Alma Mater Studiorum · Università di Bologna
Campus di Cesena

dipartimento di informatica – scienza e ingegneria

Corso di Laurea Magistrale in Ingegneria e scienze informatiche

DEVOPS PER APPLICAZIONI MOBILE MULTIPIATTAFORMA: UN CASO DI STUDIO INDUSTRIALE

Tesi di Laurea Magistrale in
Laboratorio di Sistemi Software LM

Relatore:
Prof. Danilo Pianini

Presentata da:
Filippo Paganelli

Correlatore:
Prof.ssa Catia Prandi

II Sessione
Anno Accademico 2021/2022

Sommario

Distribuire un'applicazione che soddisfi i requisiti del cliente e che sia in grado di accogliere rapidamente eventuali modifiche è, al giorno d'oggi, d'obbligo per aziende che si occupano di applicazioni mobile che vogliano rimanere competitive sul mercato. Il principale fattore chiave in grado di mantenere un'azienda al passo con la continua evoluzione sia del mercato che delle tecnologie mobile è la continua innovazione e ottimizzazione dell'intero processo di sviluppo.

La collaborazione e la comunicazione tra diversi team, l'utilizzo di cicli iterativi di sviluppo, i rilasci frequenti e l'automazione dei test sono alcune delle pratiche incentivate dalla cultura DevOps, adottabile con successo (si vedrà in questa tesi) anche per lo sviluppo di applicazioni mobile.

Innovare e ottimizzare il processo di sviluppo non significa solo automatizzare l'esecuzione dei task. Un ruolo importante è giocato anche da aspetti legati all'applicazione: l'architettura, il paradigma di programmazione, e gli strumenti utilizzati. In particolare, al fine di applicare il principio "*Don't repeat yourself*" (DRY) e semplificare la manutenzione, diversi moderni framework per lo sviluppo di applicazioni mobile, detti multipiattaforma, propongono meccanismi che consentono di condividere codice tra piattaforme differenti.

L'obiettivo di questa tesi è dunque quello di discutere (capitoli 1, 2 e 3) e mostrare, applicate ad un caso di studio industriale (capitoli 4, 5 e 6), l'uso di tecniche DevOps nell'ambito di applicazioni mobile, ed in particolare mostrando come queste siano applicabili in congiunzione ai framework di sviluppo multipiattaforma (in particolare, Kotlin Multiplatform).

Indice

Capitolo 1

Metodologia DevOps

1.1 Introduzione

A causa della continua evoluzione delle tecnologie e dell'aumento della concorrenza le aziende hanno bisogno di realizzare prodotti sempre più velocemente, mantenendo (e possibilmente aumentando) la loro qualità [7]. In risposta a questo problema è nata una vera e propria cultura, chiamata DevOps, la quale definisce sia un modo di pensare che una metodologia di lavoro fortemente basata sulla collaborazione tra i diversi team.

Questa cultura fornisce infatti un insieme di pratiche al fine di ridurre le barriere tra il team di sviluppo (Dev) e il team operativo (Ops). Da un lato gli sviluppatori vogliono innovare e rilasciare software velocemente, mentre dall'altro lato l'obiettivo è quello di garantire la stabilità e la qualità dei sistemi. Oltre alla collaborazione tra i diversi team esistono altri principi alla base della cultura DevOps:

- **Automazione** - L'intervento umano deve essere ridotto il più possibile al fine di ridurre gli errori e le tempistiche. In questo modo vengono rimossi tutti i compiti ripetitivi che sarebbero stati a carico dello sviluppatore facendogli risparmiare tempo.
- **Monitoraggio** - Dev'essere possibile analizzare in ogni momento lo stato dei processi che compongono il sistema con lo scopo di reagire, prevenire e prevedere le situazioni critiche ma anche fornire feedback allo sviluppatore per aumentare la qualità del prodotto.

1.2 Tecniche di automazione e vantaggi

Non è possibile definire un unico metodo di adozione della cultura DevOps: ogni azienda ha i propri vincoli, requisiti e metodi che la rendono unica permettendole di distinguersi dalle altre. Esistono invece diverse tecniche che rispettano i principi DevOps e che possono essere applicate e adattate ad ogni azienda al fine di aumentare la qualità del software prodotto e diminuire il "*time-to-market*" (TTM) [2].

I principali benefici derivanti dall'adozione della cultura DevOps sono [7]:

- Migliore comunicazione e collaborazione all'interno dei team con un impatto umano e sociale a livello aziendale.
- Tempi di consegna in produzione minori che comportano un aumento sia delle performance del processo che della soddisfazione dell'utente finale.
- Risparmio di tempo e risorse dovuti dall'utilizzo di un ciclo di sviluppo iterativo e tecniche di automazione.

I team che intendono adottare le tecniche e pratiche DevOps nel proprio processo di sviluppo devono seguire metodologie Agile con fasi iterative che permettono maggiore qualità delle funzionalità e feedback rapidi da parte dell'utente. Le principali pratiche che vengono adottate sono conosciute come (*i*) Continuous Integration, (*ii*) Continuous Delivery, (*iii*) Continuous Deployment, (*iv*) Continuous Monitoring e (*v*) Continuous Inspection.

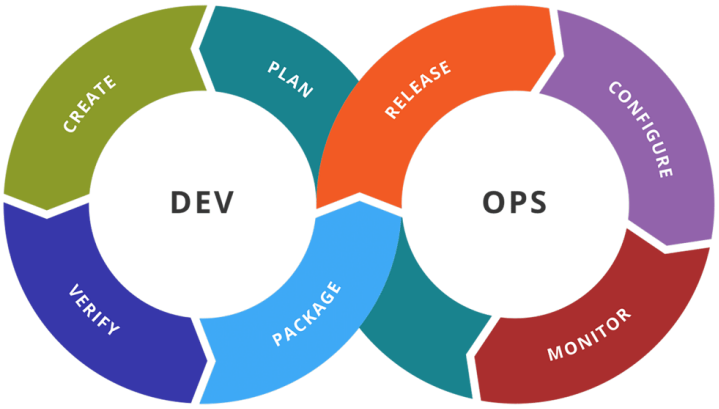

Figura 1.1: Fasi del ciclo di sviluppo software con tecniche DevOps

1.2.1 Continuous Integration

Maggiore è la complessità di un progetto e maggiore è la necessità di integrare frequentemente e preventivamente i componenti software al fine di verificare che essi funzionino correttamente [3]. Con Continuous Integration (CI) si intende una pratica di sviluppo software dove i membri di un team integrano frequentemente il loro lavoro e ogni integrazione è testata e verificata da un sistema automatico in grado di intercettare gli errori il più velocemente possibile[1].

Con pipeline si intende una sequenza ordinata di elaborazioni che vengono applicate al codice sorgente in modo automatico. Solitamente una pipeline di CI prevede tre task:

- **Build** - Task iniziale della pipeline che consiste nella verifica della corretta compilazione del codice sorgente.

[1] https://martinfowler.com/articles/continuousIntegration.html

- **Test** - Successivamente vengono eseguiti i task di testing, tipicamente riguardanti la logica applicativa (unit testing).
- **Package** - Lo scopo dell'ultimo task della pipeline di CI consiste nella verifica della corretta pacchettizzazione del codice. Deve essere possibile creare correttamente l'artefatto che verrà poi passato alle fasi successive di rilascio (delivery).

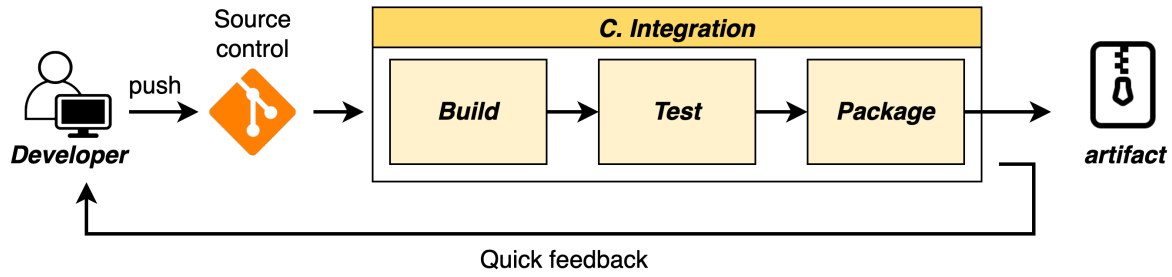

Figura 1.2: Pipeline di Continuous Integration

Ogni sviluppatore lavora effettuando piccole modifiche al codice, dette *commit*, che possono essere facilmente analizzate e risolte in caso di errori: maggiore è la dimensione/-complessità della modifica applicata al codice e maggiore è il rischio di problemi, oltre all'accumulo di debito tecnico. Grazie ai sistemi di versionamento del codice (VCS[2]) e agli strumenti che essi forniscono, come ad esempio *branch*, *commit*, *pull request* e *code review*, gli sviluppatori possono collaborare sul codice. Unendo le funzionalità dei VCS con le funzionalità di altri sistemi appositi per l'automazione di task, come GitLab CI[3], GitHub Actions[4], Travis CI[5] e Bitbucket Pipelines[6], è possibile adottare efficacemente le tecniche di CI nel processo di sviluppo. Tali strumenti a supporto dei sistemi di automazione sono descritti in modo dettagliato nella sezione 1.3 di questo capitolo.

1.2.2 Continuous Delivery

Quando la fase di CI termina con successo è possibile procedere con la fase successiva di rilascio automatico del codice. Con Continuous Delivery (CD) si intende una pratica di sviluppo dove il software viene sviluppato in una maniera tale da poter essere installato nell'ambiente di produzione in qualsiasi momento[7]. Questo significa che lo stesso concetto di piccole modifiche molto frequenti definito in fase di integrazione è valido anche nella fase di rilascio.

Il compito principale della fase di CD è quello di prendere il software impacchettato nell'ultimo task della fase di CI, chiamato *package*, e renderlo disponibile alla fase suc-

[2]Version Control System
[3]https://docs.gitlab.com/ee/ci/
[4]https://github.com/features/actions
[5]https://www.travis-ci.com/
[6]https://bitbucket.org/product/it/features/pipelines
[7]https://martinfowler.com/bliki/ContinuousDelivery.html

cessiva d'installazione (deployment) tramite la pubblicazione su appositi sistemi chiamati *package manager*.

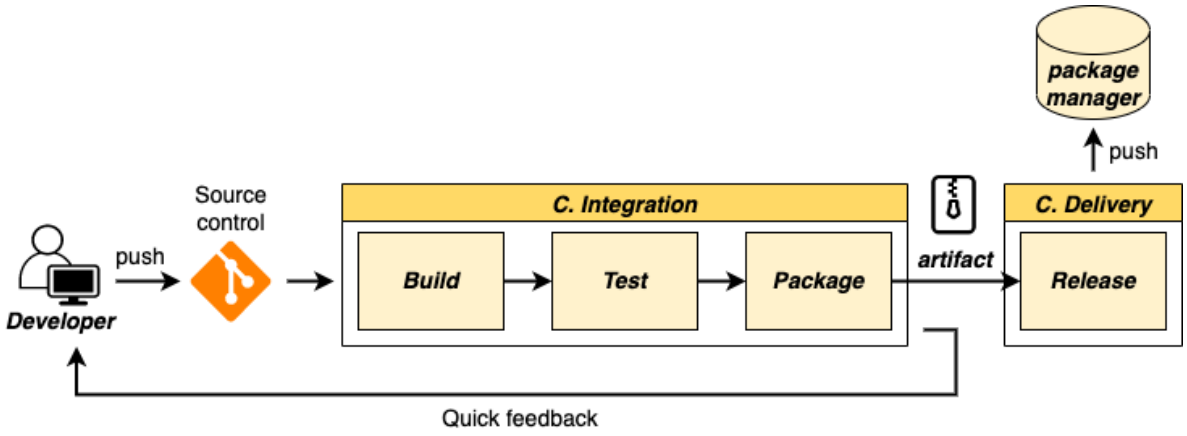

Figura 1.3: Pipeline di Continuous Delivery

I package manager devono fornire lo spazio necessario all'archiviazione dei pacchetti creati durante la fase di CI e pubblicati durante la fase di CD ma devono anche supportare altre funzionalità come il versionamento dei pacchetti, la gestione di diverse tipologie di pacchetto e l'autenticazione per il caricamento/scaricamento dei pacchetti in caso di package manager privati. Tra i più diffusi package manager è possibile indicare certamente Nexus Sonatype[8], Maven Central Repository[9], NPM[10] e PyPi[11].

E' fondamentale che il pacchetto pubblicato sul package manager sia dato come output dalla fase di CI. Questo serve a garantire che il codice sorgente abbia passato con successo tutti i task di controllo e di verifica e che sia lo stesso identico pacchetto ad essere poi utilizzato nella fase successiva d'installazione in un ambiente d'esecuzione.

1.2.3 Continuous Deployment

In base alla natura del software che si sviluppa, il processo potrebbe terminare con la fase di delivery, come ad esempio nel caso di librerie e applicazioni mobile oppure continuare con l'installazione del prodotto in un ambiente d'esecuzione, come nel caso di applicazioni web e microservizi.

Nel caso in cui sia necessaria l'installazione del prodotto in un ambiente, la fase di deployment recupera il pacchetto pubblicato in fase di delivery e lo installa. Infatti, non può esistere continuous deployment senza continuous delivery, ma non è vero il contrario.

[8]https://www.sonatype.com/products/nexus-repository
[9]https://mvnrepository.com/repos/central
[10]https://www.npmjs.com/
[11]https://pypi.org/

Con Continuous Deployment si intende quindi l'estensione della continuous delivery con un processo che automatizza l'intera pipeline dal momento in cui lo sviluppatore modifica il codice all'installazione di quella modifica nell'ambiente di staging e/o di produzione [7].

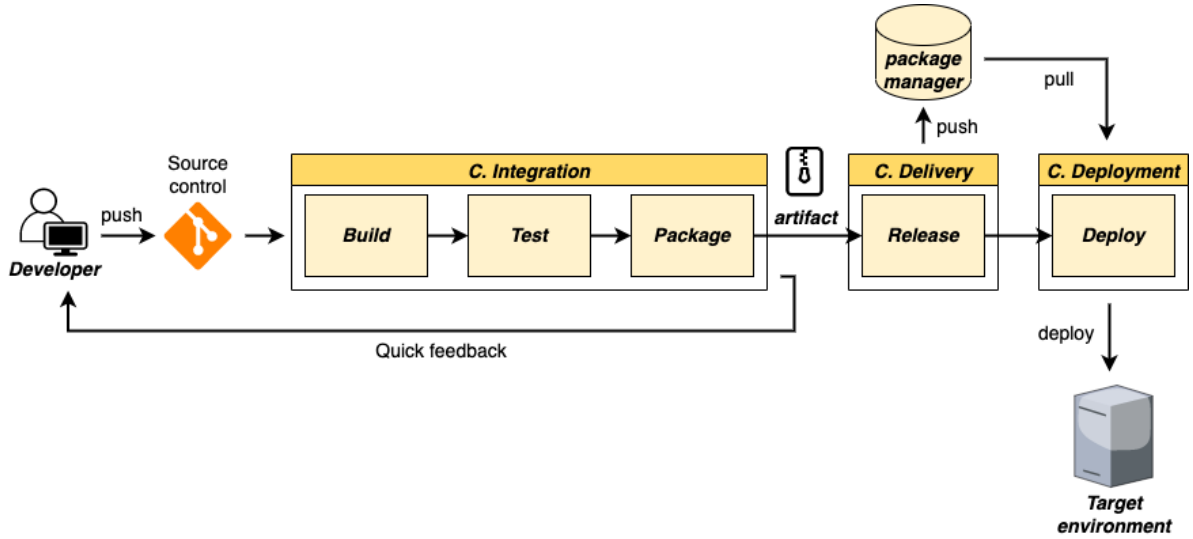

Figura 1.4: Pipeline di Continuous Deployment

1.2.4 Continuous Monitoring

Poter raccogliere dati dal sistema per poterlo analizzare è uno dei punti chiave della cultura DevOps poiché permette di migliorare sia il prodotto che il processo garantendo integrità, prestazioni e affidabilità in ogni fase del ciclo di vita del software.

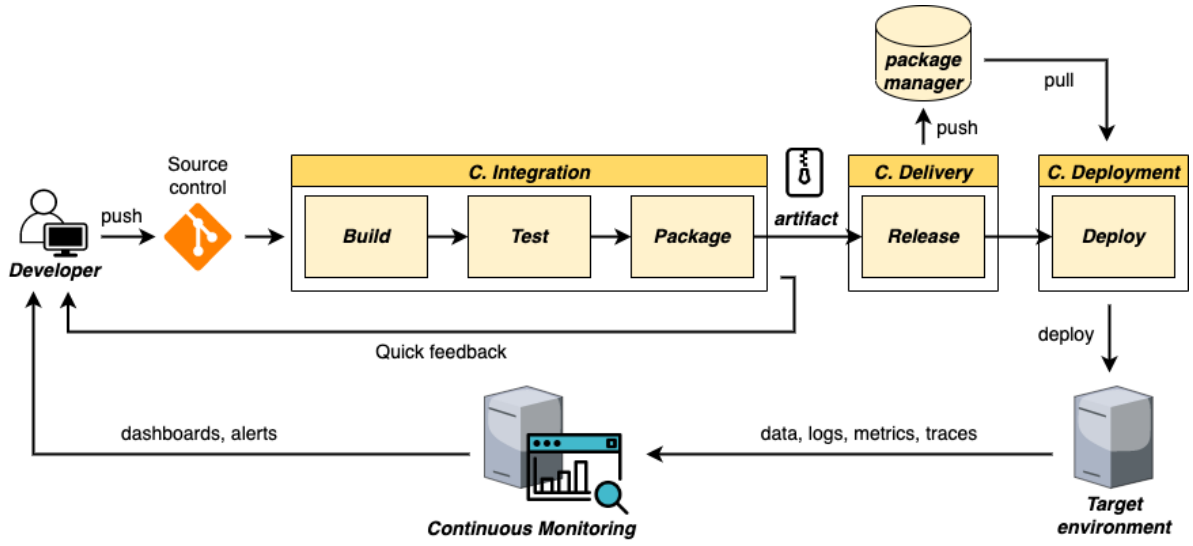

Figura 1.5: Integrazione tipica di un sistema di Continuous Monitoring

Anche la fase di monitoring deve essere continua e automatica, ma a differenza della CI/CD, le pratiche di monitoraggio eseguono task che riguardano tutto il processo indipendentemente dal lavoro svolto dallo sviluppatore. Alcuni task che vengono tipicamente svolti in questa fase del processo sono:

- raccolta di metriche riguardanti il processo di sviluppo (commit effettuate, merge request chiuse e approvate, issues, pipeline fallite, ...),
- raccolta di metriche riguardanti il software rilasciato (numero di download effettuati, recensioni, ...),
- raccolta di metriche riguardanti l'ambiente d'esecuzione del software installato (log, metriche, traces),
- centralizzazione di tutti i dati raccolti,
- invio di notifiche (alerting) al raggiungimento di una certa soglia da parte di una specifica metrica,
- creazione di dashboard per la visualizzazione grafica.

Lo scopo principale della pratica di Continuous Monitoring è ridurre il più possibile il tempo che intercorre tra il momento in cui un difetto è introdotto nel codice e/o nel processo e il momento in cui quel difetto viene intercettato e risolto.

Le stesse pratiche adottate in questa fase possono essere applicate anche al monitoraggio dell'utente, con l'obiettivo di raccogliere dati sul suo comportamento per migliorare l'esperienza utente, fare analisi di mercato o creare raccomandazioni. Solitamente ci si riferisce a questa pratica con il termine "*Analytics*".

1.2.5 Continuous Inspection

Tipicamente vengono definiti a livello aziendale degli standard di qualità e di sicurezza che devono essere rispettati da ogni team di sviluppo per qualsiasi tipologia di prodotto software. La definizione di linee guida sullo stile di programmazione o sul livello di *severity* accettato sono esempi di standard di qualità. La pratica di analizzare automaticamente e molto frequentemente il codice al fine di validare il rispetto degli standard di qualità e sicurezza è chiamata Continuous Inspection.

L'obiettivo è quello di ottenere un feedback sullo stato del codice, sullo stato del processo e sulla presenza di problemi di sicurezza al fine di pianificare interventi risolutivi nel minor tempo possibile. Come nel caso del monitoraggio, esiste un sistema terzo a supporto dei task necessari e in grado di fornire feedback allo sviluppatore. I principali task di analisi che compongono la fase d'ispezione continua sono:

- **Static Application Security Testing** (SAST) - Analisi *white-box* dell'applicazione al fine di individuare vulnerabilità, code smell e bug ma anche di verificare il rispetto di un certo livello di qualità.
- **Software Composition Analysis** (SCA) - Analisi delle dipendenze di progetto con lo scopo di individuare vulnerabilità pubbliche (CVE[12]) associate ad esse.

[12]Common Vulnerability and Exposure

In entrambe le tipologie di analisi è fondamentale ottenere come risultato un report della scansione, in grado di descrivere in modo dettagliato eventuali entità individuate dalla fase di analisi. I report vengono prodotti tipicamente sia in formati *human-readable* (es. pagina HTML) che in formati *machine-readable* (es. JSON, XML) per poter essere utilizzati da applicazioni terze come ad esempio sistemi centralizzati per la consultazione di report eterogenei, chiamati *Vulnerability Management System*.

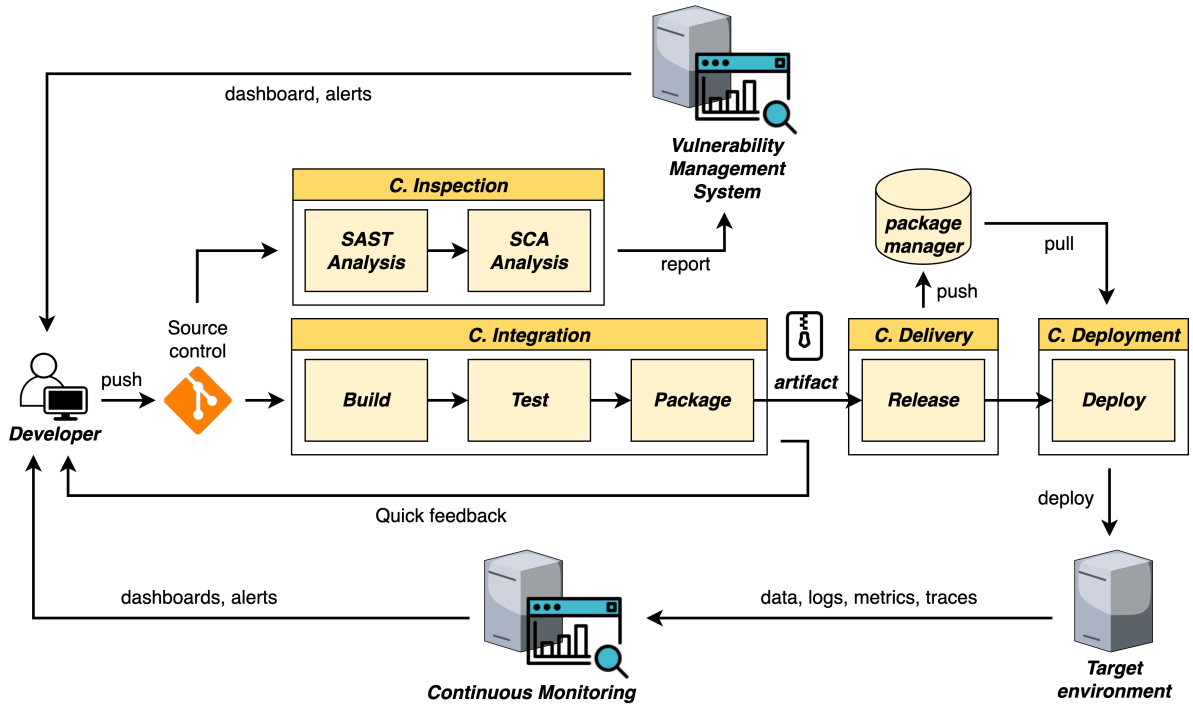

Figura 1.6: Integrazione tipica di un sistema di Continuous Inspection

1.3 Strumenti

Per poter mettere in pratica le tecniche di CI/CD in modo da adottarle efficacemente nel processo di sviluppo è necessaria un'infrastruttura complessa, composta da tanti servizi e strumenti. Data la grande complessità derivante soprattutto dalla gestione, dalla manutenzione e dall'integrazione di tutti questi componenti, spesso si fa uso di una o più piattaforme cloud in modo da creare un sistema eterogeneo composto da servizi in grado di fornire i seguenti strumenti fondamentali:

- **Version Control System** - Strumento per il versionamento del codice e tracciamento delle modifiche che permette la collaborazione tra i componenti di un team. Con l'avvento delle metodologie Agile e della cultura DevOps, l'uso di un VCS nei processi è diventato obbligatorio.
- **Package Manager** - Strumento per l'archiviazione e la condivisione del software sviluppato sotto forma di pacchetti, come descritto nella sezione 1.2.2.

- **CI Server** - Strumento in grado di eseguire elaborazioni in modo automatico su un server remoto e restituire il risultato. Rappresenta il motore di tutta l'automazione a supporto delle tecniche CI/CD.

1.3.1 Pipeline as Code

Nella sezione 1.2.1 il concetto di pipeline è stato definito come una sequenza ordinata di elaborazioni che vengono applicate al codice sorgente in modo automatico. In realtà una pipeline di CI/CD è suddivisa in più passi, detti *stage*, i quali contengono una o più elaborazioni, dette *job*.

Lo strumento adottato da quasi tutte le principali piattaforme per la definizione degli stage e dei job che compongono una pipeline è chiamato *Pipeline as Code* e permette d'istruire un CI Server all'esecuzione effettiva dei task. Questo metodo permette di descrivere il processo di una pipeline in un file testuale, tipicamente in formato YAML[13], situato nello stesso repository in cui risiede il codice sorgente.

Il seguente codice[14] contiene la definizione di una pipeline descritta utilizzando la sintassi per la piattaforma specifica GitLab. In questo caso la pipeline è composta da due stage (*build* e *deploy*), ognuno dei quali contiene a sua volta un job. Il cuore del job, ovvero le istruzioni bash che il CI Server deve eseguire, è definito tramite la keyword *script*.

```
1  stages:
2    - build
3    - deploy
4
5  build-job:
6    stage: build
7    script:
8      echo ‘‘Build Job’’
9    rules:
10     - if: $CI_COMMIT_BRANCH =~ /^(dev|test)$/
11       changes: ["*", "src/**/*"]
12
13 deploy-test-job:
14   stage: deploy
15   script:
16     echo ‘‘Deploy Job’’
17   rules:
18     - if: $CI_COMMIT_BRANCH == ‘‘main’’
```

Listing 1: Pipeline d'esempio per la piattaforma GitLab

I vantaggi derivanti dall'utilizzo del meccanismo Pipeline as Code sono:

[13] https://yaml.org/

[14] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/1-metodologia-devops/.gitlab-cy.yml

- versionamento dei file che descrivono le pipeline di CI/CD in modo da tracciare anche le modifiche che vengono apportate al processo automatizzato,
- utilizzo di metodologie simili a quelle utilizzate nello sviluppo software (ereditarietà, incapsulamento, riuso, ...),
- i file che descrivono il processo sono accessibili a tutto il team ed è possibile collaborare alla loro modifica in modo identico allo sviluppo software.

1.3.2 Runners

L'architettura di un CI Server dev'essere costruita in modo robusto e scalabile per poter supportare efficacemente le tecniche CI/CD. Basta un team con un numero minimo di sviluppatori e un numero minimo di progetti ai quali sono applicate le tecniche CI/CD e la metodologia di sviluppo agile per comprendere l'enorme quantità di lavoro a cui un CI Server può essere sottoposto.

Ogni evento che si verifica sulla piattaforma in cui è mantenuto il codice sorgente viene intercettato dal CI Server: se nel file che descrive la pipeline è definito un job in associazione a quello specifico evento, il CI Server invia tutte le informazioni utili per l'esecuzione del job ad un altro componente software, conosciuto in letteratura come *runner*, e rimane in attesa del risultato.

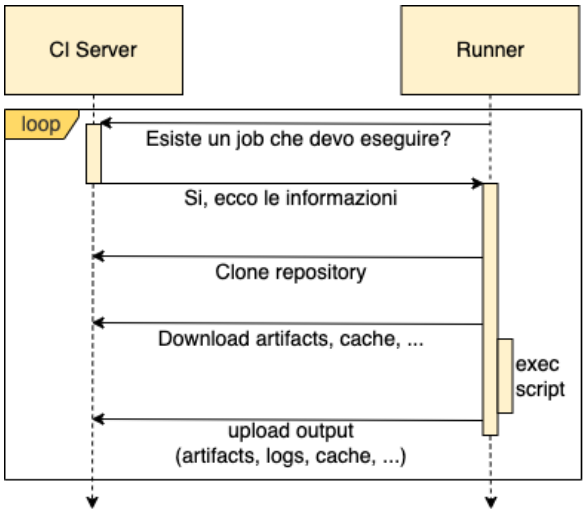

Figura 1.7: Diagramma di sequenza interazione CI Server-Runner

Il runner consiste in un componente architetturale dell'infrastruttura che necessita di un ambiente d'esecuzione e per svolgere il suo compito consuma delle risorse. In base a chi lo gestisce, il runner può essere:

- **Managed** - Funzionalità fornita *as-a-Service* dalla piattaforma che si utilizza. In base al piano di licenza sottoscritto con la piattaforma, l'utilizzo dei runner è soggetto a limiti che riguardano tipicamente (*i*) i minuti d'utilizzo e (*ii*) lo spazio di archiviazione. Il vantaggio di questa modalità è che non è richiesto alcuno sforzo

nella configurazione e nella manutenzione del runner e nessuna spesa per l'ambiente d'esecuzione.
- **Self-Hosted** - Il componente è gestito dall'utilizzatore in tutti i suoi aspetti ma non è soggetto ad alcun tipo di vincolo in termini di risorse utilizzate.

Capitolo 2

Mobile Application Development Lifecycle

2.1 Introduzione

In questo capitolo viene analizzato il processo di sviluppo tipico delle applicazioni mobile al fine di porre le basi per la progettazione dello stesso processo di sviluppo, applicando le pratiche e le tecniche DevOps d'automazione viste nel capitolo precedente. Lo scopo di questa fase iniziale di progetto è quindi la definizione di tutti i principali task e sotto-task necessari allo sviluppo di applicazioni mobile, dalla scrittura del codice sorgente al rilascio sui marketplace delle relative piattaforme target.

A prescindere dalla tipologia di applicazione, il processo di sviluppo è composto concettualmente dalla stessa sequenza di fasi di lavoro: (*i*) pianificazione, (*ii*) progettazione, (*iii*) sviluppo, (*iv*) stabilizzazione, (*v*) rilascio e (*vi*) monitoraggio. Ognuna di queste macro-fasi comprende a sua volta una serie di sotto-fasi tra le quali esistono specifiche dipendenze:

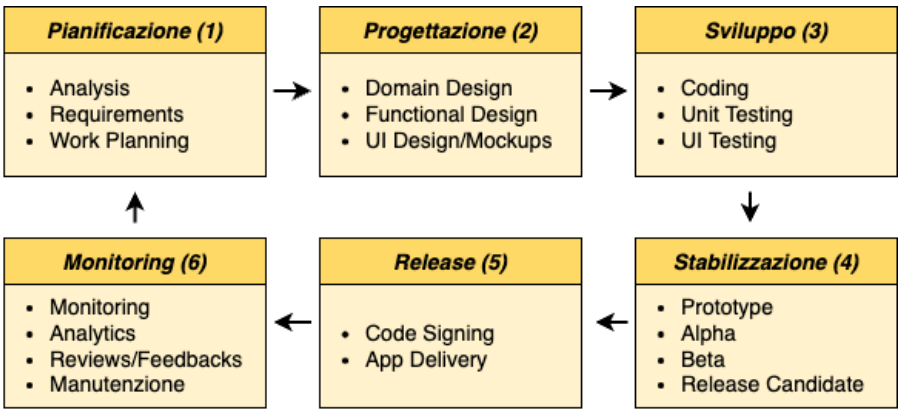

Figura 2.1: Ciclo di vita di sviluppo tipico delle applicazioni mobile

Il processo di sviluppo delle applicazioni mobile è simile a quello di qualsiasi altra applicazione software ma in questo caso l'obiettivo è distribuire l'applicazione al fine di dare la possibilità all'utente finale d'installarla sul proprio dispositivo. Nei processi di sviluppo di altre tipologie di applicazioni, come per esempio le Web Application, l'obiettivo è invece quello di mettere in esecuzione l'applicazione in un ambiente target accessibile dall'utente finale.

2.2 Pianificazione

In questa prima fase del processo di sviluppo si formalizzano i requisiti, funzionali e non funzionali, che devono essere soddisfatti dall'applicazione per ottenere l'approvazione del committente e si pianifica il lavoro per le successive fasi in termini di task, risorse e tempo.

Tramite la definizione di casi d'uso si rappresentano le interazioni tra gli attori all'interno del sistema e i loro ruoli. Questi casi d'uso definiscono dei possibili scenari dove il sistema riceve delle richieste esterne, ad esempio l'input dell'utente, e come esso risponde a quell'input. Dopo aver acquisito un numero appropriato di casi d'uso e attori è molto più semplice iniziare a progettare un'applicazione poiché è possibile concentrarsi su come creare l'applicazione anziché sulla definizione delle sue funzionalità.

2.3 Progettazione

La fase di progettazione dell'applicazione è composta da un insieme di sotto-task tra cui i principali sono (*i*) la modellazione del dominio applicativo, (*ii*) la scelta dell'architettura da utilizzare e (*iii*) la prototipazione dell'esperienza utente e dell'interfaccia grafica (UX/UI[1]).

Tipicamente la progettazione UX/UI viene svolta tramite l'ausilio di mockup per definire prima come l'utente intende utilizzare l'applicazione (esperienza) e poi aspetti grafici come colori, font e icone (interfaccia).

2.4 Sviluppo

Una volta progettata l'applicazione è possibile partire con la fase di sviluppo. Solitamente l'obiettivo è far iniziare la fase di sviluppo il prima possibile in modo da sviluppare un prototipo funzionante, spesso chiamato *Minimum Viable Product* [14], e ottenere la validazione da parte del committente, la quale è l'obiettivo principale della fase successiva di stabilizzazione.

[1]User Experience/User Interface

2.5 Stabilizzazione

La stabilizzazione consiste nella risoluzione di problemi sia a livello funzionale che a livello d'usabilità e di prestazioni al fine d'ottenere una versione di applicazione pronta da distribuire.

Adottare la cultura DevOps e la metodologia Agile significa applicare il concetto "*Release Early, Release Often*": le funzionalità preziose devono essere raggruppate insieme e rilasciate prima per ottenere miglioramenti in termini di valore [15]. Questa parte del ciclo di sviluppo dovrebbe iniziare il più presto possibile in modo da individuare e risolvere i problemi prima che diventino un costo. Tipicamente per qualsiasi applicazione, anche non specifica per i dispositivi mobile, sono previste le seguenti sotto-fasi del processo di stabilizzazione[2]:

- **Prototype** - L'applicazione include soltanto alcune delle funzionalità principali e sono presenti bug maggiori. In questa fase il focus è sulla singola funzionalità implementata fornita dal prototipo per il testing.
- **Alpha** - Tutte le principali funzionalità sono completate e devono essere testate.
- **Beta** - Gran parte delle funzionalità, sia principali che ausiliarie, sono state completate e i bug maggiori sono stati risolti.
- **Release Candidate** - Tutte le funzionalità sono state completate e testate, ma potrebbero essere presenti ancora bug minori.

2.5.1 Alpha

La prima versione funzionante di un'applicazione è detta *alpha* ed è utilizzata per il testing interno di specifiche funzionalità. Questo significa che può presentare anche bug o funzionalità mancanti ma deve contenere almeno le funzionalità che devono essere testate per quella specifica versione.

Solitamente prima della release di un'applicazione, anche se in fase di testing, è necessario attendere la sua approvazione da parte del gestore del servizio: tale processo d'approvazione pre-release è detto *App Review*. Nel caso del testing interno è possibile distribuire l'applicazione ad un insieme ristretto di tester. Continuando con i rilasci di versioni alpha vengono aggiunte nuove funzionalità e/o risolti eventuali bug: quando la versione è considerata pronta viene eseguita la sua promozione, ovvero il rilascio di una versione alpha in versione beta.

2.5.2 Beta

A questo punto del processo di sviluppo l'applicazione è considerata completa a tutti gli effetti, a meno di bug e/o problemi di stabilità. La versione *beta* rappresenta dunque la prima versione dell'applicazione resa disponibile ai tester esterni, ovvero quegli utenti che non hanno partecipato alle fasi di sviluppo e che svolgono il ruolo di validazione delle funzionalità. Si distinguono due tipologie di beta testing:

[2]https://docs.microsoft.com/itit/xamarin/cross-platform/get-started/introduction-to-mobile-sdlc

- **Aperto** - L'applicazione è rilasciata per la fase di testing esterno permettendo l'accesso a qualsiasi utente con account da beta tester.
- **Chiuso** - L'accesso all'applicazione di test è limitato ad un insieme ristretto di tester, tipicamente gestiti tramite mailing list o link di condivisione.

Dopo aver ottenuto la validazione da parte dei tester, la quale potrebbe richiedere più iterazioni di sviluppo e rilascio di versioni alpha-beta, anche per la versione beta si effettua la promozione, rilasciando l'applicazione in produzione.

2.6 Release

Dopo che l'applicazione è stata stabilizzata è possibile procedere con la distribuzione. In questa fase l'applicazione viene prima firmata digitalmente utilizzando un certificato protetto da chiave privata e poi pubblicata sullo specifico marketplace della piattaforma target. La firma dell'applicazione permette ai dispositivi, ai mezzi di distribuzione e ai marketplace di sapere quali applicazioni hanno origine dal proprietario di uno specifico certificato e di verificare che il codice non sia stato modificato da quando è stato firmato [11].

2.7 Monitoraggio

La fase di monitoraggio (e manutenzione) è quella più lunga e dispendiosa in termini di tempo e risorse. Per le applicazioni mobile esistono alcune situazioni che rendono il monitoraggio più complesso rispetto ad altre tipologie di applicazioni, come ad esempio le Web Application. Bisogna infatti considerare che[3]:

- le applicazioni mobile eseguono su una vasta gamma di dispositivi con caratteristiche diverse e può essere quindi difficile ottenere una chiara visibilità delle prestazioni lato client,
- se gli utenti riscontrano un problema, mentre una patch per applicazioni web può essere distribuita quasi all'istante, la distribuzione degli aggiornamenti delle applicazioni mobile richiede tempo e soprattutto l'attivazione da parte degli utenti per scaricarli.

Per effettuare un monitoraggio efficace è necessario misurare e controllare continuamente le performance dell'applicazione mobile, il comportamento dell'utente e gli errori che essi riscontrano. Alcuni esempi di metriche fondamentali sono: tempo d'avvio dell'applicazione, network performance, utilizzo delle risorse (CPU, disco e memoria) e metriche custom derivanti dalle azioni dell'utente.

[3]https://www.datadoghq.com/blog/mobile-monitoring-best-practices/

Capitolo 3

Applicazioni Multipiattaforma

3.1 Introduzione

La continua crescita della quantità di dispositivi mobile in circolazione ha reso molto importante a livello globale il mercato delle applicazioni mobile e per questo motivo sono sempre di più le aziende che decidono di investire risorse nello sviluppo e nella vendita di applicazioni mobile. Un'azienda che intende puntare al maggior numero di utenti possibile con le proprie applicazioni deve considerare che l'intero mercato è spartito in base alla diffusione dei sistemi operativi per dispositivi mobile.

Ad oggi, i sistemi operativi più diffusi sono Android (Google) e iOS (Apple), i quali coprono quasi la totalità del mercato con quote rispettivamente del 71% e del 28%[1]. Questi dati comportano per un'azienda la necessità di sviluppare la stessa applicazione per almeno due piattaforme completamente differenti tra loro. A tal proposito sono nate nuove metodologie e tecniche basate sul concetto "*Write Once, Run Anywhere*"[2] (WORA) con lo scopo di ottimizzare lo sviluppo delle applicazioni mobile al fine di ridurre i costi e aumentare l'efficienza del processo di sviluppo.

Le principali tecniche moderne di sviluppo per applicazioni mobile che seguono questi concetti sono:

- **Cross-platform** - Rispetta completamente la filosofia WORA. Lo stesso codice può essere eseguito su diverse piattaforme grazie ad uno strato applicativo aggiuntivo che si occupa d'interpretare il codice e tradurlo nel linguaggio specifico della piattaforma target.
- **Multi-platform** - Tecnica più recente che permette di sviluppare applicazioni native condividendo solamente la logica applicativa. In questo caso non è necessario uno strato software aggiuntivo perché l'applicazione può essere eseguita direttamente

[1] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[2] https://www.computerweekly.com/feature/Write-once-run-anywhere

dalla piattaforma target.

3.2 Cross-platform vs Multi-platform

Sia nel caso cross-platform che nel caso multi-platform i principali vantaggi, che sono la riduzione dei costi e l'ottimizzazione del processo di sviluppo, derivano dalla condivisione e dal riuso del codice e quindi meno risorse impiegate rispetto allo sviluppo classico delle applicazioni mobile native.

Esistono però alcune differenze tra loro, fondamentali durante la scelta della metodologia da adottare da parte di un'azienda per lo sviluppo di un'applicazione mobile.

- **Cross-platform**
  - Condivisione/riuso totale del codice. Sia la logica applicativa che l'interfaccia utente sono le stesse per qualsiasi piattaforma.
  - Performance limitate rispetto al nativo, dovute alla presenza di uno strato software aggiuntivo che interpreta e traduce il codice.
  - Accesso alle funzionalità hardware del dispositivo limitato e/o con overhead, dovuto sempre alla presenza dello strato software aggiuntivo.
- **Multi-platform**
  - Condivisione/riuso della sola logica applicativa. Lo sviluppo dell'interfaccia utente rimane nativo.
  - Performance elevate, equivalenti a quelle native.
  - Accesso completo e senza overhead a tutte le funzionalità hardware del dispositivo.

3.3 Strumenti

La scelta degli strumenti da utilizzare nel processo di sviluppo è influenzata da tantissimi fattori come ad esempio la necessità di una licenza commerciale, i gusti dello sviluppatore e certamente le tecnologie coinvolte. A prescindere da questi fattori, è possibile definire delle categorie di strumenti, utilizzati per qualsiasi tipologia di software: (*i*) linguaggio di programmazione, (*ii*) ambiente di sviluppo (IDE[3]), (*iii*) build automation, (*iv*) distribuzione software e (*v*) framework di sviluppo.

Un altro fattore che vincola la scelta degli strumenti è la piattaforma target, ovvero l'ambiente dove eseguirà il codice, e lo è in particolare per le applicazioni mobile. A differenza dello sviluppo di applicazioni Android, dove gran parte degli strumenti più diffusi è open-source, lo sviluppo di applicazioni iOS è soggetto a vincoli stringenti imposti da Apple: tutta la toolchain necessaria è disponibile solamente per il sistema operativo macOS.

[3]Integrated Development Environment

Linguaggio di programmazione

Una moderna applicazione nativa per iOS è sviluppata in codice Swift[4], un linguaggio di programmazione open-source progettato da Apple e realizzato per sostituire il linguaggio Objective-C [6].

Come nel caso delle applicazioni iOS, anche per quelle Android è stato adottato un nuovo linguaggio per sostituire quello precedentemente utilizzato. Il linguaggio di programmazione ufficiale per lo sviluppo di applicazioni Android è il linguaggio open-source creato da JetBrains chiamato Kotlin[5]. Questo linguaggio ha sostituito Java grazie alla sua interoperabilità con quest'ultimo: il codice Kotlin può infatti essere compilato in bytecode Java e quindi essere eseguito ovunque può eseguire una JVM[6] [8].

Ambiente di sviluppo

L'ambiente di sviluppo consiste in un programma software a supporto dello sviluppatore per la fase di scrittura del codice. Un IDE consiste di più componenti: oltre all'editor principale infatti solitamente comprende tool di build automation, debugger, compilatore/interprete e emulatori.

Per poter sviluppare ed eseguire un'applicazione iOS, l'unico IDE stabile è XCode, disponibile solamente per macOS. Per lo sviluppo di applicazioni Android invece esistono diversi IDE ma sicuramente quello più diffuso è Android Studio: un'ambiente di sviluppo open-source costruito sul robusto e popolare IntelliJ di JetBrains, distribuito con tutte le funzionalità necessarie per lo sviluppo Android già preinstallate.

Build automation

Con build automation si intende l'automazione di alcuni task del processo di sviluppo come la compilazione del codice, l'esecuzione dei test, la firma del codice e la pacchettizzazione. Gli strumenti di build automation sono nati dunque per supportare lo sviluppatore permettendogli di automatizzare operazioni semplici e ripetibili. Tipicamente un task viene descritto per il raggiungimento di un *goal* tramite un elenco ordinato di sotto-task dipendenti tra loro.

La build automation per lo sviluppo di applicazioni Android è supportata dal tool Gradle[7], uno strumento open-source per build automation e dependency management. Gradle fornisce il proprio DSL[8], disponibile sia in Groovy che Kotlin, per la definizione degli script di build [13].

In ambiente Apple è necessario installare il pacchetto "*Command Line Tools for XCode*", il quale contiene un insieme di strumenti di sviluppo tra cui Swift Package Manager,

[4]https://www.swift.org/
[5]https://kotlinlang.org/
[6]Java Virtual Machine
[7]https://gradle.org/
[8]Domain Specific Language

dedicato alla build automation e alla dependency management. Anche se disponibile tra le funzionalità del tool ufficiale Apple, la gestione delle dipendenze tipicamente viene svolta tramite un altro strumento molto diffuso chiamato CocoaPods[9].

Distribuzione software

In questo caso è più corretto parlare di servizi di distribuzione software invece che di strumenti. Gli strumenti utilizzati dallo sviluppatore corrispondono infatti a quelli di build automation con lo scopo di firmare ed effettuare l'upload dell'applicazione su una piattaforma apposita per la distribuzione. I servizi forniti da queste piattaforme vengono utilizzati sia durante la fase di stabilizzazione del software, per svolgere testing interno ed esterno (alpha/beta), che durante la fase di pubblicazione effettiva delle applicazioni sui marketplace delle piattaforme target.

La piattaforma Google Play Console permette di gestire tutti i task correlati alla stabilizzazione e alla pubblicazione sul marketplace Google Play Store di applicazioni Android. Per le applicazioni iOS è necessario invece utilizzare più strumenti per le specifiche attività: App Store Connect per la pubblicazione sul marketplace App Store e Testflight per la stabilizzazione.

Framework di sviluppo

E' in questa tipologia di strumenti che si collocano tutti i framework a supporto dello sviluppo di applicazioni multipiattaforma. I più popolari framework open-source per lo sviluppo di applicazioni cross-platform sono: (*i*) Ionic[10], (*ii*) Flutter[11] e (*iii*) React Native[12]. Il paradigma multi-platform è più recente rispetto a quello cross-platform e il principale framework open-source in questo caso è Kotlin Multiplatform.

3.4 Kotlin Multiplatform Mobile

Kotlin Multiplatform Mobile[13] (KMM) è un framework per lo sviluppo di applicazioni Android e iOS, basato sul concetto di condivisione della logica applicativa mantenendo lo sviluppo nativo della UX/UI. Con il rilascio di Kotlin 1.7.20 (Settembre 2022), KMM è passato dalla fase Alpha alla fase Beta, la quale è considerata come fase "*pre-stable*"[14] nonostante sia già stato adottato in produzione per lo sviluppo delle proprie applicazioni mobile da tante aziende tra le quali è possibile trovare nomi rilevanti come Netflix, VMware e Philips[15]. In base al risultato dell'indagine di mercato svolta nei primi due quadrimestri del 2021[16], le porzioni di codice condiviso nelle applicazio-

[9] https://cocoapods.org/
[10] https://ionicframework.com/
[11] https://flutter.dev/
[12] https://reactnative.dev/
[13] https://kotlinlang.org/lp/mobile/
[14] https://kotlinlang.org/docs/components-stability.html#current-stability-of-kotlin-components
[15] https://kotlinlang.org/lp/mobile/case-studies/
[16] https://blog.jetbrains.com/kotlin/2021/10/multiplatform-survey-q1-q2-2021/

ni sviluppate con KMM sono: 85% networking, 75% data storage, 70% utilities, ∼60% algoritmi/computazione, ∼55% state management e ∼50% presenters/controllers/view models.

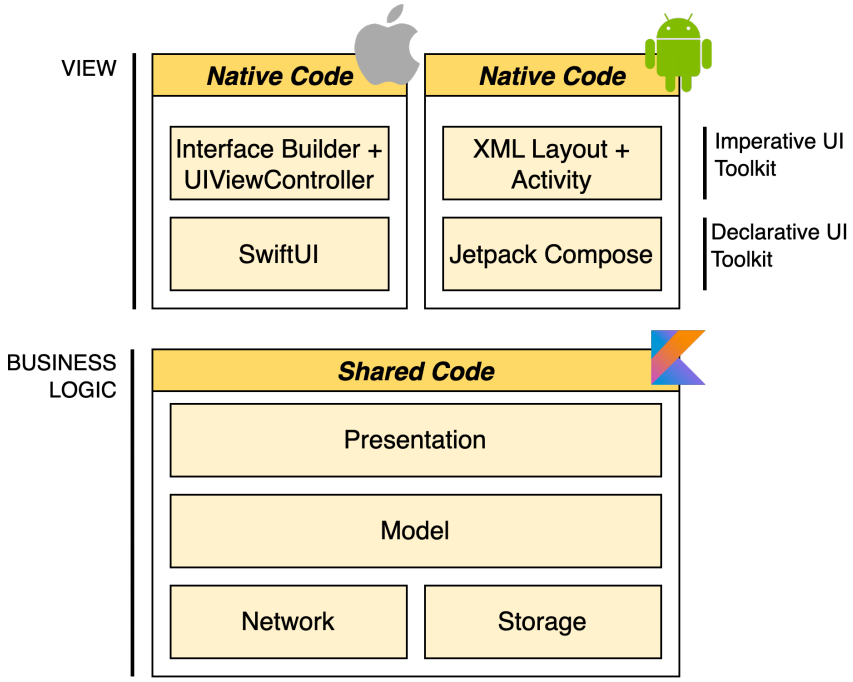

Figura 3.1: Stack architetturale Kotlin Multiplatform Mobile

KMM consiste in un caso d'uso specifico (e il più diffuso) del framework Kotlin Multi-Platform (KMP), il quale permette di sviluppare il codice in modo agnostico rispetto le piattaforme target e di condividerlo tra loro. Il framework KMM per poter svolgere la sua principale funzionalità di condivisione di codice si basa fortemente sui seguenti compilatori inclusi nell'ecosistema Kotlin [13]:

- **Kotlin/JVM** - Utilizzato per la piattaforma Android, permette di compilare codice Kotlin in bytecode Java (*.class*), il quale può essere eseguito direttamente sulla JVM. Nel caso di Android è necessario un ulteriore passaggio per tradurre il bytecode Java in bytecode Dalvik (*.dex*).
- **Kotlin/Native** - Utilizzato per la piattaforma iOS. A differenza del compilatore Kotlin/JVM, il compilatore Kotlin/Native è progettato per quelle situazioni dove non è possibile o non si vuole avere una Virtual Machine (VM), come nel caso dei dispositivi embedded e della piattaforma iOS. Per fare ciò include un backend basato su Low Level Virtual Machine (LLVM)[17] in grado di compilare il codice Kotlin in binari nativi che possono essere eseguiti senza VM [13]. Le piattaforme supportate da Kotlin/Native attualmente sono macOS, iOS, tvOS, watchOS, Linux, Windows (MinGW) e Android NDK[18] e per ognuna di esse esistono differenti

[17] https://llvm.org/
[18] https://kotlinlang.org/docs/native-overview.html#target-platforms

architetture. Nel caso di iOS le differenti architetture supportate da KMM sono *Arm64*, *Arm32* e *x64*. Anche in questo caso sono necessarie due fasi di compilazione: (*i*) il codice Kotlin viene compilato nella Rappresentazione Intermedia (IR) LLVM e (*ii*) successivamente compilato nel binario nativo.

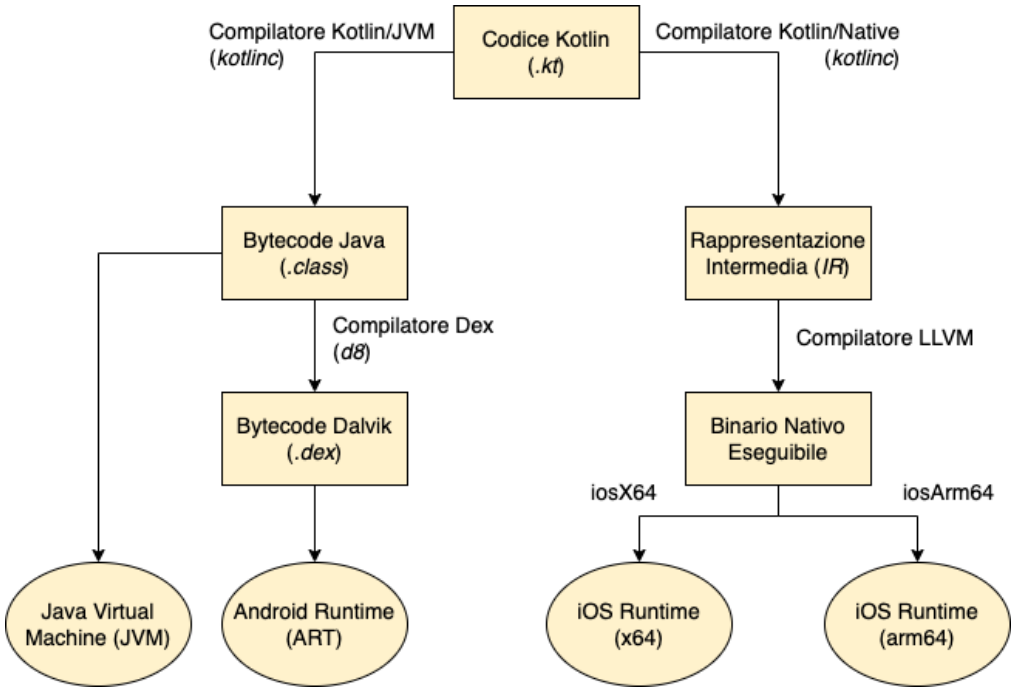

Figura 3.2: Fasi di compilazione Kotlin/JVM e Kotlin/Native

3.4.1 Struttura Applicazione KMM

Un'applicazione sviluppata con KMM segue lo stack definito dalla condivisione della business logic e la separazione della UX/UI (fig. 3.1). Il modulo *shared* contiene tutta la business logic condivisa, la quale può essere sviluppata tramite l'utilizzo di librerie con supporto nativo al framework KMM, cioè librerie che forniscono già al loro interno specifiche implementazioni per le diverse piattaforme target, oppure tramite l'utilizzo del meccanismo "*expect/actual*".

Nel caso di utilizzo del meccanismo expect/actual è necessario definire le funzionalità nel modulo *commonMain* e fornire le implementazioni per le specifiche piattaforme nei relativi moduli *androidMain* e *iosMain*. Gli stessi concetti vengono applicati per la struttura dei moduli di test: *commonTest*, *androidTest* e *iosTest*. I moduli UX/UI delle relative piattaforme includono il codice condiviso come dipendenza di progetto, in particolare come dipendenza Gradle (aar[19]) per Android e come dipendenza CocoaPods (Pod) per iOS.

[19]Android Archive

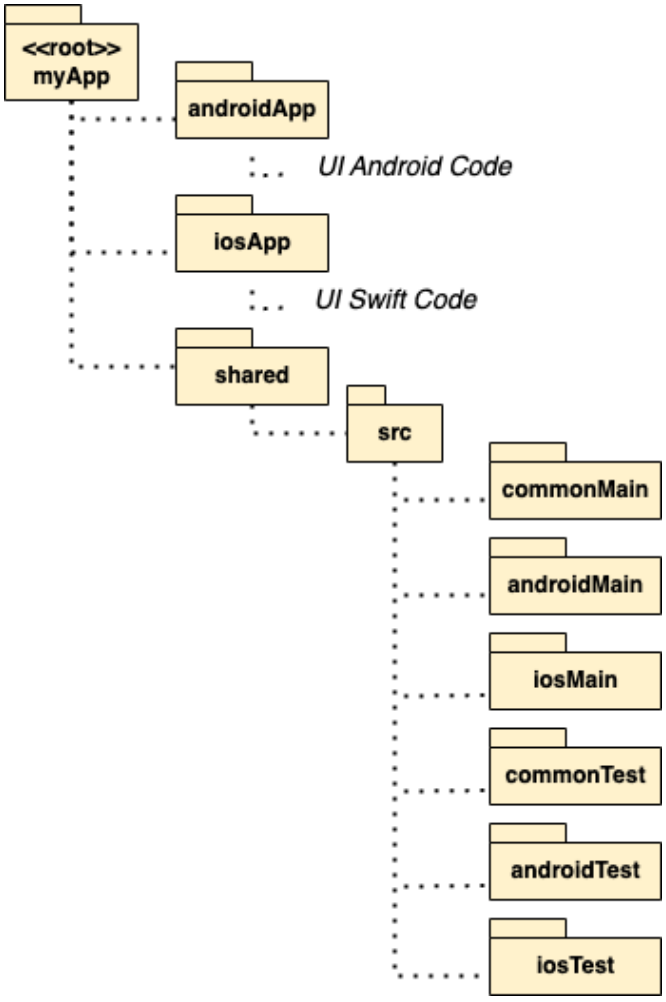

Figura 3.3: Struttura dei moduli di una applicazione KMM

3.4.2 Expect/Actual

Quando si sviluppa codice condiviso è spesso necessario definire come determinate funzionalità debbano essere implementate sulla specifica piattaforma target per utilizzare i relativi SDK[20]. Il framework KMM fornisce il meccanismo expect/actual per assolvere a questo compito in modo del tutto analogo al design pattern Template Method [5]:

- **Expect** - Astrazione della funzionalità necessaria. Tramite la keywork *expect* si definisce lo scheletro astraendo dalla specifica implementazione.
- **Actual** - Implementazione specifica per una determinata piattaforma. Tramite la keywork *actual* si definisce l'implementazione, reificando l'astrazione definita tramite il concetto di *expect*.

Il seguente codice[21] mostra un esempio elementare d'utilizzo del meccanismo descritto:

[20]Software Development Kit

[21]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/3-applicazioni-multipiattaforma/kmm-example/shared

```kotlin
// src/commonMain/kotlin/it/filo/myapplication/Platform.kt
expect class Platform() {
  val platform: String
}

// src/androidMain/kotlin/it/filo/myapplication/Platform.kt
actual class Platform actual constructor() {
  actual val platform: String = "Android ${android.os.Build.VERSION.SDK_INT}"
}

// src/iosMain/kotlin/it/filo/myapplication/Platform.kt
actual class Platform actual constructor() {
  actual val platform: String = UIDevice.currentDevice.systemName() + " " +
    UIDevice.currentDevice.systemVersion
}
```

Listing 2: Esempio di applicazione expect/actual per ottenere informazioni sulla piattaforma

I seguenti screenshot, catturati tramite l'esecuzione degli appositi emulatori delle piattaforme target, mostrano l'efficacia del meccanismo expect/actual: l'applicazione è stata compilata utilizzando le differenti implementazioni in modo corretto.

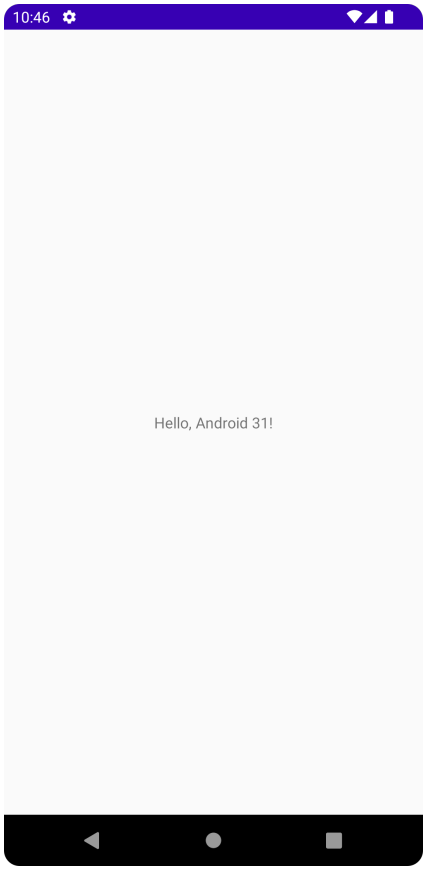

Figura 3.4: Esempio expect/actual in esecuzione sulla piattaforma Android

Figura 3.5: Esempio expect/actual in esecuzione sulla piattaforma iOS

3.4.3 KMM Gradle Plugins

Gli strumenti di build automation permettono la gestione di tutti quei task riguardanti la compilazione del codice. Gradle rappresenta il tool di build automation ufficiale Android ed è possibile sviluppare delle estensioni, chiamate *plugins*, per aggiungere task custom allo strumento. E' proprio tramite questo meccanismo che KMM permette l'esecuzione dei task relativi allo sviluppo di applicazioni mobile multipiattaforma.

Lo strumento di build automation Gradle e i relativi plugins utilizzati vengono configurati tramite specifici file locati insieme ai sorgenti del progetto. Il seguente codice[22] mostra come è possibile configurare il modulo condiviso tramite l'apposito file definendo le specifiche dipendenze e gli specifici task per le piattaforme target:

```
plugins {
  kotlin("multiplatform") // org.jetbrains.kotlin.multiplatform
  id("com.android.library")
  kotlin("native.cocoapods") // org.jetbrains.kotlin.native.cocoapods
}

kotlin {
  android()
  iosArm64()
  cocoapods {  // Configurazione Cocoapods
    ios.deploymentTarget = "14.1"
    podfile = project.file("../iosApp/Podfile")
    framework { baseName = "shared" }
  }

  sourceSets {
    val commonMain by getting { dependencies {} } // Dipendenze comuni Android e iOS
    val androidMain by getting { dependencies {} } // Dipendenze specifiche Android
    val iosMain by creating { dependencies {} } // Dipendenze specifiche iOS
  }
}

android { } // Configurazione android
```

Listing 3: Definizione utilizzo Plugin Gradle KMM nel file *build.gradle.kts* del modulo condiviso (Kotlin)

Il plugin Gradle KMM fornisce uno specifico DSL per definire e configurare i task necessari a compilare il codice condiviso per le relative piattaforme target[23] ma permette anche l'esecuzione di un insieme di task predefiniti, classificati nelle seguenti tipologie:

- **Build** - Tasks per compilazione e linking.
- **CocoaPods** - Tasks per la gestione delle dipendenze Swift/Objective-C.

[22] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/3-applicazioni-multipiattaforma/kmm-example/shared/build.gradle.kts

[23] https://kotlinlang.org/docs/multiplatform-dsl-reference.html

- **Interop** - Tasks relativi all'utilizzo del *commonizer*[24].
- **Verification tasks** - Tasks per l'esecuzione dei test.

3.5 Fastlane

Nonostante vi sia la necessità di utilizzare diversi strumenti di build automation per l'esecuzione dei task di sviluppo delle due differenti piattaforme Android e iOS, il processo di sviluppo e le fasi che lo compongono sono comuni, indipendentemente dalla piattaforma target. Come descritto nei capitoli precedenti, lo sviluppatore deve essere in grado non solo d'eseguire task strettamente correlati alla fase di sviluppo ma anche di eseguire tutti quei task che riguardano il testing, la stabilizzazione e il rilascio di un'applicazione.

Tra gli strumenti open-source dedicati allo sviluppo di applicazioni multipiattaforma uno dei più popolari è sicuramente Fastlane[25]. Il punto di forza di questo tool è il supporto a tutte le fasi del processo di sviluppo di applicazioni mobile per entrambe le piattaforme Android e iOS, il quale pone lo sviluppatore nella condizione di poter operare su tutto il processo tramite un unico tool.

I task delle varie fasi del processo definiscono il comportamento di Fastlane e devono essere descritti negli appositi file di configurazione, in particolare *Fastfile* e *Appfile*, utilizzando due concetti principali:

- **Action** - Elaborazione predefinita e configurabile tramite passaggio di parametri, la quale rappresenta il task che deve essere eseguito.
- **Lane** - Insieme di action definito dall'utente per descrivere elaborazioni complesse, ovvero task composti da più sotto-task.

Il seguente codice[26] rappresenta un esempio di lane Fastlane per il rilascio in versione beta di applicazioni iOS. Tale lane, chiamata *beta*, è composta da quattro action rispettivamente per: (*i*) inizializzare l'ambiente per connettersi ai servizi cloud Apple e scaricare tutto il necessario per la fase di firma del codice, (*ii*) compilazione del codice Swift e creazione del pacchetto *ipa* contenente l'applicazione iOS, (*iii*) pubblicazione in versione beta su Testflight e (*iv*) invio della notifica dell'avvenuta pubblicazione su un canale Slack dedicato.

```
1 lane :beta do # lane
2   sync_code_signing(type: "appstore") # action
3   build_app(scheme: "MyApp") # action
4   upload_to_testflight # action
5   slack(message: "Successfully distributed a new beta build") # action
6 end
```

Listing 4: Esempio di lane Fastlane per il rilascio applicazioni iOS in beta

[24]https://github.com/JetBrains/kotlin/tree/master/native/commonizer
[25]7https://github.com/fastlane/fastlane
[26]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/3-applicazioni-multipiattaforma/fastlane/fastlane/Fastfile

Capitolo 4

Caso di studio: applicazione MaggioliEbook

4.1 Introduzione

In questo capitolo vengono inizialmente descritte le motivazioni che hanno spinto l'azienda Maggioli S.p.A.[1] a dedicare risorse per la ricerca e la sperimentazione nei campi delle applicazioni multipiattaforma e delle tecniche DevOps in ambito mobile. Successivamente sono indicati i requisiti del caso di studio industriale individuato, il quale può essere suddiviso in due macroaree:

- definizione del processo di sviluppo per applicazioni multipiattaforma tramite l'adozione della cultura DevOps,
- sviluppo di un'applicazione mobile multipiattaforma utilizzando il processo di sviluppo definito.

4.2 Contesto aziendale

Tra i core business dell'azienda Maggioli S.p.A. è rimasto centrale il ruolo dell'editoria, ma col trascorrere degli anni e il mutare delle esigenze dei clienti, i quali sono principalmente pubblica amministrazione (PA) e professionisti privati, come avvocati, architetti, commercialisti ed ingegneri edili, si è verificata una transizione verso il mondo digitale.

I servizi digitali erogati per la consultazione delle pubblicazioni hanno superato considerevolmente il formato cartaceo, il quale rimane comunque un metodo secondario di consultazione disponibile seppur in forma molto ridotta. Per l'editoria digitale esiste in Maggioli una business unit dedicata, chiamata *Digital Media*, il cui ruolo principale consiste nella realizzazione e manutenzione dei siti Web Maggioli dedicati alla ricerca e visualizzazione delle pubblicazioni digitali. I seguenti sono soltanto alcuni dei siti gesti-

[1] https://www.maggioli.com/en-us

ti dal team *Digital Media*: Biblioteca Digitale[2], Appalti & Contratti[3] e Periodici[4]. La necessità principale è dunque quella di fare innovazione tramite lo sviluppo di un'applicazione mobile in modo da fornire ai clienti Maggioli un nuovo metodo di accesso alle pubblicazioni che sia più accessibile e comodo.

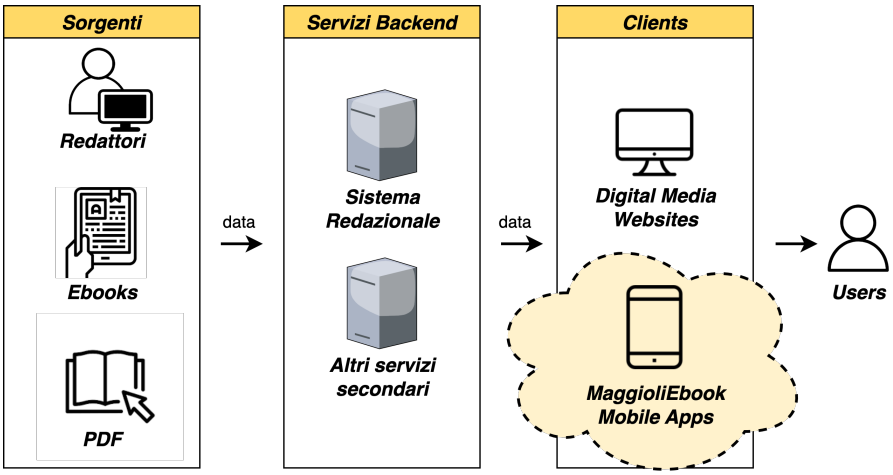

Figura 4.1: Schema del contesto aziendale in cui è collocato il caso di studio

Le motivazioni che stanno alla base della scelta della cultura DevOps e delle applicazioni multipiattaforma per lo sviluppo di questo caso di studio sono comuni a qualsiasi tipologia d'azienda: come descritto nei capitoli 1 e 3 è possibile ottimizzare il processo di sviluppo diminuendo le risorse impiegate e quindi i costi ma allo stesso tempo aumentando la qualità del software e la frequenza di rilascio, i quali comportano una maggiore soddisfazione sia da parte dell'utente finale che da parte dell'azienda.

4.3 Definizione processo di sviluppo

Considerando l'intero contesto aziendale esistono degli standard, consolidati con l'esperienza maturata nello sviluppo software, che devono essere adottati sia per quanto riguarda il processo di sviluppo che la scelta degli strumenti necessari. L'obiettivo è riuscire a definire un modello di processo fortemente basato sugli standard aziendali ma adattato alle esigenze dello sviluppo di applicazioni mobile multipiattaforma e che possa essere introdotto nei team che si occupano di applicazioni mobile. I principali standard aziendali riguardanti gli strumenti e le tecnologie che devono essere adottati sono:

- **GitLab**[5] (*DVCS/CI Server*) - Le funzionalità necessarie al versionamento del codice, alla collaborazione/pianificazione e all'automazione (Sezione 1.3) sono tutte

[2]https://bibliotecadigitale.maggioli.it/
[3]https://www.appaltiecontratti.it/
[4]https://www.periodicimaggioli.it/
[5]https://about.gitlab.com/

soddisfatte dalla piattaforma cloud GitLab con la quale è presente una sottoscrizione di piano di licenza aziendale.
- **SonarQube**[6] (*Vulnerability Management System*) - Questo servizio self-hosted è reso disponibile a tutti i team aziendali e permette di soddisfare gran parte dei task di Continuous Inspection. Grazie a SonarQube è possibile effettuare l'analisi statica del codice, validare il rispetto di determinate policy aziendali e visualizzare dashboard sullo stato delle scansioni.
- **Renovate**[7] (*Dependency Management*) - Strumento utilizzato per automatizzare la gestione delle dipendenze dei progetti.

La pipeline standard a livello aziendale, base di partenza per la definizione della pipeline per applicazioni mobile multipiattaforma, rispetta a pieno tutte le considerazioni fatte nel capitolo 1 per tutte le tecniche d'automazione indicate (fig. 1.6). Tramite l'adattamento delle fasi di integration, delivery e inspection di questa pipeline con le necessità dello sviluppo mobile indicate nei capitoli 2 e 3 si ottengono i seguenti requisiti e dunque la pipeline obiettivo da realizzare.

4.3.1 Requisiti

- **R1** - Continuous Integration
  - **R1.1** - Stage di compilazione, testing e packaging con i relativi sotto-task per entrambe le piattaforme Android e iOS.
  - **R1.2** - L'output dell'ultimo stage deve essere un pacchetto contenente l'applicazione da passare come artefatto alla fase successiva di delivery.
  - **R1.3** - Utilizzo del sistema aziendale di gestione automatica delle dipendenze Renovate.
- **R2** - Continuous Delivery
  - **R2.1** - Stage di stabilizzazione suddivisi tra *Alpha* e *Beta* per il testing interno ed esterno tramite gli appositi servizi forniti da Google e Apple: Google Play Console per l'applicazione Android e Testflight per l'applicazione iOS.
  - **R2.2** - L'applicazione soggetta a stabilizzazione è data in input dalla fase precedente di integrazione.
  - **R2.3** - Stage di pubblicazione sui relativi marketplace Google Play Store o App Store. L'applicazione soggetta a pubblicazione deriva dalla terminazione con successo della fase di stabilizzazione e si trova già sui portali utilizzati.
- **R3** - Continuous Inspection
  - **R3.1** - Stage di analisi statica e analisi delle dipendenze per entrambe le piattaforme.
  - **R3.2** - Utilizzo del sistema aziendale di gestione delle vulnerabilità centralizzato SonarQube.
- **R4** - Tutto il sistema di automazione deve poter essere utilizzato da altri team di sviluppo all'interno dell'azienda.

[6] https://www.sonarqube.org/
[7] https://docs.renovatebot.com/

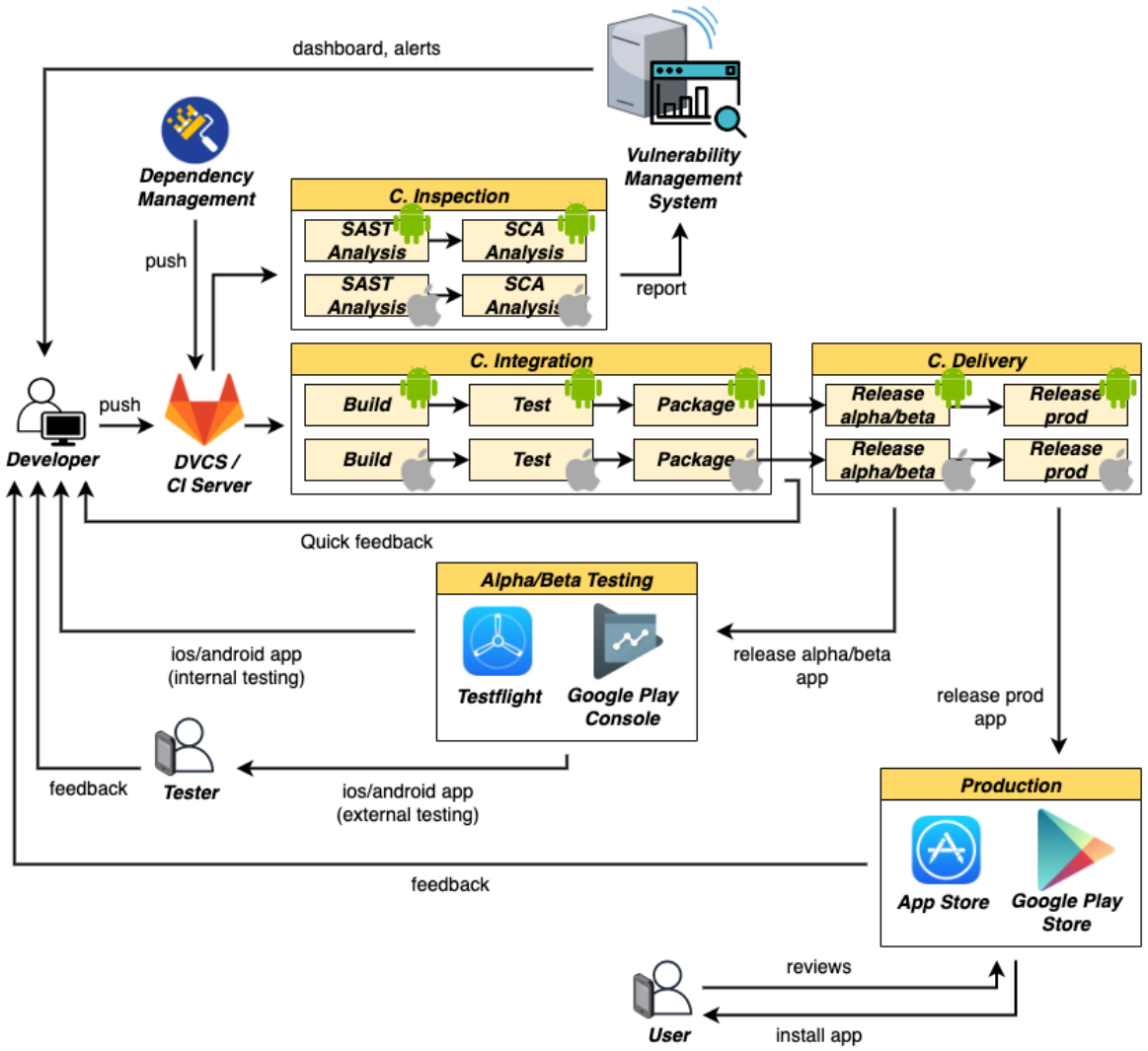

Figura 4.2: Schema globale del processo di sviluppo automatizzato che si intende realizzare

4.4 Definizione applicazione

Dato il contesto aziendale dell'editoria digitale in cui si colloca il caso di studio, è possibile catalogare l'applicazione mobile da sviluppare come *E-Reader*. Un e-book reader, chiamato anche e-book device o e-reader, è un dispositivo elettronico mobile progettato principalmente per la lettura di e-book e periodici digitali. Ogni dispositivo in grado di mostrare del testo su uno schermo potrebbe essere considerato un e-reader, ma i dispositivi progettati appositamente per questo compito hanno caratteristiche e funzionalità specifiche come l'ottimizzazione della portabilità, la leggibilità e la durata della batteria [12].

Per poter avere un valore aggiunto rispetto alle funzionalità già fornite dai siti Web sviluppati dal team *Digital Media*, l'applicazione "MaggioliEbook" deve comportarsi come

un vero e proprio e-reader, integrandosi ai servizi backend Maggioli per l'editoria digitale già esistenti. Data la complessità del dominio da modellare sono stati svolti degli incontri con figure interne esperte di dominio appartenenti ai team *Ricerca e Sviluppo* e *Digital Media*, al fine di definire (*i*) terminologia, (*ii*) casi d'uso e (*iii*) requisiti.

4.4.1 Terminologia e casi d'uso

E' necessario che il team di sviluppo, composto da figure tecniche, e i committenti, i quali sono invece esperti interni di dominio, utilizzino lo stesso linguaggio per far si che le successive fasi del processo siano efficaci ed efficienti.

Il concetto di *Ubiquitous Language* definisce un vocabolario condiviso da entrambe le parti per la discussione del software [4] ed è stato adottato per la realizzazione del seguente glossario, il quale racchiude tutti i principali termini utilizzati negli incontri tra team di sviluppo ed esperti di dominio, suddivisi in *entità* e *casi d'uso*:

Entità

Termine	Descrizione
Reader	Lettore di documenti in grado di visualizzarli ed interagire con essi.
Documento	Contenuto digitale pubblicato da Maggioli Editore.
Documento Statico	Documento con una certa struttura definita (PDF).
Documento Fluido	Documento senza struttura in grado di adattarsi al dispositivo in cui viene aperto (EPUB).
Libro	Tipologia principale di documento fluido fruibile tramite l'applicazione MaggioliEbook.
Rivista	Tipologia principale di documento statico fruibile tramite l'applicazione MaggioliEbook.
Bookmark	Identifica una specifica pagina di un libro o di una rivista.
Progression	Progresso di lettura di un libro o di una rivista, calcolato in percentuale (numero di pagine lette sul totale del documento).
Highlight	Annotazione per una certa porzione testuale di documento. Può essere una evidenziazione, sottolineatura o annotazione testuale.
Favorite	Documento preferito dall'utente.
User	Utente con uno o più abbonamenti attivi.
Token	Autentica e autorizza l'utente ad accedere ai vari documenti per i quali esiste un abbonamento attivo.

Tabella 4.1: Glossario dei termini (Entità)

Casi d'uso

Termine	Descrizione
Apertura Documento	Richiesta di apertura in lettura di uno specifico documento.
Chiusura Documento	Richiesta di chiusura del documento aperto in lettura.
Ricerca Documento	Richiesta di ricerca documento tramite query testuale.
Lettura Metadati Documento	Richiesta di lettura metadati documento.
Creazione Highlight	Richiesta di creazione annotazioni.
Lettura Highlight	Richiesta di lettura annotazioni.
Eliminazione Highlight	Richiesta di eliminazione annotazioni.
Creazione Bookmark	Richiesta di creazione segnalibri.
Lettura Bookmark	Richiesta di lettura segnalibri.
Eliminazione Bookmark	Richiesta di eliminazione segnalibri.
Creazione Progression	Richiesta di salvataggio dell' avanzamento di lettura di un documento.
Lettura Progression	Richiesta di lettura dell' avanzamento di lettura di un documento.
Eliminazione Progression	Richiesta di eliminazione dell' avanzamento di lettura di un documento.
Creazione Favorite	Richiesta di creazione preferiti.
Lettura Favorite	Richiesta di lettura preferiti.
Eliminazione Favorite	Richiesta di eliminazione preferiti.
Conversione PDF2EPUB	Richiesta di conversione di un documento statico in documento fluido (dal formato PDF al formato EPUB).
Download Contenuto Documenti	Scaricamento del contenuto dei documenti.
Download Copertina Documenti	Scaricamento della immagine di copertina dei documenti.
Login User	Richiesta di login dell'utente (lettura token).
Logout User	Richiesta di logout dell'utente (eliminazione token).
Controllo Login User	Controllo di autenticazione dell'utente già avvenuta (esistenza token).
Lettura Account Utente	Richiesta informazioni utente (dati anagrafici e mail).

Tabella 4.2: Glossario dei termini (Casi d'uso)

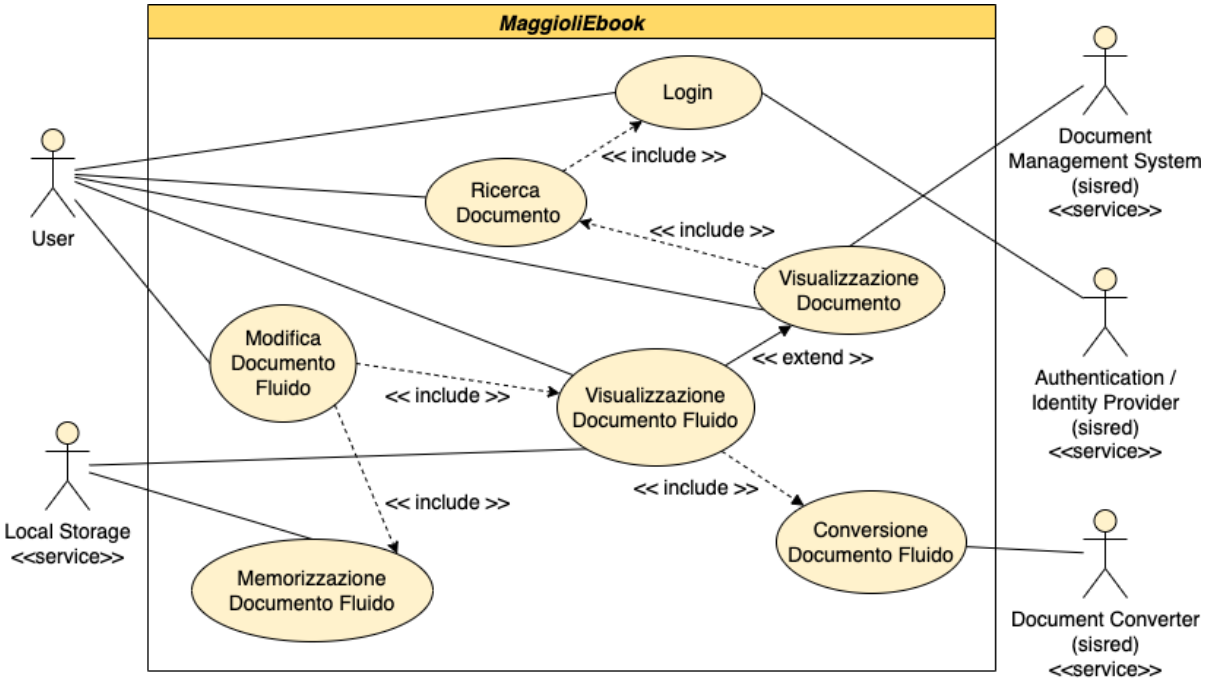

Figura 4.3: UML - Diagramma dei casi d'uso: funzioni/servizi offerti dalla applicazione MaggioliEbook

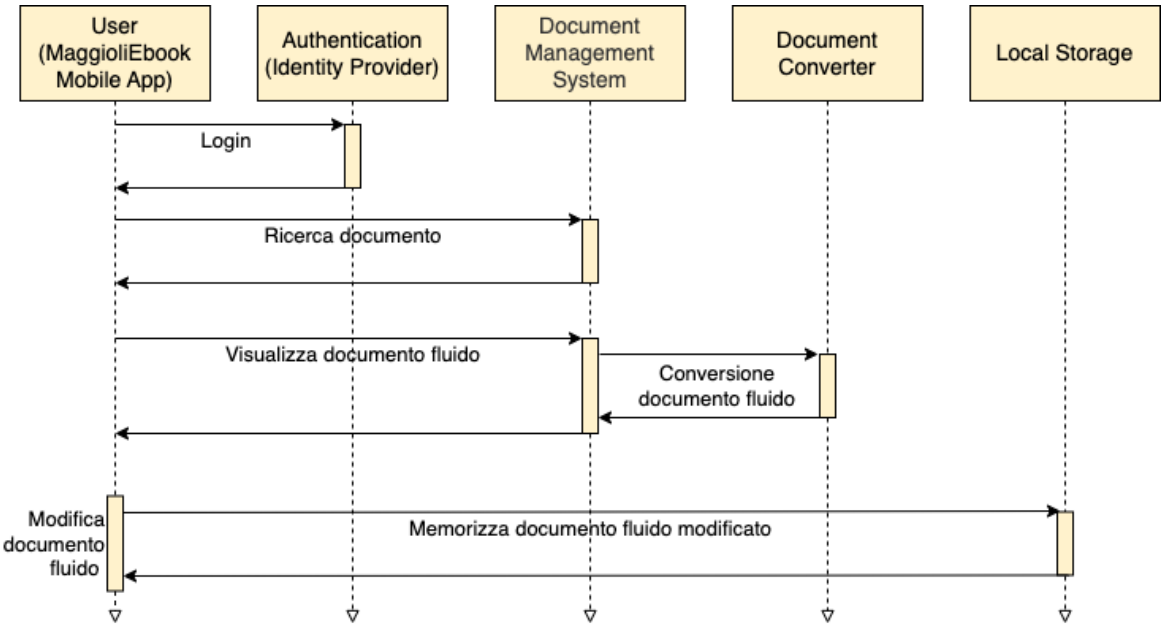

Figura 4.4: UML - Diagramma di sequenza: scenario di modifica di un nuovo documento "fluido"

4.4.2 Requisiti

- **R1** - Visualizzazione documenti.

  - **R1.1** - In modo fluido, mostrando il contenuto adattato al dispositivo in cui viene mostrato.
  - **R1.2** - In modo statico, mostrando il contenuto con uno specifico layout indipendente dal dispositivo in cui viene mostrato.
- **R2** - Modifica documenti fluidi (lato utente).
  - **R2.1** - Aggiunta segnalibri, commenti, sottolineature, evidenziazioni, al contenuto fluido.
  - **R2.2** - Memorizzazione segnalibri, commenti, sottolineature, evidenziazioni apportate al contenuto fluido.
- **R3** - Gestione utente.
  - **R3.1** - Login (autenticazione) utente.
  - **R3.2** - Visualizzazione documenti a cui l'utente è abbonato.
- **R4** - Ricerca documenti.
- **R5** - Conversione documenti da modo statico a modo fluido.
- **R6** - Modifica documenti in modo fluido (lato azienda).
  - **R6.1** - Aggiunta elementi/contenuti al documento in modo fluido (hyperlink, quiz, video, immagini, ...).
  - **R6.2** - Memorizzazione elementi/contenuti aggiunti al documento fluido (hyperlink, quiz, video, immagini, ...).

Capitolo 5

Automazione del processo di sviluppo

5.1 Introduzione

Nei precedenti capitoli sono stati introdotti i concetti alla base della cultura DevOps, del ciclo di vita del processo di sviluppo di applicazioni mobile e delle applicazioni multipiattaforma, arrivando a definire un caso di studio industriale. In questo capitolo viene descritto come è stato effettivamente realizzato il sistema per l'automazione del processo di sviluppo nel rispetto dei requisiti e delle specifiche indicate nel capitolo precedente.

In questa fase di realizzazione del sistema d'automazione e d'implementazione della pipeline, viene considerato il progetto base[1] fornito dal plugin KMM[2] per Android Studio al fine di mantenere il focus sul processo in modo più agnostico possibile rispetto ad una specifica applicazione utilizzatrice come può essere MaggioliEbook, argomento trattato nel capitolo successivo.

5.2 Self-Hosted MacOS GitLab Runner

Come anticipato nel capitolo 3 tutta la toolchain per lo sviluppo iOS è disponibile solamente per il sistema operativo macOS, il che implica la necessità di un'ambiente macOS anche per l'esecuzione della pipeline, almeno per tutti i task riguardanti l'applicazione iOS. Esistono diversi modi per realizzare un sistema d'automazione compatibile con i vincoli imposti da Apple e possono essere suddivisi nelle seguenti categorie:

- **Soluzione completa as-a-Service** - Il grande interesse per l'automazione e lo sviluppo di applicazioni iOS da parte delle aziende ha portato alla nascita di servizi cloud dedicati a questo scopo come ad esempio *Bitrise*[3] e *XCode Cloud*[4].

[1] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/3-applicazioni-multipiattaforma/kmm-example

[2] https://plugins.jetbrains.com/plugin/14936-kotlin-multiplatform-mobile

[3] https://www.bitrise.io/home

[4] https://developer.apple.com/xcode-cloud/

- **Runner macOS managed** - Un runner, ovvero il componente che esegue effettivamente i task della pipeline, può essere *managed* o *self-hosted*, come descritto nel capitolo 1. Nel caso di un runner managed con sistema operativo macOS si evita lo sforzo di configurare e mantenere un componente importante del sistema d'automazione ma si hanno costi elevati: solitamente al consumo di risorse di questa tipologia di runner è applicato un fattore moltiplicativo poco sostenibile in termini di costi. Alcuni esempi di piattaforme con questo modello di business per i runner managed sono GitHub Action[5] e GitLab CI[6].
- **Runner macOS self-hosted** - L'altra tipologia di runner consiste nell'installazione del componente su una macchina con sistema operativo macOS che deve essere configurata e mantenuta dall'utilizzatore. In questo caso è possibile utilizzare macchine virtuali as-a-Service, come quelle fornite da AWS[7] (Amazon Web Services), oppure hardware fisico Apple per installare ed eseguire il runner.

Data la disponibilità di tutta la toolchain Android per il sistema operativo macOS e il costo nullo in caso di runner self-hosted, è stata scelta quest'ultima tipologia per l'esecuzione dell'intera pipeline. L'installazione e la configurazione di un runner di questa tipologia può essere più o meno complicata in base alle funzionalità necessarie per il sistema d'automazione ed è una procedura chiamata registrazione. Nel caso specifico del caso di studio di questo progetto, i principali parametri per la registrazione del runner sono: (*i*) shell executor, (*ii*) token di registrazione al Server CI, (*iii*) cache condivisa e (*iv*) livello di concorrenza.

Definire un executor di tipo *shell* per il runner permette di aprire una nuova shell sulla macchina host per l'esecuzione di ogni job della pipeline e quindi di utilizzare tutti gli strumenti installati sulla stessa macchina. La cache condivisa e il livello di concorrenza permettono invece di ottimizzare il processo parallelizzando l'esecuzione dei task. Il seguente comando bash[8] mostra i parametri utilizzati per l'installazione e la configurazione del runner GitLab macOS self-hosted su una macchina fisica Apple:

```
sudo gitlab-runner register --non-interactive --url "https://gitlab.com/" \
  --registration-token "MY-TOKEN" --executor "shell" --description "macos-runner" \
  --request-concurrency 3 --tag-list "macos-m1" --run-untagged="false" \
  --locked="true" --access-level="not_protected" --cache-shared --cache-type="gcs" \
  --cache-gcs-access-id="${CACHE_GCS_ACCESS_ID}" \
  --cache-gcs-private-key="${CACHE_GCS_PRIVATE_KEY}" \
  --cache-gcs-bucket-name="${CACHE_GCS_PRIVATE_NAME}"
```

Listing 5: Comando bash per l'installazione e la configurazione del runner macOS self-hosted

[5]https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions#minute-multipliers

[6]https://docs.gitlab.com/ee/ci/pipelines/cicd_minutes.html#additional-costs-on-gitlab-saas

[7]https://aws.amazon.com/ec2/instance-types/mac/

[8]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/setup-gitlab-macos-runner.sh

Il seguente codice[9] mostra invece il contenuto del file di configurazione del runner, generato in seguito all'esecuzione del comando sopra indicato:

```
concurrent = 3
check_interval = 0
[session_server]
  session_timeout = 1800
[[runners]]
  name = "macos-runner"
  url = "https://gitlab.com/"
  token = "MY-SECRET-TOKEN"
  executor = "shell"
  [runners.cache]
    Type = "gcs"
    Shared = true
    [runners.cache.gcs]
      BucketName = "gitlabcom-xxxxxx-shared-cache"
      AccessID = "xxxx@xxxx.iam.gserviceaccount.com"
      PrivateKey = "-----BEGIN PRIVATE KEY-----\nXXXX\n-----END PRIVATE KEY-----\n"
```

Listing 6: File di configurazione (*config.toml*) generato al momento dell'installazione del runner

5.3 Modello di branching

L'utilizzo di un adeguato flusso di lavoro è fondamentale per definire un'efficiente automazione CI/CD. Con *branching* si intende l'utilizzo di uno o più flussi principali di lavoro dai quali divergono altri flussi per svolgere determinati lavori per poi convergere al loro termine: in base alle modalità di apertura e chiusura di questi flussi si definiscono diversi modelli di branching.

Il modello che si intende utilizzare (fig. 5.1) è basato sul modello di branching GitFlow[10] e prevede tre branch principali:

- **dev** - Flusso principale di sviluppo. Ogni modifica apportata a questo branch corrisponde al rilascio di una nuova versione *alpha* per la validazione interna. E' da questo branch che vengono aperti e chiusi nuovi branch, sia per lo sviluppo di nuove funzionalità (*feature*) che per la risoluzione di bug/patch (*bugfix*).
- **test** - Branch modificato solamente tramite merge di modifiche provenienti dal branch *dev* con lo scopo di rilasciare una nuova versione *beta* per la validazione esterna.
- **main** - Branch modificato solamente tramite merge di modifiche provenienti dal branch *test* con lo scopo di rilasciare una nuova versione in produzione (*prod*).

[9] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/config.toml

[10] https://www.atlassian.com/it/git/tutorials/comparing-workflows/gitflow-workflow

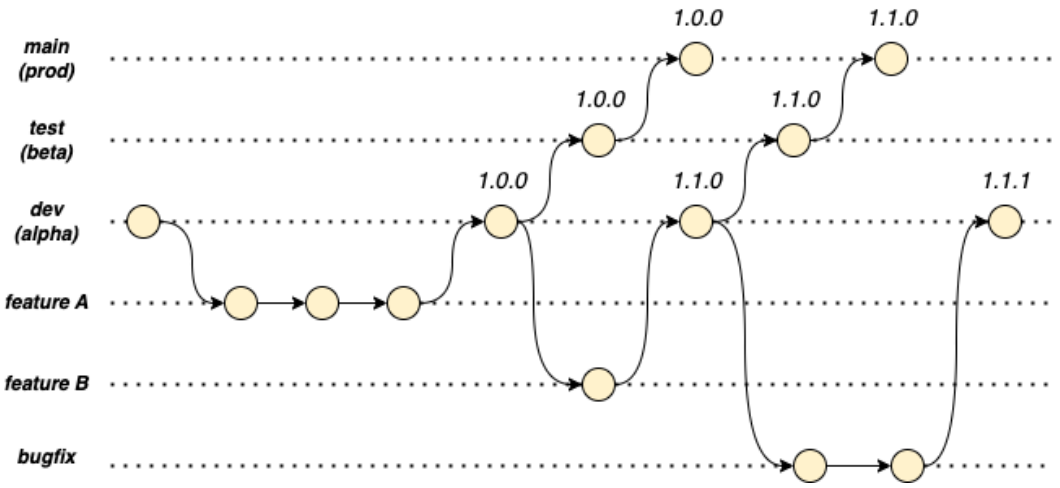

Figura 5.1: Esempio di flusso di sviluppo adottando il modello di branching indicato

Grazie ai meccanismi d'automazione a supporto della CI/CD fornite dal CI Server si definiscono specifiche regole di attivazione, chiamate *trigger rules*, che permettono di indicare quando uno specifico job deve essere eseguito. Queste regole si basano su eventi che si verificano sul sistema di versionamento come ad esempio *commit*, *tag* e *merge request*. Tramite questa funzionalità è possibile dunque discriminare quali file sono stati modificati su quale branch per eseguire le operazioni associate come descritto sopra.

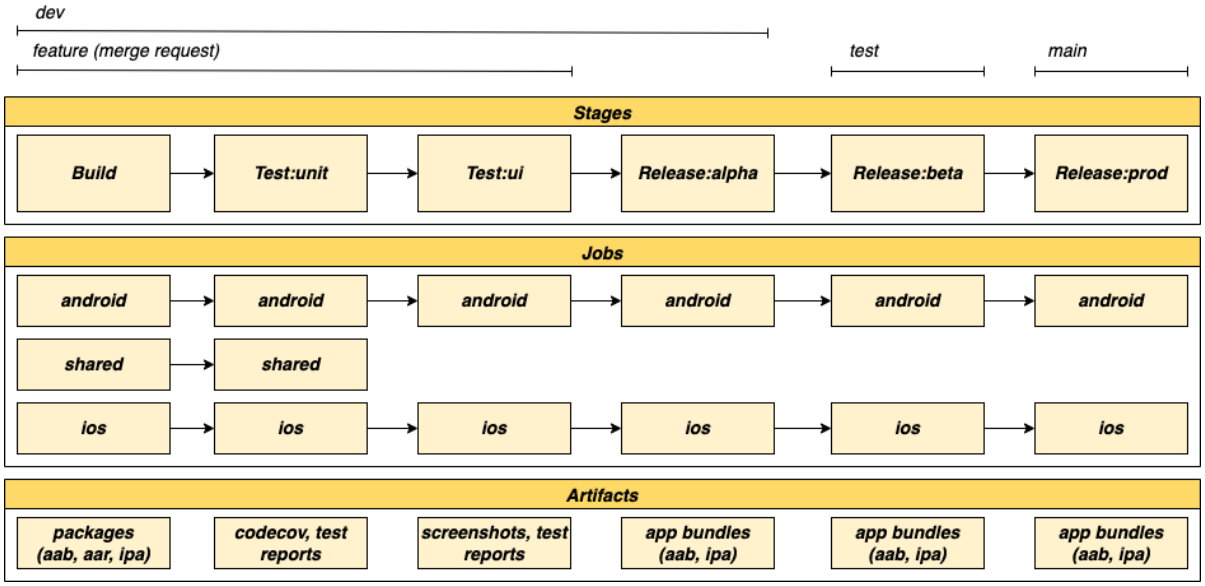

Figura 5.2: Stage, job e artefatti associati agli eventi sui branch che compongono l'intera pipeline

5.4 Templating

Uno dei principali vantaggi del meccanismo Pipeline as Code, anticipato nel capitolo 1, è la possibilità di adottare metodologie simili a quelle utilizzate nella programmazione per la definizione delle pipeline. Tramite i seguenti costrutti forniti dalla piattaforma GitLab è possibile utilizzare tecniche come ereditarietà e incapsulamento per definire pipeline riutilizzabili, il quale è un importante requisito del processo, definito nel capitolo precedente:

- **Template** - I file YAML che definiscono una pipeline o sottoparte di essa, possono risiedere in un repository diverso da quello in cui si trova il progetto che la utilizza. In questo modo si definisce una sola volta il flusso base per poterlo successivamente includere e/o estendere negli specifici progetti che necessitano il suo utilizzo.
- **Include** - Definiti i template è necessario includerli all'interno dello specifico progetto per poterli effettivamente utilizzare.
- **Extend** - Definendo lo script del job padre in modo parametrico è possibile pilotarne il comportamento tramite la definizione di variabili d'ambiente nel job figlio. Questa funzionalità è un'alternativa alle *anchors*[11] fornite nativamente da YAML ma più flessibile e leggibile.
- **Hidden Jobs** - I job definiti con un punto all'inizio del nome non vengono considerati dal linter GitLab e quindi vengono esclusi dall'esecuzione tramite il runner. Definendo job "nascosti" è possibile specificare tutti quegli aspetti comuni e parametrizzabili da estendere con altri job.

Il seguente codice[12] mostra il job padre di ogni job dedicato all'applicazione iOS, definito come hidden job parametrico: ogni job figlio dovrà estendere questo job e definirne dei valori per le variabili d'ambiente.

```
1  .base-ios:
2    variables:
3      PROJECT_DIR: to_be_extended
4      GYM_WORKSPACE: to_be_extended
5      GYM_SCHEME: to_be_extended
6      APP_STORE_CONNECT_API_KEY_PATH: to_be_extended
7    before_script:
8      - cd $PROJECT_DIR
9      - bundle install
10     - echo $APP_STORE_CONNECT_API_KEY_BASE64 | base64 -d >
       ↪  $APP_STORE_CONNECT_API_KEY_PATH
```

Listing 7: Hidden Job parametrico per i job iOS

Considerando la composizione della pipeline obiettivo (fig. 5.2), una possibile organiz-

[11] https://yaml.org/spec/1.2.2/#3222-anchors-and-aliases

[12] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-base.yml

zazione ottimale è la definizione di un template[13] per ogni fase del processo, in modo da poter importare anche solamente una sottoparte della pipeline:

- **kmm-base** - Definisce tutte le configurazioni di base della pipeline come ad esempio gli stage, le variabili d'ambiente globali e i job di pre-configurazione.
- **kmm-build** - Definisce i job relativi alla fase di compilazione e packaging.
- **kmm-test** - Definisce i job per le fasi di unit testing e ui testing.
- **kmm-analysis** - Definisce i job per le fasi di analisi statica del codice, analisi delle dipendenze e integrazione con SonarQube.
- **kmm-release** - Definisce i job utili alla stabilizzazione e rilascio delle applicazioni (alpha-beta-prod).

Realizzati i template è possibile utilizzarli tramite inclusione remota come nel seguente codice[14] d'esempio:

```yaml
include:
  - project: path/to/my/template/repo
    file:
      - '/kmm-templates/kmm-base.yml'
      - '/kmm-templates/kmm-build.yml'
      - '/kmm-templates/kmm-test.yml'
      - '/kmm-templates/kmm-release.yml'
```

Listing 8: Esempio d'utilizzo dei template GitLab da repository remoto

5.5 Continuous Integration

5.5.1 Pre

Molti dei passi che compongono una pipeline utilizzano tipicamente gli stessi tools e le stesse configurazioni per svolgere task diversi. Ad esempio, la compilazione del codice e l'esecuzione degli unit test per un'applicazione Java utilizzano in entrambi i casi la stessa JDK[15], lo stesso tool di build automation e devono essere scaricate le stesse dipendenze di progetto dal package manager di riferimento. Utilizzando meccanismi di caching e passaggio di artefatti tra i vari job è possibile eseguire tutti quei task di configurazione una sola volta all'inizio della pipeline, risparmiando tempo e risorse in tutte le fasi successive.

Nel caso della pipeline progettata per lo sviluppo di applicazioni multipiattaforma con KMM è necessario eseguire i seguenti task di configurazione iniziale:

[13] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/5-automazione-del-processo-di-sviluppo/kmm-templates

[14] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/.gitlab-ci.yml

[15] Java Development Kit

- Configurazione dell'ambiente per lo sviluppo Kotlin tramite l'installazione e il caching degli SDK e del sotto-compilatore *Kotlin/Native*[16] per lo sviluppo multipiattaforma.
- Configurazione dell'ambiente per lo sviluppo Android tramite l'installazione del SDK Android target e dei vari tools necessari tramite *sdkmanager*[17].
- Configurazione dell'ambiente per lo sviluppo iOS tramite impostazione di XCode e CLI Developer Tools.
- Installazione e caching di tutte le dipendenze dei vari moduli.
- Configurazione delle chiavi per l'autenticazione ai servizi cloud forniti da Google e Apple.

Il seguente codice[18] mostra il job di pre-configurazione per tutti i job successivi riguardanti la piattaforma Android:

```
pre:android:
  extends: .base-android
  stage: .pre
  variables:
    JDK_VERSION: 8.0.332-zulu
  script:
    - fastlane run validate_play_store_json_key json_key:$JK_PATH
    - sdkmanager "platforms;android-${ANDROID_COMPILE_SDK}" >/dev/null
    - sdkmanager "platform-tools" >/dev/null
    - sdkmanager "build-tools;${ANDROID_BUILD_TOOLS}" >/dev/null
    - set +o pipefail
    - yes | sdkmanager --licenses
    - set -o pipefail
    - echo "Installed Android SDK
    ↪  ${ANDROID_COMPILE_SDK}-${ANDROID_BUILD_TOOLS}-${ANDROID_SDK_TOOLS}"
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" || $CI_COMMIT_BRANCH == "dev"
```

Listing 9: Job di pre-configurazione Android

5.5.2 Build e Packaging

Tipicamente la fase d'integrazione continua inizia con la verifica della corretta compilazione del codice sorgente. La compilazione rappresenta un vincolo essenziale per tutte le successive fasi e per questo è definita come fase bloccante: in caso di compilazione fallita la pipeline termina senza procedere con le fasi successive definite.

Nel caso dello sviluppo di applicazioni mobile, per lo stage iniziale di build la pratica più diffusa è quella di validare sia la compilazione del codice che la pacchettizzazione della applicazione nei formati richiesti dalle piattaforme target. Dato il funzionamento di un'applicazione KMM (capitolo 3) è necessario prima compilare il codice del modulo

[16] https://kotlinlang.org/docs/native-improving-compilation-time.html#general-recommendations
[17] https://developer.android.com/studio/command-line/sdkmanager
[18] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-base.yml

condiviso *shared*, poi quello specifico delle piattaforme Android e iOS e infine impacchettarlo negli artefatti che verranno passati in input alla fase di delivery, rispettivamente aar[19], aab[20] e ipa[21].

Il seguente codice[22] mostra il template che definisce il job base di compilazione e pacchettizzazione dell'applicazione Android tramite l'utilizzo combinato dei tools Fastlane e Gradle:

```
.android:build:
  stage: build
  extends: .base-android
  script:
    - cd $PROJECT_DIR
    - bundle install
    - export FL_GRADLE_TASK="assembleDebug"
    - bundle exec fastlane run gradle
  artifacts:
    name: shared-build
    expire_in: 1 day
    paths:
      - ${PROJECT_DIR}/build/outputs
```

Listing 10: Pipeline job dedicato alla compilazione e pacchettizzazione dell'applicazione Android

5.5.3 Testing

Terminato con successo lo stage di verifica della compilazione e della pacchettizzazione del codice segue la fase di test, per la quale si distinguono due tipologie principali di testing: la prima con lo scopo di validare la logica applicativa, chiamata *Unit Testing* e la seconda per la validazione dell'interfaccia grafica, chiamata *UI Testing*.

Quando si parla di testing in ambito mobile ci si riferisce tipicamente alla specifica tipologia di test chiamata *Instrumented Testing*, dove è necessario realizzare un progetto di test apposito in modo da installare e testare l'applicazione in un ambiente simulato, come per esempio quello fornito dagli emulatori [1]. In realtà, l'instrumentazione di un'applicazione è necessaria solamente per quei test dove vengono richiamate le API dello specifico SDK della piattaforma target, come solitamente accade nel caso della UI testing.

Unit Testing

Con Unit Testing si intende l'attività di verifica di certe porzioni del codice, dette unità, implementata tipicamente utilizzando librerie predisposte per ciascun specifico linguag-

[19] Android Archive
[20] Android App Bundle
[21] iOS App Store Package
[22] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-build.yml

gio di programmazione. I movimenti Agile e TDD[23] incoraggiano la scrittura di unit test automatizzati, i quali dovrebbero essere scritti prima dello sviluppo effettivo del codice [10].

Nonostante la struttura di un'applicazione KMM contenga tutta la logica applicativa nel modulo condiviso, è necessario eseguire test anche su porzioni di codice specifico delle piattaforme nel caso vengano richiamate API dei relativi SDK. Le librerie utilizzate in questa fase di unit testing sono:

- **JUnit**[24] - Framework di unit testing per il linguaggio di programmazione Java, utilizzato nel caso di studio per il testing unitario del modulo condiviso e dell'applicazione Android.
- **XCTest**[25] - Framework standard di testing per progetti XCode, tra cui unit testing di codice Swift/Objective-C.

UI Testing

Analogamente agli unit test, dove si verifica l'integrità della logica applicativa a fronte di una modifica del codice sorgente, nella UI Testing si verifica l'integrità dell'interfaccia grafica e dell'esperienza utente. In questa sotto-fase di testing sono coinvolti solamente i moduli che implementano la UI e vengono testati tramite le seguenti librerie:

- **Espresso** - Framework standard per la UI testing di applicazioni Android.
- **XCTest** - Framework standard di testing per progetti XCode, tra cui UI testing di applicazioni iOS (XCUI[26]).

Questa specifica fase della pipeline è tipicamente l'unica che richiede la simulazione degli ambienti d'esecuzione tramite l'emulazione dei dispositivi. Per poter eseguire un emulatore di qualsiasi tipo attraverso un sistema d'automazione è necessario considerare alcuni aspetti fondamentali come ad esempio (*i*) la necessità d'esecuzione senza interfaccia grafica, chiamata modalità *headless* e (*ii*) l'accesso concorrente alle risorse del sistema host sul quale eseguono.

Nel caso in cui esistano anche altre operazioni che possono essere automatizzate e che necessitano di un emulatore, solitamente si tende ad eseguirle in questa fase. Una pratica comune è la combinazione dell'esecuzione dei test sull'interfaccia grafica con la cattura delle schermate, la quale permette di ottimizzare il flusso di lavoro e ottenere direttamente dal sistema d'automazione gli screenshot desiderati. Con la diffusione di questa pratica sono stati creati tool appositi, come quelli forniti dallo strumento Fastlane adottato:

- **Screengrab**[27] - Tool specifico per Android che richiede la definizione di appositi test utilizzando (*i*) Espresso per navigare nell'applicazione in esecuzione e (*ii*) la

[23] Test Driven Development
[24] https://junit.org/junit5/
[25] https://developer.apple.com/documentation/xctest
[26] https://developer.apple.com/documentation/xctest/user_interface_tests
[27] https://docs.fastlane.tools/actions/screengrab/

libreria Screengrab[28] per la cattura delle schermate.
- **Snapshot**[29] - Tool specifico per iOS che genera codice[30] Swift in grado di sfruttare XCTest per la navigazione dell'applicazione e la cattura delle schermate.

Il seguente job[31] base mostra la strategia utilizzata nel caso di studio per avviare un emulatore Android, eseguire i test sull'interfaccia grafica e restituire, come artefatto, le immagini delle schermate catturate:

```
.base-android-ui-test:
  stage: test:ui
  extends: .base-android
  resource_group: ui-test
  variables:
    SCREENGRAB_APP_APK_PATH: to_be_extended
    SCREENGRAB_TESTS_APK_PATH: to_be_extended
    AVD: to_be_extended
    SCREENGRAB_DEVICE_TYPE: to_be_extended
    SCREENGRAB_SKIP_OPEN_SUMMARY: "true"
    SCREENGRAB_REINSTALL_APP: "true"
  script:
    - cd $PROJECT_DIR
    - bundle install
    - nohup emulator -avd $AVD -no-window > nohup-$AVD.out 2>&1 &
    - echo $! > pid-$AVD.txt
    - sleep 10
    - export FL_GRADLE_TASK="assembleAndroidTest"
    - bundle exec fastlane run gradle
    - bundle exec fastlane screengrab
    - kill -9 `cat pid-$AVD.txt` || adb devices | grep emulator | cut -f1 | while
    ↪  read line; do adb -s $line emu kill; done || true
  artifacts:
    name: android-ui-test-phone
    expire_in: 1 day
    paths:
      - ${PROJECT_DIR}/nohup*.out
      - ${PROJECT_DIR}/fastlane/metadata/android
```

Listing 11: Job base Android dedicato al testing dell'interfaccia grafica e alla cattura delle schermate

5.5.4 Dependency Management

Al giorno d'oggi è difficile immaginare un'applicazione software che non faccia uso di librerie di terze parti. Queste librerie, dette dipendenze, vengono incluse nel software tipicamente per il vantaggio di poter riutilizzare soluzioni a problemi che sono già stati

[28] https://mvnrepository.com/artifact/tools.fastlane/screengrab
[29] https://docs.fastlane.tools/actions/snapshot/
[30] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/SnapshotHelper.swift
[31] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-test.yml

risolti ma comportano anche il bisogno di essere gestite. Maggiore è il numero di dipendenze incluse in un progetto e maggiore è la complessità nella loro gestione: per questo motivo l'automazione di questo task sta diventando una pratica sempre più comune.

Automatizzare la gestione delle dipendenze di un progetto significa configurare un sistema in grado d'interagire con:

- **Package Manager** - Fondamentale per ottenere le versioni *latest* delle dipendenze e confrontarle con quelle utilizzate dal progetto, discriminando il tipo di aggiornamento necessario (*major*, *minor* o *patch*).
- **Version Control System** - La presenza di un aggiornamento di versione per una o più dipendenze deve essere applicata al codice in modo automatico e devono essere adottate le stesse tecniche di continuous integration per la validazione delle modifiche apportate dallo sviluppatore.

Anche in questo caso è possibile utilizzare sistemi "*managed*" forniti *as-a-Service* oppure *self-hosted*. Uno dei vincoli sul processo di sviluppo, definiti nel capitolo 4, consiste nell'integrazione del servizio *self-hosted* aziendale *Renovate* per la gestione automatizzata delle dipendenze. Tramite la schedulazione di un task dedicato, asincrono rispetto il normale flusso di sviluppo, questa fase del processo viene eseguita come indicato nel seguente diagramma:

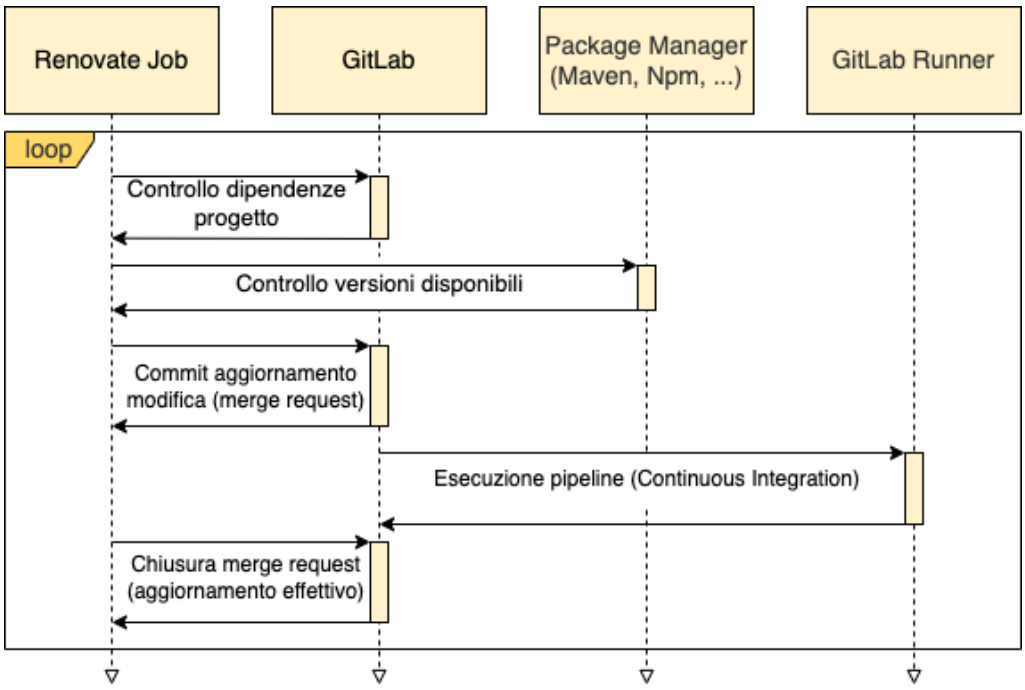

Figura 5.3: Sequenza di aggiornamento automatico delle dipendenze con Renovate e GitLab Runner

Tipicamente l'aggiornamento automatico avviene per versioni *patch* e *minor*, ovvero per quegli aggiornamenti che spesso non presentano *breaking changes*: in tutti gli altri casi

(*major*) è necessaria l'approvazione manuale da parte di uno sviluppatore.

Il comportamento di Renovate può essere configurato ad-hoc per le diverse esigenze di ogni progetto tramite un apposito file, come quello seguente[32], versionato insieme al codice.

```
{
  "$schema": "https://docs.renovatebot.com/renovate-schema.json",
  "labels": ["dependencies"],
  "automerge": true,
  "semanticCommitScope": "{{manager}}",
  "packageRules": [
    {
      "description": "Trigger manual intervention for major updates",
      "automerge": false,
      "dependencyDashboardApproval": true,
      "matchUpdateTypes": ["major"]
    },
    {
      "description": "Automatic PR and Merge if success",
      "automerge": true,
      "dependencyDashboardApproval": false,
      "matchUpdateTypes": ["minor", "patch", "pin", "digest"]
    },
    {"matchPackagePatterns": ["^io.insert-koin"], "groupName": "koin"},
    {"matchPackagePatterns": ["^androidx"], "groupName": "androidx"},
    {"matchPackagePatterns": ["^io.ktor"], "groupName": "ktor"},
    {"matchPackagePatterns": ["^org.jetbrains.kotlinx"], "groupName": "kotlinx"}
  ]
}
```

Listing 12: Configurazione custom di un progetto Android per l'aggiornamento automatico delle dipendenze con Renovate

5.6 Continuous Delivery

Le precedenti fasi d'integrazione continua, se completate con successo, restituiscono come output il pacchetto contenente tutte le informazioni utili alla stabilizzazione e distribuzione dell'applicazione. Nel caso dello sviluppo di applicazioni multipiattaforma con KMM bisogna considerare che le due applicazioni potrebbero essere rilasciate singolarmente, ad esempio a fronte di una modifica all'interfaccia grafica della sola versione Android, oppure essere rilasciate entrambe, tipicamente a causa di una modifica della logica applicativa che è condivisa dalle due applicazioni.

Data in input l'applicazione da distribuire è necessario svolgere una serie di operazioni, comuni ad entrambe le piattaforme Android e iOS, le quali richiedono considerazioni differenti dagli aspetti del normale sviluppo software, come (*i*) l'identificazione ai servizi

[32] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/renovate.json

di distribuzione, (*ii*) l'ottenimento dei permessi per l'utilizzo di certe API (creazione API-key), (*iii*) la creazione dei certificati per la firma del codice (*code signing*) e (*iv*) la definizione dei gruppi di tester [11]. Tutti questi sotto-task della fase di Continuous Delivery sono svolti sfruttando i servizi cloud forniti da Google e Apple per le relative piattaforme proprietarie, come anticipato nel capitolo 3.

5.6.1 Google Play Console

A differenza delle applicazioni iOS, le quali richiedono due strumenti separati per la loro stabilizzazione e distribuzione, le applicazioni Android vengono gestite interamente da un'unica piattaforma chiamata *Google Play Console*. Tramite il concetto di promozione, una specifica versione di applicazione viene promossa a versione *alpha*, promossa da *alpha* a *beta* o da *beta* a *produzione*, quest'ultima equivalente alla pubblicazione sul marketplace *Google Play Store*.

Per poter abilitare il sistema d'automazione ad eseguire la fase di delivery per la piattaforma Android sono richiesti i seguenti passaggi:

- Creazione account *Google Developer*.
- Creazione scheletro iniziale dell'applicazione sul portale Google Play Console. Devono essere compilati questionari obbligatori, come quello per la classificazione PEGI[33], create le mailing list per gli alpha/beta tester e impostate preferenze come la monetizzazione dell'applicazione e la presenza di pubblicità/annunci.
- Creazione Service Account per autenticare le chiamate alle API Google. Queste informazioni devono essere inserite in GitLab per poter essere usate durante l'esecuzione delle pipeline.
- Creazione chiavi e certificati, in formato *Java KeyStore* (jks) per la firma del codice. Tale formato dev'essere codificato in esadecimale per poter essere utilizzato in GitLab al fine di automatizzare il processo di firma del codice. Il seguente script[34] bash mostra un esempio per la creazione, codifica e decodifica della chiave jks:

```
1 # Generazione chiave jks
2 keytool -genkey -v -keystore release-key.jks -keyalg RSA -keysize 2048 -validity
  ↪  10000 -alias my-alias
3
4 # Codifica chiave in esadecimale
5 xxd -p release-key.jks > hex-encoded-release-key
6
7 # Decodifica chiave
8 cat hex-encoded-release-key | xxd -r -p - > decoded-release-key.jks
```

Listing 13: Comandi bash d'esempio per la creazione, codifica e decodifica di una chiave in formato jks

[33] https://pegi.info/it

[34] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/jks-android-gitlab.sh

5.6.2 App Store Connect

Nell'ecosistema Apple non esiste il concetto di promozione di versione. La tipologia di rilascio viene discriminata in base al gruppo di tester target definiti sul portale *Testflight*, nello specifico il gruppo *Internal Tester* comprende i tester con accesso alla versione *alpha* mentre il gruppo *External Tester* quelli con accesso alla versione *beta*.

La sequenza di step necessari a configurare il sistema d'automazione è molto simile a quella sopra descritta per le applicazioni Android:

- Creazione account *Apple Developer*.
- Creazione scheletro iniziale dell'applicazione sul portale App Store Connect: anche in questo caso devono essere compilati questionari obbligatori e impostate le preferenze analoghe al caso Android.
- Creazione gruppi per i tester interni ed esterni sul portale Testflight.
- Creazione API key[35] per l'autenticazione ai servizi Apple e App-Specific Password per l'autenticazione del tool Fastlane. Come per i servizi Google, anche in questo caso è necessario inserire la chiave in GitLab per poterla utilizzare durante l'esecuzione della pipeline.
- Creazione certificato di tipo *iOS Distribution* (Certificate Signing Request) e provisioning file del tipo *App Store* per il processo di firma del codice. In questo caso non è necessario inserire le informazioni su GitLab perchè vengono automaticamente ottenute dal tool Fastlane autenticandosi ai servizi Apple.

5.6.3 Alpha/Beta Release

Considerando il modello di branching definito (fig. 5.1), ognuno dei tre branch principali corrisponde ad un ambiente e quindi ad un preciso target di utilizzatori. Nel caso del branch di sviluppo principale *dev*, le applicazioni vengono rilasciate a scopo di testing interno da parte di figure tecniche tipicamente coinvolte nello sviluppo.

Per rendere più veloce questa prima fase di stabilizzazione, dato che non vengono coinvolte persone esterne, sia Google che Apple permettono la distribuzione diretta senza il loro intervento per lo svolgimento del processo di *App Review*. Il discorso cambia invece per tutte le altre fasi di delivery: l'applicazione deve essere revisionata e approvata prima di poter procedere sia per il rilascio in *beta* che per la vera e propria pubblicazione sui marketplace.

Il seguente codice[36] definisce il job base per la distribuzione in versione *alpha* di un'applicazione Android:

[35] https://docs.fastlane.tools/app-store-connect-api/#using-fastlane-api-key-json-file

[36] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-release.yml

```yaml
.android:release:alpha:
  extends: .base-android:release
  variables:
    FL_SLACK_MESSAGE: "_*NEW ANDROID ALPHA RELEASE*_\n"
    SUPPLY_AAB: to_be_extended
    FL_GRADLE_TASK: "bundleRelease"
    SUPPLY_TRACK: "alpha"
  script:
    - cd $PROJECT_DIR && bundle install
    - bundle exec fastlane run gradle upload_to_play_store slack
  rules:
    - if: $CI_COMMIT_BRANCH == "dev" && $CI_PIPELINE_SOURCE != "schedule"
      changes: [ "$PROJECT_DIR/**/build.gradle.kts" ]
      when: manual
```

Listing 14: Job base per il rilascio in versione *alpha* dell'applicazione Android

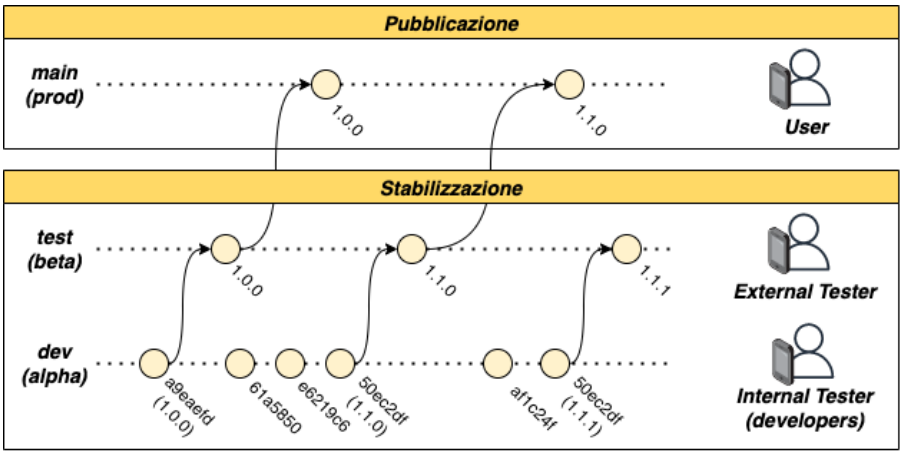

Figura 5.4: Stage, job e artefatti associati agli eventi sui branch che compongono l'intera pipeline

5.7 Continuous Inspection

L'ultimo requisito del processo mancante è la fase d'analisi automatica del codice sviluppato. Come descritto nel capitolo 1, la pratica di Continuous Inspection può comprendere tantissime tecniche d'analisi, tra cui quelle implementate nel caso di studio industriale che sono l'analisi statica (SAST) e l'analisi delle dipendenze (SCA).

5.7.1 Composizione software

Con composizione software si intende l'insieme di codice sviluppato da terze parti e che viene incluso all'interno dell'applicazione, tipicamente sotto forma d'importazione di librerie, dette dipendenze. Queste dipendenze includono a loro volta altre dipendenze e potrebbero introdurre vulnerabilità nell'applicazione che le utilizza: per questo motivo

è necessario analizzare le dipendenze al fine di individuare le vulnerabilità ed essere tempestivi nel loro aggiornamento non appena vengono rilasciate delle patch di sicurezza.

Il tool adottato a livello aziendale per effettuare questa tipologia d'analisi è *DependencyCheck*[37], un tool open-source rilasciato e mantenuto da OWASP[38], disponbile come plugin Gradle, binario eseguibile e immagine Docker. Uno dei principali punti di forza di questo tool è la possibilità di analizzare più progetti di diversa natura, come applicazioni Android e iOS, senza la necessità di configurazioni complesse[39]:

```
.dependency-check:
  stage: analysis
  image:
    name: owasp/dependency-check:7.1.1
    entrypoint: [""]
  dependencies: []
  allow_failure: true
  variables:
    REPORT_FORMAT: XML
    PROJECT_DIR: to_be_extended
    REPORT_PATH: ${CI_PROJECT_DIR}/reports
  script:
    - /usr/share/dependency-check/bin/dependency-check.sh --format="${REPORT_FORMAT}"
    ↪  --scan="${PROJECT_DIR}" -o="${REPORT_PATH}"
  artifacts:
    name: dependency-check-report
    expire_in: 1 day
    paths:
      - "reports/dependency-check-report.*"
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"
```

Listing 15: Job base dedicato all'analisi delle dipendenze sia Android che iOS tramite il tool OWASP DependencyCheck

5.7.2 Analisi statica

Per quanto riguarda l'analisi statica del codice non è possibile invece un livello di semplicità di configurazione come quello dell'analisi delle dipendenze. Gli strumenti necessari per questa fase d'analisi dipendono fortemente dalle tecnologie adottate e dalle piattaforme target. I moduli dell'applicazione KMM, ovvero (*i*) applicazione Android, (*ii*) applicazione iOS e (*iii*) logica condivisa, devono essere considerati come moduli separati per utilizzare i seguenti strumenti di analisi individuati:

- **Android Lint** - Tool per l'analisi statica di applicazioni Android, disponibile come binario e plugin Gradle (a partire dalla versione 16 ADT[40]).

[37] https://github.com/jeremylong/DependencyCheck
[38] https://owasp.org/
[39] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-analysis.yml
[40] Android Development Tools

- **Detekt** - Tool specifico per l'analisi statica di codice Kotlin, disponibile come binario eseguibile e plugin Gradle.
- **SwiftLint** - Tool specifico per l'analisi statica di codice Swift, disponibile come binario eseguibile e dipendenza CocoaPods.
- **SonarQube** - Client tool per l'interazione con il sistema aziendale SonarQube, disponibile come binario eseguibile e plugin Gradle.

Il seguente codice[41] mostra l'utilizzo dello strumento SwiftLint per l'analisi statica di applicazioni iOS tramite il job base schedulato:

```
.static-analysis:ios:
  stage: analysis
  image: ghcr.io/realm/swiftlint:5.5-latest
  dependencies: []
  allow_failure: true
  variables:
    PROJECT_DIR: to_be_extended
    PODS_DIR: ${CI_PROJECT_DIR}/${PROJECT_DIR}/Pods
    REPORT_PATH: swiftlint.json
  script:
    - cd $PROJECT_DIR
    - swiftlint . --reporter="sonarqube" --output $REPORT_PATH
  artifacts:
    name: static-analysis-report-ios
    when: always
    expire_in: 1 day
    paths:
      - ${PROJECT_DIR}/${REPORT_PATH}
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"
```

Listing 16: Job base dedicato all'analisi statica del codice Swift dell'applicazione iOS

5.7.3 Schedulazione Job

Le fasi di analisi del codice possono essere introdotte nel processo di sviluppo distinguendo due modalità d'esecuzione:

- **Sincrona** - L'analisi del codice viene eseguita ad ogni commit su uno specifico branch (in questo caso *dev*). Il vantaggio consiste nella ricezione di un feedback più rapido sulla modifica apportata al codice mentre lo svantaggio è dato da un incremento considerevole nel tempo d'esecuzione di ogni pipeline.
- **Asincrona** - Tramite la schedulazione delle fasi di analisi in un momento diverso da quello in cui viene apportata la modifica al codice è possibile sia ridurre i tempi d'esecuzione delle pipeline che ottenere un feedback cumulativo sull'insieme delle modifiche apportate tra due esecuzioni delle pipeline d'analisi del codice. Ad esempio, schedulando alla mezzanotte di ogni giorno l'esecuzione delle fasi di

[41] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/5-automazione-del-processo-di-sviluppo/kmm-templates/kmm-analysis.yml

analisi, è possibile accedere al portale SonarQube all'inizio della giornata lavorativa successiva e pianificare gli interventi di risoluzione degli eventuali bug introdotti nella giornata precedente.

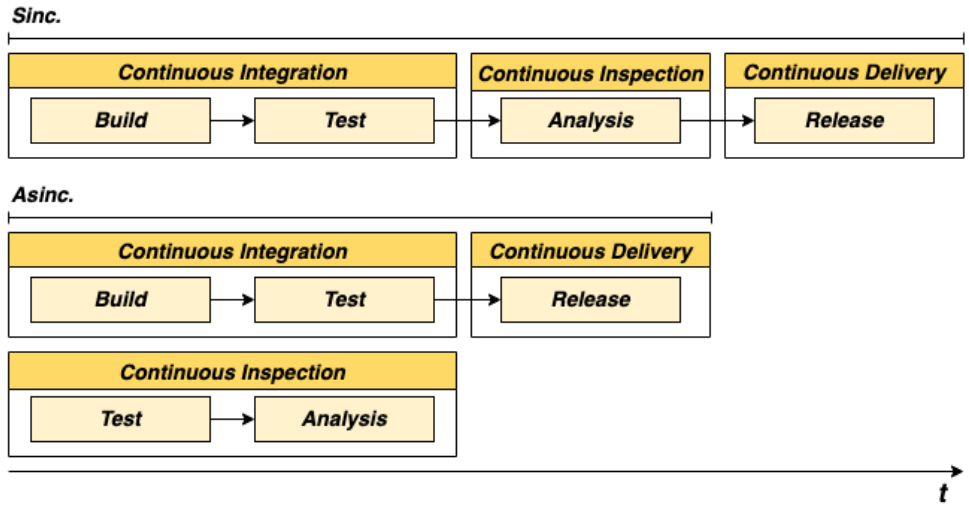

Figura 5.5: Confronto tra esecuzione sincrona e asincrona della fase d'analisi

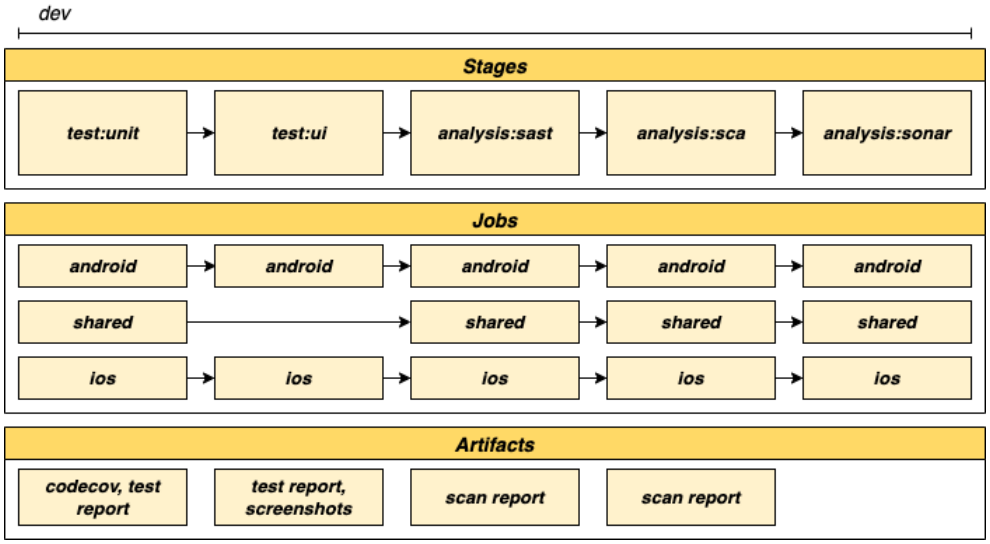

Figura 5.6: Struttura della seconda pipeline schedulata per l'analisi del codice

La modalità d'esecuzione scelta per la fase di Continuous Inspection è quella asincrona per i suoi vantaggi, preferiti dalla maggior parte dei team di sviluppo in azienda rispetto a quelli dell'esecuzione sincrona. E' necessario dunque schedulare una seconda pipeline in grado di: (*i*) eseguire i test, (*ii*) eseguire l'analisi statica del codice, (*iii*) eseguire l'analisi delle dipendenze e (*iv*) collezionare tutti i report prodotti dalle fasi precedenti al fine di caricarli sul sistema aziendale SonarQube per la gestione centralizzata delle vulnerabilità.

Capitolo 6

Sviluppo applicazione MaggioliEbook

6.1 Introduzione

Dato il processo di sviluppo e configurato il sistema d'automazione, è possibile partire con lo sviluppo effettivo dell'applicazione mobile multipiattaforma MaggioliEbook. In questo capitolo vengono descritte le fasi di progettazione e implementazione di un'applicazione mobile tramite l'utilizzo del framework Kotlin Multiplatform Mobile, rispettando i requisiti definiti nel capitolo 4.

6.2 Progettazione

La fase di progettazione, come anticipato nel capitolo 2, comprende diverse attività tra cui le principali sono (*i*) la modellazione del dominio, (*ii*) la progettazione dell'interfaccia grafica/esperienza utente tramite l'ausilio di mockup e (*iii*) la progettazione della architettura.

6.2.1 Modellazione del dominio

Seguendo la cultura Domain-Driven Design adottata per la raccolta dei requisiti e per la definizione dei casi d'uso e della terminologia, è necessario individuare i differenti contesti al fine di realizzare una mappa, detta *Context Map*, utile a dare una visione globale del progetto e delle relazioni tra i vari contesti del dominio [4]. La context map è composta dai seguenti tre contesti:

- **Reader** (Core) - Contesto principale del progetto. Racchiude tutti gli aspetti con maggiore valore per l'utente riguardanti la lettura e la personalizzazione dei contenuti digitali.
- **Sisred** - Rappresenta il contesto della sorgente dei contenuti digitali Maggioli. In questo contesto non esistono i concetti di *favorite*, *highlight*, *bookmark* e *progression* mentre è condiviso il concetto di libro e utente.

- **User** - Contesto che definisce tutti gli aspetti a riguardo degli utenti. In questo contesto esiste il solo concetto di utente, il quale è condiviso con gli altri contesti.

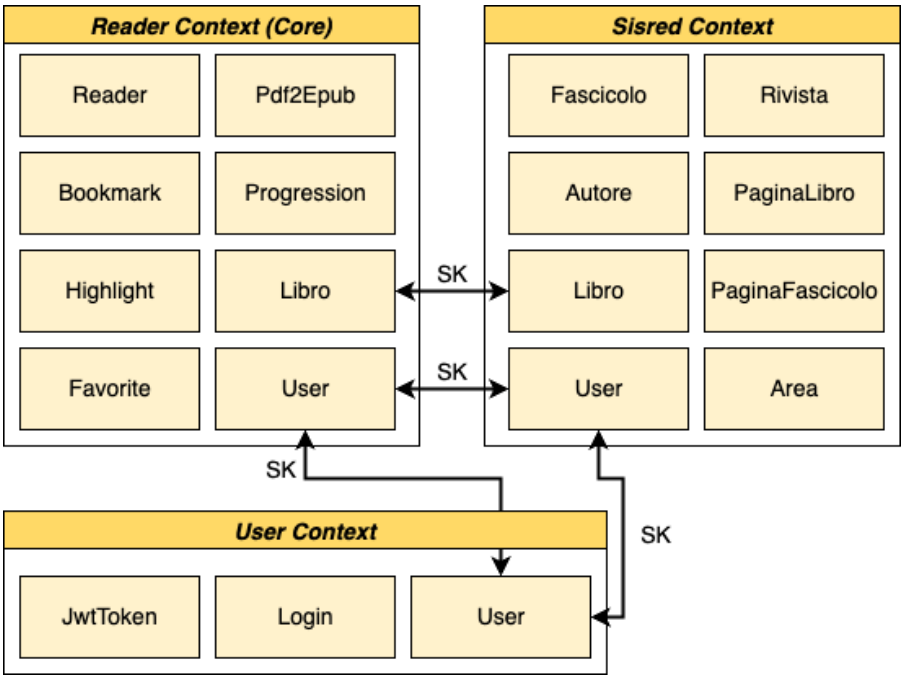

Figura 6.1: Panoramica globale dei contesti del progetto e delle relazioni che intercorrono tra di essi

Tra i contesti definiti esistono delle relazioni di tipo *Shared Kernel* [4] (SK) con l'obiettivo di evitare duplicazioni e semplificare l'integrazione. Relazioni di questo tipo consistono nella condivisione di un sottoinsieme del dominio modellato, che corrisponde tipicamente al dominio core. Un esempio di relazione SK è l'utilizzo di codice o schemi DB condivisi[1].

Per la modellazione del dominio si fa uso dei seguenti concetti [4]:

- **Entity** - Oggetto definito dalla sua identità e non dai suoi attributi. Ogni libro è univoco, identificato da uno specifico codice, chiamato ISBN[2].
- **Value Object** - Al contrario delle entità, questi oggetti sono definiti dai loro attributi e non hanno un'identità concettuale ma servono a descrivere alcune caratteristiche di un oggetto. Un esempio è il segnalibro: ciò che è rilevante è la pagina del libro che esso referenzia e non la sua identità.
- **Aggregate** - Insieme di oggetti legati da un'entità padre chiamata *Root* (radice di aggregazione). L'aggregato composto da *Bookmark*, *Highlight*, *Progression*, *Favorite* ha come radice l'entità *Libro*.

[1] https://github.com/ddd-crew/context-mapping
[2] International Standard Book Number

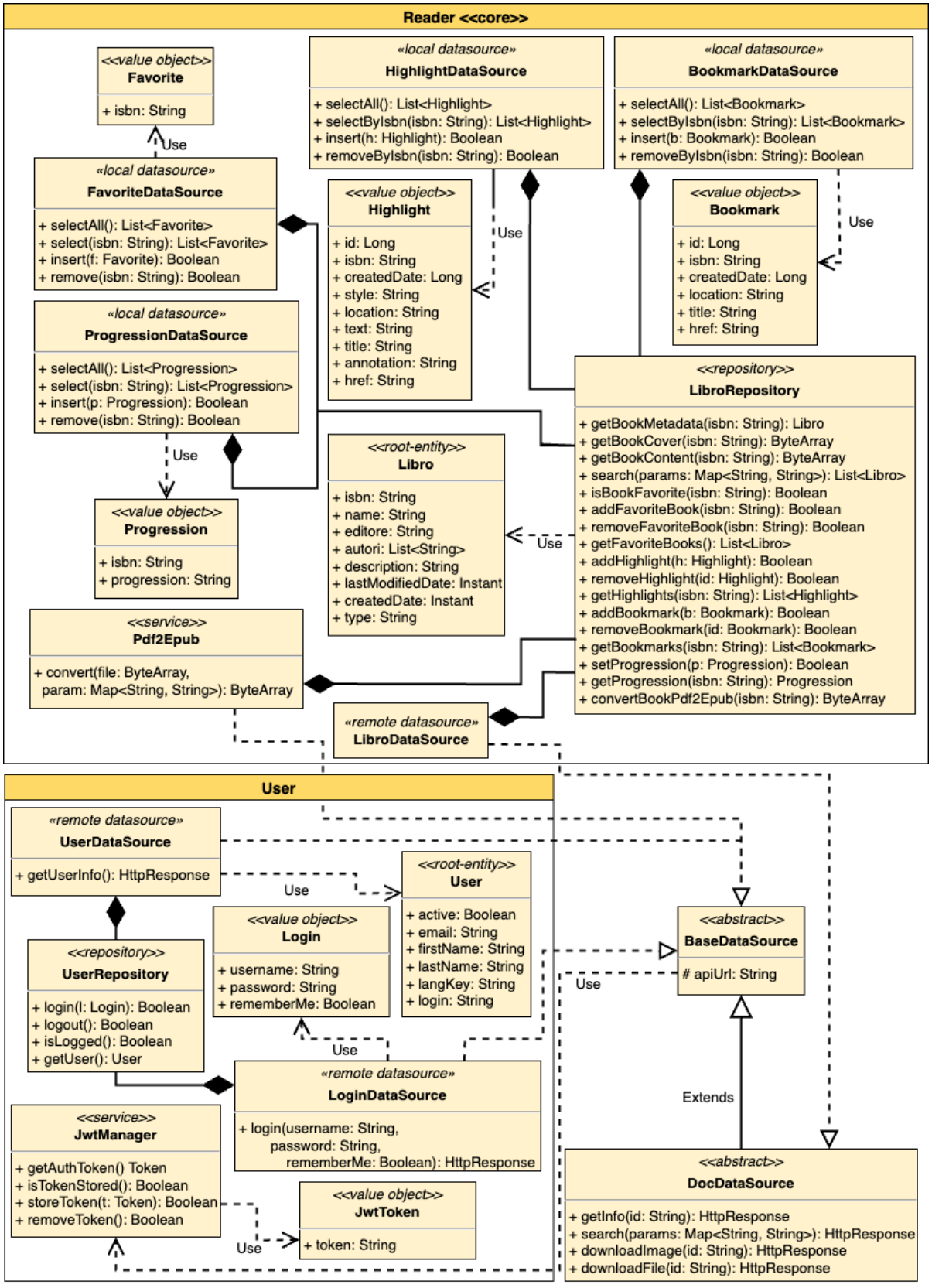

Figura 6.2: UML - Diagramma delle classi: Reader Core Domain e User Subdomain

- **Service** - Operazione che non appartiene logicamente a nessun oggetto. In questo caso la conversione di formato non appartiene alla sola entità libro ma appartiene invece a qualsiasi documento che è possibile convertire.
- **Repository** - Oggetto per il recupero di altri oggetti di dominio e per la gestione del loro ciclo di vita. Le entità *User* e *Libro* sono esempi di oggetti di dominio che necessitano di un repository. Permette di disaccoppiare applicazione e domain design dalle specifiche tecnologie/strategie di persistenza come multipli database e datasource (locali e/o remoti).

6.2.2 Progettazione UI/UX

Successivamente alla modellazione del dominio e alla progettazione dell'architettura, le quali coinvolgono principalmente il modulo condiviso delle due applicazioni che si intende sviluppare grazie al framework KMM, è necessario progettare l'interfaccia grafica e l'esperienza utente che dovrà essere implementata invece per le rispettive piattaforme Android e iOS. Considerando l'utilizzo tipico di un'applicazione con funzionalità simili a quella che deve essere realizzata, possono essere adottati pattern di navigazione e visualizzazione dei contenuti diventati oramai standard de-facto nella progettazione della UI/UX e non solo per le applicazioni mobile[3].

Alcuni esempi di questi elementi utilizzati sono il menu laterale a scomparsa (schermata 4), l'icona "hamburger" per l'apertura del menu (schermata 3), l' elenco di documenti con scroll infinito verticale (schermata 2-5) e la barra di ricerca nella parte alta della schermata con icona "lente di ingrandimento" (schermata 2-5).

Per ottenere la validazione dei mockup da parte del committente sono state necessarie due iterazioni del processo di progettazione UX/UI. L'interfaccia utente desiderata deve infatti soddisfare alcuni vincoli caratteristici del brand Maggioli, come ad esempio l' utilizzo del colore blu #00379E come colore primario, l'utilizzo del font Karla[4] e la presenza del logo Maggioli. Le schermate necessarie per la realizzazione dell'applicazione sono:

- **Reader** - Responsabile della visualizzazione del contenuto digitale in formato EPUB.
- **Login** - Schermata iniziale responsabile dell'autenticazione dell'utente.
- **Home** - Schermata principale responsabile alla visualizzazione dei contenuti digitali a cui l'utente è autorizzato ad accedere.
- **Preferiti** - Responsabile alla visualizzazione dei contenuti digitali preferiti dall'utente.
- **Impostazioni** - Responsabile alla visualizzazione e modifica delle impostazioni.
- **About** - Responsabile alla visualizzazione di informazioni generali come versione dell'applicazione, autore e copyright.

[3] https://xd.adobe.com/ideas/principles/app-design/
[4] https://github.com/googlefonts/karla

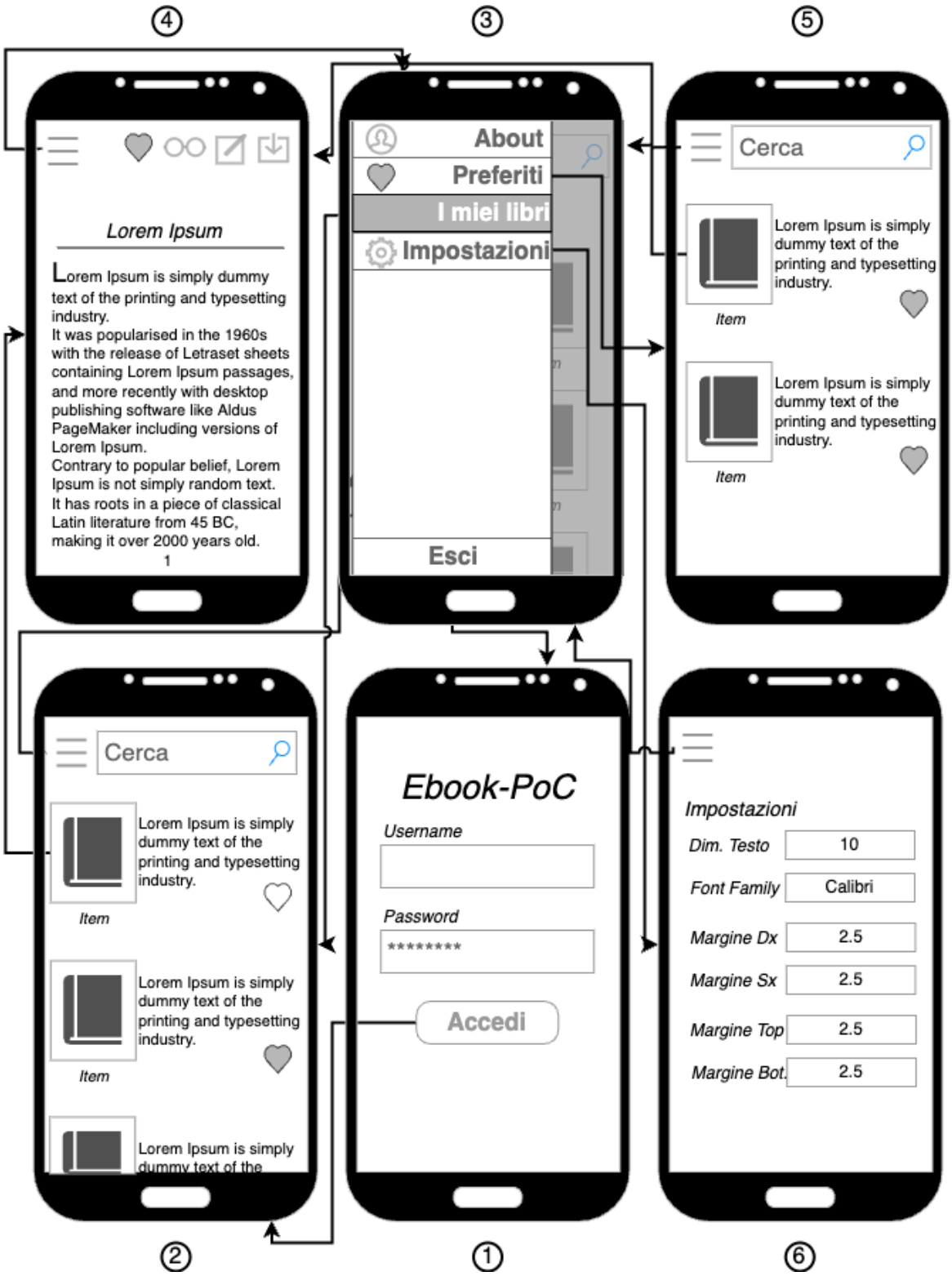

Figura 6.3: Alcuni dei mockup realizzati per la progettazione e la validazione della UX/UI (prima iterazione)

Figura 6.4: Modifiche apportate ai mockup per ottenere la validazione della UX/UI (seconda iterazione)

6.2.3 Architettura

Lo scopo di un'applicazione multipiattaforma è la condivisione della sola logica applicativa come mostrato nella struttura di un'applicazione KMM (fig. 3.1). L'architettura in grado di supportare efficacemente questo concetto è conosciuta come *Clean Architecture* [9] ed è composta da tre strati. Il primo strato (*Data Layer*) consiste negli aggregati di dominio sopra descritti ed è responsabile per il recupero e il mantenimento dei dati su differenti sorgenti. Il secondo strato (*Domain Layer*) facilità la comunicazione tra i differenti repository: qui si trovano tutti i casi d'uso del sistema che gestiscono il flusso dei dati provenienti o diretti verso il primo strato. Il terzo ed ultimo strato (*View Layer*) si occupa dell'interfaccia grafica mostrando i dati e gestendo gli eventi provenienti dall'utente o dal sistema.

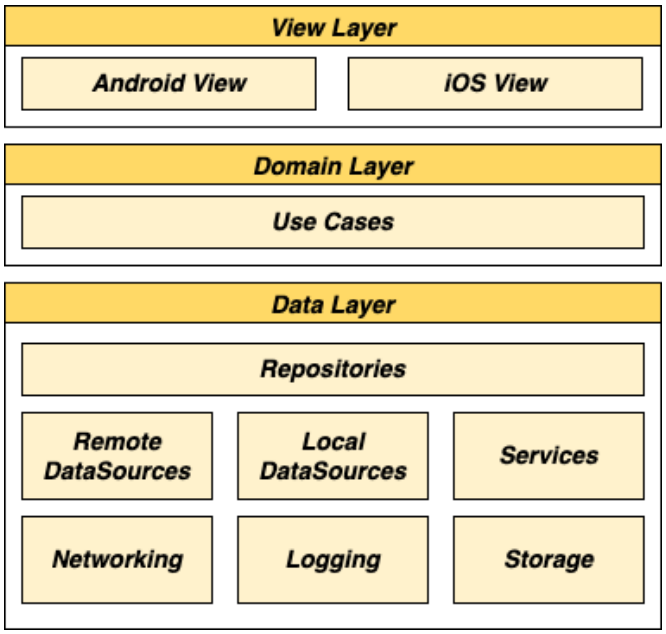

Figura 6.5: Clean Architecture in MaggioliEbook

6.3 Sviluppo

Tipicamente lo sviluppo effettivo di un'applicazione multiplatform con il framework KMM ha inizio con la creazione dello scheletro dell'applicazione tramite l'IDE Android Studio e il relativo plugin Gradle KMM, come indicato nel capitolo 3.

La prima attività successiva alla creazione dello scheletro del progetto consiste nella ricerca delle librerie, ovvero le stesse dipendenze che sono gestite in modo automatico dal sistema d'automazione come descritto nel capitolo precedente. Considerando le funzionalità che devono essere implementate è necessario individuare tutte le librerie utili partendo da quelle core, come la visualizzazione e la modifica dei documenti, arrivando a quelle di utilità a supporto del modulo condiviso come networking e storage. Raggruppate per modulo di utilizzo, le librerie individuate e adottate sono:

Shared

- **Ktor**[5] - Framework asincrono per lo sviluppo di microservizi e applicazioni web, utilizzato per la parte di networking come client HTTP.
- **Kotlinx-Serialization** - Libreria multiplatform per la serializzazione dei dati.
- **Kotlinx-Datetime** - Libreria multiplatform per la gestione delle date e del tempo.
- **Kvault**[6] - Libreria multiplatform per la persistenza dei dati sicura in formato chiave-valore. Tramite un'unica API si comporta come wrapper di *Keychain*[7], nel caso iOS, e *SharedPreferences*[8] nel caso Android.
- **Koin**[9] - Dependency Injection framework multiplatform.
- **Napier**[10] - Logging framework multiplatform.
- **SqlDelight**[11] - Libreria multiplatform per la persistenza dei dati tramite database relazionale locale.

Android

- **Readium** (*Kotlin-toolkit*)[12] - Libreria per la manipolazione e la visualizzazione di pubblicazioni digitali. Implementazione specifica per la piattaforma Android.

iOS

- **Readium** (*Swift-toolkit*)[13] - Libreria per la manipolazione e la visualizzazione di pubblicazioni digitali. Implementazione specifica per la piattaforma iOS.

6.3.1 Readium

La necessità di un tool open-source, robusto e performante per la manipolazione e la lettura di formati editoriali digitali è alla base del progetto Readium. Essa infatti consiste in un insieme di toolkit per diversi formati (come EPUB, audiolibri e libri image-based) e diverse piattaforme (Android, iOS, Desktop e Web).

L'architettura della libreria Readium è composta da tre moduli principali:

- **Publication Server** - Fornisce le pubblicazioni tramite un server locale HTTPS.
- **Streamer** - Modulo composto dai seguenti due sottomoduli:
  - **Parser** - Responsabile del parsing delle pubblicazioni e della loro esposizione utilizzando un modello in-memory.

[5] https://github.com/ktorio/ktor
[6] https://github.com/Liftric/KVault
[7] https://developer.apple.com/documentation/security/keychain_services
[8] https://developer.android.com/reference/android/content/SharedPreferences
[9] https://github.com/InsertKoinIO/koin
[10] https://github.com/AAkira/Napier
[11] https://github.com/cashapp/sqldelight
[12] https://github.com/readium/kotlin-toolkit
[13] https://github.com/readium/swift-toolkit

  - **Fetcher** - Si occupa di ottenere i contenuti delle pubblicazioni e della loro manipolazione (in particolare injection di CSS e Javascript nelle risorse HTML).
- **Navigator** - Utile alla navigazione delle risorse di una pubblicazione secondo diverse strategie basate sulla natura della pubblicazione (ebook, audiolibri, ...). Interagisce con il modulo *Streamer* per utilizzare il modello in-memory o per ottenere il manifesto JSON attraverso il modello condiviso (*Shared*).

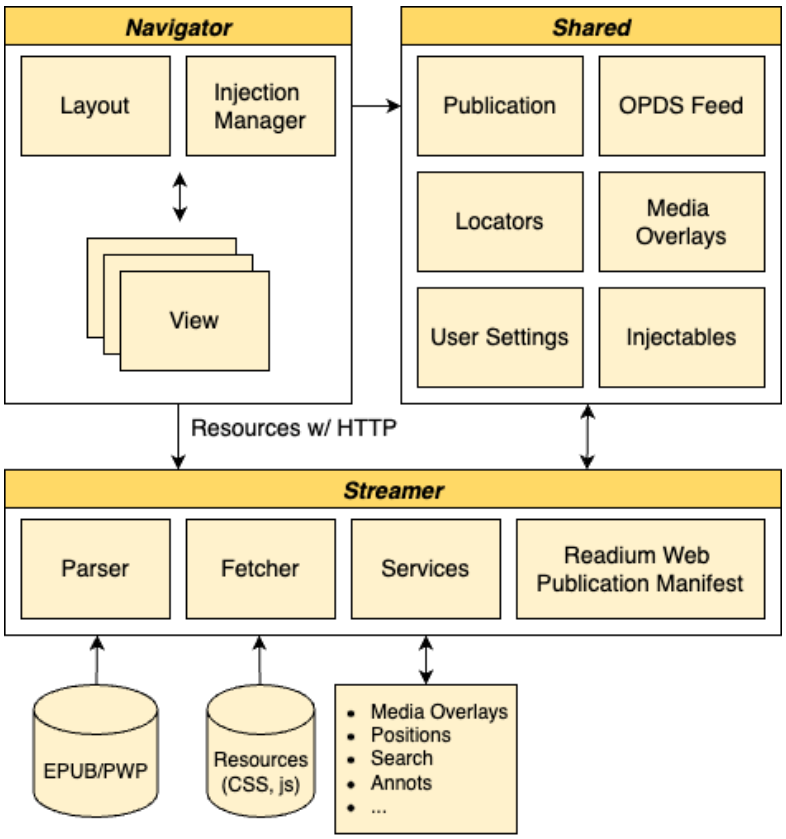

Figura 6.6: Architettura libreria Readium

6.3.2 Modulo Shared

Come anticipato nel capitolo 3 la parte condivisa di un'applicazione multipiattaforma racchiude tutta o una sottoparte della logica applicativa, compresi quindi anche aspetti "infrastrutturali" specifici della piattaforma target come ad esempio logging, persistenza dei dati e networking.

Gli elementi del dominio che sono stati aggiunti dal contesto core (fig. 6.1) devono essere persistiti sul dispositivo locale ed è necessario svolgere operazioni CRUD[14] come su qualsiasi database, che sia locale o remoto. I relativi schemi per il database locale

[14]Create-Read-Update-Delete

sono stati definiti utilizzando appositi file[15], uno per ciascuna delle entità che necessita di persistenza, che sono *Highlight*, *Favorite*, *Progression* e *Bookmark*.

Il seguente codice[16] definisce, tramite la sintassi SqlDelight, lo schema *Bookmark* e l'operazione associata di selezione tramite ISBN:

```
CREATE TABLE bookmark(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created_date INTEGER,
    isbn TEXT NOT NULL,
    publication_id TEXT NOT NULL,
    resource_index INTEGER NOT NULL,
    resource_href TEXT NOT NULL,
    resource_type TEXT NOT NULL,
    resource_title TEXT NOT NULL,
    location TEXT NOT NULL,
    locator_text TEXT NOT NULL
);

selectByIsbn:
    SELECT * FROM bookmark WHERE isbn = ?;
```

Listing 17: Esempio di definizione schema *Bookmark* tramite sintassi SqlDelight

Tramite il task *generateSqlDelightInterface* fornito dal plugin gradle SqlDelight è possibile generare automaticamente l'interfaccia padre *MaggioliEbookDB* e le relative implementazioni. Nel caso dello schema sopra indicato si ottiene il seguente codice[17] Kotlin:

```
public interface BookmarkQueries : Transacter {

  public fun <T : Any> selectByIsbn(isbn: String, mapper: (
    id: Long, created_date: Long?, isbn: String, publication_id: String,
    resource_index: Long, resource_href: String, resource_type: String,
    resource_title: String, location: String, locator_text: String
  ) -> T): Query<T>

  public fun selectByIsbn(isbn: String): Query<Bookmark>
}
```

Listing 18: Codice Kotlin autogenerato per lo schema *Bookmark* tramite SqlDelight

[15] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/shared/src/commonMain/sqldelight/it/filo/maggioliebook/db

[16] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/shared/src/commonMain/sqldelight/it/filo/maggioliebook/db/Bookmark.sq

[17] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/shared/build/generated/sqldelight/code/MaggioliEbookDB/commonMain/it/filo/maggioliebook/db/BookmarkQueries.kt

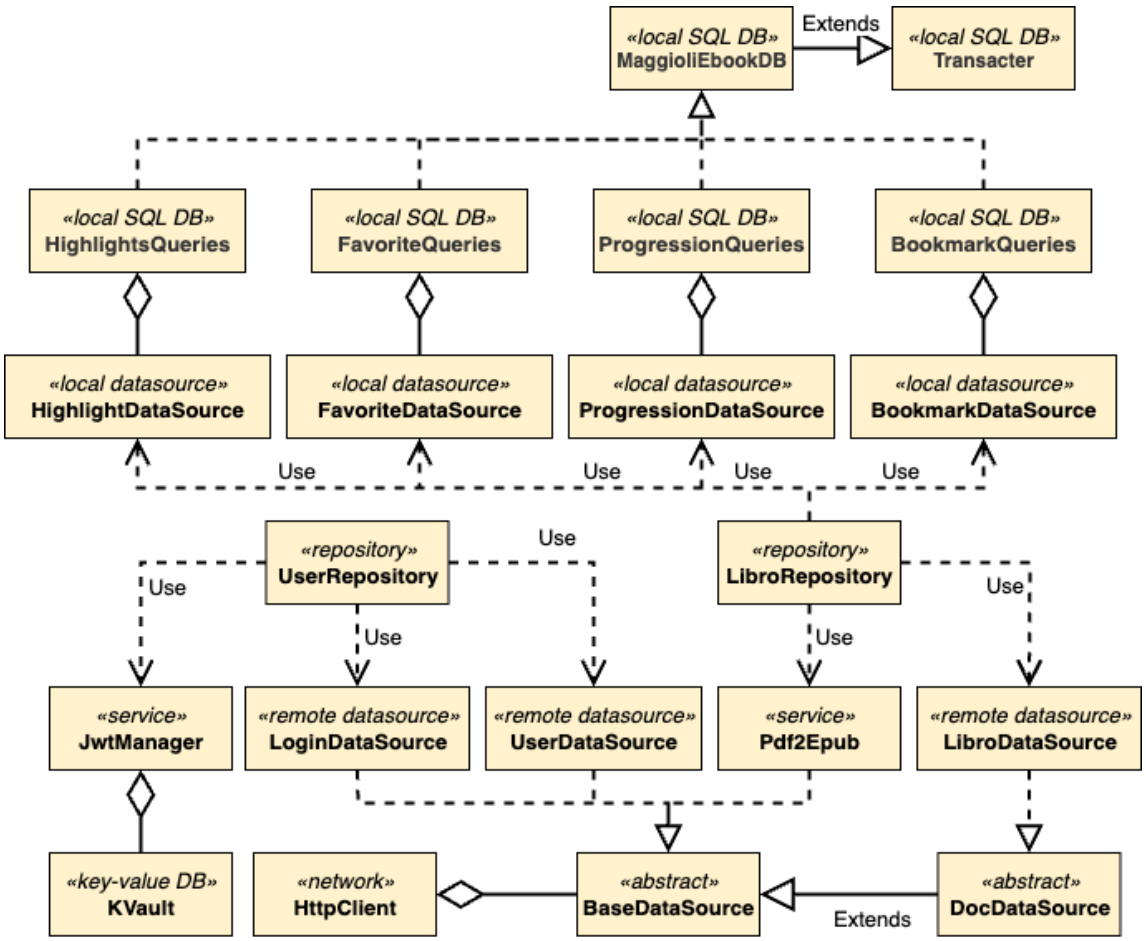

Figura 6.7: UML - Diagramma delle classi: Implementazioni data source (persistenza dati e networking)

Tipicamente le librerie per applicazioni sviluppate con Kotlin Multiplatform possono essere utilizzate direttamente dal codice condiviso, come ad esempio KVault, lasciando al compilatore Kotlin il compito di scegliere la giusta implementazione per la piattaforma target durante la fase di compilazione. In altri casi è necessario invece utilizzare il meccanismo *expect/actual* (discusso nel capitolo 3) per definire il comportamento atteso e fornire un'implementazione specifica per la piattaforma target.

Tale meccanismo è stato utilizzato nello sviluppo dell'applicazione per poter effettuare dependency injection tramite la libreria Koin dei componenti riguardanti la persistenza dei dati (Kvault e SqlDelight) e l'esecuzione asincrona (CoroutineDispatcher). Ogni dipendenza che deve essere iniettata viene inserita all'interno di uno o più moduli, i quali vengono utilizzati da Koin per inizializzare tutto il contesto applicativo, come nel seguente esempio[18] del caso di studio realizzato.

[18]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/shared/src/commonMain/kotlin/it/filo/maggioliebook/di/KoinModule.kt

Le dipendenze possono essere principalmente di due tipi:

- **Factory** - Ogni volta che la dipendenza viene iniettata ne viene creata una nuova istanza (paragonabile al design pattern Factory Method [5]).
- **Single** - In tutto il contesto applicativo esiste una sola istanza della dipendenza iniettata (paragonabile al design pattern Singleton [5]).

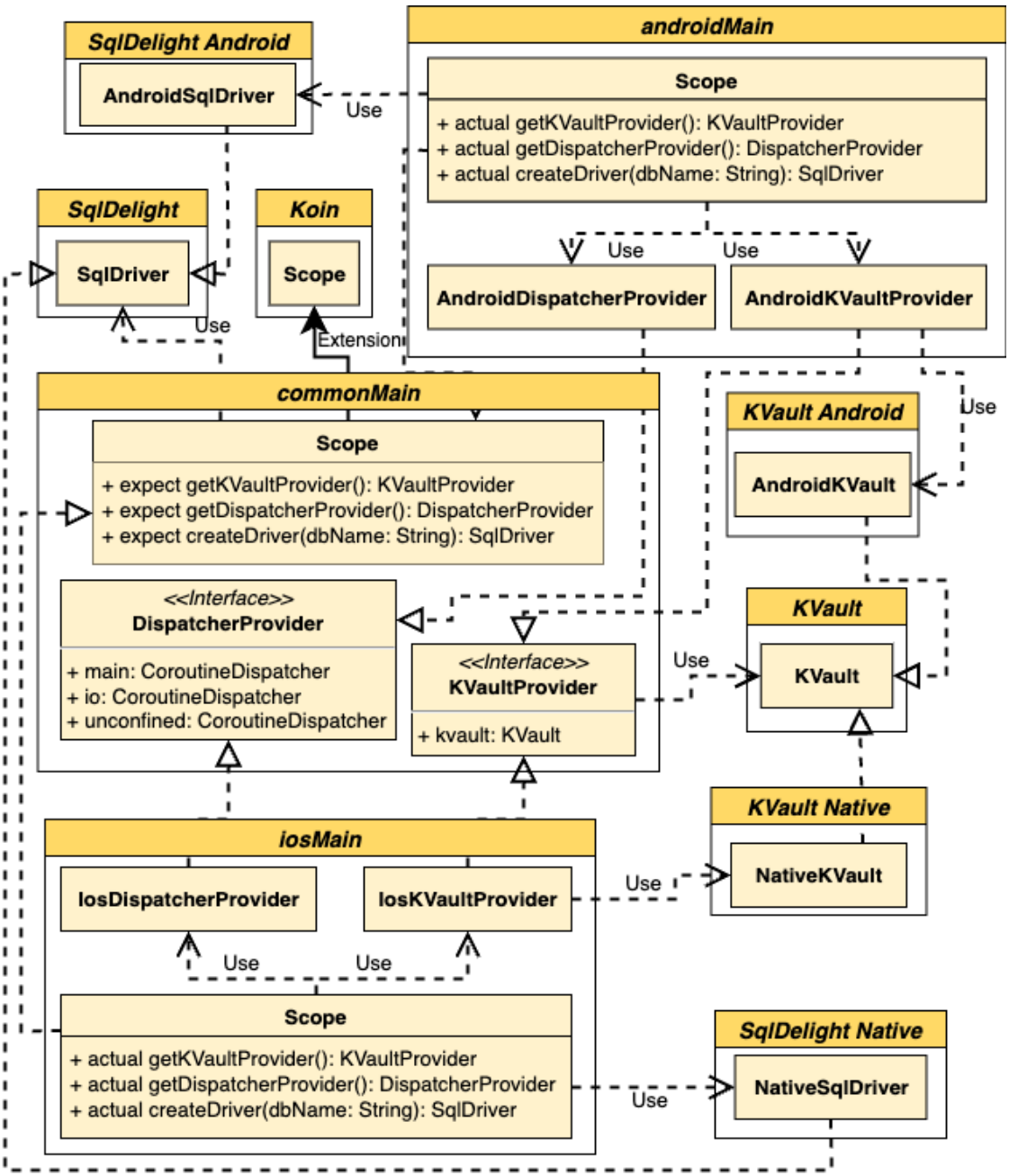

Figura 6.8: Strategia *Expect/Actual* adottata per l'implementazione dei servizi infrastrutturali specifici delle piattaforme (persistenza e concorrenza) sfruttando la tecnica *dependency injection*

```
private val utilModule = module {
    factory { getDispatcherProvider() }
    single { JwtManager(getKVaultProvider().kvault) }
    single { MaggioliEbookDB(createDriver("maggioliebook.db")) }
}

private val dataSourceModule = module {
    factory { FavoriteDataSource(get(), get()) }
    factory { HighlightDataSource(get(), get()) }
    factory { BookmarkDataSource(get(), get()) }
    factory { ProgressionDataSource(get(), get()) }
    factory { LoginDataSource(get(), get()) }
    factory { UserDataSource(get(), get()) }
    factory { LibroDataSource(get(), get()) }
}

private val repositoryModule = module {
    single { LibroRepository() }
    single { UserRepository() }
}

fun initKoin(appDeclaration: KoinAppDeclaration = {}) = startKoin {
    appDeclaration()
    modules(listOf(utilModule, dataSourceModule, repositoryModule))
}
```

Listing 19: Configurazione Dependency Injection: definizione dei moduli Koin e inizializzazione del contesto applicativo

6.3.3 Applicazione Android

Completato lo sviluppo della logica applicativa condivisa e quindi degli strati *Domain* e *Data* (fig. 6.5) dell'architettura, è possibile iniziare lo sviluppo dello strato *View*. In questa fase si realizza la vera e propria applicazione Android, sfruttando il toolkit Kotlin della libreria Readium per la funzionalità core di lettura dei contenuti digitali.

Single-Activity Architecture

L'architettura scelta per l'implementazione dell'applicazione Android[19] è basata su sole due activity:

- **MainActivity** - Unica activity principale dell'applicazione.
- **ReaderActivity** - Activity secondaria, utilizzata sia per l'interazione che la gestione del lettore delle pubblicazioni digitali.

[19]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/androidMaggioliEbookApp/src/main/java/it/filo/maggioliebook/android

Questa tipologia di architettura permette di avere una singola activity che svolge la funzione di contenitore per tutti i fragment che rappresentano le varie schermate, definite in fase di progettazione della UI/UX, in cui l'utente può navigare.

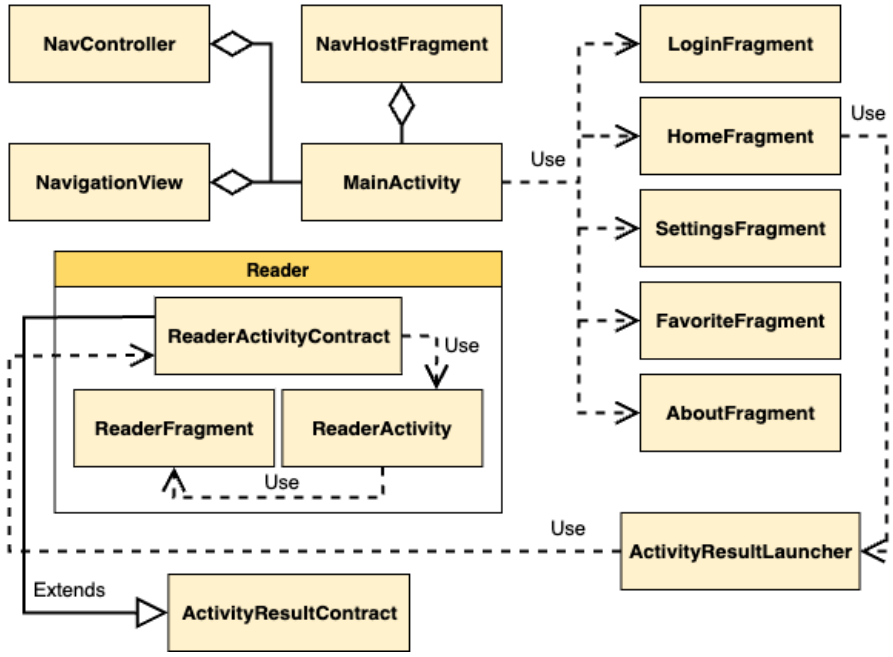

Figura 6.9: Architettura Single-Activity adottata per l'applicazione Android

La navigazione all'interno dell'applicazione è gestita tramite la combinazione dei seguenti componenti:

- **NavController** - Componente necessario per la gestione delle transizioni da un fragment all'altro. Quando inizializzato, richiede la presenza di un grafo di navigazione tra le risorse XML: tale grafo contiene tutti i fragment che possono essere visualizzati e le possibili transizioni tra di essi.
- **NavHostFragment** - Rappresenta il contenitore del fragment in cui ci si trova. Ogni transizione nel grafo corrisponde alla sostituzione del fragment visualizzato con quello di destinazione (sempre che la transizione sia ammessa dal grafo).
- **NavigationView** - Permette la visualizzazione del menu dove si trovano i possibili fragment raggiungibili da quello in cui l'utente si trova attualmente. Nel caso dell'applicazione sviluppata, in questo componente si trova il menu laterale con tutte le schermate indicate in fase di progettazione (*Home*, *Settings*, *About* e *Preferiti*).

Come mostrato nella seguente figura (fig. 6.10), la schermata principale (*Home*) rappresenta la destinazione iniziale del grafo, ovvero la prima schermata che viene visualizzata dall'applicazione. D a questa schermata è poi possibile effettuare transizioni verso altre schermate sfruttando il meccanismo chiamato *navigazione condizionale*[20]. Prima di creare la vista del fragment viene effettuato un controllo sull'autenticazione dell'utente: se

[20]https://developer.android.com/guide/navigation/navigation-conditional

l'utente non è autenticato viene effettuata una transizione al fragment *Login* per permettere all'utente di autenticarsi e procedere all'utilizzo dell'applicazione. E' fondamentale in questo scenario ripulire lo stack di navigazione per evitare che l'utente possa "tornare indietro" alla schermata precedente, ovvero la schermata principale, senza effettuare l'autenticazione.

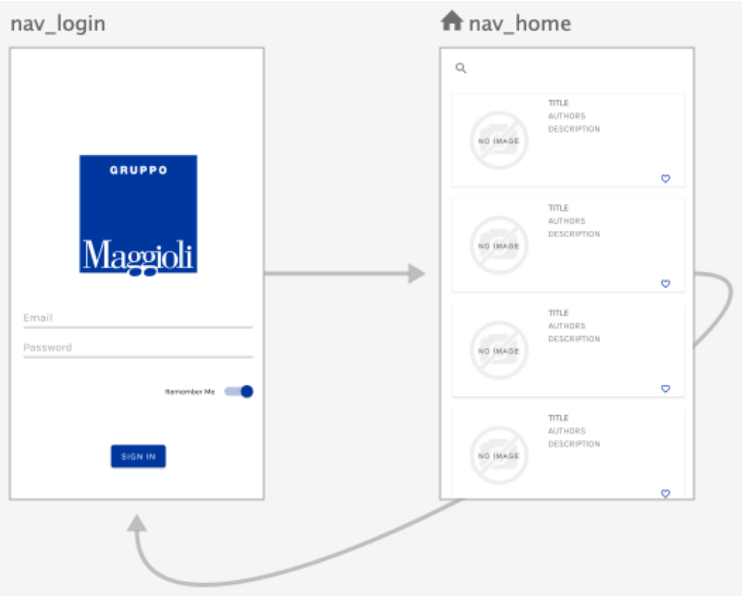

Figura 6.10: Sottoparte del navigation graph dell'applicazione, renderizzato automaticamente tramite l'IDE Android Studio

Paging

Un altro aspetto importante dell'architettura è il meccanismo utilizzato sia dalla schermata principale che da quella dei preferiti per mostrare i dati all'utente. Il backend utilizzato fornisce i dati tramite paginazione: in questo modo è possibile indicare in una richiesta la dimensione delle pagine, ovvero quanti elementi devono essere restituiti al massimo in una pagina, e la pagina desiderata. Per poter caricare tali dati dinamicamente in una *RecyclerView*[21], in modo da permettere all'utente di effettuare lo scroll "infinito", è stata utilizzata[22] la libreria *Paging*[23] fornita nativamente dall'SDK Android.

I componenti fondamentali della libreria *Paging* sono:

- **Pager** - Punto d'ingresso principale della libreria con il compito di ottenere nuovi dati quando necessario, ovvero quando l'utente ha raggiunto nella schermata uno specifico punto. Richiede la configurazione di alcuni parametri come la dimensione della pagina, la pagina iniziale e la direzione della paginazione.

[21] https://developer.android.com/reference/kotlin/androidx/recyclerview/widget/RecyclerView
[22] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/androidMaggioliEbookApp/src/main/java/it/filo/maggioliebook/android/home/LibroPagingSource.kt
[23] https://developer.android.com/topic/libraries/architecture/paging/v3-overview

- **PagingSource** - Sorgente dei dati paginati. Componente interrogato dal *Pager* ogni volta che sono necessari nuovi dati.
- **PagingDataAdapter** - Componente fondamentale per la rappresentazione di dati paginati in una *RecyclerView*.

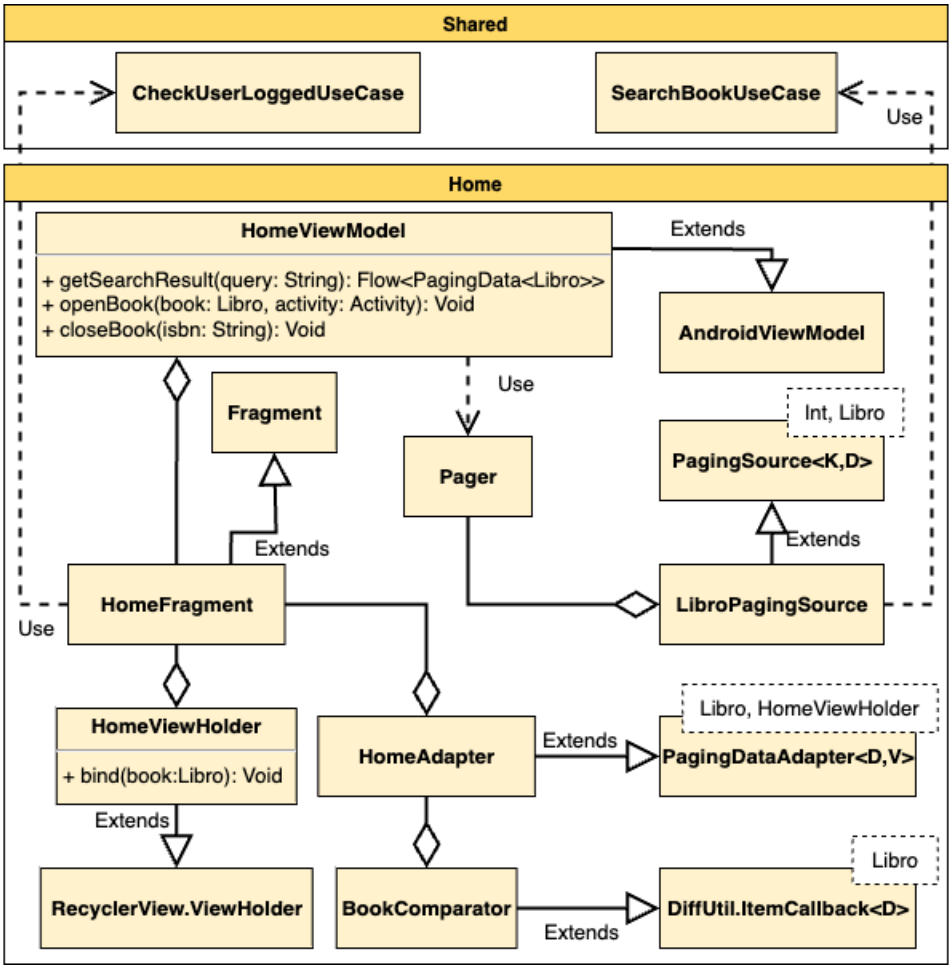

Figura 6.11: UML - Diagramma delle classi: Paginazione dati nella schermata principale

Reader

Il lettore dei documenti digitali in formato EPUB rappresenta la funzionalità core dell'applicazione sviluppata. Nel caso dell'interfaccia grafica Android, il lettore riceve il contenuto da visualizzare del documento selezionato dall'utente e lo mostra in un fragment basato sul componente *Navigator* (fig. 6.6). Questo componente della libreria Readium fornisce tutte le funzionalità necessarie per gestire la navigazione dei documenti e gli eventi relativi al comportamento dell'utente come scorrimento pagine e selezione testo. I principali componenti sviluppati[24] per le funzionalità del reader sono:

[24]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/androidMaggioliEbookApp/src/main/java/it/filo/maggioliebook/android/reader

- **ReaderActivity** - Unica activity presente nell'applicazione oltre a quella principale, utilizzata sia per l'interazione che per la gestione del lettore di contenuti digitali.

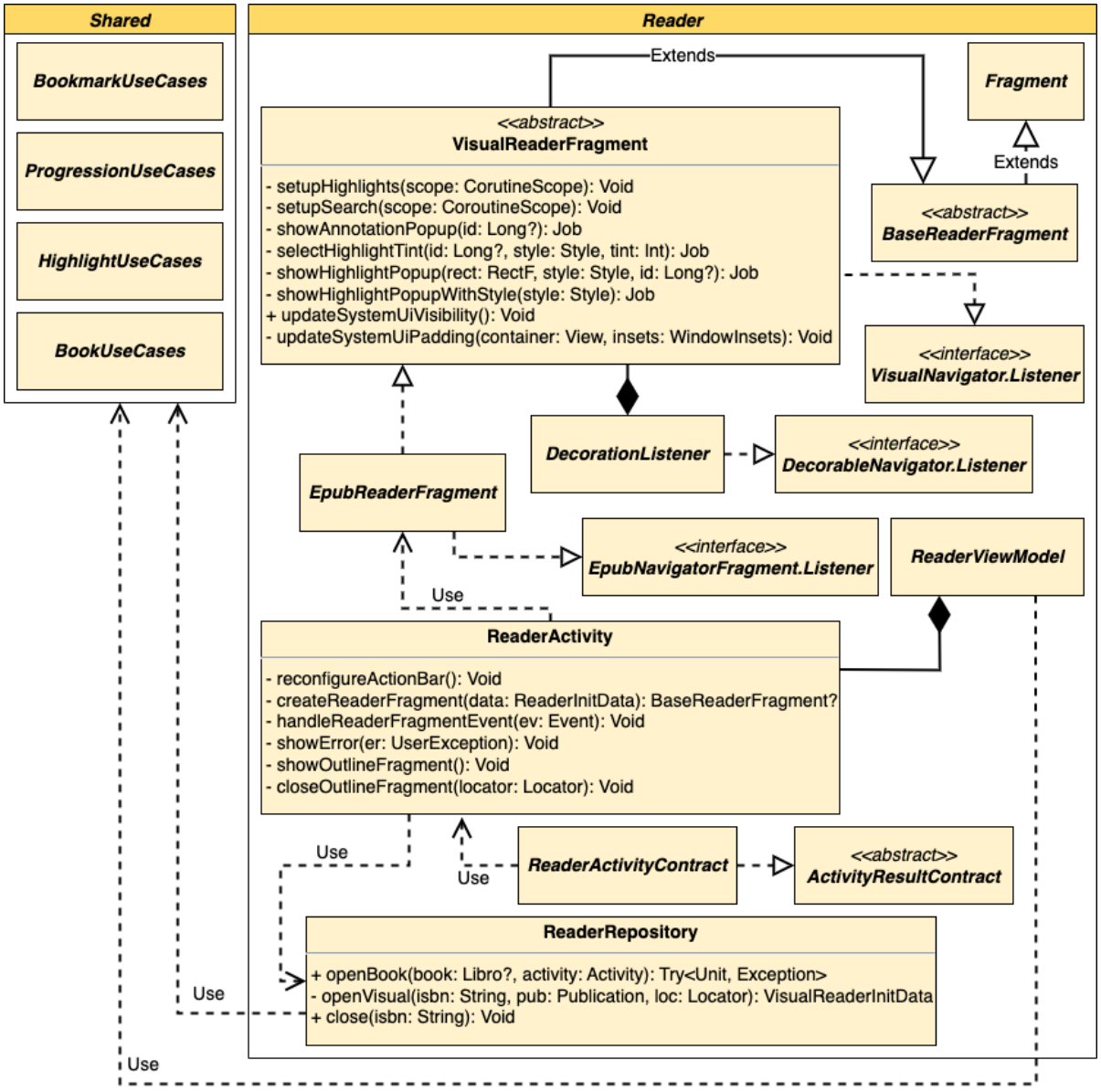

Figura 6.12: UML - Diagramma delle classi: Reader

- **EpubReaderFragment** - Componente che si occupa effettivamente di mostrare il contenuto del documento all'utente.
- **DecorationListener** - Permette la gestione grafica delle annotazioni come evidenziazioni e sottolineature, le quali sono aggiunte alla vista del fragment tramite l'utilizzo di Jetpack Compose.

Screenshot

Figura 6.13: Login

Figura 6.15: Schermata home principale

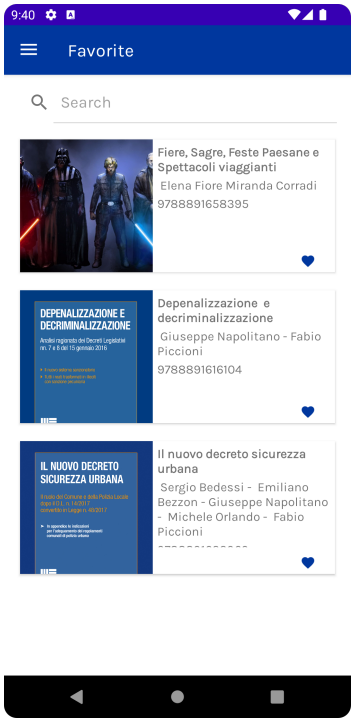

Figura 6.17: Preferiti

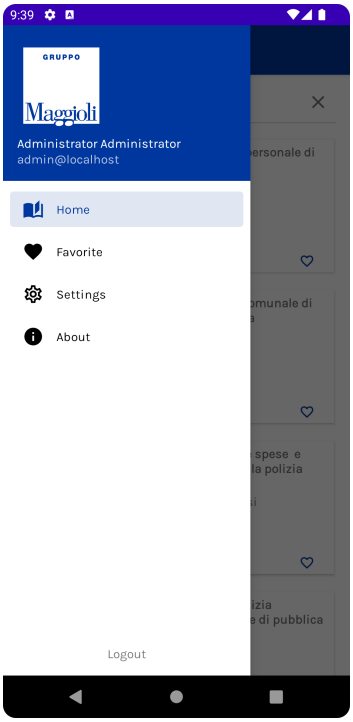

Figura 6.14: Menu laterale - Sidenav

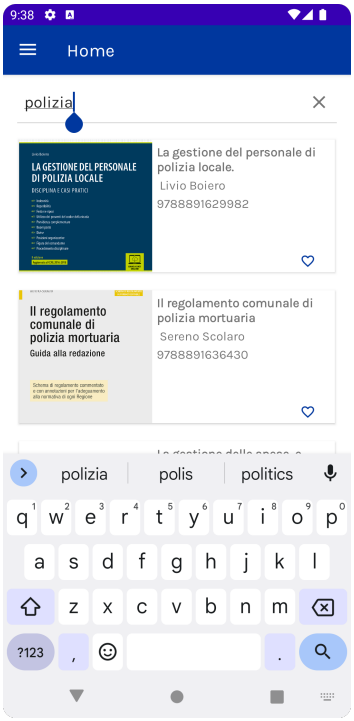

Figura 6.16: Esempio di ricerca

Figura 6.18: Info generali (About)

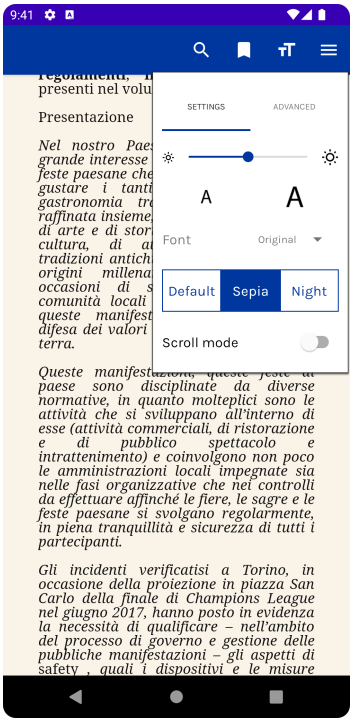

Figura 6.19: Reader

Figura 6.21: Annotazioni

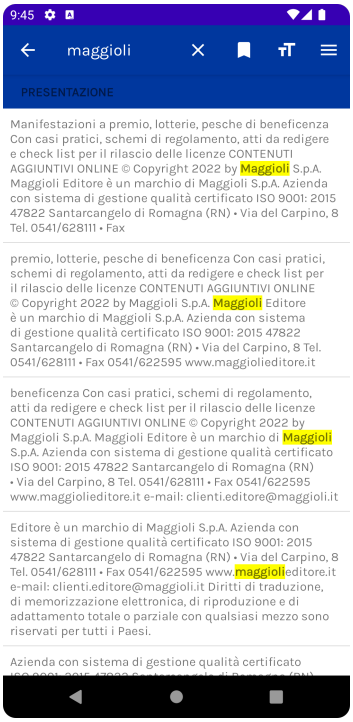

Figura 6.23: Ricerca testuale nel contenuto del documento

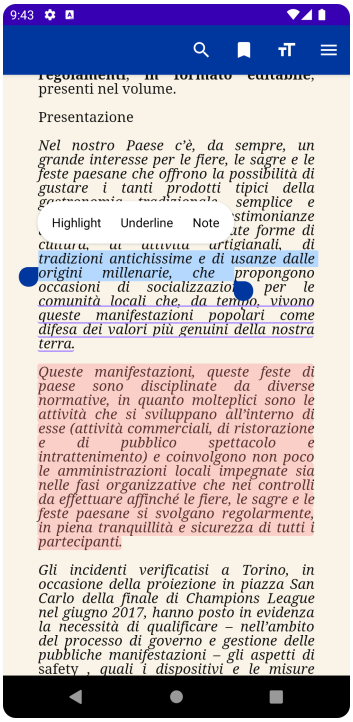

Figura 6.20: Esempio di annotazioni, evidenziazioni, sottolineature, ...

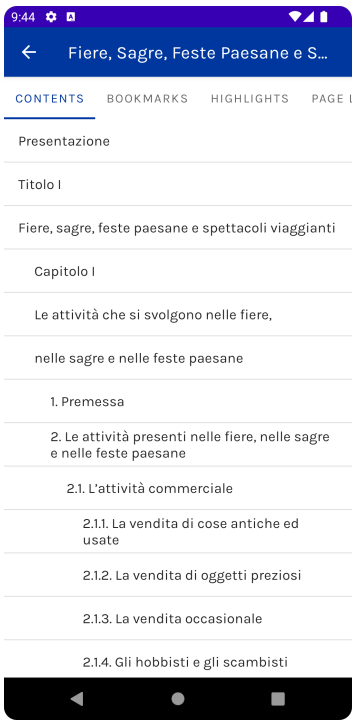

Figura 6.22: Elenco dei contenuti (TOC) del documento

Figura 6.24: Evidenziazioni automatiche delle occorrenze del testo cercato

6.3.4 Applicazione iOS

Terminato lo sviluppo dell'applicazione Android è possibile dedicarsi all'applicazione iOS, per la quale rimangono valide le stesse considerazioni fatte sul processo e sull'architettura del caso di studio ma con la necessità di qualche intervento per il corretto funzionamento del framework KMM. Il progetto XCode[25] di partenza per l'applicazione iOS è stato generato all'inizio della fase di sviluppo con l'inizializzazione dell'intero progetto tramite il plugin Gradle KMM per l'IDE Android Studio.

Per poter utilizzare il codice Kotlin del modulo condiviso nel codice Swift dell'applicazione iOS è necessario generare due tipi di pacchetto: (*i*) framework di sviluppo per l'IDE XCode e (*ii*) libreria CocoaPods da includere come dipendenza di progetto. Il framework di sviluppo può essere creato tramite l'apposito task *embedAndSignPodAppleFramework-ForXCode* disponibile tramite il plugin Gradle KMM. Questo task permette di definire la modalità di compilazione, la versione e la posizione del framework.

CocoaPods

Come anticipato nel capitolo 3, CocoaPods è un dependency manager per progetti Swift e Objective-C scritto in Ruby. Le dipendenze, chiamate *Pods*, sono definite nel file *Podfile* nella root di progetto. Il pod shared viene generato durante la fase di impacchettamento del modulo shared tramite l'esecuzione del task Gradle *assemble* per il quale sono definiti appositi sotto-task per svolgere questo compito.

Il seguente codice[26], ovvero il contenuto del Podfile del progetto, definisce le dipendenze incluse nell'applicazione iOS sviluppata:

```
target 'iosMaggioliEbookApp' do
  use_frameworks!
  platform :ios, '14.1'

  # KMM Shared Business Logic Module
  pod 'shared', :path => '../shared'

  # Readium swift-toolkit
  pod 'R2Shared'
  pod 'R2Streamer'
  pod 'R2Navigator'
  pod 'ReadiumOPDS'

  pod 'MBProgressHUD', '~> 1.2'
  pod 'GCDWebServer'
end
```

Listing 20: Dipendenze CocoaPods dell'applicazione iOS sviluppata

[25] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/tree/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp

[26] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/Podfile

A questo punto è possibile importare i moduli nel codice Swift e sviluppare l'applicazione utilizzando le funzionalità fornite dall'IDE XCode come per una qualsiasi libreria Swift importata. I task forniti dal plugin Gradle KMM a supporto dello sviluppo iOS tramite il dependency manager CocoaPods sono:

- **podDownload** - Scarica le dipendenze CocoaPods senza effettuarne l'installazione.
- **podImport** - Task di utility utilizzato da altri task per l'import delle dipendenze.
- **podInstall** - Esegue l'installazione dei pod, in modo equivalente all'esecuzione del comando *pod install*.
- **podPublishDebugXCFramework** - Crea il framework XCFramework in modalità debug (chiamato dal task *embedAndSignPodAppleFrameworkForXcode*).
- **podPublishReleaseXCFramework** - Crea il framework XCFramework in modalità release (chiamato dal task *embedAndSignPodAppleFrameworkForXcode*).
- **podPublishXCFramework** - Crea i framework XCFrameworc in entrambe le modalità.
- **podspec** - Genera il file *podspec*[27], utile all'importazione del pod nell'applicazione iOS.

Navigator Architecture

L'architettura dell'applicazione iOS sviluppata segue lo stesso pattern utilizzato nella versione Android ed è conosciuto col nome *Navigator*. Tramite il costrutto *NavigationView*[28] è possibile definire una schermata "contenitore" per le viste dei percorsi di navigazione all'interno dell'applicazione.

Le schermate da realizzare sono le stesse individuate nella progettazione della UX/UI ed è necessario utilizzare i *NavigationLink*[29] per poterle navigare nell'architettura scelta. I NavigationLink, i quali identificano una specifica transizione di schermata da quella attuale a quella di destinazione, sono definiti nel componente *NavController*[30] ma vengono attivati dal componente *Sidebar*[31]. A differenza di Jetpack Compose, i l toolkit utilizzato nello sviluppo della UI Android, SwiftUI non fornisce nativamente dei costrutti per la realizzazione di schermate con scroll infinito e dati paginati (come descritto nella sezione 6.3.3): per questo motivo è stata necessaria l'implementazione di un meccanismo ad-hoc[32] per la gestione di questo scenario.

[27] https://guides.cocoapods.org/syntax/podspec.html
[28] https://developer.apple.com/documentation/swiftui/navigationview
[29] https://developer.apple.com/documentation/swiftui/navigationlink
[30] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/iosMaggioliEbookApp/Common/NavController.swift
[31] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/iosMaggioliEbookApp/Common/Sidebar.swift
[32] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/iosMaggioliEbookApp/Home/HomeViewModel.swift

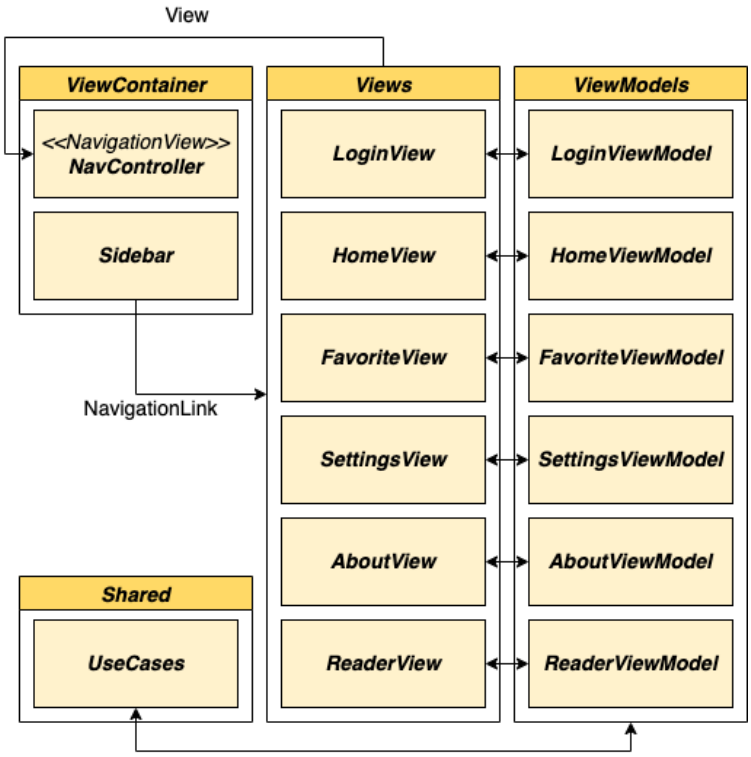

Figura 6.25: Architettura Navigator adottata per l'applicazione iOS

Integrazione iOS-Shared

Grazie al framework Kotlin Multiplatform Mobile e ai realtivi plugin Gradle di supporto è stato possibile completare con successo una serie di compilazioni (fig. 3.2) in grado di "trasformare" codice Kotlin in codice Swift. Nonostante il codice Kotlin venga compilato correttamente in codice Swift, per ottenere codice funzionante in grado di svolgere le stesse funzionalità sviluppate nell'applicazione Android è stato necessario svolgere dell'ulteriore lavoro d'integrazione, riguardante principalmente la computazione asincrona e i tipi di dato:

- Realizzazione di un metodo specifico, chiamato *invokeNative*[33], nel modulo condiviso per l'invocazione dei casi d'uso. Le *coroutines* di Kotlin vengono infatti mappate da KMM in funzioni con callbacks, chiamate *completion handlers* e non esiste modo di definire un *CoroutineScope* in Swift. In questo modo è stato invece possibile definire uno scope per l'esecuzione del codice asincrono anche su target iOS.
- Impostazione del nuovo modello sperimentale di gestione della memoria nel file *gradle.properties*[34] per il modulo condiviso tramite la configurazione del parametro

[33]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/shared/src/commonMain/kotlin/it/filo/maggioliebook/usecase/core/ConvertPdf2EpubUseCase.kt

[34]https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/gradle.properties

kotlin.native.binary.memoryModel=experimental[35].

- Realizzazione di un metodo[36] di conversione da *ByteArray* Kotlin a *NSData* Swift in modo da permettere la corretta gestione di dati binari, come ad esempio le immagini delle copertine e il contenuto EPUB delle pubblicazioni digitali nello specifico del caso di studio.
- Realizzazione di un metodo[37] di gestione delle date e del tempo equivalente a quello utilizzato nel dominio modellato, in particolare per i tipi di dato utilizzati durante la persistenza dei dati tramite SqlDelight.
- Estensione[38] delle entità di dominio per indicare l'identificativo utilizzato da Swift per ciclare su liste di elementi di quel tipo.

Grazie al lavoro aggiuntivo svolto è stato possibile utilizzare efficacemente il modulo condiviso all'interno dell'applicazione iOS, come ad esempio nel seguente caso[39] asincrono di ottenimento e conversione della copertina di una pubblicazione digitale:

```
1  import shared
2
3  final class CoverLoader: ObservableObject {
4      @Published var data: Data?
5
6      init(_ isbn: String) {
7          GetBookCoverUseCase.init().invokeNative(isbn: isbn) { result in
8              DispatchQueue.main.async {
9                  if result != nil {
10                     self.data = ByteArrayConverter.init().toData(byteArray: result!)
11                 }
12             }
13         } onError: { throwable in
14             print(throwable.getStackTrace())
15             self.data = Data()
16         }
17     }
18 }
```

Listing 21: Esempio di utilizzo dei casi d'uso del modulo condiviso (computazione asincrona e gestione di dati binari)

[35] https://kotlinlang.org/docs/native-memory-manager.html

[36] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/shared/src/iosMain/kotlin/it/filo/maggioliebook/util/extensions/ByteArrayConverter.kt

[37] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/iosMaggioliEbookApp/Common/Extensions/Date.swift

[38] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/iosMaggioliEbookApp/Common/Extensions/Libro.swift

[39] https://github.com/paganellif/DevOps-per-applicazioni-mobile-un-caso-di-studio-industriale/blob/6-sviluppo-applicazione-maggioliebook/maggioliEbookApp/iosMaggioliEbookApp/iosMaggioliEbookApp/Common/CoverLoader.swift

Screenshot

Figura 6.26: Login

Figura 6.28: Schermata home principale

Figura 6.30: Ricerca testuale nel contenuto

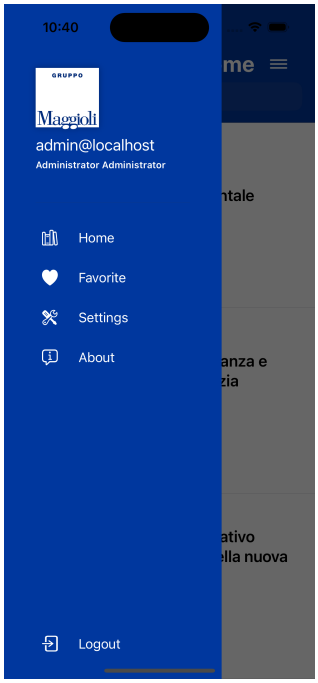

Figura 6.27: Menu laterale - Sidenav

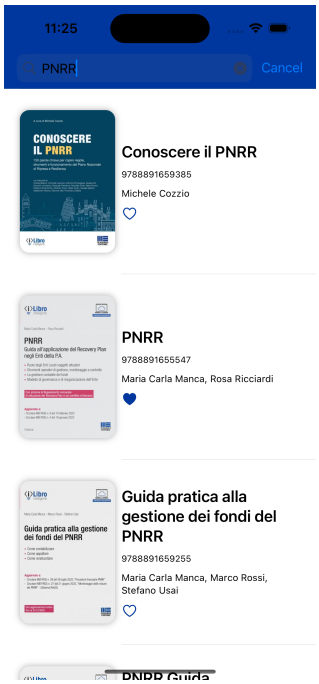

Figura 6.29: Esempio di ricerca

Figura 6.31: Schermata dei preferiti

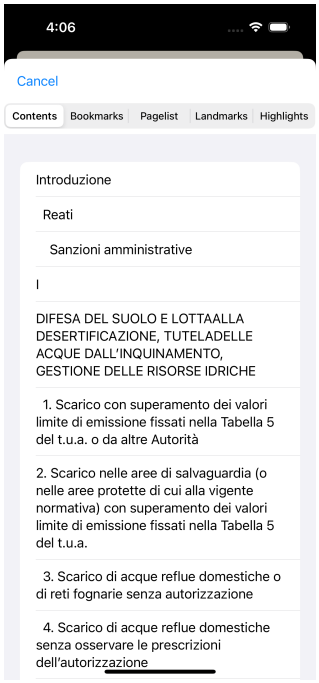

Figura 6.32: Elenco dei contenuti (TOC) del documento

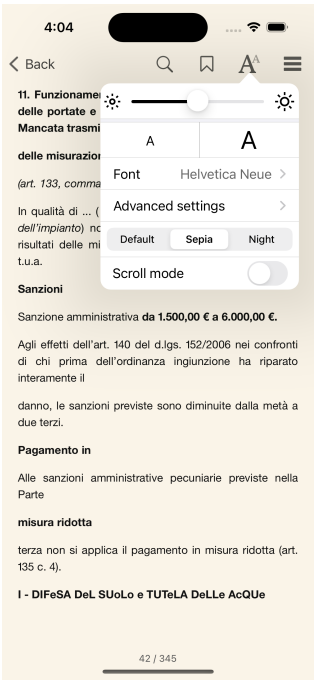

Figura 6.33: Reader

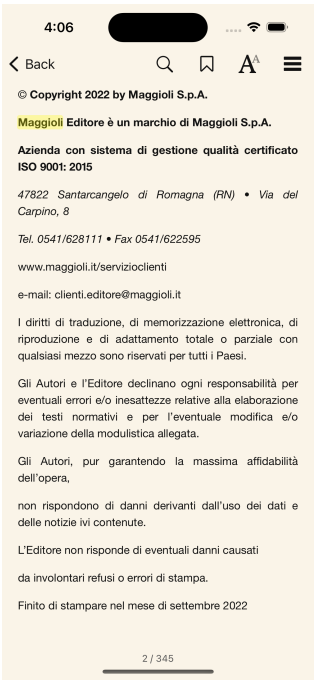

Figura 6.34: Evidenziazioni automatiche delle occorrenze del testo cercato

Capitolo 7

Risultati raggiunti

7.1 Introduzione

In questo capitolo vengono mostrati i risultati ottenuti dalla realizzazione dell'applicazione multipiattaforma MaggioliEbook adottando il processo di sviluppo descritto nel capitolo 5.

Vengono inizialmente considerati i requisiti definiti nel capitolo 4 e paragonati con quanto realizzato. Successivamente sono indicate alcune statistiche e metriche, raccolte principalmente tramite la piattaforma GitLab adottata, al fine di creare un primo modello di confronto per i lavori futuri che saranno svolti in azienda sulle tematiche principali di questo caso di studio industriale.

7.2 Riuso

I template che definiscono gli stage e i relativi job del processo di sviluppo progettato sono stati versionati e organizzati in un apposito repository, come descritto in modo dettagliato nel capitolo 5.

Questa soluzione non solo permette il riuso dell'intera pipeline o di solamente una sua parte, ma abilita anche un processo di lavoro collaborativo fra tutti i possibili utilizzatori per la modifica del processo. Ogni sviluppatore all'interno dell'azienda è infatti in grado di accedere al repository dei template, visionare i file YAML che definiscono la pipeline e aprire merge request per richiedere la modifica o l'aggiunta di funzionalità necessarie al processo di sviluppo automatizzato. A tal proposito è stato scelto un team di sviluppo all'interno dell'azienda che si occupa di applicazioni mobile per svolgere un primo esperimento d'integrazione nel proprio processo di sviluppo, già consolidato su tecnologie puramente native e senza alcun sistema d'automazione.

7.3 Stabilizzazione e rilascio

La fase di stabilizzazione e rilascio di entrambe le versioni di applicazione sviluppate tramite Kotlin Multiplatform Mobile sono state realizzate rispettando tutti i vincoli definiti nel capitolo 4. Durante il processo di sviluppo del caso di studio è stato infatti possibile rilasciare automaticamente applicazioni, sia Android che iOS, sfruttando la pipeline realizzata. In entrambi i casi sono stati definiti gruppi di tester composti sia dalle figure aziendali esperte di dominio che dai Professori relatori di questa tesi:

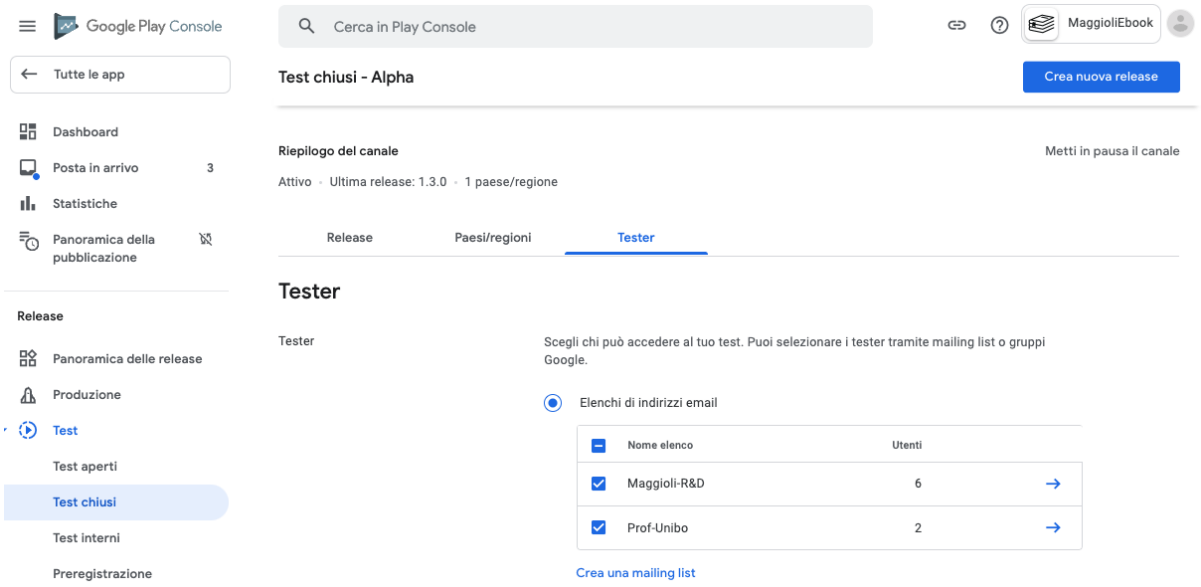

Figura 7.1: Schermata Google Play Console per la fase di stabilizzazione dell'applicazione Android

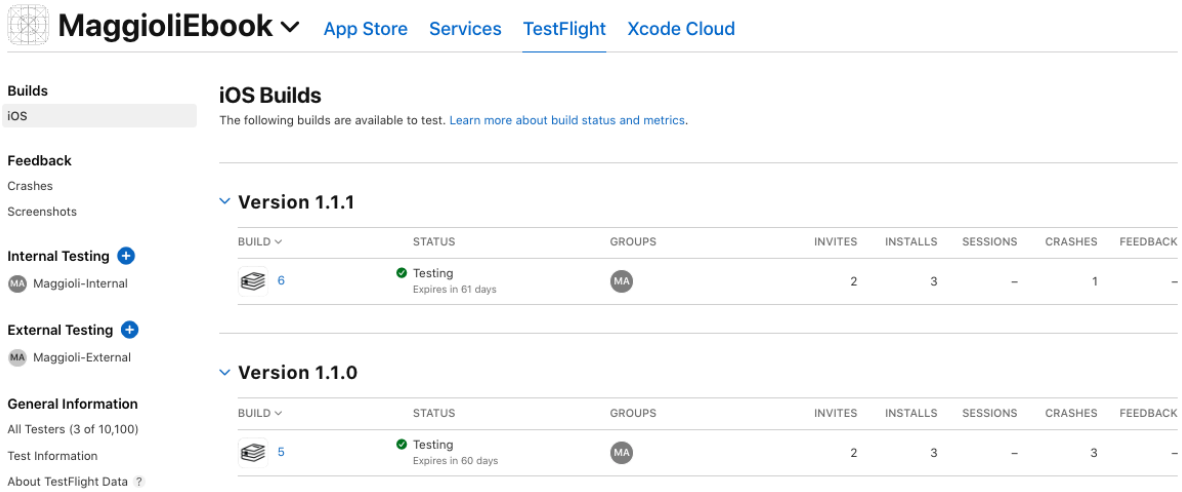

Figura 7.2: Schermata App Store Connect per la fase di stabilizzazione dell'applicazione iOS

Tutti i tester appartenenti ai gruppi configurati come nelle schermate precedenti sono poi stati in grado di installare con successo l'applicazione sul proprio dispositivo:

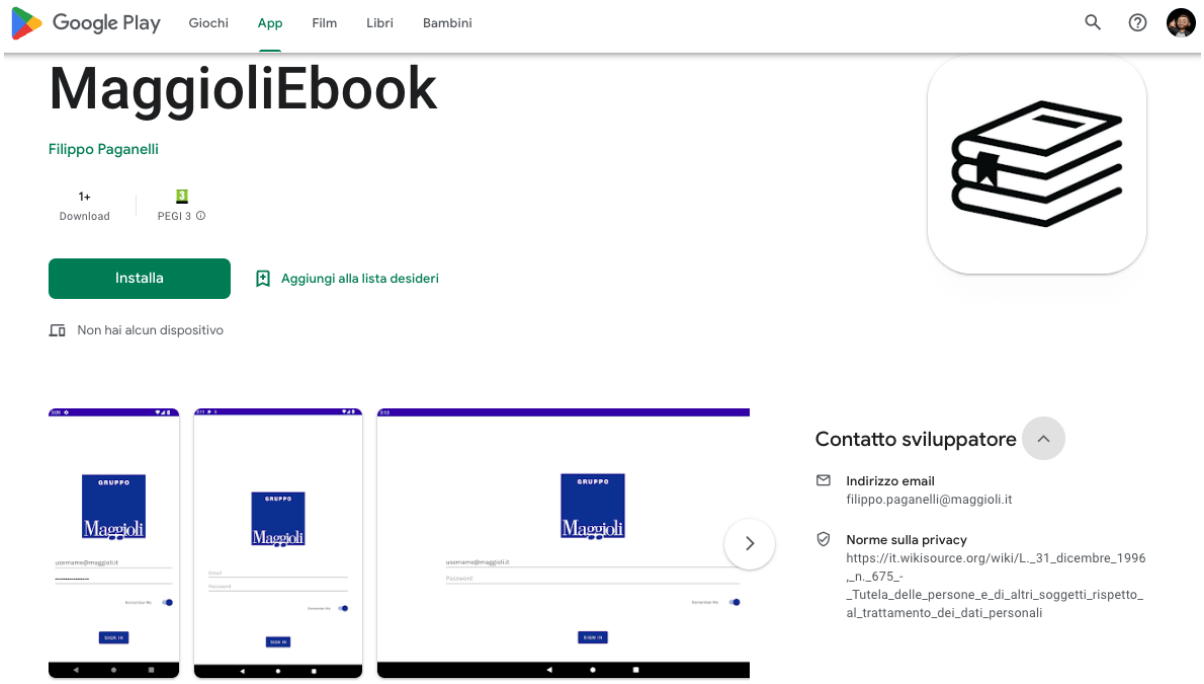

Figura 7.3: Schermata Google Play Store per la fase di stabilizzazione dell'applicazione Android

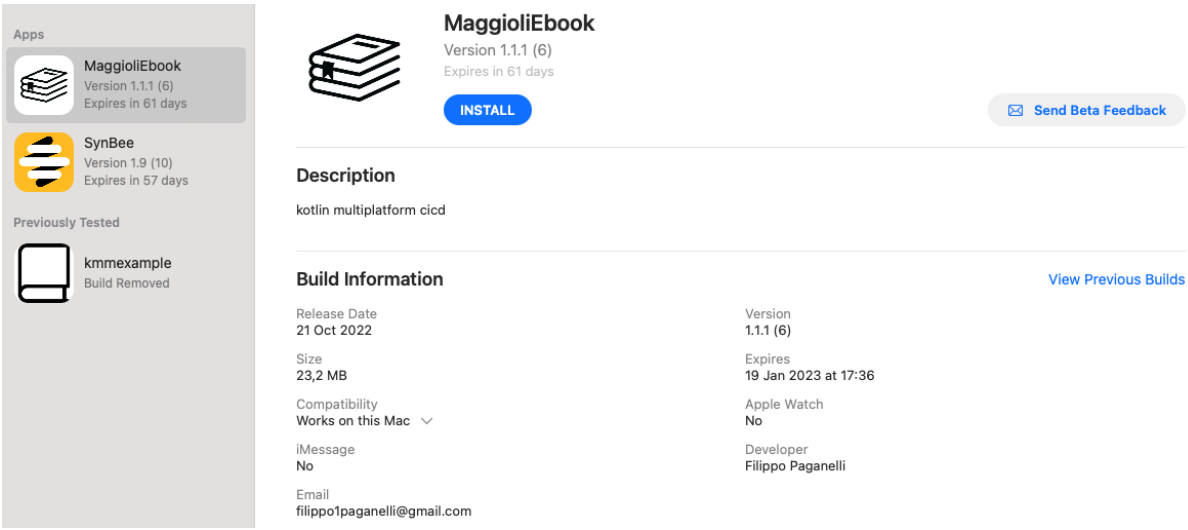

Figura 7.4: Schermata Testflight per la fase di stabilizzazione dell'applicazione iOS

Un ciclo di stabilizzazione *alpha-beta* risulta nell'esecuzione di due pipeline attivate rispettivamente con (*i*) la modifica sul branch *dev* del codice dell'applicazione e (*ii*) il merge del branch *dev* sul branch *test*. Le seguenti schermate, catturate dalla piattaforma

GitLab utilizzata come sistema di versionamento e automazione, mostrano l'esecuzione di un esempio di queste due pipeline nel caso della sola applicazione Android:

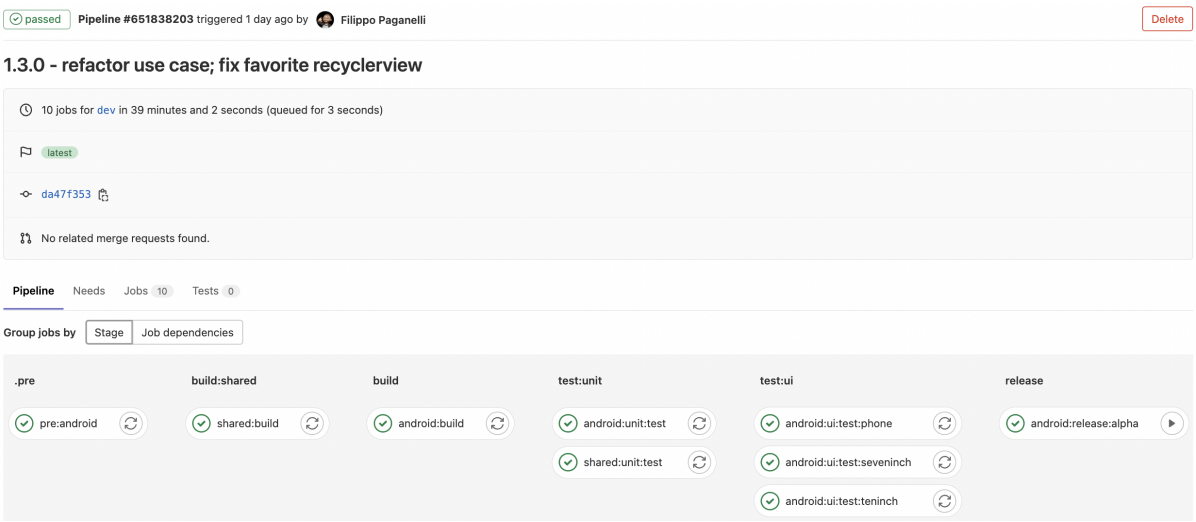

Figura 7.5: Schermata GitLab d'esecuzione della pipeline completa per il rilascio dell'applicazione Android in versione *alpha*

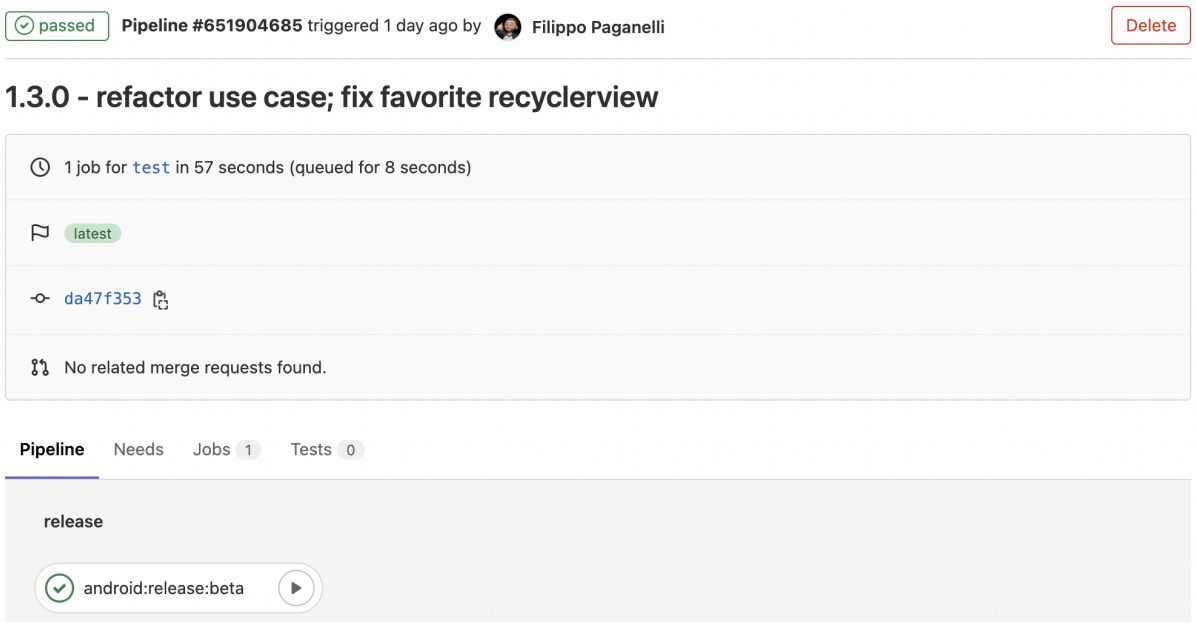

Figura 7.6: Schermata GitLab d'esecuzione della pipeline completa per la promozione da versione *alpha* a versione *beta* dell'applicazione Android

7.4 Analisi del codice

La fase di analisi del codice è stata realizzata con successo seguendo i vincoli aziendali e le pratiche di Continuous Inspection come descritto rispettivamente nei capitoli 4 e 5.

La seguente schermata catturata dalla piattaforma GitLab rappresenta l'esecuzione di una pipeline schedulata per l'analisi del codice terminata con successo:

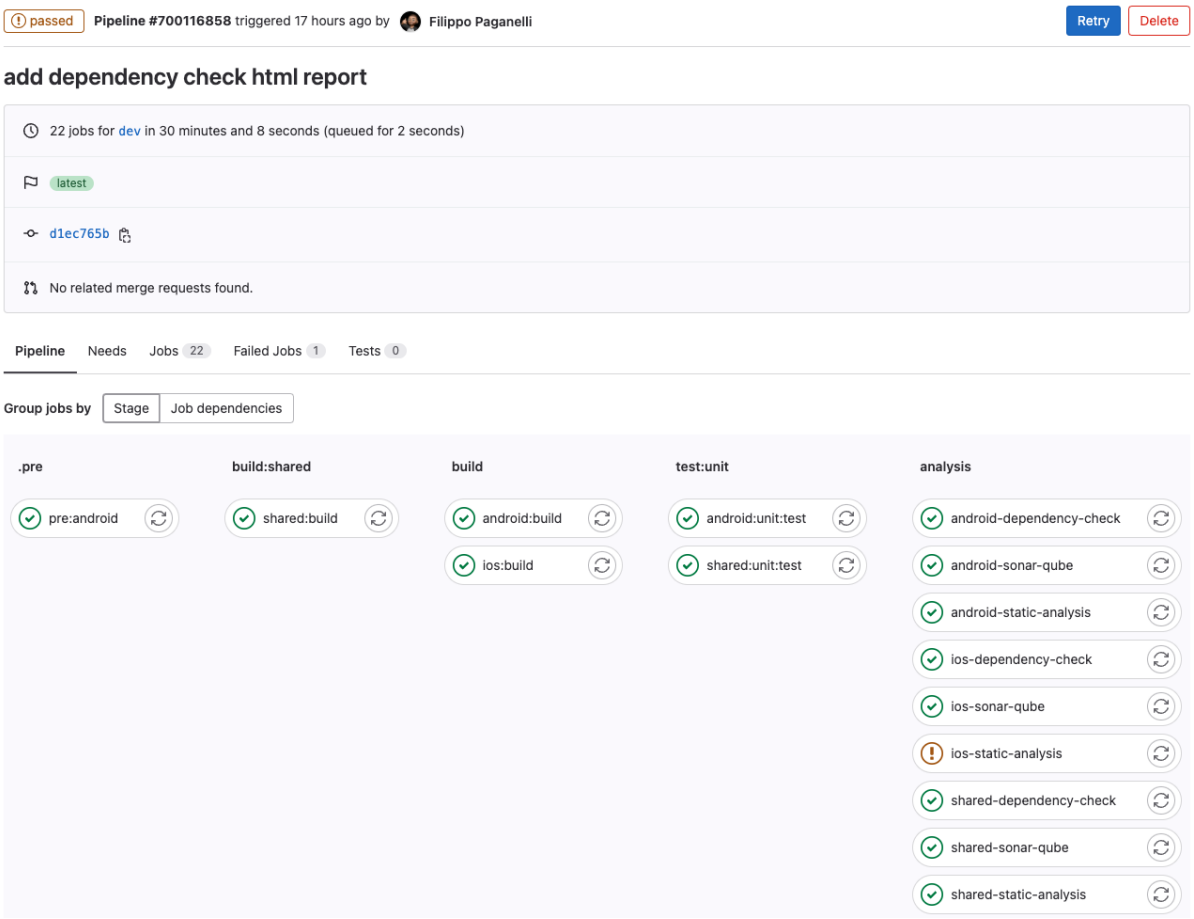

Figura 7.7: Schermata GitLab d'esecuzione della pipeline completa per l'analisi del codice dei tre moduli sviluppati

Nonostante il sistema d'automazione implementato per questa fase d'analisi funzioni correttamente come atteso, non è possibile dire lo stesso per i risultati ottenuti dalle scansioni dei vari tools utilizzati, come Dependency-Check, Detekt e SwiftLint, e soprattutto per la loro integrazione con il servizio aziendale SonarQube:

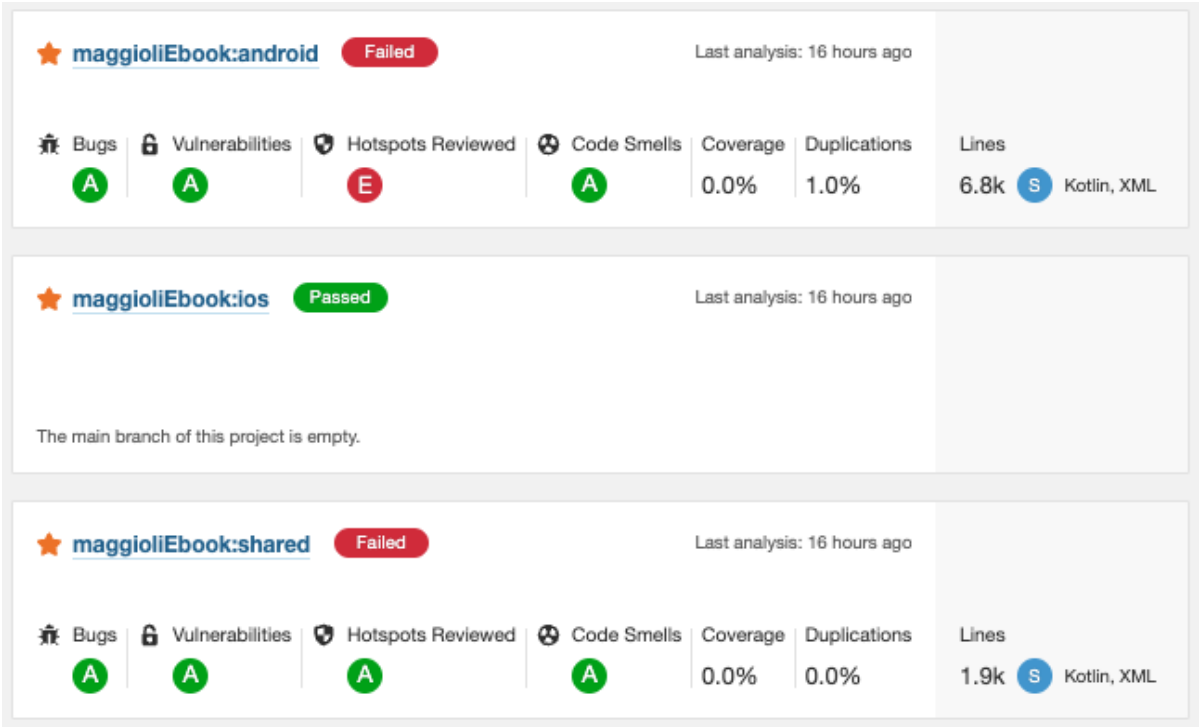

Figura 7.8: Schermata SonarQube aziendale contenente i tre moduli del caso di studio analizzati

Per esempio, nella seguente schermata rappresentante la panoramica del progetto iOS è indicata la presenza di 837 risultati classificati come *code smells*, dei quali 815 con severity *minor* e 22 con severity *major* ma nella schermata precedente, ovvero la Homepage del servizio SonarQube aziendale, non è indicato alcun risultato.

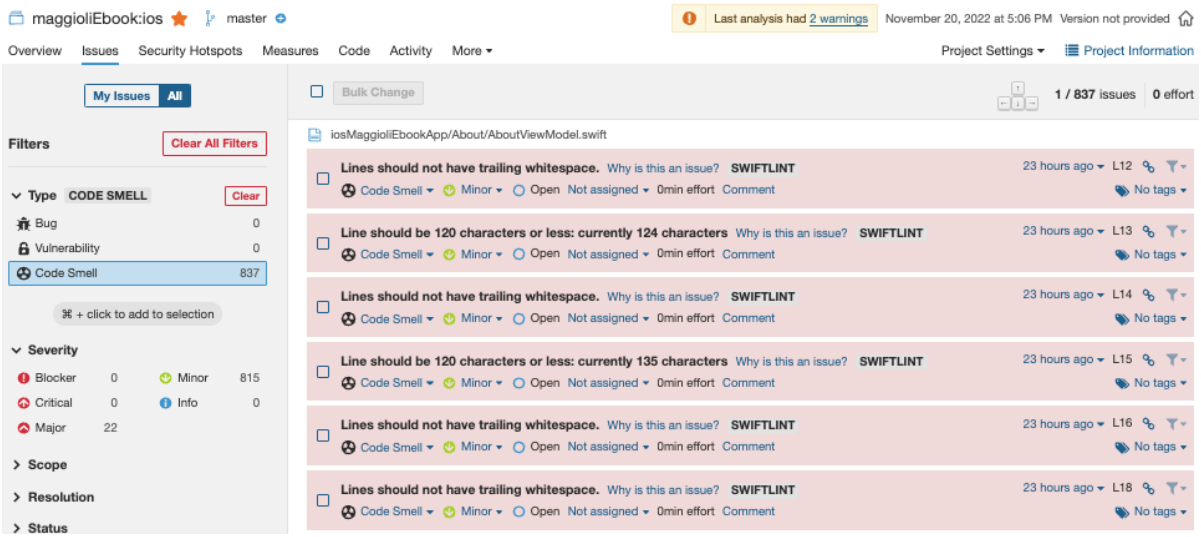

Figura 7.9: Schermata SonarQube aziendale in cui è mostrata la panoramica del progetto iOS e i risultati ottenuti dalle scansioni tramite il tool SwiftLint

7.5 Statistiche

Considerando il contesto in cui è stato realizzato il caso di studio industriale è molto complesso estrarre metriche e valori confrontabili: gli esistenti team di sviluppo in azienda che si occupano di applicazioni mobile al momento non adottano nessuna tecnologia moderna per lo sviluppo multi-platform o cross-platform e nessun sistema d'automazione per il processo.

L'adozione della cultura DevOps e l'applicazione delle pratiche abilitanti, come descritto nel capitolo 1, in ambito mobile utilizzando moderne tecniche di sviluppo multipiattaforma, come indicato nel capitolo 3, consiste in un progetto di forte innovazione realizzato nel contesto aziendale di Ricerca e Sviluppo. Tipicamente sia le risorse assegnate a questi progetti che la complessità dei domini modellati non sono paragonabili ai prodotti commercializzati dalle altre business unit aziendali ma sono tarati per lo sviluppo di progetti dimostrativi, chiamati anche *Proof of Concept*.

Le seguenti metriche e statistiche indicate, raccolte tramite l'ausilio della piattaforma GitLab utilizzata, forniscono dunque un'analisi del caso di studio definendo un punto di partenza per il confronto dei lavori futuri svolti nelle stesse aree tematiche e indicate nella sezione successiva.

Percentuale linguaggi utilizzati

La seguente metrica, fornita dal servizio GitLab *Repository Analytics*[1], definisce la composizione del repository contenente il sorgente dell'applicazione KMM MaggioliEbook (capitolo 6) in termini di percentuale di linguaggi di programmazione utilizzati: 57,38% Kotlin, 42,09% Swift e 0,54% Ruby.

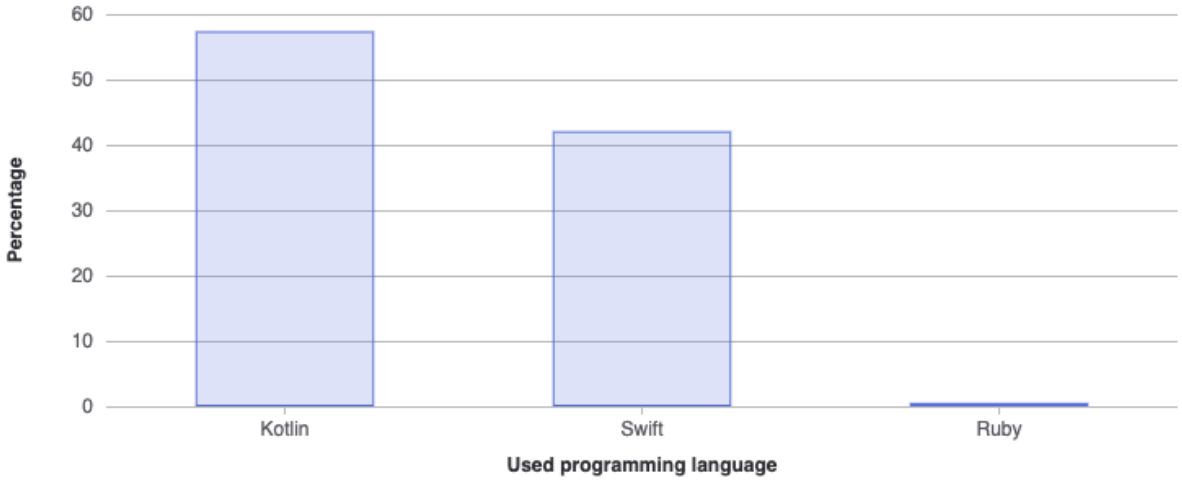

Figura 7.10: Composizione del repository dell'applicazione KMM MaggioliEbook in termini di percentuale di linguaggi di programmazione utilizzati

[1] https://docs.gitlab.com/ee/user/analytics/repository_analytics.html

Commit effettuate e distribuzione temporale

Anche le seguenti metriche sono fornite dallo stesso servizio GitLab *Repository Analytics* e definiscono la distribuzione temporale delle commit effettuate sul repository dell'applicazione KMM MaggioliEbook.

E' importante considerare che i valori indicati nei seguenti grafici si riferiscono ad un totale di 185 commit realizzate da un solo autore nel periodo compreso tra il 17 Luglio 2022 (data di creazione del repository) e il 20 Novembre 2022.

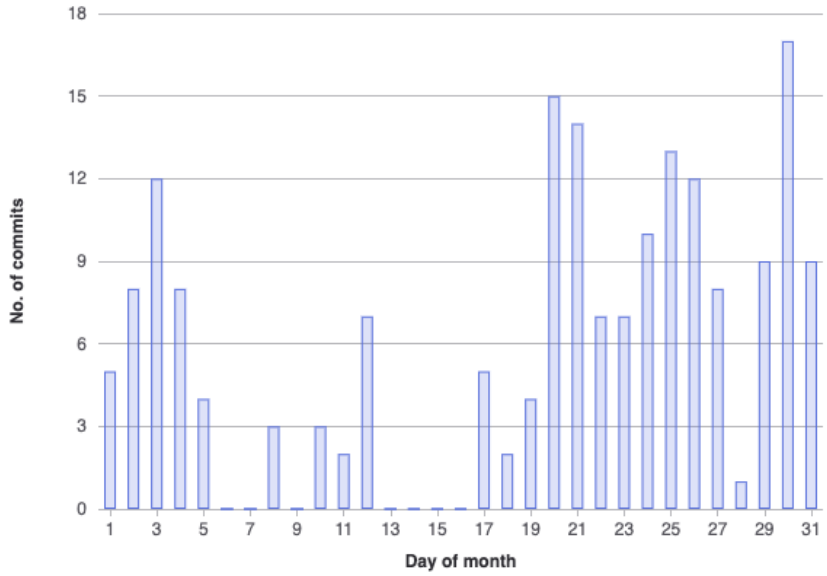

Figura 7.11: Distribuzione delle commit effettuate per giorno del mese

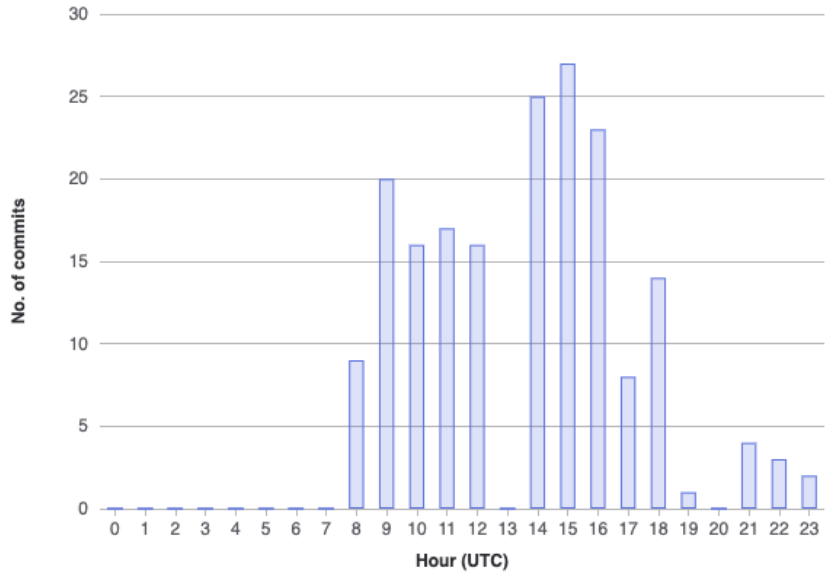

Figura 7.12: Distribuzione delle commit effettuate per ora del giorno

Tasso di successo delle pipeline

La seguente metrica e quelle successive sono estratte da GitLab tramite un altro servizio, chiamato *CI/CD Analytics*[2]. Il tasso di successo delle pipeline eseguite per l'applicazione KMM MaggiolieBook è del 45,71% ed è dato dal numero di pipeline terminate con successo (16) sul totale delle pipeline eseguite (35). Dal grafico è possibile notare che dopo un primo periodo di'ntegrazione e adattamento della pipeline progettata tramite template, come descritto nel capitolo 5, il tasso di successo è aumentato da circa il 16,67% a Settembre 2022 a circa 45,71% a Novembre 2022.

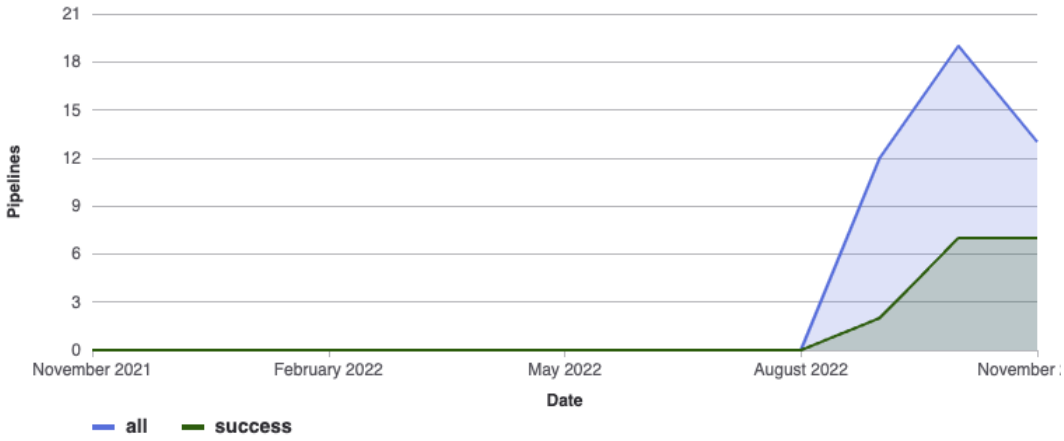

Figura 7.13: Tasso di successo delle pipeline (Luglio 2022 - Novembre 2022)

Durata delle pipeline per le ultime 30 commit effettuate

Un altro grafico interessante fornito dal servizio GitLab *CI/CD Analytics* è la durata delle pipeline eseguite per le ultime 30 commit per le quali era associata una regola di attivazione.

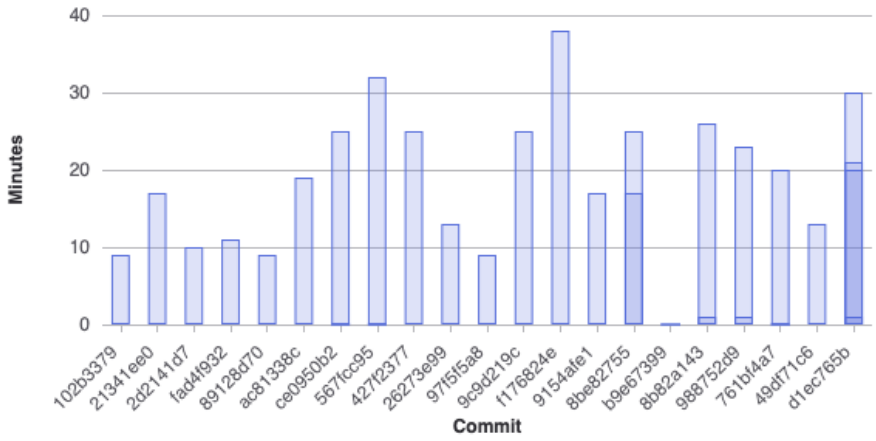

Figura 7.14: Durata delle pipeline per le ultime 30 commit effettuate sul repository dell'applicazione KMM MaggioliEbook

[2]https://docs.gitlab.com/ee/user/analytics/ci_cd_analytics.html

Durata media delle pipeline per fase di sviluppo e piattaforma coinvolta

Considerando che le pipeline possono coinvolgere una o entrambe le versioni dell'applicazione sviluppata in base alla modifica effettuata al codice, una metrica utile a future valutazioni e confronti è data dal tempo medio d'esecuzione delle pipeline in base alla piattaforma coinvolta, ovvero Android e/o iOS, e alla fase del processo: (*i*) rilascio versione *alpha*, (*ii*) rilascio versione beta e (*iii*) analisi statica del codice.

I seguenti valori sono stati calcolati utilizzando le tempistiche d'esecuzione delle pipeline rilevate dalla piattaforma GitLab utilizzata.

Piattaforma	**Rilascio Alpha**	**Rilascio Beta**	**Analisi**
Android	$\sim$ *20 min*	$\sim$ *1 min*	$\sim$ *15 min*
iOS	$\sim$ *23 min*	$\sim$ *30 sec*	$\sim$ *15 min*
Android e iOS	$\sim$ *28 min*	$\sim$ *1 min*	$\sim$ *20 min*

Tabella 7.1: Durata media delle pipeline per fase di sviluppo e piattaforma coinvolta

Per valutare queste metriche correttamente è necessario considerare gli stage e i job che compongono le varie fasi, come descritto nel capitolo 5. La fase "*Rilascio Alpha*" è la più lunga ed è composta infatti da tutti gli stage e i job che comprendono le pratiche di Continuous Integration e Continuous Delivery. La fase "*Rilascio Beta*" svolge invece il solo compito di promozione di una versione da *alpha* a *beta* e per questo è molto più breve rispetto la precedente. L'ultima fase per cui sono state definite le metriche è quella di analisi, la quale deve eseguire tutti i job di analisi statica del codice e delle dipendenze per ognuna delle piattaforme.

7.6 Lavori futuri

I risultati descritti nella sezione precedente sono decisamente positivi, soprattutto in termini di processo di sviluppo e di sistema d' automazione realizzato. L'intero sistema attualmente è basato su una infrastruttura ridotta, dedicata alla sola riuscita del caso di studio. Dalla re-ingegnerizzazione di quest'ultima al fine di renderla abbastanza robusta e scalabile per supportare realmente gli altri team di sviluppo in azienda che si occupano di applicazioni mobile è normale attendersi grandi miglioramenti in termini di prestazioni. A tal proposito, come già anticipato nel capitolo 5, è necessario valutare le possibili alternative al componente GitLab runner self-hosted adottato, come ad esempio soluzioni complete, virtual machine macOS e runner forniti as-a-Service.

Altri aspetti importanti riguardanti il processo di sviluppo e il sistema d'automazione trattati sicuramente nei lavori futuri sono il monitoraggio e la configurazione remota[3]. Il monitoraggio, come descritto nel capitolo 2, è la fase del processo di sviluppo che segue

[3]https://firebase.google.com/docs/remote-config

alla fase di distribuzione delle applicazioni e permette di monitorare sia il comportamento del dispositivo che quello dell'utente applicando tecniche e pratiche che rispettano la cultura DevOps (capitolo 1). La configurazione remota è un meccanismo che permette di ottimizzare l'intero processo di sviluppo poiché permette di modificare aspetti dell'applicazione in modo dinamico senza dover rilasciare nuove versioni: un tipico esempio di utilizzo è la configurazione dell'url di connessione al database (*JDBC*[4]), effettuata ogni volta che viene migrato il database evitando valori cablati nel codice dell'applicazione.

Nonostante i buoni risultati ottenuti a livello generale di lavoro svolto è stata evidenziata nella sezione precedente la carenza nei report restituiti dalle fasi di analisi statica e nella loro integrazione con il Vulnerability Management System aziendale. Questo rappresenta un requisito importante per l'azienda e che per essere soddisfatto necessita di ulteriori approfondimenti.

Per quando riguarda invece lo sviluppo dell'applicazione mobile multipiattaforma, i risultati raggiunti tramite il framework Kotlin Multiplatform Mobile sono stati soddisfacenti ma hanno dimostrato, soprattutto nel capitolo 6, che la condivisione di logica tra gli ecosistemi Android e iOS richiede ancora qualche miglioramento. Di fatto il framework KMM e tutti i componenti che utilizza, come il compilatore Kotlin/Native e i plugin Gradle, sono tutti in fasi *pre-stable*.

Al fine di completare lo studio e la sperimentazione sulle moderne tecniche di sviluppo di applicazioni mobile e sulle pratiche DevOps ad esse applicabili è necessario sviluppare lo stesso caso di studio industriale tramite l'utilizzo dei framework cross-platform, come ad esempio quelli indicati nel capitolo 3 e la realizzazione di altri template per il riuso della pipeline nel caso di sviluppo con questi framework.

[4]Java DataBase Connectivity

Capitolo 8

Conclusioni

L'output finale di questa tesi dimostra che è stato possibile adottare le pratiche e gli strumenti abilitanti la cultura DevOps per il processo di sviluppo di applicazioni mobile multipiattaforma. Il sistema d'automazione realizzato è stato in grado infatti di compilare, testare, impacchettare e rilasciare un'applicazione, sia in versione Android che in versione iOS, mediamente in circa 28 minuti.

Per poter raggiungere questo risultato decisamente positivo è stata necessaria una prima fase di analisi approfondita sulla cultura DevOps, sul ciclo di sviluppo delle applicazioni mobile e sui framework multipiattaforma. Successivamente è stato individuato un caso di studio industriale per i quali sono stati raccolti i requisiti e i vincoli aziendali sia per il processo di sviluppo che per l'applicazione da realizzare. In questa fase sono stati definiti gli obiettivi reali del caso di studio adeguando le necessità dell'azienda alle pratiche e tecniche descritte nella fase iniziale di studio. L'infrastruttura aziendale a supporto dei già esistenti sistemi di automazione adottati in altri ambiti di sviluppo software ha fornito un solido punto di partenza per la realizzazione del caso di studio. In questa infrastruttura sono stati integrati i componenti in grado di soddisfare tutte le esigenze e i vincoli delle applicazioni mobile al fine di automatizzare il processo di sviluppo in ambiente macOS. Da questo punto di vista, la soluzione realizzata ha dimostrato la possibilità di realizzare un sistema di automazione in ambiente macOS, ma ha evidenziato anche che la complessità della configurazione e della gestione di questi componenti rendono il modello as-a-Service preferibile, nonostante i costi elevati rispetto a quello self-hosted.

Il paradigma multipiattaforma scelto per lo sviluppo dell'applicazione mobile ha contribuito al raggiungimento del risultato finale grazie alla condivisione della logica applicativa fra le diverse piattaforme. Il bisogno di dover scrivere codice extra per poter riutilizzare correttamente il modulo condiviso nella applicazione iOS ha comunque evidenziato alcuni dei limiti di gioventù del framework utilizzato. La vera sfida affrontata nell'automatizzare il processo di sviluppo di applicazioni mobile è stata la gestione e la configurazione dei numerosi strumenti utilizzati nelle varie fasi del processo. Le due piattaforme infatti, richiedono strumenti diversi adibiti alle medesime funzioni, fattore che aumenta notevolmente la complessità del sistema di automazione. Come nel caso dei

framework multipiattaforma sono nati anche dei tool, come quello utilizzato in questa tesi, con lo scopo di diminuire la complessità del processo di sviluppo da un unico punto che funge da involucro per tutti gli strumenti richiesti dalle diverse piattaforme.

Bibliografia

[1] Ian Darwin. *Android Cookbook*. O'Reilly Media, 2011.

[2] Jennifer Davis and Ryn Daniels. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*. O'Reilly Media, 2016.

[3] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[4] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] Rob Kerr and Morstøl Kåre. *Beginning Swift*. Packt Publishing, 2018.

[7] Mikael Krief. *Learning DevOps: The complete guide to accelerate collaboration with Jenkins, Kubernetes, Terraform and Azure DevOps*. Packt Publishing, 2019.

[8] P.O. Laurence, A. Hinchman-Dominguez, G.B. Meike, and M. Dunn. *Programming Android with Kotlin: Achieving Structured Concurrency with Coroutines*. O'Reilly Media, 2021.

[9] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[10] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2017.

[11] Zigurd Mednieks, Laird Dornin, G.B. Meike, and Masumi Nakamura. *Programming Android*. O'Reilly Media, 2011.

[12] Shoba K. N. *Vocabulary 2.0: Smart Words of the 21st Century*. Notion Press, 2014.

[13] Robert Nagy. *Simplifying Application Development with Kotlin Multiplatform Mobile: Write Robust Native Applications for IOS and Android Efficiently*. Packt Publishing, 2022.

[14] Eric Ries. *The Lean Startup : How Constant Innovation Creates Radically Successful Businesses*. Portfolio Penguin, 2011.

[15] James Shore and Shane Warden. *The Art of Agile Development*. O'Reilly Media, 2008.