ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
FACOLTA' DI SCIENZE E TECNOLOGIE INFORMATICHE

**Corso di Laurea in**
SCIENZE E TECNOLOGIE INFORMATICHE

# SVILUPPO MOBILE MULTI-PIATTAFORMA: FATTORI DA CONSIDERARE NELLA SCELTA TECNOLOGICA

**Tesi di Laurea Triennale in**
MOBILE WEB DESIGN

RELATORE:
Dott. Mirko Ravaioli

PRESENTATA DA:
Mattia Pagini

SESSIONE Dicembre
ANNO ACCADEMICO 2018/2019

# Appendice

# 1. INTRODUZIONE

In molte delle aziende che si occupano di sviluppo di applicazioni mobile (quindi per applicazioni che devono funzionare su sistema operativo iOS e Android) continua ad aumentare l'incertezza su che tecnologia utilizzare ogni giorno per portare a termine nel migliore dei modi le richieste dei clienti, massimizzando i profitti.
La questione non è per nulla facile e anche se in molti nell'ambiente informatico continuano a dire che bisogna utilizzare "la giusta tecnologia per il giusto progetto" la sensazione è che questa frase sia sempre più utopistica con il passare del tempo, la realtà ci dice che al giorno d'oggi è impossibile conoscere tutto il necessario per fare sempre la scelta giusta e ci costringe ad arrivare a dei compromessi.

Fare la scelta giusta non è semplice, ci sono tante opzioni e spesso solo team di grandi aziende hanno le risorse e il tempo da investire per creare prototipi con diverse tecnologie e scegliere la migliore per il progetto da sviluppare.

L'obbiettivo di questa tesi è rispondere alle seguenti domande:

1. Quando conviene sviluppare un'app in maniera nativa?
2. Quando conviene sviluppare un'app multi-piattaforma?
3. Se si intuisce che è possibile sviluppare un'app multi-piattaforma su che base va scelto il framework da usare?

Per fare ciò si effettuerà un'analisi di tutto l'ecosistema collegato allo sviluppo mobile, compresi linguaggi di programmazione, studio di alcune metriche e sondaggi consultabili online di portali web specializzati nella raccolta di queste informazioni.

Si analizzeranno i risultati ottenuti dal sondaggio rilasciato a sviluppatori mobile multi-piattaforma, che daranno una mano nell'identificare eventuali problemi sul framework che usano maggiormente evidenziando quello che non è possibile valutare nella costruzione di una semplice demo.

Poiché non è corretto valutare dei framework includendo solo pareri esterni, verrà creata un'app TODO di demo sviluppata usando tutti i framework che ci permetterà di valutare il guadagno in termini di tempo di sviluppo e visualizzare eventuali problematiche che si possono incontrare.
Inoltre verrà approfondita la storia che ha portato alla nascita di questi framework multi-piattaforma e del loro funzionamento dietro le quinte.

Le considerazioni personali fatte nella tesi sono frutto di 5 anni all'interno dell'ambiente informatico come sviluppatore mobile Android nativo.

# 2. DISTRIBUZIONE TECNOLOGICA

Nel caso decidiamo di valutare altre tecnologie, dobbiamo aggiungere all'equazione anche un fattore puramente contestuale che dipende nella maggior parte dei casi dalla situazione lavorativo che ci circonda.
Delle volte per portare a termine un progetto dobbiamo affidarci a persone esterne che spesso vengono contrattualizzate solamente per il periodo necessario a terminarlo quindi è necessario tenere in considerazione il fattore "ricerca" di nuovo personale, perché le scelte tecnologiche che facciamo non dovrebbero essere prese senza prima valutare cosa il mercato offre.

Nel nostro caso specifico, dove stiamo valutando lo sviluppo mobile, possiamo ridurre le conoscenze necessarie ad alcuni linguaggi di programmazione così distribuiti:

1) Java o Kotlin per lo sviluppo Android
2) Objective-C o Swift per lo sviluppo iOS
3) Dart per lo sviluppo con Flutter
4) Javascript o Typescript per lo sviluppo con React Native
5) C# per lo sviluppo con Xamarin Forms

Per aiutarci a capire la diffusione di questi linguaggi possiamo usare qualche statistica, tra le tante presenti ne ho scelte due che a mio avviso sono tra le più usate e citate nelle varie comparazioni e articoli che si trovano in rete.
Queste sono l'indice **TIOBE**[1] e la statistica annuale di **Stack Overflow**[2].

**NOTE**: Prima di procedere analizzando le statistiche, è doveroso fare un accenno sui linguaggi C e C++ che nonostante possano essere usati senza problemi nello sviluppo mobile essi hanno solitamente scopi differenti.
Le librerie scritte in C e C++ sono utilizzate soprattutto quando si vogliono sviluppare logiche indipendenti dalla piattaforma e/o ad alte prestazioni riusabili in qualsiasi contesto, ecco perché linguaggi così vecchi hanno addirittura subito un aumento della loro diffusione negli ultimi anni, dovuta alla più eterogenea richiesta di implementazione su sistemi estremamente diversi tra loro, come IoT, sistemi integrati per auto, TV, desktop e mobile.
Detto questo possiamo escluderli dall'analisi di diffusione dei linguaggi di programmazione poiché nonostante siano linguaggi che possono essere usati per lo sviluppo mobile, essi possono essere collegati in vari modi a tutti gli stack tecnologici citati sopra senza poter influenzare la nostra classifica.

## 2.1 INDICE TIOBE

Partiamo dall'indice TIOBE, la cui composizione esatta è visibile al seguente link
https://www.tiobe.com/tiobe-index/programming-languages-definition/
in sintesi potremmo dire che è composto dalle ricerche che contengono le parole "*<language>* programming" effettuate dagli utenti nei motori di ricerca come Google

[1] https://www.tiobe.com/tiobe-index/
[2] https://insights.stackoverflow.com/survey/

(dove in <*language*> dobbiamo sostituire il nome di un qualsiasi linguaggio di programmazione), in sostanza rappresenta il livello di interesse che hanno programmatori e non per un determinato linguaggio ed è utile anche come proiezione per il futuro.

La statistica TIOBE Ottobre 2018 indica inoltre la variazione di interesse in percentuale confrontata con i dati di Ottobre 2017.
Abbiamo riportato la tabella presente sul sito in figura 1 e abbiamo evidenziato in giallo i linguaggi di programmazione che sono interesse della nostra analisi.

| Oct 2018 | Oct 2017 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Java | 17.801% | +5.37% |
| 2 | 2 | | C | 15.376% | +7.00% |
| 3 | 3 | | C++ | 7.593% | +2.59% |
| 4 | 5 | ^ | Python | 7.156% | +3.35% |
| 5 | 8 | ^ | Visual Basic .NET | 5.884% | +3.15% |
| 6 | 4 | v | C# | 3.485% | -0.37% |
| 7 | 7 | | PHP | 2.794% | +0.00% |
| 8 | 6 | v | JavaScript | 2.280% | -0.73% |
| 9 | - | ^^ | SQL | 2.038% | +2.04% |
| 10 | 16 | ^^ | Swift | 1.500% | -0.17% |
| 11 | 13 | ^ | MATLAB | 1.317% | -0.56% |
| 12 | 20 | ^^ | Go | 1.253% | -0.10% |
| 13 | 9 | vv | Assembly language | 1.245% | -1.13% |
| 14 | 15 | ^ | R | 1.214% | -0.47% |
| 15 | 17 | ^ | Objective-C | 1.202% | -0.31% |
| 16 | 12 | vv | Perl | 1.168% | -0.80% |
| 17 | 11 | vv | Delphi/Object Pascal | 1.154% | -1.03% |
| 18 | 10 | vv | Ruby | 1.108% | -1.22% |
| 19 | 19 | | PL/SQL | 0.779% | -0.63% |
| 20 | 18 | v | Visual Basic | 0.652% | -0.77% |

*Figura 1 – Indice TIOBE Ottobre 2018*

Come possiamo notare, Java ricopre ancora un ruolo preponderante negli interessati alla programmazione, a seguire troviamo C# che è ancora in ottima posizione, poi Javascript, Swift e Objective-C che è ormai nella parabola discendente di vita visto che Apple sta cercando di spingere tutti i programmatori iOS ad adottare Swift (ma ciò richiederà ancora molto tempo vista la quantità di librerie scritte in passato in Objective-C).

Quindi da questa statistica sembra che lo sviluppo nativo Android sia al sicuro come quello iOS visto che entrambi i linguaggi principali sono ben piazzati; anche C# e Javascript sono ben posizionati quindi l'interesse che hanno i programmatori per questi linguaggi potrebbero poi portarli a sviluppare applicazioni mobile rispettivamente con Xamarin Forms

e React Native.
Non troviamo in classifica nulla riguardo a Dart, il linguaggio di programmazione di Flutter.

## 2.2 DISTRIBUZIONE BADGE STACKOVERFLOW

Parlando di sviluppo, non è possibile non considerare le immense statistiche possedute da StackOverflow, il portale più usato da praticamente tutti gli sviluppatori del mondo.
Una parte interessante di questo portale è che tantissime metriche sono pubbliche, ed è quindi possibile procedere facendo addirittura delle query su alcune tabelle del loro database[3] per ottenere dati importanti su alcuni parametri che riguardano l'ambiente stesso.
Su StackOverflow, quando un utente vuole pubblicare una domanda o un problema, è obbligato ad inserire almeno un "tag", ovvero delle etichette che servono al motore interno di StackOverflow per catalogare le domande e indirizzarle agli sviluppatori più affini a quell'ambiente usando i filtri disponibili. Quindi una persona che ha un problema con lo sviluppo su React Native aggiungerà il tag "react-native", una con un problema su iOS nativo, aggiungerà "ios" e così via.
Quando un utente risponde ad una di queste domande, oltre al punteggio assegnato ad ogni risposta, c'è un altro contatore che aumenta ogni volta che rispondiamo ad una domanda con lo stesso tag e dopo un certo punto questo punteggio si trasforma in badge, che può essere di color bronzo, argento e oro.
Se andiamo a leggere quante persone hanno ottenuto il badge argento, quindi coloro che hanno ottenuto 400 punti su 80 domande per i tag che stiamo considerando, possiamo costruire la tabella sottostante aggiornata a novembre 2018

| Tag | Numero di persone con badge argento |
|---|---|
| **android** | > 1000 |
| **ios** | 570 |
| **xamarin.forms** | 6 |
| **react-native** | 7 |
| **flutter** | 4 |

Questi dati ci fanno capire che ci sono molti più esperti e molte più persone attive nel risolvere problemi a chi sviluppa nativamente di quanto succeda per gli altri. Questi numeri vanno comunque presi con cautela, poiché qualcuno può fare una domanda su Flutter che riguardi parzialmente anche Android, quindi usare entrambi i tag facendo assegnare punti a tutti e due in caso di risoluzione; tuttavia rimane un'ottima metrica per capire le proporzioni del fenomeno mobile multi-piattaforma a livello mondiale. Pochi badge implicano poche domande su questi framework che a loro volta implicano un utilizzo moderato da parte delle aziende (fino ad ora almeno). I più maligni potrebbero pensare che poche domande stiano a significare pochi problemi perché il framework funzionano perfettamente o perché tutti i

[3] https://data.stackexchange.com/stackoverflow/query/new

problemi possono essere risolti tramite la documentazione ufficiale, questa visione è tuttavia difficilmente condivisibile.

## 2.3 SONDAGGIO STACKOVERFLOW 2018

Il sondaggio annuale di Stack Overflow è compilato a mano dagli iscritti alla piattaforma che vogliono contribuire alla raccolta dei dati, quindi in un certo senso è forse la statistica con più valore presente in rete data la quantità di sviluppatori che la usa. Per quello che ci interessa, andremo a riportare i dati relativi ai linguaggi di programmazione più utilizzati nella sezione tecnologia del sondaggio del 2018 indicato al seguente link: https://insights.stackoverflow.com/survey/2018/#technology-programming-scripting-and-markup-languages

Vista la lunghezza del sondaggio riportiamo qui sotto l'estratto dei risultati con un grafico in cui sono stati aggiunti solamente i linguaggi di programmazione di nostro interesse.
Il campione degli intervistati è stato di circa 100.000 persone, 39.000 delle quali europee e andando a filtrare i dati delle risposte (che essendo pubbliche è possibile scaricare sotto forma di CSV) ne sono state individuate 1535 appartenenti a persone che si sono identificate come sviluppatori residenti in Italia, quindi c'è abbastanza rilevanza anche per quanto riguarda il nostro territorio.

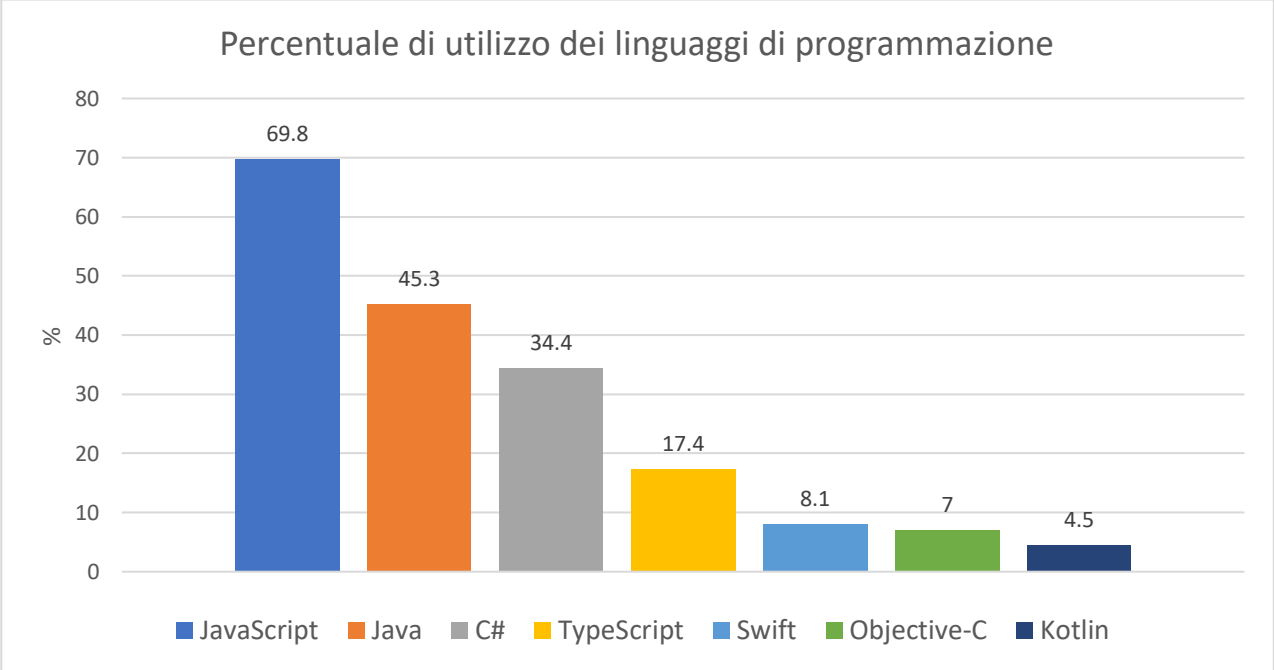



*Figura 2 – Grafico estratto dai risultati del sondaggio Stack Overflow 2018*

Il sondaggio chiedeva di inserire tutti i linguaggi di programmazione, scripting e markup con cui si aveva lavorato in maniera consistente nell'ultimo anno, la percentuale quindi rappresenta quante persone hanno risposto affermativamente all'uso di quella tecnologia rispetto al totale delle risposte.
Possiamo quindi interpretare il 69,8% raggiunto dal linguaggio Javascript come il fatto che più di 2/3 degli sviluppatori sia coinvolto in sviluppi web o in sviluppi mobile attraverso tecnologie web, inoltre a questo risultato va aggiunto il 17,4% di TypeScript che è uno dei modi moderni di programmare in Javascript, il tutto a favore del framework React Native.

Seguono Java e C# che sono comunque dei capisaldi dell'informatica da ormai tantissimi anni che non fanno che confermare quanto visto prima con la statistica TIOBE, aggiungendo quindi valore alle eventuali scelte nell'uso di Android nativo e Xamarin Forms.
A chiudere ci sono Swift e Objective-C che godono comunque di una buona percentuale, possiamo notare come Swift stia già superando Objective-C in quanto a diffusione perché è chiaro a tutti che il linguaggio è molto più potente e semplice da imparare.

Una menzione speciale va fatta a Kotlin, che pure essendo un linguaggio molto nuovo ha già conquistato il favore di moltissimi sviluppatori Android, infatti alla domanda su quale fosse la tecnologia che amate di più, Kotlin ha conquistato la seconda posizione dietro Rust, come è possibile vedere in figura 3.

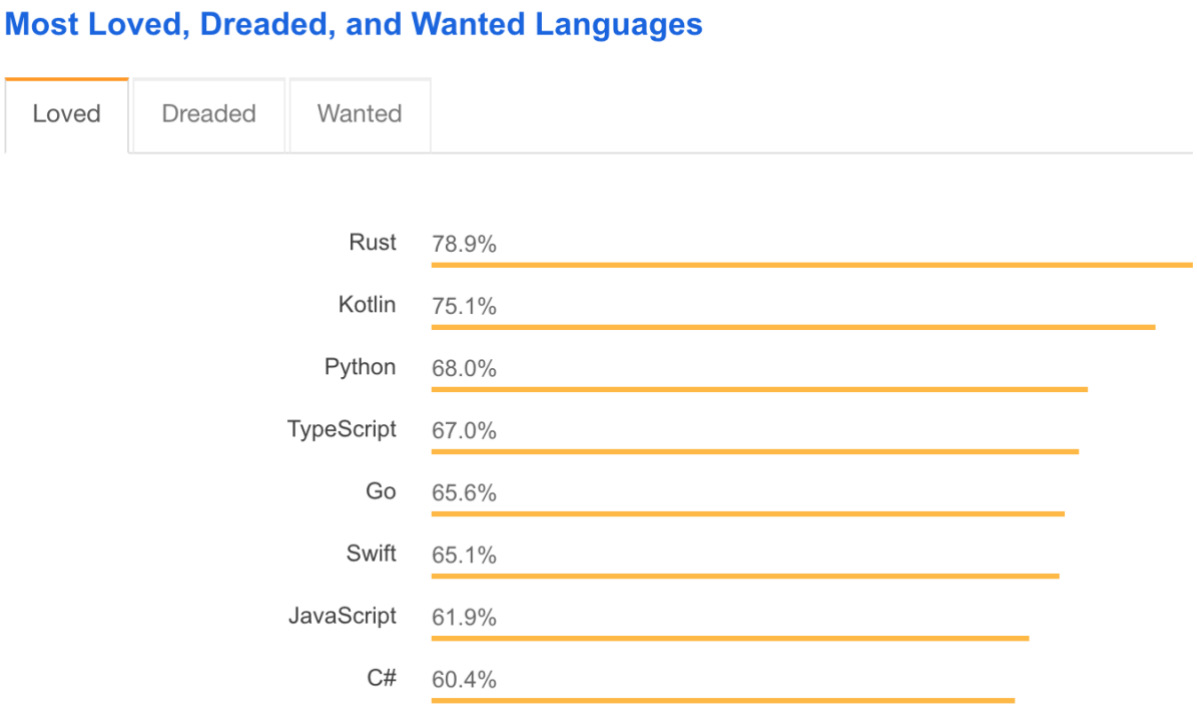



*Figura 3 - Prima parte della statistica sui linguaggi più amati di Stack Overflow 2018*

Neppure in questa classifica troviamo il linguaggio Dart, che a questo punto ci conferma come non sia effettivamente utilizzato in maniera consistente da nessuno che non sia qualche team interno a Google e poco altro, a tutto svantaggio della diffusione di Flutter.

## 2.4 STATISTICHE GITHUB

Un altro fattore di cui tenere conto è probabilmente la quantità di codice e librerie già esistenti per una determinata piattaforma, sfruttando le API di GitHub, un portale dove gli sviluppatori inseriscono il codice open source del loro lavoro, possiamo farci un'idea abbastanza chiara del livello di espansione dei vari framework.
GitHub è un portale molto conosciuto nel panorama degli sviluppatori, probabilmente il più visitato insieme a StackOverflow per trovare delle librerie da utilizzare negli sviluppi di tutti i giorni che qualcuno ha già scritto e quindi risparmiare tempo prezioso, queste librerie o progetti in generale, prendono il nome di "repository".

Per effettuare delle ricerche su questi repository, possiamo leggere la documentazione di GitHub[4] dove si trovano i riferimenti alle API v3, ovvero a delle chiamate REST che sono pubbliche con cui è possibile effettuare le ricerche di vario tipo sui dati pubblici di GitHub.

Nel nostro caso, per ricercare i repository possiamo iniziare partendo da questa chiamata

```
https://api.github.com/search/repositories?q={QUERY}
```

che permette di effettuare la ricerca di tutti i repository che nel nome contengono la chiave da sostituire a `{QUERY}`.
Poiché quello che ci interessa è di trovare solamente i repository collegati ai framework che stiamo valutando in questo documento, dobbiamo sfruttare un altro parametro disponibile per questa chiamata, ovvero il "topic".
I topic su GitHub sono l'equivalente di un "tag", ovvero sono parole che possono essere associate ad un repository per facilitarne l'individuazione da parte delle persone che cercano una determinata libreria per una specifica piattaforma/framework.

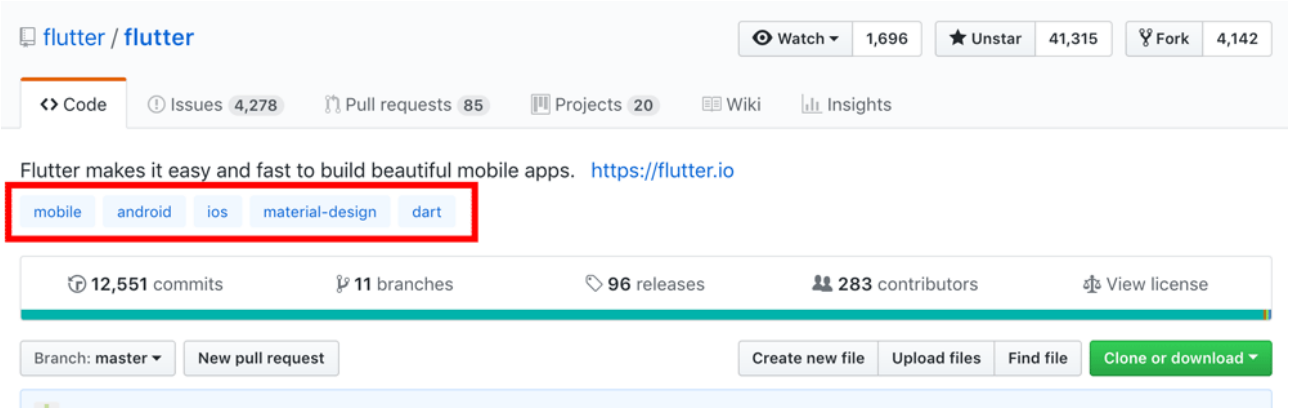



*Figura 4 - Topic nel repository di Flutter*

In figura 4 abbiamo un esempio dei topic usati dal repository GitHub di Flutter, evidenziati dal rettangolo rosso che è stato aggiunto per mostrare con precisione dove sono posizionati.

Chiarito l'utilizzo dei topic, il nostro link definitivo diventa il seguente:

```
https://api.github.com/search/repositories?q={QUERY}+topic:{TOPIC}
```

Per procedere quindi possiamo ignorare il parametro `{QUERY}` e al posto della chiave `{TOPIC}` inserire i vari nomi delle tecnologie come: "android" e "ios".

Per conoscere tutti i dettagli dei repository con contengono il topic "android", effettuiamo la seguente chiamata in GET

```
https://api.github.com/search/repositories?q=+topic:android
```

A questo punto abbiamo la risposta dal server che sarà un testo in formato JSON e di questo ci interessa solamente il campo "total_count" che contiene il totale che cerchiamo.

[4] https://developer.github.com/v3/search/#search-repositories

Ripetendo l'operazione anche per tutti gli altri topic otteniamo i risultati riportati nella sottostante tabella:

| Piattaforma | Topic | Numero repository |
|---|---|---|
| Android | android | 43404 |
| iOS | ios | 16230 |
| React Native | react-native | 10045 |
| Flutter | flutter | 1795 |
| Xamarin.Forms | xamarin-forms | 1196 |

Da questi risultati ricaviamo altre indicazioni utili ma prima dobbiamo fare alcune doverose premesse:

1. Il valore ottenuto comprende qualsiasi repository a cui sia stato assegnato il topic relativo, questo però non impedisce a nessuno di creare un repository vuoto o di altro tipo e aggiungergli il topic "android" senza che al suo interno ci sia veramente qualcosa di collegato ad Android.
2. Tantissimi repository su GitHub non contengono topic, persino quelli di alcune delle librerie più famose e utilizzate. Questo perché non è obbligatorio in fase di creazione di un repository e l'inserimento è lasciato a discrezione del suo creatore.

Detto questo, rimane comunque un'ottima approssimazione e aiuta a capire il rapporto di grandezza tra le varie tecnologie, i numeri reali sono sicuramente molto più alti di quelli evidenziati in questa tabella che rimane comunque ottima come tassello da aggiungere alla nostra analisi.

# 3. SONDAGGIO MULTI-PIATTAFORMA

Nella valutazione di una nuova tecnologia per sviluppo mobile multi-piattaforma è importante sapere anche cosa pensano gli sviluppatori che usano quelle tecnologie ogni giorno, poiché il successo di una applicazione passa anche dalla soddisfazione che il team di sviluppo ha nel portarla avanti.

Imporre un nuovo framework senza il consenso di gran parte del team è sempre un rischio, tantomeno se in giro si legge che la soddisfazione delle persone che lo usano non è delle migliori. Per rispondere a questa domanda, si è creato un sondaggio sfruttando un'applicazione web di Google chiamata Google Forms, che di base permette di creare sondaggi abbastanza complessi in maniera semplice e intuitiva, inoltre è possibile rendere univoche le risposte sfruttando l'account Gmail dell'utente che era quindi obbligatorio per procedere.

Nella creazione del sondaggio era necessario trovare il giusto compromesso tra informazioni da acquisire e tempo di completamento. Sapendo di doverli pubblicare su gruppi Facebook a cui siamo iscritti, non era possibile creare un sondaggio troppo lungo poiché avrebbe scoraggiato i più dal rispondere. Abbiamo optato quindi per un sondaggio con 2 domande generiche e 5 domande specializzate in base alla risposta di una delle due domande iniziali, per un totale di 7 domande.

I gruppi Facebook che hanno fatto parte dell'esperimento sono i seguenti:

- DevMarche: https://www.facebook.com/groups/developermarche/

- Programmatori Need for Nerd – Code Community https://www.facebook.com/groups/1703058016594724/

Il bacino potenziale di utenti (considerando che variano di giorno in giorno e che alcuni sono iscritti sia ad un gruppo che all'altro) era di circa 12.500 persone ma le risposte pervenute in circa 2 giorni che il sondaggio è rimasto compilabile sono state solo 24 quindi circa lo 0,19% degli utenti.

All'apparenza il numero risposte è sembrato molto basso ma dobbiamo prima tenere conto delle seguenti considerazioni valide in questi gruppi:

- La maggior parte delle persone è iscritta ma non è attiva e non legge le discussioni
- Molti sono studenti di università o superiori che quindi non lavorano ne hanno approfondito un particolare framework tra quelli richiesti
- C'è molto personale di reclutamento di aziende quindi non sviluppatori
- Lo sviluppo mobile multi-piattaforma è qualcosa abbastanza di nicchia, sono ancora in pochi che si affidano a queste soluzioni per immaginiamo che non siano moltissimi gli sviluppatori che conoscono a fondo queste tecnologie.

Fatte queste premesse il numero di risposte è ragionevole per un sondaggio distribuito in questo modo, ovvero ad una platea ampia ma non specializzata nelle tecnologie sopra citate senza nessuna forma di visibilità aggiuntiva.

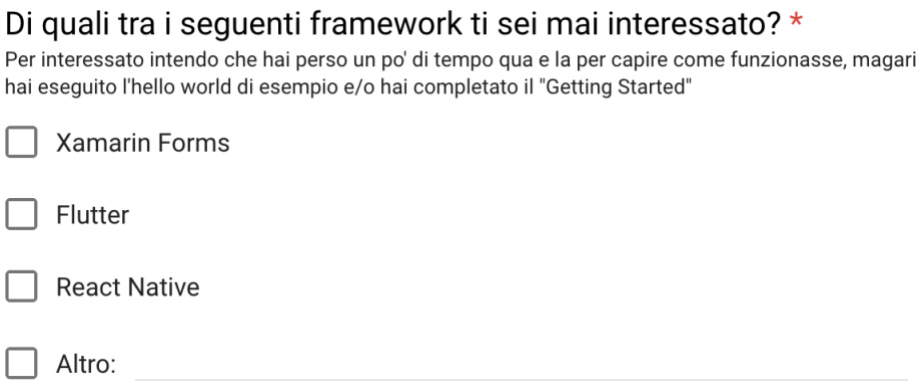
Volendo fare un paragone con il sondaggio del 2018 effettuato da StackOverflow che conta 9.7 milioni di utenti registrati (aggiornato a novembre), hanno iniziato il sondaggio 101.592 utenti e di questi solo 67.441 lo hanno completato[5], si parla quindi dello 0,69% nonostante sia stato possibile parteciparvi per 20 giorni e che nel sito in alto fosse presente un banner ad altissima visibilità per incoraggiare gli utenti a partecipare.

Detto questo andiamo ora ad analizzare le domande poste e ricordando che come di consueto per i sondaggi, l'ordine delle opzioni da scegliere per ogni domanda è stato reso casuale per evitare che i partecipanti fossero indotti a selezionare una risposta in base alla posizione.

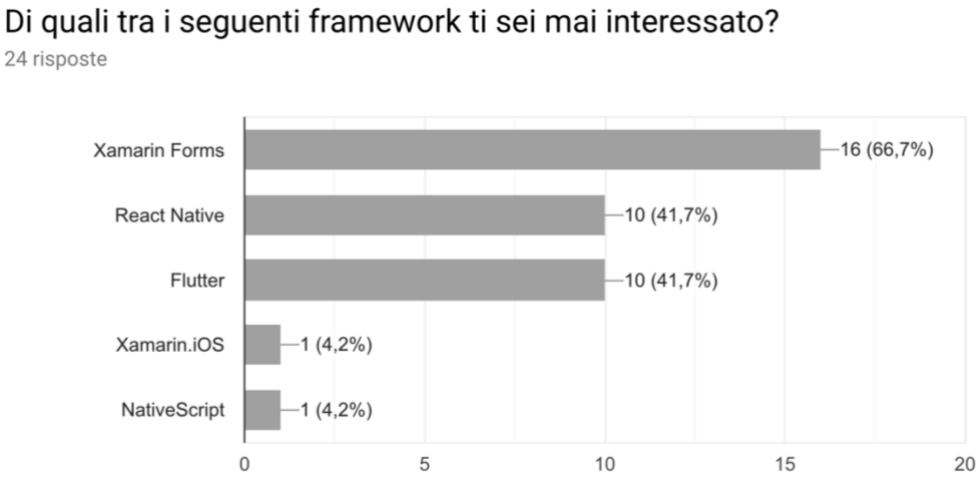
## 3.1 DOMANDA 1

[5] https://insights.stackoverflow.com/survey/2018/#methodology

Ogni utente è libero di selezionare un qualsiasi numero di opzioni tra quelle proposte e di aggiungerne delle altre con l'opzione Altro



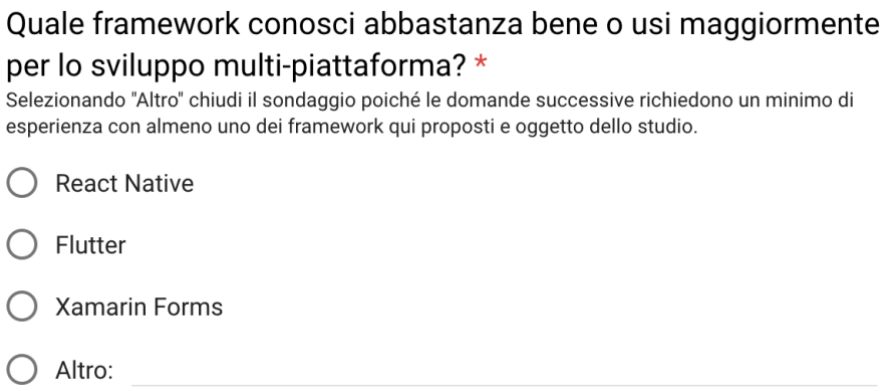

Risposta:



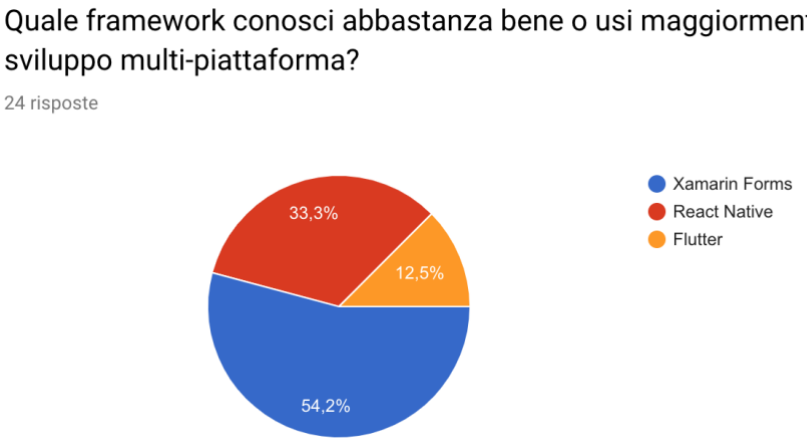

La domanda serve per capire la capacità che ha la società di sviluppo che sta dietro ad un particolare framework di far conoscere quello che stanno facendo e gli obbiettivi che stanno portando avanti.
Gli sviluppatori possono non usare direttamente quella tecnologia ma il fatto di non esserne mai venuti a conoscenza o di non essere nemmeno stati invogliati a scoprirne di più può essere indice di diversi fattori tra i quali un'errata strategia di comunicazione.
Al contrario, dalle risposte ottenute si può notare che tutte le aziende stanno facendo un ottimo lavoro poiché molti si sono interessati anche a tecnologie che non usano direttamente. Il fatto che Xamarin.Forms sia tra i più studiati è sicuramente dovuto al fatto che è stato il primo framework di sviluppo multi-piattaforma presentato e quindi ci aspettiamo anche che ad oggi sia uno tra i più utilizzati.
Possiamo ignorare la risposta relativa a Xamarin.iOS poiché è relativa ad uno specifico SDK di Xamarin per scrivere applicazioni iOS in C# (quindi non multi-piattaforma) mentre una persona ha accennato a NativeScript[6], che è stato escluso da questo sondaggio perché ancora poco utilizzato rispetto agli altri ma sicuramente da tenere in considerazione per il futuro.

## 3.2 DOMANDA 2

[6] https://www.nativescript.org/




Risposta:



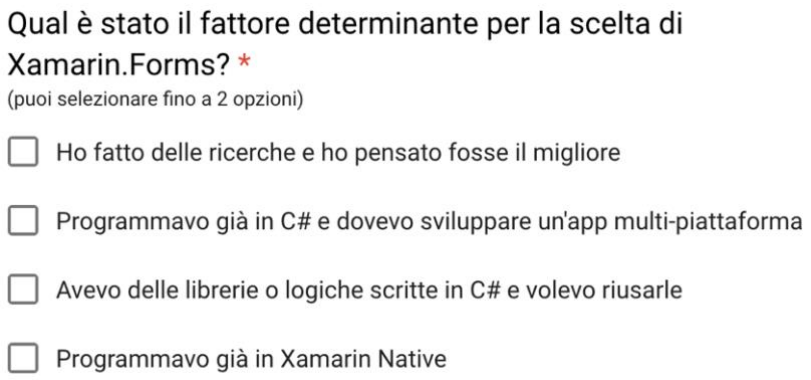

Questa domanda è la più importante del sondaggio, poiché ci mostra la percentuale d'uso dei vari framework oggetto di questa tesi. Xamarin.Forms è il framework più usato con 13 preferenze, poi React Native con 8 e chiude Flutter con 3. Nessuno dei partecipanti ha votato "Altro" nonostante fosse l'opzione di fuga per evitare che utenti che non conoscessero almeno una delle tre piattaforme procedessero nel questionario.
Questi valori confermano l'interesse per Xamarin.Forms e React Native della prima domanda ma non lo fanno per Flutter, questo significa che molte persone si stanno interessando a Flutter ma ancora in pochissimi hanno avuto il coraggio o l'opportunità di usarlo in maniera costante a lavoro o per i loro progetti personali.

La risposta a questa domanda creava un bivio nel sondaggio, ovvero chi ha selezionato Xamarin.Forms si è ritrovato in un sondaggio con delle voci inserite appositamente per Xamarin.Forms e così anche per React Native e Flutter. La decisione di procedere in questo modo è dovuta al fatto che altrimenti non saremmo riusciti a fornire le domande corrette non potendole specializzare in base alla risposta data.

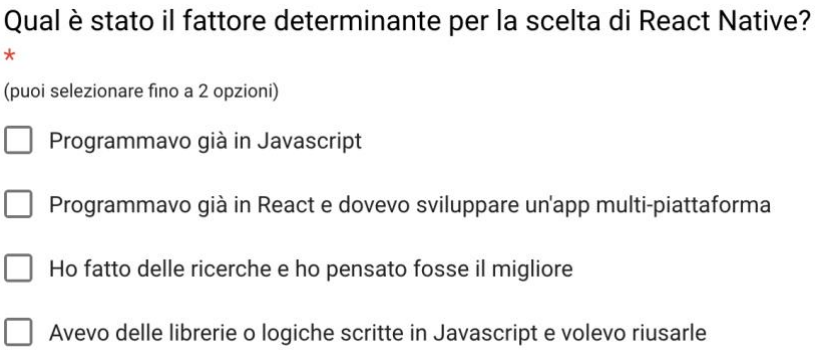
Andiamo quindi ad analizzare la seconda parte del sondaggio, raccogliendo le domande e le risposte per ogni Framework in modo da valutarle nell'insieme dove possibile.

## 3.3 DOMANDA 3

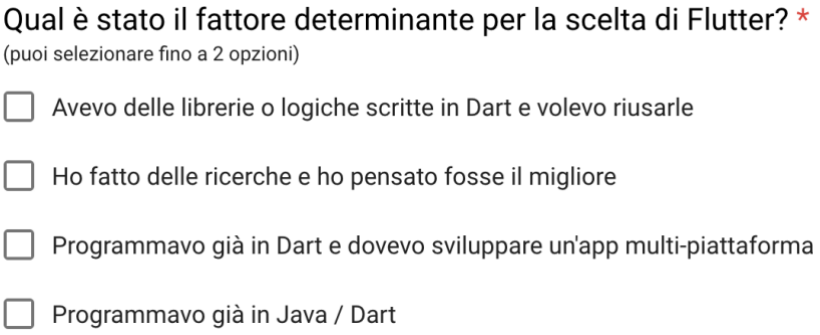
- Xamarin Forms




- React Native

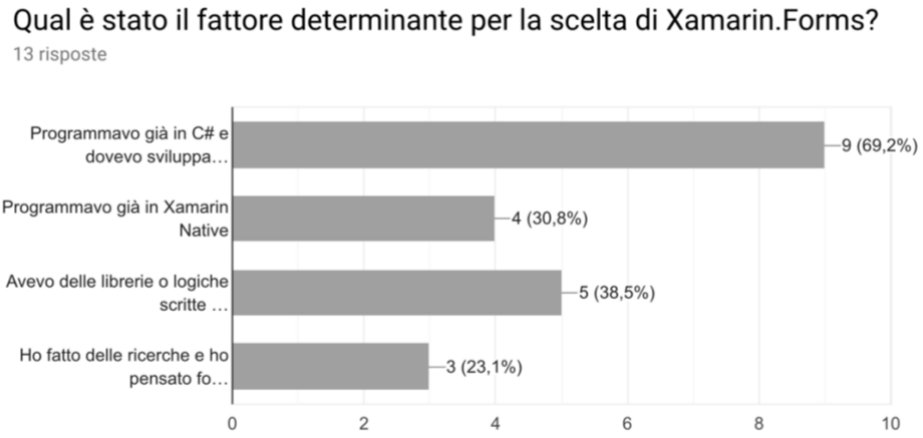



- Flutter

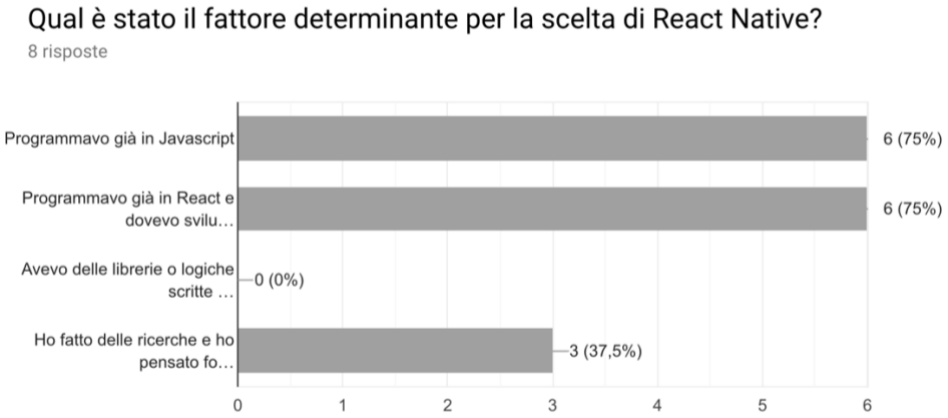



Risposta:

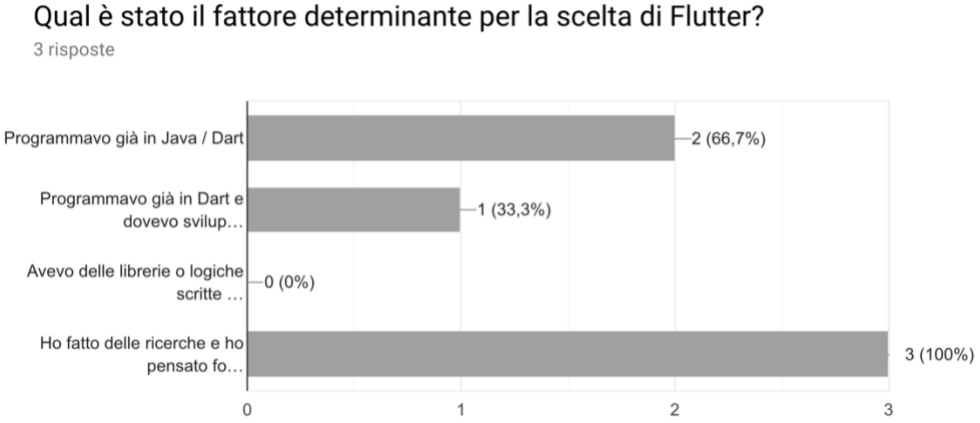
- Xamarin.Forms




- React Native



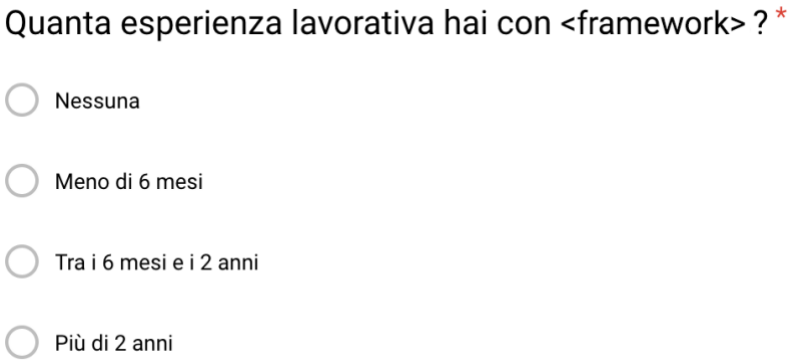

- Flutter




Dalle risposte ottenute sembra che saper programmare nello stesso linguaggio di programmazione con cui il framework è stato costruito è il principale fattore di scelta.
Il secondo fattore di scelta per Xamarin.Forms è l'avere già librerie/logiche scritte in C# da poter riusare e aver già utilizzato Xamarin.Native per condividere le logiche (ma non le interfacce), rispettivamente con 5 e 4 preferenze.
Per React Native un grosso bonus è dato dall'aver lavorato con React, ovvero il framework per sviluppo web che è alla base di React Native con 6 preferenze mentre per Flutter il motivo principale che tutti e 3 gli utilizzatori hanno dichiarato è l'aver deciso avendo fatto delle ricerche e aver pensato che fosse il migliore.

## 3.4 DOMANDA 4

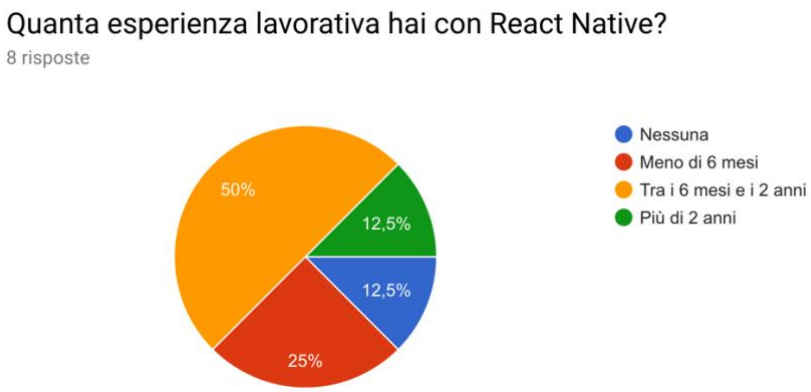
(<framework> è da sostituire con un valore tra Xamarin.Forms, React Native e Flutter)



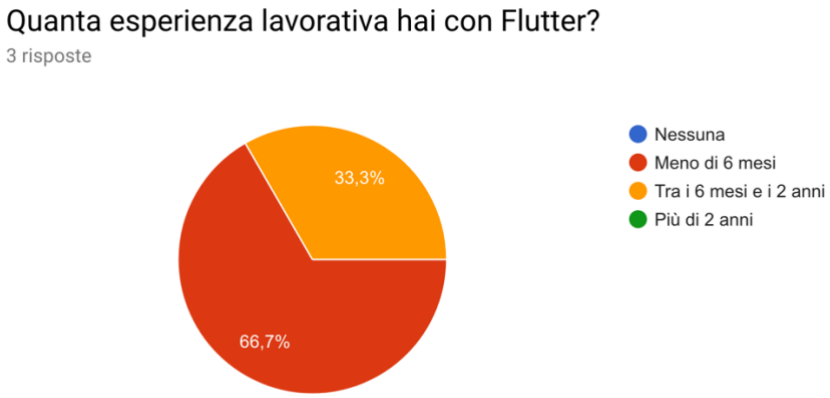

Riposta:

- Xamarin.Forms


- React Native



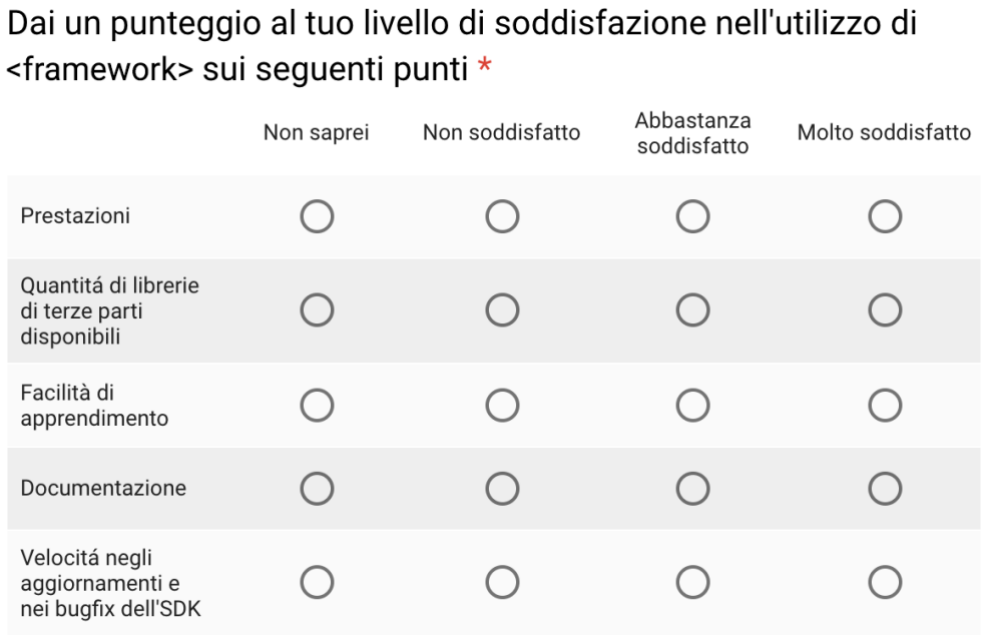

- Flutter




Possiamo notare come ci siano in generale poche persone che usano i framework di sviluppo mobile multi-piattaforma da più di 2 anni, infatti su 13 sviluppatori per Xamarin.Forms, solo 2 (15,4%) sono in questa fascia mentre per React Native solo 1 (12,5%). Ovviamente non ci saremmo aspettati che qualcuno avesse superato la barriera dei 2 anni di utilizzo con Flutter visto che nonostante la sua presentazione risalga al 2015, la diffusione del framework è iniziata solo a metà 2017.
La maggior parte degli sviluppatori di Xamarin.Forms e React Native è nella fascia “6 mesi – 2 anni” mentre per Flutter la fascia con più preferenze è quella “meno di 6 mesi”.
Questo ci può aiutare a capire a che livello si possono trovare nuovi sviluppatori nel mercato per un determinato framework, quindi se si deve sviluppare un progetto mobile importante, è giusto tenere conto che avremo più facilita nel trovare personale con più esperienza in Xamarin.Forms e React Native rispetto a Flutter.

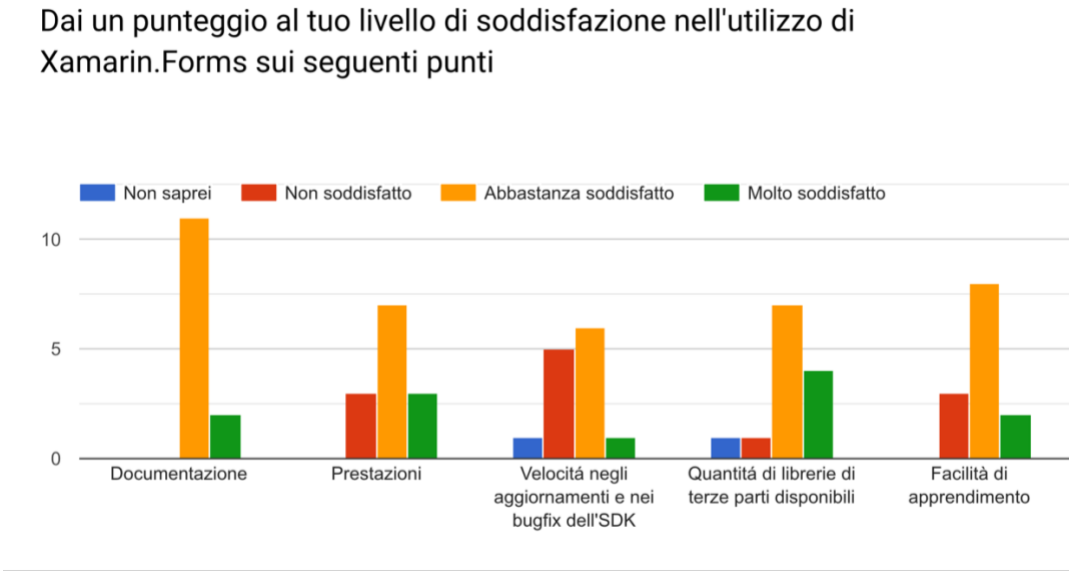
## 3.5 DOMANDA 5

(<framework> è da sostituire con un valore tra Xamarin.Forms, React Native e Flutter)

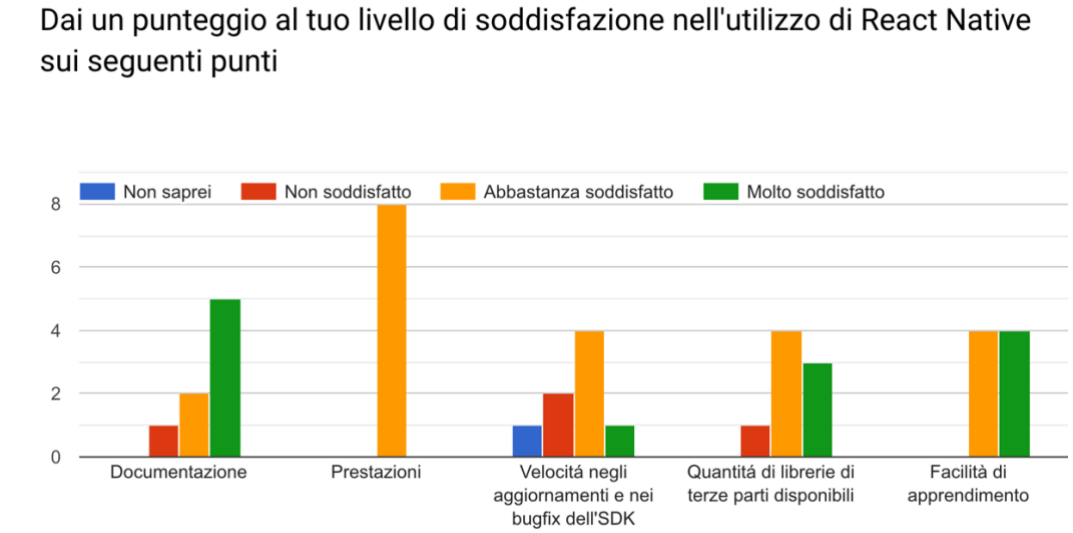



Risposta:

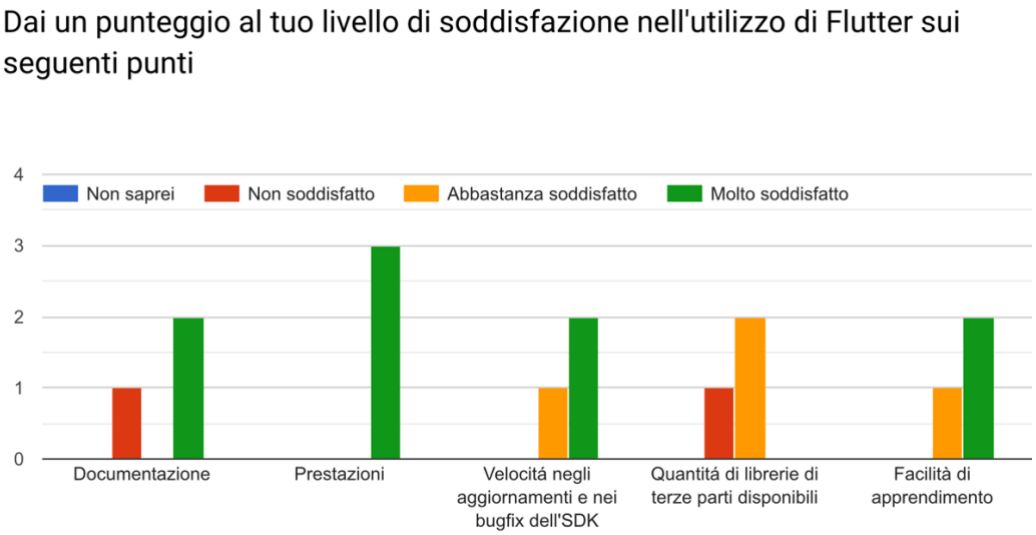
- Xamarin.Forms




- React Native

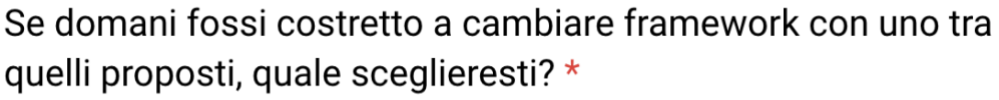



- Flutter




La domanda è stata posta per capire il livello di soddisfazione su dei punti riguardanti sia il framework sia l'ecosistema che gli ruota attorno, fondamentale per un'esperienza di sviluppo appagante in tutte le sue parti.
Per quanto riguarda Xamarin.Forms la maggior parte dei partecipanti ha detto di trovarsi abbastanza bene su tutte le opzioni proposte, soprattutto sulla documentazione dove praticamente tutti hanno dichiarato di trovarsi "abbastanza bene". Il punto debole sembrano essere gli aggiornamenti dell'SDK di Xamarin.Forms, ritenuti da 5 persone su 13 troppo lenti. Da rivedere sembrano essere anche le prestazioni e la facilità di apprendimento, giudicate da 3 persone non adeguate.
In React Native invece, la documentazione e la facilità di apprendimento risultano essere i punti migliori del framework rispettivamente con 5 e 4 preferenze con "molto soddisfatto". Due sviluppatori esprimono la loro insoddisfazione per la lentezza degli aggiornamenti dell'SDK.
Analizzando i dati di Flutter, i partecipanti hanno tutti espresso in maniera unanime che le prestazioni sono ottime, probabilmente merito del motore grafico alla base del framework e dell'obbiettivo degli sviluppatori di farlo girare a 60fps costanti anche con animazioni complesse. Il punto peggiore su cui si sono trovati d'accordo è la bassa quantità di librerie disponibili, segno della gioventù del framework e che migliorerà col tempo solamente se sempre più sviluppatori abbracceranno Flutter come strumento di sviluppo mobile.

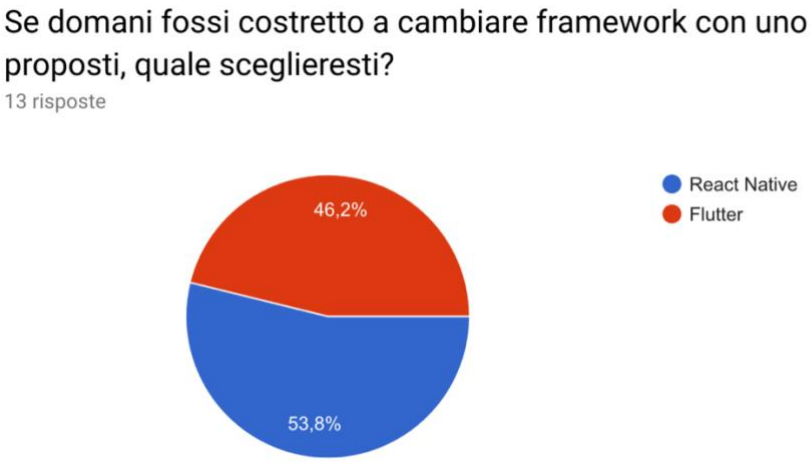
### 3.6 DOMANDA 6



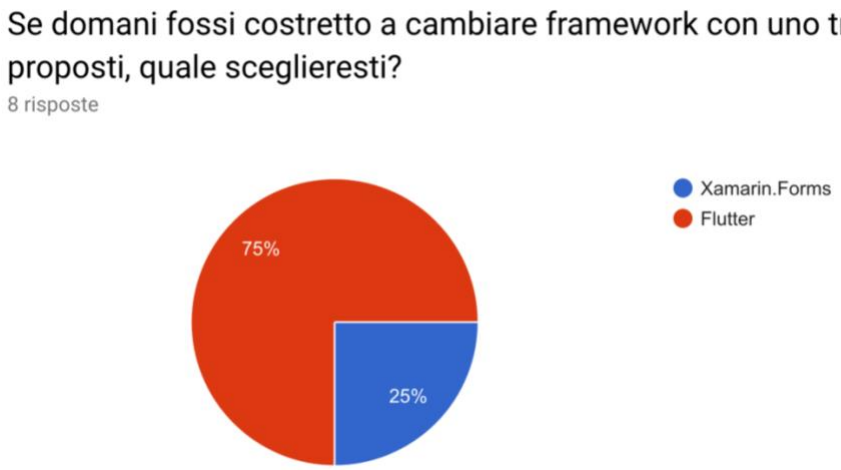

Le opzioni proposte in questo caso variavano in base al sondaggio in cui ci si trovava, ovvero se si era scelto Xamarin.Forms come linguaggio più conosciuto nella 2° domanda, le alternative disponibili erano proprio gli altri 2 linguaggi scartati cioè React Native e Flutter, e così via per tutti e 3 i framework

Risposta:

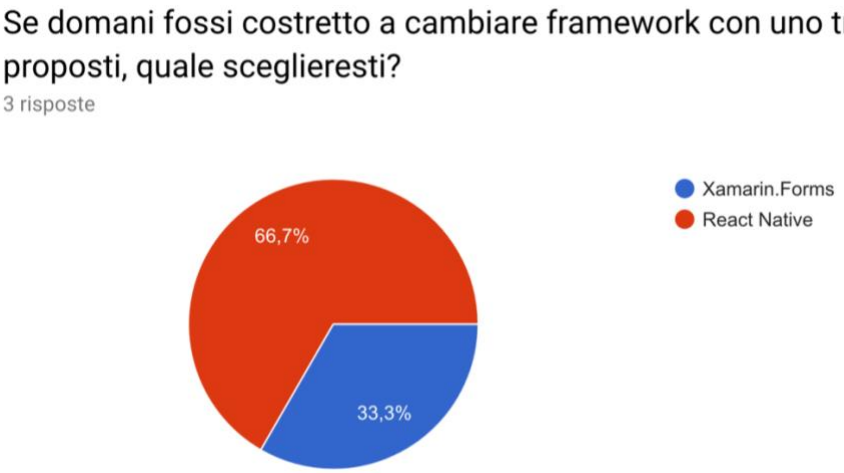
- Xamarin.Forms




- React Native




- Flutter

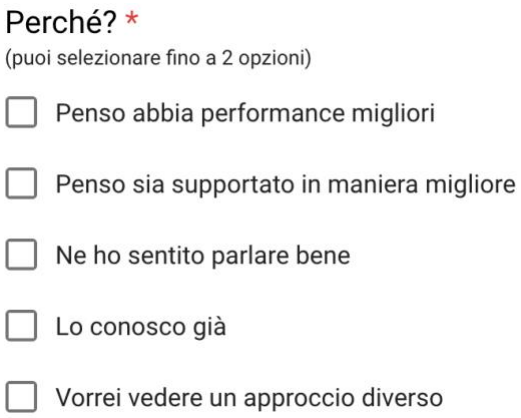



La domanda era per mettere il partecipante in una situazione particolare, ovvero una domanda che simula un'imposizione dall'alto (vedibile come il capo del team che decide di cambiare framework senza sentire il parere di nessuno) che di punto in bianco si trova a

dover scegliere su 2 opzioni che non conosce ma di cui potrebbe essersi informato aver sentito parlare dei colleghi a riguardo.
Gli sviluppatori Xamarin.Forms sono praticamente divisi, infatti su 13, 7 cambierebbero con React Native e 6 con Flutter. Questo equilibrio viene rotto per React Native dove 6 sviluppatori preferirebbero passare a Flutter mentre solo 2 preferirebbero passare a Xamarin.Forms.
Per ultimi, 2 sviluppatori Flutter su 3 preferirebbero passare a React Native mentre solo 1 a Xamarin.Forms.

Quello che colpisce di queste risposte è che nonostante siano ancora pochissimi ad usare Flutter, molti sarebbero disposti a dargli fiducia se dovessero scambiarlo con il loro framework di riferimento. Rimane da analizzare l'ultima domanda per capire il perché di questa scelta.

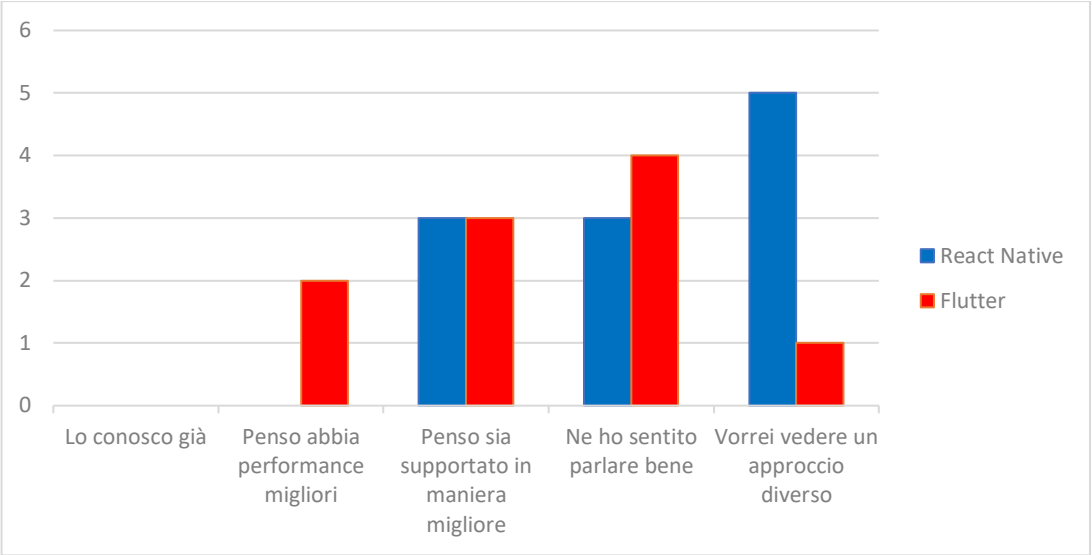
### 3.7 DOMANDA 7



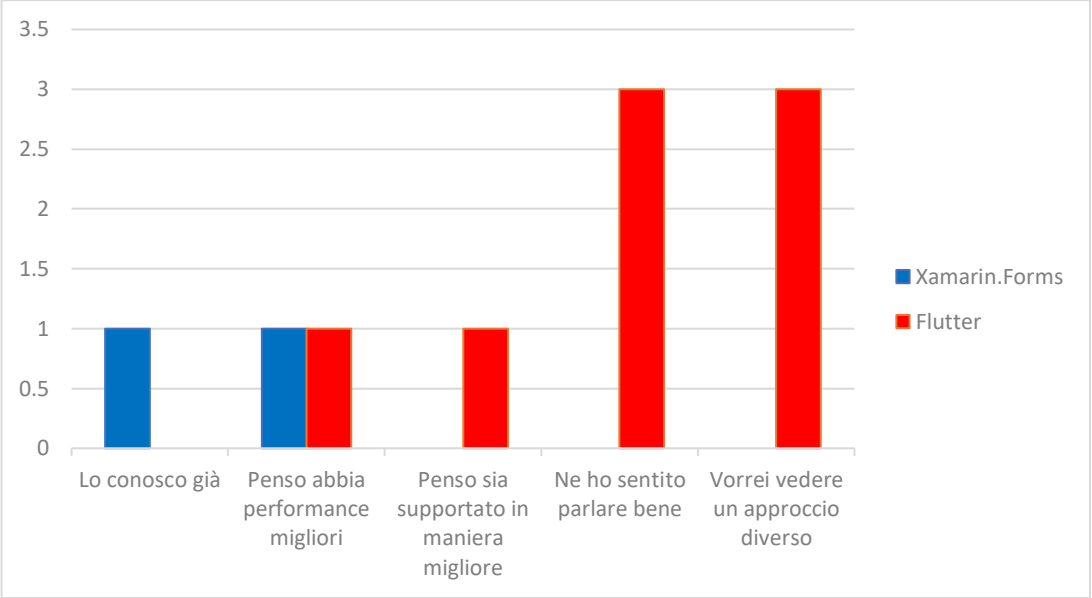

Le risposte vanno divise rispetto alla selezione di una delle due alternative della domanda precedente, quindi è necessario separare ad esempio il perché di chi usando Xamarin.Forms ha scelto React Native e di chi ha invece scelto Flutter. Sono stati creati 3 grafici che rappresentano le motivazioni dietro ogni scelta alla precedente domanda.

Risposta:

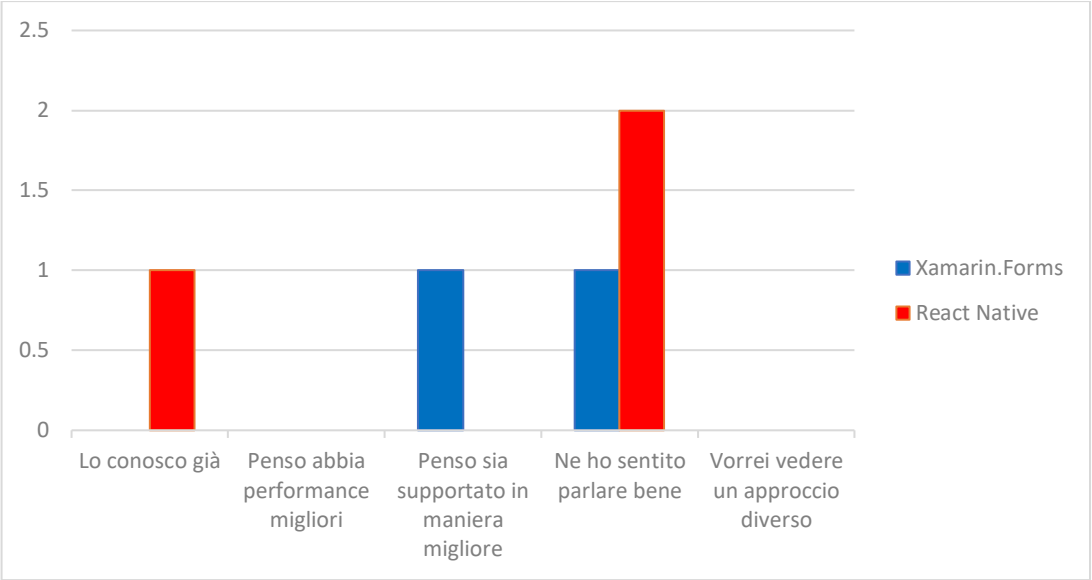
- Xamarin.Forms




- React Native




- Flutter




Chi usa Xamarin.Forms e cambierebbe in React Native per prima cosa vorrebbe vedere un approccio diverso all'implementazione multi-piattaforma. In effetti lo sviluppo con i due framework è alquanto diverso, se da un lato abbiamo Xamarin.Forms che segue i principi del MVVM dall'altro abbiamo React Native con la gestione dello stato e delle "props". Per chi invece sceglierebbe Flutter le voci "ne ho sentito parlare bene" e "penso sia supportato in maniera migliore" hanno vinto rispetto alle altre.

Per chi proviene da React Native e preferirebbe passare a Flutter le voci "ne ho sentito parlare bene" e "vorrei vedere un approccio diverso" staccano nettamente le altre, dimostrando ancora una volta come intorno a Flutter Google si stia impegnando seriamente per farlo conoscere al mondo e mostrare le sue potenzialità.

Chiudono poi i 3 sviluppatori di Flutter dei quali 2 passerebbero a React Native soprattutto per averne sentito parlare bene e l'ultimo che passerebbe a Xamarin.Forms per gli stessi motivi.

# 4. SVILUPPO APP DEMO

Per valutare questi framework, si è deciso di creare un'applicazione TODO di demo, ovvero un'app in cui è possibile inserire del testo affiancato da uno switch che permette di impostare l'elemento come completato. La rimozione del TODO dalla lista è affidata ad una apposita icona con la classica rappresentazione a forma di "bidone".

Per valutare al meglio ogni framework si è deciso di gestire la lista dei TODO tramite un database. Questo permette all'utente di riprendere da dove era rimasto ogni volta che l'app viene aperta e permette a noi di valutare l'integrazione delle librerie disponibili che accedono alle API specifiche di ogni SDK nativo.
Sia Android che iOS usano al loro interno SQLite[7] che è il database più leggero in assoluto ad implementare le specifiche del linguaggio SQL ed è per questo che è stato scelto per

[7] https://www.sqlite.org/index.html

essere inserito nei sistemi operativi mobile da Apple e Google.

Lo schema dell'applicazione che andremo a creare, è il seguente:

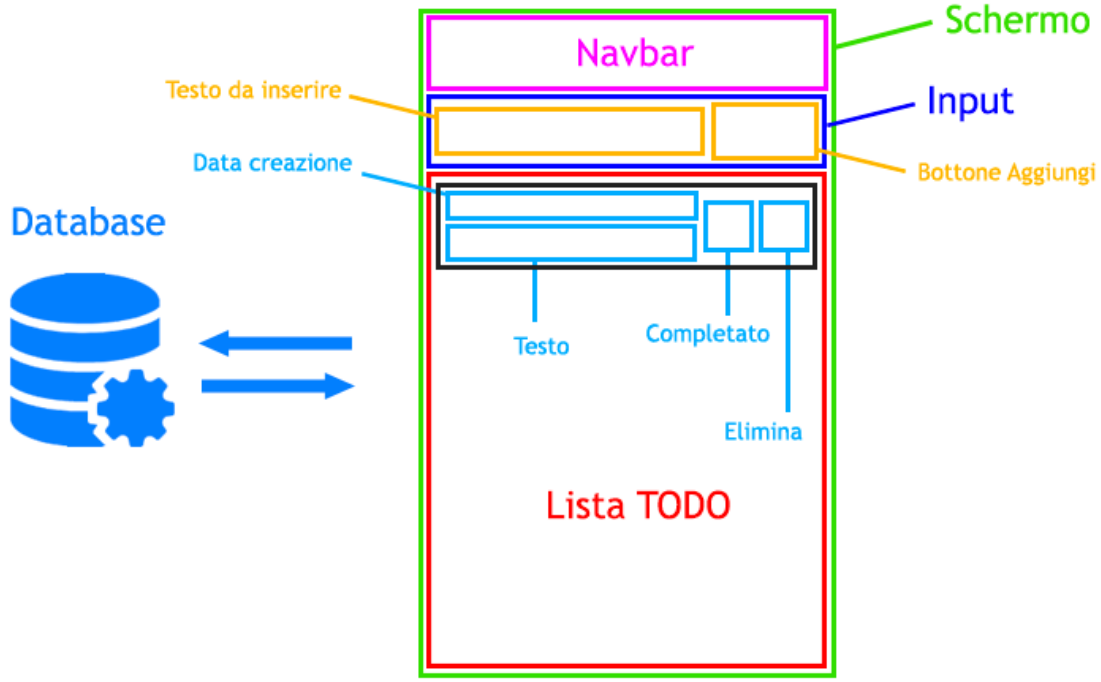



*Figura 5 - Wireframe applicazione demo*

Questo sarà la base su cui costruire l'interfaccia e servirà da linea guida per l'implementazione in ogni piattaforma.

L'ultima cosa da tenere a mente è che l'applicazione implementata nelle 5 piattaforme non segue i dettami dell'architettura perfetta, il codice non è stato ottimizzato e non è sarà "l'implementazione migliore possibile per quella piattaforma", per due motivi:

1. Per scrivere del codice ottimizzato alla perfezione con tutte le migliori librerie e pattern serve una grandissima esperienza con quella piattaforma e tutto l'ecosistema annesso.
2. Lo scopo della demo non è quello di produrre del codice perfetto ma di valutare in quanto tempo una persona che non lavora con quel framework può comunque riuscire a creare qualcosa di funzionante.

Tutti gli sviluppi sono stati effettuati su un emulatore Android equipaggiato con le API 27 che simulava un dispositivo di tipo Nexus 5.
Su iOS invece è stato usato un simulatore che riproduceva le caratteristiche di un iPhone XR con iOS 12.1.

## 4.1 SVILUPPO ANDROID NATIVO

Iniziamo dall'implementazione nativa su Android, la piattaforma con la quale colui che scrive questa tesi ha maggiore dimestichezza dato che è parte del suo lavoro quotidiano.
Tutto il lavoro è stato svolto su Android Studio, ambiente ufficiale di sviluppo di app native per Android.

Per il database è stata usata la classe standard messa a disposizione dall'SDK di Android per gestire i database, l'**SQLiteOpenHelper**, che con poche righe di codice ci permette di creare il database e la tabella che conterrà i nostri elementi TODO.
Per la parte di interfaccia abbiamo bisogno di un container che contenga tutti gli altri componenti, su Android ne esistono di moltissimi tipi e prendono il nome di **ViewGroup**, ovvero delle classi particolari che permettono al loro interno di avere dei componenti (o widget) che possono essere visualizzati e posizionati tra loro. Quelli più conosciuti e utilizzati dagli sviluppatori sono il **LinearLayout** e il **RelativeLayout**, ma nell'ultimo periodo, con l'aumento considerevole delle diagonali dei display e della loro risoluzione, usare queste classi per interfacce dinamiche per riposizionare gli elementi al loro interno in base alla dimensione, era diventato molto complesso. Google ha intuito il problema e da circa un anno a questa parte ha introdotto una nuova ViewGroup chiamata **ConstraintLayout,** che è da poco diventato lo standard in ambito Android per creare interfacce di ogni genere vista la sua versatilità.

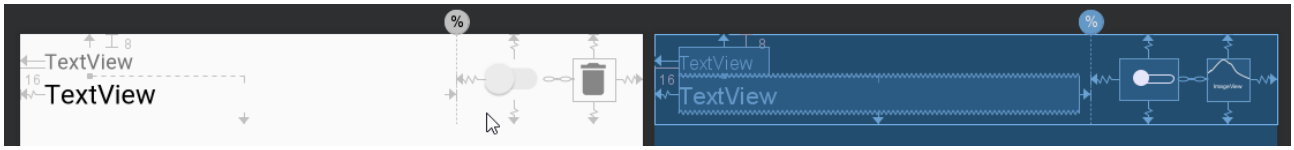
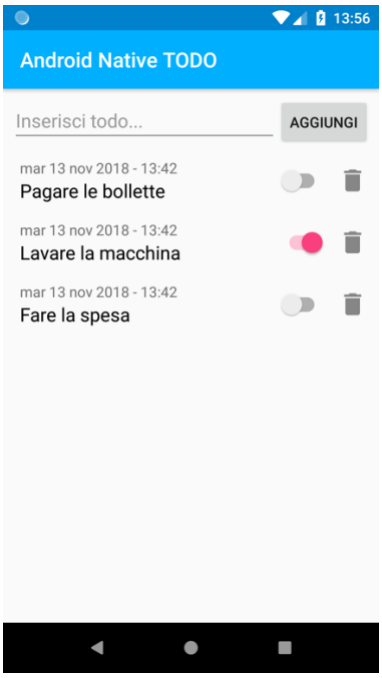


*Figura 6 - ContraintLayout all'opera*

Come è possibile vedere in figura 6, questo nuovo layout è completamente basato sui vincoli, ovvero l'insieme delle frecce che collegano i vari elementi gli uni agli altri per garantire un posizionamento sempre preciso all'interno dell'interfaccia.
Iniziamo quindi mettendo un widget di tipo **EditText** affiancato da un bottone di tipo **Button.** L'EditText su Android è il componente base che permette all'utente di inserire del testo.

Una volta che l'utente inserisce il testo e preme il bottone aggiungi, l'elemento deve essere salvato nel database ed essere mostrato nella lista. Per questo compito si è fatto affidamento sulla **RecyclerView**, ovvero una particolare ViewGroup che genera delle righe per ogni elemento di una lista che gli viene passato; per fare questo ha bisogno che implementiamo una classe particolare sfruttando un pattern di programmazione che si chiama "Adapter pattern". L'adapter è una classe che permette la comunicazione tra due interfacce distinte, in questo caso da indicazioni alla RecyclerView su come devono essere costruiti gli elementi di ogni TODO da mostrare.
Questo è il risultato finale dello sviluppo:




Questa la relativa tabella di valutazione associata allo sviluppo:

| | Valutazione |
|---|---|
| Documentazione | **4/5** |
| Tempo di sviluppo | **circa 3 ore** |

La documentazione funziona molto bene ed è piena di guide con esempi completi. Il tempo di sviluppo è stato praticamente irrisorio poiché se si lavora quasi quotidianamente con Android un'app demo così semplice non è un problema.

## 4.2 SVILUPPO iOS NATIVO

Per sviluppare su iOS per prima cosa è necessario un computer Apple con installato MacOS e Xcode, l'ambiente di sviluppo ufficiale per creare applicazione iOS native.

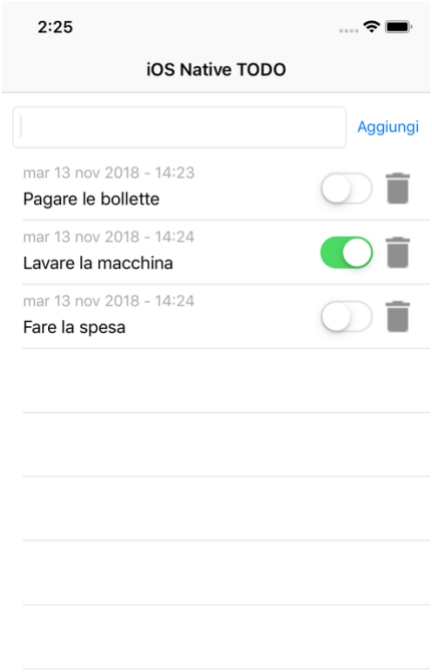
Lo sviluppo dell'applicazione è proceduto senza intoppi, l'unica mancanza riscontrata è legata all'interfaccia, infatti dopo aver iniziato lo sviluppo delle varie applicazioni usando una checkbox per segnalare il completamento del TODO, abbiamo riscontrato che nell'SDK di iOS non è presente nessun componente che rappresenti una normale checkbox. Apple per i suoi dispositivi offre un'alternativa chiamata "Switch" che di base è la stessa cosa ma con una grafica leggermente diversa, quindi piuttosto che inserire una libreria solamente per avere la checkbox come in Android, si è preferito cambiare le altre implementazioni in modo che tutte potessero allinearsi graficamente ed evitare quindi di inserire una libreria apposita solo per questo.

Per la parte database si è fatto affidamento ad una libreria già pronta chiamata

**SQLite.Swift**[8], questo perché l'accesso al database tramite SQLite è abbastanza complesso e richiede di dover sfruttare funzioni a basso livello presenti nella libreria di sistema **libsqlite3.dylib** da collegare tramite Xcode al progetto. Usando uno dei principali gestori di dipendenze per la piattaforma iOS, **Cocoapods**[9], si è aggiunta la libreria all'interno del file di configurazione così da poterla scaricare e usare.
L'interfaccia è stata creata usando lo **Storyboard** che è il default per la navigazione dell'app e i componenti di interfaccia sono stati inseriti usando i vincoli dell'**AutoLayout**. L'AutoLayout è qualcosa di molto simile al ConstraintLayout di Android, esso permette di definire la posizione dei componenti a schermo in relazione a quelli che lo circondano o in relazione alla posizione di quelli gerarchicamente più in alto (quindi più esterni).

In iOS, le liste di oggetti vengono mostrate a schermo da un componente chiamato **UITableView**, esso permette di controllare i layout che vengono impiegati nella creazione di ogni riga e le proprietà dei widget al suo interno, esattamente come succede per la RecyclerView di Android.

Questo il risultato finale dello sviluppo con iOS nativo:




Questa la relativa tabella di valutazione associata allo sviluppo:

| | Valutazione |
|---|---|
| Documentazione | **2/5** |
| Tempo di sviluppo | **circa 15 ore** |

[8] https://github.com/stephencelis/SQLite.swift
[9] https://cocoapods.org/

La documentazione Apple non si è resa utile quanto quella Android, l'unica guida ottima è quella sull'AutoLayout ma si trovano comunque tantissime guide online fatto molto bene che sopperiscono alla documentazione ufficiale in maniera ottima. Swift sembra essere un ottimo linguaggio anche se non è facile abituarcisi velocemente.

## 4.3 XAMARIN FORMS

### 4.3.1 STORIA

Iniziamo con un po' di storia di Xamarin e di come siamo arrivati a Xamarin Forms.
Il progetto da cui tutto è nato si chiama Mono, un progetto iniziato nel lontano 2001 dall'azienda chiamata Ximian che vide l'opportunità di sfruttare le specifiche di Microsoft definite nella CLI "**Common Language Infrastructure**" l'anno prima.
L'obbiettivo dichiarato del team era uno, fare in modo che il codice scritto con il .NET Framework potesse girare sia su Windows che su Linux.
La versione 1.0 di Mono uscì solo nel 2004, tre anni dopo l'annuncio, e la cosa non fu accolta benissimo poiché in molti avevano paura che Microsoft denunciasse l'azienda per infrazione di brevetti relativi all'implementazione (cosa che poi non avvenne mai).
Nel 2009 arrivò **MonoTouch**, la versione di Mono che permetteva di far girare lo stesso codice su dispositivi iOS e inizio poco dopo lo sviluppo di **Mono for Android** che era la stessa cosa di MonoTouch solamente per il sistema Android.
Entrambi sono dei contenitori chiamati "wrapper", che al loro interno trasformano le chiamate C# in chiamate alle API native dei rispettivi SDK iOS e Android.

A questo evento seguirono varie acquisizioni dell'azienda da parte di aziende più grandi, fino ad arrivare al 2011, quando **Xamarin**[10] fu fondata a partire dagli sviluppatori originali di Mono che nel frattempo si erano staccati.
Capirono che potevano creare il loro business fornendo un sistema multi-piattaforma basato sul linguaggio C# e grazie ad un accordo con l'azienda da cui si separarono poterono continuare a sviluppare Mono e i suoi derivati senza incorrere in problemi legali. L'azienda iniziò ad essere conosciuta a livello mondiale intorno al 2013 quando presentarono la versione 2.0 dell'**SDK Xamarin**, rinominarono MonoTouch in **Xamarin.iOS** e Mono for Android in **Xamarin.Android**, rinominarono Monodevelop (l'ambiente di sviluppo di Mono) in **Xamarin Studio** e aggiunsero l'integrazione con **Visual Studio**. Tutto questo contribuì a far espandere l'azienda e ha trovare consensi tra gli sviluppatori C#, considerato che tra il 2010 e il 2013 c'è stata l'entrata in scena di **Windows Phone**[11] nel mercato dei sistemi operativi mobile (supportato fin da subito da Xamarin) e avere una piattaforma di sviluppo che era in grado di creare applicazioni per 3 sistemi contemporaneamente fece gola a moltissimi.
Con l'andare del tempo almeno questa attrazione è andato scemando a causa del fallimento del progetto Windows Phone, che la stessa Microsoft ha poi deciso di abbandonare

[10] https://en.wikipedia.org/wiki/Xamarin Curiosità: Il nome Xamarin deriva dalla scimmia Tamarino (in inglese Tamarin) con la X al posto della T iniziale, come usavano solitamente chiamare i prodotti nell'azienda fondatrice di Mono, Ximian.
[11] https://it.wikipedia.org/wiki/Windows_Phone

rimpiazzandolo con **UWP**[12] "Universal Windows Platform" a partire da Windows 10.

Fin da subito il vantaggio nell'uso di Xamarin.iOS e Xamarin.Android è stato quello di poter passare allo sviluppo mobile multi-piattaforma senza essere obbligati ad imparare Java e Objective-C, potendo inoltre riusare logica e librerie scritte in precedenza con C#.
Il problema di non poter condividere la stessa interfaccia per entrambe le applicazioni era comunque non trascurabile poiché obbligava gli sviluppatori a continuare a progettare le interfacce in maniera nativa.
Per risolverlo, nel 2014, l'azienda presenta **Xamarin.Forms,** che sulla carta promette di essere lo strumento adatto in quei casi in cui l'interfaccia utente non necessità di particolare cura o animazioni avanzate.

La figura 7 ci illustra la differenza tra il metodo di sviluppo tradizionale con Xamarin chiamato anche Xamarin.Native e quello con Xamarin Forms:

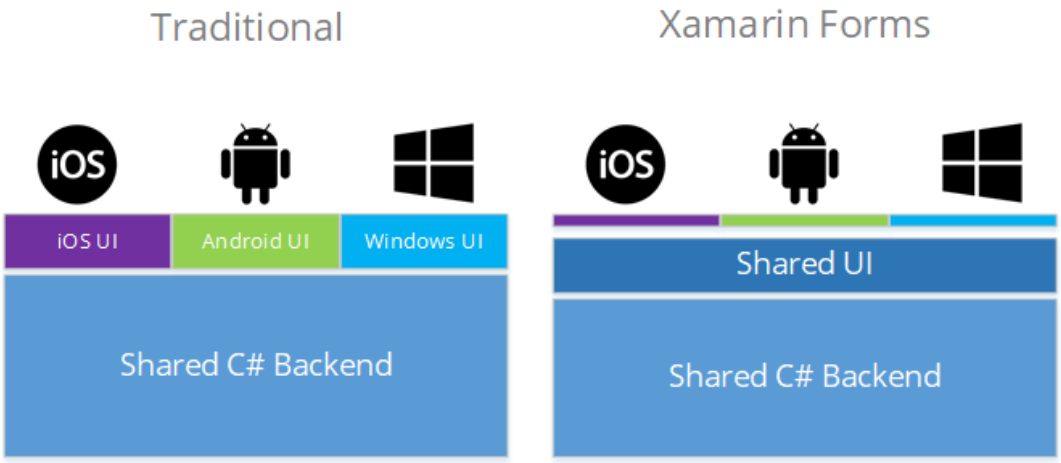



*Figura 7 - Xamarin VS Xamarin.Forms*

Nel 2016 arriva l'acquisizione di Xamarin da parte di Microsoft che rende tutto il codice open source rimuovendo i costi di licenza per l'utilizzo (ultimo vero scoglio rimasto) e inizia così un percorso in cui grazie alle ingenti risorse fornite, Xamarin migliora di molto sia a livello di stabilità che in velocità di sviluppo di nuove funzionalità.

### 4.3.2 BACKGROUND TECNICO

Veniamo quindi ora alla parte tecnica e implementativa, in figura 8 è mostrato il funzionamento ad alto livello di Xamarin.Forms[13]

[12] https://it.wikipedia.org/wiki/Universal_Windows_Platform
[13] https://blogs.msdn.microsoft.com/whereismysolution/2018/02/26/xamarin-forms-using-plugins-vs-native-ui-part-2/

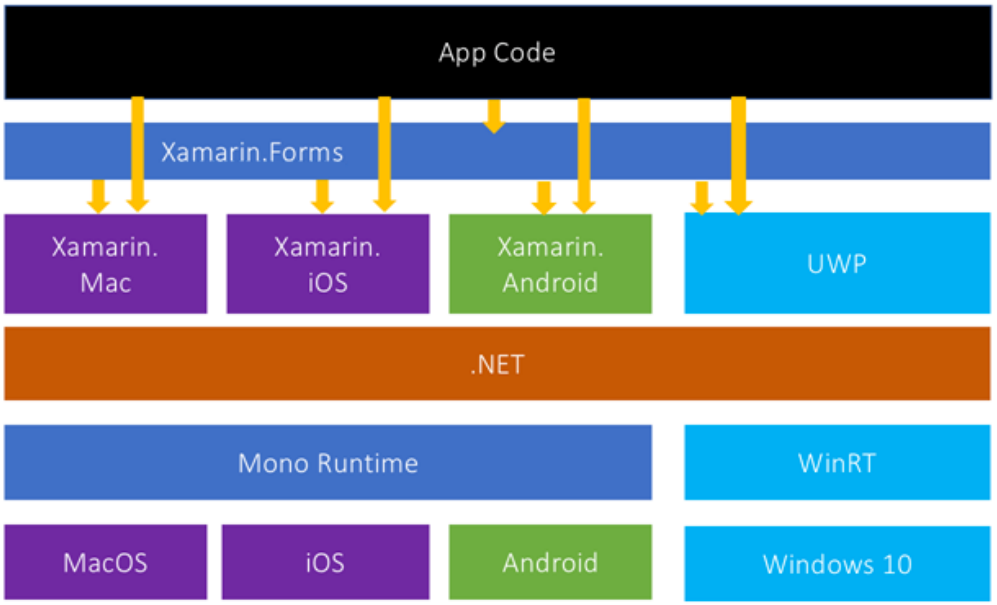



*Figura 8 - Xamarin.Forms in dettaglio*

Vediamo come tutto il codice multi-piattaforma che scriviamo nella nostra applicazione, viene gestito da Xamarin.Forms che internamente lo converte per farlo girare con le API nella specifica implementazione, con la possibilità di connetterci quando necessario direttamente a Xamarin.iOS e Xamarin.Android (a loro volta collegati alle API native).

Per fare questo Xamarin.Forms si serve del cosiddetto "**Renderer**", ovvero un qualcosa che è in grado di tradurre un oggetto dell'interfaccia in quello nativo della piattaforma.

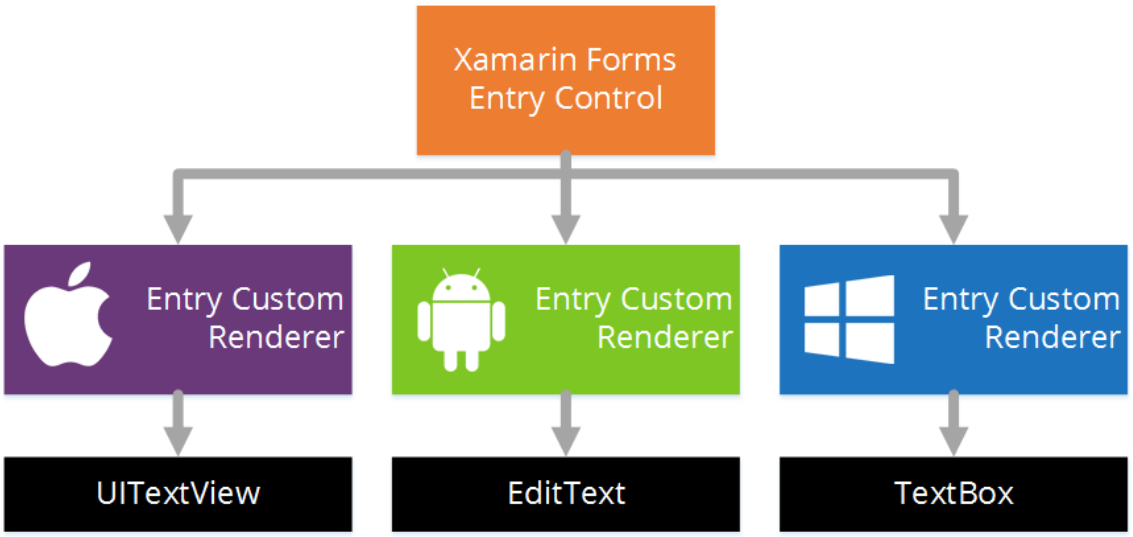



*Figura 9 - Funzionamento Renderer*

Nella figura 9 qui sopra possiamo vedere come l'oggetto "**Entry**" che rappresenta un campo in cui l'utente può scrivere del testo, viene tradotto grazie ai rispettivi renderer in quello nativo di destinazione, perciò un oggetto Entry verrà tradotto su Android in una EditText mentre in iOS in una UITextView.
Nel caso ci servisse un componente nuovo, possiamo crearlo da zero o modificarne uno già esistente sfruttando lo stesso metodo. Per creare le interfacce può essere usato sia codice C#

che codice XAML, di cui viene riportato un esempio qui sotto:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Phoneword.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
      <Label Text="Enter a Phoneword:" />
      <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
      <Button x:Name="translateButon" Text="Translate" Clicked="OnTranslate" />
      <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall"
/>
    </StackLayout>
</ContentPage>
```

XAML, ovvero “eXtensible Application Markup Language” (una versione potenziata dell’XML) è il linguaggio scelto da Microsoft per la costruzione delle interfacce multi-piattaforma. Gli elementi che si intravedono nel codice come **ContentPage** o **StackLayout**, sono tutti “widget” ovvero elementi di interfaccia che sfruttano il meccanismo dei renderer descritto sopra che uniti insieme compongono le varie schermate della nostra applicazione.

Altra cosa molto importante in Xamarin.Forms è la possibilità di scegliere 3 differenti approcci per la creazione dei progetti[14].
Gli approcci disponibili sono:

1. .NET Standard (consigliato)
2. Shared Projects
3. Portable class libraries o PCL (deprecato)

L’approccio .NET Standard è molto simile a quello delle PCL, in realtà è un’evoluzione di quest’ultime che grazie al .NET Core 2.0 semplifica di molto la precedente implementazione e permette quindi l’utilizzo di quasi tutte le API definite nel .NET Framework 4.6.1.
La parte più interessante di questa implementazione è che per referenziare del codice nativo all’interno del progetto possiamo usare la classe `DependencyService<NomeInterfaccia>`[15], che ci permette di creare un’interfaccia C# di dialogo con le implementazioni native effettuate su ogni singola piattaforma, in questo modo l’implementazione nativa rimane confinata nel progetto di origine come possiamo vedere in figura 10 sottostante.

[14] https://docs.microsoft.com/it-it/xamarin/cross-platform/app-fundamentals/code-sharing
[15] https://docs.microsoft.com/it-it/xamarin/xamarin-forms/app-fundamentals/dependency-service/introduction

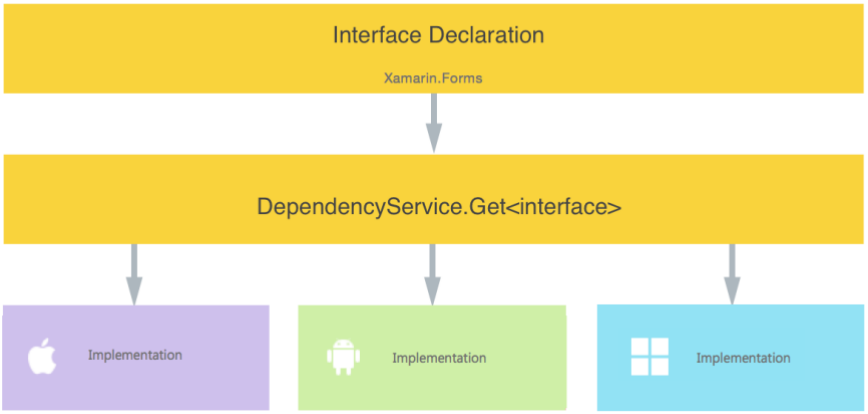



*Figura 10 – DependencyService*

Al contrario, il vantaggio degli Shared Projects è la possibilità di aggiungere delle direttive per il compilatore direttamente nel codice condiviso che gli permettono di capire che parte di codice usare quando compiliamo per una specifica piattaforma.
Ad esempio, è possibile aggiungere un blocco di codice di questo tipo in qualsiasi punto del progetto condiviso,

```
#if __ANDROID__
     //codice specifico per Android
#elif __IOS__
     //codice specifico per iOS
#endif
```

dove se si sta compilando l'applicazione per Android verrà aggiunto solo il primo blocco dell'if, per iOS viene invece usato l'altro blocco.
Questo semplifica le operazioni, nei casi in cui si ha molto codice che utilizza le API della piattaforma nativa ma lo rende anche molto meno leggibile, la stessa Microsoft consiglia ad oggi di utilizzare i progetti in modalità .NET Standard perché migliori degli altri approcci sotto tutti i punti di vista.

Come ultima cosa dobbiamo tenere presente che un'applicazione Xamarin.Forms per funzionare ha bisogno della sua macchina virtuale, ovvero Mono, che abbiamo descritto abbondantemente in precedenza, più le altre librerie fondamentali per il funzionamento che contribuiscono ad aumentare la dimensione del pacchetto di installazione finale, nonostante venga fatto un lavoro di ottimizzazione che rimuove tutte le parti non utilizzate del framework.

### 4.3.3 SVILUPPO DEMO

Per sviluppare l'app con Xamarin Forms, è stato usato un ambiente di sviluppo abbastanza nuovo chiamato Visual Studio Community for Mac (così da poter compilare anche per iOS). Diciamo che quello che si è presentato è un ambiente molto simile a quello che era il vecchio Xamarin Studio (basato su MonoDevolop) con alcune caratteristiche migliorate ed altre aggiunte.
Per quello che abbiamo potuto provare l'ambiente è molto buono pur riscontrando qualche bug che delle volte ha impedito di continuare lo sviluppo. Ad esempio mentre inserivamo

l'icona per cancellare i TODO nella apposita cartella "Assets.xcassets" di iOS, qualcosa è andato storto e il progetto non era più compilabile. Dopo aver perso mezz'ora per capire dove fosse il problema, abbiamo deciso di aprire il progetto con Visual Studio Community su di un altro PC con Windows e siamo riusciti a sistemare il tutto.

La parte database dell'app è stata affidata ad una libreria chiamata **sqlite-net**[16] che espone dei metodi C# implementati in maniera nativa in ogni piattaforma. In questo modo abbiamo una interfaccia in comune agli SDK dei due sistemi operativi e questo ci fa risparmiare tempo non dovendo ricreare la stessa cosa da zero.

Per creare la prima versione multi-piattaforma della nostra applicazione, siamo partiti dal prendere il layout e studiare in che modo potesse essere riproposto in modalità multi-piattaforma.
Xamarin.Forms mette a disposizione vari container per i componenti di interfaccia, uno di quelli più usati è sicuramente lo **StackLayout** che ci permette di posizione i componenti in verticale o in orizzontale mimando gli stessi comportamenti del **LinearLayout** presente su Android nativo.
Sfruttando questo componente siamo riusciti a disegnare l'interfaccia velocemente scrivendo il codice XAML che possiamo vedere qui sotto:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamarinApp"
             x:Class="XamarinApp.MainPage">

    <StackLayout Orientation="Vertical" HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" Margin="5">
        <StackLayout Orientation="Horizontal" HorizontalOptions="FillAndExpand">
            <Entry x:Name="textInput"  HorizontalOptions="FillAndExpand" Placeholder="Inserisci todo..." />
            <Button HorizontalOptions="Start" Text="Aggiungi" Clicked="OnItemAdded"/>
        </StackLayout>
        <ListView x:Name="listView" Margin="5" HasUnevenRows="true">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <StackLayout Margin="5" Orientation="Horizontal" HorizontalOptions="FillAndExpand">
                            <StackLayout Orientation="Vertical" HorizontalOptions="FillAndExpand">
                                <Label Text="{Binding Date}" HorizontalOptions="StartAndExpand" TextColor="Gray"/>
                                <Label Text="{Binding Text}" VerticalOptions="StartAndExpand"
                                       HorizontalOptions="StartAndExpand" FontSize="16" TextColor="Black"/>
                            </StackLayout>
                            <StackLayout Orientation="Horizontal" HorizontalOptions="End" VerticalOptions="Center">
                                <local:TodoSwitch IsToggled="{Binding Completed}" Todo="{Binding .}"
                                                  Toggled="SwitchToggled" />
                                <Image Source="delete" WidthRequest="30">
                                    <Image.GestureRecognizers>
                                        <TapGestureRecognizer  CommandParameter="{Binding .}" Tapped="Image_Tapped" />
                                    </Image.GestureRecognizers>
                                </Image>
                            </StackLayout>
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>

</ContentPage>

```

*Figura 11 - Codice XAML interfaccia*

La cosa che ci ha portato via più tempo è stato capire le logiche del binding (o collegamento) tra l'interfaccia e il codice C# vero e proprio. Abbiamo quindi appreso che Xamarin funziona al meglio con il binding degli oggetti, ovvero con la possibilità di collegare le proprietà del codice reale (detto anche code behind) a quelle dell'interfaccia evitando così di dover impostare a mano tutti i valori per i singoli componenti. Questa cosa

[16] https://github.com/praeclarum/sqlite-net

ovviamente è fattibile anche in Android e iOS nativi ma diciamo che non è la scelta principale che spesso viene proposta dalle rispettive documentazioni.
Per effettuarne uno è necessario solo definire la parola "**Binding**" seguito dal nome della proprietà che vogliamo collegare. Quindi prendendo il codice mostrato in figura, per collegare il testo del TODO all'interfaccia, ci è bastato collegare il valore della proprietà testo della Label con quello della proprietà Text dell'oggetto TODO, in questo modo:

```
<Label Text={Binding Text}…
```

Nel caso dell'icona per eliminare i TODO, abbiamo sfruttato una proprietà dell'oggetto **TapGestureRecognizer** che permette di ascoltare i tap effettuati su alcuni componenti di interfaccia e poter passare così l'oggetto tramite la proprietà **CommandParameter** dichiarata nel codice XAML. Per lo Switch invece abbiamo avuto qualche problema maggiore, poiché non è possibile chiamare una proprietà CommandParameter come fatto per il componente **Image**. È stato necessario quindi creare una versione personalizzata del componente Switch che ho chiamato **TodoSwitch,** per fare il binding con tutto l'oggetto TODO nel codice XAML, in questo modo:

```
<local:TodoSwitch IsToggled="{Binding Completed}" Todo="{Binding .}" Toggled="Sw
itchToggled" />
```

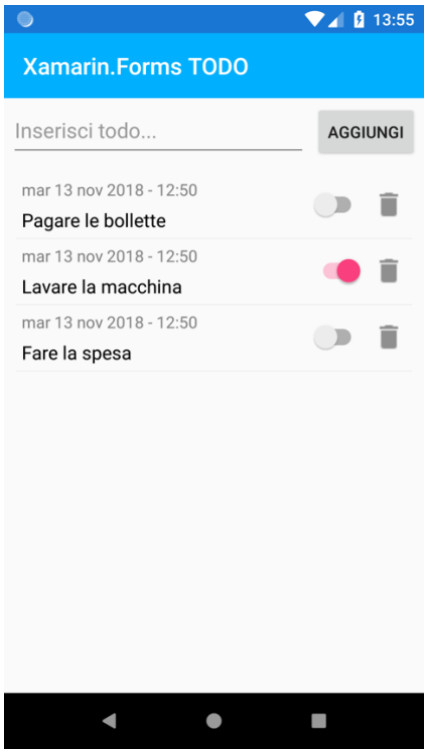
Al tap sullo switch viene quindi chiamato il metodo SwitchToggled presente nella classe MainPage.xaml.cs che contiene l'id del TODO da aggiornare nel database. Questa parte di implementazione crediamo possa essere eseguita in maniera migliore avendo qualche conoscenza aggiuntiva che la documentazione ufficiale però non ci ha fornito.

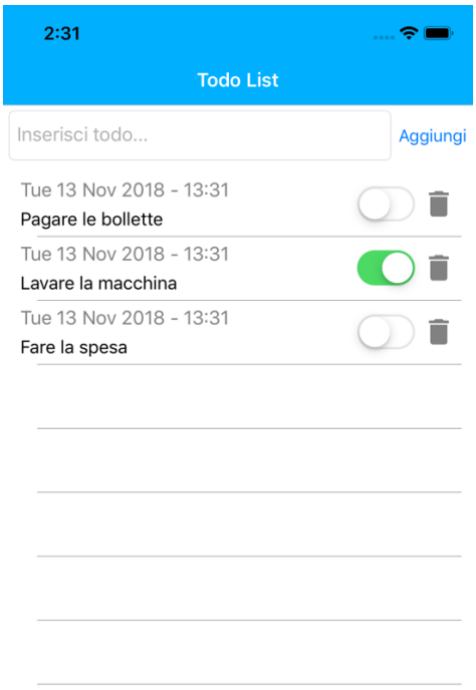
Per contenere la lista dei TODO si è deciso di usare un **ObservableRangeCollection**[17], una classe fornita da uno degli sviluppatori di Xamarin stessa, che contiene al suo interno una lista di oggetti costruita in modo tale che ogni volta che questa viene modificata, emette degli eventi che aggiornano in automatico la **ListView** con cui sono renderizzate le varie righe, tenendo così sincronizzata la lista dei TODO con quelli visibili a schermo.

Queste le schermate ottenute per Android e iOS al termine della demo in Xamarin.Forms:

Android iOS

[17] https://github.com/jamesmontemagno/mvvm-helpers/blob/master/MvvmHelpers/ObservableRangeCollection.cs




Questa la relativa tabella di valutazione associata allo sviluppo con Xamarin.Forms:

| | **Valutazione** |
|---|---|
| Documentazione | **3/5** |
| Tempo di sviluppo | **circa 15 ore** |

Documentazione abbastanza chiara e la maggior parte dei problemi è possibile risolverla leggendo soluzioni a domande poste sul forum ufficiale di Xamarin. Su GitHub sono presenti moltissimi esempi utili e aggiornati che danno una grossa mano nel partire correttamente. Nessun problema con il linguaggio poiché conoscendo Java fare il passaggio mentale a C# è stato semplice.

## 4.4 REACT NATIVE

### 4.4.1 STORIA

Era il 2012 quando Mark Zuckerberg ammetteva in una intervista che Facebook aveva riposto troppe speranze nell'HTML5 e che ora ne stavano pagando le conseguenze[18].
Il problema più grosso che la sua azienda stava affrontando erano le numerose critiche alle loro applicazioni mobile, da tutti riconosciute come troppo lente e non abbastanza responsive. Il problema è che inizialmente avevano deciso di utilizzare esclusivamente le `WebView`, ovvero una sorta di browser incorporabile all'interno dell'applicazione (che è

[18] https://mashable.com/2012/09/11/html5-biggest-mistake/

parte delle API native di ogni piattaforma), che non faceva altro che mostrare agli utenti il sito di Facebook in versione mobile.
Si è venuto a scoprire poi che gli smartphone non erano così potenti come si pensava e che spesso e volentieri il problema era la lentezza con cui i motori Javascript renderizzavano le pagine all'interno delle `WebView`, facendo percepire all'utente una latenza eccessiva ai suoi input.
Come segno che Zuckerberg aveva preso la questione sul serio, qualche mese dopo le sue affermazioni fu rilasciata la nuova applicazione di Facebook in versione nativa per iOS e poco dopo iniziò la riscrittura nativa di quella di Android, completata in più passaggi[19].

Nel frattempo, un anno prima, Facebook aveva rilasciato un framework di sviluppo web scritto in Javascript di nome React (simile per scopi al vecchio AngularJS) che si concentrava su due aspetti principali, ovvero la facilità di utilizzo e una gestione semplificata dello stato dell'applicazione, una cosa che ogni sviluppatore web conosce bene e sa che spesso è la parte più problematica.
I feedback su React sono da subito positivi e la community open source inizia ad utilizzarlo per creare i primi prodotti. Sull'onda di questo successo nel 2013, il creatore di React Jordan Walke (un ingegnere di Facebook), annunciò nel team in cui lavorava che era riuscito a creare un prototipo che permetteva di generare dei componenti dell'interfaccia nativa di iOS con Javascript sfruttando quanto era stato fatto con React. A quel punto, con la complicità dei manager dell'azienda fu indetto un Hackaton interno (uno dei tanti che vengono effettuati) per cercare di capire se era possibile migliorare il prototipo e spingerlo a fare qualcosa di veramente utilizzabile.
Dopo due giorni di lavoro qualcuno fu entusiasta del risultato ottenuto e dopo qualche mese decisero di dedicare un team completo allo sviluppo del framework viste le ottime aspettative che si erano venute a creare. Il team proseguì il lavoro per i successivi due anni fino ad arrivare alla "React.js Conf 2015", l'evento organizzato da Facebook ogni anno per mostrare in anteprima le nuove funzionalità di React, in cui venne presentato al mondo per la prima volta quello che poi prese il nome di React Native.

Facebook ha poi deciso di dimostrare le potenzialità di quello che aveva creato convertendo alcune piccole schermate delle sue app di maggiore successo, ovvero Facebook e Instagram[20] usando React Native.

### 4.4.2 BACKGROUND TECNICO

Il funzionamento di React Native è del tutto simile a quello di React, infatti in entrambi il concetto base per gli elementi visibili a schermo è quello dei "componenti".
Il componente è l'unità fondamentale dell'interfaccia, ed esso rappresenta qualcosa che dovrà essere visualizzato a schermo per comporre le varie viste dell'applicazione.

[19] https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-android/10151189598933920/
[20] https://instagram-engineering.com/react-native-at-instagram-dd828a9a90c7

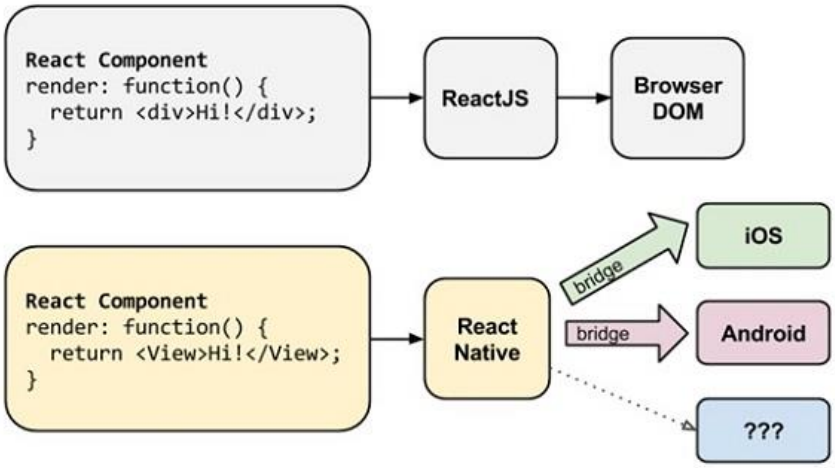



*Figura 12 - React e React Native a confronto*

Come possiamo vedere in figura 12, in React un componente viene dato in pasto al motore Javascript per comporre il DOM, ovvero l'insieme dei tag HTML e CSS che vengono usati dai vari browser per mostrare le pagine web agli utenti. In React Native invece, il componente deve essere convertito nella sua controparte nativa, esattamente come avviene per Xamarin.Forms, quindi abbiamo bisogno di quello che qui viene chiamato "bridge" tra la parte Javascript e la parte nativa di ogni piattaforma.

Un esempio di componente scritto in React Native:

```
export default class HelloComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {text: ''};
  }

  render() {
    return (
      <View style={{padding: 10}}>
        <TextInput
          style={{height: 40}}
          placeholder="Type here to translate!"
          onChangeText={(text) => this.setState({text})}
        />
        <Text style={{padding: 10, fontSize: 42}}>
          {this.state.text.split(' ').map((word) => word && 'Hello').join(' ')}
        </Text>
      </View>
    );
  }
}
```

La parte interna alla funzione `render()` è scritta usando il linguaggio proprio di React Native chiamato JSX (Javascript Extension), inventato dalla stessa Facebook quando è stato introdotto React. È un linguaggio che permette di avere del codice HTML e del codice Javascript mischiato insieme unendo il meglio di entrambi i mondi.

Il funzionamento esatto della parte di rendering di React Native è illustrato in figura 13 qui sotto:

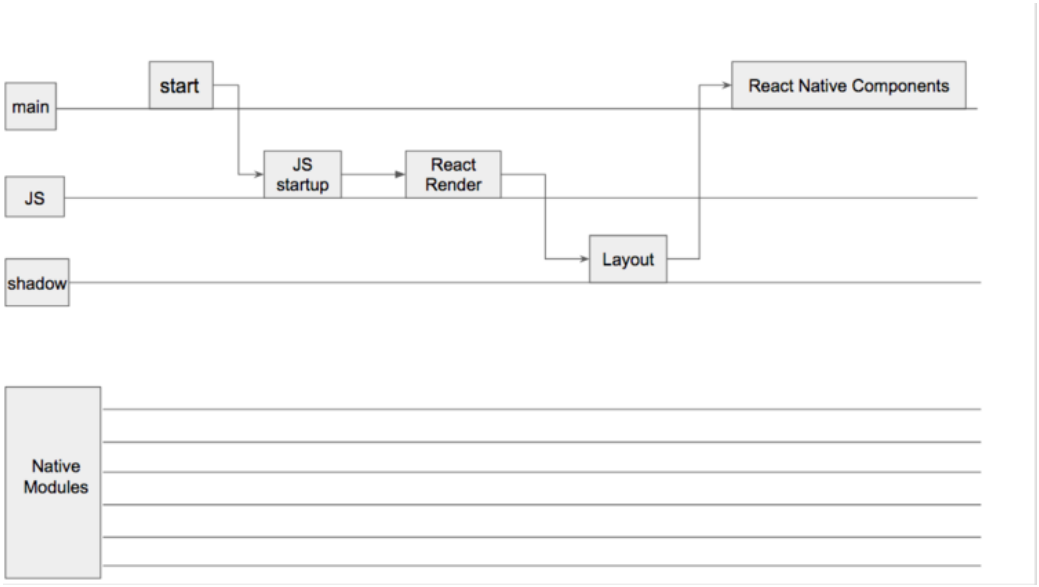



*Figura 13 - Divisione dei thread*

Abbiamo i seguenti thread che si occupano delle varie parti:

1. Main thread o UI thread
2. JS thread
3. Shadow thread
4. Native modules thread

Quando avviamo la nostra applicazione, per prima cosa viene lanciato il **main thread** sulla parte nativa e questo inizia caricando il Javascript presente nel bundle compilato. Una volta finito passa tutti i file caricati al JS thread che inizia quindi a caricare tutti i componenti che fanno parte della schermata da visualizzare per convertirli nella versione nativa della piattaforma su cui l'app sta girando. Completata questa fase i componenti vengono passati allo **shadow thread** che calcola le dimensioni che questi elementi devono avere a schermo. Finiti i calcoli, lo shadow thread rimanda al main thread gli oggetti convertiti con aggiunte le dimensioni calcolate dando la possibilità a quest'ultimo di completare il ciclo mostrando a schermo tutto il necessario per comporre la schermata.

La maggior parte degli eventi dell'interfaccia, come il click di un bottone, viene gestita direttamente dal JS thread, tuttavia con alcuni componenti che richiedono alte prestazioni e bassa latenza come lo scroll di una lista, potrebbero verificarsi piccoli "scatti" che penalizzano l'esperienza utente se il JS thread è impegnato in altre elaborazioni.
Per ovviare a questo problema, React Native per i componenti più critici mette a disposizione delle alternative ottimizzate costruite in modo da "bypassare" il JS thread e che quindi lavorano direttamente sul main thread che è quello che può ottenere le performance migliori dato che ha il controllo diretto sulle API della piattaforma. Ad esempio i componenti `NavigatorIOS` e `ScrollView` sono tra quelli che possono sfruttare questa ottimizzazione.

Quando si ha bisogno di utilizzare un pezzo di codice nativo all'interno di React Native, possiamo utilizzare i "native modules", ovvero del codice scritto nel linguaggio di piattaforma (Java/Kotlin su Android e Obective-C/Swift su iOS) che viene poi reso disponibile tramite dei meccanismi alla parte Javascript condivisa. In iOS ogni native module viene eseguito in un ThreadPool separato mentre su Android di default fanno tutti

parte dello stesso ThreadPool.
Un ThreadPool è solitamente composto da un insieme di thread pre-istanziati messi in una coda, che rimangono in attesa di ricevere del lavoro da svolgere. Ogni volta che il ThreadPool riceve del lavoro da fare, prende uno dei thread disponibili nella coda e gli assegna quel compito e una volta che il thread lo porta a termine torna a riposo nella coda. Questo metodo è molto efficiente nel caso si abbiano tanti piccoli lavori da svolgere per cui la creazione continua di nuovi thread sarebbe contro-producente in termini di prestazioni.

La velocità di sviluppo di un'app è molto alta grazie all'**hot reloading**, ovvero la capacità di vedere subito nel dispositivo di test le modifiche effettuate al codice.
Dietro queste semplici parole si nasconde un sistema complesso rappresentato parzialmente nella documentazione ufficiale dallo schema in figura 14:

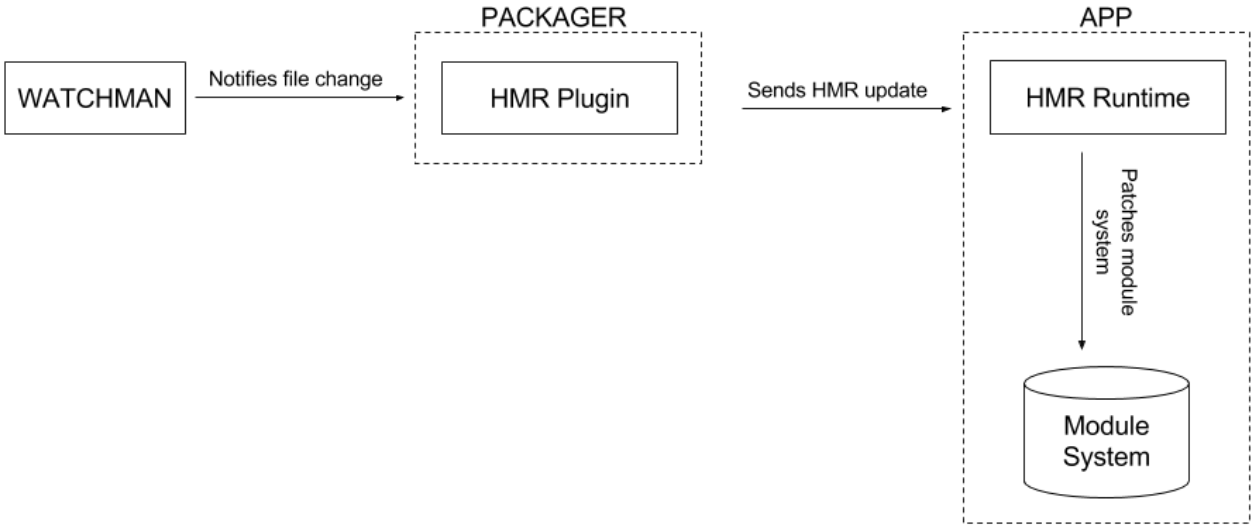



*Figura 14 - Funzionamento Hot Reload*

Per prima cosa bisogna installare un componente di nome **Watchman**, che permette di rilevare le modifiche ai file sorgenti, poi ogni volta che si salva un file modificato nell'editor, l'HMR (Hot Module Replacement) tramite il packager di React Native invia il codice aggiornato all'app che internamente rimpiazza solo la porzione di codice interessata, evitando così di dover ricaricare tutto.

### 4.4.2 SVILUPPO DEMO

L'ambiente di sviluppo usato è Visual Studio Code, un editor di testo multi-uso, open source e sviluppato da Microsoft come parte della sua offerta di prodotti gratuiti, ad oggi uno dei miglior ambienti di sviluppo web disponibili.
Essendo React Native sviluppato sulla base di React, il framework adotta tutte le metodologie web classiche, come l'uso di npm o yarn per gestire le dipendenze delle librerie Javascript nonché tutti i concetti più importanti come i componenti e la gestione dello stato.

Anche qui la parte database è stata affidata ad una libreria esterna chiamata **react-native-sqlite-storage**[21] che internamente implementa i metodi SQLite per ogni piattaforma. La configurazione della libreria per iOS e Android è abbastanza semplice ma più complessa di quella affrontata con Xamarin.Forms e Flutter a causa della registrazione del modulo

[21] https://github.com/andpor/react-native-sqlite-storage

manuale in entrambi i progetti nativi.

Per lavorare con React Native la prima cosa che abbiamo dovuto imparare è il significato di **Props** e di **State**.
Le Props contengono i valori dei parametri passati dai componenti più in alto nella gerarchia, questi valori non possono essere modificati una volta istanziati e l'unico modo per aggiornarli è ricreare il componente con i nuovi valori. Lo State invece è qualcosa che va istanziato alla creazione del componente con valori di default e ogni volta che vogliamo variare lo stato degli oggetti che lo compongono possiamo effettuare il cambiamento chiamando la funzione `setState()`.

L'uso di JSX è molto naturale per chi ha un minimo di dimestichezza con qualche framework web e probabilmente è l'implementazione più semplice che abbiamo provato. Lo rende molto simile ai framework di sviluppo web come Angular o VueJS e quindi non è stato difficile capire il funzionamento anche in React/React Native.

L'assegnazione e la creazione degli stili dei vari componenti è uguale a come si lavora con i CSS e in particolare a quello che avviene con il cosiddetto **FlexBox**[22]. FlexBox è una specifica non ancora ufficiale dei CSS (dovrebbe diventarlo a breve) che semplifica di molto il posizionamento degli elementi HTML, ad oggi tutti i maggiori browser la supportano già da tempo considerando che la bozza di proposta iniziale risale al 2009 e in React Native la cosa è stata sfruttata per permettere agli sviluppatori web di ritrovare un meccanismo di posizionamento molto simile a quello a cui sono abituati e devo dire che la cosa funziona bene.

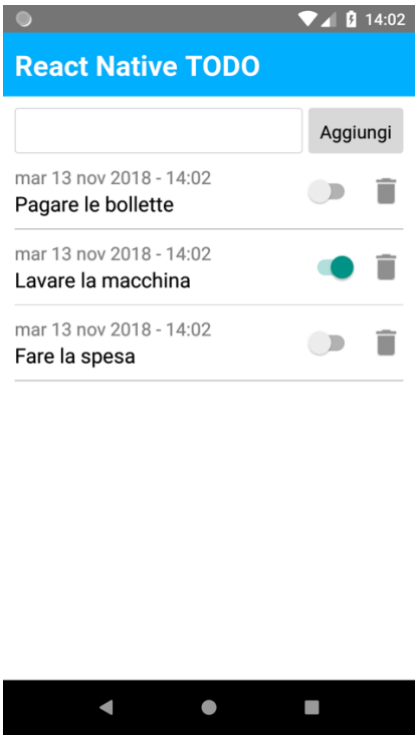
Durante i test sul simulatore di iOS è uscito fuori un bug molto fastidioso (nella versione 0.57.2), ovvero l'incapacità di resettare il valore del campo Input una volta che l'utente ha inserito il TODO nella lista. Indagando un po' si è scoperto che il problema[23] è già stato segnalato 6 mesi fa e nonostante questo non è ancora stato risolto. Per procedere si è deciso di implementare uno dei workaround proposti all'interno della segnalazione stessa.

Per compilare i progetti è necessario l'uso della riga di comando a differenza di Xamarin.Forms e Flutter dove è stato possibile fare tutto da interfaccia dei rispettivi ambienti di sviluppo.

Questi le schermate ottenute per Android e iOS al termine della demo in React Native:

Android iOS

[22] https://facebook.github.io/react-native/docs/flexbox
[23] https://github.com/facebook/react-native/issues/18272





Questa la relativa tabella di valutazione associata allo sviluppo con React Native:

| | **Valutazione** |
|---|---|
| Documentazione | **4/5** |
| Tempo di sviluppo | **circa 13 ore** |

La documentazione è molto buona, la velocità di sviluppo anche. Pur non conoscendo molto bene Javascript i concetti alla base del framework sono veramente facili da fare propri.
L'hot reloading dell'emulatore aiuta molto in termini di velocità di sviluppo ed è facile intuire il perché anche gli sviluppatori di Flutter abbiano voluto puntare molto su questa funzionalità.

## 4.5 FLUTTER

### 4.5.1 STORIA

Essendo il più giovane di tutti i framework multi-piattaforma qui descritti, Flutter non ha una grandissima storia alle sue spalle come gli altri.
Possiamo dire comunque che Flutter nasce con il nome originale di "Sky" e viene presentato per la prima volta al Dart Developer Summit[24] nel 2015 come progetto sperimentale da uno suoi fondatori di nome Eric Seidel.
I principali motivi per cui Google intraprese lo sviluppo di questo framework era perché non

[24] https://www.youtube.com/watch?v=PnIWl33YMwA

aveva ancora trovato una soluzione di sviluppo multi-piattaforma che risolvesse la maggior parte dei problemi che avevano riscontrato, ovvero: performance basse, lentezza nel ciclo di compilazione/debug e problemi con l'uso di componenti multi-piattaforma.

Nella conferenza del 2015 venne mostrata per la prima volta un'app Android scritta completamente in linguaggio Dart (sviluppato sempre da Google) che girava a 60 fotogrammi al secondo, ovvero con la stessa fluidità con cui girano le applicazioni native su iOS e Android.
Google ha poi cambiato nome del progetto in Flutter e l'ha presentato ufficialmente al mondo intero alla sua conferenza Google I/O nel maggio 2017[25] ottenendo un grandissimo successo e suscitando la curiosità di moltissimi esperti del settore.

Attualmente Flutter è ancora nella fase Beta ma il rilascio della versione 1.0.0 dovrebbe essere molto vicino tanto che alcune persone hanno già iniziato ad usarlo per i loro progetti.

### 4.5.2 BACKGROUND TECNICO

Le prestazioni raggiunte da Flutter, ovvero 60 o 120 fotogrammi al secondo (se lo schermo ha un refresh rate di 120Hz) sono frutto di un enorme lavoro architetturale. Il team dietro il suo sviluppo ha capito che se volevano ottenere queste prestazioni, dovevano assolutamente ripensare il modo in cui i framework multi-piattaforma lavorano dietro le quinte.
L'intuizione è arrivata quando hanno deciso di non usare i componenti nativi delle singole piattaforme, ma di emularli riscrivendoli da zero in Dart.

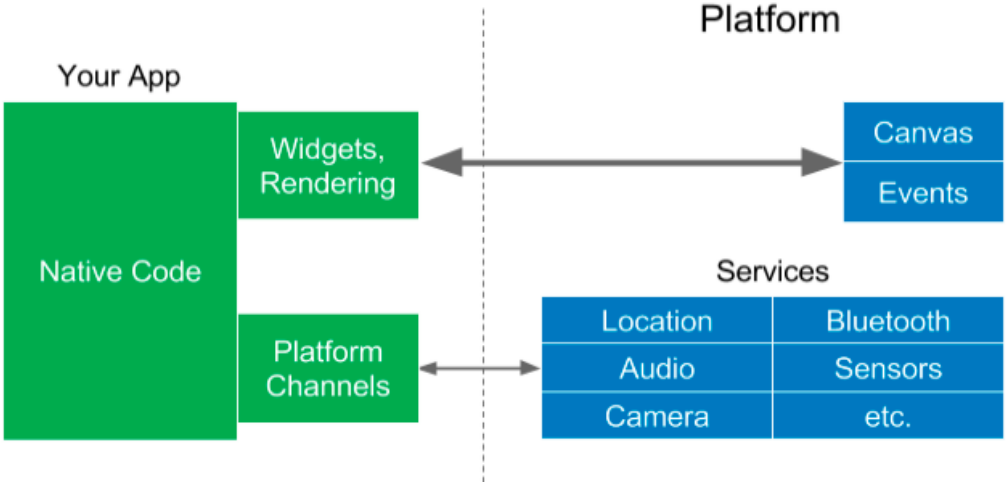



*Figura 15 - Funzionamento Flutter*

Come vediamo dalla figura 15, il codice dell'applicazione e tutti i componenti di interfaccia chiamati "widget", vengono gestiti dal framework interamente in Dart, senza dover essere tradotti nella loro controparte nativa come avviene per Xamarin.Forms e React Native.
Questo ha un grandissimo vantaggio in termini di prestazioni poiché permette di effettuare tutti i calcoli relativi al posizionamento delle viste e alle loro dimensioni in maniera molto più veloce e di lasciare alla parte di SDK della piattaforma solo il compito di mostrare e aggiornare la canvas.
Il metodo utilizzato è molto simile a ciò che avviene con i motori di gioco tipo **Unity**, un

[25] https://www.youtube.com/watch?v=w2TcYP8qiRI

framework di sviluppo usato per giochi 2D e 3D multipiattaforma, il quale sfrutta le librerie grafiche presenti nel sistema operativo come **OpenGL ES** o le **API Vulkan**, per disegnare le qualsiasi oggetto a schermo in maniera efficiente.

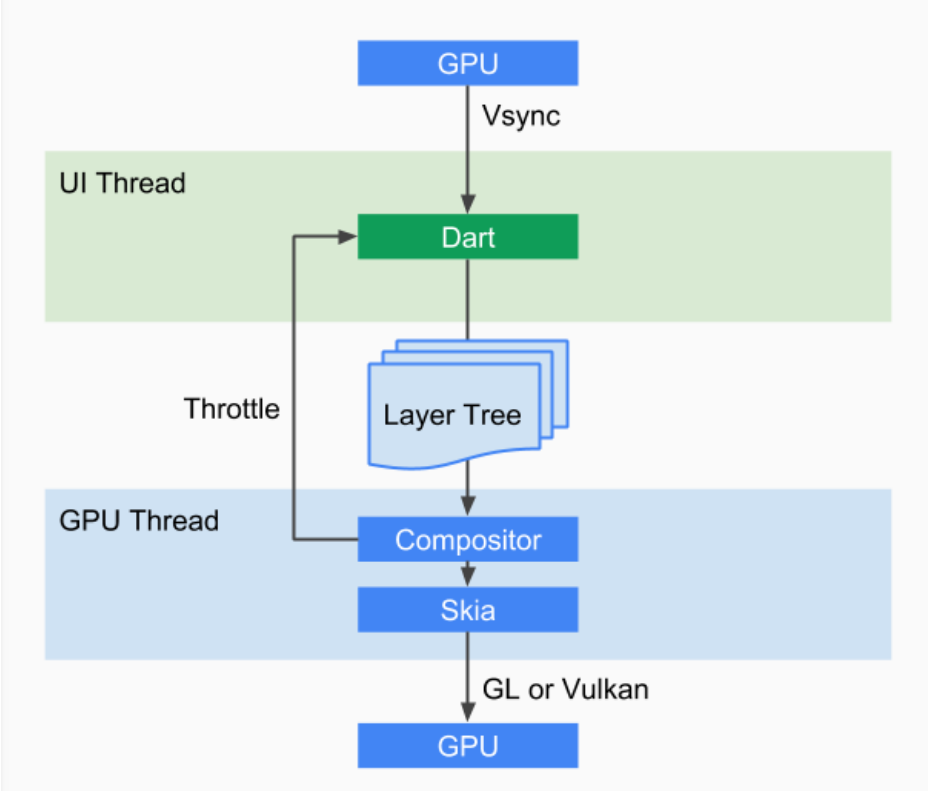



*Figura 16 - Gestione grafica di Flutter*

Come possiamo vedere in figura 16, Flutter sfrutta la GPU per disegnare gli elementi a schermo e questo lo fa in un'apposita Canvas chiamata "**Skia**" implementata in C++ che sfrutta le API grafiche disponibili sulla piattaforma in cui gira.

Un'altra particolarità di Flutter è il fatto di essere "widget" centrico, ovvero qualsiasi cosa viene mostrata a schermo è un widget i quali possono essere innestati uno dentro l'altro a formare delle gerarchie complesse e super personalizzabili. Il motivo dietro è che questo consente un altissimo livello di modularità, con piccole classi Dart specializzate a fare pochissime cose che se unite insieme possono dar vita a complessi meccanismi.

Questo è un pezzo di codice di un'interfaccia scritta in Flutter preso dalla documentazione ufficiale:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    Widget titleSection = Container(
      padding: const EdgeInsets.all(32.0),
      child: Row(
        children: [
          Expanded(
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                Container(
                  padding: const EdgeInsets.only(bottom: 8.0),
                  child: Text(
```

```
                    'Oeschinen Lake Campground',
                    style: TextStyle(
                      fontWeight: FontWeight.bold,
                    ),
                  ),
                ),
                Text(
                  'Kandersteg, Switzerland',
                  style: TextStyle(
                    color: Colors.grey[500],
                  ),
                ),
              ],
            ),
          ),
          Icon(
            Icons.star,
            color: Colors.red[500],
          ),
          Text('41'),
        ],
      ),
    );
  //...
}
```

Vediamo come i widget presenti nel codice come: **Text**, **Icon**, **Expanded** e gli altri, sono assemblati in maniera gerarchica per ottenere il risultato desiderato. Questo modo di creare l'interfaccia direttamente in codice (cosa possibile in tutti i framework ma solitamente sconsigliata) diventa qui l'unico modo di procedere. Con questo metodo gli sviluppatori di Flutter sono anche riusciti ad implementare l'**hot reload**, ovvero la possibilità di fare una modifica sul codice e di vederla riflessa nel dispositivo con l'applicazione in modalità debug in meno di un secondo, esattamente come avviene per React Native.
Un altro concetto ripreso del tutto da React, è quello riguardante la gestione dello stato, ovvero abbiamo a che fare con widget che possono essere sostanzialmente di due tipi, "**Stateless**" o "**Stateful**"[26] rispettivamente senza la capacità di aggiornare sé stessi una volta creati o con la capacità di aggiornarsi ogni volta che lo stato cambia. Esempi di StatelessWidget sono il componente **Text**, **RaisedButton** e **Icon** mentre alcuni StatefulWidget sono **Image**, **Form** e **CheckBox**.
L'ultimo concetto fondamentale sono le "**Keys**", ovvero quelle proprietà che permettono al motore di rendering di flutter di aggiornare i componenti di interfaccia tenendoli sincronizzati con lo stato a cui erano collegati prima di ricaricarsi. Le keys o chiavi, possono essere di vario tipo tra le quali: **GlobalKey**, **ValueKey**, **UniqueKey** e **ObjectKey**, ognuna di queste con specifici scopi per ottimizzare le performance.

### 4.5.3 SVILUPPO DEMO

L'ultima implementazione dell'app demo è quella in Flutter; iniziando a lavorare con questo framework ci siamo ritrovati molti dei concetti visti in React Native e rivisitati come già spiegato in precedenza nella parte di background tecnico.

Gli ambienti di sviluppo consigliati da Google per lo sviluppo di app in Flutter sono tre, ovvero Android Studio, Intellij (su cui è basato Android Studio) e Visual Studio Code. Per

[26] https://flutterdoc.com/stateful-or-stateless-widgets-42a132e529ed

praticità abbiamo deciso di procedere con Android Studio dato che molte delle cose da usare sono già integrate negli strumenti visuali e non dobbiamo quindi affidarci alla riga di comando per qualsiasi cosa.

Il database anche qui è stata affidato ad una libreria esterna di nome **sqflite**[27] che implementa in maniera nativa le chiamate ad SQLite di ogni piattaforma, nonostante sia abbastanza giovane non abbiamo riscontrato problemi nell'utilizzo. L'installazione di nuove dipendenze in Flutter è molto facile, basta aprire il file "pubspec.yaml", aggiungere il nome della libreria con la versione nella sezione "`dependencies:`" e premere il bottone in alto che Android Studio mette a disposizione con scritto "packages get".

Durante lo studio iniziale delle varie parti che compongono il framework, la cosa che ci ha rallentato maggiormente è stato il linguaggio di programmazione Dart. Venendo da Java, pensavamo di essere relativamente veloci ad imparare la nuova sintassi ma in realtà abbiamo trovato molte parole chiave nuove che in Java non sono presenti come async/await. Molti di questi costrutti sono presenti anche in Kotlin (sotto forme diverse ovviamente) perciò ci siamo chiesti perché Google che da un lato sta spingendo Kotlin per lo sviluppo nativo su Android, non abbia sfruttato la palla al balzo per spingere il linguaggio anche in Flutter e facilitare quindi la transizione da una piattaforma di sviluppo all'altra. Le risposte alla domanda "Perché proprio Dart e non un altro linguaggio più popolare?" le abbiamo trovate in un post su internet scritto da uno degli sviluppatori di Flutter[28] in cui spiegava che ci sono motivazioni tecniche dietro la scelta, una delle quali è la capacità di Dart di poter essere compilato sia AOT (ahead of time) che JIT (just in time), potendo quindi sfruttare la modalità AOT per la release dell'app poiché produce un bundle più piccolo e ottimizzato e quella JIT durante lo sviluppo per implementare l'**hot reload**.

Probabilmente dovrebbe bastare un mese di uso continuo con il linguaggio Dart per diventare produttivi e abituarsi alla sintassi, niente di trascendentale insomma ed è qualcosa che capita comunque con tutti i linguaggi di programmazione che usano pattern molto potenti già integrati.

Tornando all'app, l'interfaccia non ci ha dato nessun problema, la documentazione è molto efficace nello spiegare cosa fanno i singoli widget e come devono essere usati. L'unico problema che si può riscontrare inizialmente è capire che tutti i vari widget sono innestabili uno dentro l'altro, un concetto del tutto nuovo a qualsiasi framework multi-piattaforma provato in precedenza. Non è facile all'inizio capire che per aggiungere del padding ad un widget, cosa che normalmente si fa impostando una proprietà del componente, qua significa inglobare l'oggetto in un widget di nome **Padding** in questo modo:

[27] https://github.com/tekartik/sqflite
[28] https://hackernoon.com/why-flutter-uses-dart-dd635a054ebf

```
Padding(
  padding: EdgeInsets.only(top: 5),
  child: Text(
    '${widget.todo.text}',
    style: TextStyle(
        color: Colors.black,
        fontSize: 18.0
    ), // TextStyle
    textAlign: TextAlign.start,
  ), // Text
) // Padding
```

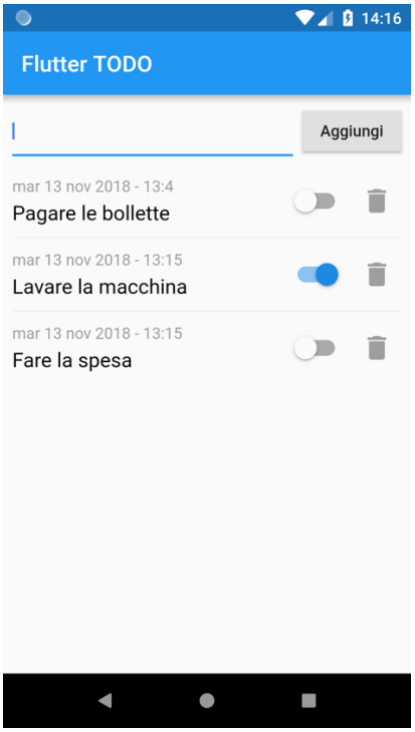
dove il valore del padding viene inserito nell'omonima proprietà del componente mentre l'elemento che deve subire la variazione di padding viene spostato nella proprietà "**child**".

Stessa cosa per le differenze tra **StatefulWidget** e **StatelessWidget** che sono abbastanza chiare fin da subito soprattutto paragonandole ai concetti di React Native quali i "props" e lo "state". I punti di similitudine tra i due framework aumentano considerando che lo stato si varia chiamando una funzione di nome `setState(…)` esattamente come in React Native.

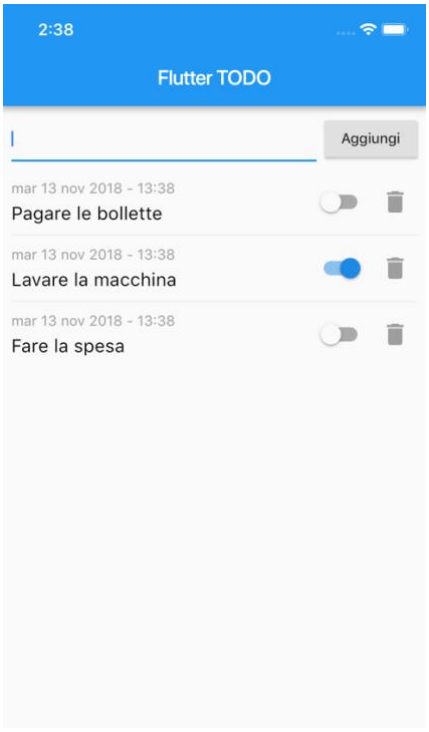
Nessun problema nel creare i bundle della demo tranne che per la modalità release per iOS, dove attualmente non c'è modo di compilare l'app senza avere un account sviluppatore Apple (dal costo di 99$ annuali).
Nonostante questa limitazione sia già oggetto di discussione nel repository GitHub di Flutter a questo link: https://github.com/flutter/flutter/issues/11754 non è ancora stato fatto nulla a riguardo.

Queste le schermate ottenute per Android e iOS al termine della demo in Flutter:

Android




iOS



Questa la relativa tabella di valutazione associata allo sviluppo:

| | **Valutazione** |
|---|---|
| Documentazione | **3/5** |
| Tempo di sviluppo | **circa 17 ore** |

La documentazione è buona anche se si è faticato a trovare informazioni concrete riguardo le chiavi (che sicuramente hanno un uso diverso da quello fatto nella demo) e in che modo si può far leggere lo stato ai widget in fondo alla gerarchia (da approfondire l'**InheritedWidget**). Tuttavia l'hot reload come su React Native da un aiuto non indifferente in termini di produttività.
Dando per scontato che la documentazione sia ancora in fase iniziale e quindi sicuramente migliorabile con il passare del tempo, l'unico vero scoglio incontrato è Dart, che soffre di poca diffusione e di conseguenza trovare aiuto su StackOverflow quando non si sa come procedere non è banale.

# 5. CONCLUSIONI

Sono stati analizzati due modi di sviluppare applicazioni, quello nativo e quello multi-piattaforma.
Era necessario rispondere alle tre domande poste nell'introduzione di questa tesi, ovvero:

1. Quando conviene sviluppare un'app in maniera nativa?
2. Quando conviene sviluppare un'app multi-piattaforma?
3. Se si intuisce che è possibile sviluppare un'app multi-piattaforma su che base va scelto il framework da usare?

Dopo aver provato tutti i vari framework, si è potuto intuire che la prima domanda è strettamente legata a queste tre domande, ovvero:

A. Che qualità e livello di affidabilità si devono ottenere?
B. È probabile che si dovrà aggiungere del personale al progetto?
C. È probabile che si dovranno usare librerie native?

Andando con ordine adesso possiamo rispondere ad ogni punto in maniera chiara:

A) Per le massime prestazioni e la massima flessibilità non c'è altro modo se non sviluppare applicazioni native. Le stesse Microsoft, Facebook e Google, che propongono i 3 framework multi-piattaforma qui studiati, sono le prime ad usarli pochissimo nei loro prodotti (almeno fino ad oggi).
Facebook ha dichiarato che solo piccolissime parti della sua app principale "Facebook" scaricabili dai vari store sono scritte in React Native. Su Microsoft non si trovano notizie a riguardo e sul sito di Flutter vengono portate ad esempio solo app di Google marginali come

AdSense.
Ovviamente non si deve dimenticare che parliamo di aziende con centinaia di milioni di utenti in tutto il mondo, è ovvio che puntino al meglio per le loro applicazioni principali ed è molto probabile che clienti così grossi abbiano al loro interno già i team per gestire le varie applicazioni.
Tornando a ragionare quindi in un'ottica più ristretta, ovvero con la mentalità di un'azienda informatica che fa app per i suoi clienti, l'affidabilità dell'applicazione è qualcosa che solamente il cliente può decidere e ovviamente nel caso voglia ottenere il massimo dell'affidabilità e delle prestazioni scegliere la strada dello sviluppo nativo è l'unica via percorribile.

Sviluppare multi-piattaforma in uno scenario critico o ad alto tasso di utenti è pericoloso, un esempio può essere portato da AirBnb, la nota azienda per la prenotazione di posti in cui alloggiare, che ha dichiarato[29] che dopo aver dato una lunga chance a React Native nella propria applicazione di punta, hanno deciso di tornare al nativo poiché si sono ritrovati sviluppatori con opinioni contrastanti e i contro avevano iniziato a superare i pro, ecco il motivo del passo indietro. La scelta di React Native in questo caso era dovuta al fatto che la maggior parte dell'infrastruttura di AirBnb è già su React e quindi hanno intravisto la possibilità di condividere i team di sviluppo tra web e app o quantomeno le conoscenze.
Nel lungo report scritto si fa cenno al fatto che la maggior parte dei problemi sono stati dati dalla lentezza degli aggiornamenti dell'SDK nel risolvere i bug, che è forse l'unico vero tallone di Achille di tutti i framework multi-piattaforma (anche se sembra che tutti stiano lentamente migliorando sotto questo punto di vista).

B) Se si pensa che il progetto da svolgere richiederà personale aggiuntivo, ora come ora conviene puntare sullo sviluppo nativo. Le statistiche sui linguaggi di programmazione non sono bloccanti come si è potuto vedere, praticamente la maggior parte dei framework multi-piattaforma usa linguaggi già molto diffusi ad eccezione di Flutter.
Quello che ci interessa è però il dato sui badge di StackOverflow, ci può dare un'indicazione sul fatto che attualmente ci sono molti più esperti di sviluppo nativo che sviluppatori multi-piattaforma, di conseguenza sarà molto più semplice trovare persone disposte a venire nella nostra azienda se il progetto verrà sviluppato in maniera nativa.

C) Le statistiche di GitHub indicano che le librerie native sono e probabilmente saranno sempre in numero maggiore rispetto a quelle multi-piattaforma, per il fatto che gli SDK nativi sono in circolazione ormai da moltissimi anni, ben prima che il primo framework multi-piattaforma (Xamarin.Forms nel 2014) fosse disponibile. Alcuni progetti possono richiedere un uso intenso di librerie native e quando questo accade è difficile non procedere con lo sviluppo nativo. Perché se è vero che è possibile usare queste librerie anche in progetti multi-piattaforma, è sempre richiesto del lavoro aggiuntivo per poterle utilizzare e delle volte questo lavoro può essere più grande di quanto si immagini, soprattutto con librerie molto complesse.

Con questo abbiamo risposto alla prima domanda, adesso è il momento di capire quando conviene utilizzare un framework multi-piattaforma rispondendo alla seconda domanda:

[29] https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c

- Quando conviene sviluppare un’app multi-piattaforma?

Conviene quando:

D. Si vogliono ottenere tempi di sviluppo migliori
E. Si vuole condividere il personale tra team di sviluppo Android e iOS

D) Se unifichiamo in un grafico i tempi di sviluppo impiegati per creare l’app prototipale sia Android che iOS, ci troviamo di fronte a questo:

| Piattaforma | Tempo di sviluppo totale |
|---|---|
| Android nativo | 15* + 15 = 30 |
| iOS nativo | |
| Xamarin.Forms | 15 |
| React Native | 13 |
| Flutter | 17 |

*Il tempo di sviluppo Android è stato allineato a quello iOS per non falsare la valutazione anche se in realtà è stato impiegato meno tempo.

Si evince che c’è un ottimo risparmio in termini di tempo e questo dimostra che anche partendo da zero, per creare dei prototipi che funzionino su entrambe le piattaforme non c’è niente di più veloce di usare un framework tra quelli proposti, che in questo caso mostrano un risparmio netto del 50% pur considerando la formazione effettuata per ognuno di essi. Ovviamente le cose cambiano al crescere della complessità dell’applicazione, è possibile credere che nello sviluppo di una applicazione di media complessità il risparmio di tempo reale non superi il 30-40% ma è comunque un’enormità e delle volte può fare la differenza tra guadagnare o perdere un cliente.

E) La condivisione degli sviluppatori significa avere personale intercambiabile tra i team di sviluppo Android o iOS. Questo è un grandissimo vantaggio per qualsiasi azienda che così è ragionevolmente sicura di avere sempre qualcuno in grado di lavorare allo stesso tempo su iOS e Android evitando colli di bottiglia causati dal sovraccarico di uno o dell’altro team.

Ora rispondiamo all’ultima domanda

- Se si intuisce che è possibile sviluppare un’app multi-piattaforma su che base va scelto il framework da usare?

Basandoci sulle risposte ottenute dal sondaggio fornito, sembra che i punti principali di scelta del framework multi-piattaforma siano i seguenti:

F. Conoscere già il linguaggio con cui si andrà a programmare nel framework
G. Avere già librerie o logiche scritte in precedenza e la necessità di riutilizzarle
H. Aver fatto delle ricerche e averne sentito parlare bene

Come emerso dal sondaggio fornito agli sviluppatori mobile multi-piattaforma, molti di loro hanno scelto il framework in base ai punti scritti sopra.
Questi sono i punti a cui gli sviluppatori danno più peso in assoluto e quindi quelli assolutamente più utili.

Da queste osservazioni otteniamo questo schema che può essere valido per la maggior parte dei team:

| Stato attuale | Framework suggerito |
|---|---|
| Conoscenza e/o librerie scritte in C# | Xamarin.Forms |
| Conoscenza e/o librerie scritte in Javascript/TypeScript | React Native |
| Conoscenza e/o librerie scritte in Java/Dart | Flutter |

Per color che invece si muovono sul fattore "ne ho sentito parlare bene" e che fanno quindi delle ricerche approfondite indipendentemente dalle loro conoscenze attuali possono tenere conto di questi fattori emersi dal sondaggio proposto:

1. Nonostante solo 3 delle 24 persone che hanno risposto al sondaggio usino Flutter e che questo framework sia ancora poco utilizzato a livello globale, molti hanno espresso il loro parere favorevole per quello che hanno visto o per quello che hanno letto in rete. Sta di fatto che è l'unico ad oggi a proporre un paradigma multi-piattaforma completamente diverso da tutti gli altri framework e che sulla carta assicura prestazioni comparabili a un motore per videogiochi.
2. Xamarin.Forms si conferma il framework più utilizzato, seguito a stretto giro da React Native.
3. Indipendentemente dal framework gli sviluppatori multi-piattaforma si trovano in generale bene e i problemi maggiori sono legati solo alla lentezza degli aggiornamenti per risolvere i bug (segnalati soprattutto da chi usa Xamarin.Forms e React Native).

Un ultimo accenno riguarda le differenze di dimensioni dei vari bundle creati in modalità "release" tra le versioni native e le versioni multi-piattaforma.
Inizialmente si era pensato di confrontare anche queste differenze ma alla fine ci si è resi conto che quest'ultime sono trascurabili in quasi tutti gli scenari odierni. L'unico momento in cui può essere interessante approfondire una valutazione del genere, è se ci viene chiesto di pubblicare l'app anche in stati che non possiedono connessioni Wi-Fi stabili o piani con traffico dati inclusi, per evitare che gli utenti non installino la nostra applicazione per colpa delle dimensioni eccessive.

# RINGRAZIAMENTI

Ringrazio il professore Mirko Ravaioli per avermi dato l'opportunità di sviluppare la tesi sull'argomento da me proposto con le sue indicazioni e lo ringrazio ancora di più per avermi trasmesso 5 anni fa durante il corso di "Mobile Web Design" la passione per lo sviluppo mobile che è poi diventata anche il mio lavoro quotidiano.

Un ringraziamento va alla mia famiglia, a mamma Nadia, a babbo Brunello e a mio fratello Paolo, che mi hanno sempre supportato in questi lunghi anni in attesa di raggiungere questo traguardo.

Un altro ringraziamento va alla mia ragazza Laura, che mi ha spronato nel terminare questo capitolo della mia vita e ha sopportato le lunghe notti di studio dopo il lavoro. Il raggiungimento di questo traguardo è in parte anche merito suo.