java

```java
// Codice Java
public class JavaExample {
    public String getGreeting() {
        return "Hello from Java!";
    }
}
```

Codice Kotlin

```kotlin
// Codice Kotlin
fun main() {
```

```
    val example = JavaExample()
    println(example.greeting)  // Interoperabilita con Java
}
```

In questo esempio, la classe `JavaExample` scritta in Java può essere utilizzata direttamente in Kotlin, dimostrando la perfetta **interoperabilità** tra i due linguaggi.[8]

3.0.3 Descrizione del Linguaggio e Tipizzazione Statica

Kotlin è un linguaggio con **tipizzazione statica**, il che significa che i tipi delle variabili sono determinati durante la compilazione e non possono cambiare durante l'esecuzione.
Questo riduce la possibilità di errori runtime e migliora la sicurezza del codice.

```
val name: String = "Kotlin"
name = 42  // Questo generer  un errore di compilazione
```

Kotlin supporta diversi paradigmi di programmazione, inclusi la **programmazione orientata agli oggetti** (OOP) e la **programmazione funzionale**.
La sintassi di Kotlin è progettata per essere concisa e leggibile, eliminando la necessità di codice boilerplate[3] tipico di Java, come getter e setter.

3.0.4 Descrizione del Linguaggio e Tipizzazione Statica

Le annotazioni in Kotlin sono metadati che forniscono informazioni aggiuntive al compilatore o agli strumenti di runtime. Alcune delle annotazioni più comuni in Kotlin includono:

@JvmStatic: Rende una funzione in un oggetto Kotlin accessibile come funzione statica da Java.

@JvmOverloads: Genera automaticamente versioni sovraccaricate di funzioni con parametri di default.

@NotNull e **@Nullable**: Aiutano a gestire i tipi nullabili e non nullabili, migliorando la sicurezza del codice contro i `null pointer exception`.

```
class Example {
    @JvmOverloads
    fun greet(name: String = "World") {
        println("Hello, $name")
```

[3]boilerplate: sono sezioni di codice che si ripetono in posizioni diverse con minime modifiche

```
    }
}
```

Queste annotazioni rendono Kotlin altamente interoperabile con Java, mantenendo al contempo una forte tipizzazione e sicurezza del codice.

3.0.5 Coroutines e Programmazione Asincrona

Kotlin introduce le **coroutines**, un potente strumento per la programmazione asincrona.
Le **coroutines** facilitano l'esecuzione di operazioni lunghe, come richieste di rete o accessi a database, senza bloccare il thread principale.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Coroutines!")
    }
    println("Hello,")
}
```

Le coroutines permettono di scrivere codice asincrono in maniera sequenziale, rendendo più semplice la gestione del codice rispetto ai tradizionali callback o promesse.

3.1 Kotlin Multiplatform

Kotlin Multiplatform (KMP) è una delle innovazioni più significative di Kotlin, permettendo agli sviluppatori di scrivere codice condiviso che può essere eseguito su diverse piattaforme come Android, iOS, Web, e Desktop.
KMP non è un framework completo come **Flutter** o **React Native**, ma piuttosto una soluzione che permette di condividere logica di business, modelli di dati, e altro codice non dipendente dalla piattaforma, lasciando al contempo la possibilità di scrivere codice specifico per ogni piattaforma quando necessario.

Ad esempio, in **KMP**, una funzione `expect` può essere dichiarata nel codice comune, mentre le sue implementazioni specifiche (`actual`) sono fornite per ogni piattaforma. [8]

```
expect fun platformName(): String

actual fun createApplicationScreenMessage() : String
    {return "Hello from ${platformName()}"}
```

KMP consente di ridurre la duplicazione del codice, facilitando la manutenzione e accelerando lo sviluppo di applicazioni che devono funzionare su più piattaforme. Il codice comune può includere logica di business, algoritmi di crittografia, gestione della rete e molto altro, lasciando solo la parte di interfaccia utente o di interazione con l'hardware come codice specifico della piattaforma.

Kotlin Multiplatform sta guadagnando popolarità grazie alla sua flessibilità e alla capacità di adattarsi a esigenze diverse, permettendo alle aziende di sviluppare applicazioni mantenendo un'unica base di codice per più piattaforme, con evidenti vantaggi in termini di tempo e costi.

3.1.1 Altri Aspetti Importanti di Kotlin

Kotlin include molte altre caratteristiche avanzate che lo rendono un linguaggio potente e moderno:

Estensioni di Funzione: Consentono di aggiungere nuove funzionalità a classi esistenti senza modificarne il codice originale.

Smart Casts: Il compilatore di Kotlin è in grado di effettuare cast intelligenti, riducendo la necessità di cast espliciti e migliorando la leggibilità del codice. [13]

3.2 Conclusione

Kotlin si distingue come un linguaggio di programmazione moderno e potente, particolarmente adatto per lo sviluppo Android e per progetti multipiattaforma grazie a Kotlin Multiplatform.
Con caratteristiche come **tipizzazione statica**, **interoperabilità** con Java, **coroutines** per la programmazione asincrona, e un solido sistema di annotazioni, Kotlin offre agli sviluppatori un set di strumenti completo per affrontare le sfide dello sviluppo moderno.
Kotlin Multiplatform, in particolare, rappresenta un approccio flessibile e innovativo per sviluppare applicazioni che possono essere eseguite su diverse piattaforme mantenendo una base di codice comune.

4 Soluzioni Tecnologiche

4.1 Kotlin Multiplatform

Questo linguaggio è stato rapidamente adottato da importanti aziende che necessitano di sviluppare applicazioni per una varietà di piattaforme, tra cui **JVM, Android, iOS, Web, Desktop e Server**.

Secondo un sondaggio "*The State of Kotlin Multiplatform*", una grande percentuale di sviluppatori utilizza KMP in produzione, mentre una percentuale minore ha partecipato a più di un progetto KMP.
Questo indica che KMP è ampiamente utilizzato e apprezzato nella comunità degli sviluppatori.

La caratteristica distintiva di **Kotlin Multiplatform** è la capacità di compilare lo stesso codice per diversi target, producendo librerie native ottimizzate per ogni piattaforma. Questo permette di ottenere prestazioni elevate su ogni dispositivo, mantenendo al contempo un'elevata condivisione del codice.

È importante sottolineare che Kotlin Multiplatform non è un framework come Flutter o React Native, ma un linguaggio di programmazione completo con un compilatore poliglotta e una vasta gamma di librerie multipiattaforma. Questo approccio offre una flessibilità superiore rispetto ai framework tradizionali, consentendo agli sviluppatori di scrivere codice nativo ottimizzato per ciascuna piattaforma, pur mantenendo un alto livello di condivisione del codice.

KMP utilizza compilatori diversi per produrre codice nativo per target specifici. Il codice che viene scritto non è lo stesso che verrà eseguito sulle piattaforme finali, poiché viene tradotto in un altro linguaggio da un programma speciale chiamato compilatore Kotlin.
Kotlin supporta principalmente tre linguaggi: **Java, JavaScript e C/Objective-C**, per i quali sono disponibili tre diversi compilatori: **Kotlin/JVM, Kotlin/JS e Kotlin/Native**.[1]

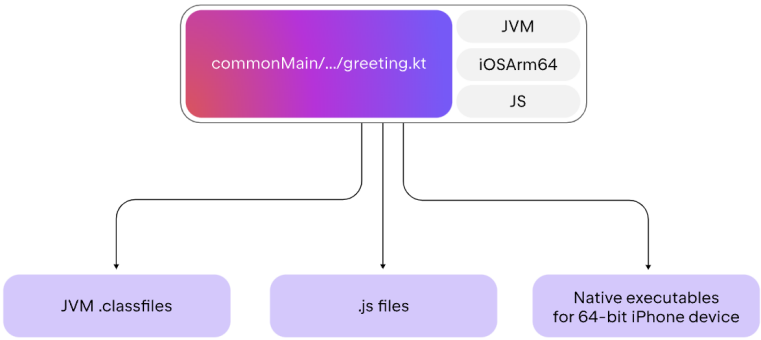

Figure 1: Schema del compilatore Kotlin Multiplatform[8]

4.1.1 Condivisione del Codice

Kotlin Multiplatform consente agli sviluppatori di condividere codice tra diverse piattaforme, migliorando significativamente i tempi di sviluppo e riducendoli di circa il **40%**.
In KMP, esistono vincoli di condivisione del codice che permettono di distinguere chiaramente le parti condivise da quelle specifiche per ciascuna piattaforma.
Solitamente, il codice condiviso si trova nella cartella *"commonMain"*, ma non tutto il codice può essere condiviso: il compilatore impedisce l'uso di funzioni specifiche per le piattaforme all'interno di commonMain.
Fortunatamente, KMP è in rapida espansione e stanno emergendo numerose librerie e **SDK multiplatform**, che possono essere facilmente integrati nei progetti, offrendo funzionalità avanzate e rendendo lo sviluppo ancora più efficiente.

4.1.2 Struttura di un Progetto Kotlin Multiplatform

Un progetto Kotlin Multiplatform è organizzato in modo tale da separare chiaramente il codice condiviso da quello specifico per ciascuna piattaforma.
Il codice che può essere riutilizzato su più piattaforme viene inserito nel source set `commonMain`, che tipicamente contiene la logica di business e, grazie a **Compose Multiplatform**, anche componenti dell'interfaccia utente.
Questo codice condiviso viene quindi compilato in librerie native per ogni piattaforma target, ottimizzando le prestazioni senza compromettere la portabilità.
Oltre a `commonMain`, i progetti Kotlin Multiplatform possono includere altri

source set intermedi, che consentono di condividere codice tra gruppi specifici di piattaforme, come `androidMain` o `iosMain`.
Questi source set intermedi permettono di gestire meglio le specificità delle diverse piattaforme, riducendo al minimo la duplicazione del codice.

Il codice che non può essere condiviso viene invece collocato nei source set specifici delle piattaforme, come `androidMain`, `iosMain`, o `jsMain`.
Quando si aggiunge un nuovo target al progetto, è necessario aggiornare il file `build.gradle.kts` per includere il nuovo target, il che genererà automaticamente una nuova cartella per quella piattaforma.
All'interno di questa cartella, si potrà implementare il codice specifico per la piattaforma di riferimento. Inoltre, è possibile includere librerie e SDK specifici per alcune piattaforme direttamente nei rispettivi source set, garantendo che ogni piattaforma utilizzi le librerie più adatte alle sue esigenze, mantenendo al contempo la struttura del progetto modulare e flessibile.

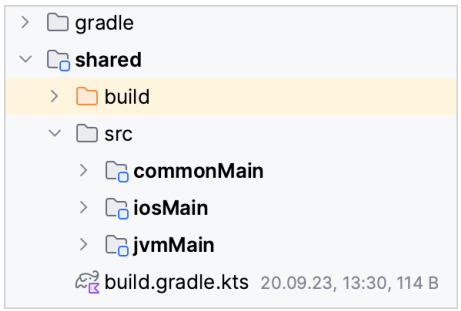

Figure 2: Struttura delle directory in un progetto Kotlin Multiplatform[8]

4.1.3 Expect e Actual

Nonostante i numerosi vantaggi offerti da Kotlin Multiplatform, esistono anche alcune limitazioni.
Una delle principali è legata alla variabilità delle componenti hardware tra le diverse piattaforme target.
In questi casi, non è possibile condividere direttamente alcune parti di codice, richiedendo l'uso del pattern `expect/actual`.

Il pattern `expect/actual` è utilizzato per gestire le funzioni specifiche di una piattaforma all'interno di un progetto Kotlin Multiplatform.
Una funzione `expect` viene dichiarata nel codice condiviso (commonMain), mentre la sua implementazione concreta `actual` deve essere fornita per ciascun target specifico.
Durante la compilazione, la funzione `expect` viene associata al corrispondente `actual`, il che può comportare una duplicazione del codice per ciascun target.
Tuttavia, questa soluzione temporanea sarà probabilmente superata con il crescente utilizzo di Kotlin Multiplatform e lo sviluppo di nuove librerie, come dimostra l'incremento del numero di librerie disponibili, passate da meno di 500 nel 2021 a oltre 2000 nel 2024.[9]

4.1.4 Dependency Injection

L'Iniezione delle Dipendenze **(Dependency Injection)** è un design pattern di programmazione orientato agli oggetti che mira a decoppiare i componenti di un'applicazione e a gestirne le dipendenze in modo flessibile e modulare.
In questo pattern, la responsabilità di creare e fornire le dipendenze viene spostata dal componente dipendente a un'entità esterna, nota come **injector**.
Quando un componente necessita di determinate dipendenze, l'**injector** si occupa di risolverle al momento dell'istanza del componente.
Se la dipendenza viene risolta per la prima volta, l'**injector** istanzia il componente, lo salva in un contenitore di istanze e lo restituisce.
In caso contrario, restituisce la copia salvata nel contenitore.
Questo approccio implementa un pattern importante, il **Singleton**, come descritto nel libro **"Gang of Four"**, e garantisce un'istanza **Lazy (pigra)** degli elementi, migliorando così l'efficienza e la gestione della memoria.

4.1.5 Koin

Koin è un framework leggero e pragmatico per l'iniezione delle dipendenze, specificamente progettato per gli sviluppatori Kotlin.

Koin semplifica il processo di gestione delle dipendenze nei progetti Kotlin Multiplatform (KMP), permettendo di decoppiare il codice e rendendolo più facile da testare e mantenere.[12]

Obiettivi di Koin:

- **Semplificazione dell'infrastruttura**: **Koin** offre un'API intelligente che semplifica l'infrastruttura di Dependency Injection.
- **Kotlin DSL intuitiva**: Fornisce una **Domain-Specific Language (DSL)** in Kotlin che è facile da leggere e usare, permettendo di scrivere qualsiasi tipo di applicazione.
- **Integrazione versatile**: **Koin** supporta l'integrazione con diverse parti dell'ecosistema Kotlin, dall'ambiente Android a necessità di backend come **Ktor**.
- **Supporto per annotazioni**: Consente l'uso delle annotazioni per una configurazione ancora più flessibile delle dipendenze.

4.1.6 Napier

Napier è una libreria di logging essenziale per lo sviluppo di applicazioni Kotlin Multiplatform.
Questa libreria consente di integrare un sistema di logging efficace per il debug dell'applicazione, garantendo che i messaggi di log generati nel codice condiviso vengano visualizzati correttamente nelle rispettive console di log dei singoli target (Android, iOS, etc.) su cui l'applicazione è eseguita.
Una delle caratteristiche distintive di Napier è la sua capacità di gestire diversi livelli di log, come informazioni, avvisi, errori, e debug, permettendo agli sviluppatori di tracciare specifiche condizioni dell'applicazione.
Ad esempio, in caso di errori critici, permette di utilizzare flag specifici per evidenziare e isolare tali eventi, facilitando così la diagnosi e la risoluzione dei problemi. Inoltre, Napier supporta la personalizzazione del comportamento dei log

a seconda della piattaforma e dell'ambiente, offrendo un controllo granulare su come e dove vengono registrati i messaggi di log.

Grazie alla sua flessibilità e alla facile integrazione, **Napier** si è dimostrata una risorsa preziosa per mantenere il codice multipiattaforma pulito e facilmente debuggabile, migliorando significativamente il flusso di sviluppo e la qualità del software.[2]

4.2 Jetpack Compose Multiplatform

Jetpack Compose Multiplatform è un toolkit UI dichiarativo che consente di implementare interfacce utente condivise su una varietà di dispositivi.
La combinazione di Kotlin Multiplatform e Compose Multiplatform offre un approccio unificato, che sta trasformando radicalmente il modo in cui vengono sviluppate le applicazioni.
In Compose, un widget è definito come una funzione Kotlin annotata con `@Composable`.
Questa annotazione informa il compilatore che la funzione deve essere gestita in modo speciale, permettendo al sistema di ottimizzare la gestione dello stato e delle interfacce utente.
Una delle caratteristiche distintive di questo toolkit è che, a differenza di molti altri framework, non è necessario specificare manualmente quando e quale porzione di codice della componente deve essere aggiornata in seguito al cambiamento di stato di una variabile.
Compose si occupa automaticamente di ricomporre solo la porzione di interfaccia utente in cui la variabile modificata è visualizzata, garantendo un aggiornamento efficiente e reattivo dell'interfaccia.
Compose offre anche la libreria **Material3**, che include una vasta gamma di widget predefiniti e facilmente integrabili, rendendo più veloce lo sviluppo di interfacce utente moderne e consistenti.

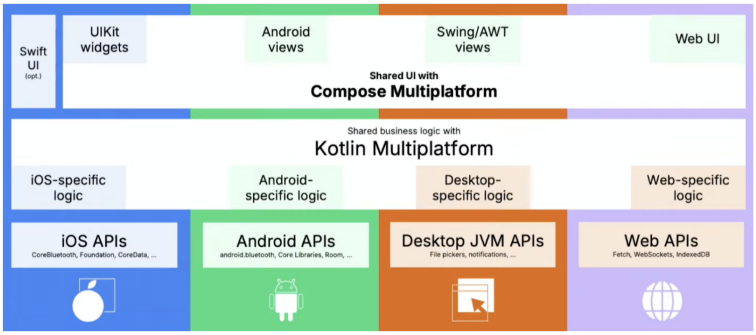

Figure 3: target di Jetpack Compose Multiplatform [8]

4.3 Logo Dinamico

In questa sezione della tesi analizzeremo in dettaglio il concetto di **Logo Dinamico**, esplorando sia le sue caratteristiche tecniche sia le valutazioni personali in merito. Approfondiremo le possibili implementazioni e discuteremo come questo strumento possa essere integrato efficacemente all'interno di applicazioni, in particolare in ambito bancario.

4.3.1 Che cos'è un Logo Dinamico?

Un **Logo Dinamico** è un'immagine visualizzata su un dispositivo che cambia costantemente aspetto nel tempo.
A differenza di un logo statico, un **Logo Dinamico** non mantiene mai la stessa forma, ma subisce variazioni continue, rendendolo unico e difficile da replicare o contraffare.
Questa proprietà lo rende particolarmente utile nelle applicazioni bancarie, dove viene utilizzato per autenticare in modo sicuro e univoco i processi eseguiti dagli utenti.

4.3.2 Vantaggi del Logo Dinamico

L'utilizzo di un Logo Dinamico nei sistemi di sicurezza bancaria introduce una barriera significativa contro le frodi e gli attacchi illeciti.
Poiché il logo cambia continuamente, diventa estremamente difficile per un malintenzionato catturare un'immagine statica che possa essere utilizzata per attività fraudolente.
Il processo di scannerizzazione deve avvenire in tempo reale e in modo continuo, poiché il logo è valido solo per un periodo di tempo molto limitato.
Questo approccio aumenta notevolmente la sicurezza delle transazioni e delle operazioni sensibili, offrendo un ulteriore livello di protezione per i clienti.

4.3.3 Implementazione di un Logo Dinamico

Esploreremo le tecniche di implementazione di un Logo Dinamico utilizzando Kotlin Multiplatform, esaminando le sfide e le opportunità che questo approccio può offrire.
Un caso d'uso tipico prevede l'interazione tra due dispositivi: uno genera il logo

dinamico e l'altro lo scannerizza.
Quando l'operazione di scannerizzazione ha successo, il dispositivo scannerizzante ottiene l'accesso alle operazioni richieste.
Il primo passo nell'implementazione di un logo dinamico consiste nel valutare i requisiti tecnici per la sua creazione.
È necessario un metodo per inserire informazioni all'interno di un'immagine o di un codice che possa nascondere tali informazioni all'occhio umano, ma che siano leggibili da un dispositivo che effettua la scansione.
Dopo una lunga ricerca, ho identificato diversi metodi che consentono di inserire informazioni in un oggetto visibile.

4.4 Codice a Barre

Un codice a barre è un sistema di identificazione nel quale l'alto contrasto tra le barre e gli spazi permette di trasmettere informazioni a un dispositivo dotato di un metodo di scansione.
Esistono due principali tipi di codici a barre, che si distinguono per la loro struttura: **Lineari e Bidimensionali**.

Figure 4: esempi di codici bidimensionale e lineare

4.4.1 Codici a Barre Lineari

I codici a barre lineari sono ampiamente utilizzati nelle grandi distribuzioni e nelle farmacie per velocizzare il sistema di identificazione dei prodotti.
Essi presentano un carattere di start per l'inizio della lettura e uno di stop per delimitarne la fine.

I codici a barre lineari possono essere ulteriormente classificati in **discreti** e **continui**:

- I codici discreti non contengono informazioni negli spazi bianchi tra le barre.
- Nei codici continui, anche gli spazi bianchi contengono informazioni, che vengono scansionate insieme alle barre.

4.4.2 Codici a Barre Bidimensionali

I codici a barre bidimensionali, più comunemente noti come **QRCode** (Quick Response Code), sono progettati per essere letti rapidamente. Questi codici consistono in moduli neri disposti in una matrice bidimensionale, generalmente di forma quadrata, e possono essere scansionati da dispositivi come gli smartphone.
Un QR Code può contenere fino a 7.089 caratteri numerici o 4.296 caratteri alfanumerici, permettendo l'inserimento di indirizzi web, numeri di telefono, testi, e altre informazioni.
Tuttavia, in alcune situazioni, un QR Code potrebbe essere danneggiato o parzialmente illeggibile, ad esempio a causa di macchie o altri ostacoli. In questi casi, entra in gioco l'algoritmo di correzione degli errori Reed-Solomon, che consente il recupero delle informazioni mancanti o danneggiate.[15]

4.4.3 Correzione degli Errori nei QRCode

Esistono diversi livelli di correzione degli errori nei QRCode, che determinano la percentuale di dati che possono essere recuperati:

- **Livello L**: circa il 7% delle parole del codice può essere ripristinato.
- **Livello M**: circa il 15% delle parole del codice può essere ripristinato.
- **Livello Q**: circa il 25% delle parole del codice può essere ripristinato.
- **Livello H**: circa il 30% delle parole del codice può essere ripristinato.

4.4.4 Metodi di Criptazione dei QR Code

I QR code sono uno strumento versatile, ma vulnerabile ad attacchi come phishing, malware e manomissioni. L'implementazione di tecniche crittografiche nei QR code

può migliorare la sicurezza, garantendo confidenzialità, autenticità e integrità dei dati.
Tra i principali metodi di criptazione implementabili nei QR code, vi sono:

- **Firma Digitale (RSA e ECDSA):** Le firme digitali sono tra i metodi più utilizzati per garantire l'autenticità dei dati. Gli algoritmi RSA e Elliptic Curve Digital Signature Algorithm (ECDSA) sono due opzioni principali. RSA utilizza chiavi di lunghezza variabile (1.024, 2.048 e 3.072 bit), mentre ECDSA è più efficiente con chiavi di 256 bit. Entrambi gli algoritmi possono essere integrati nei QR code per garantire che i dati non siano stati alterati e che provengano da una fonte attendibile.
  Per applicazioni moderne, si raccomanda l'uso di chiavi RSA di 3.072 bit o ECDSA di 256 bit per una maggiore sicurezza.

- **Codici di Autenticazione dei Messaggi (HMAC):** HMAC (Hash-based Message Authentication Code) è una tecnica di autenticazione basata su chiavi simmetriche. Questa tecnologia assicura l'integrità dei dati, garantendo che il contenuto del QR code non sia stato modificato durante il trasferimento.
  HMAC utilizza una funzione di hash sicura, come SHA-256, e può essere una soluzione altamente sicura e performante, soprattutto in contesti aziendali dove la gestione delle chiavi è già ben strutturata.

- **Crittografia Simmetrica (AES):** L'Advanced Encryption Standard (AES) è ampiamente utilizzato per proteggere la confidenzialità dei dati.
  AES può essere implementato in modalità CBC, OFB, CFB e GCM. Quest'ultima modalità è particolarmente interessante perché fornisce sia confidenzialità sia integrità dei dati, rendendo i QR code crittografati molto più sicuri. AES supporta chiavi di 128, 192 e 256 bit, con un impatto minimo sulle prestazioni.[10]

4.5 Steganografia

La steganografia è l'arte e la scienza di nascondere informazioni all'interno di altri dati in modo tale che la presenza del messaggio segreto non sia rilevabile.
A differenza della crittografia, che si concentra sulla protezione del contenuto del messaggio attraverso tecniche di cifratura, la steganografia si occupa di nascondere l'esistenza stessa del messaggio.
L'obiettivo è rendere il messaggio invisibile o indistinguibile all'occhio umano o ai sistemi di rilevamento, spesso mascherandolo all'interno di supporti apparentemente innocui come immagini, video, o file audio.

4.5.1 Differenza tra Steganografia e Crittografia

Mentre la crittografia modifica il contenuto di un messaggio per renderlo incomprensibile a chi non possiede la chiave di decrittazione, la steganografia si concentra sull'occultamento del messaggio stesso.
In un sistema steganografico, il messaggio segreto viene nascosto all'interno di un "vettore" (come un'immagine o un file audio), che funge da contenitore e consente il trasporto del messaggio senza destare sospetti.

4.5.2 Classificazioni della Steganografia

La steganografia può essere suddivisa in due principali modelli, ciascuno con un approccio diverso alla nascondigli:

- **Steganografia Iniettiva:** Questa è la tecnica più comunemente utilizzata e consiste nell'inserire il messaggio segreto all'interno di un altro messaggio che funge da contenitore.
  Ad esempio, i bit di un'immagine digitale possono essere alterati in modo impercettibile per incorporare il messaggio segreto.
  Il contenitore originale rimane quasi inalterato, rendendo difficile per un osservatore casuale notare la presenza del messaggio nascosto.

- **Steganografia Generativa:** In questo approccio, il messaggio segreto viene utilizzato per generare un nuovo contenitore che lo nasconde nel miglior modo possibile.
  Questo tipo di steganografia è più sofisticato poiché il contenitore non è

preesistente, ma viene creato appositamente per adattarsi al messaggio che deve nascondere.
Ciò può comportare la generazione di nuove immagini, video o audio che integrano il messaggio nascosto fin dall'inizio.

4.5.3 Applicazioni e Implicazioni

La steganografia è ampiamente utilizzata in vari campi, dalla protezione della proprietà intellettuale (ad esempio, attraverso il watermarking digitale) alla comunicazione segreta.
Tuttavia, come ogni tecnologia, può essere utilizzata anche per scopi malevoli, come nascondere malware all'interno di file apparentemente innocui o per la trasmissione di informazioni in modo che sfugga ai controlli di sicurezza. [16]

4.6 Confronto delle Soluzioni e Scelta Finale

Dopo aver condotto approfondite ricerche e sperimentato diverse implementazioni, ho scelto di utilizzare i QR Code combinati con la crittografia del messaggio come metodo di sicurezza per la generazione di un logo dinamico.
Questa soluzione si è rivelata molto più semplice da gestire e implementare rispetto all'uso della steganografia, che avrebbe complicato notevolmente il progetto e allungato i tempi di sviluppo.
Di conseguenza, nel prosieguo di questa tesi, mi concentrerò sulla generazione dei QRCode, sulla crittografia del messaggio, sulla scannerizzazione del QRCode e sulla successiva decriptazione del messaggio con tecnologia Kotlin Multiplatform.

5 Realizzazione del Prototipo

In questo capitolo, verrà trattata l'implementazione di un'applicazione multipiattaforma sviluppata in Kotlin Multiplatform, che utilizza un sistema di Logo Dinamico basato su QR Code.
L'attenzione sarà posta sull'architettura sviluppata per assicurare la corretta visualizzazione e gestione dinamica dei QR Code.
Nei capitoli successivi, verranno affrontati in dettaglio i processi di generazione e scannerizzazione dei QR Code, oltre alla crittografia e decriptazione delle chiavi fornite dagli utenti.
Per la gestione delle dipendenze del prototipo è stato utilizzato gradle e i source set sviluppati sono per i dispositivi con sistema operativo Android e iOS.

5.1 Configurazione dell'ambiente di sviluppo

Per la creazione dell'applicazione multipiattaforma, è stato necessario configurare correttamente l'ambiente di sviluppo, garantendo che l'applicazione potesse essere compilata e distribuita sia su Android che su iOS.

5.1.1 Source Set

In Kotlin Multiplatform, le source set sono utilizzate per definire il codice comune e specifico per ciascuna piattaforma. Il progetto è stato organizzato con due principali source set: `AndroidMain` e `IosMain`. Questi definiscono il codice specifico per ciascuna piattaforma e permettono di utilizzare librerie native.

- `AndroidMain`: Qui sono stati inseriti i componenti specifici per Android, come le classi che interagiscono con le API Android, come la gestione del QR Code e l'integrazione con i servizi di sistema.
- `IosMain`: Questo source set include il codice nativo per iOS, consentendo all'app di utilizzare le API native di iOS per la gestione dei QR Code e altre funzionalità specifiche del sistema operativo.

5.1.2 Gestione delle dipendenze con Gradle

Per la gestione delle dipendenze e la compilazione dei source set, è stato utilizzato Gradle, uno strumento di build che permette di gestire in modo efficiente le

dipendenze del progetto e di definire le configurazioni per le varie piattaforme.
La configurazione di Gradle è stata cruciale per garantire che le librerie di terze parti utilizzate (come quelle per la gestione dei QR Code o per la dependency injection) fossero compatibili sia con Android che con iOS.
Per esempio, l'uso di una libreria per la scannerizzazione dei QR Code, come `ZXing` su Android e `AVfoundation` iOS, è stato gestito separando le dipendenze nelle rispettive source set.
A differenza di Zxing, che è una libreria di terze parti per Android sviluppata da Google per l'analisi delle immagini tramite la camera, AVfoundation non deve essere importata nel file Gradle, ma è direttamente utilizzata nel progetto in quanto è sviluppata da Apple.

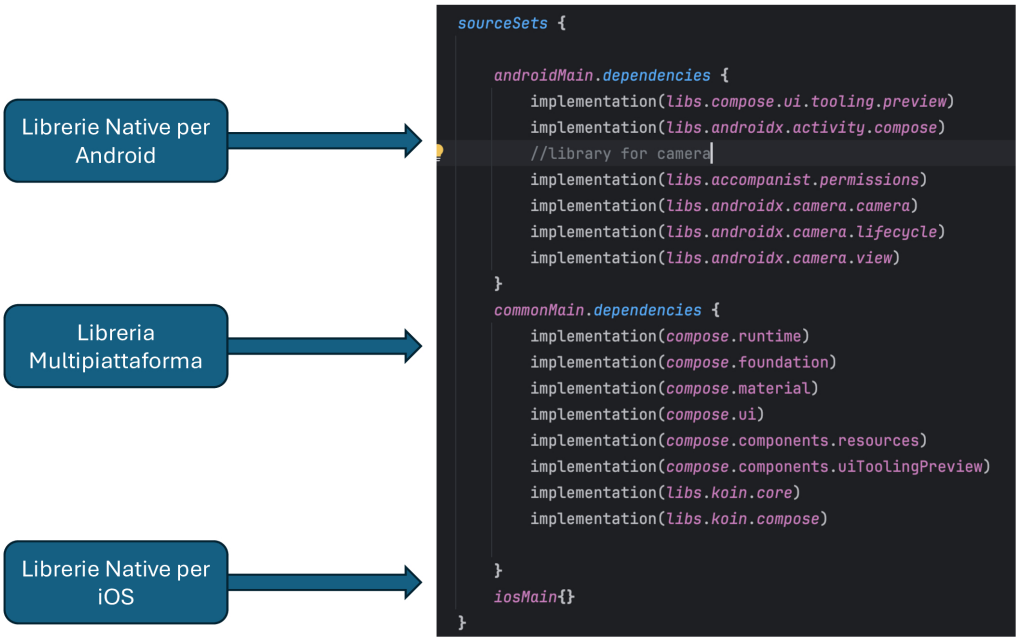

Figure 5: Source Set, importazione delle librerie

5.2 Dependency Injection (DI)

L'uso della Dependency Injection (DI) è un aspetto cruciale dello sviluppo moderno, in quanto permette di mantenere un codice modulare e facilmente testabile.
Nel progetto, è stata utilizzata la libreria **Koin** per gestire la dependency injection. Koin, come già visto in precedenza, è una libreria leggera e flessibile, che permette di definire le dipendenze tramite un DSL (Domain-Specific Language) semplice e intuitivo.
La sua integrazione nel progetto ha facilitato la gestione delle dipendenze comuni e specifiche della piattaforma, rendendo il codice più pulito e modulare.
Le dipendenze sono state dichiarate nelle cartelle `androidMain` e `iosMain`, ognuna delle quali contiene una cartella **`di`** (dependency injection), che definisce i moduli utilizzati dai vari componenti dell'app.

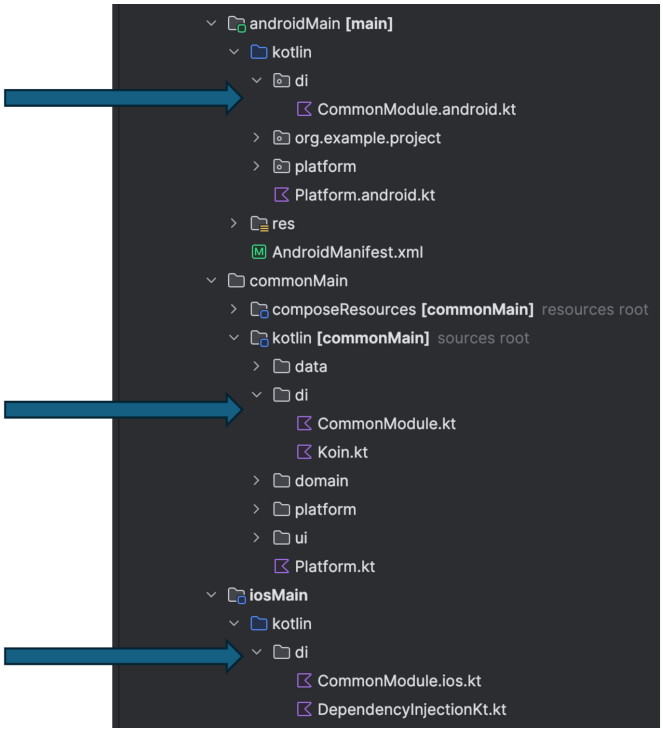

Figure 6: Dependency Injection, Common Module configurazione

5.2.1 Integrazione di Koin nel Progetto

La configurazione di Koin è un passaggio cruciale per la gestione delle dipendenze nel progetto. Nella cartella commonMain è stato definito il modulo comune, mentre nei source set specifici per piattaforma (androidMain e iosMain) sono stati definiti i moduli per le funzionalità della piattaforma..

```
1
2 import ...
3 fun MainViewController() = ComposeUIViewController { App() }
4
5 fun initKoin() {
6     startKoin{
7         modules(commonModule() + platformModule())
8     }
9 }
```

Il `commonModule` è definito all'interno della cartella `commonMain` in questo modo:

```
1 fun commonModule() = module {
2     singleOf(::GenerateViewModel)
3     singleOf(::ScannerViewModel)
4     singleOf(::KeyManagerGeneration)
5     singleOf(::KeyManagerScanner)
6     Napier.base(DebugAntilog())
7 }
8
9 expect fun platformModule(): Module
```

Il modulo `commonModule` definisce le dipendenze che possono essere utilizzate in tutto il progetto senza doverle ridefinire o passare come parametri tra i file.
Questo approccio sfrutta un importante pattern di programmazione chiamato Singleton, ampiamente riconosciuto e descritto nel libro Design Patterns: Elements of Reusable Object-Oriented Software, noto come Gang of Four.

Il **pattern Singleton** è un design pattern creazionale il cui obiettivo è assicurare che una classe abbia una sola istanza in tutto il sistema e che fornisca un punto di accesso globale a tale istanza. In pratica, significa che un componente, una volta creato, viene riutilizzato ogni volta che è necessario, evitando la duplicazione e migliorando l'efficienza. Inoltre, il Singleton è implementato in modo "lazy",

cioè l'istanza viene creata solo la prima volta che un componente ne ha bisogno; successivamente, la stessa istanza viene restituita per tutte le richieste successive.

Questo pattern è particolarmente utile in situazioni dove più file o moduli devono condividere le stesse risorse.
Nel caso del progetto, ad esempio, il `KeyManagerGeneration` – che gestisce le funzioni di generazione e gestione delle chiavi crittografate – viene creato come Singleton. Ciò significa che una sola istanza di `KeyManagerGeneration` viene condivisa tra tutti i componenti dell'applicazione che ne hanno bisogno, garantendo un utilizzo efficiente delle risorse e facilitando la coerenza dei dati.

5.3 Panoramica architetturale dell'App

L'applicazione prototipo sviluppata come base per la creazione dell'SDK utilizza l'architettura **MVVM** (Model-View-ViewModel). Questo pattern architetturale è stato scelto per separare nettamente la logica di business dall'interfaccia utente (UI), migliorando così la modularità, la manutenibilità e la testabilità dell'app.

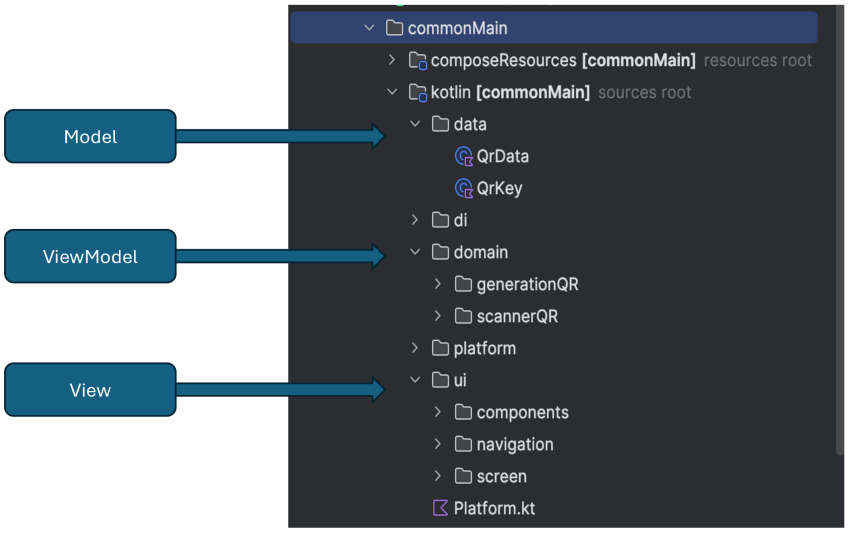

Figure 7: Architettura del progetto

5.3.1 Model

Nel pattern MVVM, il Model rappresenta la logica di business e i dati dell'applicazione.
Il Model gestisce la manipolazione dei dati e l'accesso alle fonti di dati esterne, come API o database.
In Kotlin Multiplatform (KMP), il Model è spesso implementato utilizzando data class, che sono classi progettate per contenere dati immutabili e che semplificano la gestione degli stessi, fornendo automaticamente metodi utili come copy(), toString(), equals() e hashCode().

Nel mio progetto, i file **QrData** e **QrKey** rientrano nella categoria del Model:

- ***QrData*:** Contiene le informazioni relative ai QR Code, gestendo i dati necessari per la loro generazione e manipolazione.

- ***QrKey*:** Contiene i dati relativi alla chiave fornita dall'utente. Include anche il metodo *isStillValid()*, che restituisce un valore booleano per verificare se la chiave è ancora valida.

5.3.2 View

La View è la componente dell'architettura MVVM che si occupa della visualizzazione dei dati e dell'interazione con l'utente. In Kotlin Multiplatform, la View può essere implementata utilizzando Jetpack Compose su Android o tecnologie specifiche per altre piattaforme. La View è progettata per essere reattiva ai cambiamenti di stato nel ViewModel, eseguendo automaticamente il re-rendering delle parti dell'interfaccia utente interessate dalle modifiche, senza influenzare le altre sezioni. Questo garantisce un'esperienza utente fluida e reattiva.

Nel progetto, la UI è suddivisa in diversi componenti, ciascuno con funzionalità specifiche:

- ***TransactionCard*:** Rappresenta le transazioni fittizie nella sezione riservata all'utente.
- ***AppNavigation*:** Utilizza il *NavController* per gestire le rotte e la navigazione tra le schermate.
- **Componenti di scannerizzazione e generazione dei QR Code**: Gestiscono la parte interattiva dell'app, consentendo all'utente di generare o scansionare QR Code.
- **Cartella *screen*:** Contiene le schermate principali dell'app, ciascuna progettata come un componente autonomo per una migliore organizzazione del flusso.

5.3.3 ViewModel

Il ViewModel funge da intermediario tra il Model e la View. È responsabile della logica di presentazione e della trasformazione dei dati provenienti dal Model per renderli utilizzabili dalla View.
Il ViewModel gestisce il data binding tra la View e il Model, consentendo alla View di osservare i cambiamenti dei dati e reagire di conseguenza.

Nel mio progetto, la cartella `domain` rappresenta il ViewModel. Questa cartella è suddivisa in due sottosezioni principali:

- **Crittografia delle chiavi e generazione dei QR Code:** Gestisce la logica per la generazione dei QR Code e la crittografia delle chiavi.
- **Scannerizzazione dei QR Code e decriptazione delle chiavi:** Si occupa della logica per la scannerizzazione dei QR Code e la decrittazione delle chiavi fornite dagli utenti.

Successivamente verranno approfonditi i metodi di generazione e di scannerizzazioni creati partendo dal prototipo per poi essere estrapolarli e ottenere la logic.

6 Interfaccia Utente

In questo capitolo, verrà descritta l'implementazione dell'interfaccia utente (UI) dell'applicazione multipiattaforma sviluppata in **Kotlin Multiplatform** e **Jetpack Compose Multiplatform**.
La UI è stata progettata per essere moderna, reattiva e facilmente utilizzabile su più piattaforme, con particolare attenzione alla modularità e alla manutenibilità del codice.

6.1 Panoramica dell'architettura UI

L'architettura della UI è stata costruita utilizzando **Jetpack Compose**, che consente di definire componenti UI riutilizzabili tramite funzioni annotate con **@Composable**.
Compose Multiplatform permette di sviluppare interfacce utente coerenti su più piattaforme, come Android e iOS, sfruttando i vantaggi della programmazione dichiarativa.

6.1.1 Composizione delle schermate principali

L'applicazione si compone di diverse schermate ('screen'), ciascuna dedicata a una funzionalità specifica. Di seguito vengono illustrate le principali:

Componente `App` Questo file rappresenta il punto di ingresso dell'app e utilizza la funzione **AppNavigation** per gestire la navigazione tra le varie schermate.
La composable **App** applica un tema coerente attraverso **MaterialTheme**.

```
@Composable
@Preview
fun App() {
    MaterialTheme {
        AppNavigation()
    }
}
```

Componente `Home` La schermata principale dell'applicazione, chiamata `Home`, presenta una barra di navigazione in alto, seguita da una card che mostra i dettagli dell'utente, tra cui il nome, il numero di conto e lo stato attuale.

Viene utilizzata una `LazyColumn` per caricare dinamicamente un elenco di transazioni fittizie, garantendo prestazioni ottimali anche con un gran numero di elementi. Questa schermata viene visualizzata solo nel momento in cui ilprocesso viene eseguito con successo e se viene garantito il premesso di procedere dopo aver scannerizzato il logo dinamico.

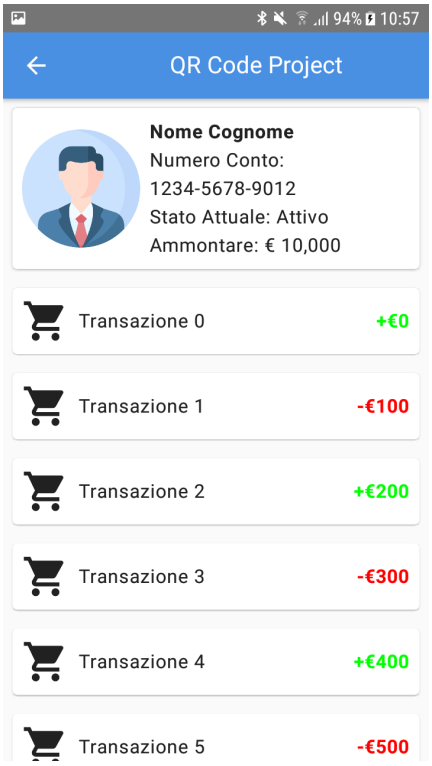

Figure 8: Schermata principale dell'applicazione

6.1.2 Generazione dei QR Code

Una parte cruciale del progetto riguarda la generazione di QR Code dinamici. Questa operazione è gestita attraverso una schermata apposita, `QrScreen`, che guida l'utente nel processo di generazione e visualizzazione dei QR Code. Il layout è strutturato verticalmente, con componenti come bottoni e immagini che facilitano l'interazione con l'utente. In questa schermata si hanno due opzoioni che si diversificano in base all'operazione che si vuole eseguire sui diversi device, le opzioni sono evidenziate nel codice da due bottoni che innescano, con un click da parte dell'utente, un processo di **scannerizzazione** oppure un processo di **generazione** dei QRcode

```
@Composable
fun QrScreen(navController: NavController) {
    MaterialTheme {
        Column(modifier = Modifier.fillMaxSize()) {
            TopAppBar(...) { ... }
            Column(...) { ... }
            Box(...) {
            Button( Column( ... ) {
                    Column( ... )
                        Button(
                            onClick = { navController.navigate("
                                scanner") },
                            ...
                        ) {
                            Text(text = "Scan QR",...)
                        }
                        Button(
                            onClick = {navController.navigate("
                                generator")},
                            ...
                        ) {
                            Text(text = "Generate QR",...)
                        }
                    }
                }
            }
        }
    }
}
```

Componente `QRCodeViewer` è responsabile della visualizzazione del QR Code generato.
Questo componente riceve i dati necessari dal **ViewModel** e li visualizza in maniera reattiva, garantendo che ogni aggiornamento dei dati venga immediatamente riflesso nell'interfaccia utente.
La gestione dello stato avviene tramite 'collectAsState()', che osserva i flussi di dati provenienti dal **ViewModel**.

```
@Composable
fun QRCodeViewer(
    generateViewModel: GenerateViewModel = koinInject(),
    keyManagerGeneration: KeyManagerGeneration = koinInject()
) {
    val qrCodes = generateViewModel.qrCodes.collectAsState()
    val startQrGeneration = keyManagerGeneration.startGeneration.
       collectAsState()

    // Logica per generare i QR Code
    LaunchedEffect(Unit) {
        // Inizializza la generazione dei QR Code
    }

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        if (startQrGeneration.value) {
            // Mostra un indicatore di caricamento mentre il QR Code
               viene generato
            QRCodeShimmer()
        } else {
            // Visualizza il QR Code generato
            Image(
                painter = rememberImagePainter(data = qrCodes.value),
                contentDescription = "Generated QR Code",
                modifier = Modifier.size(200.dp)
            )
        }
    }
}
```

Componente `QRCodeShimmer` implementa un effetto shimmer che funge da placeholder visivo durante la generazione del QR Code. Questo effetto migliora l'esperienza utente fornendo un feedback visivo mentre l'applicazione elabora i dati necessari per generare il QR Code.
L'animazione utilizza gradienti di colore e transizioni fluide per simulare un effetto di caricamento.

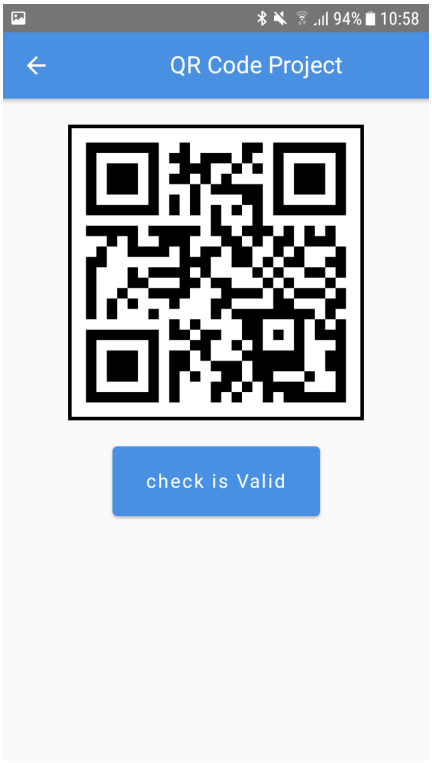

Figure 9: Schermata di generazione dei QRcode

6.2 Scannerizzazione dei QR Code

La funzionalità di scannerizzazione dei QR Code è un aspetto centrale dell'applicazione, poiché permette agli utenti di scansionare un QR Code tramite la fotocamera del dispositivo e ottenere il contenuto crittografato.
L'interfaccia e la logica associate alla scannerizzazione sono gestite attraverso due file principali: `QrScannerCompose` e `QrScannerScreen`.

Il Componente `QrScannerCompose` si occupa della gestione dell'interfaccia per la scannerizzazione dei QR Code.
La schermata iniziale mostra un pulsante per avviare la scannerizzazione e, una volta che l'utente l'ha attivata, la fotocamera viene aperta per eseguire la scansione.
La gestione dello stato avviene tramite variabili `mutableStateOf` in `Jetpack Compose`, che aggiornano dinamicamente l'interfaccia in base agli input dell'utente e agli eventi della scannerizzazione.
La gestione dei permessi per l'accesso alla fotocamera è realizzata tramite `PermissionCallback`, che richiede le autorizzazioni necessarie all'utente.

```
@Composable
fun QrScannerCompose( ... ) {
    var startBarCodeScan by remember { mutableStateOf(false) }
    var launchCamera by remember { mutableStateOf(false) }

    // Gestione dei permessi per la fotocamera
    val permissionsManager = createPermissionsManager(object :
       PermissionCallback { ... })

    if (startBarCodeScan) {
        QrScannerScreen(navController)  // Lancia lo scanner
    } else {
        Column {
            TopAppBar(...)
            Image(...)  // Icona di avvio scanner
            Button(onClick = { launchCamera = true; startBarCodeScan =
                true }) {
                Text("Scan Qr")
            }
        }
    }
}
```

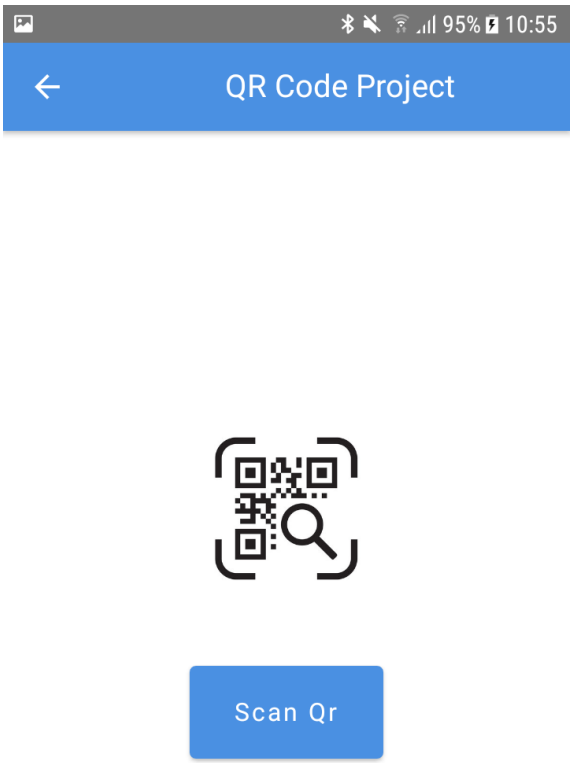

Figure 10: Avvio del processo di scannerizzazione

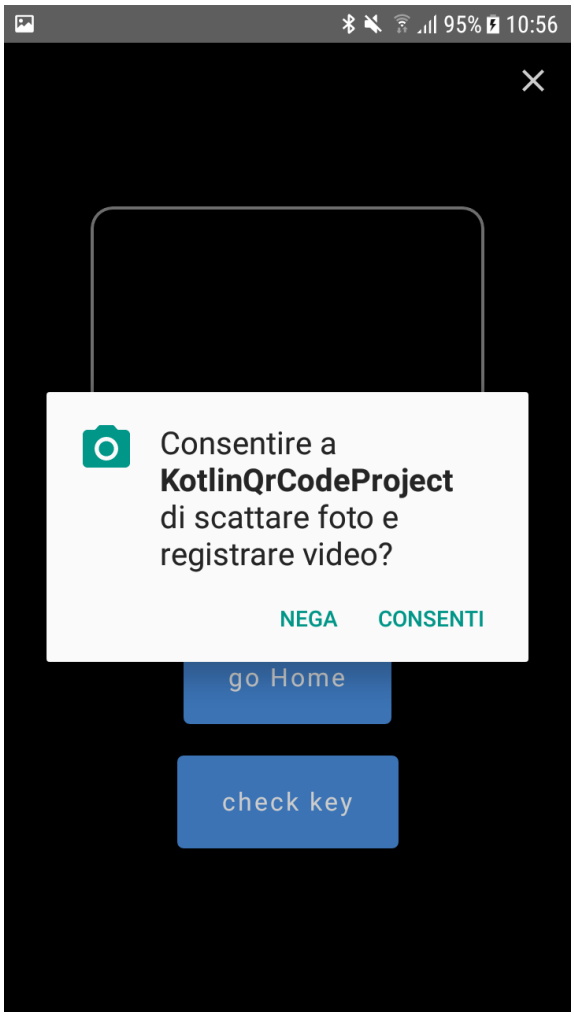

Figure 11: Richiesta dei permessi della fotocamera

Il Componente `QrScannerScreen` gestisce l'intera logica legata alla scannerizzazione dei QR Code e all'integrazione con la fotocamera del dispositivo. Utilizza un componente personalizzato chiamato `CameraPreviewWithQRCodeScanner`, che è stato sviluppato in modo specifico per ciascuna piattaforma utilizzando la funzionalità `expect/actual` di Kotlin Multiplatform.
Questo approccio consente di definire la funzionalità comune del componente in modo multipiattaforma, ma di implementarla in modo specifico per i target Android e iOS, poiché la scannerizzazione di un QR Code si basa su librerie e API diverse per ciascun sistema operativo.
L'elemento visivo principale del componente è una casella animata, che simula un'area di scansione nella quale l'utente può posizionare il QR Code da scansionare. Questa animazione migliora significativamente l'esperienza utente, fornendo un feedback visivo chiaro che guida l'utente attraverso il processo di scannerizzazione. Inoltre, un'animazione è stata aggiunta per fornire un effetto visivo continuo che indica l'attività di scansione, utilizzando `rememberInfiniteTransition` per gestire il movimento di una barra rossa lungo l'area di scannerizzazione.
La logica per la gestione dei risultati della scannerizzazione viene gestita attraverso l'uso di `LaunchedEffect`, che consente di monitorare in tempo reale lo stato del processo e reagire dinamicamente una volta che il QR Code è stato correttamente scansionato.
I dati scansionati vengono quindi collegati al `ViewModel`, permettendo alla UI di aggiornarsi automaticamente e consentendo la navigazione all'interno dell'app in base ai risultati ottenuti.

La logica legata alla gestione della fotocamera e al rendering della `CameraPreviewWithQRCodeScanner` sarà approfondita nel capitolo dedicato alla scannerizzazione dei QR Code, dove verranno trattati in dettaglio i vari passaggi per la configurazione e l'implementazione della preview della fotocamera nei diversi target della piattaforma.

```
@Composable
fun QrScannerScreen(navController: NavController, ...) {
    val scannerEnd = keyManagerScanner.endScanner.collectAsState()
    val infiniteTransition = rememberInfiniteTransition()
    val offsetY = infiniteTransition.animateFloat(...)

    Column {
        Box(modifier = Modifier.size(250.dp)) {
```

```
9               CameraPreviewWithQRCodeScanner { Napier.d("QR Scanned: $it
                    ") }
10              Box(modifier = Modifier.offset(y = offsetY.value.dp).
                    background(Color.Red)) {
11                  // Linea di scannerizzazione animata
12              }
13          }
14          Button(onClick = { navController.navigate("home") }) {
15              Text("Go Home")
16          }
17      }
18  }
```

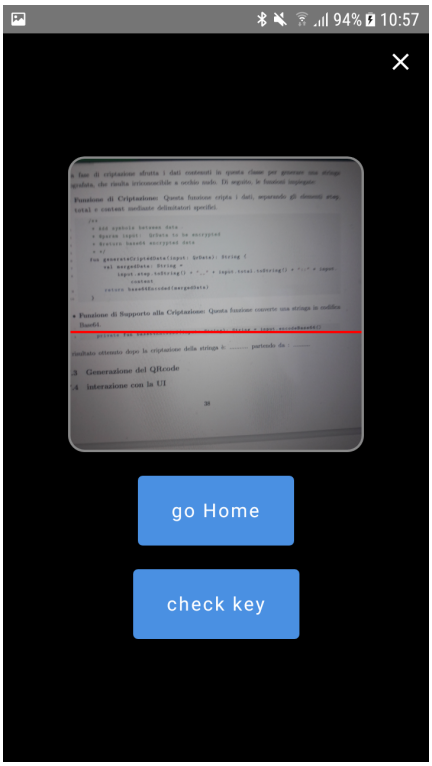

Figure 12: Schermata di scannerizzazione

6.3 Considerazioni sull'Implementazione

L'interfaccia per la scannerizzazione dei QR Code è stata progettata con un'attenzione particolare all'**usabilità** e alla **reattività** dell'app.
Le transizioni fluide tra le schermate, la gestione del flusso di permessi per la

fotocamera e l'animazione durante la scannerizzazione contribuiscono a rendere l'esperienza utente più intuitiva e piacevole.

- **Gestione dello Stato**: L'uso di `mutableStateOf` in **Jetpack Compose** permette di monitorare e aggiornare dinamicamente lo stato della UI in base all'interazione dell'utente e all'output della scannerizzazione.
- **Integrazione con la Fotocamera**: La logica per gestire la fotocamera è stata implementata in modo sicuro, richiedendo l'accesso ai permessi solo quando necessario e fornendo un feedback visivo all'utente quando il permesso è negato.
- **Animazioni**: L'effetto animato durante la scannerizzazione e il shimmer durante la generazione dei QR Code migliorano la chiarezza visiva e l'interattività dell'applicazione, aumentando la comprensione del processo da parte dell'utente.
- **Reattività della UI**: Grazie a **Jetpack Compose**, le modifiche ai dati vengono automaticamente riflesse nell'interfaccia utente, garantendo una sincronizzazione costante tra lo stato dell'applicazione e la sua rappresentazione visiva.

6.4 Conclusioni

L'implementazione della UI in **Kotlin Multiplatform** con **Jetpack Compose** ha permesso di creare un'applicazione interattiva e reattiva, che supporta in modo efficiente le piattaforme Android e iOS.
La combinazione di componenti riutilizzabili, la gestione dello stato e l'uso di transizioni fluide ha contribuito a garantire un'esperienza utente di alta qualità.
La funzionalità di **scannerizzazione dei QR Code** è stata progettata per essere intuitiva e sicura, grazie all'uso di animazioni visive, alla gestione dei permessi della fotocamera e all'integrazione del processo di scannerizzazione con la logica di business del progetto.
Inoltre, la generazione dei QR Code è stata ottimizzata con l'uso di placeholder animati tramite `QRCodeShimmer`, migliorando ulteriormente l'interattività e la percezione di fluidità dell'applicazione.

7 Generazione dei QRcode

7.1 Introduzione

La generazione dei QR Code è una parte cruciale del sistema di sicurezza basato sul Logo Dinamico descritto in questo progetto. In questa fase, il sistema deve generare un QR Code crittografato a partire da una chiave fornita dall'utente.
Inizialmente, la generazione del QR Code avveniva attraverso l'uso di una libreria di terze parti, garantendo così compatibilità multipiattaforma. Durante lo sviluppo dell'SDK, però, è stata implementata una soluzione interna che evita la dipendenza da librerie esterne, integrando codici specifici per la generazione sui diversi target.

7.2 Gestione della Chiave

La generazione dei QR Code richiede una chiave fornita dall'utente immediatamente prima dell'avvio del processo di generazione.
Durante la fase di progettazione, è stato deciso che la chiave sarebbe stata una **stringa**, con una lunghezza minima consigliata di 10 caratteri e una massima di 50 caratteri.

7.2.1 Importanza della Lunghezza della Chiave

La **lunghezza della chiave** influisce significativamente sul processo di generazione dei QR Code.
Se la chiave è troppo corta, il numero di QR Code generati sarà limitato, mentre una chiave eccessivamente lunga *potrebbe* rendere impossibile scansionare l'intero set di QR Code prima che la validità della chiave scada.

7.2.2 Il Concetto di Logo Dinamico

Per capire meglio la relazione tra la chiave e i QR Code, è utile rifarsi al concetto di **Logo Dinamico**.
Nel contesto di questo progetto, un "Logo Dinamico" è definito come *"un'immagine che varia nel tempo"*, specificamente un QR Code che cambia. Per soddisfare questo requisito, è stata concepita l'idea di un set di QR Code generati da un **insieme di chiavi**.

7.2.3 Suddivisione della Chiave in Segmenti

Per aumentare il livello di sicurezza del sistema, la chiave viene suddivisa in segmenti, denominati `Chunks`. Questi chunks sono stringhe di lunghezza variabile tra 5 e 8 caratteri, create dalla scomposizione della chiave originale.
La strategia adottata consente una migliore gestione e sicurezza nella generazione dei QR Code.
L'idea di suddividere la chiave in chunks ha portato allo sviluppo di un codice specifico che gestisce sia l'interazione con l'interfaccia utente, per il passaggio della chiave come parametro, sia con il manager dei QR Code per la loro effettiva generazione.

7.2.4 Ruolo del `KeyManagerGeneration`

La gestione della chiave è affidata a un componente importante del ViewModel, il `KeyManagerGeneration`.
Questa componente prepara la chiave prima che venga passata al manager dei QR Code. Le sue responsabilità includono la generazione di una lista di chunks a partire dalla chiave originale e la crittografia di questi dati prima della generazione del QR Code.

7.2.5 Funzioni per la preparazione del contenuto dei QR Code del `KeyManagerGeneration`

- **Funzione di generazione della chiave**: Crea l'oggetto QrKey passando i valori forniti dall'untente e delega la chiave (in formato stringa) alla funzione che genera i chunks.

```
1    fun generateKey(qrContent: String, timeValid: Int) {
2        val timeSource = TimeSource.Monotonic
3        key = QrKey(qrContent, timeSource.markNow(), timeValid)
4
5        generateChunks()
6        _startGeneration.value = true
7    }
```

- **Funzione di generazione dei chunks:** Suddivide la chiave originale in segmenti più piccoli e gestibili, pronti per essere criptati e convertiti in QR Code.

```
1     private fun generateChunks() {
2         key?.let {
3             val newChunks = splitString(it.key, (5..8).random())
4             listChucks.addAll(newChunks)
5         }
6     }
```

Funzione di supporto che ***"splitta"***[4] la stringa in chunks:

```
1
2     private fun splitString(originalString: String, chunkSize:
          Int): List<String> {
3         val chunks = mutableListOf<String>()
4         var startIndex = 0
5
6         while (startIndex < originalString.length) {
7             val endIndex = startIndex + chunkSize
8             if (endIndex <= originalString.length) {
9                 chunks.add(originalString.substring(startIndex,
                      endIndex))
10            } else {
11                chunks.add(originalString.substring(startIndex))
12            }
13            startIndex = endIndex
14        }
15
16        return chunks
17    }
```

7.2.6 Funzioni per la Criptazione della Chiave

Nel `KeyManagerGenerator`, le funzioni dedicate alla criptazione non operano direttamente sulla chiave, ma su una **data class** denominata `QrData`.

La Classe `QrData` riveste un ruolo cruciale nel progetto, poiché contiene tutte le informazioni necessarie per facilitare lo scambio di dati e garantire la corretta esecuzione del processo. La struttura della classe è la seguente:

```
1 data class QrData(val step: Int, val total: Int, val content: String,
      val idKey: Int)
```

[4] suddividere secondo un criterio

La classe `QrData` funge da rappresentazione interna del QR Code prima della criptazione. Le sue proprietà sono:

- `step`: Un intero che indica il numero del chunk corrente, essenziale per riordinare o identificare segmenti mancanti della chiave.
- `total`: Il numero totale di chunks generati dalla chiave.
- `content`: Il contenuto effettivo del segmento della chiave, o chunk.
- `idKey`: Un identificativo unico per la chiave in questione.

La fase di criptazione sfrutta i dati contenuti in questa classe per generare una stringa crittografata, che risulta incomprensibile all'occhio umano.
Di seguito, le funzioni impiegate:

- **Funzione di Criptazione:** Questa funzione cripta i dati, separando gli elementi `step`, `total` e `content` mediante simboli che fungono da separatori tra i dati.

```
1    /**
2     * Add symbols between data
3     * @param input:  QrData to be encrypted
4     * @return base64 encrypted data
5     * */
6    fun generateCriptedData(input: QrData): String {
7        val mergedData: String =
8            input.step.toString() + "__" + input.total.toString()
                 + "::" + input.content
9        return base64Encoded(mergedData)
10   }
```

- **Funzione di Supporto alla Criptazione:** Questa funzione converte una stringa in codifica Base64.

```
1     private fun base64Encoded(input: String): String = input.
          encodeBase64()
```

7.3 Generazione dei QR Code

La generazione dei QR Code rappresenta una fase essenziale nel flusso operativo dell'applicazione.
Il processo di **ideazione** ha indagato approfonditamente le modalità per rendere dinamico un QR Code, esplorando le potenzialità e i limiti del linguaggio Kotlin.
Si è osservato che, nonostante la logica di generazione e la visualizzazione dei QR Code abbiano una forte coesione, è cruciale mantenerle separate.
Analisi di benchmark sulle prestazioni dei dispositivi hanno evidenziato che su modelli di alta gamma, le prestazioni dell'interfaccia utente possono non essere sincronizzate con quelle della generazione. Invece, per dispositivi di bassa gamma la generazione risultava molto meno performante dell'interfaccia. Di conseguenza, è stato deciso di eseguire questi due processi in **parallelo su thread distinti** per minimizzare il divario tra dispositivi di fascia alta e bassa, assicurando che la lentezza nella generazione non influenzi la visualizzazione.

L'iniziativa di implementazione ha pertanto privilegiato la parallelizzazione dei processi, integrando strutture dati appropriate per gestire e visualizzare i QR Code generati in momenti diversi rispetto alla loro produzione.

Il processo di generazione inizia con il `KeyManagerGeneration`, che prepara la chiave e sviluppa una lista di **chunks** da includere nei QR Code durante la generazione. Questa lista è successivamente passata al `GenerationQrManager`, il componente incaricato della gestione dell'intero processo, in attesa di essere attivato.

7.3.1 Il GenerationQrManager

Il `GenerationQrManager` gioca un ruolo fondamentale nella gestione del processo di generazione dei QR Code. Contiene tutte le funzioni essenziali per interagire con l'interfaccia utente e per supportare operazioni secondarie.
Situato nello strato logico del Business, identificato come **ViewModel**, il `GenerationQrManager` si interfaccia con lo strato dell'interfaccia utente tramite il `QRCodeViewer`, che innesca il processo di generazione dei QR Code tramite un pulsante azionato dall'utente.

Come menzionato, la generazione dei QR Code in questo prototipo è affidata a una libreria esterna denominata ***"QRkit Compose Multiplatform"***[11], importata tramite Gradle.

Questa libreria offre una soluzione grafica multipiattaforma e facilita la generazione e la scansione dei QR Code.
Nel prototipo, il suo utilizzo si limita ad una funzione di generazione situata nel `GenerationQrManager`, permettendo così di risparmiare la necessità di scrivere codice specifico per ogni piattaforma.

Una tra le funzioni principali all'interno della componente è: `generateQrCode()`, questa funzione genera i QR Code partendo da una lista di input che contiene le informazioni che devono essere inserite all'interno.
La lista chiamata `listInputQr` è stata popolata prima dell'innesco della generazione all'interno di una funzione che prende come parametri la **lista di chunks** appartenente al `KeyManagerGeneration`, ecco la funzione di preparazione:

```
fun generateListInputQr(listChucks: MutableList<String>) {
        for (i in listChucks.indices) {
            listInputQr.add(QrData(i, listChucks.lastIndex, listChucks
                [i], i))
        }
    }
```

Il processo di generazione dei QR Code inizia con la preparazione dei dati necessari. Questo avviene attraverso la struttura dati `QrData`, che viene inizializzata con i valori corrispondenti a ciascun chunk di dati. Questi dati saranno poi il contenuto del QR Code generato.
Ogni `QrData` viene aggiunto a una lista che funge da coda temporanea per la generazione dei QR Code.

Il `GenerationQrManager` rimane in attesa dell'azione dell'utente per avviare il processo di generazione. Una volta che l'utente attiva il processo, la funzione `generateQrCode()` viene eseguita.
Durante questa fase, viene utilizzata la classe **'ImageBitmap'**, fornita da Kotlin, per gestire le immagini dei QR Code generati.

ImageBitmap e la sua utilità: `ImageBitmap` è una classe Kotlin usata per rappresentare immagini **bitmap**[5] in modo efficiente e ottimizzato per le operazioni di rendering. Questa classe è particolarmente utile in applicazioni che richiedono la manipolazione intensiva di immagini, poiché permette operazioni di trasformazione

[5]Una bitmap: è una matrice di bit che specificano il colore di ogni pixel in una matrice rettangolare di pixel

e composizione senza perdite di performance significative.
Nel contesto della generazione dei QR Code, 'ImageBitmap' è utilizzata per memorizzare temporaneamente ogni QR Code generato prima di essere salvato o visualizzato.
Questo approccio assicura che le operazioni di generazione e visualizzazione dei QR Code siano efficienti e meno suscettibili a rallentamenti dovuti alla gestione delle risorse di sistema.[4]

```
suspend fun generateQrCode() {
    var tmpQrData: ImageBitmap? = null
    coroutineScope {
        launch(Dispatchers.Default) {
            startTimeGeneration = timeSource.markNow()
            for (element in listInputQr) {
                generateQrCode(
                    keyManagerGeneration.generateCriptedData(element),
                    onSuccess = { _, qrCode ->
                        tmpQrData = qrCode
                    },
                    onFailure = {
                        Napier.d("TEST : FAILS GENERATION QR")
                    }
                )
                generateQrCodelist(tmpQrData)
                keyManagerGeneration.startValidationTime(element)
            }
        }
    }
}
```

Nel codice si può notare che è presente una funzione di supporto che viene chiamata dopo che il QR Code è stato generato, si tratta di una funzione che concatena, in una lista ordinata, i QR Code generati.
Questo si rivelerà utile nel momento in cui la sequenza viene visualizzata ripetutamente, infatti, permette al dispositivo scannerizzatore di catturare i QR Code che posso essere sfuggiti durante la prima fase di visualizzazione.
Ecco la funzione di supporto che concatena la lista:

```
  private suspend fun generateQrCodelist(tmpQrData: ImageBitmap?) {
        tmpQrData?.let {
            mutex.withLock {
```

```
                listQrImg.add(it)
            }
        }
        startVisualization = listQrImg.size > 2
    }
```

Il metodo gestisce l'aggiunta sicura di nuove immagini QR alla coda, preparando il sistema per la visualizzazione sequenziale dei QR Code.
La visualizzazione inizia quando ci sono almeno tre QR Code nella coda, come indicato dalla condizione

```
startVisualization = listQrImg.size > 2.
```

```
    suspend fun generateVisualizationUI() {
        coroutineScope {
            launch(Dispatchers.Default) {
                while (true) {
                    if (startVisualization) { //innesco del processo
                        listQrImg.size
                        for (index in listInputQr.indices) {
                            if (startTimeGeneration.elapsedNow() >=
                                durationLimit) {
                                return@launch
                            }
                            mutex.withLock {
                                _qrCodes.value = listQrImg[index]
                            }
                            delay(600)
                        }
                    }
                    delay(300)
                }
            }
        }
    }
```

La funzione gestisce il ciclo di visualizzazione dei QR Code, nel momento in cui la variabile `startVisualization` diventa `true` il processo di visualizzazione ha inizio.
All'interno del brocco del ciclo di esecuzione viene verificato se la chiave è ancora valida, se lo è allora il processo sceglie un QR Code dalla lista e lo mostra, altrimenti, termina il processo.

Inoltre, sono inseriti dei ritardi nell'esecuzione del codice per assicurare che ogni QR Code venga esposto per un tempo sufficiente per essere catturato da un dispositivo scannerizzatore, mantenendo così l'efficienza del processo di decodifica.

7.4 interazione con la UI

Di seguito è mostrato un diagramma di sequenza che cerca di illustrare il flusso tra i lo strato della View quello del ViewModel in cui interagiscono il `KeyManagerGeneration` e `GenerationQrManager`

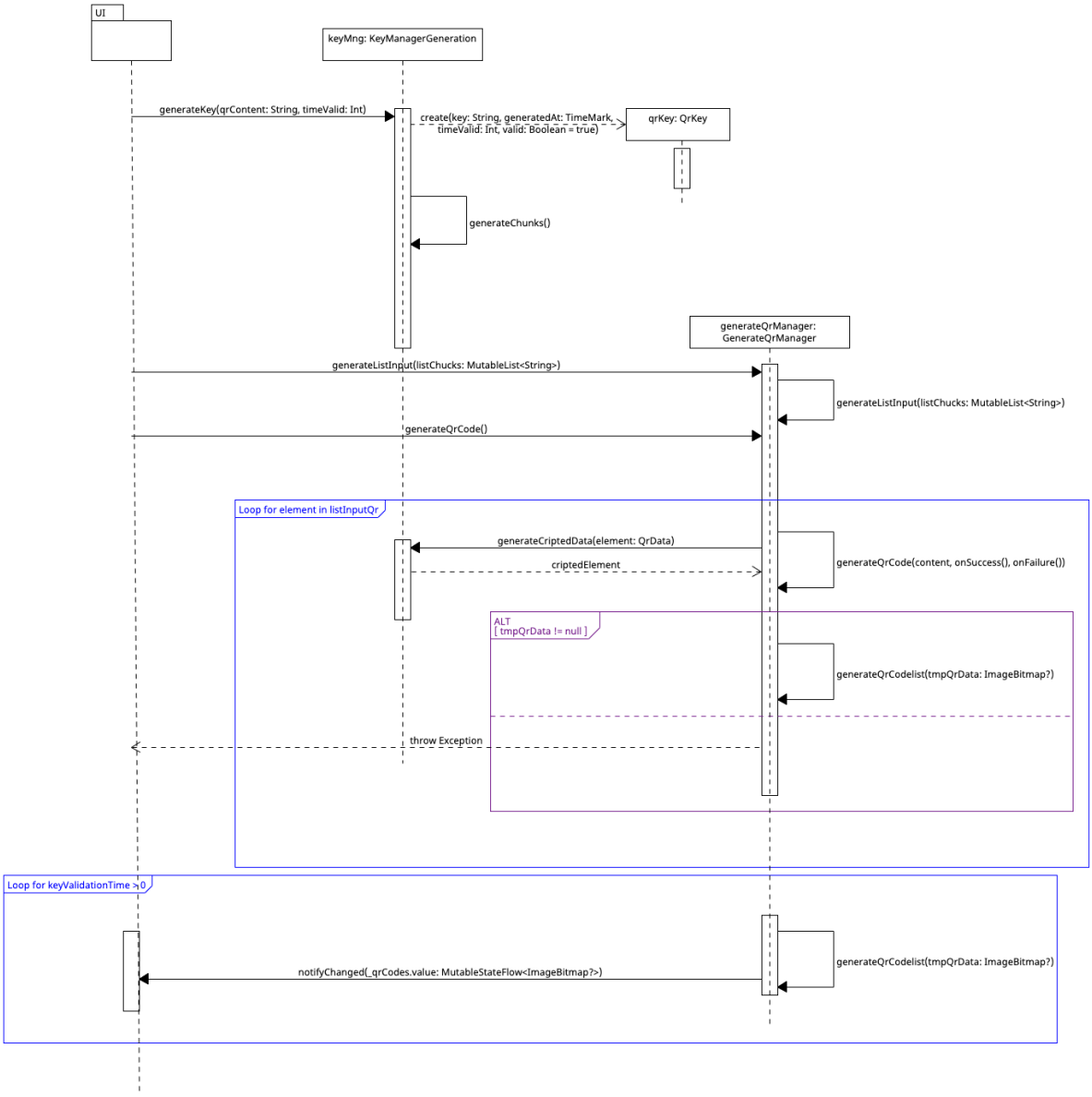

Figure 13: diagramma di sequenza della Generazione

8 Scannerizzazione dei QRcode

8.1 introduzione

Questo capitolo descrive il processo di scannerizzazione dei QR Code all'interno del prototipo.
Durante la fase di ideazione, è stata analizzata la coesione tra il generatore di QR Code e lo scanner, con l'obiettivo di minimizzare il divario funzionale tra i due componenti.
L'analisi ha comportato la definizione di requisiti specifici per lo scanner, affinché potesse elaborare rapidamente i QR Code, estrarne il contenuto, salvarlo e poi ripetere il processo fino alla completa acquisizione di tutti i QR Code generati.

Nel corso di questa fase progettuale, è emersa la necessità di includere nei QR Code il numero di sequenza dei chunks che compongono la chiave, così da tracciare ogni QR Code scannerizzato, assicurando che ogni elemento del set sia correttamente identificato e processato.

Per il processo di scannerizzazione non sono state usate librerie multipiattaforma di terze parti in quanto non permettevano la totale gestione del QR Code scannerizzato, per questo motivo è stato necessario implementare metodi specifici per ogni target.

8.2 Gestione dei permessi

Nelle applicazioni mobile, e in altre situazioni che coinvolgono l'uso delle funzioni hardware del dispositivo come la fotocamera, è fondamentale richiedere i permessi all'utente. Questa necessità deriva da vari fattori critici:

- **Privacy e Sicurezza:** La fotocamera può catturare immagini e video contenenti informazioni personali. Pertanto, sistemi operativi come Android o iOS richiedono che le app ottengano il consenso esplicito dell'utente prima di accedere alla fotocamera.

- **Trasparenza dell'Uso:** Informare l'utente sui componenti del dispositivo che l'app intende utilizzare migliora la trasparenza e permette agli utenti di essere consapevoli delle interazioni dell'app con il loro dispositivo.

- **Funzionalità dell'Applicazione:** L'accesso alla fotocamera è essenziale per le app che scannerizzano i QR Code. Senza questo permesso, l'app non può funzionare correttamente.
- **Conformità con le Politiche dell'App Store:** Le piattaforme di distribuzione di app impongono la richiesta di permessi espliciti per garantire la conformità con le normative sulla privacy e protezione dei dati.
- **Esperienza Utente Migliorata:** Ottenere i permessi all'inizio assicura un'esperienza utente fluida e senza interruzioni durante l'uso dell'app.

8.2.1 Codice per la gestione dei permessi

Il codice per la gestione dei permessi è implementato utilizzando la tecnologia **expect/actual**.
La necessità di implementare i permessi specifici per le piattaforme Android e iOS deriva dalle differenze strutturali e di gestione della sicurezza che esistono tra questi due sistemi operativi.
Ogni piattaforma ha il proprio modo di gestire le risorse sensibili (come la fotocamera), e quindi richiede un approccio personalizzato per assicurare che la tua applicazione rispetti le norme di sicurezza e privacy.

Differenze nell'architettura di sicurezza:

- **Android:** Android utilizza un modello basato sui manifesti per dichiarare i permessi richiesti da un'applicazione. I permessi devono essere richiesti dinamicamente all'interno dell'app se sono classificati come "pericolosi" (ad esempio, l'accesso alla fotocamera). Android fornisce una serie di API, come `rememberPermissionState` e `Manifest.permission`, per gestire queste richieste di permessi.
- **iOS:** iOS gestisce i permessi attraverso un sistema di autorizzazioni, in cui gli utenti concedono l'accesso alle risorse (come la fotocamera) al momento dell'uso.
  Per controllare e richiedere i permessi, iOS utilizza API come `AVCaptureDevice.authorizationStatusForMediaType` e `requestAccessForMediaType`.

L'approccio su iOS è leggermente diverso da Android in quanto richiede l'uso di un dialogo che viene visualizzato automaticamente quando l'app tenta di accedere alla risorsa per la prima volta.

Comportamento diverso del sistema operativo:

- **Android:** Se un utente ha negato un permesso in precedenza, Android permette di mostrare una spiegazione all'utente (`shouldShowRationale`) prima di ripresentare la richiesta del permesso.
- **iOS:** In iOS, se un utente nega l'accesso a una risorsa come la fotocamera, l'app non può chiedere nuovamente il permesso. L'utente deve essere indirizzato manualmente alle impostazioni del dispositivo per consentire l'accesso.

8.3 Preview della Camera e il Riconoscimento dei QR Code

In seguito alla gestione dei permessi per garantire l'accesso alla camera, in sicurezza, del dispositivo, il passo successivo si concentra sull'apertura della camera e sul riconoscimento dei QR Code.

8.3.1 Preview della camera

Il termine "preview della camera" si riferisce alla visualizzazione in tempo reale dell'immagine catturata dalla fotocamera di un dispositivo, mostrata sullo schermo prima di scattare una foto o registrare un video.
Questa funzionalità è particolarmente rilevante in molti contesti tecnologici, inclusi i dispositivi mobili, le applicazioni di videocamera di sicurezza e i sistemi integrati. Nel contesto della scansione dei QR Code, la preview della camera permette agli utenti di vedere in tempo reale quale parte dell'ambiente stanno inquadrando.
Questo aiuta l'utente a posizionare correttamente il QR Code all'interno del campo visivo della fotocamera per una scansione efficace.

Come per la gestione dei permessi, la differenza tra: i sistemi operativi, l'hardware, le API e i metodi di gestione della camera, anche per la preview della camera è stato implementato il codice sui target specifici utilizzando librerie diverse. Per iOS è stata utilizzata `AVFoundation`[3], mentre per Android `Camera Kit`[6].

8.4 Codice per Scannerizzare QR su Android

Questa sezione descrive l'implementazione di un sistema di analisi di immagini per la scansione di QR Code all'interno di un'applicazione mobile che utilizza la fotocamera del dispositivo Android.
L'analizzatore di QR Code è progettato per essere efficiente e reattivo, utilizzando metodi avanzati di elaborazione delle immagini e di decodifica.

8.4.1 Gestione dei Permessi della Fotocamera

Il sistema necessita dell'accesso alla fotocamera del dispositivo per funzionare.
Se il permesso per la fotocamera non è stato precedentemente concesso dall'utente, viene richiesto automaticamente quando si tenta di avviare la preview della fotocamera, così da garantire che l'applicazione operi nel rispetto delle normative sulla privacy e che l'utente sia consapevole delle funzionalità hardware utilizzate dall'app.

8.4.2 Configurazione della Preview della Fotocamera:

L'implementazione della preview della fotocamera avviene attraverso l'uso di **PreviewView**, una vista dedicata che mostra in tempo reale ciò che la fotocamera vede:

```
val preview = Preview.Builder().build().also {
    it.setSurfaceProvider(previewView.surfaceProvider)
}
```

Questo componente è essenziale per la scansione dei QR Code, in quanto permette all'utente di inquadrare correttamente il codice da scansionare.
La configurazione della camera e l'analisi delle immagini sono gestite tramite `ProcessCameraProvider`, che lega le funzioni della fotocamera al ciclo di vita dell'app, ottimizzando le risorse e migliorando le prestazioni.

8.4.3 Analisi dei QR Code

La scansione e l'analisi dei QR Code sono gestite dalla classe `QrCodeAnalyzer` che implementa l'interfaccia `"ImageAnalysis.Analyzer"` di **CameraX** permettendo di impostare un metodo di analisi delle immagini. Questa classe processa le immagini acquisite dalla fotocamera in tempo reale e rileva i dati contenuti nei QR Code:

```
it.setAnalyzer(
    ContextCompat.getMainExecutor(ctx),
    QRCodeAnalyzer { result ->
        onQRCodeDetected(result)
    })
```

L'analizzatore è fondamentale per la funzionalità di scansione, in quanto permette di estrarre e utilizzare immediatamente le informazioni dal QR Code, facilitando operazioni come l'accesso a contenuti specifici o la verifica di autenticità.

Variabili e Proprietà: ci sono diverse variabili significative all'interno di questa classe, eccone alcune:

- **isActive**: è una variabile booleana che controlla se l'analisi dei QR Code è attiva o meno. Se false, l'analisi si interrompe.
  Il valore di questa variabile è modificato da funzioni che interagiscono con l'esterno della classe, queste fuinzioni sono: `terminateScanProcess` e `startScanProcess`
- **keyManagerScanner**: è una istanza della classe `KeyManagerScanner` che gestisce la decriptazione e ricomposizione della chiave.
- **formatPermitted**: è una lista di formati di immagine permessi che l'analizzatore può processare. In questo caso, sono inclusi diversi formati YUV, che sono comuni per le immagini della fotocamera.
- **qrCodeReader**: Istanza di MultiFormatReader da ZXing, configurata per riconoscere i QR Code attraverso un set di hint.

Analisi dell'immagine: la classe è specializzata nell'analisi delle immagini e di ottenere, nel caso in cui sia presente, il contenuto del QR Code. il metodo principale che svolge questo compito è : `analyze`. Questa funzione sovrascive quella dell'interfaccia `ImageAnalysis.Analyzer` che prende come parametro una `ImageProxy`, la quale sarà l'immagine scansionata.

Il primo passaggio è una condizione che controlla se la scansione è ancora attiva verificando il valore della variabile `isActive`. Se la condizione è soddisfatta, prima di procedere con l'elaborazione dell'immagine, il codice verifica se il formato dell'immagine catturata (`imageProxy.format`) è uno dei formati permessi

(`formatPermitted`), che includono vari formati YUV tipicamente usati per video e fotocamere, questo controllo assicura che il processo di analisi proceda solo se l'immagine è in un formato compatibile, ottimizzando l'uso delle risorse e prevenendo errori durante la decodifica.

Successivamente, se il formato è corretto, il codice procede con l'estrazione dei dati grezzi dell'immagine:

```
val byteBuffer = imageProxy.planes[0].buffer
val imageData = ByteArray(byteBuffer.capacity())
byteBuffer.get(imageData)
```

Qui, i dati grezzi dell'immagine vengono estratti dal primo piano (`planes[0]`) del `ImageProxy`.

Il buffer di questo piano contiene i dati YUV dell'immagine, che sono poi copiati in un array di byte (`imageData`).

la terza fase crea la "**Sorgente di Luminanza**" che è necessaria per la binarizzazione dell'immagine prima della decodifica:

```
val source = PlanarYUVLuminanceSource(
    imageData, imageProxy.width, imageProxy.height, 0, 0, imageProxy.
       width, imageProxy.height, false
)
```

`PlanarYUVLuminanceSource` è una classe di **ZXing** che converte i dati **YUV** in una mappa di luminanza, la quale rappresenta la luminosità di ogni pixel, essenziale per il rilevamento dei codici a barre.

L'immagine viene quindi binarizzata usando `HybridBinarizer`, si tratta di una classe che implementa un algoritmo di sogliatura adattiva per convertire l'immagine in bianco e nero (**binario**), questa classe prepara l'immagine per la decodifica.

Il Qr Code viene decodificato:

```
val result = qrCodeReader.decodeWithState(binaryBitmap)
```

il suo contenuto verrà utilizzato per operazioni future di decriptazione della chiave.

Infine, ogni **ImageProxy** deve essere chiuso per liberare risorse, garantendo che la camera possa continuare a catturare nuove immagini senza esaurire la memoria.

8.5 Codice per Scannerizzare QR su iOS

Per quanto riguarda iOS, il codice implementato rispetta la logica sviluppata per la piattaforma Android, ma con API diverse che si basano sulla libreria **AVFoundation**.
Anche in questo caso, sono state implementate due classi, quella che gestisce l'interfaccia dello scanner, implementando un metodo di apertura della camera, dopo aver verificato i permessi, e una classe che sviluppa la logica di buisness per l'analisi dell'immagine.

8.5.1 Definizioni generali

Nella classe `CameraPreviewWithQRCodeScanner` è definita globalmente una "`sealed interface`", questa interfaccia fornisce gli stati di autorizzazione all'accesso della fotocamera, nel caso in cui non venga fornito precedentemente, i permessi vengono richiesti e possono essere classificati come: *"Undefined"*, *"Denied"* o *"Authorized"*.
Anche un'altra variabile è definita globalmente, si tratta di `deviceTypes`, una lista di tipi di fotocamera supportati dal sistema, utilizzati per configurare le sessioni di acquisizione.

8.5.2 Accesso alla camera

La classe `CameraPreviewWithQRCodeScanne` contiene l'interfaccia grafica per mostrare la camera nel caso in cui i permessi siano stati concessi dall'utente.
Se una fotocamera è disponibile, viene chiamata la classe `QrCodeAnalyzer` per analizzare i frame della fotocamera alla ricerca di codici QR.

8.5.3 QrCodeAnalyzer:

Nell'ambito dello sviluppo di applicazioni mobili che richiedono funzionalità di riconoscimento in tempo reale di codici QR, è stata implementata la funzione `QrCodeAnalyzer`.
Questa funzione sfrutta l'ecosistema iOS per catturare e processare immagini provenienti dalla fotocamera, identificando e analizzando i codici QR tramite tecniche asincrone e componenti dichiarativi di interfaccia utente.

8.5.4 Struttura e Funzionamento della Funzione

`QrCodeAnalyzer` è una funzione di tipo `@Composable` basata su **Jetpack Compose**, il framework dichiarativo per il design dell'interfaccia, e utilizza **AVFoundation**, la libreria multimediale di iOS, per interagire con l'hardware della fotocamera.
Questa combinazione permette di gestire in modo efficiente l'acquisizione delle immagini, la visualizzazione in tempo reale e il riconoscimento dei codici QR.
Per le operazioni asincrone, come l'elaborazione dei dati catturati dalla fotocamera, la funzione utilizza le **Coroutines** di Kotlin, garantendo che le operazioni complesse avvengano in background senza bloccare l'interfaccia utente.

Il processo di cattura inizia con la configurazione di una sessione di acquisizione tramite `AVCaptureSession`. Questa sessione è composta da un input, rappresentato dalla fotocamera del dispositivo, e da due output principali: uno per la cattura delle immagini tramite `AVCapturePhotoOutput` e uno per il riconoscimento dei metadati, in particolare i codici QR, tramite `AVCaptureMetadataOutput`.
Una volta che un codice QR viene rilevato, i suoi dati vengono estratti e passati a una funzione di callback, che gestisce il contenuto del codice.

8.5.5 Elaborazione del Codice e Sicurezza

Una caratteristica fondamentale della funzione è l'integrazione con il componente **KeyManagerScanner**, che si occupa di elaborare e decriptare il codice QR.
Il processo è particolarmente rilevante in applicazioni che richiedono elevati standard di sicurezza, come sistemi di autenticazione o transazioni crittografate. Il codice QR rilevato viene decomposto e la chiave contenuta al suo interno viene ricomposta e decriptata per permettere l'accesso sicuro alle risorse o ai servizi desiderati.

8.5.6 Gestione dell'Orientamento e Interfaccia Utente

Un ulteriore elemento di innovazione della funzione è la gestione dinamica dell'orientamento del dispositivo.
Tramite l'utilizzo di un listener per le notifiche di cambiamento dell'orientamento del dispositivo (`UIDeviceOrientationDidChangeNotification`), `QrCodeAnalyzer` è in grado di adattare automaticamente l'orientamento della fotocamera e della preview dell'interfaccia, garantendo una corretta visualizzazione del flusso video anche durante la rotazione del dispositivo.

Questo migliora l'esperienza dell'utente, rendendo l'interfaccia più fluida e reattiva.

8.5.7 Esecuzione Asincrona e Ottimizzazione delle Performance

Per garantire un'esecuzione efficiente e senza interruzioni, la sessione di acquisizione e l'analisi del codice QR avvengono in modo asincrono utilizzando coroutines su un thread separato (`GlobalScope.launch` su **Dispatchers.IO**).
Tale approccio evita che l'interfaccia utente subisca rallentamenti durante il processo di acquisizione e decodifica dei codici QR, migliorando così le performance complessive dell'applicazione.

8.6 Gestione della Chiave

Dopo la fase di scannerizzazione e acquisizione del contenuto dei QR Code, i dati vengono passati a una componente dedicata alla gestione della **decriptazione** e **ricomposizione** della chiave. Questa componente fondamentale è rappresentata dalla classe `KeyManagerScanner`, la quale si occupa di processare i segmenti di chiave contenuti nei QR Code e ricomporre la chiave completa per l'utilizzo successivo.

8.6.1 Ruolo del `KeyManagerScanner`

La classe `KeyManagerScanner` è responsabile della gestione della chiave durante e dopo il processo di scannerizzazione dei codici QR.
L'implementazione di questa classe è stata realizzata in modo cross-platform, consentendo di utilizzare la stessa logica sia su Android che su iOS senza bisogno di sviluppare codice specifico per ciascuna piattaforma. Questa logica condivisa è collocata nel modulo `commonMain`, che consente un controllo centralizzato del processo di ricostruzione della chiave.

Seguendo il pattern **Singleton**, `KeyManagerScanner` viene iniettato nei vari componenti dell'applicazione tramite **dependency injection**, garantendo che una singola istanza della classe venga utilizzata durante l'intero ciclo di vita dell'applicazione.

La prima occorrenza di `KeyManagerScanner` si trova all'interno della funzione di analisi dei QR Code, che viene invocata nelle piattaforme specifiche (Android e iOS). Una volta che il contenuto di un QR Code viene estratto, viene chiamata la

funzione `composeKey(qrContent: String)`, che accetta il contenuto del codice QR in formato stringa e si occupa della decriptazione e della ricomposizione della chiave.

8.6.2 Funzionamento della `composeKey(qrContent: String)`

La funzione `composeKey` rappresenta il fulcro del processo di ricostruzione della chiave. Al suo interno, i dati codificati del QR Code vengono prima decodificati utilizzando il metodo `base64Decoded`, per poi essere ulteriormente processati.
In particolare, la funzione delega alcune operazioni minori a metodi di supporto per suddividere il lavoro, garantendo maggiore modularità e mantenibilità del codice.

I principali compiti svolti dalla funzione di supporto sono:

- `getContent(s: String)`: Estrae il contenuto informativo rilevante dalla stringa del QR Code.
- `getTotalStep(parti: List<String>)`: Identifica il numero totale di segmenti che compongono la chiave.
- `getStep(parti: List<String>)`: Determina a quale segmento appartiene il dato corrente.
- `insertSegment(actualStep: Int, qrData: String)`: Inserisce il segmento nel punto corretto dell'array `arrayChunk`, che conserva i frammenti della chiave.

8.6.3 Processo di Ricomposizione della Chiave

Durante l'analisi dei QR Code, i segmenti della chiave vengono raccolti in un array chiamato `arrayChunk`.
La funzione `insertSegment` inserisce ciascun segmento nella posizione appropriata in base al suo "**step**", ovvero l'ordine in cui il frammento deve essere posizionato nella chiave finale.
Quando tutti i segmenti sono stati acquisiti (cioè quando il numero di segmenti raccolti nell'array corrisponde al numero totale di segmenti identificati da `getTotalStep`), viene chiamata la funzione `composeSegment`.
La funzione concatena tutti i frammenti per formare la chiave completa, memorizzata nella variabile `key`.

Una volta ricomposta, il processo di scansione viene terminato chiamando `terminateScanProcess`, e la variabile `endScanner` viene aggiornata per indicare che il processo è stato completato, l'aggiornamento di questa variabile causa una ricomposizione delle componenti dell'interfaccia e dalla schermata di scannerizzazione si passa alla Home, nel caso in cui la chiave fosse identica a quella inserita come input.

8.6.4 Ulteriori Dettagli Tecnici

- **Decodifica Base64**: Il contenuto del QR Code è codificato in Base64, quindi viene utilizzata la funzione `base64Decoded` per convertirlo in una stringa leggibile.
- **Composizione Segmenti**: Ogni QR Code contiene una porzione della chiave, e il processo di ricomposizione avviene mediante la concatenazione di questi segmenti.
- **Controllo degli Step**: Le funzioni `getStep` e `insertSegment` assicurano che ogni segmento sia inserito nella posizione corretta. Questo evita la duplicazione di segmenti e garantisce che la chiave venga ricostruita in modo corretto.

8.7 Conclusioni

L'implementazione della funzionalità di scannerizzazione dei codici QR in tempo reale su piattaforme Android e iOS ha evidenziato differenze significative nelle API e nei meccanismi di gestione delle risorse hardware, ma al contempo ha sottolineato la possibilità di creare soluzioni cross-platform efficienti e sicure.

In Android, l'uso di `CameraX` ha reso possibile gestire l'acquisizione delle immagini e la scansione dei QR Code in modo flessibile, sfruttando API avanzate per l'analisi delle immagini.
L'integrazione del sistema di gestione dei permessi garantisce un controllo preciso e continuo sull'accesso alle risorse hardware, offrendo all'utente una maggiore consapevolezza e controllo. Allo stesso tempo, l'implementazione della preview della camera e del flusso di analisi in tempo reale permette una scannerizzazione efficiente dei codici, con prestazioni ottimizzate grazie alla gestione asincrona delle operazioni tramite coroutines.

In iOS, l'uso di `AVFoundation` ha garantito un'alta qualità di gestione delle risorse multimediali, con un approccio user-friendly alla gestione dei permessi e un sistema robusto per l'elaborazione dei dati provenienti dalla fotocamera.
La preview dinamica, combinata con la gestione dell'orientamento del dispositivo, ha offerto un'esperienza utente fluida e adattabile, mentre le coroutines hanno ottimizzato l'esecuzione delle operazioni di decodifica del QR code.

Unendo queste due implementazioni, è possibile affermare che lo sviluppo di funzionalità avanzate di scansione di codici QR su dispositivi mobili può beneficiare di un approccio cross-platform, adattando le specificità di ciascun sistema operativo pur mantenendo elevati standard di prestazioni, sicurezza e usabilità.
Entrambe le soluzioni offrono un'integrazione completa con i rispettivi ecosistemi (Android e iOS), garantendo che l'applicazione possa operare in modo efficace e sicuro su entrambe le piattaforme.

Per quanto riguarda la gestione della chiave la classe `KeyManagerScanner` è una componente fondamentale per un controllo sicuro della chiave durante il processo di scansione dei QR Code.
La modularità e la possibilità di utilizzare codice condiviso tra piattaforme rendono il sistema flessibile e facilmente estendibile.
Grazie alla dependency injection e all'uso del pattern Singleton, l'architettura è ottimizzata per ridurre l'uso di risorse e garantire un'esperienza utente fluida durante la scansione e la decriptazione delle chiavi.

9 Sviluppo di un SDK Multiplatform

9.1 Introduzione

Un Software Development Kit (SDK) è un insieme di strumenti forniti per facilitare lo sviluppo di applicazioni su specifiche piattaforme, sistemi operativi o linguaggi di programmazione.
L'SDK fornisce librerie, strumenti di debugging, documentazione e API, consentendo agli sviluppatori di creare applicazioni in modo efficiente e standardizzato.
Nel contesto del Kotlin Multiplatform, l'obiettivo è creare un SDK che sfrutti la condivisione del codice su più piattaforme, garantendo al contempo un'interfaccia coerente e prestazioni ottimizzate per ciascun sistema operativo.[7]

9.2 Soluzione

La creazione di un SDK richiede particolare attenzione alla modularità e alla gestione del codice, considerando che non tutte le funzioni e componenti interne dell'SDK sono visibili agli utilizzatori finali.
È buona pratica definire interfacce che permettano di utilizzare il set di strumenti in modo sicuro e controllato.
Dopo aver ideato e definito le parti del SDK con visibilità limitata, le funzioni che possono interfacciarsi con l'esterno sono state classificate, spesso subendo modifiche rispetto alla configurazione iniziale del prototipo per adattarsi meglio alle necessità del progetto.

9.2.1 Modulo Comune

Il modulo comune del SDK in Kotlin Multiplatform contiene interfacce e logica condivisa che definiscono le operazioni base per la gestione, generazione e analisi dei QR Code, applicabile a tutte le piattaforme.

- **Modulo comune di scannerizzazione:**
  - `KeyManagerScanner.kt` e `KeyManagerScannerImpl.kt`: la prima è un'interfaccia che contiene la firma della funzione di avvio per la

composizione della chiave e una variabile di terminazione nel momento in cui la chiave è completata.
La seconda è una classe che implementa le funzioni dell'interfaccia con la logica di decriptazione e ricomposizione della chiave estratta dai codici QR.

- `QrAnalyzerManager.kt`: interfaccia che contiene le funzioni di avvio e terminazione del processo di scannerizzazione.

- **Modulo comune di generazione**:
  - `QRCodeManager.kt`: interfaccia che controlla la funzione di generazione e l'inizio della visualizzazione dei QR code.
  - `QrCodeGeneratorImpl.kt`: si tratta di una classe definita internamente e xghe utilizza la tecnologia `expect/actual` per delegare ai target specifici l'Implementazione della logica per la generazione di codici QR.
  - `KeyManagerGeneration.kt`: classe interna che mantiene il controllo della chiave durante la fase di generazione.

- **Data Class**: contiene i modelli di dati che vengono utilizzati durante il processo, sono **QrData** e **QrKey** che sono stati ampiamente analizzati nei capitoli precedenti.

9.3 Implementazione Android

La parte Android del SDK utilizza il metodo `expect/actual` di Kotlin Multiplatform per integrare le implementazioni specifiche della piattaforma con il codice comune definito nel modulo `commonMain`. Questo approccio consente una gestione fluida delle differenze tra le piattaforme pur mantenendo una base di codice comune.

- Modulo Utils:
  - `FormatConverter.kt` e `FormatConverterImpl.kt`: Definiscono le funzioni di conversione necessarie per manipolare e convertire i codici QR in formati utilizzabili.

- Modulo di generazione:

  - `QrCodeGeneratorImpl.android.kt`: Gestisce la generazione di codici QR con parametri specifici adattati alle necessità dell'applicazione Android.

- Modulo di scannerizzazione:

  - `QrAnalyzerManager.android.kt`: Implementa le funzioni specifiche di Android per l'analisi dei codici QR.

9.4 Implementazione iOS

Similmente ad Android, le implementazioni specifiche per iOS, definite nel modulo `iosMain`, comprendono la stessa struttura e gli stessi file con i corpi delle funzioni diverse basate sulle API disponibili. Anche in questo caso la logica utilizzata non si discosta molto da quella sviluppata per il prototipo. Come già affermato in precedenza, la generazione dei QR Code è ora definita in modo specifico sulle piattaforme, garantendo che le peculiarità di ciascuna piattaforma siano accuratamente considerate durante lo sviluppo dell'SDK.

Queste componenti garantiscono che il nostro SDK sia flessibile, potente e ottimizzato per sfruttare al meglio le caratteristiche native di ciascuna piattaforma.

9.5 Pubblicazione

Il processo di pubblicazione di un SDK è relativamente semplice e non richiede configurazioni complesse del progetto. Per la pubblicazione di una libreria, o in questo caso un SDK, esistono diverse modalità e livelli di visibilità a seconda dell'uso previsto.
Per questo progetto, abbiamo optato per una pubblicazione locale, ossia il salvataggio dell'SDK in una repository interna del nostro ambiente di sviluppo, denominata **Meaven Local**.
Per pubblicare su Meaven, è necessario aggiungere nel file build.gradle.kts il nome dell'artefatto e configurare il comando per ogni target Android su cui si è sviluppato codice, come ad esempio:

```
1 publishLibraryVariants("release", "debug")
```

Nel codice sovrastante, l'opzione "debug" non è obbligatoria, ma consente di includere funzionalità di debug nella libreria.

La configurazione per la pubblicazione di un artefatto per iOS differisce leggermente. Per pubblicare l'SDK, è necessario aggiungere le dipendenze del target nel source set **commonMain**:

```
val iosMain by getting{
    dependsOn(commonMain);
    ios64Main.dependsOn(this);
}
```

Una volta configurati i source set specifici per iOS e eseguito il comando di pubblicazione, l'SDK sarà pubblicato localmente. [5]

9.6 Conclusioni

La creazione di un SDK è stata una fase molto importante che ha richiesta particolare attenzione sulla modularizzazione dei componenti, è stato importante astrarre su livelli la logica protagonista del processo per suddividere lo strato visibile e utilizzabile attraverso delle interfacce che espongono all'esterno le funzioni, da quello che invece opera in background, eseguendo funzioni complesse e con un tempo computazionale dovuto alla complessità dell'operazione elevato.

Per quanto riguarda la condivisione del codice, Kotlin Multiplatform non permette di condividerlo totalmente a causa di problematiche legate alla diversità tra l'hardware e il sistema operativo dei dispositivi, perciò soluzioni complesse devono ancora essere sviluppate in modo specifico all'interno dei target.
Nonostante questo, una gran percentuale di codice prodotto è stato condiviso, perciò ha permesso di velocizzare molto il processo di sviluppo del sistema.

Un altro aspetto importante è che una volta che l'SDK è stato sviluppato può essere integrato su piattaforme diverse e questo cio permette di velocizzare il processo di integrazione in quanto tra le piattaforme l'utilizzo delle funzioni è lo stesso infatti questo ci permetterà di risparmiare successivamente molto codice.

10 Benchmark

10.1 Introduzione

Questo capitolo analizza le prestazioni dei dispositivi nella generazione e nella scansione dei codici QR, in oltre, viene anche valutata la portabilità di Kotlin multiplatform riguardo alla condivisione del codice per questo progetto.
Durante la fase di sviluppo, abbiamo condotto diverse valutazioni per ridurre il divario prestazionale tra i dispositivi di alta e bassa gamma.
È emerso che, nei dispositivi di gamma inferiore, il tempo di generazione dei primi codici QR era significativamente elevato, stabilizzandosi solo nelle fasi successive del processo. Al contrario, nei dispositivi di alta gamma, il tempo di generazione rimaneva costante per tutta la durata dell'operazione.

Per armonizzare le prestazioni tra dispositivi diversi, abbiamo optato per la parallelizzazione del processo computazionale su più thread: uno dedicato alla gestione dell'interfaccia utente e uno alla generazione dei codici QR.
Questa scelta ha permesso di temporizzare l'avvio della fase di visualizzazione dei QR Code anche mentre la loro generazione era ancora in corso, migliorando l'esperienza utente.
Tuttavia, questo approccio ha richiesto l'implementazione di meccanismi di sincronizzazione per prevenire inconsistenze, data la condivisione dei QR Code come risorse tra i vari thread. Inoltre, il codice è stato sottoposto a un'attenta revisione per eliminare calcoli e cicli superflui, ottimizzando ulteriormente il processo.

Per quanto riguarda la scansione dei QR Code, abbiamo notato che era sostanzialmente più rapida della generazione. Abbiamo quindi ridotto il tempo di visualizzazione di ciascun QR sullo schermo del dispositivo generatore, un cambiamento che ha prevenuto la rilettura ripetuta dello stesso codice QR da parte del dispositivo scanner. Diminuendo la durata di esposizione di ogni QR sullo schermo, abbiamo significativamente ridotto le operazioni di scansione non necessarie.

10.2 Condivisione del codice

Nei capitoli precedenti abbiamo introdotto Kotlin Multiplatform, un linguaggio di programmazione versatile che semplifica lo sviluppo su più piattaforme, accelerando il processo grazie alla condivisione del codice e riducendo la duplicazione della logica di business.

In questa tesi, analizziamo due principali progetti: il prototipo, che integra sia un'interfaccia grafica sia la logica per la generazione e la scansione dei codici QR, e l'SDK, che implementa gran parte della logica di questi due processi.
L'obiettivo è valutare il grado di condivisione del codice tra le diverse piattaforme confrontando i due progetti.

Per lo sviluppo del prototipo, i risultati sono incoraggianti. Come evidenziato nel grafico a torta sottostante, la maggior parte del codice è stata sviluppata nel modulo condiviso. Esempi significativi includono il processo di generazione dei codici QR e l'interfaccia grafica, resa possibile da (Jetpack) Compose Multiplatform. La porzione rimanente rappresenta il codice specifico per la gestione dei permessi della fotocamera e il processo di scansione dei QR code.

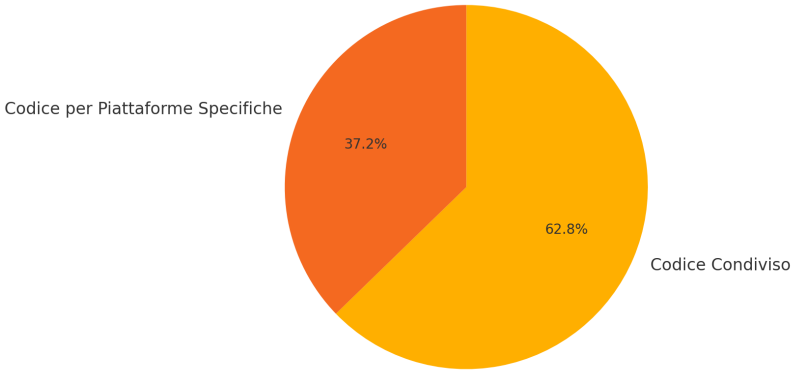

Figure 14: Applicazione: condivisione del codice

Per quanto riguarda lo sviluppo dell'SDK, la condivisione del codice è stata meno efficiente. Circa il 70% del codice è stato scritto specificamente per le piattaforme, con parti separate per iOS e Android legate ai processi di generazione e scansione. Solo la gestione delle chiavi, che non dipende dall'hardware o dal sistema operativo, è stata condivisa tra le piattaforme.

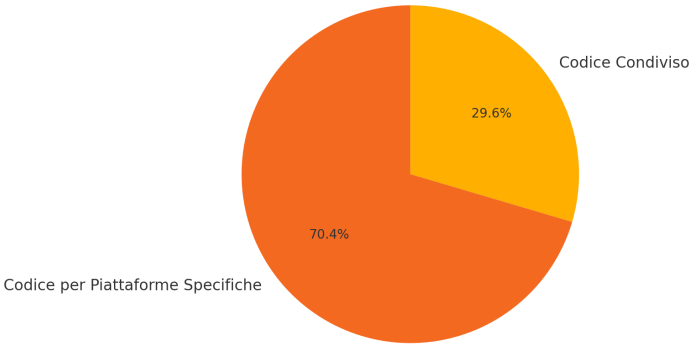

Figure 15: SDK: condivisione del codice

10.3 Analisi delle prestazioni

Durante lo svolgimento di questo progetto, i tempi di computazione sono stati monitorati e classificati in diverse fasi significative, come ad esempio la funzione di generazione dei QR Code.
Per misurare il tempo di computazione, è stata utilizzata la libreria **TimeMark**, che fornisce dei **marker** per misurare il tempo trascorso tra due punti specifici del codice. Questo approccio ha permesso di eseguire dei benchmark per analizzare le prestazioni dei vari dispositivi, oltre che migliorare continuamente il codice durante lo sviluppo del sistema di Logo Dinamico.

Le analisi riportate in questa sezione si concentrano principalmente su due dei sistemi operativi più utilizzati al mondo, ovvero **iOS** e **Android**.
L'obiettivo era comprendere come le prestazioni del sistema potessero variare tra questi due ambienti. I risultati evidenziano differenze significative nelle performance, in particolare in alcune fasi critiche del processo.

Nel grafico qui sotto, viene rappresentata la generazione dei QR Code utilizzando chiavi di diverse lunghezze. Queste chiavi vengono scomposte in frammenti (*chunks*) e passate come input alla funzione di generazione dei QR Code.

Come si può notare, c'è una netta differenza di prestazioni tra i dispositivi Android e iOS. In particolare, iOS risulta essere molto più performante rispetto ad Android per tutte le lunghezze delle chiavi testate.

I risultati ottenuti mostrano che non è tanto la dimensione complessiva della chiave a influenzare le prestazioni, quanto piuttosto la grandezza dei *chunks* in cui la chiave viene scomposta. Infatti, le chiavi di dimensioni maggiori possono, in

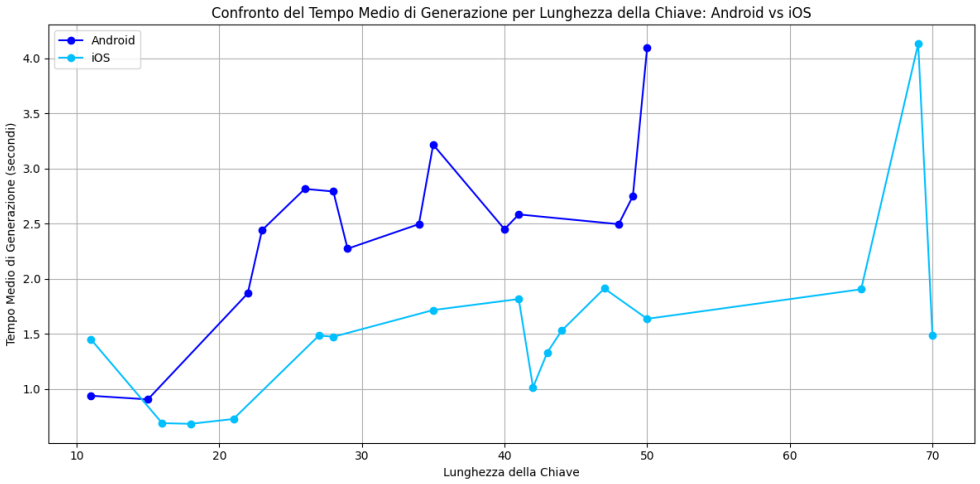

Figure 16: Confronto dei tempi di generazione con chiavi di diverse lunghezze

alcuni casi, richiedere meno tempo di elaborazione rispetto a quelle più piccole, a seconda di come vengono suddivise.
Questo effetto sarà più evidente nei grafici successivi, dove chiavi suddivise in segmenti più grandi migliorano significativamente le prestazioni dei dispositivi.

Nel grafico successivo, è fornita una visualizzazione delle prestazioni medie per la generazione dei QR Code, considerando chiavi di diverse lunghezze. Questo confronto fornisce un'indicazione chiara delle differenze di velocità tra i due sistemi operativi e delle prestazioni complessive nei diversi intervalli di lunghezza delle chiavi considerati.

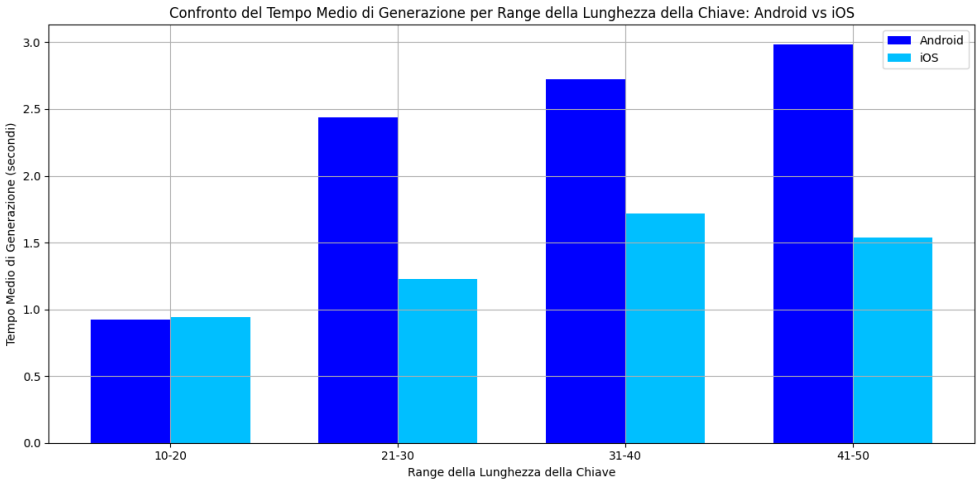

Figure 17: Prestazioni medie di generazione dei QR Code con chiavi di diverse lunghezze

10.3.1 Analisi sulle lunghezze dei chunks

Ho calcolato il tempo medio di generazione per diverse lunghezze dei chunk sia su Android che su iOS.
I risultati mostrano che i dispositivi iOS mantengono tempi di generazione più brevi, garantendo un'esecuzione più efficiente in quasi tutte le condizioni rispetto ai dispositivi Android. Le lunghezze dei chunk comprese tra 6 e 10 si sono rivelate ottimali per entrambi i sistemi, minimizzando il tempo di generazione.

Raccomandazioni sulla lunghezza del chunk:

- Per Android, la lunghezza ottimale del chunk è tra 6 e 10, anche se potrebbero essere necessari ulteriori test per ottimizzare le prestazioni con chiavi più lunghe.
- Per iOS, la lunghezza ideale del chunk è compresa tra 8 e 9, garantendo un buon equilibrio tra complessità e tempi di generazione.

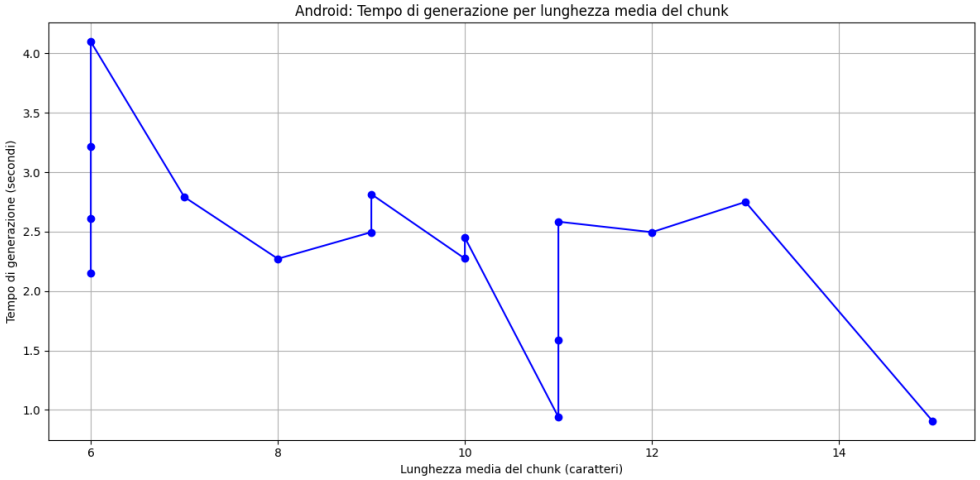

Figure 18: Tempo di generazione in base alla lunghezza del chunks per Android

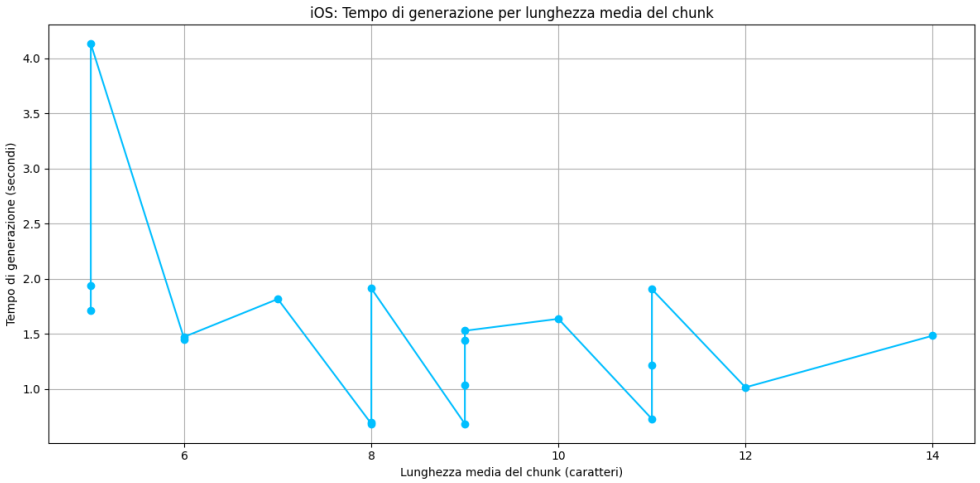

Figure 19: Tempo di generazione in base alla lunghezza del chunks per iOS

10.3.2 Tempi di visualizzazione e generazione a confronto

Un'altra significativa rappresentazione è riportata nel grafico seguente, che mostra i tempi medi di generazione e visualizzazione dei QR Code per entrambi i sistemi operativi.

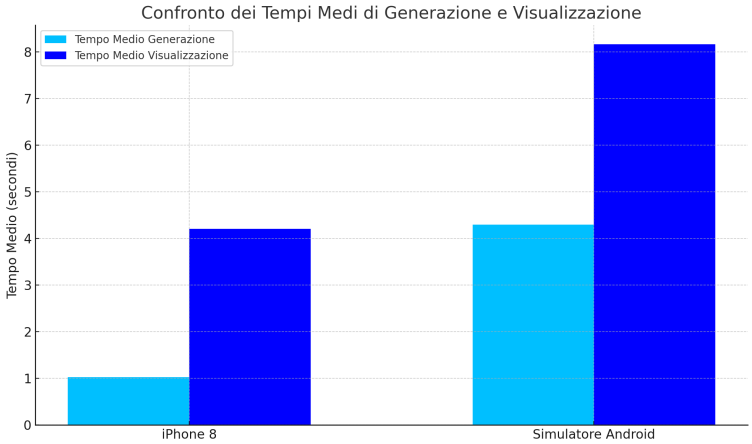

Figure 20: Tempi medi di generazione dei QR Code

La differenza tra i due sistemi operativi si manifesta ulteriormente nella fase di generazione del primo QR Code. In particolare, i dispositivi Android richiedono un tempo di computazione maggiore rispetto ai dispositivi iOS per la generazione del primo QR Code, e questo ritardo ha un impatto significativo sulle operazioni successive.

Nel grafico seguente, si può osservare come il tempo di generazione del primo QR Code su Android sia, in media, più lungo rispetto a quello sui dispositivi iOS. Questo comporta un ritardo nelle operazioni successive, inclusa la visualizzazione del primo QR Code, che avviene con un notevole ritardo su Android rispetto a iOS.

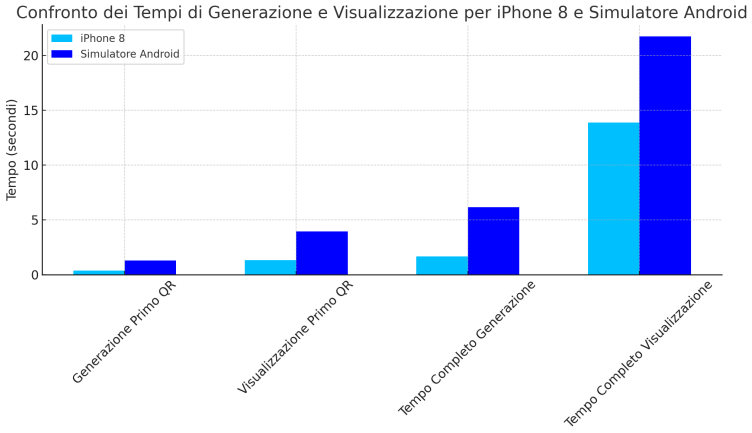

Figure 21: Tempi di generazione e visualizzazione dei QR Code

Questo grafico mostra i tempi di visualizzazione dei QR Code sui due dispositivi a confronto. Come si può osservare, la visualizzazione segue una tendenza lineare crescente, a indicare che la temporizzazione inserita è efficace nel garantire una visualizzazione uniforme. Questo approccio contribuisce a ridurre la differenza nelle prestazioni tra i due dispositivi, migliorando l'esperienza utente in modo coerente su entrambe le piattaforme.

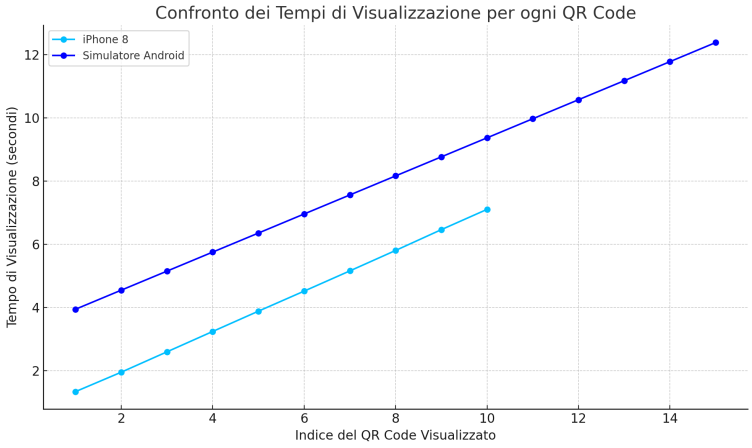

Figure 22: Tempi di visualizzazione del primo QR Code

I risultati ottenuti dimostrano chiaramente come i dispositivi iOS offrano prestazioni migliori nella generazione e visualizzazione dei QR Code, rendendoli più adatti a scenari in cui la velocità di risposta è critica. Questa differenza di prestazioni è stata uno dei principali punti di attenzione durante lo sviluppo del sistema, poiché l'obiettivo era garantire un'esperienza utente il più uniforme

possibile su entrambi i sistemi operativi.

10.4 Valutazioni delle Prestazioni nella Scannerizzazione dei QR Code

Un aspetto cruciale per l'implementazione dei QR code crittografati è la valutazione delle prestazioni nella loro scannerizzazione. L'aggiunta di crittografia ai QR code può influenzare le prestazioni della scansione, specialmente in termini di tempo di decodifica e usabilità. Gli studi hanno evidenziato diversi fattori che incidono su queste prestazioni:

- **Dimensione del QR Code e Complessità dei Dati:**La quantità di dati crittografati inseriti in un QR code influisce sulla sua densità. Maggiore è la dimensione del payload e più denso sarà il codice, rendendo la scansione più lenta o problematica. Esperimenti hanno dimostrato che QR code più grandi (ad esempio 400x400 pixel) sono più facili da scansionare rispetto a quelli di dimensioni ridotte con lo stesso volume di dati.

- **Tempo di scansione:** Il tempo di scansione varia notevolmente a seconda della dimensione del QR code e della quantità di dati crittografici che contiene. Le firme digitali (soprattutto RSA con chiavi di 3.072 bit) e gli HMAC più complessi possono aumentare leggermente il tempo di scansione, ma moderni smartphone gestiscono questi tempi in maniera efficiente, con un overhead di circa 20-30 millisecondi per la verifica delle firme digitali.

- **Usabilità e Esperienza Utente:** La densità del QR code ha un impatto diretto sull'usabilità. Codici molto densi, generati da grandi quantità di dati crittografati, possono portare a scansioni fallite o errate, richiedendo più tentativi. Nei test condotti, è stato evidenziato che la percentuale di scansioni corrette diminuisce all'aumentare della densità del codice, specialmente quando il codice contiene più di 1.600 byte di dati. La scelta della dimensione ottimale e del formato del QR code è dunque fondamentale per bilanciare sicurezza e usabilità.[10]

10.5 Conclusioni sulle analisi condotte

Le analisi effettuate mostrano chiaramente come i dispositivi Android abbiano tempi di generazione generalmente più elevati rispetto a iOS, sia per quanto riguarda la lunghezza delle chiavi sia per la dimensione dei chunk. Tuttavia, la lunghezza ottimale del chunk per entrambi i sistemi risulta compresa tra 6 e 10, con iOS che gestisce con maggiore efficienza chunk più grandi rispetto ad Android.

In tutte le categorie esaminate, iOS si è dimostrato superiore in termini di prestazioni, suggerendo una migliore ottimizzazione del software per la gestione dei QR code rispetto ad Android. Questo conferma che iOS è più adatto a scenari in cui la velocità di risposta è un fattore critico, garantendo un'esperienza utente più fluida e reattiva.

Per garantire, in fase di scannerizzazione una buona esperienza utente, è consigliabile limitare la dimensione dei dati nei Qr code a meno di 1200 byte, questo aiuta a mantenere i codici scansionabili e velocizza il processo senza compromettere la sicurezza.

L'introduzione di firme digitali, in particolare quelle basate su ECDSA con chiavi di 256 bit, offre un equilibrio ottimale tra sicurezza e velocità di elaborazione. Questa tecnologia non solo rafforza la protezione dei dati, ma anche mantiene la fluidità dell'esperienza utente durante la scansione del codice.

In conclusione, per ottenere le migliori prestazioni senza sacrificare la sicurezza, è cruciale trovare un equilibrio tra la dimensione dei dati, il tipo di crittografia utilizzata e la dimensione fisica del QR code. Con questi accorgimenti, possiamo assicurare non solo la sicurezza dei dati, ma anche un'esperienza utente senza intoppi.

Bibliografia

References

[1] Kotlin documentation. https://kotlinlang.org/docs/kotlin-tour-welcome.html. Last modified: 01 August 2024.

[2] AAkira. Napier: logging library. https://github.com/AAkira/Napier. Last modified: 12 January 2024.

[3] Apple. Avfoundation. https://developer.apple.com/av-foundation/. Last visited: 2024-10-08.

[4] Android Developers. Imagebitmap. https://developer.android.com/reference/kotlin/androidx/compose/ui/graphics/ImageBitmap. Last updated: 2024-09-05.

[5] Kotlin documentation. Publish sdk. https://kotlinlang.org/docs/multiplatform-publish-lib.html. Last modified: 25 September 2024.

[6] Snap for Developers. Camera kit. https://developers.snap.com/camera-kit/home. Last visited: 2024-10-08.

[7] Red Hat. Software development kit. https://www.redhat.com/it/topics/cloud-native-apps/what-is-SDK. Last modified: 4 August 2023.

[8] JetBrains. Case studies. https://www.jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html. Last modified: 18 January 2024.

[9] Kotlin. Expect/actual. https://kotlinlang.org/docs/multiplatform-expect-actual.html. Last modified: 12 March 2024.

[10] Heider A. M. Wahsheha Riccardo Focardia, Flaminia L. Luccioa. Usable security for qr code. https://iris.unive.it/retrieve/e4239dde-a7bc-7180-e053-3705fe0a3322/FocardiLuccioWahsheh2019.pdf. Publicated: 2019-08-06.

[11] ssvaghasiya. Qrkit compose multiplatform. https://github.com/Chaintech-Network/QRKitComposeMultiplatform?source=post_page-----858ec3e1e2b2--------------------------------. Last update: 2024-07-19.

[12] Koin Team. Koin - dependency injection. https://insert-koin.io/. Last visited: 2024-10-5.

[13] Kotlin Framework team. Kotlin programming language. https://kotlinlang.org/docs/null-safety.html. Last modified: 11 September 2023.

[14] Wikipedia. Kotlin programming language. https://en.wikipedia.org/wiki/Kotlin_(programming_language).

[15] Wikipedia. Qrcode. https://it.wikipedia.org/wiki/Codice_QR.

[16] Wikipedia. Steganografia. https://it.wikipedia.org/wiki/Steganografia. Last updated: 2024-05-24.

11 Ringraziamenti

Con profonda gratitudine, desidero ringraziare tutte le persone che mi hanno sostenuto e accompagnato durante questo percorso, contribuendo in maniera significativa alla realizzazione di questo lavoro:

- **Il Prof. Ferruccio Damiani**, che, oltre ad essere stato il mio docente nel corso di Sviluppo Applicazioni Software, mi ha seguito con grande dedizione durante il mio stage e nella stesura di questa tesi, fornendomi costante supporto

- **Marco Balanzino**, Manager di IrisCube Reply, per avermi accolto calorosamente in azienda e per il suo continuo sostegno durante tutto il periodo dello stage, permettendomi di crescere sia professionalmente che personalmente.

- **Salvatore Vitobello**, il mio supervisore e mentore, che mi ha guidato passo dopo passo nello sviluppo del progetto. Grazie a lui, ho potuto apprendere un'enorme quantità di conoscenze fondamentali, che hanno arricchito il mio percorso di formazione.

- **La mia famiglia**, per il loro sostegno incondizionato, senza il quale non avrei potuto superare le sfide di questo viaggio universitario.

- **I miei amici**, compagni di avventure con cui ho condiviso discussioni, risate e momenti indimenticabili, che hanno reso più leggero e piacevole questo cammino.

- **La mia fidanzata**, che con amore e pazienza è stata sempre al mio fianco, soprattutto nei momenti più difficili, infondendomi forza e serenità.