POLITECNICO DI TORINO

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

Tesi di Laurea Magistrale

# Studio e analisi del framework Xamarin per lo sviluppo di applicazioni multipiattaforma


**Relatori:**
prof. Maurizio Morisio
ing. Luca Ardito

**Candidato:**
Federico Cotto

Anno accademico 2016-2017

*Ai miei genitori*
*Alla mia famiglia tutta*
*Ai miei amici più cari*

# Sommario

Ad oggi, la sempre più ampia diffusione di smartphone e tablet ha fatto sì che il tempo trascorso online dagli utenti tramite le applicazioni *mobile* abbia superato il 50%. Questo, unito all'abbandono progressivo dell'utilizzo del computer desktop o laptop da parte di un numero sempre più crescente di persone, obbliga le aziende a rendere disponibili i propri servizi online anche tramite app, per poter mantenere la propria competitività sul mercato.

Un grande ostacolo, tuttavia, è la frammentazione del mercato dei sistemi operativi mobili, la quale crea problemi nel raggiungimento della totalità degli utenti per la fornitura dei propri servizi. Accade spesso, infatti, che le aziende si trovano a dover decidere per quali piattaforme sviluppare la propria applicazione, concentrandosi su quelle maggiormente diffuse e che possano avere un bacino d'utenza maggiore; tale decisione è tutt'altro che semplice, dato che il mercato degli smartphone, costantemente in movimento, con il suo andamento instabile può portare a cattivi investimenti da parte dell'azienda stessa.

Proprio in questo contesto trovano spazio i *framework* per lo sviluppo multipiattaforma di applicazioni mobili, che rappresentano un valido supporto per lo sviluppatore che si trova a voler progettare e sviluppare per le singole piattaforme senza però entrare nei dettagli di esse. L'utilità che ne deriva è sicuramente legata alla volontà di ridurre i costi di produzione e manutenzione dell'applicazione, ma anche per avere un modo più rapido per aprirsi in futuro verso altre piattaforme mobili oltre a quelle target a cui inizialmente ci si era dedicati.

Ci sono attualmente molti framework nati per questo scopo, ognuno classificabile in base all'approccio multipiattaforma utilizzato (approccio web, approccio ibrido, approccio interpretato e approccio cross-compilato). Dopo una prima analisi generale di ciascuno di essi, la soluzione più interessante e al momento più diffusa verrà approfondita, studiata e valutata tramite lo sviluppo di un caso di studio reale.

La tesi è strutturata nel modo seguente:

- nel primo capitolo viene introdotto l'argomento, descrivendo la tecnica per lo sviluppo di applicazioni mobili multipiattaforma e identificando i motivi

che stanno alla base di questo approccio;

- nel secondo capitolo viene analizzato lo stato dell'arte degli approcci di sviluppo multipiattaforma e dei relativi framework utilizzati;
- nel terzo capitolo viene svolto un approfondimento su *Xamarin*, il framework scelto come oggetto di questa tesi, descrivendone gli aspetti più rilevanti, il suo funzionamento interno e le modalità di utilizzo con le piattaforme e gli ambienti di sviluppo più noti;
- nel quarto capitolo viene presentato lo sviluppo del caso di studio, ovvero l'applicazione scelta da cui partire per valutare l'approccio multipiattaforma e il framework *Xamarin*;
- nel quinto capitolo vengono mostrati i risultati dei test e delle analisi di performance applicate al progetto realizzato, a confronto con l'utilizzo dell'approccio nativo;
- nel sesto capitolo vengono riportate le considerazioni finali sul lavoro svolto e, in generale, sul prodotto *Xamarin*.

# Indice

# Elenco delle figure

# Capitolo 1

# Introduzione

Negli ultimi quindici anni il nostro vivere quotidiano è migliorato notevolmente, conseguenza soprattutto di alcune invenzioni rivelatesi fondamentali per l'evoluzione della vita dell'uomo in molti suoi campi.

Una recente ricerca [1] sulle dieci invenzioni più rivoluzionarie fa emergere senza ombra di dubbio che è la tecnologia il settore che la fa da padrone, nel quale lo *smartphone*, in particolare, si colloca al primo posto. Se, fino a pochi anni fa, il termine era quasi privo di significato, oggi è abbastanza comune, per giovani e adulti, possedere almeno uno di questi piccoli dispositivi mobili "intelligenti" con i quali è possibile eseguire numerose operazioni, tramite le applicazioni software installate in essi, più comunemente note con l'abbreviativo *app*.

Dal momento in cui lo smartphone è stato introdotto nella nostra vita si è verificata una crescita esponenziale degli utenti che ne hanno fatto uso, volenti o nolenti. Uno studio effettuato nel 2014 [2] aveva già ipotizzato che entro il 2018 ci sarebbero stati più del 50% degli utenti ad utilizzare un tablet o uno smartphone per svolgere le proprie attività online, a scapito dei dispositivi più tradizionali quali PC desktop o laptop. Ciò, in effetti, è quanto sta accadendo. Senza avere la presunzione di affermare che i computer tradizionali sono scomparsi, si può dire che essi continuano ad esistere e restano vitali per le attività che richiedono tastiere e schermi di grandi dimensioni (programmazione, scrittura, reportistica e monitoraggio dati). Ma ora gran parte delle operazioni si svolge su dispositivi più piccoli (informazioni rapide, attività multimediali e social networking): tablet e smartphone hanno un paradigma di interazione fondamentalmente differente, basato principalmente sul tocco, con una tastiera che si apre solo quando necessario.

Un altro studio italiano condotto a inizio 2017 [3] ha confermato questo trend di crescita, rilevando che quasi l'80% del tempo totale trascorso online da parte degli utenti italiani viene fruito da un dispositivo mobile, come mostrato graficamente

in figura 1.1; di questo, in particolare, quasi l'89% è raggiunto tramite la comodità delle sue app, con un incremento sostanziale rispetto all'anno precedente.

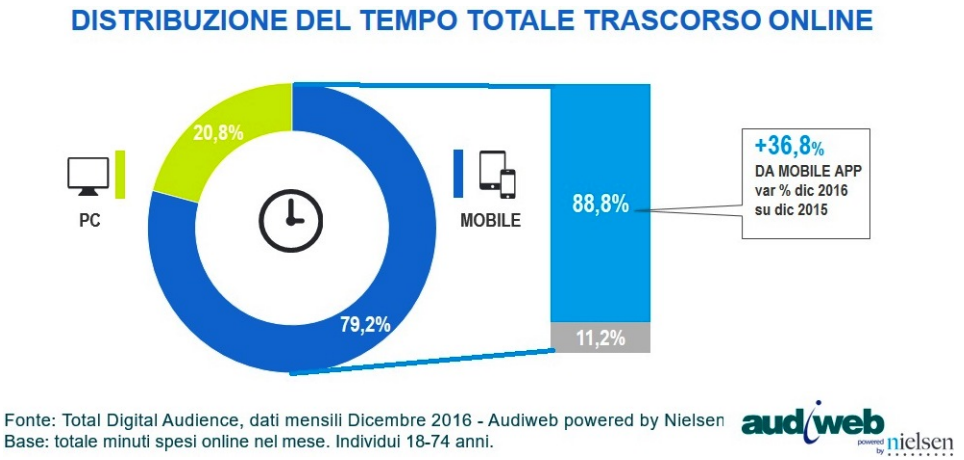



Figura 1.1. Tempo trascorso online dagli italiani a fine 2016

Le applicazioni mobili al loro esordio sono sembrate fin da subito un qualcosa di veramente innovativo, una grande novità che avrebbe rivoluzionato il modo d'utilizzo degli smartphone; per le aziende, commissionare la realizzazione di un'app significava principalmente contribuire a dare un'immagine innovativa al proprio marchio. Ma oggi che le app sono all'ordine del giorno e che quasi ogni azienda ne possiede una, l'effetto novità sta svanendo. E allora cosa può fare un brand per differenziarsi e far sì che i clienti utilizzino il più possibile la propria app? La tendenza che si sta affermando e che sembra essere la strategia vincente è quella di espandere le capacità della propria applicazione, fornendo agli utenti la possibilità di avere a disposizione il maggior numero di funzionalità in un'unica app. Le aziende che sapranno fare ciò prima dei propri concorrenti riusciranno senza dubbio a trarne un netto vantaggio e a mantenere la propria competitività sul mercato.

Analizzando la situazione attuale, salta subito all'occhio l'elevato numero di applicazioni attualmente esistenti, nate con funzioni diverse più o meno complesse e per tutte le esigenze. Gli utilizzatori di smartphone sono abituati ormai a cercare negli *store* della propria piattaforma l'app che li può aiutare a soddisfare al meglio i propri bisogni, prima ancora di effettuare la ricerca tramite un browser; il fatto di non essere presenti crea sicuramente una perdita significativa di clienti. Considerando, ad esempio, una azienda che si occupa di vendita online tramite un sito web, essa non può ignorare la crescita esponenziale dell'utilizzo del *mobile* descritta in questo stesso capitolo; in altre parole, essa non può non dotarsi anche di una app che permetta di gestire l'e-commerce in parallelo al suo sito web tradizionale.

Proprio da questo esempio e da questa considerazione è partita la scelta del caso di studio trattato, che verrà descritto in uno dei prossimi capitoli.

Appurata l'importanza per le aziende di svilupparsi anche sul fronte *mobile*, occorre però specificare che, realisticamente parlando, è praticamente impossibile raggiungere la totalità dei dispositivi oggi in commercio. Infatti attualmente esistono diversi sistemi operativi specifici per smartphone, ciascuno con caratteristiche e funzionalità più o meno diverse; il fatto di dover raggiungere tutte le piattaforme si traduce quindi in uno sforzo notevole nel dover realizzare una versione della propria applicazione per ogni piattaforma (e quindi per ogni sistema operativo esistente). D'altro canto però, la scelta di limitarsi ad un solo sistema operativo, magari il più conosciuto dal programmatore, ridurrebbe di gran lunga il bacino di utenza a cui si potrebbe benissimo aspirare: un'applicazione così popolare sull'iPhone, ad esempio, potrebbe essere ancora più popolare sui dispositivi Android e c'è solo un modo per scoprirlo... realizzarla!

| Period | Android | iOS | Windows Phone | Others |
|---|---|---|---|---|
| 2016Q1 | 83.4% | 15.4% | 0.8% | 0.4% |
| 2016Q2 | 87.6% | 11.7% | 0.4% | 0.3% |
| 2016Q3 | 86.8% | 12.5% | 0.3% | 0.4% |
| 2016Q4 | 81.4% | 18.2% | 0.2% | 0.2% |
| 2017Q1 | 85.0% | 14.7% | 0.1% | 0.1% |

Source: IDC, May 2017

Figura 1.2. Quote di mercato dei mobile OS tra il 2016 e il 2017

Da un'analisi periodica ed aggiornata ai primi mesi del 2017 [4] emerge che i principali sistemi operativi in termini di volumi di vendita ed utilizzo da parte degli utenti attualmente sono due, i quali, da soli, detengono una quota di mercato che supera il 99%: Android, sviluppato da *Google*, con l'85% e iOS, sviluppato da *Apple*, con il 14,7%. È curioso notare che un altro sistema operativo, il quale negli anni passati ha avuto il suo momento di gloria, ora sta subendo un declino che lo sta portando ad aumentare in maniera esponenziale il divario rispetto agli altri: si tratta di Windows Phone, sviluppato da *Microsoft*, azienda con un'importante storia nell'industria dei personal computer.

I dati qui esposti, come si evince dalla tabella in figura 1.2 e, a più alto livello, dal grafico in figura 1.3, possono essere soggetti a variazioni anche forti a seconda del periodo, motivo per cui spesso la scelta dei sistemi su cui focalizzarsi nello sviluppo di un'app può risultare la migliore nell'immediato, ma non nel medio-lungo periodo.

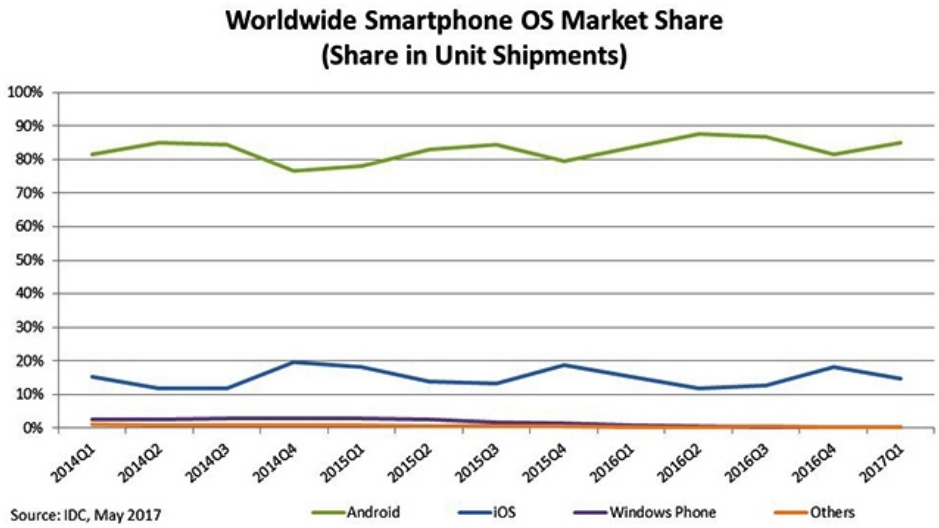



Figura 1.3. Quote di mercato dei mobile OS dal 2014 al 2017

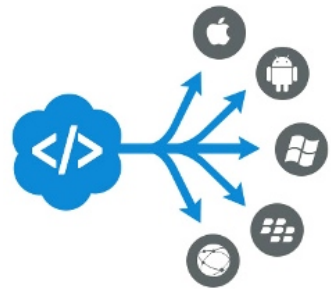
Il cosiddetto approccio di sviluppo *nativo*, ovvero specifico per ciascun sistema operativo, risulta pertanto inefficiente per reagire ad un mercato soggetto a così rapide variazioni, come la crescita d'uso di un sistema o il suo abbandono improvviso. Tale metodo di sviluppo inoltre prevede un costo non indifferente, dovendo ipotizzare il coinvolgimento di diverse persone con conoscenze tecniche, competenze e abilità differenti. E ancora, programmando in parallelo sulle diverse piattaforme è facile rendersi conto che tutto il futuro lavoro di manutenzione, revisione e miglioramento debba essere moltiplicato per un numero pari alle applicazioni native realizzate, al fine di mantenere per quanto possibile la stessa struttura.

La soluzione alternativa che può essere di aiuto per cercare di superare, almeno in parte, questi problemi è lo sviluppo cosiddetto *multipiattaforma* o *cross-platform*, che unifica e risolve problematiche alla base di ogni piattaforma. Il discorso verrà approfondito nei prossimi capitoli, con la descrizione dei diversi approcci possibili e dei relativi strumenti di sviluppo a disposizione.

# Capitolo 2

# Stato dell'arte dello sviluppo multipiattaforma

## 2.1 Generalità



In contrasto con la tecnica di sviluppo nativo, secondo la quale l'applicazione viene appositamente progettata per ogni singola piattaforma, lo sviluppo multipiattaforma comprende tutti quegli approcci e metodologie di sviluppo che cercano di raggiungere la filosofia *"Write once, run everywhere"*, ovvero *"Scrivi una volta, esegui ovunque"*, o, più realisticamente parlando, *"Scrivi una volta e fai eseguire al meglio possibile su più piattaforme"*. Si tratta di un'idea estremamente innovativa che sta diventando sempre più apprezzata nel mondo informatico ed ingegneristico, da sempre attento alla portabilità, alla qualità ed ai costi di manutenzione del software.

L'obiettivo di questa tecnica è quello di produrre applicazioni che possano funzionare in modo identico su piattaforme diverse, in modo da rendere disponibile un unico prodotto con un unico sforzo realizzativo alla maggior parte degli utenti indipendentemente dalla piattaforma utilizzata dal loro dispositivo. Il programmatore avrà la possibilità di scrivere il codice sorgente dell'applicazione una sola volta e tradurlo su più piattaforme, concentrandosi sulla logica di business e non sulla complessità degli SDK di ciascuna piattaforma.

L'approccio multipiattaforma è nato dalla necessità di realizzare app disponibili a tutti gli utenti, nonostante l'eterogeneità delle piattaforme, dei relativi ambienti

e linguaggi di sviluppo e dei dispositivi, senza necessità di disparate conoscenze né stanziamenti cospicui di risorse. Le possibilità di diffusione di un'app, infatti, sono tanto maggiori quanto più questa è trasversale e versatile, riuscendo a soddisfare alcune caratteristiche fondamentali per l'approccio di sviluppo in questione, tra le quali si evidenziano le seguenti:

- **Portabilità**: la capacità di funzionare su più piattaforme;
- **Produttività**: l'efficienza del processo di produzione in termini di velocità di consegna, grazie all'utilizzo di elementi e strumenti comuni tra le piattaforme;
- **Semplicità di manutenzione**: la facilità nell'apporre eventuali modifiche correttive e di aggiunta di funzionalità, le quali saranno effettuate in un'unica soluzione.

Sulla base di quanto detto, occorre in ogni caso pensare a quanto e in che modo la scelta di questa tecnica di sviluppo influisca su altre caratteristiche altrettanto importanti quali efficienza, affidabilità e semplicità d'uso. Questo dipende sicuramente in buona parte dall'approccio multipiattaforma utilizzato e dal suo essere in grado di affrontare e superare le differenze sostanziali tra i sistemi operativi delle diverse piattaforme.

## 2.2 Differenze principali tra le piattaforme mobili

Lo sviluppo multipiattaforma, nonostante sia di notevole vantaggio poiché ci consente di scrivere il codice una sola volta e tradurlo automaticamente per un uso su più sistemi operativi, deve tener conto delle differenze fondamentali tra le diverse piattaforme mobili, che potrebbero far sorgere eventuali problemi in fase di sviluppo. Alcune di esse, analizzate più in dettaglio nel seguito di questo paragrafo, le possiamo individuare nei seguenti ambiti:

- Linguaggio di programmazione;
- Interfacce di programmazione;
- Ambiente di sviluppo e sistema operativo;
- Interfaccia utente e *user experience*;
- Supporto al multitasking;
- Consumo della batteria e prestazioni;
- Pubblicazione dell'applicazione.

### 2.2.1 Linguaggio di programmazione

Considerando le principali piattaforme mobili attuali, possiamo notare che ognuna di esse rilascia i propri SDK (Software Development Kit) che consentono di sviluppare applicazioni basandosi soltanto sullo sviluppo nativo per ogni piattaforma e, quindi, supportando soltanto il linguaggio di programmazione proprio del sistema operativo e delle librerie proprietarie. Realizzare un'app nativa con queste premesse vorrebbe dire impiegare tre diversi team di sviluppatori, ciascuno specializzato su un particolare linguaggio e API.

La seguente tabella fornisce una panoramica delle principali piattaforme e dei rispettivi linguaggi di programmazione da esse supportati.

| **Piattaforma** | **Linguaggi supportati** |
|---|---|
| iOS | Objective-C, Swift |
| Android | Java, Kotlin |
| Windows Phone | C#, C++, Visual Basic |

Pensando ora allo sviluppo multipiattaforma, è chiaro che, essendo il suo obiettivo primario la condivisione del codice, un approccio di questo tipo deve essere in grado di andare oltre ai linguaggi di programmazione di base, ovvero deve indirizzarsi sempre di più verso un unico linguaggio comune riducendo l'uso dei linguaggi nativi.

In tal senso, inoltre, già molte piattaforme forniscono anche un insieme di API (Application Programming Interface) multipiattaforma, che consentono di accedere a determinate funzioni del dispositivo quali la geolocalizzazione, elementi multimediali, e funzionalità di rete.

### 2.2.2 Interfacce di programmazione

Le tre principali piattaforme analizzate sono basate su diversi sistemi operativi con diverse API. In molti casi però, esse implementano tipi simili di oggetti di interfaccia, la cui differenza sta solo nel nome con il quale si fa riferimento.

Ad esempio, tutte e tre le piattaforme hanno qualcosa che consente all'utente di commutare un valore booleano:

- su iOS è una "view" chiamata *UISwitch*;
- su Android è un "widget" chiamato *Switch*;
- su Windows Phone è un "control" chiamato *ToggleSwitch*.

Tali differenze, che in generale vanno ben oltre i nomi delle interfacce stesse, devono necessariamente essere tenute in considerazione e ridotte, nei limiti del possibile, ai fini del passaggio ad uno sviluppo multipiattaforma.

### 2.2.3 Ambiente di sviluppo e sistema operativo

Realizzare un'applicazione nativa per una specifica piattaforma presuppone di installare sul proprio PC l'IDE (Integrated Development Environment) e l'SDK specifici della piattaforma, i quali spesso obbligano ad utilizzare uno specifico sistema operativo, l'unico in grado di supportarli.

La seguente tabella mostra per le principali piattaforme gli ambienti di sviluppo da esse supportati. Essa mette in risalto come, per esempio, uno sviluppatore che voglia creare applicazioni iOS debba necessariamente utilizzare un computer Mac, mentre per applicazioni Android lo sviluppo può avvenire indistintamente su computer Windows, Mac o Linux.

| **Piattaforma** | **IDE** |
|---|---|
| iOS | Xcode (su Mac) |
| Android | Android Studio, Eclipse |
| Windows Phone | Visual Studio (su Windows) |

Eliminare questi obblighi, prediligendo dei *tools* di sviluppo comuni e lasciando al massimo agli specifici sistemi le operazioni di compilazione ed esecuzione, è un altro obiettivo da porsi se si vuole sviluppare con un approccio multipiattaforma.

### 2.2.4 Interfaccia utente e user experience

Le diverse interfacce utente e, conseguentemente, la differente *user experience* nell'uso delle piattaforme mobili può essere considerato uno dei grandi ostacoli allo sviluppo multipiattaforma. Con tale approccio, infatti, si potrebbero perdere dettagli di interfaccia più o meno importanti, perché magari disponibili solo su un'unica piattaforma e difficilmente portabili su altre. Ecco che anche la realizzazione del layout partendo da un linguaggio comune, un'operazione che a prima vista potrebbe sembrare facile, diventa spesso complicata, specie se si ha la necessità di implementare funzioni complesse da far funzionare sulle diverse piattaforme. Di conseguenza, potrebbe succedere di dover sacrificare o riadattare alcune parti di layout, altrimenti non implementabili, condizionando in questo modo anche l'esperienza utente.

La seguente tabella mostra i linguaggi di *markup* supportati ed utilizzati dalle principali piattaforme.

| **Piattaforma** | **Linguaggi supportati** |
|---|---|
| iOS | XIB, Storyboard |
| Android | XML |
| Windows Phone | XAML |

Un aspetto importante è la filosofia di progettazione delle piattaforme mobili: mentre molte di esse cercano un design realistico, e quindi puntano a migliorare la *user experience* partendo dall'interfaccia utente, altre tendono a creare degli stili ben precisi, e quindi, basandosi su un'interfaccia utente più o meno unica, cercano di migliorare l'esperienza utente.

In passato, ad esempio, la Apple tendeva ad utilizzare uno stile in cui le icone, i pulsanti, ecc. erano più complessi, ma oggi l'azienda preferisce un design più semplice e minimalista. Google, invece, agisce contrariamente: con la loro nuova concezione, denominata *material design*, si sono creati uno stile tutto loro, delicato e naturale. Da ciò ne deriva che, per lo sviluppo nativo di un'applicazione, bisogna studiare le linee guida di entrambi gli approcci.

Lo sviluppo multipiattaforma, dunque, semplifica tutto questo, anche se nella maggior parte delle situazioni è di fondamentale importanza ottenere una *user experience* nativa, per garantire la facilità di utilizzo da parte di utenti abituati con il dispositivo ed il sistema operativo da loro scelti.

### 2.2.5 Supporto al multitasking

Il multitasking consente di mantenere attivi servizi in background e di eseguire contemporaneamente diverse applicazioni. È realizzato in modo differente tra i sistemi operativi; per esempio, su alcune piattaforme non è possibile chiudere del tutto le applicazioni che rimangono attive in modalità background, le quali vengono chiuse definitivamente dal sistema operativo soltanto quando il dispositivo esaurisce la memoria. Su altre piattaforme, invece, vi è una selezione limitata di attività in background che possono continuare a funzionare dopo la chiusura dell'applicazione.

Per questi motivi, le soluzioni di sviluppo multipiattaforma dovrebbero valutare strategie e fornire strumenti che consentano l'utilizzo della modalità background e, conseguentemente, che siano in grado di adattarsi al meglio alla piattaforma sottostante.

### 2.2.6 Consumo della batteria e prestazioni

Il consumo della batteria è strettamente legato al concetto di multitasking. Secondo la *legge di Moore* il numero di transistor all'interno dei microprocessori raddoppia ogni 24 mesi; di conseguenza anche le prestazioni del microprocessore raddoppiano. Tuttavia, la capacità della batteria viene raddoppiata solo ogni sette anni.

In un sistema multipiattaforma, più si è vicini ad essa, più è possibile controllare il consumo della batteria e le prestazioni dell'applicazione. In generale, più velocemente si vuole eseguire l'applicazione, maggiori prestazioni sono necessarie al dispositivo e, quindi, maggiore sarà il consumo della batteria.

### 2.2.7 Pubblicazione dell'applicazione

Per essere fruibili, le applicazioni vengono distribuite attraverso uno *store* ufficiale, diverso per ogni piattaforma, dal quale gli utenti possono scaricarle. Questa diversità potrebbe comportare differenze nei meccanismi di acquisto e costi diversi a seconda della piattaforma per la quale è stata sviluppata l'applicazione.

Inoltre, anche i tempi di pubblicazione sono diversi: gli sviluppatori di Android, per esempio, pensano che il proprio *store* sia più flessibile perché gli utenti possono scaricare le app entro poche ore, mentre su quello della Apple è possibile scaricare versioni nuove di un prodotto attendendo addirittura parecchie settimane. Questo però fa pensare che probabilmente l'app distribuita da Apple sia priva di bug in quanto sottoposta ad un maggiore controllo, contrariamente alla flessibilità e alla facilità con cui l'app Android viene distribuita.

Al fine di pubblicare la propria app su uno specifico *store*, poi, occorre generare il suo archivio binario appropriato (file eseguibile), anch'esso diverso a seconda della piattaforma.

La tabella seguente mostra per le principali piattaforme lo *store* di riferimento e il formato del file eseguibile: ancora una volta si evidenziano evidenti differenze.

| **Piattaforma** | **Store** | **File eseguibile** |
|---|---|---|
| iOS | Apple App Store | .ipa, .app |
| Android | Google Play Store | .apk |
| Windows Phone | Microsoft Windows Phone Store | .xap, .appx |

Volendo utilizzare strumenti di sviluppo multipiattaforma, occorre che questi siano in grado di provvedere ad una soluzione per facilitare, nella miglior maniera possibile, l'accesso agli *store*, l'approvazione dell'app e la sua pubblicazione.

## 2.3 Principali approcci multipiattaforma

Esistono diversi approcci e strategie di sviluppo che tentano di superare i problemi descritti nel paragrafo precedente (eterogeneità delle piattaforme, dei relativi ambienti e linguaggi di sviluppo e dei dispositivi), ciascuno con le proprie caratteristiche, ma con il comune obiettivo di sviluppare applicazioni portabili su più piattaforme diverse.

Nel seguito di questo paragrafo verranno analizzate alcune di queste strategie multipiattaforma, ed in particolare le seguenti:

- Approccio web;
- Approccio ibrido;

- Approccio interpretato;
- Approccio cross-compilato.

### 2.3.1 Approccio web

Lo sviluppo mediante approccio web permette di generare applicazioni accessibili senza il bisogno di essere installate sul dispositivo; la loro esecuzione avviene, infatti, all'interno del comune browser presente in ogni dispositivo, quasi come un sito web, dal momento che esse si trovano fisicamente su server che le rendono accessibili tramite rete Internet attraverso un URL. A tal fine, è necessario quindi dotarsi di uno spazio web a costi più o meno elevati in base alla complessità dell'applicazione ed al numero di utilizzatori.

Tali applicazioni presentano un'architettura client-server nella quale il back-end, corrispondente alla logica computazionale, è implementato su server remoto ed il front-end, coincidente con l'interfaccia grafica, è implementato rigorosamente utilizzando linguaggi di programmazione per il web interpretabili dal browser, quali HTML, CSS e JavaScript. Ed è proprio l'utilizzo del browser, insieme con la logica server-driven, a fare in modo che le applicazioni realizzate con questo approccio siano completamente indipendenti dalla piattaforma.

Come già detto, quindi, non ci sarà alcun bisogno per gli utenti di installare nulla nel proprio device, né di effettuare specifici aggiornamenti; le applicazioni verranno aggiornate, se necessario, in maniera immediata e totalmente automatica tutte le volte che vi si accederà attraverso il browser.

Con questo approccio si ha un notevole risparmio di risorse: la maggior parte delle operazioni di calcolo non viene svolta localmente, ma su un apposito server web, ottenendo in questo modo un potenziale aumento di capacità di calcolo rispetto a quella fornita dal dispositivo.

Le app web non riescono però a soddisfare diverse richieste che nella maggior parte dei casi sono indispensabili per creare un prodotto di qualità. Ad esempio, un'importante limitazione di questo approccio è la difficoltà, che spesso diventa impossibilità, di accesso alle risorse e all'hardware del dispositivo, non essendo mai diretto, ma sempre tramite browser.

Inoltre, un altro aspetto da considerare è l'impossibilità di essere presenti negli *store* delle varie piattaforme, rendendo le applicazioni difficili da recuperare, dato che l'utente è abituato a cercare proprio sullo *store* ufficiale.

Ancora, le performance dell'applicazione potrebbero essere condizionate ed appesantite dall'utilizzo del browser e del collegamento a Internet, due strumenti indispensabili al funzionamento, in quanto potrebbero sorgere problemi in caso di assenza di rete o di servizio.

Esistono attualmente parecchi strumenti online, gratuiti o a pagamento, che offrono la possibilità di realizzare applicazioni con questo approccio in modo semplice e senza scrivere alcuna riga di codice; **AppsBuilder**, **iBuildApp**, **ShoutEm**, **GoodBarber** e **Mobincube** sono solo alcuni esempi. Il modo di utilizzo di tali servizi, molto simile, è reso semplice ed immediato dall'ambiente grafico offerto: si parte con la scelta del tema principale e poi si prosegue con l'aggiunta delle *view* e dei relativi contenuti.

È facile intuire però che la facilità e l'automatismo con cui si può creare un'applicazione con questi strumenti porti ad un prodotto finale non particolarmente soddisfacente su diversi aspetti. Innanzitutto, il fatto di poter scegliere solo tra uno dei *template* proposti rende la struttura dell'app rigida e la sua grafica minimale. Inoltre non è possibile accedere a nessuna delle API del sistema operativo nativo e, non potendo scrivere codice, la personalizzazione è davvero riduttiva. Questi servizi sono pertanto consigliati ad utenti inesperti che desiderano realizzare un'applicazione mobile basilare installabile su tutte le piattaforme in poco tempo e senza avere alcuna conoscenza di programmazione.

### 2.3.2 Approccio ibrido

L'approccio ibrido si colloca a metà tra quello nativo e quello web, come messo a confronto dalla figura 2.1. L'idea alla base di questa metodologia, infatti, è quella di combinare tecnologie di sviluppo native con soluzioni basate sul web, così da racchiudere in sé alcune caratteristiche delle app native ed altre delle app web e sfruttare al meglio i rispettivi punti di forza e potenzialità. L'architettura ibrida prevede che l'app vera e propria, sviluppata utilizzando tecnologie web, venga eseguita all'interno di un *wrapper* che funge da involucro, sviluppato in un linguaggio nativo.

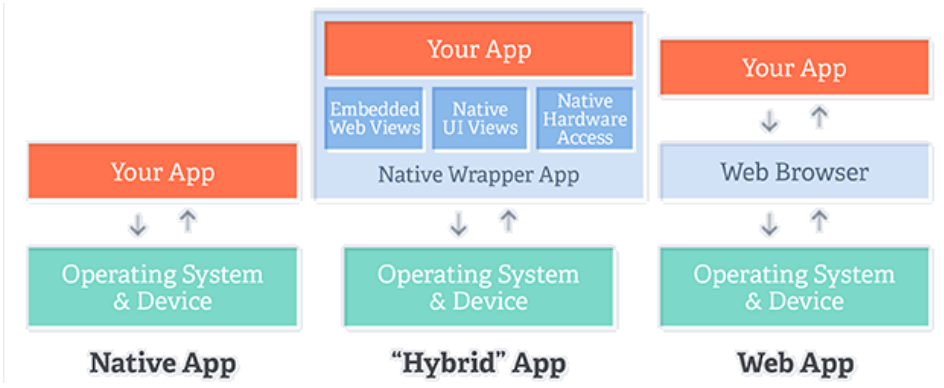



Figura 2.1. Stack delle tecnologie mobili native, web e ibride a confronto

L'interfaccia utente è, quindi, mostrata attraverso un browser web, dato che il "core" dell'applicazione è sviluppato prevalentemente in HTML, CSS e JavaScript. L'integrazione con le risorse e con le caratteristiche del dispositivo sul quale viene installata l'app, invece, è facilitata dal layer astratto ottenuto grazie al wrapper nativo. Il wrapper deve essere riscritto per ogni sistema operativo per il quale si vuole distribuire l'applicazione, e può essere realizzato direttamente dagli sviluppatori oppure appoggiandosi a framework multipiattaforma esistenti. In genere, lo sviluppatore può sfruttare le funzionalità del dispositivo utilizzando API JavaScript, che vengono mappate in codice nativo dal wrapper.

La disponibilità o meno di una funzionalità nativa hardware e software del dispositivo (GPS, fotocamera, microfono, contatti, sistema notifiche, ecc.), l'utilizzo del motore di rendering dei linguaggi del web fornito dal framework o di quello tipicamente integrato nel browser della piattaforma, e la disponibilità o meno di componenti della user interface nativi dipendono fortemente dalle scelte degli sviluppatori e dal framework adottato.

Nel caso in cui non sia possibile utilizzare UI native, il rendering della parte grafica è affidato ad una *WebView*, un componente incorporato nel browser inserito nella UI dell'app. Essa nasce con l'idea di supportare le applicazioni native nell'esecuzione e visualizzazione di contenuti web ottenuti da un server remoto, senza ricorrere alla barra degli indirizzi o ad altri tipici elementi dei browser, ma fornendo la possibilità di visualizzare le pagine appena l'applicazione ibrida viene lanciata. Questo approccio offre la possibilità di creare soluzioni davvero uniche e flessibili, come ad esempio realizzare UI affiancando componenti nativi a WebView, simulando il look nativo con gradevoli risultati.

Nel caso in cui, invece, il framework permetta di utilizzare la UI nativa, non sarà necessario ricorrere all'utilizzo delle WebView, ma si potranno sfruttare le specifiche API di ciascuna piattaforma, garantendo un'esperienza di utilizzo totalmente nativa.

In ogni caso, comunque, all'atto della compilazione il framework distribuirà l'applicazione sulle varie piattaforme ed otterrà i corrispondenti file di installazione specifici per il dispositivo. Proprio per questo, diversamente da quel che accade alle applicazioni web, quelle ibride possono essere distribuite sullo *store* e quindi più facilmente accessibili, reperibili e scaricabili.

Inoltre, in alcuni casi di app vetrina o app che inglobano tutti i dati al loro interno, non è indispensabile avere una connessione Internet per eseguire un'app ibrida, al contrario delle app web. Infatti esse possono funzionare in maniera autonoma anche offline, grazie al caching realizzato dalle tecnologie di storage fornite da HTML5, che rendono possibile l'utilizzo della WebView anche nel caso in cui l'utente non sia connesso. In generale però, essendo molto più frequenti le applicazioni che si appoggiano ad un server per svolgere determinate funzioni, nella maggior parte dei casi le app ibride avranno funzionalità limitate se prive di

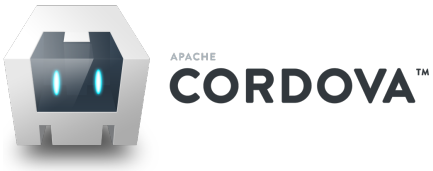
connessione di rete, anche se potranno comunque avviarsi, al contrario delle app web che non partirebbero nemmeno.

Analizzando ora gli svantaggi spicca il fatto che le performance ottenute con questo approccio sono inferiori rispetto a quelle delle applicazioni native, in quanto l'accesso alle risorse del dispositivo comporta un overhead computazionale perché non è diretto, ma sempre interfacciato da API JavaScript gestite da un motore di rendering dei linguaggi del web. Poi, l'interfacciamento con le risorse dei dispositivi dipende dal framework utilizzato; pertanto, spesso alcune funzionalità potrebbero non essere supportate per alcune piattaforme o esserlo solo parzialmente. Inoltre, la UI è meno reattiva poiché la velocità di rendering della WebView non è ancora paragonabile a quella delle UI delle applicazioni native. E ancora, il codice JavaScript può essere interpretato in modo differente da ogni dispositivo. Infine, lo stile dell'interfaccia grafica generalmente non è paragonabile a quello delle applicazioni native, nonostante venga fatto comunque uno riutilizzo di base su diverse piattaforme.

Tra i framework esistenti che seguono questo approccio troviamo **Cordova** e **PhoneGap**, descritti brevemente nel seguito di questo paragrafo.

### Cordova

*Apache Cordova* è un framework open source per lo sviluppo di applicazioni mobili ibride tramite l'utilizzo delle WebView native. Esso consente di utilizzare le tecnologie web standard, quali HTML, CSS e JavaScript, per lo sviluppo multipiattaforma. Le applicazioni vengono eseguite in wrappers destinati a ciascuna piattaforma e si basano su API standard per accedere alle funzionalità native di ciascun dispositivo.




*Cordova* dispone di una serie di *plugin*, ovvero del codice che fornisce un'interfaccia JavaScript verso un componente nativo; per ognuno di essi è disponibile una documentazione ricca di indicazioni ed esempi frutto del lavoro della community appositamente creata.

Per quanto concerne l'interfaccia grafica, è possibile creare dei look nativi, o quasi, tramite l'utilizzo di framework grafici come *Ionic*, *Famo.us*, *Framework7* e *OnsenUI* che realizzano per tutti gli elementi grafici nativi la loro versione web.

### PhoneGap

*Adobe PhoneGap* è un framework multipiattaforma che consente di sviluppare delle applicazioni mobili native attraverso l'utilizzo di tecnologie web moderne, quali HTML, CSS e JavaScript.

Esso non è altro che una implementazione di *Apache Cordova*, anche se offre più funzionalità. Una delle estensioni più interessanti che *PhoneGap* offre in esclusiva è un servizio per la compilazione di app native, grazie al quale è possibile effettuare il loro collaudo localmente dopo averle sviluppate senza preoccuparsi di configurare correttamente questo processo per ogni piattaforma a cui ci si vuole rivolgere.


Non disponendo di un proprio IDE, *PhoneGap* viene utilizzato all'interno degli ambienti di sviluppo nativi dei sistemi operativi mobili, con il conseguente vantaggio di potersi integrare facilmente in piattaforme differenti.

Tramite una delle API che *PhoneGap* possiede, si può accedere alle funzionalità native del sistema operativo; in JavaScript è scritta invece la logica dell'applicazione. Oltre a JavaScript stesso, con HTML e CSS viene definita anche l'interfaccia grafica dell'applicazione, mostrata all'utente grazie al browser web del dispositivo.

Il prodotto finale è un archivio binario dell'app che può essere distribuito sulle diverse piattaforme, proprio come se l'applicazione fosse stata implementata con linguaggi nativi.

### 2.3.3 Approccio interpretato

Con questo tipo di approccio, molto vicino a quello ibrido visto precedentemente, prima viene caricato il codice dell'applicazione sul device e solo successivamente viene interpretato, eseguendo il codice sorgente a *runtime* ed offrendo così la possibilità di riutilizzare la logica applicativa su diverse piattaforme e quindi di supportare uno sviluppo di tipo multipiattaforma.

Tramite l'utilizzo di un framework apposito è possibile scendere di livello e accedere all'hardware e al software specifico della piattaforma. L'applicazione interpretata, inoltre, accede alle API native e agli elementi dell'interfaccia grafica specifici della piattaforma attraverso un layer astratto, ottenendo un'interfaccia utente pari a quella della corrispondente versione nativa.

Gli aspetti negativi si hanno nelle performance, che potrebbero essere drasticamente peggiori per l'interpretazione del codice a runtime; inoltre, il livello di astrazione del framework scelto influenza in modo diverso lo sviluppo e il riutilizzo della UI.

Tra le piattaforme che seguono questa strada troviamo **Titanium** e **Rhodes**, descritte brevemente nel seguito di questo paragrafo.

### Titanium

*Appcelerator Titanium* è un framework open source per scrivere applicazioni mobili basato esclusivamente su un SDK JavaScript. Come riportato nella documentazione, il suo obiettivo è quello di fornire agli sviluppatori uno strumento che elimini quanto più è possibile quelli che sono i compromessi dello sviluppo ibrido.

Il codice sorgente è scritto totalmente in JavaScript; anche con *Titanium*, poi, è consentito accedere alle funzionalità del sistema sottostante attraverso apposite API del framework, così come è possibile avvicinare l'aspetto dell'interfaccia grafica e le sue prestazioni a quelli della corrispondente versione nativa. Durante la fase di compilazione, *Titanium* combina il codice sorgente con un interprete JavaScript e altri contenuti statici inglobandoli in un package per poter essere distribuito, mentre a runtime l'interprete processa codice JavaScript, facendolo interagire con l'ambiente nativo attraverso oggetti proxy che esistono su tutti gli ambienti supportati.


*Titanium* è facile da usare, con moduli già pronti cross-platform o platform-specific utilizzabili per soddisfare ogni esigenza. Inoltre, tramite la possibilità di utilizzare servizi cloud, permette di gestire dati ed utenti tramite un back-end già pronto e funzionale.

### Rhodes

*Rhodes* è un ambiente di sviluppo open source basato sul linguaggio *Ruby*. Anche in questo caso l'ambiente è di facile comprensione per chi è già in possesso di competenze per lo sviluppo web, tramite le quali è possibile realizzare applicazioni mobili interpretate.


*Rhodes* si basa sul pattern MVC (Model-View-Controller), dove le *view* sono un insieme di file scritti con i linguaggi del web (HTML, CSS e JavaScript) eseguiti sul dispositivo attraverso un piccolo web server locale presente in esso, mentre il *model* e il *controller* sono costituiti entrambi da un insieme di file di script Ruby con estensione *.rb*.

Anche le app prodotte con *Rhodes* possono essere integrate con servizi cloud e, inoltre, possono fare uso di database locali come SQLite.

## 2.3.4 Approccio cross-compilato

Le applicazioni realizzate con un approccio cross-compilato vengono scritte utilizzando un linguaggio di programmazione comune; sarà poi un cross-compilatore

ad essere responsabile della traduzione dei file sorgente comuni in file binari nativi. Dato che l'operazione di cross-compilazione del codice, specie se questo è particolarmente complesso ed articolato, è tutt'altro che immediato, risultano particolarmente importanti l'efficienza e l'affidabilità del compilatore.

Il grosso vantaggio di questa metodologia di sviluppo è che le applicazioni realizzate hanno accesso a tutte le caratteristiche native del sistema considerato e tutti i componenti dell'interfaccia grafica nativa possono essere utilizzati senza difficoltà.

Il problema, però, è che l'interfaccia grafica non può essere riutilizzata, così come le caratteristiche hardware native utilizzate. Queste caratteristiche, infatti, dipendono dalla specifica piattaforma e il modo con cui accedervi è strettamente legato ad essa. Inoltre, l'approccio non è adatto per lo sviluppo di applicazioni sofisticate, dove la cross-compilazione diventa complicata, anche a causa delle reali differenze esistenti tra le diverse piattaforme.

Tra i framework più utilizzati che utilizzano questo approccio spiccano **Xamarin** e **Corona**, descritti brevemente nel seguito di questo paragrafo.

### Xamarin

*Xamarin*, acquisito recentemente da Microsoft e reso completamente gratuito, è un ambiente di sviluppo per applicazioni mobili cross-compilate che sta riscuotendo molto successo attualmente. Grazie all'utilizzo del linguaggio C#, esso permette al programmatore di realizzare applicazioni eseguibili su diverse piattaforme.

*Xamarin* offre tutte le API native delle diverse piattaforme come normali librerie C#, permettendo così l'interazione con lo specifico dispositivo in maniera simile a come accadrebbe con il codice nativo. Alla base di *Xamarin* c'è *Mono*, un'implementazione del framework *.NET* di Microsoft open source, che consente di eseguire il codice unico ed ottenere tanti output diversi quanti sono le piattaforme target, affiancandosi per tale scopo alla sua macchina virtuale nativa.

### Corona

*Corona* è anch'esso un ambiente di sviluppo gratuito per lo sviluppo di applicazioni mobili cross-compilate; la differenza sta nel linguaggio di programmazione utilizzato, *Lua*.

L'IDE di *Corona* può essere esteso con numerosi plugin per tutte le necessità; possiede anche un proprio SDK aggiornato dalla community costantemente ed un proprio

simulatore. Questo tool è considerato particolarmente utile per lo sviluppo di videogiochi, ma risulta inadatto allo sviluppo di applicazioni mobili tradizionali.

### 2.3.5 Considerazioni finali

La classificazione riportata ai paragrafi precedenti, sebbene copra la maggior parte degli approcci multipiattaforma per lo sviluppo di applicazioni mobili, non è esaustiva, poiché come è facilmente comprensibile il mondo delle applicazioni mobili è in continua evoluzione.

In ogni caso, la scelta dell'approccio e del framework da utilizzare a questo scopo è un aspetto molto importante per la buona riuscita del progetto. Per questo è fondamentale aver compiuto preventivamente un'analisi di ciò che si ha a disposizione e di ciò che si vuole ottenere, al fine di mettere in luce le priorità che ci si vogliono dare nella fase progettuale (requisiti funzionali, sistemi operativi mobili da supportare, competenze e conoscenze personali, tempo, budget, ecc.).

Nel seguito di questa tesi è stato ritenuto di particolare interesse approfondire l'approccio cross-compilato ed in particolare il framework *Xamarin* che, a seguito dell'acquisizione dell'azienda omonima da parte di Microsoft nel marzo 2016 che l'ha reso open source e gratuito, è divenuto sempre più noto ed utilizzato, anche grazie al C#, un linguaggio ad oggetti con anni di esperienza.

# Capitolo 3

# Il framework Xamarin

## 3.1 Un po' di storia

*Xamarin* è un'azienda produttrice di software statunitense con sede a San Francisco (California), fondata nel maggio 2011 dagli ingegneri Nat Friedman e Miguel de Icaza, già noti per aver fondato circa un decennio prima l'azienda *Ximian* ed aver avviato il progetto open source chiamato *Mono* (implementazione alternativa del compilatore C# e del framework .NET di Microsoft su sistemi Linux e macOS).

La volontà della neonata azienda era quella di offrire agli sviluppatori un modo semplice e veloce per creare app multipiattaforma attraverso nuovi e prestanti ambienti di sviluppo, basandosi su un unico linguaggio orientato agli oggetti. Da questa idea, mantenendo caratteristiche e punti forti di *Mono*, si è giunti alla creazione del framework *Xamarin* omonimo.

Negli anni il progetto ha raggiunto l'approvazione non solo degli sviluppatori (oltre 1.2 milioni), ma anche delle aziende (oltre 15 mila). Tra le manifestazioni di interesse più significative è da segnalare quella di Microsoft, che nel febbraio 2016 ha acquisito l'azienda dando inizio a una nuova era per lo sviluppo di applicazioni multipiattaforma. A partire dal marzo 2016, inoltre, i prodotti *Xamarin* sono diventati ufficialmente open source e gratuiti per sviluppatori indipendenti, piccoli team e studenti, che possono utilizzarli nei propri progetti.

## 3.2 Il framework

L'obiettivo di *Xamarin* è quello di offrire strumenti che permettano agli sviluppatori di raggiungere i principali sistemi operativi per dispositivi mobili, ovvero Android, iOS e Windows, massimizzando il riutilizzo del codice e delle competenze

acquisite. Per fare questo, come già detto, *Xamarin* sfrutta *Mono*, il quale permette di utilizzare C#, il potente e versatile linguaggio di programmazione orientato agli oggetti di Microsoft, per sviluppare applicazioni multipiattaforma.

*Mono* e C# sono quindi la base su cui *Xamarin* costruisce i propri strumenti e le proprie tecnologie di sviluppo. Questo è un fattore fondamentale per gli sviluppatori .NET, che possono quindi riutilizzare le proprie competenze anche su sistemi non Microsoft. Per rendere possibile tutto questo, *Xamarin* ha sviluppato delle librerie che rielaborano ed espongono in forma .NET/Mono le API native delle varie piattaforme:

- *Xamarin.Android*;
- *Xamarin.iOS*;
- *Xamarin.Mac*.

Oltre ad esse si aggiunge *Xamarin.WinPhone* che, ovviamente, è stata sviluppata facilmente, essendo che per lo sviluppo di applicazioni per Windows Phone viene fatto già uso nativamente del C# e di API in forma .NET.

Con un unico codice condiviso basato sul linguaggio C#, in ottica totalmente orientata a .NET, gli sviluppatori possono usare gli strumenti *Xamarin* per scrivere applicazioni native Android e iOS, oltre che ovviamente Windows, con interfacce utente native. Da ciò è facile notare, tuttavia, come l'eventuale ampliamento futuro del supporto verso altri sistemi operativi possa essere effettuato in maniera più o meno rapida aggiungendo esclusivamente una nuova libreria *Xamarin.X*.

Oltre al runtime e alle librerie, sono stati pensati e creati appositamente per questo scopo degli ambienti di sviluppo e tools, dei quali si parlerà nei paragrafi successivi.

Non di minore importanza è il fatto che le librerie e i tool di sviluppo sono stati rilasciati come progetto open source, a cui la comunità di sviluppatori può contribuire. In realtà, *Xamarin* offre agli sviluppatori tutto ciò di cui hanno bisogno per sviluppare, pubblicare, analizzare e mantenere app native per Android, iOS e Windows con riferimento a tutto il ciclo di vita dell'applicazione stessa. Ciò significa che *Xamarin* non è soltanto più sviluppo di app, ma ingloba anche servizi correlati.

### 3.2.1 API e copertura C#

Come riportato nella documentazione ufficiale online di *Xamarin*, il C# è considerato il miglior linguaggio di programmazione per lo sviluppo di applicazioni mobili. Qualsiasi cosa si possa fare in Objective-C, Swift o Java, può essere fatta in C#.

*Xamarin* ha scelto dunque C# come unico linguaggio, permettendo agli sviluppatori di ottenere applicazioni native senza dover necessariamente conoscere altri linguaggi e utilizzando, come vedremo in seguito, un unico ambiente di sviluppo.

Tutto ciò è permesso grazie alle sopracitate librerie sviluppate da *Xamarin*, le quali permettono di accedere a qualsiasi API dei sistemi operativi nativi tramite meccanismi di *binding* e realizzare un *mapping* in C#, con una copertura delle traduzioni che raggiunge il 100%. Tutte le API native diventano quindi utilizzabili in *Xamarin* tramite la sintassi e le *naming convention* del C# e del framework .NET; in figura 3.1 sono mostrati alcuni esempi. Con il binding nativo, le librerie *Xamarin* permettono inoltre di chiamare codice esistente scritto nei linguaggi nativi da C#, consentendo il riutilizzo del codice, così come utilizzare librerie scritte in altri linguaggi, qualora disponibili esclusivamente in un linguaggio specifico.

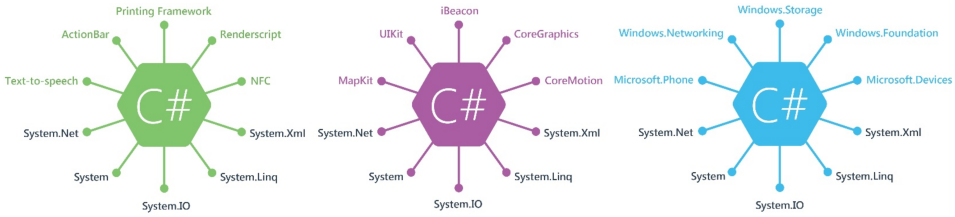



Figura 3.1. Alcune delle librerie C# raggiungibili (Android, iOS e Windows Phone rispettivamente)

### 3.2.2 Cross-compilazione

Dopo aver scritto il codice sorgente in C#, occorrerà compilarlo per ottenere il file binario eseguibile dell'applicazione nativa. In prima istanza tale codice viene convertito dal framework in un linguaggio intermedio, chiamato CIL (Common Intermediate Language), indipendente dal linguaggio e comprensibile dal CLR (Common Language Runtime), l'ambiente di esecuzione .NET. Il codice CIL però non è ancora eseguibile direttamente dal sistema operativo: questo avviene solo in un secondo processo di compilazione, che però varia a seconda della piattaforma:

#### Android

Il CIL ottenuto viene incluso nell'APK e, solamente quando l'applicazione verrà eseguita, viene convertito in codice nativo tramite compilazione **Just-In-Time (JIT)** dal dispositivo, con un approccio simile al framework .NET. *Xamarin* produce quindi un pacchetto che offre le stesse performance di un'applicazione scritta in Java.

#### iOS

A causa di alcune restrizioni proprie di Apple, la generazione dinamica di codice tramite un compilatore JIT è proibita. Il CIL ottenuto viene ricompilato, ove

possibile, direttamente in codice macchina (assembly ARM) tramite compilazione **Ahead-Of-Time (AOT)**. Il processo di build produce direttamente codice nativo, offrendo le stesse performance di un'app realizzata con Objective-C o Swift.

#### Windows Phone

Ottenuto il CIL non c'è bisogno di alcun tool particolare per essere eseguito, essendo già presente nella piattaforma l'ambiente di esecuzione CLR .NET.


Figura 3.2. Compilazione in iOS e Android rispettivamente

## 3.3 L'approccio tradizionale

L'approccio classico di *Xamarin* permette di realizzare attraverso le librerie sopra citate un layer comune condiviso tra le varie piattaforme scritto in C# che rappresenta il back-end, ovvero la logica dell'app *code behind*. Esso, a seconda della complessità del codice dell'applicazione da realizzare, può coprire in media il 75% del codice nativo che può essere condiviso (in alcuni casi questa percentuale raggiunge anche il 100%).

Su questa base comune di back-end si poggerà poi la realizzazione dell'interfaccia grafica nativa, una per ogni piattaforma considerata, sfruttando i linguaggi di *markup* e i tools nativi. Nello specifico, quindi, si potrà avere:

- Android: C# + XML;
- iOS: C# + XIB/Storyboard;
- Windows Phone: C# + XAML.

Quanto descritto è mostrato graficamente in figura 3.3, tratta dalla documentazione ufficiale.

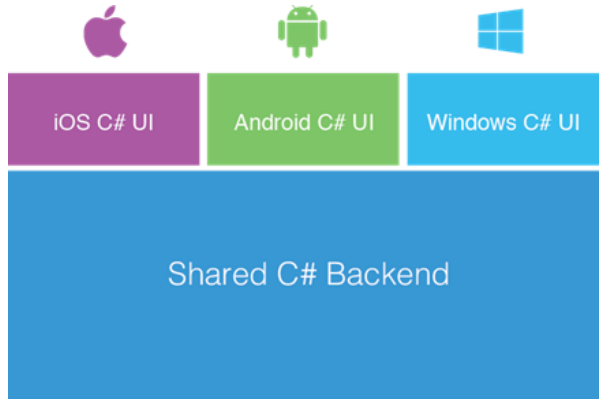



Figura 3.3. Approccio offerto da *Xamarin* "tradizionale"

## 3.4 Il nuovo approccio Xamarin.Forms

L'evoluzione dell'approccio classico, introdotta nel maggio 2014, ha portato alla nascita di una nuova libreria chiamata *Xamarin.Forms*, che permette di realizzare la condivisione del codice tra le piattaforme a tutti i livelli, compreso quello dell'interfaccia grafica, massimizzando in questo modo il riutilizzo del codice. L'obiettivo a cui si vuole ambire, infatti, è il raggiungimento del 100% del codice condiviso.

A tal fine, come graficamente mostrato in figura 3.4, *Xamarin.Forms* fornisce un livello di astrazione che consente di definire l'interfaccia grafica una solta volta, attraverso una derivazione di XAML (eXtensible Application Markup Language), il linguaggio di markup caro agli sviluppatori .NET per la progettazione della UI in modalità dichiarativa, oppure da codice C#.

Lo sviluppatore utilizzerà esclusivamente tali astrazioni per costruire l'interfaccia grafica della propria applicazione attraverso i classici componenti (pagine, layout, viste, bottoni, etichette, ecc.) e, solo in fase di esecuzione, *Xamarin.Forms* farà il render selezionando i corrispettivi elementi visuali appropriati specifici del sistema su cui l'applicazione sta girando, ottenendo un aspetto nativo. Ad esempio, il controllo *Label* di *Xamarin.Forms* diventerà una *TextView* in Android, una *UILabel* in iOS e un *TextBlock* in Windows Phone.

Questo codice condiviso è contenuto in uno specifico progetto all'interno della *solution* e può essere di due tipi, analizzati più in dettaglio nel prossimo paragrafo:

- *shared*: file condivisi tra i progetti tramite il meccanismo dei link;
- *portable*: una vera e propria *portable class library*, che produce in output una libreria DLL.

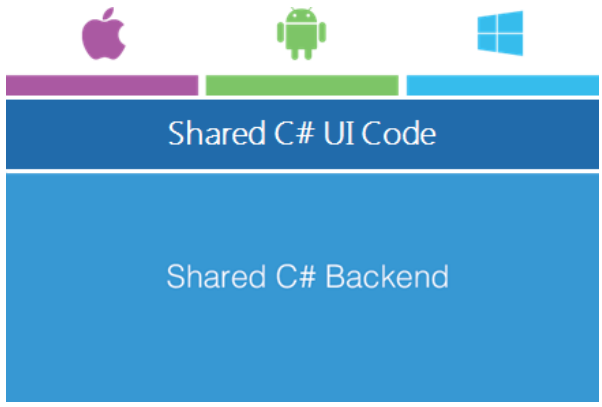



Figura 3.4. Approccio offerto da *Xamarin.Forms*

All'interno del progetto condiviso, lo sviluppatore scriverà tutto il codice che può funzionare con certezza su tutte le piattaforme, anche se ci sono degli aspetti di cui tenere conto. Il primo è che *Xamarin.Forms* è in grado di mappare solo quegli elementi che hanno una corrispondenza in tutte le piattaforme. Ad esempio, sia iOS che Android che Windows hanno caselle di testo ed etichette, per cui *Xamarin.Forms* definisce controlli *Entry* e *Label* che con certezza saranno utilizzabili; tuttavia, iOS, Android e Windows gestiscono in modo diverso entrambi gli elementi visuali, con proprietà e comportamenti diversi. *Xamarin.Forms*, quindi, definisce controlli che hanno esclusivamente proprietà e comportamenti comuni a tutti, lasciando fuori aspetti specifici di ciascuna piattaforma (anche se recentemente è stata inserita la possibilità di utilizzare controlli nativi). Funzionalità come l'accesso ai sensori o quelle telefoniche, ad esempio, essendo basate su API completamente diverse tra loro, necessitano di essere gestite in modo differente a seconda della piattaforma. In *Xamarin.Forms* ci sono tre possibilità per fare ciò:

- *Custom Renderer*: riguarda la personalizzazione di tutti i layout e gli elementi visuali in generale, utilizzando controlli nativi in modo diretto, tramite l'implementazione di una classe per ogni progetto specifico per descrivere proprietà tipiche di quella piattaforma;
- *Dependency Injection*: riguarda l'accesso a funzionalità hardware e software (sensori, fotocamera, file system) specifiche della piattaforma, realizzata tramite la definizione di un'interfaccia comune per descrivere i membri di una classe e fornendo nei progetti specifici la relativa implementazione;
- *Effects*: riguarda la situazione in cui si vogliono utilizzare controlli nativi, ma senza dover intervenire anche sul loro comportamento, ad esempio solo per

la modifica di alcuni stili estetici. In tal caso è possibile ridefinire alcuni stili dei controlli che possono poi essere consumati nello XAML.

Il secondo aspetto è che *Xamarin.Forms* è una piattaforma piuttosto giovane e in piena evoluzione, quindi non sempre è disponibile ciò che siamo abituati ad utilizzare nelle piattaforme specifiche o in altri contesti di sviluppo, ma la sua evoluzione è talmente rapida che le nuove versioni includono sempre importanti miglioramenti, strumenti e controlli nuovi. *Xamarin.Forms* è offerto attraverso l'omonimo pacchetto *NuGet* (il package manager gratuito realizzato per le piattaforme Microsoft), dal quale è possibile anche scaricare numerosi *plugin* per *Xamarin*, ovvero librerie per l'accesso a funzionalità specifiche di piattaforma tramite API unificate fruibili dai progetti condivisi senza ricorrere alle tecniche descritte sopra.

### 3.4.1 Componenti della UI

Esistono quattro gruppi principali nei quali si possono classificare i componenti utilizzati per creare l'interfaccia utente di un'applicazione *Xamarin.Forms*: pages, layouts, views e cells.

#### Pagine

Le pagine rappresentano le schermate dell'applicazione. In altre parole, sono elementi visivi che occupano la maggior parte (o tutto) lo schermo e contengono un singolo figlio. Una pagina rappresenta un *View Controller* in iOS o una *Page* in Windows Phone. Su Android ogni pagina può essere paragonata ad una *Activity*, anche se le pagine in *Xamarin.Forms* non sono vere e proprie *Activity*.

La tipologia più semplice è la *ContentPage*, che rappresenta una generica pagina con del contenuto.

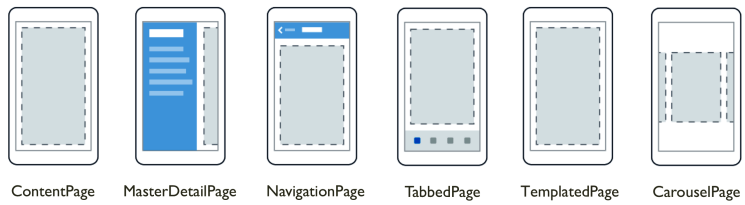



Figura 3.5. Classificazione delle pagine in *Xamarin.Forms*

#### Layouts

I layout vengono usati per organizzare i controlli dell'interfaccia utente (views e cells) in strutture logiche. Essi sono un sottotipo specializzato di *View*, che fungono

da contenitori per altri layout o viste. Di solito contengono la logica per impostare la posizione e la dimensione degli elementi figlio nelle applicazioni *Xamarin.Forms*.

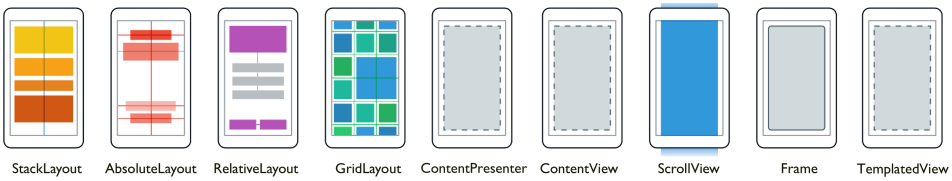



Figura 3.6. Classificazione dei layouts in *Xamarin.Forms*

### Controlli (viste e celle)

Con il termine vista, *Xamarin.Forms* definisce i blocchi di costruzione delle interfacce utente come pulsanti, etichette o caselle di testo, che possono essere più comunemente conosciuti come controlli dei widget. Questi elementi sono tipicamente sottoclassi di *View*.

Con il termine cella, invece, si intende un elemento specializzato utilizzato per descrivere l'aspetto dei singoli item di una tabella o di una lista. Essa non è un elemento visivo, ma descrive solo un template per la creazione di un elemento visivo. Le celle sono elementi progettati per essere aggiunti ai controlli *ListView* o *TableView*.

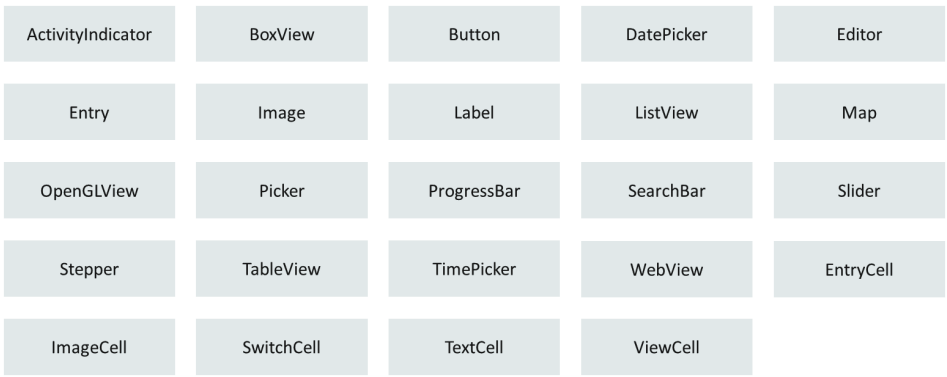



Figura 3.7. Classificazione dei controlli in *Xamarin.Forms*

## 3.5 Condivisione del codice

Come accennato nel precedente paragrafo, *Xamarin* offre due diverse modalità di condivisione e riutilizzo di codice tra i progetti delle piattaforme che comporranno la soluzione multipiattaforma:

- *Shared Assets Project (SAP)*: Offre la possibilità di condividere il codice attraverso l'intera soluzione, sfruttando le direttive del compilatore per includere differenti porzioni di codice, specifiche per ogni piattaforma.
- *Portable Class Libraries (PCLs)*: A differenza degli Shared Assets Project, l'approccio PCL permette di usare un sottoinsieme di librerie offerte dal framework .NET per sfruttare lo stesso codice nei progetti che compongono l'intera soluzione. L'insieme di librerie da utilizzare è limitato dalle piattaforme che si desidera supportare per la propria applicazione.

L'obiettivo di una strategia di condivisione di codice è quello di supportare l'architettura mostrata in figura 3.8, in cui un singolo codice base può essere utilizzato da più piattaforme.

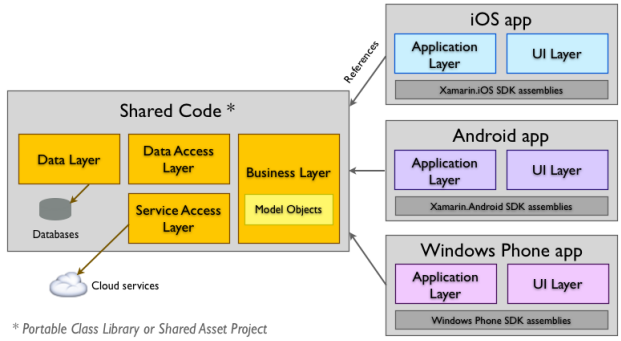



Figura 3.8. Architettura di un'app multipiattaforma

Spesso viene fatto uso delle PCL, ma questa non è sempre la scelta migliore; infatti, per app sufficientemente semplici, potrebbe essere più conveniente usare una soluzione SAP invece che PCL, da adattare al meglio alle proprie esigenze di sviluppo.

### 3.5.1 Shared Assets Project

L'approccio SAP consente di scrivere codice comune che viene condiviso tra i progetti di interesse durante la compilazione della soluzione. I file inclusi nel progetto,

infatti, vengono copiati all'interno dei progetti specifici delle piattaforme e compilati come parte di essi. Durante il processo di build non viene prodotta una .dll compilata, ma più semplicemente un assortimento di file.

Quindi, una tipica soluzione potrebbe contenere quattro progetti: uno Shared, uno specifico Android, uno specifico iOS, uno specifico Windows Phone.

Inoltre, questo approccio supporta la compilazione condizionale tramite le direttive del compilatore, per gestire le differenze tra le varie piattaforme e includere codice specifico solo in un sottoinsieme dei progetti.

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

In figura 3.9 si può vedere come cambia l'architettura vista precedentemente.

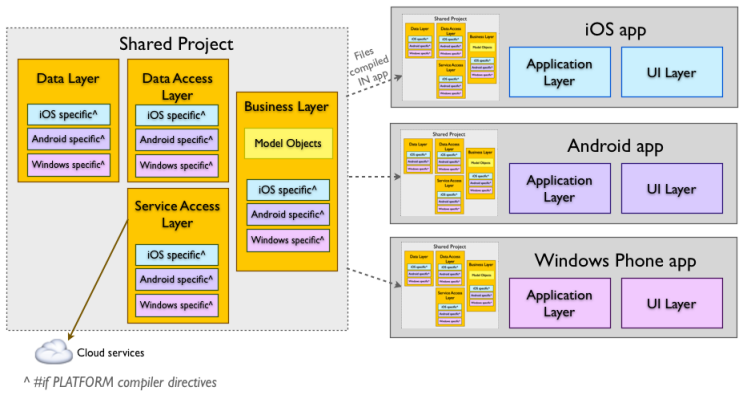



Figura 3.9. Architettura di un'app multipiattaforma secondo l'approccio SAP

### 3.5.2 Portable Class Libraries

Le PCL costituiscono un approccio alternativo agli Shared Project, per la condivisione del codice tra le diverse versioni dell'app destinate a diversi sistemi operativi. Tuttavia, anche in questo caso, lo sviluppo dell'applicazione è diviso in più progetti distinti, uno per ogni specifica piattaforma supportata, più uno Portable, il quale racchiude tutti quegli elementi comuni dello sviluppo che, solo in fase di compilazione, vengono iniettati nei rispettivi progetti specifici.

Le Portable Class Libraries in altre parole sono delle librerie portabili, che possono quindi essere consumate su più piattaforme a condizione che abbiano come target solo il sottoinsieme di API comune e supportato su tutte le piattaforme di destinazione. Le PCL producono una .dll come output, che può essere distribuita e utilizzata in altri progetti. Inoltre non supportano la compilazione condizionale, e quindi non possono contenere codice specifico di una certa piattaforma, proprio perché altrimenti perderebbero la loro caratteristica della portabilità. La figura 3.10 mostra la sua architettura.

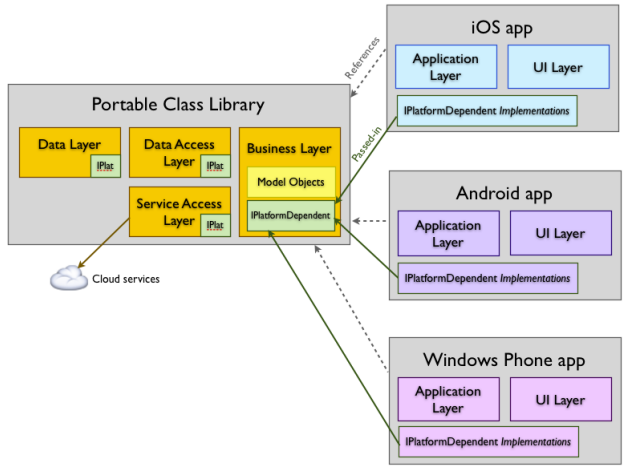



Figura 3.10. Architettura di un'app multipiattaforma secondo l'approccio PCL

Oltre al progetto PCL automaticamente incluso nelle soluzioni *Xamarin.Forms*, è possibile creare altri progetti di tipo PCL che facciano parte dell'architettura di un'app oppure che siano delle librerie autonome e riutilizzabili. Ad esempio, con il pattern *Dependency Injection* si possono includere nella soluzione le funzionalità hardware e software specifiche di una piattaforma.

## 3.6 Ambienti di sviluppo

*Xamarin*, contestualmente alla creazione del framework, aveva realizzato un proprio ambiente di sviluppo chiamato *Xamarin Studio*. Esso permetteva di sviluppare le proprie applicazioni solo per Android e Windows Phone, se usato su PC tradizionale, oppure per Android e iOS, se usato su Mac. Oggi, a seguito dell'acquisizione di *Xamarin* da parte di Microsoft, *Xamarin Studio* non è praticamente più utilizzato: Windows, preferendo l'utilizzo dell'alternativa **Visual Studio**, ne ha tolto il

supporto e su Mac si sta gradualmente sostituendo con l'alternativa **Visual Studio for Mac**, un IDE specifico per Mac OS che combina l'ambiente di *Xamarin Studio* con peculiarità tipiche di *Visual Studio*.

*Visual Studio*, tramite l'aggiunta di un plugin, integra facilmente il supporto a *Xamarin* ed alle sue librerie; inoltre rende possibile lo sviluppo per tutti e tre i principali sistemi operativi (Android, iOS e Windows Phone) e, se si possiedono già conoscenze di tale ambiente, non si farà alcuna fatica poiché si andranno ad utilizzare gli stessi strumenti già noti. Per questi motivi quindi *Visual Studio* si sta rivelando la scelta vincente.


L'intero sviluppo di una applicazione Android può essere effettuato utilizzando *Visual Studio*, sia su Windows che su Mac, sia su macchina fisica che virtuale; per le applicazioni iOS, invece, la fase di compilazione necessita della presenza di una macchina fisica Mac con installata la versione aggiornata dell'IDE *Xcode* e dell'SDK iOS. Se si utilizza *Visual Studio* in ambiente Windows, *Xamarin* fornisce un servizio per poter compilare il proprio codice su una macchina Mac che si trova in rete per poi visualizzare il risultato su un simulatore o su un device fisico. Questo metodo, a quanto specificato sul loro sito ufficiale, funziona anche con *Visual Studio* in esecuzione all'interno di una macchina virtuale Windows su un computer Mac.

Infine, senza entrare troppo nel dettaglio, per quanto riguarda il sistema operativo Linux non è previsto alcun supporto a *Xamarin*; l'alternativa è l'uso del suo precursore *Mono*, con l'appoggio dell'IDE *MonoDevelop*, una delle migliori alternative a *Visual Studio* per sviluppare in C# con Mono. Esso, tra le altre cose, facilita la possibilità di effettuare porting di software scritti su *Visual Studio* per Windows anche su Linux e Mac, e viceversa.

Nel seguito di questo paragrafo vengono approfonditi gli *steps* e i dettagli di configurazione seguiti durante lo sviluppo del caso di studio scelto, necessari per poter sviluppare con *Xamarin* in *Visual Studio* su Windows (il riferimento è alla versione più recente disponibile al momento della stesura di questa tesi).

### 3.6.1 Installare Visual Studio con Xamarin su Windows

Poiché *Xamarin* è ora incluso in tutte le edizioni di *Visual Studio* senza alcun costo aggiuntivo e non richiede una licenza separata, è possibile utilizzare il proprio programma di installazione per scaricare e installare gli strumenti *Xamarin*.

Dopo aver scaricato *Visual Studio 2017* nelle sue versioni *Community*, *Professional* o *Enterprise* (la versione *Express* non supporta *Xamarin*) occorre avviarne l'installazione, avendo cura di selezionare *Mobile development with .NET* dalle voci presenti nella schermata di installazione (fig. 3.11).

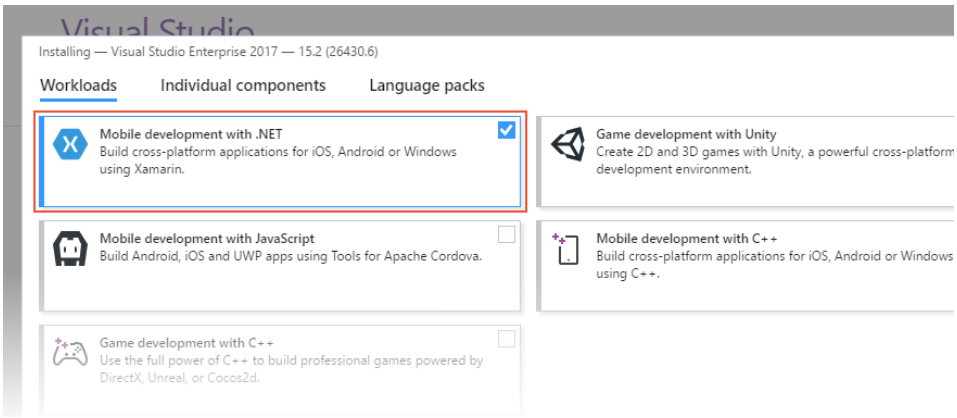



Figura 3.11. Schermata di installazione in Visual Studio 2017

Qualora fosse già installato Visual Studio sul proprio PC, è possibile aggiungere *Xamarin* alla versione esistente rieseguendo l'installazione e selezionando, quando richiesto, la voce corrispondente allo sviluppo cross-platform in *Xamarin*.

Per verificare che *Xamarin* sia stato correttamente installato occorre aprire il menu *Help* e accertarsi che compaia una voce relativa a *Xamarin*; nelle versioni più vecchie invece occorre recarsi, sempre a partire dallo stesso menu, su *About Microsoft Visual Studio* e controllare la lista dei prodotti installati.

### 3.6.2 Creare un progetto Xamarin.Forms in Visual Studio

Per creare una nuova applicazione in Visual Studio occorre innanzitutto fare click su *File* > *New* > *Project* per iniziare un nuovo progetto (fig. 3.12).

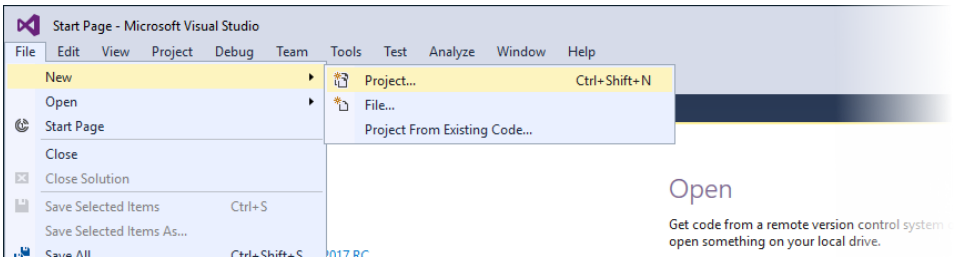



Figura 3.12. Iniziare un nuovo progetto in Visual Studio 2017

Nella finestra di dialogo *New project*, recarsi su *Cross-Platform* e selezionare l'opzione *Cross Platform App (Xamarin.Forms or Native)*; poi scegliere il nome e una posizione appropriata per il progetto (fig. 3.13).

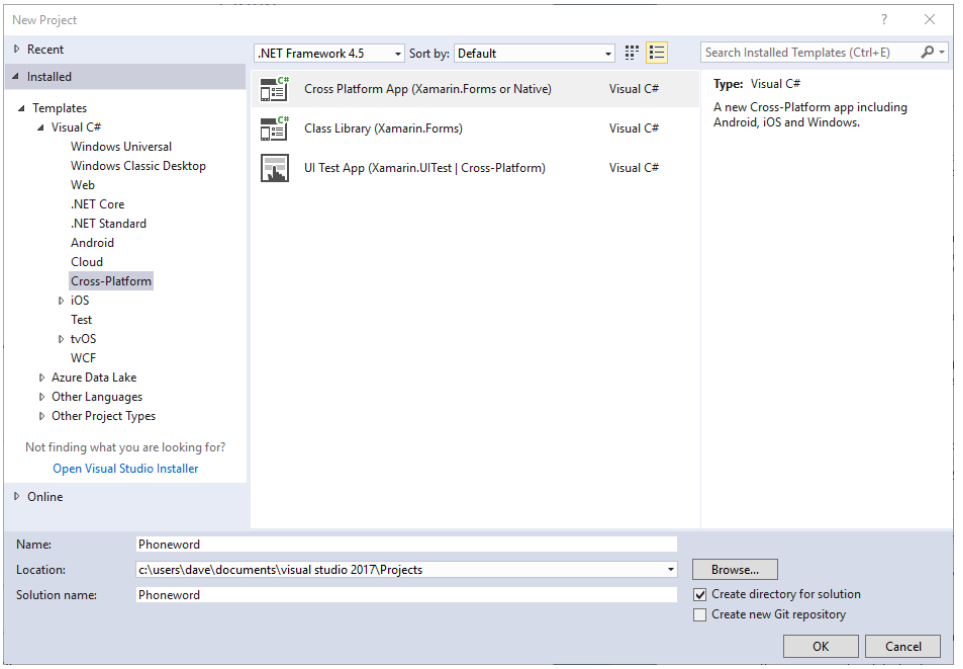



Figura 3.13. Scegliere il tipo di progetto da creare in Visual Studio 2017

Successivamente apparirà una nuova finestra di dialogo, *New Cross Platform App*, dalla quale scegliere il template (*Blank App*), la tecnologia UI (*Xamarin.Forms*) e la strategia di condivisione del codice (PCL o SAP) da utilizzare (fig. 3.14).

Qualora si voglia anche sviluppare una soluzione per Windows, potrebbe ancora aprirsi la finestra di dialogo *New Universal Windows Project*, per scegliere le versioni massima e minima di Windows 10 supportate.

Il numero di progetti generati varierà a seconda di quanti SDK sono installati; nel caso vi siano tutti, in *Solution Explorer* si vedranno sei progetti (fig. 3.15) facenti parte di un'unica soluzione (Portable o

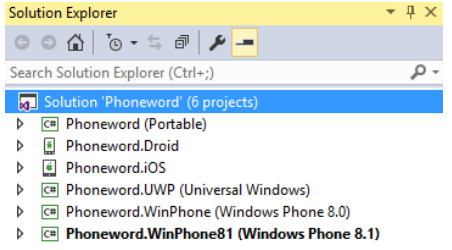



Figura 3.15. Progetti della soluzione *Xamarin.Forms*

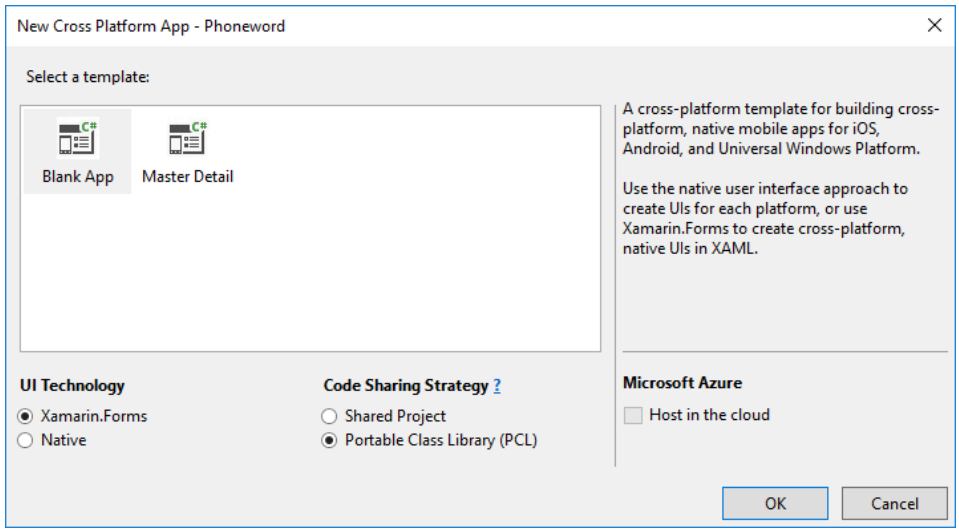



Figura 3.14. Configurare i dettagli del progetto *Xamarin* in Visual Studio 2017

Shared, iOS, Android, UWP, Windows Phone 8.1 e Windows 8.1), ciascuno con la propria struttura di piattaforma che ci si aspetterebbe, anche se è comunque possibile escludere quelli che non interessano.

Il cuore di questa soluzione è nel progetto Portable o Shared: è qui infatti che dovrà essere scritto il codice condiviso, incluso quello relativo alla definizione della UI. Tra i files che compongono il progetto condiviso vi sono i seguenti:

- *App.xaml*: il codice di markup XAML per la classe *App*, che definisce un dizionario di risorse (stili, data template) fruibili dall'applicazione nel suo complesso.
- *App.xaml.cs*: il *code-behind* per la classe *App*, responsabile di instanziare la prima pagina mostrata dall'applicazione su ogni piattaforma e di gestire gli eventi del ciclo di vita dell'applicazione stessa.
- *MainPage.xaml*: il codice di markup XAML per la classe *MainPage*, che definisce l'interfaccia utente della pagina mostrata all'avvio dell'applicazione.
- *MainPage.xaml.cs*: il *code-behind* per la classe *MainPage*, che contiene la logica di business eseguita quando l'utente interagisce con la pagina.

L'operazione preliminare alla scrittura del codice che è bene sempre fare è l'aggiornamento dei pacchetti all'ultima versione stabile. Pertanto, da *Solution*

*Explorer* bisogna fare click con il tasto destro del mouse sull'intera solution e selezionare *Manage NuGet packages for solution*. Nella finestra mostrata, occorre aprire la scheda *Updates*, selezionare il package *Xamarin.Forms* e aggiornarlo, tralasciando gli altri pacchetti eventualmente aggiornabili poiché potrebbero far sorgere problemi di compatibilità con *Xamarin.Forms*.

Infine, dopo aver sviluppato la propria soluzione occorrerà compilarla e testarla. Selezionando la voce di menu *Build* > *Build*, l'applicazione verrà generata e verrà mostrato un messaggio di successo nella barra di stato di Visual Studio oppure, qualora vi siano degli errori, essi verranno segnalati in modo tale da poterli correggere e rieseguire la procedura.

Per quanto riguarda il test dell'applicazione, occorre innanzitutto selezionare il progetto di avvio a seconda della piattaforma su cui si vuole eseguire: facendo click con il tasto destro del mouse sul progetto specifico contenuto in *Solution explorer*, selezionare *Set as startup project*. Dopodichè, nella toolbar di Visual Studio, premere il bottone triangolare verde *Start* (fig. 3.16) per lanciare l'applicazione in esecuzione sull'emulatore o sul dispositivo fisico. Ulteriori dettagli in merito allo sviluppo ed al testing sulle specifiche piattaforme sono analizzati nei successivi paragrafi.


Figura 3.16. Toolbar per il testing in Visual Studio 2017

### 3.6.3 Sviluppare per Android su Windows

*Xamarin.Android* utilizza Java JDK e Android SDK per creare applicazioni. Durante l'installazione, Visual Studio colloca questi strumenti nelle loro posizioni predefinite e configura l'ambiente di sviluppo con il path appropriato, che, per la maggior parte degli utenti, funziona senza ulteriori modifiche. Tuttavia, qualora questi strumenti siano già installati in una posizione diversa, è possibile utilizzarli e personalizzare il riferimento al path da Visual Studio facendo click su *Tools* > *Options* > *Xamarin* > *Android Settings* (fig. 3.17).

Inoltre, dato che per determinare la compatibilità dell'applicazione tra le varie versioni di Android vengono usati livelli di API differenti, a seconda di quali di essi si desiderano specificare potrebbe essere necessario scaricare e installare altri componenti SDK Android o strumenti opzionali. A tale scopo, *Visual Studio* consente di utilizzare l'*Android SDK Manager*, accessibile facendo click su *Tools* > *Android* > *Android SDK Manager* (fig. 3.18).

Per impostazione predefinita, *Visual Studio* installa *Google Android SDK Manager*, dal quale è possibile installare le versioni del pacchetto *Android SDK Tools*

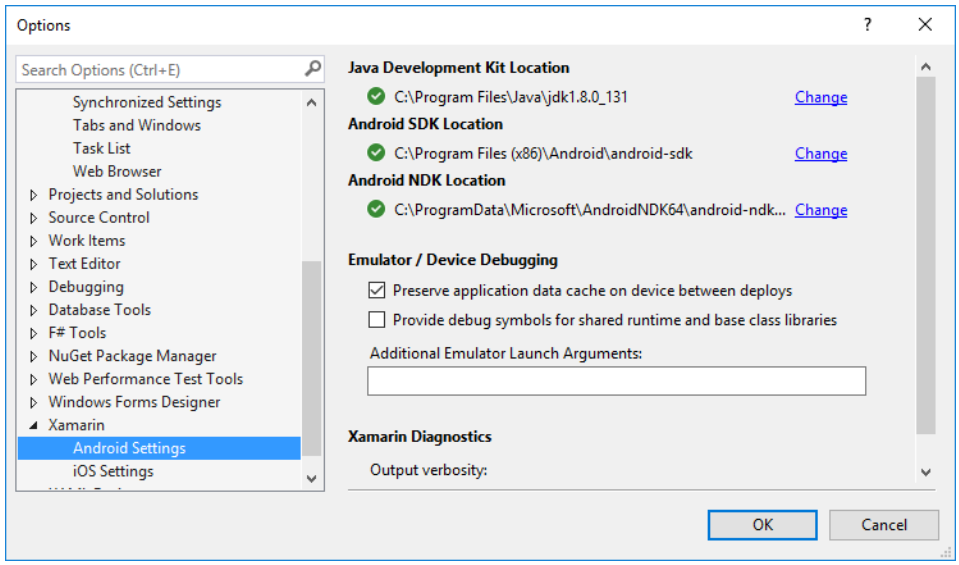



Figura 3.17. Configurare le impostazioni di Android per *Xamarin*

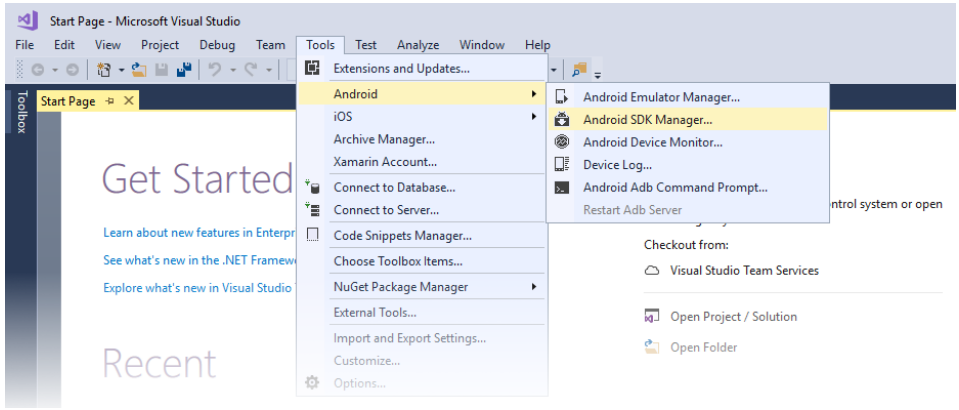



Figura 3.18. Accedere all'*Android SDK Manager* in Visual Studio 2017

fino alla 25.2.3. Tuttavia, qualora sia necessario utilizzare una versione successiva, occorre installare il nuovo plugin *Xamarin Android SDK Manager*. Ciò è necessario perché l'SDK Manager di Google è divenuto deprecato in tale versione di tale pacchetto.

Una volta creata l'applicazione *Xamarin.Forms* secondo gli steps descritti nel

paragrafo precedente, si può testare la sua versione Android. A tal fine, se non si dispone di un dispositivo Android fisico da utilizzare per il test è possibile utilizzare un emulatore, come ad esempio *Google Android SDK Emulator* o *Visual Studio Emulator for Android*, ciascuno basato su una particolare tecnologia di virtualizzazione (*Intel HAXM* o *Hyper-V*, rispettivamente nell'esempio). Poiché la CPU di un computer di sviluppo può supportare una sola tecnologia di virtualizzazione alla volta, è meglio avere solo un emulatore in uso.

Se invece si dispone di un dispositivo Android fisico da utilizzare per il test, è opportuno configurarlo per lo sviluppo, quindi collegarlo al computer per eseguire ed effettuare il debug della versione Android dell'applicazione. Questa strada è stata quella seguita nello svolgimento dei test sul caso di studio.

### 3.6.4 Sviluppare per iOS su Windows

*Xamarin.iOS* consente alle applicazioni iOS di essere scritte e testate sui computer Windows, con l'uso di un Mac in rete per la compilazione e la distribuzione. A tal fine, sono supportate più configurazioni: l'esecuzione di *Visual Studio* all'interno di una macchina virtuale Windows su un Mac oppure l'esecuzione su macchina separata connessa in rete con un Mac (la seconda configurazione è stata quella utilizzata per il caso di studio). Indipendentemente dalla configurazione scelta, *Visual Studio* si connette al Mac in modo rapido e sicuro utilizzando SSH, come schematizzato nel diagramma in figura 3.19. L'ausilio di un computer Mac è necessario poiché le applicazioni iOS non possono essere create senza un compilatore Apple e non possono essere distribuite senza i certificati Apple e i tool di firma.

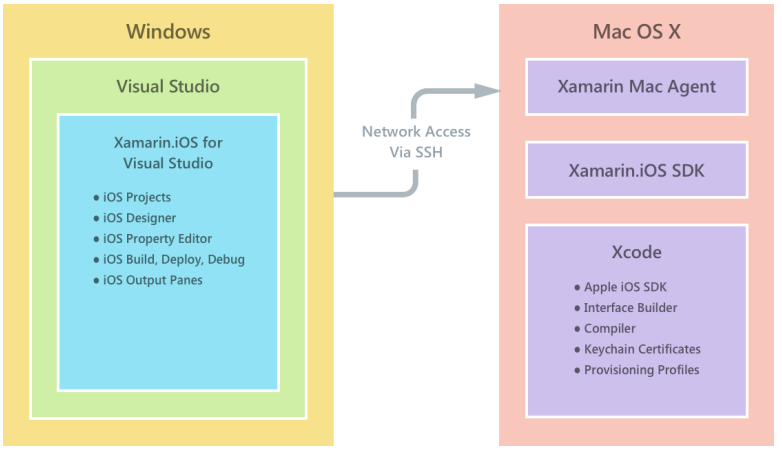



Figura 3.19. Workflow di sviluppo con *Xamarin.iOS*

*Xamarin.iOS* richiede iOS SDK e Xamarin.iOS SDK, con un Mac che esegua una versione stabile di OS X e su cui sia installato l'IDE Apple Xcode. Come già ribadito, inoltre, il computer Windows deve essere in grado di raggiungere il Mac tramite la rete. Inoltre, per la distribuzione delle applicazioni su un dispositivo fisico è necessario un account *Apple Developer*, che consente la generazione dei relativi certificati di sviluppo da installare sul Mac in rete; dettagli in merito verranno approfonditi nel seguito di questo paragrafo.

Una volta installati i tool occorre configurare il Mac per essere in grado di connettersi a *Visual Studio* su Windows. Per tale scopo, è necessario disporre di versioni corrispondenti di *Xamarin.iOS* sia sulla macchina Windows che su quella Mac (stesso canale in entrambi gli IDE oppure canale *Stable*).

Per permettere la comunicazione tra l'estensione *Xamarin* per *Visual Studio* e il Mac, è necessario abilitare il *remote login* sul Mac, seguendo i passi qui descritti. Come prima cosa, aprire *Spotlight (Cmd-Space)*, cercare *Remote login* e selezionare il risultato *Sharing*; questo aprirà la schermata *System preferences*.

Selezionare quindi l'opzione *Remote login* tra quelle presenti nella lista di *Service* (fig. 3.20) per consentire la connessione tra *Visual Studio* su Windows e Mac. Assicurarsi inoltre che, nella stessa schermata, l'accesso sia impostato su *All users* oppure che lo username dell'utente sia incluso nella lista degli utenti abilitati.

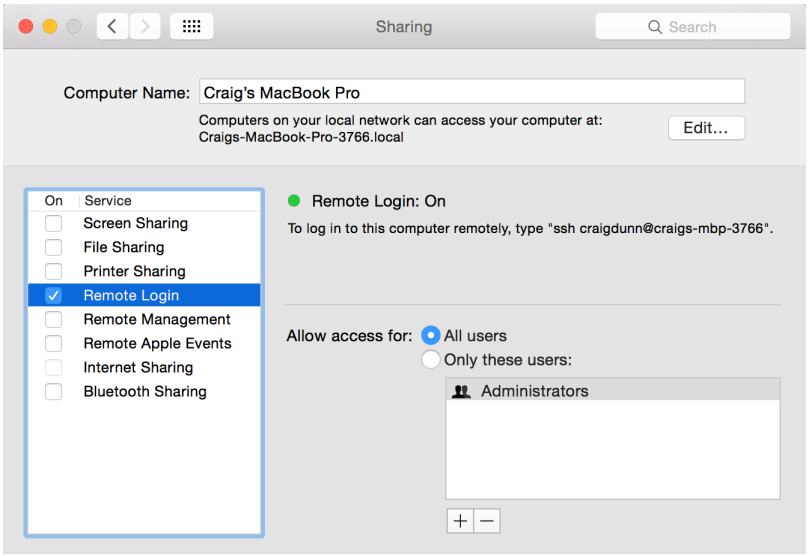



Figura 3.20. Opzione *Remote login* nella lista di *Service*

Infine, se si dispone del firewall OS X impostato per bloccare le applicazioni firmate, potrebbe essere necessario consentire a *mono-sgen* di ricevere le connessioni

in entrata. Verrà visualizzata una finestra di avviso se questo fosse il caso.

Terminata la configurazione del Mac e dopo aver creato l'applicazione *Xamarin.Forms*, si può testare la sua versione iOS. A tal fine, prima di avviare il debug, bisogna assicurarsi che l'host Mac sia connesso a *Visual Studio*. Per effettuare il collegamento, innanzitutto bisogna recarsi in *Tools* > *Options* su *Visual Studio*, selezionare *Xamarin* > *iOS Settings* (fig. 3.21) e fare click sul bottone *Find Xamarin Mac Agent*; in alternativa, un accesso veloce alla stessa finestra è dato dal pulsante apposito sulla toolbar iOS.

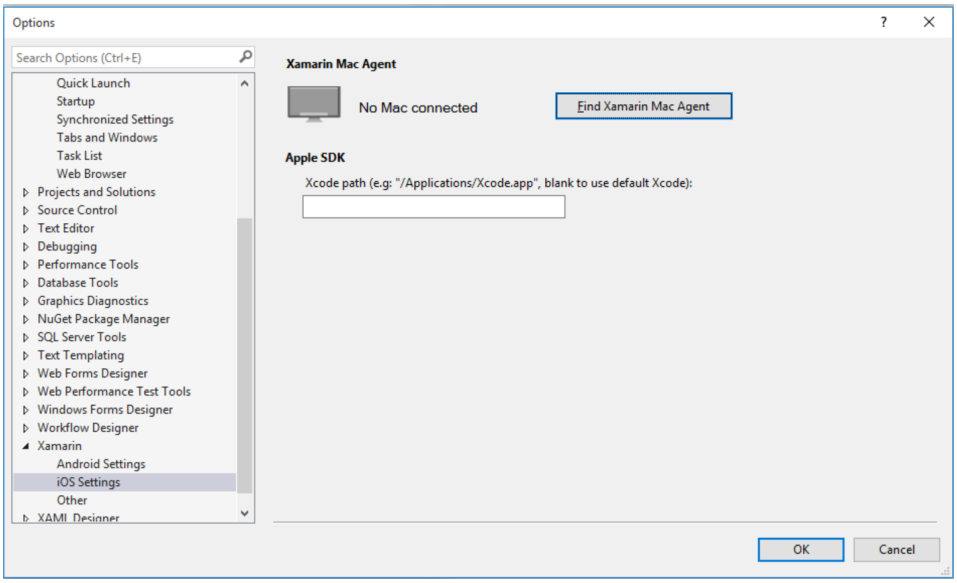



Figura 3.21. Finestra di ricerca di *Xamarin Mac Agent*

Se il computer Mac è stato configurato correttamente per consentire l'accesso remoto, sarà possibile vederlo e selezionarlo dall'elenco *Xamarin Mac Agent* (fig. 3.22), nel quale sono mostrati tutti gli host precedentemente connessi e rilevati come macchine disponibili per l'accesso remoto.

La prima volta che ci si connette a un Mac, verrà richiesto di immettere le credenziali dell'utente remoto (che deve essere un amministratore) per consentire la connessione remota SSH (fig. 3.23). Dopo aver fatto click su *Login*, se l'operazione è andata a buon fine, comparirà l'icona che indicherà l'avvenuta connessione, la quale verrà memorizzata e automaticamente mantenuta ogni volta che si avvierà nuovamente *Visual Studio*. In ogni istante ci può essere un solo Mac collegato alla volta.

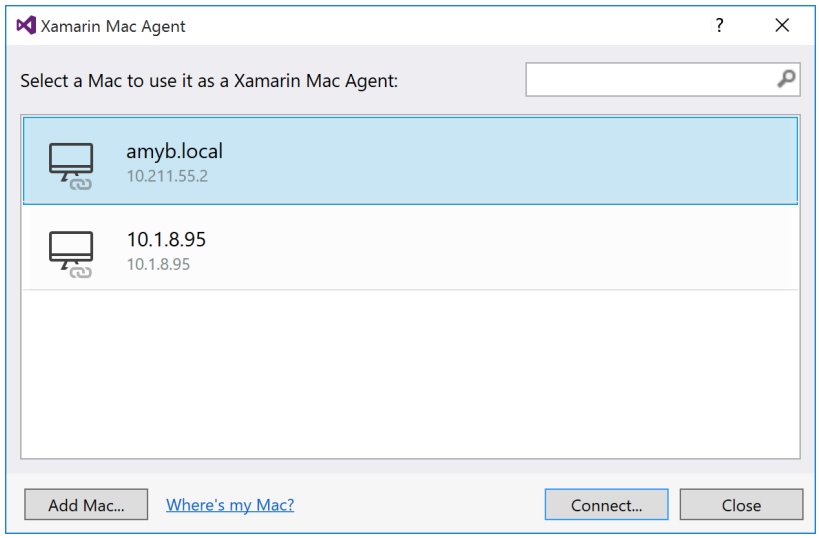



Figura 3.22. Elenco degli host rilevati da *Xamarin Mac Agent*

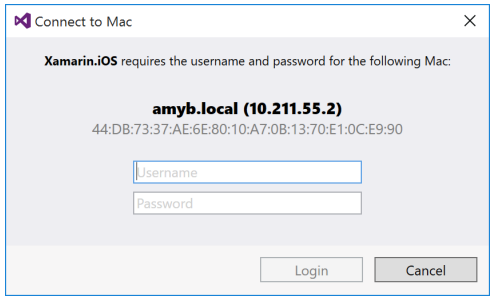



Figura 3.23. Finestra di login sul Mac

Ogni macchina presente nell'elenco, sia collegata che non, presenta un menu contestuale che consente le operazioni di *Connect*, *Disconnect* o *Forget the Mac* (quest'ultima nel caso in cui si scelga di dimenticare le credenziali eventualmente memorizzate).

In alcune circostanze, è possibile che non venga rilevato l'host anche se presente nella stessa rete oppure qualora stia in un'altra rete ma sia raggiungibile tramite IP pubblico. In questi casi si può procedere manualmente all'aggiunta del Mac recandosi alla stessa finestra di selezione dell'host *Xamarin Mac Agent* (fig. 3.22) e facendo click sul bottone *Add Mac...* per inserire l'indirizzo IP del Mac. Anche in questo caso, dopo aver fatto click su *Login*, *Visual Studio* sarà connesso con la macchina Mac tramite SSH e memorizzerà l'host come una macchina nota.

Terminata la procedura di collegamento con l'host Mac, è possibile quindi testare l'applicazione creata seguendo quanto descritto nel paragrafo dedicato, eseguendola su un simulatore (avviato sull'host Mac) oppure direttamente su un dispositivo

Apple fisico (collegato all'host Mac) da configurare opportunamente per lo sviluppo. Nella trattazione del caso di studio, anche per iOS, dopo un iniziale testing su simulatore, si è seguita la strada dello sviluppo su dispositivo fisico.

### Device provisioning

Nel caso di iOS, il testing su dispositivo fisico è più complesso rispetto ad Android; infatti non basta solamente collegare il dispositivo al computer abilitando la modalità sviluppatore, ma occorre registrarsi ad Apple come tale, crearsi un proprio profilo e i propri certificati relativi ad uno specifico device. In altre parole, il dispositivo deve essere *"provisioned"* e Apple deve essere informata che esso verrà usato come strumento di testing.

Lo schema in figura 3.24 (sezione cerchiata) mostra gli steps per configurare un dispositivo Apple per lo sviluppo e per la distribuzione, ma quest'ultima non è oggetto di questa trattazione.

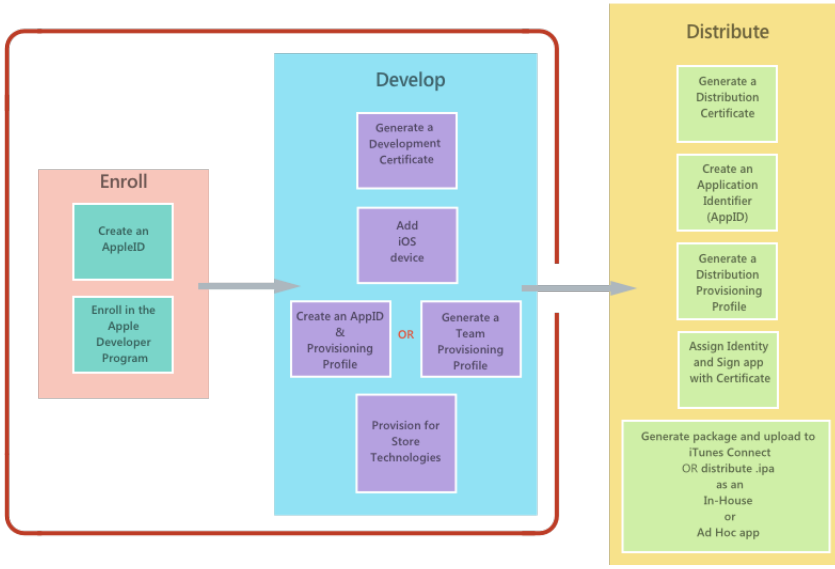



Figura 3.24. *Provisioning diagram* per lo sviluppo su dispositivo Apple fisico

Prima di iniziare la distribuzione dell'applicazione su un dispositivo, è necessario disporre di un abbonamento attivo per il programma *Apple's Developer Program* o, in alternativa, utilizzare l'approccio *Free provisioning* che consente la distribuzione della propria app senza far parte di un programma di sviluppo Apple. Proseguendo la trattazione seguendo la prima alternativa, occorre innanzitutto disporre

di un *Apple ID* e poi seguire le indicazioni per la registrazione al portale *Apple Developer*.

Ogni applicazione che viene eseguita su un dispositivo deve includere un insieme di metadati (o *thumbprint*), che contengono informazioni sull'applicazione stessa e sullo sviluppatore. Apple utilizza questo *thumbprint* per assicurarsi che l'applicazione non venga modificata quando la si esegue sul dispositivo. Ciò è ottenuto richiedendo agli sviluppatori di registrare il loro *Apple ID* come sviluppatore, di impostare un *App ID*, richiedere un certificato e registrare il dispositivo su cui verrà distribuita l'applicazione. Durante la distribuzione di un'applicazione, inoltre, sul dispositivo iOS viene installato anche un *provisioning profile* per verificare le informazioni con cui è stata firmata l'applicazione da Apple in fase di build. *Provisioning profile* e *thumbprint* sono utilizzati insieme per determinare se un'applicazione può essere distribuita su un dispositivo, controllando chi (certificati), cosa (*App ID* individuale) e dove (devices). Questi passaggi assicurano che tutto ciò che viene creato o utilizzato durante il processo di sviluppo, incluse le applicazioni e i dispositivi, può essere ricondotto a un account *Apple Developer*.

Analizzando nel dettaglio i passi operativi che portano alla completa configurazione degli strumenti per lo sviluppo su dispositivo fisico, il primo di essi consiste nel richiedere un certificato di sviluppo da parte di Apple. Esso, con le relative chiavi associate, è fondamentale per uno sviluppatore iOS poiché stabilisce la propria identità con Apple, consentendo di realizzare una cosa simile a mettere la propria firma digitale sulle proprie applicazioni. Apple fornisce due modi per fare ciò: via Xcode oppure manualmente, accedendo alla sezione dedicata del portale *Apple Developer*. Nel seguito è descritto il metodo manuale.

È importante notare che è possibile avere solo due certificati di sviluppo validi in qualsiasi momento; se ne servono altri bisogna revocarne uno esistente. Qualsiasi macchina che utilizza un certificato revocato non sarà in grado di firmare alcuna applicazione.

Effettuare innanzitutto il login al portale *Apple Developer* ed accedere alla sezione *Certificates, Identifiers & Profiles*; selezionare la scheda *Certificates* dalla colonna *iOS Apps*. Dunque, premere il bottone + per creare un nuovo certificato. Selezionare l'opzione *iOS App Development* come tipo di certificato (fig. 3.25) e proseguire.

Richiedere una *Certificate Signing Request*, che verrà caricata per generare manualmente il certificato. A tale scopo, avviare il *Keychain Access* sul Mac. Dal menu principale selezionare *Certificate Assistant* e poi *Request a Certificate from a Certificate Authority*. Compilare le informazioni richieste per il certificato e selezionare l'opzione *Save to disk*.

Ritornare al portale *Apple Developer*, nel punto in cui si era rimasti, e caricare la richiesta di certificato (fig. 3.26). Esso, se non si dispone dei privilegi di amministratore, dovrà essere approvato da un amministratore. Una volta che il certificato è stato approvato, scaricarlo dal portale.

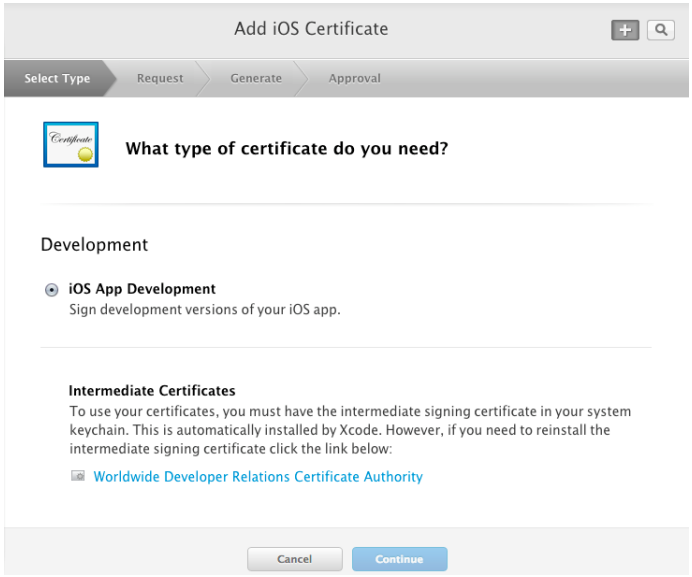



Figura 3.25. Selezione del tipo di certificato Apple

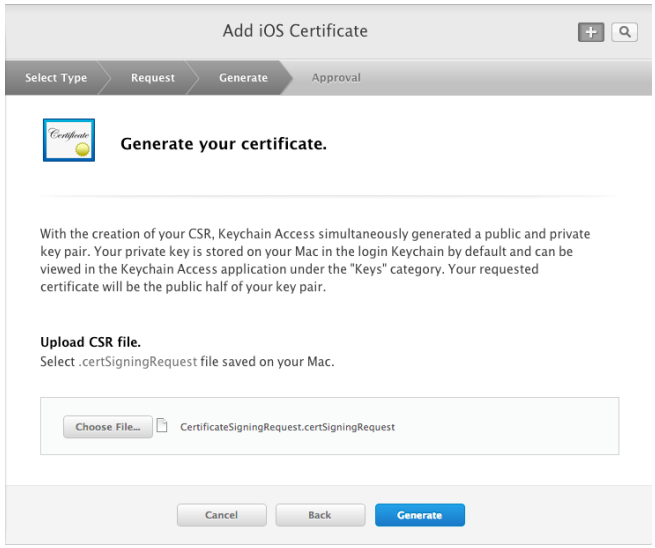



Figura 3.26. Caricamento del certificato sul portale Apple

Infine fare doppio click sul certificato scaricato per avviare il *Keychain Access* e aprire il pannello *My Certificates*, mostrando i nuovi certificati e la chiave privata associata. Al termine della procedura vi dovrà essere piena corrispondenza tra le chiavi pubbliche associate ai certificati e memorizzate sul proprio profilo del portale *Apple Developer* e le chiavi private memorizzate sul computer Mac.

Ora che, in possesso di certificato di sviluppo, è stata stabilita la propria identità con Apple è necessario creare un *provisioning profile* in modo da poter distribuire l'applicazione a un dispositivo fisico. Il dispositivo deve eseguire una versione di iOS supportata da Xcode, per cui potrebbe essere necessario aggiornare il dispositivo, Xcode o entrambi.

Sul Mac, avviare Xcode, collegare il dispositivo via USB e selezionare *Devices* dal menu *Windows*. Dall'elenco dei dispositivi, selezionare quello desiderato e copiare negli appunti la relativa stringa *Identifier* (fig. 3.27).

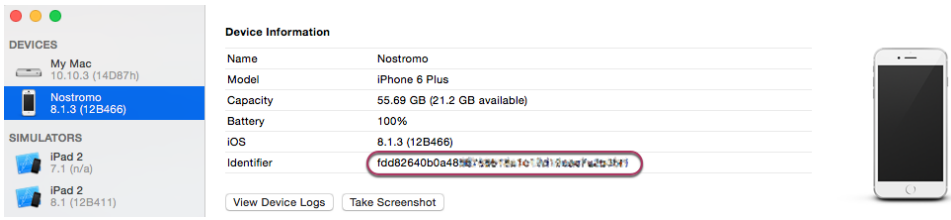



Figura 3.27. *Identifier* del dispositivo Apple

Tramite browser, accedere al portale *Apple Developer*, fare click su *Certificates, Identifiers & Profiles*, entrare in *Devices*, poi fare click sul bottone + per aggiungere il nuovo dispositivo, dandogli un nome e incollando l'*Identifier* copiato nel campo *UUID*. Terminare la registrazione, rivedendo le informazioni inserite, e fare click su *Register* (fig. 3.28).

Dopo aver aggiunto il dispositivo al portale, è necessario creare un *provisioning profile* e associare il dispositivo ad esso. Come per il certificato di sviluppo, Apple offre due modi per fare ciò: tramite Xcode, creando un profilo di team, oppure manualmente. Questa volta, viene descritto nel seguito il metodo via Xcode.

I *team provisioning profile* possono essere creati e gestiti tramite Xcode automaticamente. A tale scopo, creare un'applicazione iOS fittizia in Xcode, navigando in *File* > *New* > *Project*, e selezionare dall'elenco a discesa il proprio team. Questo genererà automaticamente un nuovo *provisioning profile*, come visualizzato nella sezione *Signing* della scheda *General* (fig. 3.29). Se tale generazione non avviene, potrebbe essere necessario selezionare la casella *Automatically Manage Signing* dalla stessa sezione.

Xcode aggiunge automaticamente un *App ID*, i certificati, il proprio team e alcune funzionalità di base al *provisioning profile*. Un dispositivo viene considerato

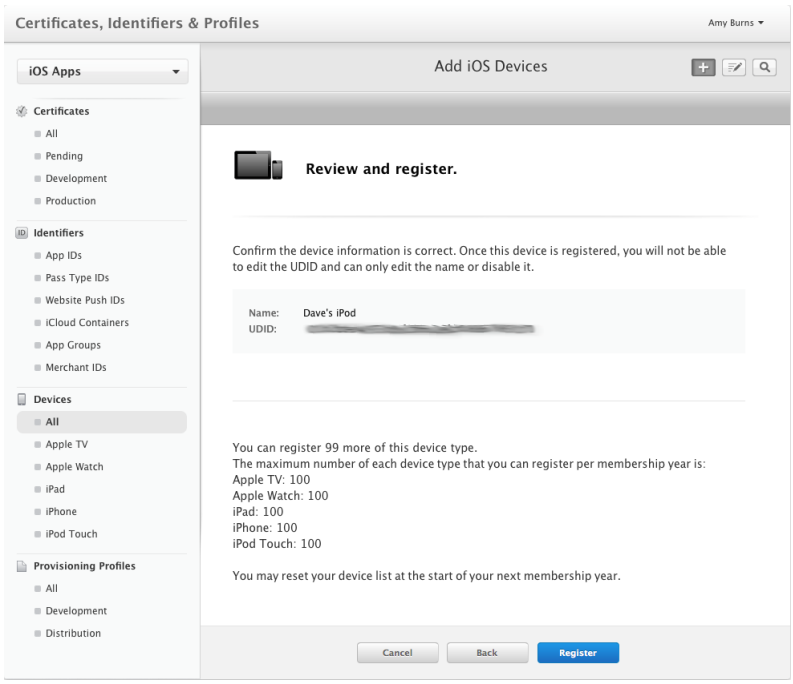



Figura 3.28. Riepilogo informazioni del dispositivo e registrazione

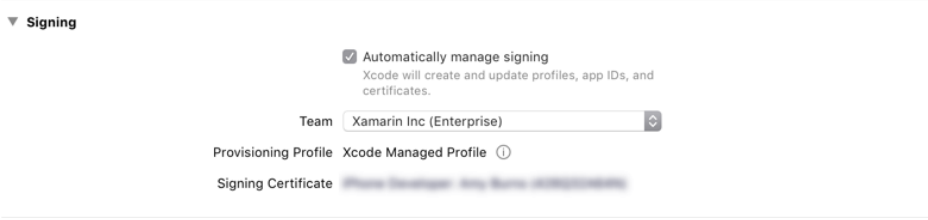



Figura 3.29. Generazione automatica del *provisioning profile*

*provisioned* quando il *provisioning profile* contenente le informazioni sul dispositivo è installato sul dispositivo stesso.

A questo punto, questa fase lunga e particolarmente complessa dovrebbe essere terminata e l'applicazione è pronta per essere distribuita sul dispositivo. Per effettuare questa operazione, verificare che *Visual Studio* possa visualizzare l'identità e i profili corretti, ed eseguire l'applicazione sul dispositivo seguendo le indicazioni generali fornite in uno dei precedenti paragrafi.

# Capitolo 4

# Il caso di studio OpenShop

## 4.1 Generalità

Come banco di prova per capire il funzionamento del framework scelto e testarne le potenzialità nello sviluppo multipiattaforma, è stato deciso di realizzare un prototipo di applicazione *Xamarin* derivante dal *porting* di un'applicazione nativa esistente.

L'app in questione è **OpenShop.io** (`http://openshop.io`), una piattaforma di e-commerce mobile open source sviluppata dalla *Business Factory s.r.o.*, azienda ceca coinvolta nell'innovazione tecnologica dell'*internet marketing*. Dopo una ricerca sul web, questa è stata ritenuta una buona base su cui lavorare poiché presenta tutte le funzioni più comuni individuabili in una qualunque app di questo ambito, oltre ovviamente al fatto di rendere il proprio codice Android e iOS accessibile liberamente senza doversi preoccupare anche della scrittura del codice nativo, compito che avrebbe esulato dagli obiettivi della presente tesi.


La stessa azienda sviluppatrice, insieme con il codice sorgente, ha messo a disposizione anche un server per svolgere i test a cui l'app si collega per leggere i dati d'esempio e per effettuare le richieste fittizie (prodotti, pagamenti, acquisti, login, ecc.). Richieste e risposte vengono veicolate tramite un'apposita API, la cui documentazione si trova sul sito `http://docs.bfeshopapiconnector.apiary.io`, usata per la comunicazione tra il client e il server di back-end nel formato JSON via HTTPS. Per questo, anche nella versione multipiattaforma realizzata si è sfruttato lo stesso server, evitando così di doverne mettere in piedi un altro ed in modo da avere gli stessi dati a disposizione.

Lo sviluppo della soluzione multipiattaforma *Xamarin* è stato incentrato solo sui progetti Android e iOS e non anche su Windows Phone. La scelta di limitarsi a queste due sole piattaforme è dovuta non tanto al fatto che i progetti nativi *OpenShop* fossero stati sviluppati solo per esse, quanto al fatto che le stesse costituiscono i due sistemi operativi per dispositivi mobili attualmente più diffusi al mondo, come già ampiamente analizzato nei capitoli iniziali.

## 4.2 Subset di funzionalità da implementare

Ai fini della realizzazione del progetto multipiattaforma *Xamarin*, è stata condotta una preliminare analisi dell'applicazione originaria volta a stilare una lista completa di tutte le funzionalità offerte. Di questo elenco ne è stato realizzato un *subset*, contenente quelle ritenute più interessanti che valeva la pena studiare e implementare nell'app condivisa, essendo la buona riuscita di un *porting* complessivo l'obiettivo primario di questa tesi al termine del lavoro.

Le funzionalità selezionate per essere implementate sono le seguenti:

- Scelta della lingua, mostrata solo alla prima esecuzione dell'app dopo l'installazione;
- Menu di navigazione, con l'elenco delle categorie di prodotti presenti da visualizzare e indicazione dell'utente loggato;
- Elenco dei prodotti di una determinata categoria, visualizzati a scelta dell'utente come lista (1 item per riga) o griglia (2 item per riga);
- Filtro prodotti (per colore, taglia o fascia di prezzo), per mostrare in ogni categoria solo un sottoinsieme di essi;
- Per ciascun prodotto, possibilità di scelta del colore e della taglia desiderati;
- Aggiunta di un prodotto al carrello acquisti;
- Elenco dei prodotti consigliati, mostrati nella scheda di ciascun prodotto;
- Cancellazione di un prodotto dal carrello;
- Nel carrello, scelta del metodo di pagamento desiderato;
- Possibilità di modifica dei dettagli utente (indirizzo, contatti, ecc.) nel carrello, prima di procedere con l'acquisto;
- Terminazione dell'ordine;
- Visualizzazione del profilo utente con le relative informazioni di registrazione;

- Login via e-mail;
- Login via *Facebook*;
- Logout;
- Possibilità di registrazione di un nuovo utente.

Seguono ora le altre funzionalità presenti nell'applicazione originale, ritenute non fondamentali ai fini del progetto e quindi non implementate:

- Scelta della categoria di prodotti da visualizzare tramite barra di ricerca;
- Possibilità di ordinare i prodotti nella categoria in base ad un criterio a scelta;
- Aggiunta di un prodotto ai preferiti;
- Segnalazione di un prodotto ad un amico su *Facebook*;
- Possibilità di modificare i dettagli di un prodotto (colore e taglia) nel carrello dopo averlo già scelto;
- Possibilità di inserire un codice di sconto prima dell'acquisto;
- Nella scheda relativa al profilo utente, possibilità di modificare le sue informazioni di registrazione, cambiare la password, visualizzare lo storico ordini e le filiali fisiche vicine;
- Impostazioni, policy, termini e condizioni.

Essendo questa una applicazione demo, che si collega ad un server di test, tutte le richieste, in particolare la registrazione e il login di un utente e l'evasione di un ordine, sono fittizie.

## 4.3 Progettazione e implementazione

Dall'analisi preliminare effettuata sull'applicazione nativa esistente emerge la necessità di sviluppare un prototipo di applicazione mobile, al fine di valutare se la tecnologia multipiattaforma individuata abbia raggiunto un grado di maturazione tale da poter essere utilizzata per progetti più complessi. Il prototipo dovrà replicare, il più fedelmente possibile, le funzionalità presenti nell'applicazione nativa esistente per il subset individuato come più significativo, sfruttando l'approccio multipiattaforma di *Xamarin.Forms* fin dove possibile, anche nei casi in cui probabilmente l'utilizzo di codice nativo specifico avrebbe permesso di superare alcuni ostacoli e condotto a risultati migliori.

L'implementazione dell'applicazione è stata effettuata in ambiente *Windows 10* utilizzando l'IDE *Visual Studio 2017*, integrato con il package *Xamarin.Forms* aggiornato all'ultima versione stabile tramite il gestore di pacchetti *NuGet* di Visual Studio. Per quanto riguarda le piattaforme supportate sono state mantenute le seguenti versioni minime: Android 4.0.3 (livello API 15) e iOS 8.0.

La soluzione generata, dal nome *openshop*, contiene quindi tre progetti come mostrato in figura 4.1:

- *openshop.Droid* è il progetto specifico generato per Android;
- *openshop.iOS* è il progetto specifico generato per iOS;
- *openshop* è il progetto comune nel quale è stato scritto il codice condiviso.

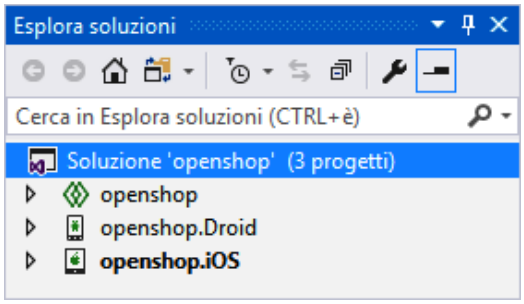



Figura 4.1. Progetti della soluzione *OpenShop* in Visual Studio

Come analizzato nel precedente capitolo, le modalità per condividere il codice in un'applicazione mobile multipiattaforma sono due: lo *Shared Project* o le *Portable Class Libraries*. La soluzione implementata ha seguito la strada dello *Shared Project*. Tale scelta permette di mantenere il codice, scritto con *Xamarin.Forms*, comune a entrambe le piattaforme in un unico progetto e senza alcun riferimento al resto della soluzione. Il codice è compilato come parte di ogni progetto referenziante e potrebbe eventualmente includere direttive di compilazione per implementare al meglio le funzionalità specifiche delle diverse piattaforme.

### 4.3.1 Componenti e librerie di terze parti

Durante lo sviluppo dell'applicazione, a volte è capitato di trovarsi davanti a difficoltà oggettive nel dover implementare una certa funzionalità in modo semplice e con i soli strumenti che *Xamarin.Forms* rende disponibili di default. In questi casi, si è fatto ricorso a librerie e componenti aggiuntivi pubblicati nell'area open source ufficiale *Xamarin Component Store* (`https://components.xamarin.com`) e

accessibili direttamente da *Visual Studio* tramite il già citato gestore di pacchetti *NuGet*. Essi, se scaricati dallo *store*, è possibile che oltre ai file di libreria strettamente necessari contengano anche degli utili esempi di utilizzo. Tra i numerosi componenti presenti, a seconda delle loro caratteristiche, se ne trovano alcuni a pagamento e altri disponibili gratuitamente. Tra le categorie coperte troviamo:

- Componenti per la gestione della *user interface*;
- Componenti di supporto per servizi *cloud* (*Amazon*, *Microsoft Azure*, ecc.);
- Componenti per l'interazione con i *social network* (*Pinterest*, *Facebook*, ecc.);
- Componenti per lo sviluppo di temi grafici e giochi.

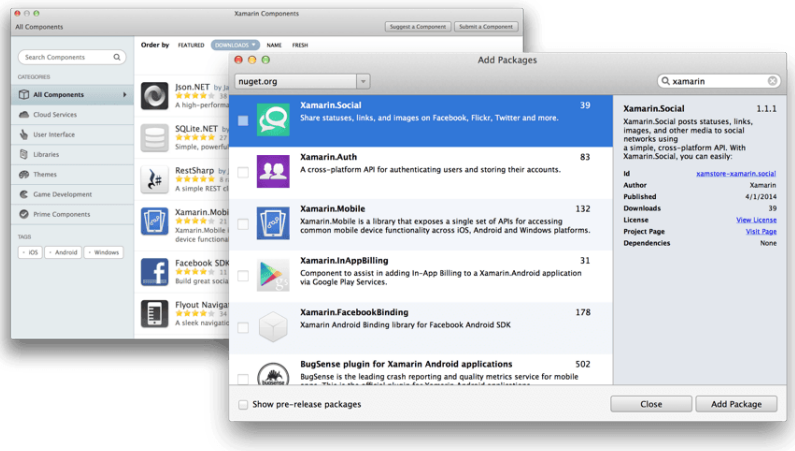



Figura 4.2. *Xamarin Component Store* e *NuGet package manager*

Ottenere componenti in un progetto è abbastanza semplice; seguire il percorso *Tools* > *NuGet Package Manager* > *Manage NuGet packages for solution* oppure, in *Solution Explorer*, fare click con il tasto destro del mouse sulla soluzione e selezionare *Manage NuGet packages*. Si aprirà una finestra che mostra i pacchetti disponibili e ne permette una ricerca mirata; selezionandone uno sarà possibile installarlo facendo click sul bottone *Install* (fig. 4.3). In questa fase inoltre, qualora si desideri installare il componente solo in uno specifico progetto (Android o iOS), sarà possibile indicarlo. Il componente verrà scaricato localmente, associato alla soluzione o al progetto selezionato e infine aggiunto come riferimento per essere immediatamente utilizzabile.

Nel caso di studio analizzato, per esempio, si è fatto uso di componenti esterni per la serializzazione in JSON degli oggetti utilizzati nelle query verso il server e per la gestione del login tramite *Facebook*.

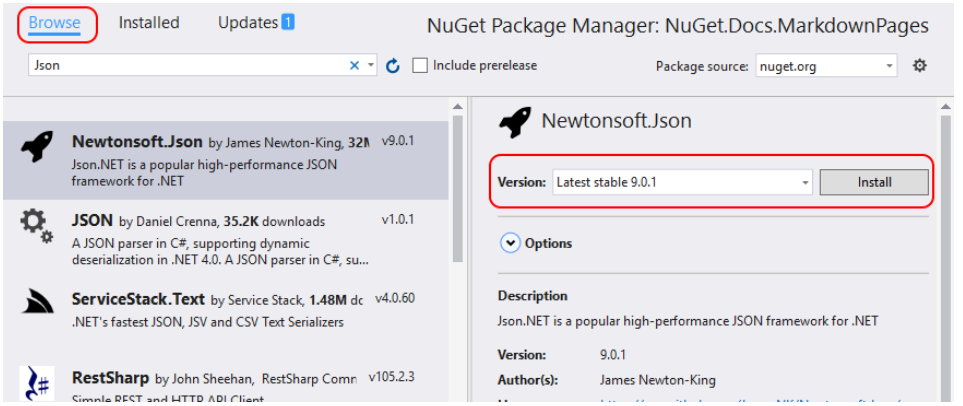


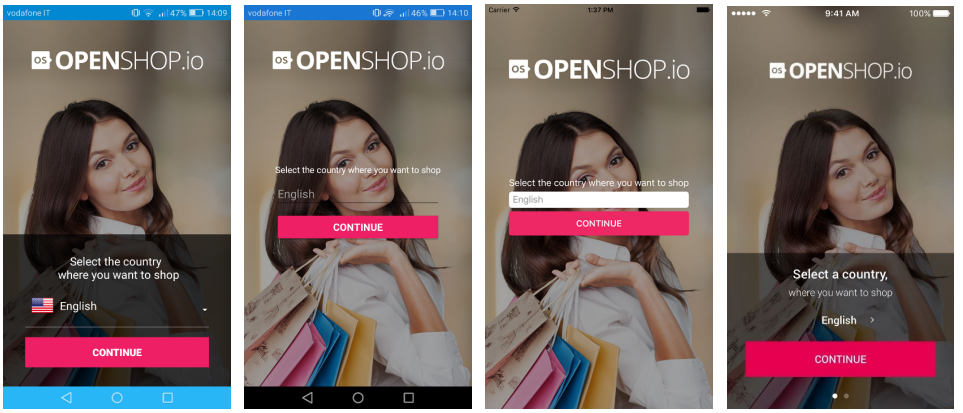

Figura 4.3. Ricerca pacchetti nel *NuGet package manager*

### 4.3.2 Progetto finale

Seguono ora gli *screenshot* di tutte le pagine dell'app implementate, nelle due versioni Android e iOS a confronto con le corrispondenti native. I progetti Android sono stati eseguiti su device *Huawei P8 Lite (Android 6.0)*, mentre quelli iOS su simulatore *iPhone 7 Plus (iOS 10.3)*.

#### Pagina di scelta della lingua

La pagina si presenta all'avvio dell'applicazione, solamente al primo accesso dopo l'installazione, e permette di scegliere la lingua desiderata.

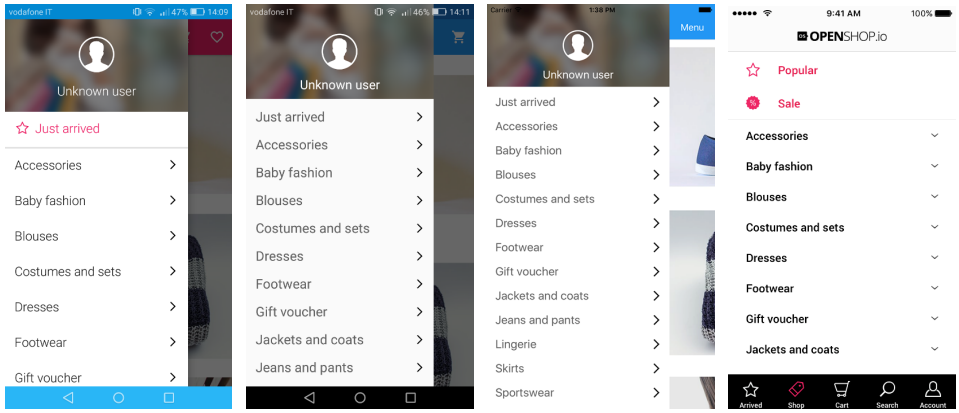



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.4. *OpenShop*: pagina di scelta della lingua

### Menu di navigazione

Il menu di navigazione, accessibile in ogni momento, contiene la lista delle categorie di prodotti e, a seguito di selezione, delle possibili sottocategorie; inoltre presenta il nome dell'utente loggato con possibilità di accedere ai dettagli dello stesso.

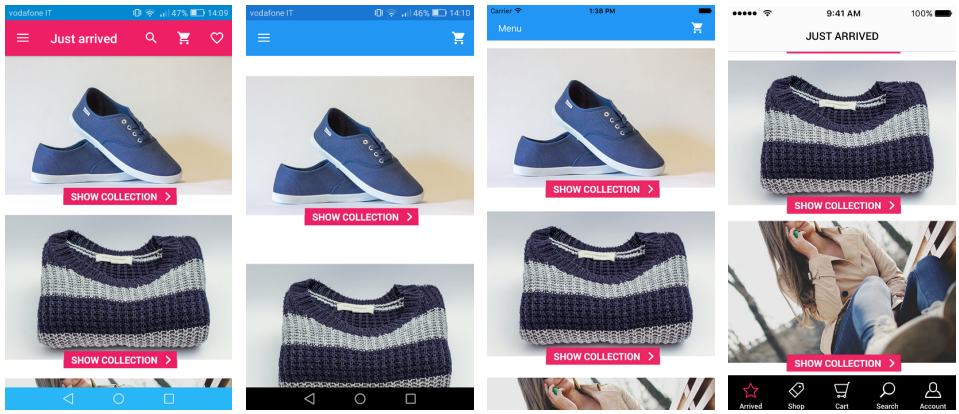



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.5. *OpenShop*: menu di navigazione

### Pagina degli ultimi arrivi

La pagina pubblicizza le categorie degli ultimi arrivi ed è quella di default che si presenta ad ogni avvio dell'applicazione.

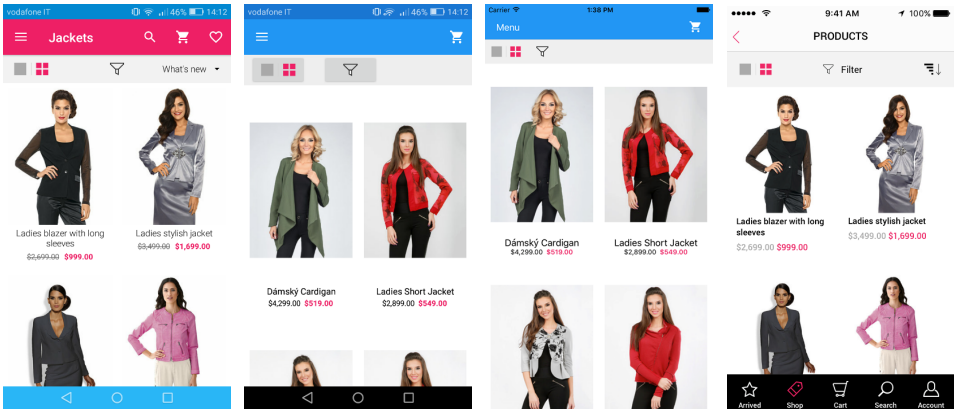



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.6. *OpenShop*: pagina degli ultimi arrivi

### Pagina della categoria di prodotti

La pagina mostra l'elenco dei prodotti di una determinata categoria, con possibilità di essere visualizzati a scelta dell'utente come lista o griglia.

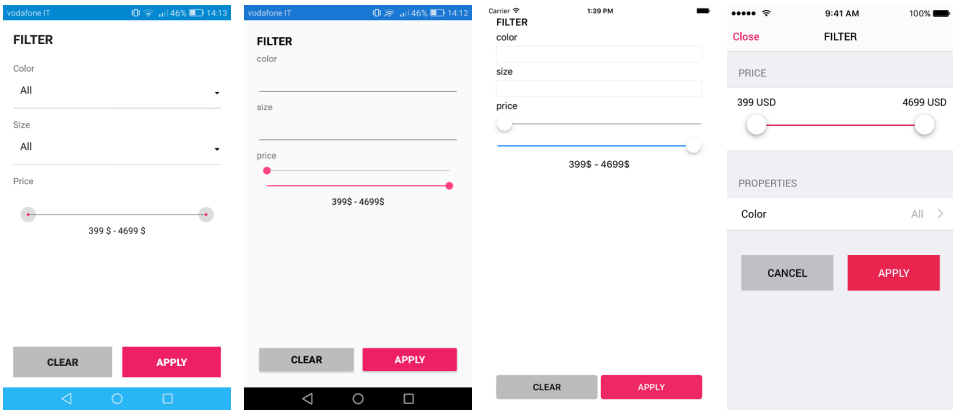



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.7. *OpenShop*: pagina della categoria di prodotti

### Pagina del filtro prodotti

La pagina, aperta a seguito di touch sul relativo bottone presente in ogni categoria, permette di restringere la visualizzazione della categoria stessa scegliendo una taglia, un colore o una fascia di prezzo specifica.

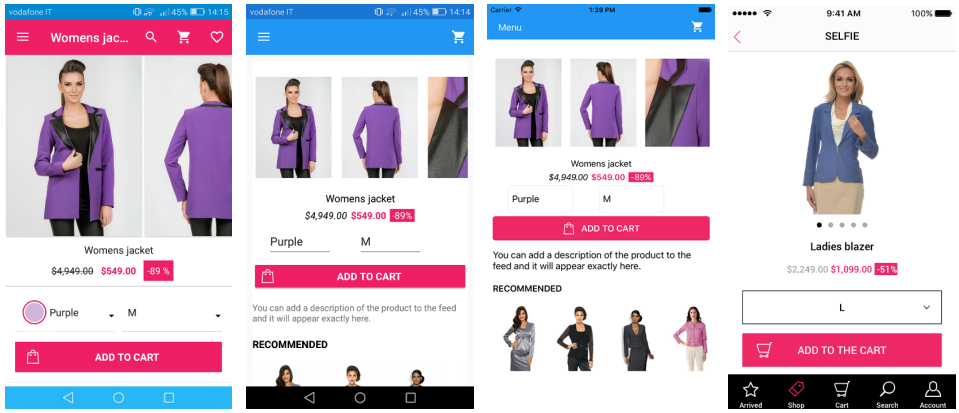



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.8. *OpenShop*: pagina del filtro prodotti

### Pagina del singolo prodotto

La pagina riguarda il singolo prodotto selezionato da una specifica categoria. Essa contiene le immagini del prodotto, le taglie e i colori disponibili, il prezzo (intero o scontato), il bottone di aggiunta al carrello e i prodotti raccomandati.

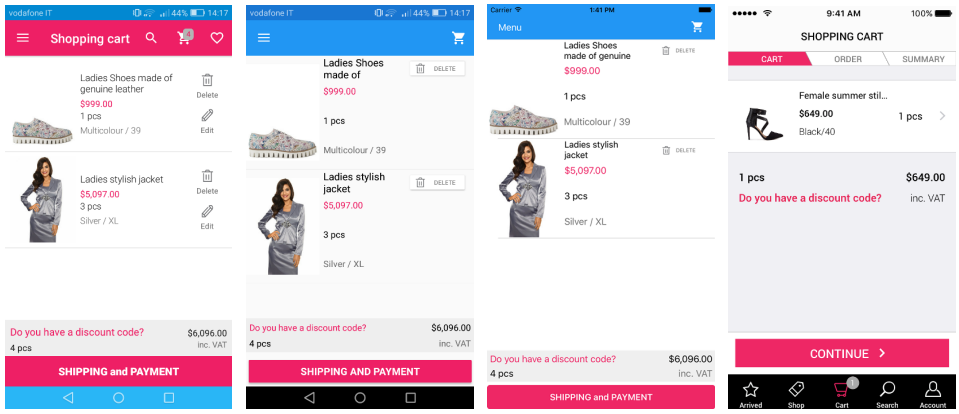



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.9. *OpenShop*: pagina del singolo prodotto

### Pagina del carrello

La pagina, accessibile solo se l'utente è loggato, contiene i prodotti in attesa di essere acquistati, con possibilità di essere cancellati dal carrello stesso qualora non più desiderati.

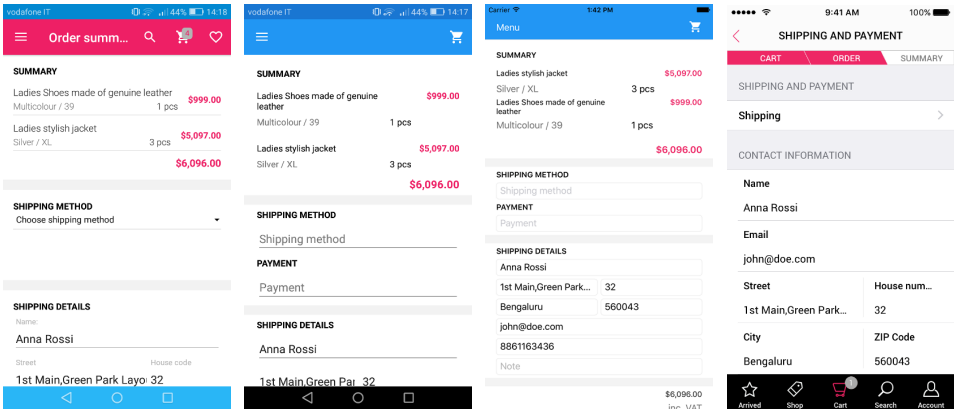



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.10. *OpenShop*: pagina del carrello

### Pagina di riepilogo dell'ordine

La pagina mostra il riepilogo dei prodotti già presenti nel carrello in attesa di essere acquistati, con possibilità di scelta del metodo di pagamento e modifica dei dettagli di spedizione, ed il bottone per terminare l'acquisto.

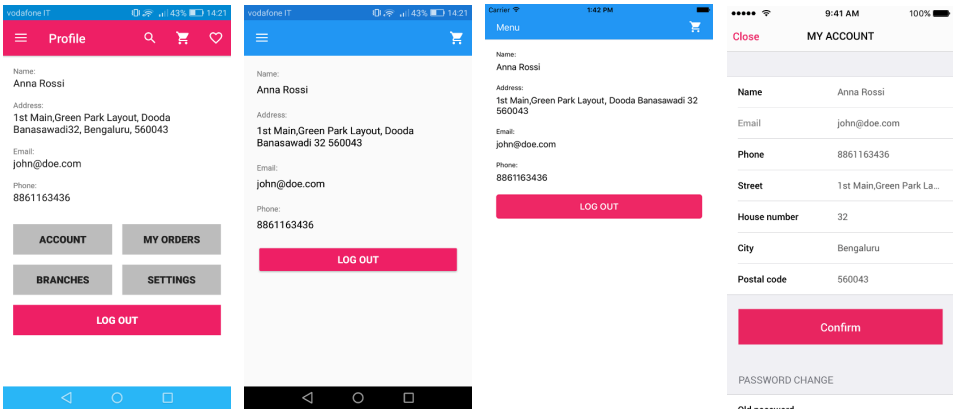



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.11. *OpenShop*: pagina di riepilogo dell'ordine

### Pagina del profilo utente

La pagina mostra i dati dell'utente (indirizzo, contatti, ecc.) e il bottone per il logout qualora lo stesso sia loggato, altrimenti non presenta altro se non il bottone per il login.

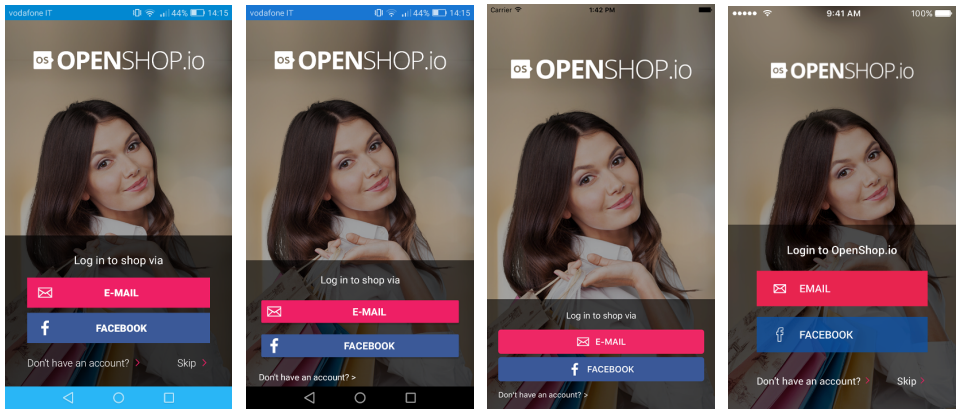



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.12. *OpenShop*: pagina del profilo utente

### Pagina di richiesta login

La pagina richiede il tipo di accesso desiderato (e-mail o *Facebook*). Essa viene mostrata tutte le volte che si tenta di accedere ad una funzione riservata ai soli utenti loggati (accesso al carrello, aggiunta del prodotto al carrello, info utente).

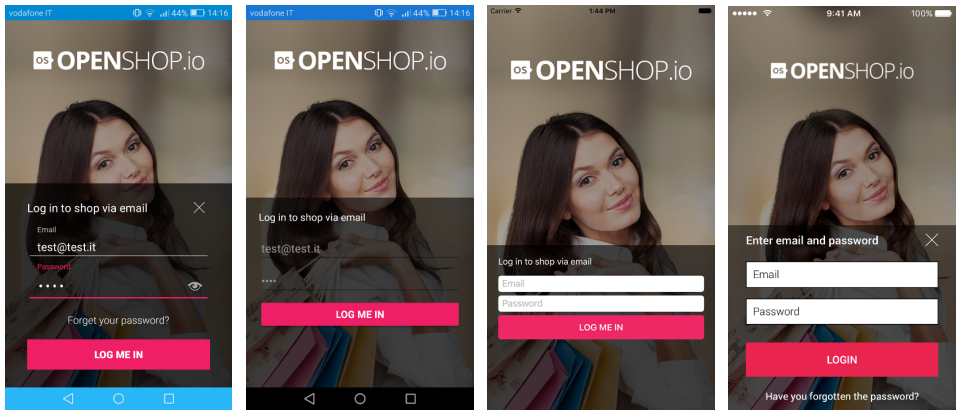



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.13. *OpenShop*: pagina di richiesta login

### Pagina di inserimento credenziali

La pagina, richiamata da quella di login a seguito di touch su apposito bottone, consente di inserire le credenziali necessarie per il login dell'utente.

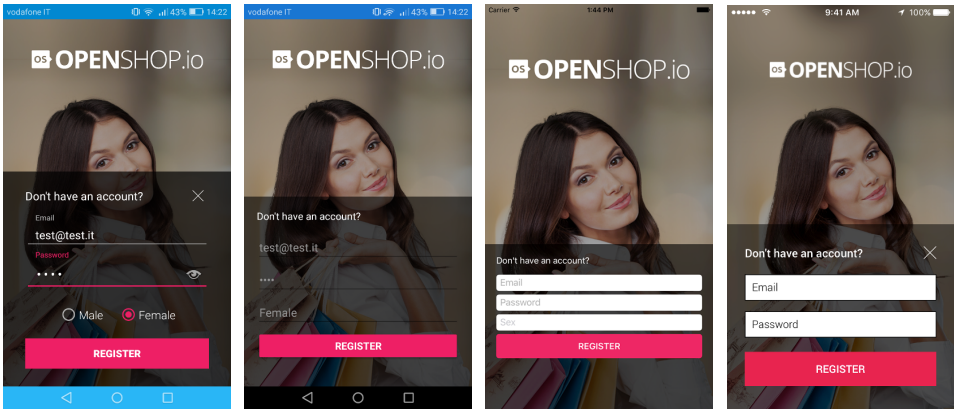



(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.14. *OpenShop*: pagina di inserimento credenziali

### Pagina di registrazione utente

La pagina, richiamata anch'essa da quella di login, permette di registrare un nuovo utente inserendo i propri dati minimali.




(a) Nativa Android (b) openshop.Droid (c) openshop.iOS (d) Nativa iOS

Figura 4.15. *OpenShop*: pagina di registrazione utente

Come facilmente intuibile dalle immagini precedenti, la versione nativa iOS, pur mantenendo le stesse funzionalità, differisce leggermente rispetto alle altre nell'aspetto grafico; ciò è dovuto al fatto che per realizzare il progetto condiviso *Xamarin* si è scelto di utilizzare come modello di riferimento la versione nativa Android.

# Capitolo 5

# Risultati ottenuti

## 5.1 Test e analisi di performance

L'attività di testing applicata al caso di studio sviluppato può essere considerata la fase finale del lavoro oggetto della presente trattazione; da essa si possono derivare alcune considerazioni importanti sull'approccio di sviluppo multipiattaforma e sul framework *Xamarin* rispetto alla tradizionale tecnica puramente nativa.

L'esecuzione dell'applicazione e i test finali sono stati eseguiti in modo manuale utilizzando i seguenti *device* fisici o simulati in ambiente desktop:

- *Huawei P8 Lite*, 5.0" (Android 6.0);
- *iPad 2*, 9.7" (iOS 9.3.5);
- simulatore *iPhone 7 Plus*, 5.5" (iOS 10.3).

Su tali dispositivi, nel complesso, l'applicazione nelle sue diverse versioni è risultata avere una corretta e fluida esecuzione in tutto il suo corso. Allo stesso modo, il risultato del processo di *rendering* dell'interfaccia grafica, in generale, è da considerarsi più che buono, anche se verosimilmente se si fosse estesa l'esecuzione su un numero maggiore di dispositivi alcuni problemi sarebbero potuti sorgere a causa delle diverse dimensioni dello schermo esistenti.

Al fine di analizzare le performance dell'applicazione sviluppata, in particolare per quanto riguarda la reattività all'esecuzione di una specifica funzionalità, sono stati innanzitutto individuati come campione i seguenti *tasks* ritenuti più significativi dei quali misurare il tempo di completamento:

A. Avvio dell'applicazione;

B. Caricamento della lista di prodotti nella pagina della relativa categoria;

C. Caricamento dei dati del singolo prodotto nella relativa pagina;

D. Aggiunta di un prodotto al carrello;

E. Caricamento dei dati del carrello nella relativa pagina;

F. Cancellazione di un prodotto dal carrello;

G. Caricamento dei dati riepilogativi dell'ordine creato nella relativa pagina;

H. Applicazione del filtro all'interno di una categoria di prodotti;

I. Accesso mediante login via e-mail.

Successivamente sono state prese come riferimento 30 esecuzioni distinte dell'applicazione durante le quali sono stati eseguiti gli stessi identici tasks elencati sopra (il lavoro è stato facilitato e reso più rapido dalla scrittura di uno *script* che in automatico ha simulato i *touch* sullo schermo). Su tali tasks è stato salvato il tempo impiegato a partire dal touch di impulso d'avvio sino alla sua completa terminazione. Gli stessi dati ottenuti dai progetti multipiattaforma Android e iOS sono poi stati messi a confronto con quelli prelevati dalla corrispondente app nativa esistente e sottoposti ad alcune analisi statistiche, raccolte nelle tabelle riepilogative di seguito riportate.

| **Task** | **Nativa Android** | | **openshop.Droid** | |
|---|---|---|---|---|
| | **Media** [ms] | **Varianza** | **Media** [ms] | **Varianza** |
| A | 1692,178 | 159,893 | 3317,966 | 522,527 |
| B | 743,448 | 548,206 | 824,157 | 190,603 |
| C | 374,497 | 162,597 | 405,758 | 67,226 |
| D | 912,040 | 606,968 | 1049,342 | 108,189 |
| E | 381,592 | 152,190 | 412,320 | 272,125 |
| F | 601,700 | 408,070 | 604,267 | 956,170 |
| G | 659,032 | 132,560 | 361,241 | 256,793 |
| H | 1398,040 | 721,638 | 386,297 | 410,209 |
| I | 931,326 | 946,956 | 1061,355 | 922,014 |

Tabella 5.1. Tempi di esecuzione dell'app *OpenShop* in Android

La colonna *Task* contiene gli stessi caratteri alfabetici con cui sono stati etichettati i tasks nell'elenco descrittivo contenuto in questo stesso paragrafo.

La colonna *Media* contiene il tempo medio di esecuzione e completamento di un task, espresso in millisecondi. In essa si possono notare leggere differenze tra la soluzione multipiattaforma e quella nativa utilizzata come modello, con alcune eccezioni, quale ad esempio il tempo eccessivo impiegato per l'avvio, causate anche

| Task | Nativa iOS | | openshop.iOS | |
|---|---|---|---|---|
| | **Media** [ms] | **Varianza** | **Media** [ms] | **Varianza** |
| A | 2259,893 | 426,568 | 2771,583 | 89,821 |
| B | 834,383 | 974,415 | 602,403 | 189,124 |
| C | 676,423 | 602,124 | 358,421 | 167,534 |
| D | 982,227 | 907,604 | 1011,179 | 508,215 |
| E | 495,779 | 977,605 | 310,620 | 31,055 |
| F | 698,779 | 1001,312 | 695,412 | 816,375 |
| G | 1114,077 | 920,205 | 433,783 | 35,330 |
| H | 1269,670 | 994,705 | 559,058 | 1,071 |
| I | 878,337 | 666,784 | 785,511 | 609,081 |

Tabella 5.2. Tempi di esecuzione dell'app *OpenShop* in iOS

dal fatto che si tratta di un *porting* semplificato di una complessa applicazione già esistente.

La colonna *Varianza* contiene, invece, la misura della variabilità dei valori, ovvero di quanto essi si discostino quadraticamente dalla media: minore è la varianza e maggiore è la concentrazione dei dati attorno al valore medio, e viceversa. Da questo ne deriva che i test effettuati hanno mostrato una discreta instabilità più o meno accentuata dovuta, oltre che al diverso livello di complessità dell'operazione, anche e soprattutto a operazioni effettuate in rete verso il server che hanno reso decisamente variabile da un'esecuzione all'altra il tempo medio misurato.

È importante far presente che i tempi prelevati dall'applicazione nativa iOS, qui riportati per completezza, sono da ritenersi puramente indicativi e meno rilevanti ai fini della corrente analisi. Le versioni native Android e iOS da cui il lavoro di tesi è partito, infatti, pur essendo identiche in termini di funzionalità, non lo sono altrettanto dal punto di vista dell'aspetto e degli elementi grafici di dettaglio. Il progetto di *porting* realizzato, essendo stato condotto per scelta prendendo come applicazione target di partenza la versione Android, ha quindi questa versione come fedele modello, tanto nell'aspetto grafico quanto nelle modalità con cui le funzionalità stesse sono offerte. Il carrello acquisti e la relativa pagina di conferma ordine, tanto per fare un esempio, nelle due versioni native di partenza sono implementati in maniera leggermente diversa e di conseguenza i tempi non possono essere paragonati esattamente.

## 5.2 Problematiche riscontrate

Durante lo sviluppo del progetto sono state riscontrate alcune problematiche legate principalmente all'uso di *Xamarin.Forms* come strumento di sviluppo per due

piattaforme molto diverse tra loro.

Un primo problema sorto è stato quello della replicazione fedele di tutte le caratteristiche di cui dispone l'interfaccia grafica dell'applicazione nativa. Le componenti messe a disposizione da *Xamarin.Forms*, infatti, sono pensate per essere il più possibile adattabili alle differenti piattaforme, senza essere troppo specifiche; una riproduzione fedele di tutti i dettagli di interfaccia spesso non è possibile a causa delle differenti linee guida utilizzate nelle diverse piattaforme. Per le direttive grafiche non previste in uno dei sistemi operativi considerati la sua implementazione avrebbe richiesto un lavoro aggiuntivo di documentazione per cercare una soluzione, qualora possibile, con uno sviluppo indipendente platform-specific o l'ausilio di apposite librerie di terze parti. In alcune situazioni, però, tale soluzione avrebbe ridotto la manutenibilità del codice, poiché in caso di modifiche future sarebbe stato necessario intervenire singolarmente su entrambe le implementazioni. Nel caso di studio implementato, tuttavia, considerando l'obiettivo primario prefissato, è stato sfruttato tutto quanto fornito da *Xamarin.Forms*, implementando tutte le parti realizzabili in comune e tralasciando solo alcuni piccoli dettagli; il risultato finale è stato ritenuto più che sufficiente ed è da considerarsi in generale ottimo.

Un altro palese problema emerso in fase di esecuzione e testing del progetto sviluppato è la sua eccessiva lentezza nell'avvio rispetto all'applicazione nativa, sui dispositivi Android specialmente, che rappresenta decisamente un aspetto negativo. Dalla tabella riepilogativa riportata nella sezione precedente, infatti, emerge che la versione Android dell'app realizzata impiega in media circa 3,3 secondi ad avviarsi contro 1,7 secondi della sua corrispondente versione nativa. Dopo una attenta ricerca in rete è emerso che il problema, ipoteticamente attribuito all'inizio soltanto a qualche errore di sviluppo, in realtà è ben noto e presente in tutte le applicazioni realizzate con *Xamarin.Forms*, conseguenza di caricamenti runtime o di numerose librerie. Posto che il problema c'è e continua ad esserci, alcuni siti dedicati all'argomento [24, 25, 26] (a cui si rimanda per un maggiore approfondimento) spiegano che, pur non essendo possibile risolverlo del tutto, si potrebbe perlomeno intervenire con qualche accortezza per incrementare le performance dell'applicazione stessa. In realtà, anche la stessa *Xamarin* è consapevole di questo e recentemente è intervenuta includendo alcuni miglioramenti all'interno degli ultimi aggiornamenti di *Xamarin.Forms* rilasciati. Tra i suggerimenti consigliati se ne riportano alcuni:

- Scegliere il layout corretto, ottimizzandone le prestazioni (specialmente per la view di avvio, in questo caso, per la quale è consigliato uno *splash screen* in modo da non bloccare il main thread);
- Abilitare il compilatore XAML (disabilitato di default per garantire *backwards compatibility*);
- Ridurre i *bindings* non necessari;

- Ridurre la dimensione del *visual tree*, mantenendo a livello di layout gli elementi strettamente necessari o non nascosti;
- Ottimizzare il *rendering*, utilizzando un modello personalizzato o eseguendo operazioni massive (*bulk*).

# Capitolo 6

# Conclusioni

Giunti al termine della presente trattazione, può essere considerato raggiunto l'obiettivo primario, ovvero quello di studiare il framework *Xamarin* per lo sviluppo multipiattaforma e trarne i suoi punti forti e deboli osservandolo all'opera su un progetto reale.

La realizzazione dell'applicazione oggetto di studio, nelle sue funzionalità implementate seppur talvolta semplificate, si è conclusa con successo senza dover ricorrere all'uso di codice platform-specific. Nel corso della sua implementazione, una volta capito come utilizzare il framework e come esso lavora studiandolo inizialmente in esecuzione su un'applicazione d'esempio minimale, non si sono incontrate particolari difficoltà. Lo sviluppo, risultato essere quindi al 100% multipiattaforma, conferma la possibilità di realizzare applicazioni anche non banali da parte di sviluppatori che hanno conoscenze limitate delle singole piattaforme o dei singoli linguaggi. Questo, se da un lato è positivo, dall'altro mette in rilievo l'impossibilità, con una soluzione completamente multipiattaforma, di apprezzare e sfruttare appieno le caratteristiche di ogni singola piattaforma; ad esempio, la volontà di ottenere delle interfacce grafiche personalizzate per ogni piattaforma oppure la forte necessità di utilizzare funzionalità native potrebbe portare ad escludere a priori l'uso di questo framework. In realtà, *Xamarin* fornisce la possibilità di intervenire singolarmente sui progetti specifici, anche per la definizione delle interfacce grafiche, rendendo ugualmente possibile l'utilizzo del framework con un mix di codice condiviso e codice specifico. Per gli scopi prefissati, tuttavia, il risultato ottenuto replica in maniera più che buona la *user experience* dell'app nativa.

Interessante è il fatto di poter integrare le funzionalità di base offerte dalle librerie proprie del framework, che a dire il vero non sono molte, con funzionalità anche complesse facilmente realizzabili grazie a componenti aggiuntivi e librerie di terze parti create da chiunque ne abbia la volontà e la capacità, in un contesto totalmente *open source*. Nel progetto implementato, ad esempio, si sono rivelate

davvero molto utili le funzioni offerte dalla libreria *Newtonsoft.Json* per semplificare di gran lunga l'organizzazione dei dati in oggetti JSON da integrare nelle query da e verso il server, così come il plugin utilizzato per la realizzazione del login attraverso i dati estratti dal proprio profilo *Facebook*.

Tra i punti di forza ulteriormente evidenziabili sicuramente vi è l'utilizzo di un unico e potente linguaggio compilato quale C#, il quale ha consentito di individuare facilmente errori che, con i linguaggi interpretati utilizzati da framework alternativi, sarebbero visibili solo runtime. L'uso dell'unico linguaggio XAML, poi, se da un lato facilita di gran lunga anche la condivisione della user interface tramite tecniche di binding, dall'altro limita fortemente le capacità di personalizzazione grafica dell'app che, al contrario, un linguaggio vicino alla piattaforma garantirebbe. Proprio con riferimento al caso di studio implementato, infatti, l'interfaccia grafica dell'app multipiattaforma realizzata ha seguito come modello base comune quello della versione nativa Android; in realtà la versione nativa iOS, pur mantenendo le medesime funzionalità, presentava alcune sottili differenze nella disposizione e gestione degli elementi grafici che, seguendo questo approccio, non è stato possibile ottenere. In ogni caso, conseguenza positiva del fatto di utilizzare un solo linguaggio è sicuramente la riduzione dei tempi e dei costi di sviluppo dell'app.

Dai test e dalle analisi di performance effettuate sul progetto sviluppato emerge invece un aspetto negativo del framework, come già ampiamente analizzato nel capitolo precedente, ovvero l'eccessivo tempo di avvio che caratterizza le applicazioni; più in generale, a parte qualche caso specifico, si è rilevata una maggiore lentezza di esecuzione, seppur talvolta impercettibile dall'utilizzatore, legata ad un *overhead* a runtime (poco meno del 10% circa rispetto alla versione nativa, a seconda della complessità delle funzioni), abbinata spesso ad una maggiore instabilità delle operazioni effettuate attraverso la rete Internet.

Dunque, uno sviluppatore che si trova a dover iniziare un nuovo progetto come può capire se *Xamarin* è la tecnologia giusta per lui? Sicuramente esso rappresenta la scelta più indicata se si desiderano riutilizzare le proprie conoscenze e competenze sul linguaggio C# e sul framework .NET, oppure se l'applicazione da creare fa largo uso di funzionalità riutilizzabili in tutte le piattaforme, oppure ancora se si ha la necessità di creare il prototipo di un'applicazione o una demo in tempi brevi senza soffermarsi sui dettagli di piattaforma e trascurando alcuni problemi come ad esempio il più lento *startup*, o infine se non si vogliono imparare tutte le specifiche di ogni singola piattaforma. È invece una soluzione da evitare se non si possiedono competenze su C# e .NET, oppure se l'applicazione è fortemente integrata con funzionalità specifiche della piattaforma, oppure ancora se il layout dell'applicazione deve essere diverso per ogni piattaforma, o infine se ci si accorge che i tempi di esecuzione delle operazioni pregiudicano la fluidità e la stabilità di esecuzione dell'applicazione stessa.

In conclusione, raggiungere il 100% di condivisione del codice tra piattaforme,

pur essendo possibile per la maggior parte di applicazioni anche non banali, porta con sé necessariamente un minimo di semplificazione nell'implementazione delle funzionalità, dovuta proprio al diverso supporto offerto dalle piattaforme target. Tuttavia un approccio di questo tipo resta molto valido, riuscendo a soddisfare ugualmente circa il 90% dei bisogni delle app più comuni.

Quindi, premesso che si può fare (quasi) tutto con tutto, se si hanno tempo, conoscenze e budget adeguati l'approccio nativo classico risulta essere sempre la scelta migliore; in caso contrario, il framework *Xamarin* si prospetta come un'ottima soluzione alternativa.

# Bibliografia


[1] https://gizblog.it/2016/01/le-10-invenzioni-piu-innovative-degli-ultimi-15-anni/ (mag-2017)
[2] http://www.gartner.com/newsroom/id/2939217 (mag-2017)
[3] http://www.audiweb.it/news/diffusione-e-total-digital-audience-a-dicembre-2016/ (mag-2017)
[4] http://www.idc.com/promo/smartphone-market-share/os (ago-2017)
[5] https://en.wikipedia.org/wiki/Comparison_of_mobile_operating_systems (mag-2017)
[6] https://www.cleveroad.com/blog/9-differences-between-ios-and-android-app-development (mag-2017)
[7] https://www.1and1.it/digitalguide/siti-web/programmazione-del-sito-web/diversi-tipi-di-app-che-cose-una-web-app/ (giu-2017)
[8] https://www.1and1.it/digitalguide/siti-web/programmazione-del-sito-web/i-vantaggi-e-gli-svantaggi-di-unapp-ibrida/ (giu-2017)
[9] https://www.websitetooltester.com/it/blog/creare-un-app/ (giu-2017)
[10] https://cordova.apache.org (giu-2017)
[11] https://phonegap.com (giu-2017)
[12] http://www.pcprofessionale.it/laboratorio/in-prova-cordova-e-phonegap-applicazioni-mobili-per-ogni-ambiente/3/ (giu-2017)
[13] http://www.appcelerator.com/mobile-app-development-products/ (giu-2017)
[14] https://rms.rhomobile.com (giu-2017)
[15] https://coronalabs.com (giu-2017)
[16] https://www.xamarin.com/platform (ago-2017)
[17] https://www.xamarin.com/forms (lug-2017)
[18] https://developer.xamarin.com/guides/ (ago-2017)
[19] http://www.iphoneandgo.it/2017/01/21/xamarin-la-piattaforma-sviluppare-app-native-cross-platform/ (lug-2017)
[20] https://www.slideshare.net/guidomagrin/introduction-to-xamarin-48120074 (lug-2017)

[21] `https://www.slideshare.net/guidomagrin/xamarin-forms-48299949` (lug-2017)
[22] `https://www.slideshare.net/dotnetcampus/intro-to-xamarin` (ago-2017)
[23] `http://www.html.it/guide/guida-xamarin/` (lug-2017)
[24] `https://xamarinhelp.com/improving-xamarin-forms-startup-performance/` (ago-2017)
[25] `https://developer.xamarin.com/guides/xamarin-forms/deployment-testing/performance/` (ago-2017)
[26] `http://movify.be/xamarin-forms-tips-smooth-launch-on-android/` (ago-2017)
[27] `http://openshop.io` (ago-2017)
[28] A. Del Sole, *Tutti X uno, uno X tutti*, Microsoft Press, 2017.
[29] C. Petzold, *Creating Mobile Apps with Xamarin.Forms*, Microsoft Press, 2015.