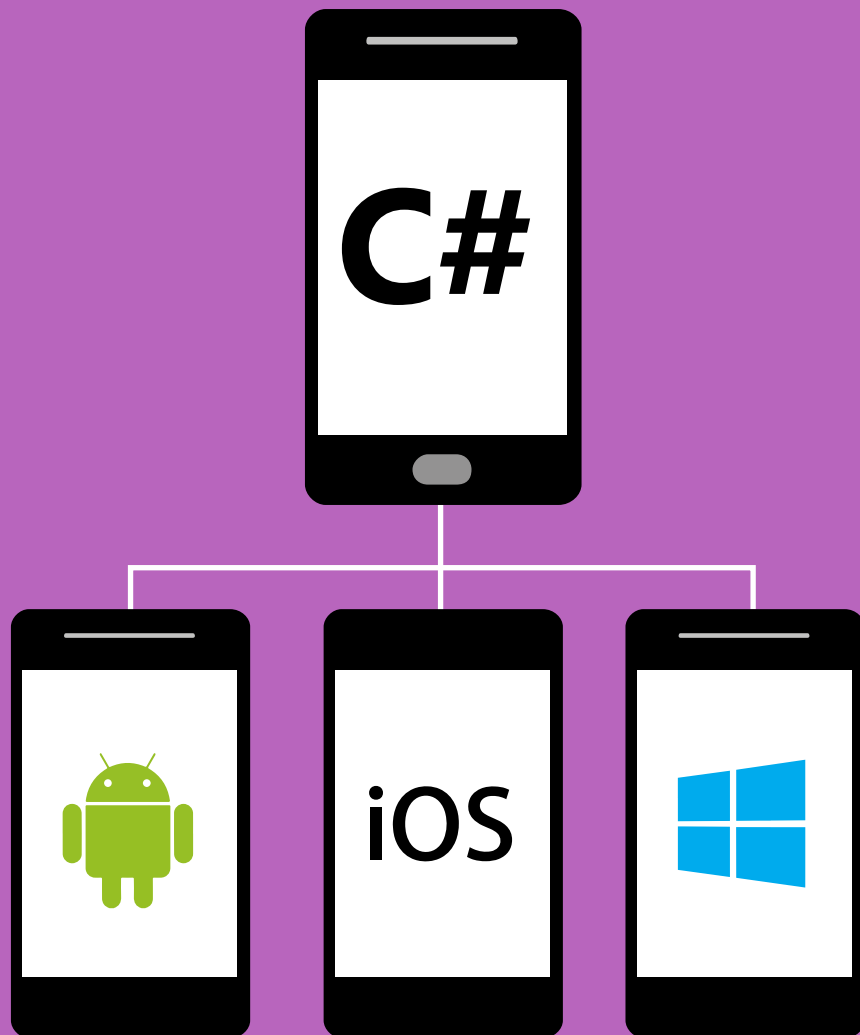


Tutti x uno, **UNO X TUTTI!**



Contenuti



01

Cos'è Xamarin

02

Introduzione
a Xamarin.iOS

03

Introduzione
a Xamarin.Android

04

Introduzione
a Xamarin.Forms

05

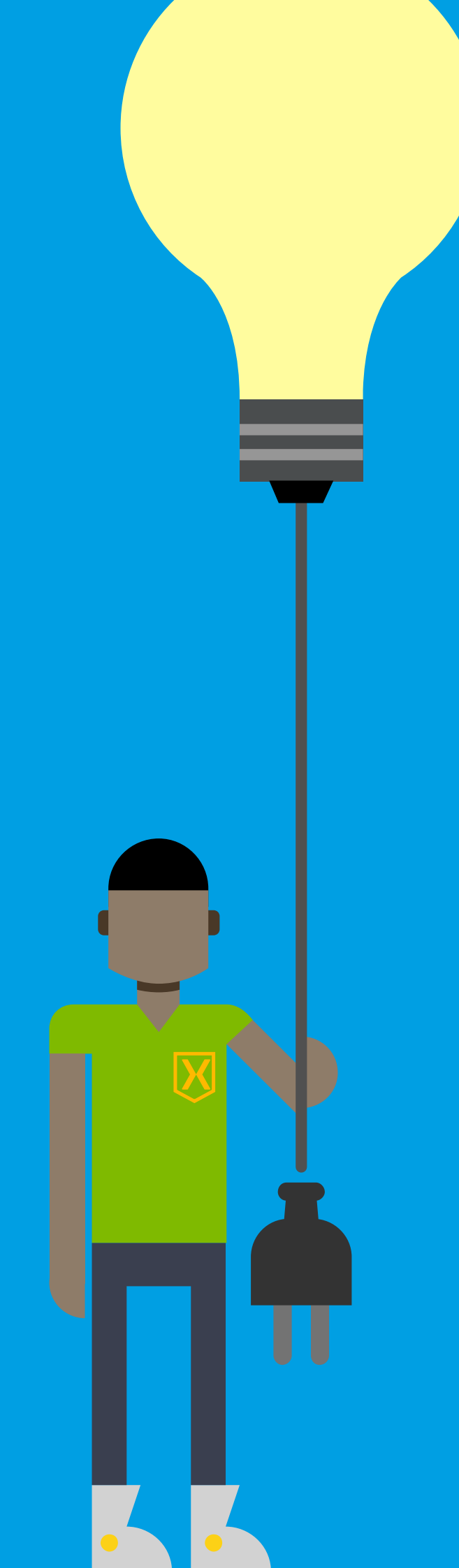
La condivisione
e il riutilizzo
del codice

06

Servizi Azure Mobile
App e Xamarin

07

Autenticazione tra-
mite i servizi Azure
Mobile App



Requisiti

Per completare gli esempi presentati in questo libro, è necessaria la seguente dotazione software:

- un PC con [Microsoft Windows 10](#);
- [Microsoft Visual Studio 2017 o 2015](#) edizione Community o superiore;
- [Xamarin per Visual Studio](#) (selezionabile dall'installazione di Visual Studio 2015 oppure, per Visual Studio 2017, tramite il modulo Mobile Development with .NET nell'installer);
- [Android SDK](#) e i componenti di sviluppo Java indicati dal programma di installazione di Visual Studio 2015;
- [Emulatore per Android di Visual Studio](#) (anch'esso elencato tra i componenti di setup);
- Opzionale, [un computer Mac per la parte di sviluppo iOS](#);

Eventuali altri componenti o programmi richiesti, saranno segnalati dove opportuno.

Cap. 1

Cos'è Xamarin



Xamarin è contemporaneamente il nome di un'azienda, acquistata da Microsoft nel febbraio 2016, e dei prodotti che essa offre agli sviluppatori.

In questo primo capitolo faremo la conoscenza delle tecnologie che mette a disposizione per sviluppare applicazioni multi-piattaforma per dispositivi mobili utilizzando C#, F# e gli ambienti di sviluppo Microsoft.



Lo sviluppo multi-piattaforma e le relative problematiche

Per capire e apprezzare [Xamarin](#), è necessario fare alcune considerazioni sullo sviluppo per dispositivi mobili e le relative problematiche. Nel corso degli ultimi anni c'è stato un incredibile incremento della diffusione di dispositivi mobili, principalmente smartphone e tablet, ma anche altre tipologie di dispositivi come i cosiddetti [wearable](#), le smart TV, i dispositivi per l'Internet of Things. L'incremento di dispositivi è strettamente correlato alla diversificazione di sistemi operativi che danno loro vita, parliamo quindi di iOS (Apple), Android (Google) e Windows (Microsoft) nelle sue versioni per device. Ogni azienda che voglia quindi essere rappresentata anche attraverso un'app, dovrebbe prevedere una versione per ciascuna piattaforma ma questo ha le seguenti, importanti implicazioni tecniche:

- [iOS, Android e Windows espongono API specifiche per il proprio sistema](#), quindi totalmente diverse tra loro e con modalità diverse di essere gestite;
- [per sistemi diversi](#), ci vogliono strumenti di sviluppo e linguaggi di programmazione di-

versi: [Apple con Swift e Xcode](#), [Google con Android Studio e Java](#), [Microsoft con Visual Studio e C#](#).

- oltre alla diversità tra tool e linguaggi, la programmazione dell'interfaccia grafica richiede familiarità con le diverse terminologie. Per esempio, un elemento visivo in [iOS si chiama View](#), in [Android si chiama Widget](#) mentre in [Windows si chiama User Control](#).

Conseguentemente, poter essere presenti sugli Store di tutti i sistemi più diffusi richiede competenze diverse con il relativo aumento di tempi e costi. L'ideale sarebbe quindi poter creare applicazioni multi-piattaforma (dette anche [cross-platform](#)) utilizzando un singolo linguaggio di programmazione e una singola infrastruttura ottenendo tre applicazioni diverse semplicemente condividendo quanto più codice possibile e diversificando il solo codice specifico per ciascuna piattaforma. E' proprio in questo contesto che si inserisce Xamarin.



Xamarin come insieme di prodotti, strumenti e servizi integrati

L'obiettivo di Xamarin è quello di offrire strumenti che permettano agli sviluppatori di raggiungere i principali sistemi operativi per device, come [iOS](#), [Android](#) e [Windows](#), massimizzando il riutilizzo del codice e delle competenze acquisite negli anni. Per fare questo, Xamarin sfrutta **Mono**, un porting open source di .NET in grado di funzionare su sistemi come Mac OS e Linux (e relative distribuzioni), oltre che su Windows. Mono permette di utilizzare C#, il potente e versatile linguaggio di programmazione orientato agli oggetti di Microsoft sfruttandone tutte le potenzialità, per sviluppare applicazioni multi-piattaforma ed è inoltre talmente versatile da essere portato, tra l'altro, anche su sistemi mobili come iOS e Android.

Mono e C# diventano quindi la base su cui Xamarin costruisce i propri strumenti e le proprie tecnologie (di cui parleremo nella prossima sezione) per lo sviluppo di app multi-piattaforma per iOS, Android e Windows. Questo è un fattore fondamentale per gli sviluppatori .NET, che possono quindi riutilizzare le proprie competenze anche su sistemi non Microsoft. Affinché tutto questo sia possibile, Xamarin sviluppa delle librerie che rielaborano ed espongono in forma .NET/Mono le API delle varie piattaforme.

Note col nome di Xamarin.iOS, Xamarin.Mac e Xamarin.Android, esse si basano su Mono e consentono di scrivere codice C#, e in ottica totalmente orientata a .NET, per poter interagire con iOS, Mac OS e Android. Tutto questo assume ancora più valore se pensiamo che Xamarin consente di generare app native. Oltre al runtime e alle librerie, ovviamente serve anche un ambiente di sviluppo e per questo scopo è possibile utilizzare [Xamarin Studio su Mac OS](#) e [Microsoft Visual Studio su Windows](#), attraverso delle apposite estensioni. Microsoft ha recentemente sviluppato Visual Studio for Mac, un'evoluzione di Xamarin Studio espressamente dedicata a Mac OS.

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 su [GitHub](#). In realtà, Xamarin offre agli sviluppatori tutto ciò di cui hanno bisogno per sviluppare, pubblicare, analizzare e mantenere app native per iOS, Android e Windows con riferimento a tutto il ciclo di vita dell'applicazione stessa. Ciò significa che Xamarin non è solo sviluppo di app, ma anche un insieme di servizi correlati. [Questa sezione descrive l'offerta Xamarin, suddivisa per categorie.](#)



Xamarin Platform

Sotto il nome di **Xamarin Platform** vanno gli strumenti di sviluppo per la creazione di applicazioni multi-piattaforma utilizzando C# come linguaggio. In dettaglio, la Xamarin Platform consiste in:

- **Xamarin.iOS**, la parte di piattaforma per lo sviluppo di applicazioni native per iOS. Questa ci consente di sviluppare non solo per iPad e iPhone, ma anche per TvOS ed Apple Watch. Fornisce accesso al 100% delle API native di iOS ed addirittura consente di invocare codice Objective-C da C#.
- **Xamarin.Android**, la parte di piattaforma per lo sviluppo di applicazioni native per dispositivi Android, inclusi smartphone, tablet e dispositivi wearable. Fornisce accesso al 100% delle API native di Google ed è costantemente aggiornata ad ogni rilascio.
- **Xamarin.Forms**, la parte di piattaforma che consente di creare applicazioni native per iOS, Android e Windows attraverso la condivisione dell'interfaccia grafica e di tutto il codice che non sia specifico del sistema. Attraverso XAML e C#, è possibile sviluppare app native che massimizzano il riutilizzo del codice e delle proprie competenze .NET.
- **Xamarin Studio**, l'ambiente integrato di sviluppo per Mac e grazie al quale è possibile sviluppare applicazioni sfruttando strumenti di design e di editing evoluto del codice C#.

Xamarin Studio si basa sulle solution MSBuild di Visual Studio.

- **Visual Studio for Mac**, un IDE specifico per Mac OS che combina l'ambiente di Xamarin Studio con peculiarità tipiche di Visual Studio, e destinato alla produzione di app con Xamarin ma anche applicazioni basate su .NET Core.
- **Estensioni e tool per Microsoft Visual Studio 2013 e superiore**, che consentono, su Windows, di sviluppare app native portando tutto ciò che la Xamarin Platform offre nello strumento "principe" di sviluppo di casa Microsoft. Visual Studio 2015 già include Xamarin e lo stesso sarà per Visual Studio 2017 quando verrà rilasciato.
- **Xamarin Profiler**, un nuovo strumento per l'analisi delle performance, del consumo di memoria e di risorse di CPU.
- **Xamarin Inspector**, un nuovo strumento che consente di analizzare il comportamento e la composizione dell'interfaccia utente nei confronti delle nostre app.

Come vedremo poi meglio nel Capitolo 2, Xamarin.iOS e nel Capitolo 4, Xamarin.Forms è necessario disporre di un computer Mac se si vogliono sviluppare app per iOS anche se si utilizza Windows come ambiente di sviluppo.



Xamarin Test Cloud

Xamarin Test Cloud è una piattaforma che consente di effettuare test automatici sull'interfaccia utente, prima di distribuire le applicazioni. Sebbene Microsoft stia lavorando per includere questa infrastruttura nel Visual Studio Mobile Center, di cui parliamo nella prossima sezione, è a tutti gli effetti ancora attiva e la transizione dovrebbe essere completata nel corso del 2017. Non si tratta di un ambiente di test locale, ma di un servizio offerto tramite Cloud. [Con Xamarin Test Cloud è possibile simulare il deploy della propria app](#) su un numero estremamente elevato di dispositivi fisici e sistemi, così da verificarne il comportamento con diverse versioni di iOS e Android, diversi firmware, diversi fattori di dimensioni del display e così via. Per esempio, è possibile simulare i gesti dell'utente, come il touch o lo scorrimento, e verificare che l'applicazione risponda a dovere. Tutto questo è possibile grazie a un framework sviluppato da Xamarin, chiamato Calabash, che consente di eseguire test automatizzati sulla UI. [Grazie a Calabash, è possibile testare il comportamento dell'applicazione su Android, iOS e Windows in modo totalmente automatizzato](#). Questo consente, inoltre, di analizzare il comportamento delle applicazioni anche quando utilizzano sensori come il GPS o pulsanti fisici e permette di analizzare le performance delle applicazioni su vari sistemi operativi.

Xamarin Test Cloud è, di fatto, un ambiente di esecuzione di test che in realtà possono essere scritti in vari linguaggi, tra cui C#, in Visual Studio e Xamarin Studio attraverso degli appositi template.

Questi template rendono più semplice il compito di programmare i gesti e i contesti da testare; è inoltre opportuno sottolineare che Xamarin Test Cloud aderisce agli standard Nunit e permette di scrivere test anche con Ruby e Java.



Xamarin Insights e il passaggio a Visual Studio Mobile Center

Tra i vari servizi, Xamarin ha offerto **Xamarin Insights**, una piattaforma basata su Cloud per il monitoraggio delle applicazioni tramite telemetria. Xamarin Insights ha poi iniziato una transizione verso HockeyApp, per poi confluire in quella che è l'attuale preview del **Visual Studio Mobile Center**, un vero e proprio centro di controllo delle applicazioni ospitato su Microsoft Azure come piattaforma Cloud. Visual Studio Mobile Center offre una serie di servizi per la gestione dell'intero ciclo di vita di un'applicazione mobile, dallo sviluppo, al test, all'analisi. Ciò che è interessante sottolineare, è che Visual Studio Mobile Services offre i suoi servizi non solo alle applicazioni create con Xamarin, ma anche ad applicazioni create con React Native, Objective-C/Swift (iOS) e Java (Android). Più nello specifico, i servizi Cloud offerti dal Mobile Center sono:

- **Automazione delle build.** Il servizio si integra con qualunque repository Git ed è in grado di automatizzare le build ad ogni commit o push del proprio codice. Nel caso di iOS, questo servizio include già un proprio agent che evita la necessità di avere un Mac collegato in rete per le varie compilazioni.

- **Automazione dei test.** Questo servizio consente di eseguire test automatici, anche sull'interfaccia utente, su più di 2000 dispositivi e 400 configurazioni diverse. Il motore di questo servizio è proprio Xamarin Test Cloud di cui abbiamo parlato in precedenza.
- **Distribuzione ai tester.** E' possibile definire un gruppo di tester selezionati che riceveranno l'applicazione una volta che i test automatici passano. Il motore di questo servizio è HockeyApp.
- **Servizi di monitoraggio.** Visual Studio Mobile Center consente di monitorare, tramite telemetria, informazioni su eventuali crash delle applicazioni e l'utilizzo che viene fatto delle applicazioni stesse, incluse informazioni sulla lingua e sulla regione dei dispositivi che le eseguono.



- **Servizi di backend.** Visual Studio Mobile Services include e offre tre servizi di backend: Identity, per rendere più semplice l'implementazione dell'autenticazione tramite Facebook, Twitter, Google+ e Microsoft Account; Tables, un servizio per la memorizzazione di dati anche senza connettività, sfruttando anche la sincronizzazione dei dati; Push notifications, un servizio per l'invio di notifiche push agli utilizzatori dell'applicazione.

Al momento della scrittura di questo capitolo il Visual Studio Mobile Center è utilizzabile in preview, quindi i servizi offerti sono destinati a crescere e potrebbero essere soggetti a modifiche.

Xamarin University

La formazione è alla base di qualunque tecnologia e la **Xamarin University** offre agli sviluppatori numerose risorse di apprendimento, tra cui corsi e percorsi di certificazione. Infatti, attraverso la frequenza di lezioni e corsi online è possibile prepararsi agli esami per il conseguimento della certificazione Xamarin Certified Mobile Developer. Gli esami possono essere sostenuti online e sempre attraverso il portale della Xamarin University. Tramite i propri canali, occasionalmente la **Xamarin University** eroga anche Webinar e sessioni formative online.



Riepilogo

Xamarin è un'azienda recentemente acquisita da Microsoft che ha sviluppato librerie, runtime e strumenti di sviluppo per scrivere applicazioni multi-piattaforma per Android, iOS e Windows utilizzando C# e le competenze .NET.

Oltre agli strumenti di sviluppo, l'offerta riguarda anche una serie di servizi per il test automatico (Xamarin Test Cloud), la gestione del ciclo di vita integrata nel Visual Studio Mobile Center, e la formazione erogata tramite la Xamarin University, che consente di frequentare lezioni online orientate alla certificazione ufficiale. Dopo questa prima panoramica, iniziamo a prendere confidenza con gli strumenti di sviluppo di Xamarin partendo dalla piattaforma Xamarin.iOS.



Cap. 2

Introduzione
a Xamarin.iOS



Xamarin.iOS è l'insieme di strumenti e librerie che consentono di sviluppare app native per iPad, iPhone, Apple Watch e tvOS con C# tramite Visual Studio e Visual Studio for Mac. Con riguardo alle librerie, Xamarin.iOS fornisce accesso anche a tutti i framework Apple, come Cocoa (fondamentale per lo sviluppo di app), e la denominazione delle classi richiama proprio quelle dei framework Apple per uniformità. Xamarin.iOS consente inoltre di sviluppare applicazioni Universal ed offre strumenti di progettazione integrati nell'ambiente di lavoro. In questo capitolo viene fatta un'introduzione a Xamarin.iOS con una semplice app di esempio, ponendo l'accento sulla strumentazione necessaria e sulla relativa configurazione. Questo tornerà utile anche quando verrà illustrato Xamarin.Forms.

Nota: In questo libro viene utilizzato Microsoft Visual Studio 2015 come ambiente di sviluppo, ma gli stessi concetti si applicano anche a Visual Studio 2017, Visual Studio for Mac e Xamarin Studio.



Prerequisiti

In linea generale, sviluppare applicazioni per iOS richiede che siano soddisfatti alcuni prerequisiti legati a specifiche policy di Apple. Questi prerequisiti non sono legati a Xamarin, ma riguardano tutte le piattaforme applicative che si rivolgono ad iOS.

Nello specifico, per sviluppare per iOS dovreste avere a disposizione:

- **un computer Mac raggiungibile in rete.**

Questo prerequisito è ovviamente già soddisfatto se sviluppate direttamente su Mac con Xamarin Studio ma dovreste dotarvene se siete su Windows con Visual Studio. Il Mac è infatti necessario per poter compilare il pacchetto dell'applicazione. Se state ancora valutando di sviluppare per iOS, una buona soluzione è costituita da servizi come **MacInCloud**, che offrono l'accesso a computer Mac remoti secondo abbonamenti di tipo pay-as-you-go. Con specifico riferimento a Xamarin, è necessario che sul Mac sia installato almeno il sistema operativo Mac OS v. 10.11 ("El Capitan"). Computer di tipo Mac Mini vanno ugualmente bene;

- **abilitare l'accesso remoto al Mac;**

- **Xcode e la più recente versione dell'SDK di iOS installati sul Mac.** Xcode è disponibile gratuitamente sull'Apple Store ed è ne-

cessario perché il processo di compilazione utilizza alcuni strumenti dell'SDK;

- **Xamarin Studio e Xamarin.iOS installati sul Mac.**

Indipendentemente dal fatto che utilizzerete o meno Xamarin Studio, la sua presenza è necessaria. Questi componenti includono lo Xamarin Mac Agent, che consente di ricevere connessioni da Visual Studio;

- **su Windows, il simulatore iOS di Xamarin.**

Si tratta di un componente opzionale, poiché il **simulatore di iOS** è già presente su Mac. Tuttavia, ogni qualvolta avviate il debug, il simulatore verrebbe avviato su Mac, mentre, grazie al simulatore iOS di Xamarin, potete avviarlo su Windows con maggiore comodità di debugging. Il simulatore è disponibile solo per le edizioni Enterprise di Visual Studio 2015 e 2017;

- **un account sviluppatore Apple per debug e test su dispositivo fisico.**

Non è più necessario registrarsi all'Apple Developer Program e pagare la relativa sottoscrizione almeno fin tanto che siete in fase di sviluppo, debug e test. Infatti è possibile ottenere un account sviluppatore gratuito e passare a quello a pagamento, con relativo rilascio dei profili di identità, solo nel momento in cui deciderete di pubblicare app sull'Apple Store;



- **un provisioning profile collegato all'account sviluppatore.**

Alcuni passaggi sono piuttosto articolati e per questi potete consultare la **documentazione ufficiale** di Xamarin relativamente alla **configurazione del Mac** e del **profilo sviluppatore**.

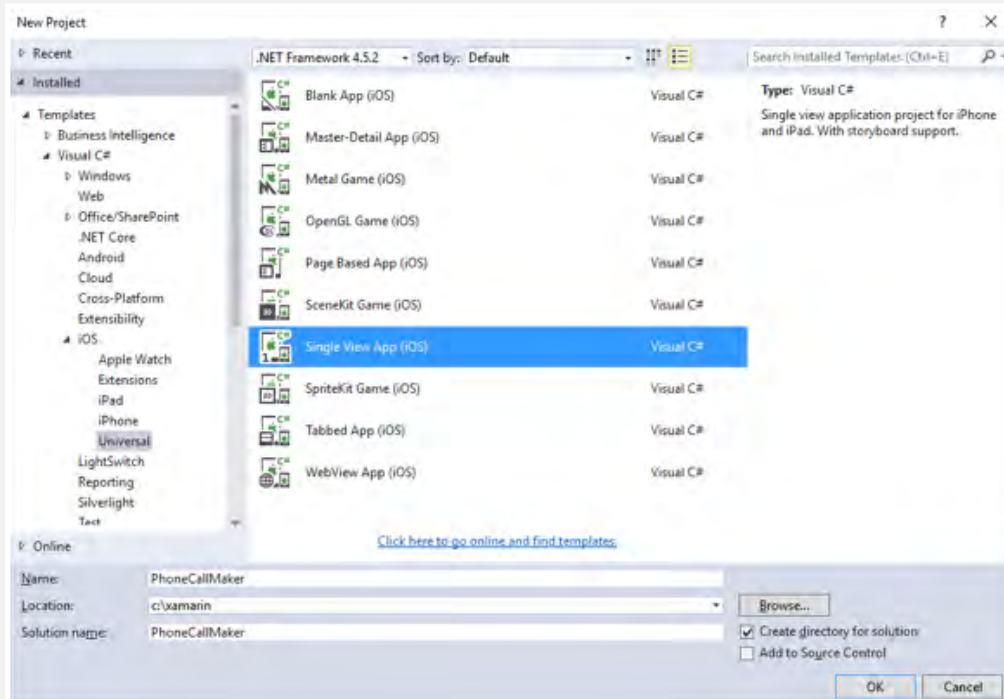
Introduzione allo sviluppo

Con Xamarin.iOS è possibile sviluppare app native utilizzando C# e il 100% delle API del sistema operativo Apple.

In questo capitolo vengono introdotti i concetti fondamentali per lo sviluppo con Visual Studio 2015 attraverso la creazione di una semplice app dimostrativa che consente di inviare un sms ad un contatto selezionato dall'utente. Questo consentirà di prendere confidenza con l'ambiente di progettazione e sviluppo, con le modalità con cui Xamarin.iOS interagisce col sistema Apple e con le modalità con cui si possono sfruttare le API di sistema. L'obiettivo di questo capitolo è mostrare come creare una semplice app in grado di inviare sms, cosa che permetterà di capire molti concetti.

In Visual Studio 2015, selezionate File, New Project. Quando la finestra di dialogo New Project appare, espandete il nodo iOS sotto Visual C#, come in Figura 2-1.

Figura 2-1 Creazione di un progetto Xamarin.iOS



Come vedete, è disponibile una serie di cartelle di template di progetto che riguardano iPad, iPhone, Apple Watch e le app Universal che hanno lo scopo di funzionare su tutti i dispositivi. A partire da iOS 8, è anche possibile sviluppare le cosiddette Extensions, ossia degli widget che si integrano con le funzionalità di sistema.

Per quanto riguarda iPhone, iPad e Universal, i vari template consentono di creare app basate su diverse strutture, come app a pagina singola, app con pagine a schede, giochi e app con pagine master-detail. Per finalità dimostrative, utilizzeremo il modello chiamato Single View App (iOS), visibile nella selezione di **Figura 2-1**, chiamando il nuovo progetto PhoneCallMaker. [Quando pronti, fate click su OK.](#)

Dopo alcuni secondi, [Visual Studio vi chiederà di specificare l'indirizzo del Mac](#) a cui connettersi per la compilazione, attraverso una finestra chiamata Xamarin Mac Agent. In questa finestra c'è un pulsante chiamato Add Mac, su cui bisogna fare click per poter specificare (almeno la prima volta) il Mac a cui connettersi. Questo avviene in una finestra chiamata Add

Mac, all'interno della quale va specificato l'indirizzo IP del Mac, come dimostrato in **Figura 2-2**.

Sebbene Visual Studio consenta di specificare anche il nome di rete del Mac, e non solo il suo IP, è preferibile specificare quest'ultimo per evitare noiosi errori di connessione. Dopo aver fatto click su Add vi verranno richieste le credenziali di accesso al Mac (nome utente e password del profilo con cui fate login su Mac) e, a connessione completata, il Mac sarà visibile nell'elenco riportato nella dialog Xamarin Mac Agent (vedi Figura 2-3).

Suggerimento: E' possibile modificare la connessione al Mac Agent in qualunque momento tramite Tools, iOS, Xamarin Mac Agent.

A creazione del progetto completata, Visual Studio mostrerà una pagina di benvenuto con alcuni link a risorse di apprendimento e ad applicazioni di esempio da scaricare. [Prima di iniziare a scrivere codice, è bene capire la struttura della solution appena creata.](#)

Come vedete, è disponibile una serie di cartelle di template di progetto che riguardano iPad, iPhone, Apple Watch e le app Universal che hanno lo scopo di funzionare su tutti i dispositivi. [A partire da iOS 8, è anche possibile sviluppare le cosiddette Extensions, ossia degli widget che si integrano con le funzionalità di sistema.](#) Per quanto riguarda iPhone, iPad e Universal, i vari template consentono di creare app basate su diverse strutture, come app a pagina singola, app con pagine a schede, giochi e app con pagine master-detail. Per finalità dimostrative, utilizzeremo il modello chiamato Single View App (iOS), visibile nella selezione di **Figura 2-1**, chiamando il nuovo progetto PhoneCallMaker. Quando pronti, fate click su OK.

Dopo alcuni secondi, Visual Studio vi chiederà di specificare l'indirizzo del Mac a cui connettersi per la compilazione, attraverso una finestra chiamata Xamarin Mac Agent. In questa finestra c'è un pulsante chiamato Add Mac, su cui bisogna fare click per poter specificare (almeno la prima volta) il Mac a cui connettersi. Questo avviene in una finestra chiamata Add Mac, all'interno della quale va specificato l'in-

dirizzo IP del Mac, come dimostrato in **Figura 2-2**.

Sebbene Visual Studio consenta di specificare anche il nome di rete del Mac, e non solo il suo IP, è preferibile specificare quest'ultimo per evitare noiosi errori di connessione. Dopo aver fatto click su Add vi verranno richieste le credenziali di accesso al Mac (nome utente e password del profilo con cui fate login su Mac) e, a connessione completata, il Mac sarà visibile nell'elenco riportato nella dialog Xamarin Mac Agent (**vedi Figura 2-3**).

Suggerimento: E' possibile modificare la connessione al Mac Agent in qualunque momento tramite Tools, iOS, Xamarin Mac Agent.

[A creazione del progetto completata, Visual Studio mostrerà una pagina di benvenuto con alcuni link a risorse di apprendimento e ad applicazioni di esempio da scaricare.](#) Prima di iniziare a scrivere codice, è bene capire la struttura della solution appena creata.

Figure 2-2 Specifica del Mac a cui connettersi

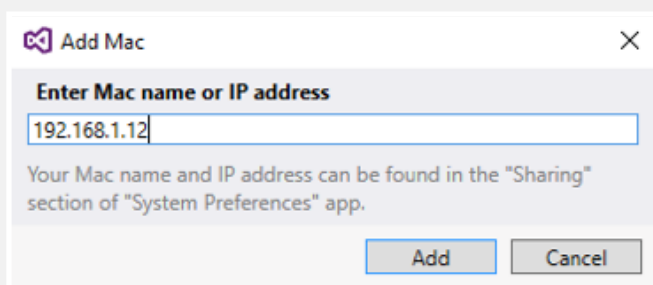
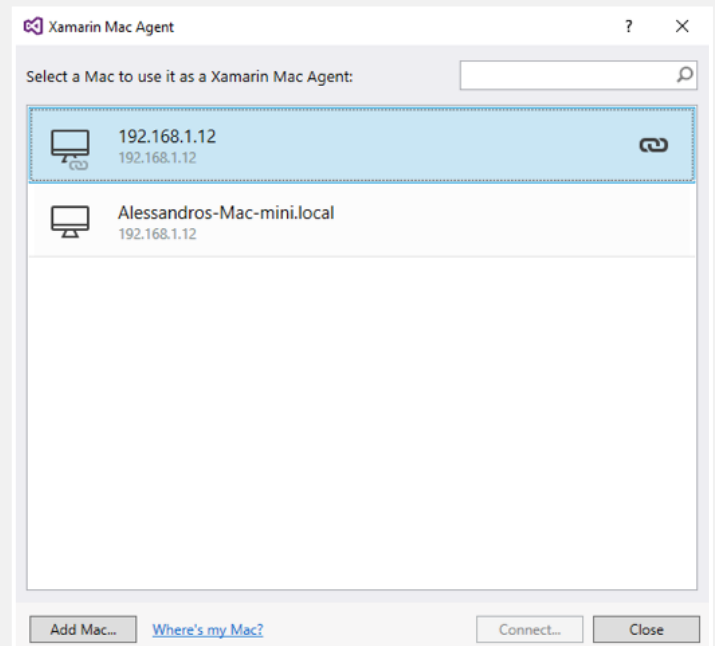


Figure 2-3 Visual Studio è connesso al Mac Agent





Concetti fondamentali sulle solution Xamarin.iOS

Nota: Poiché l'obiettivo di questo capitolo è spiegare come iniziare a sviluppare per iOS con Xamarin, non è possibile scendere nei dettagli della terminologia di iOS, dei relativi framework e del comportamento o dello scopo degli elementi visuali per cui viene dato per assunto che conosciate tali aspetti o che facciate gli approfondimenti del caso separatamente. Per maggiori informazioni al riguardo, potete consultare il [portale sviluppatori di Apple](#).

Xamarin.iOS consente di rappresentare, sotto forma di solution e di file di codice C#, quelli che sono gli elementi caratteristici di un'applicazione per iOS. In Solution Explorer questa rappresentazione è immediatamente evidente e consiste dei seguenti elementi:

- **AppDelegate.cs**, che mappa il corrispondente AppDelegate di iOS e che avvia la UI dell'applicazione. In questo file è definita la classe AppDelegate che gestisce, tra l'altro, gli eventi relativi al ciclo di vita dell'applicazione, come OnActivate, FinishedLaunching,DidEnterBackground. Riceve inoltre notifiche dal sistema operativo, incluse le push notifications.
- **MainStoryboard**, che definisce l'interfaccia grafica di un flusso di pagine nell'applicazione. Visual Studio offre, come vedremo tra breve, uno specifico designer per lavorare con i file .storyboard.
- **ViewController.cs**, che rappresenta il codice operativo attivato dagli elementi visuali (View) presenti in MainStoryboard. Mappa il concetto di ViewController di iOS e richiama il pattern Model-View-Controller.

- **Main.cs**, che rappresenta il punto di ingresso dell'applicazione e che si occupa di istanziare la classe AppDelegate. Questo file è necessario perché C# e Mono non hanno la nozione di AppDelegate e quindi hanno la necessità che il punto di ingresso sia costituito dal metodo Main.
- **Info.plist**, un file comune alle app iOS grazie al quale è possibile specificare tutte le proprietà del progetto, da quelle relative al dispositivo di destinazione, al processo di build, ai metadati, alle icone e alle risorse, alla possibilità di far funzionare codice in background.
- **Entitlements.plist**, altro file comune alle app iOS attraverso il quale si possono specificare servizi utilizzati, mappe, roaming dati e utilizzo di iCloud.
- **Asset Catalogs**, che consente di specificare icone e risorse diverse da quelle di default.

- **Native References**, un elemento che include riferimenti a librerie di Xamarin.iOS e grazie al quale è possibile aggiungere riferimenti ad altri framework e librerie.
- **Components**, un elemento che consente di aggiungere componenti e librerie dallo Store di Xamarin. Questo non è chiaramente un elemento delle app iOS, ma esclusivo di Xamarin (ed è disponibile anche per Xamarin.Android e Xamarin.Forms).
- **Resources**, una cartella in cui confluiscono risorse utilizzate dall'applicazione, inclusi i file .xib (librerie di controlli ed elementi visuali).

Nel file AppDelegate.cs può essere aggiunto codice di inizializzazione delle app, mentre in Info.plist è anche possibile aggiungere vari file .storyboard attraverso i quali implementare interfacce grafiche complesse.



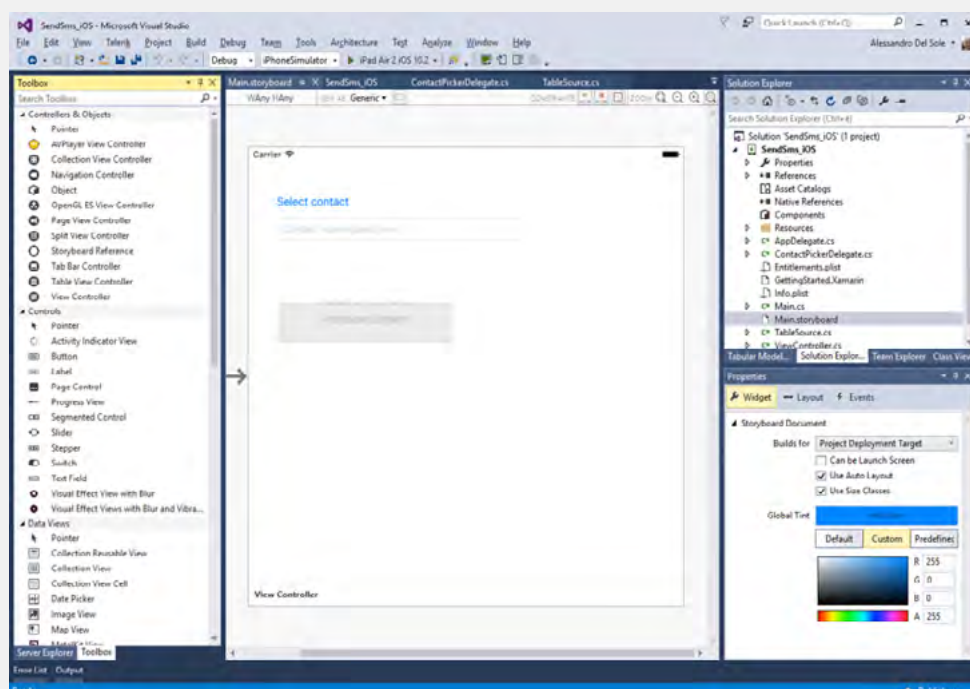
Progettazione dell'interfaccia grafica

In iOS l'interfaccia grafica si basa su file di tipo `.xib` e `.storyboard`. Questi ultimi sono i più utilizzati con le versioni più recenti degli SDK e dei tool di sviluppo e si basano sul linguaggio di markup XML.

Nel progetto appena creato c'è un file chiamato `Main.storyboard`, che rappresenta la pagina principale. [Facendo doppio click su questo file](#), verrà attivata la finestra di progettazione di Visual Studio per iOS, con tanto di controller, elementi della UI e Views nella casella degli strumenti.

Nell'applicazione di esempio [servirà un pulsante per la selezione di un contatto dalla rubrica](#), [un campo testo per visualizzare il nome del contatto selezionato](#) e [una lista in cui visualizzare l'elenco dei numeri del contatto](#), in modo che sia possibile selezionarlo per poi attivare l'applicazione di messaggistica. Dalla casella degli strumenti, quindi, trascinate un Button, un Text Field e una Table View in modo che tutto sia simile alla **Figura 2-4**.

Figura 2-4 La finestra di progettazione per iOS



La finestra delle proprietà consente, come ci si aspetterebbe, di poter assegnare proprietà dei vari elementi visuali. Poiché sarà necessario interagire con gli elementi visuali da codice C#, è bene assegnare la proprietà Name per ognuno di essi rispettivamente con `SelectContactButton`, `SelectedContactField` e `TableView`.

Per il Text Field, è anche possibile specificare un cosiddetto placeholder, ossia testo più chiaro che indica all'utente cosa va inserito in quel campo. La proprietà da assegnare si chiama Placeholder e la **Figura 2-4** ne mostra un esempio.

Nota: nello sviluppo iOS, ricorrono i concetti di Outlet e di Action. L'Outlet può essere considerato come un riferimento che un oggetto nel codice ottiene nei confronti di un oggetto in un file storyboard. Un'Action, invece, può essere paragonata ad un metodo che viene invocato in risposta di input dell'utente come gestualità touch.



Accesso alle API native: l'elenco dei contatti

Il passaggio successivo consiste nell'utilizzare API native del sistema iOS per richiamare l'elenco dei contatti attraverso l'interfaccia predefinita e di ottenere le informazioni del contatto che l'utente seleziona.

Questo esempio consente di iniziare a capire come accedere alle varie API native. L'interfaccia nativa di gestione dei contatti è accessibile attraverso i namespace ContactsUI e Contacts, mentre la sua istanza è rappresentata dalla classe CNContactPickerViewController. A tale istanza si accede mediante l'oggetto CNContactPickerDelegate, che però non può essere utilizzato direttamente e pertanto è necessario creare una classe derivata. Ciò premesso, [aggiungete al progetto una classe chiamata ContactPickerDelegate.cs](#). Noterete come la finestra di dialogo Add New Item mostra un elenco di elementi specifici per i progetti Xamarin.iOS. Il **Listato 2-1** mostra il codice completo della classe, a cui seguiranno delle considerazioni.

Listato 2-1

```
using System;
using ContactsUI;
using Contacts;

namespace SendSms_iOS
{
    public class ContactPickerDelegate : CNContactPickerDelegate
    {
        public ContactPickerDelegate()
        {
        }

        public ContactPickerDelegate(IntPtr handle) : base(handle)
        {
        }

        public override void ContactPickerDidCancel(CNContactPickerViewController picker)
        {
            // Raise the selection canceled event
            RaiseSelectionCanceled();
        }

        public override void DidSelectContact(CNContactPickerViewController picker,
```

```

        CNContact contact)
    {
        // Raise the contact selected event
        RaiseContactSelected(contact);
    }

    public override void DidSelectContactProperty(CNContactPickerViewController picker,
        CNContactProperty contactProperty)
    {
        // Raise the contact property selected event
        RaiseContactPropertySelected(contactProperty);
    }

    public delegate void SelectionCanceledDelegate();
    public event SelectionCanceledDelegate SelectionCanceled;

    internal void RaiseSelectionCanceled()
    {
        this.SelectionCanceled?.Invoke();
    }

    public delegate void ContactSelectedDelegate(CNContact contact);
    public event ContactSelectedDelegate ContactSelected;

    internal void RaiseContactSelected(CNContact contact)
    {
        this.ContactSelected?.Invoke(contact);
    }

    public delegate void ContactPropertySelectedDelegate(CNContactProperty property);
    public event ContactPropertySelectedDelegate ContactPropertySelected;

    internal void RaiseContactPropertySelected(CNContactProperty property)
    {
        this.ContactPropertySelected?.Invoke(property);
    }
}

```

Come potete osservare, [la classe gestisce i vari eventi del selettore \(come DidSelectContact e ContactPickerDidCancel\)](#) e richiama i delegate appropriati affinché il chiamante, nel nostro caso la UI, sappia che un contatto è stato selezionato, cosa che avviene nel metodo `RaiseContactSelected` a cui viene passata l'istanza della classe `CNContact` che rappresenta, appunto, il contatto.



Liste e Data-Binding

Il secondo obiettivo è quello di far sì che l'interfaccia grafica, tramite il controllo Table View, mostri l'elenco dei numeri di telefono per il contatto selezionato.

Normalmente questo tipo di assegnazione avviene **tramite data-binding, ma in Xamarin.iOS** il funzionamento è un po' diverso nel senso che è necessario creare la sorgente dati da assegnare alla tabella. Per la prima volta vediamo in azione elementi offerti dal framework Cocoa, alla base degli elementi di UI, ed esposti dal namespace UIKit. [La Table View è l'istanza di una classe UITableView, che ha una proprietà Source di tipo UITableViewSource.](#)

Bisogna quindi creare una classe che erediti da UITableViewSource e che riceva l'elenco di elementi da assegnare come sorgente sotto forma di array. In iOS, infatti, non esiste la stessa serie di collection che esistono nel mondo .NET, ma si ragiona per array tipizzati. Ciò premesso, aggiungete una nuova classe al progetto chiamata TableSource.cs. Il codice, coi commenti, è mostrato nel **Listato 2-2** in cui si vede l'utilizzo del namespace Foundation, che può essere considerato al System di .NET.

Listato 2-2

```
using System;
using UIKit;
using Foundation;

namespace SendSms_iOS
{
    public class TableSource : UITableViewSource
    {
        // An array of items as the data source
        protected string[] tableItems;

        // An identifier used to retrieve cells
        protected string cellIdentifier = "TableCell";

        // The owner of the bound Table View
        ViewController owner;

        public TableSource(string[] items, ViewController owner)
        {

```

```

        tableItems = items;
        this.owner = owner;
    }

    // Get the content of a specific cell
    public override UITableViewCell GetCell(UITableView tableView, NSIndexPath indexPath)
    {
        // request a recycled cell to save memory
        UITableViewCell cell = tableView.DequeueReusableCell(cellIdentifier);
        // if there are no cells to reuse, create a new one
        if (cell == null)
            cell = new UITableViewCell(UITableViewCellStyle.Default, cellIdentifier);
        cell.TextLabel.Text = tableItems[indexPath.Row];
        return cell;
    }

    public override nint RowsInSection(UITableView tableview, nint section)
    {
        return tableItems.Length;
    }

    // Retrieve the selected row and sends an sms
    public override void RowSelected(UITableView tableView, NSIndexPath indexPath)
    {
        owner.SendSms(tableItems[indexPath.Row]);
    }
}

```

I metodi marcati come `override` implementano i relativi metodi della classe astratta `UITableViewSource`. Di particolare interesse in questo codice c'è sicuramente il costruttore, che riceve un array di stringhe a costituire la sorgente dati (dove ciascuna stringa rappresenta un numero di telefono), e `RowSelected`, che consente di recuperare la riga selezionata nella Table View che riceve come argomento.

L'oggetto `indexPath`, di tipo `NSIndexPath`, consente di risalire alla riga corrente mediante la sua proprietà `Row` che nel nostro caso rappresenta un numero di telefono. Quest'ultimo viene passato a un metodo chiamato `SendSms` e che è definito nel ViewController chiamante.



Il ViewController

Nello sviluppo per iOS, un `ViewController` può essere paragonato al file di code-behind in C# per i file `.Xaml`.

E' quindi quella particolare classe che consente di rendere operativa la UI oppure di gestirne il comportamento da codice. Nell'esempio corrente, c'è un solo `ViewController` definito nel file `ViewController.cs` e che controlla la View iniziale all'interno del file `Main.storyboard`.

Ragionando per oggetti, un `ViewController` è un'istanza della classe `UIViewController`, anch'essa dal framework `UIKit`. Per praticità espositiva, il contenuto di questo file verrà suddiviso in più parti. Innanzitutto la definizione della classe e del metodo `SendSms`:

```
using Contacts;
using ContactsUI;
using Foundation;
using System;
using System.Linq;
using UIKit;
namespace SendSms_iOS
{
    public partial class ViewController :
    UIViewController
    {
        public void SendSms(string phoneNumber)
        {
            var smsTo = NSURL.FromString($"sms:
{phoneNumber}");
            if (UIApplication.SharedApplication.
CanOpenUrl(smsTo))
            {
                UIApplication.SharedApplication.
OpenUrl(smsTo);
            }
            else
            {
                UIAlertView alert = new UIAlertView()
                {
                    Title = "Error",
                    Message = "SMS is unavailable for the selected number."
                };
                alert.AddButton("OK");
                alert.Show();
            }
        }
    }
}
```

Azioni come invio di un sms o di un'email o l'avvio di una telefonata, in iOS avvengono tramite URL esattamente come avverrebbe l'apertura di una pagina Web. L'URL viene passato al metodo `OpenUrl` della classe `UIApplication`, una classe singleton esposta dalla proprietà `SharedApplication` e che rappresenta l'istanza corrente dell'applicazione. E' quindi necessario costruire l'URL, che per gli sms ha il prefisso sms: seguito dal numero telefonico.

E' buona regola verificare che l'URL sia formalmente corretto tramite `CanOpenUrl` prima di tentare di aprirlo. Se non è possibile, il metodo mostra un messaggio tramite la classe `UIAlertView`, alla cui istanza è possibile assegnare un titolo (**Title**), un messaggio (**Message**), il testo per il pulsante di chiusura (**AddButton**) e su cui si invoca `Show` per mostrarlo. E' poi la volta del costruttore e di due eventi relativi al ciclo di vita dell'app:

```
using Contacts;
using ContactsUI;
public ViewController(IntPtr handle) : base(handle)
{
}

public override void ViewDidLoad()
{
    base.ViewDidLoad();
    // Perform any additional setup after loading the view, typically from a nib.
    this.SelectContactButton.TouchUpInside += SelectContactButton_TouchUpInside;
}

public override void DidReceiveMemoryWarning()
{
    base.DidReceiveMemoryWarning();
    // Release any cached data, images, etc that aren't in use.
}
```

[ViewDidLoad](#) è un evento che si verifica al completamento del caricamento della pagina.

Prima del suo verificarsi, i controlli non hanno istanza (sono null) e può essere paragonato all'InitializeComponent di piattaforme come WPF o Xamarin.Forms. Qui può essere specificato codice di inizializzazione, come nel nostro caso l'assegnazione di un gestore per l'evento TouchUpInside, corrispondente ad eventi come Tapped o Click di .NET. DidReceiveMemoryWarning è invece il posto in cui bisognerà rilasciare risorse allocate nel caso in cui si stia verificando un calo preoccupante della memoria libera. L'ultima parte del codice riguarda proprio il gestore dell'evento TouchUpInside, che viene commentato in dettaglio per una migliore leggibilità.

[Notate l'utilizzo di tecniche di programmazione tipiche di C# come i delegate anonimi, le query LINQ e le espressioni lambda applicate ad iOS:](#)

```
private void SelectContactButton_TouchUpInside(object sender, EventArgs e)
{
    // Create a new picker
    var picker = new CNContactPickerViewController();

    // A contact has many properties (email, phone numbers, birthday)
    // and you must specify which properties you want to fetch,
    // in this case PhoneNumbers
    picker.DisplayedPropertyKeys = new NSString[] { CNContactKey.PhoneNumbers };
    picker.PredicateForEnablingContact = NSPredicate.FromValue(true);
    picker.PredicateForSelectionOfContact = NSPredicate.FromValue(true);

    // Respond to selection
    var pickerDelegate = new ContactPickerDelegate();
    picker.Delegate = pickerDelegate;

    // If the user cancels the contact selection,
    // show an empty string
    pickerDelegate.SelectionCanceled += () => {
        this.SelectedContactField.Text = "";
    };

    // If the user selects a contact,
    // show the full name
    pickerDelegate.ContactSelected += (contact) => {
        this.SelectedContactField.Text = $"{contact.GivenName}
{contact.FamilyName}";

        // If the contact has phone numbers
        if (contact.PhoneNumbers != null)
        {
            // Query for mobile only
            // Each PhoneNumber has a label with a description
            // and a Value with the actual phone number
            // exposed by the StringValue property
            var mobilePhone = contact.PhoneNumbers.
                Where(p => p.Label.Contains("Mobile")).
                Select(p => p.Value.StringValue);

            // If at least one mobile phone number
            if (mobilePhone != null)
            {
```

```

        // Generate a new data source
        var tblSource = new TableSource(mobilePhone.ToArray(), this);
        // and perform data-binding
        this.TableView.Source = tblSource;
    }
}

};

pickerDelegate.ContactPropertySelected += (property) => {
    this.SelectedContactField.Text = property.Value.ToString();
};

// Display picker
PresentViewController(picker, true, null);
}

}
}

```

Come ulteriori considerazioni, è bene precisare che la classe `NSString` è esposta da Cocoa e corrisponde al normale tipo `string`, mentre la classe `CNContactKey` rappresenta le varie informazioni di un contatto sotto forma di coppia chiave/valore, dove la chiave è un'etichetta (`Label`) e il valore è una stringa (`Value`). Infine, [il codice genera una nuova `TableSource` passando l'array di numeri di telefono come sorgente dati, che viene poi assegnata in data-binding alla Table View.](#)

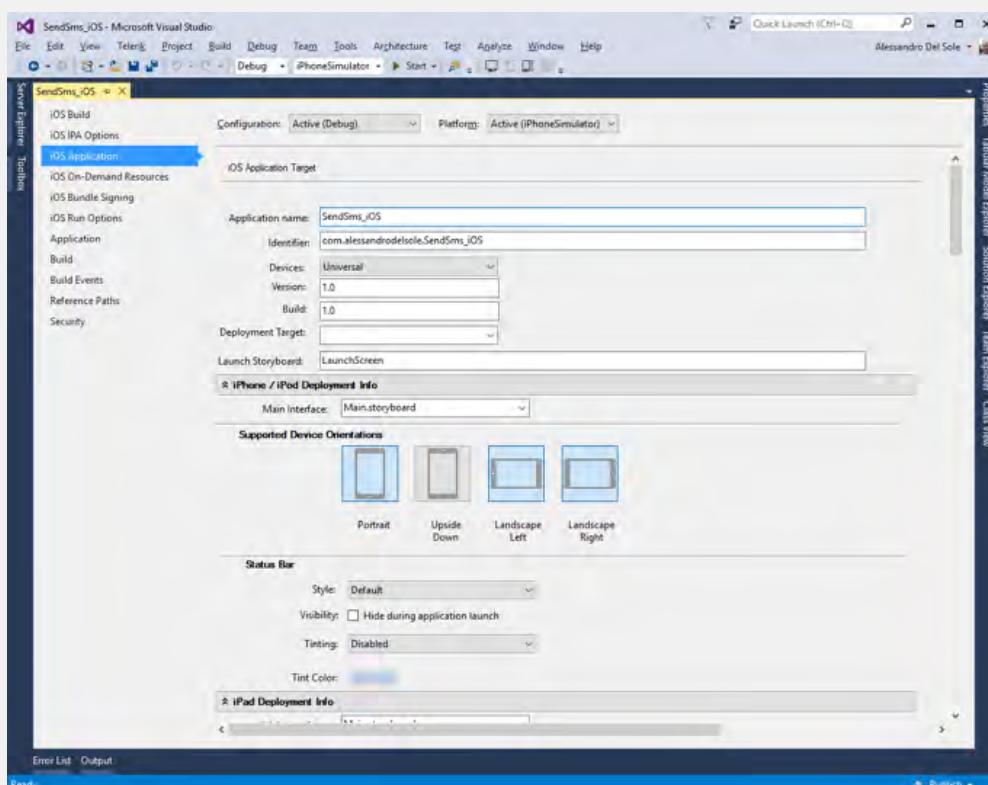


Le proprietà del progetto

Le proprietà del progetto vengono impostate attraverso il file Info.plist, per il quale Visual Studio 2015 fornisce un apposito designer a schede, visibile in **Figura 2-5**.

Nella scheda iOS Application, oltre all'identificazione dell'app visibile in **Figura 2-5**, è possibile impostare le icone dell'applicazione. [Nella scheda iOS Build è possibile specificare l'SDK da usare e la versione di destinazione di iOS](#). Particolare importanza riveste la scheda iOS Bundle Signing, all'interno della quale verranno specificati i provisioning profiles per le identità dello sviluppatore.

Figura 2-5 Le proprietà del progetto secondo iOS





Configurazioni di build

Oltre alle normali configurazioni Debug e Release, Xamarin.iOS integra le configurazioni Ad-Hoc e AppStore, da utilizzare per la distribuzione su dispositivo fisico e sull'Apple Store.

Dalla barra degli strumenti standard è anche possibile selezionare il dispositivo su cui avviare l'app.

Scegliendo iPhoneSimulator, si potrà selezionare una serie di configurazioni per il simulatore che rappresentano varie versioni di iPad e iPhone. Se avete un device fisico collegato al Mac, potrete selezionare iPhone come dispositivo target. E' importante ricordare che, in questo caso, è necessario aver configurato Xcode con il proprio account sviluppatore, che non è invece necessario col simulatore.



Avvio dell'applicazione

Un'app Xamarin.iOS può essere avviata con **F5 (con debugger)** o **Ctrl + F5 (senza debugger)** esattamente come in qualunque altro contesto .NET.

Ciò vuol dire che tutti i consueti strumenti di debug saranno disponibili. La compilazione passa tramite il Mac connesso e produce un file .ipa, che costituisce il pacchetto dell'app che verrà caricato sul dispositivo.

Supponendo di aver selezionato iPhone Simulator come dispositivo e una delle configurazioni per iPad, all'avvio l'app si presenta come in **Figura 2-6**, in cui si vede un contatto già selezionato.

Nota: si ricorda che, a partire dai rilasci del Cycle 8 di Xamarin, il simulatore iOS per Windows è disponibile solo con l'edizione Enterprise di Visual Studio 2015 e 2017, ma è ugualmente possibile sfruttare il simulatore direttamente su Mac. Una volta toccato il numero di telefono, verrà avviata l'app di messaggistica per l'invio di sms.

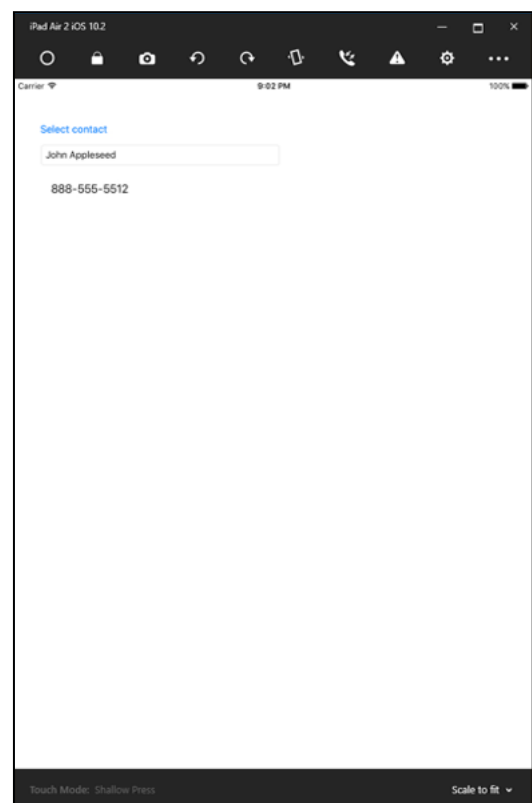


Figura 2-6 L'applicazione in esecuzione nel simulatore iOS



Pubblicazione sull'Apple Store

La pubblicazione di un'app in formato .ipa sull'Apple Store richiede una serie di passaggi, tra cui la registrazione al servizio (a pagamento), la produzione di icone in vari formati e test per la verifica dell'aderenza alle policy Apple. La **documentazione ufficiale** vi guiderà passo per passo nella procedura di pubblicazione.

Risorse di approfondimento

La documentazione di Xamarin.iOS è molto esaustiva dalla configurazione del Mac, allo sviluppo, passando per la conoscenza dei framework Apple e della struttura delle applicazioni. E' raggiungibile a questo **indirizzo**.



Riepilogo

Xamarin.iOS consente di accedere alle API di iPhone, iPad, Apple Watch e tvOS utilizzando C# e librerie .NET che fungono da wrapper dell'SDK Apple e di framework come Cocoa.

In questo modo, e grazie alla strumentazione di Microsoft Visual Studio, è possibile sviluppare app native riutilizzando le proprie competenze.

L'IDE consente di gestire tutte le proprietà dell'applicazione, incluse quelle di build e dei metadati, fino alla produzione del pacchetto da distribuire sull'Apple Store.

Cap. 3

Introduzione
a Xamarin.Android



Xamarin.iOS è l'insieme di strumenti e librerie che consentono di sviluppare app native per iPad, iPhone, Apple Watch e tvOS con C# tramite Visual Studio e Visual Studio for Mac. Con riguardo alle librerie, [Xamarin.iOS fornisce accesso anche a tutti i framework Apple](#), come Cocoa (fondamentale per lo sviluppo di app), e la denominazione delle classi richiama proprio quelle dei framework Apple per uniformità. [Xamarin.iOS consente inoltre di sviluppare applicazioni Universal ed offre strumenti di progettazione integrati nell'ambiente di lavoro](#). In questo capitolo viene fatta un'introduzione a Xamarin.iOS con una semplice app di esempio, ponendo l'accento sulla strumentazione necessaria e sulla relativa configurazione. Questo tornerà utile anche quando verrà illustrato Xamarin.Forms.

Nota: In questo libro viene utilizzato Microsoft Visual Studio 2015 come ambiente di sviluppo, ma gli stessi concetti si applicano anche a Visual Studio 2017, Visual Studio for Mac e Xamarin Studio.



Introduzione allo sviluppo

Per sviluppare app per Android, [Visual Studio](#) (e [Xamarin Studio](#)) mette a disposizione una serie di template di progetto disponibili nella cartella Android sotto il nodo Visual C#, come visibile in **Figura 3-1**.

Come vedete, è possibile creare app di diverso tipo e forma, come per smartphone/tablet e dispositivi wearable, ma anche giochi. [Ciascun template fornisce l'infrastruttura di partenza per la tipologia di app prescelta](#). E' anche possibile creare librerie e progetti per unit testing.

In questo capitolo viene proposto un esempio di app in grado di scattare una foto, di visualizzarla nell'interfaccia grafica e di inviarla come allegato di un'email. Questo tipo di esempio sarà utile per capire vari concetti alla base del funzionamento delle app su Android. Come modello di progetto è sufficiente utilizzare quello chiamato Single-View App (Android), evidenziato in **Figura 3-1**, che consente di creare un'app a pagina singola. Potete assegnare al progetto il nome che volete oppure, per uniformità, assegnare lo stesso della **Figura 3-1**. Fate click su OK ed attendete la generazione della solution.

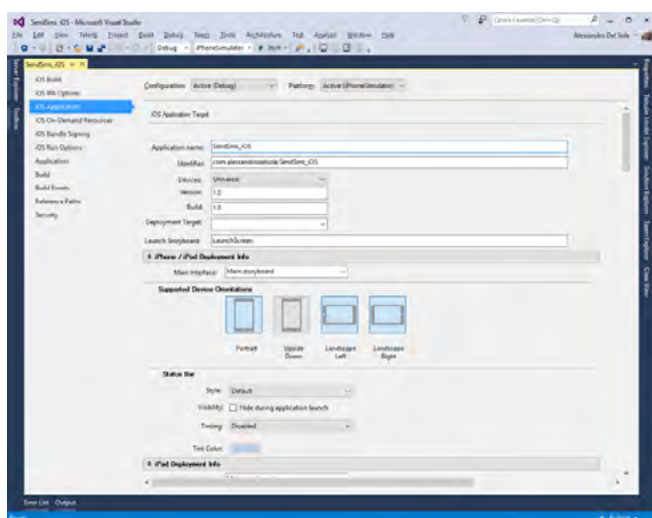


Figura 2-5 Le proprietà del progetto secondo iOS



Concetti fondamentali sulle solution Xamarin.Android

Nota: l'esempio proposto in questo capitolo è costruito sulla base di due progetti open source discussi nella documentazione di Xamarin.Android ed ospitati su GitHub, uno relativo alla **cattura delle immagini** e uno relativo all'**invio di email**.

Una tipica solution generata con Xamarin.Android contiene una serie di importanti elementi caratteristici, molti dei quali riflettono l'infrastruttura tipica delle app progettate per questo sistema operativo e, di fatto, è la stessa struttura di file prodotta da ambienti nativi come Android Studio ed Eclipse, cosa che rende estremamente semplice importare file esistenti. Tra gli elementi più importanti, in Solution Explorer noterete i seguenti:

- una cartella chiamata **Assets**, all'interno della quale potrete aggiungere file che volete siano distribuiti con la vostra app, ad eccezione di immagini che sono discusse di seguito;
- una cartella chiamata **Resources**, all'interno della quale confluiranno immagini, icone, file relativi al design dell'interfaccia grafica, risorse di tipo stringa, colori e costanti di localizzazione;
- una sottocartella chiamata **Resources\drawable**, all'interno della quale andranno inserite immagini ed icone. E' possibile fornire più risoluzioni per la stessa immagine ed Android sarà in grado di risolvere automaticamente l'immagine richiesta a seconda della risoluzione del display rilevata sul dispositivo corrente. In questo caso, è necessario creare più sottocartelle il cui nome

inizia col prefisso drawable- seguite dal valore che rappresenta i dpi, come hdpi, ldpi, xhdpi, xxhdpi e xxxhdpi;

- una sottocartella chiamata **Resources\layout**, all'interno della quale risiedono normalmente file .xml, che definiscono pagine ed elementi dell'interfaccia grafica;
- una sottocartella chiamata **Resources\values**, in cui è presente un file chiamato string.xml che definisce una serie di risorse di tipo stringa. Questo concetto è fondamentale in Android e tra breve verrà approfondito;
- un file chiamato **MainActivity.cs**, che rappresenta l'oggetto di avvio dell'applicazione.

L'obiettivo di questo capitolo è creare un'app in grado di attivare la fotocamera, catturare una foto e inviarla come allegato di un'email.

L'interfaccia grafica viene realizzata ed offerta dall'activity principale mentre la cattura, la visualizzazione e l'invio della foto sono realizzati tramite degli intent.

Prima di passare alla scrittura di codice per la creazione dell'esempio, è opportuno soffermarsi su alcuni concetti fondamentali di Android, proprio come quelli di activity e di intent.



Il concetto di Activity

Un **Activity** in Android rappresenta un oggetto grazie al quale l'utente interagisce col dispositivo tramite interfaccia grafica.

Un'Activity può essere pensata come a una pagina che contiene una serie di elementi visuali, il cui insieme serve ad eseguire un'attività.

In Xamarin.Android esiste una classe astratta chiamata `Android.App.Activity` che implementa l'infrastruttura necessaria e che costituisce un collegamento con l'omonima classe in Java (in Android tutta l'interfaccia utente è scritta in Java).

In linea di massima, nello sviluppo per Android, non si parla di pagine ma di Activity.

Altro motivo per cui le Activity sono importanti è quello secondo cui il ciclo di vita dell'applicazione è, di fatto, il ciclo di vita di ogni Activity.

Il concetto di Intent

L'**Intent** in Android rappresenta una singola operazione da eseguire ed è rappresentata dalla classe `Android.Content.Intent`.

Le informazioni richieste dall'Intent sono tipicamente due: l'azione da eseguire, come l'invio di un'email, e i dati da elaborare, come il destinatario o gli allegati.

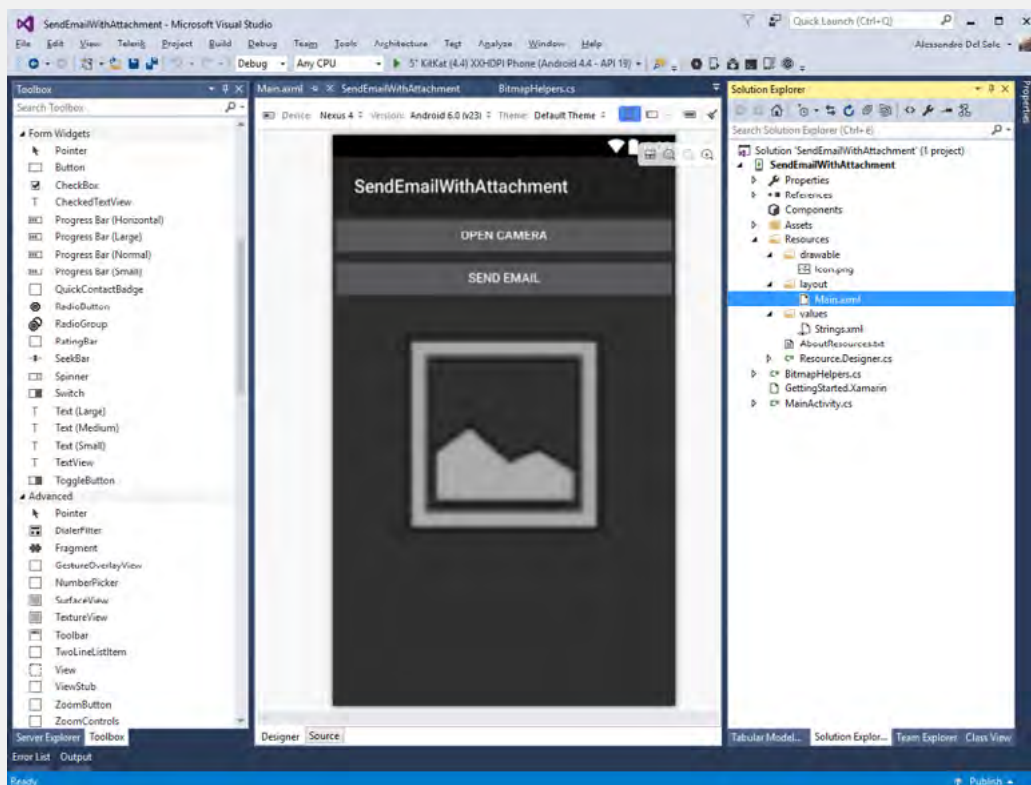


Progettazione dell'interfaccia

La progettazione dell'interfaccia grafica in Xamarin.Android può avvenire in due modalità: attraverso la finestra di progettazione o attraverso l'editor XML dei file .axml.

Se ad esempio fate doppio click sul file Main.axml sotto Resources\layout, Visual Studio mostrerà la finestra di progettazione su cui sarà possibile trascinare gli elementi dalla casella degli strumenti. La **Figura 3-2** mostra un esempio basato sul progetto da realizzare in questo capitolo.

Figura 3-2 La finestra di progettazione in Visual Studio



Nella terminologia di Android, i controlli utente si chiamano Widget e vengono organizzati in contenitori definiti Layout. L'insieme di Widget e Layout costituisce una View, che può essere l'intera interfaccia grafica di una pagina o solo un suo elemento complesso.

Nella **Figura 3-2** sono visibili due widget di tipo Button e un widget di tipo ImageView per la visualizzazione di un'immagine, organizzati in un LinearLayout.

La **documentazione** fornisce tutti i dettagli su widget e layout disponibili. Ciò premesso, sulla superficie di design troverete già un Button. Trascinatene un secondo e poi trascinate un ImageView. Attraverso la finestra **Properties (F4)** è possibile assegnare o modificare i valori delle proprietà di ciascun widget o layout. Quelle di interesse per il progetto sono collocate in un gruppo chiamato Main. In particolare si tratta di **id** e **text**. La prima serve a identificare l'elemento, come se fosse il nome, la seconda rappresenta il testo che contiene. Tuttavia, noterete che non si tratta del semplice valore testuale, ma di identificatori che puntano a delle altre stringhe.

Queste stringhe sono definite nel file Strings.xml sotto Resources\values. Per modificare la stringa relativa all'id si può semplicemente fare doppio click sull'elemento e modificare la stringa nel popup che appare. Per il testo corrispondente, si può aprire il file Strings.xml.

Ecco come dovranno essere specificate le stringhe a questo punto:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="ApplicationName"> SendEmailWithAttachment</string>
    <string name="openCamera">Open Camera</string>
    <string name="sendEmail">Send email</string>
</resources>
```

Questa struttura permette altresì di localizzare le applicazioni con maggiore facilità.

Se volete poi vedere com'è definita l'interfaccia grafica in modalità dichiarativa, o avere un maggior controllo su di essa, potete visualizzare il markup XML relativo al file .axml su cui state lavorando.

Per esempio, se in Solution Explorer fate click destro sul file Main.xml, l'editor di Visual Studio mostrerà il codice per la pagina corrente, riportato nel **Listato 3-1**.

Il markup XML rappresenta gli elementi visuali in forma gerarchica (un concetto noto a chi proviene da XAML e che sarà ripreso nel Capitolo 4), dove un elemento XML rappresenta un layout o widget mentre gli attributi XML le sue proprietà, per ciascuna delle quali il nome comincia tipicamente col prefisso android:

Notate come la proprietà android:id serva ad identificare l'elemento in modo che sia poi possibile gestirlo da codice C#.

Listato 3-1

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:id="@+id/captureButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/openCamera" />
    <Button
        android:id="@+id/sendButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/sendEmail" />
    <ImageView
        android:src="@android:drawable/ic_menu_gallery"
        android:layout_width="fill_parent"
        android:layout_height="300.0dp"
        android:id="@+id/imageView1"
        android:adjustViewBounds="true" />
</LinearLayout>
```



Accesso alle API di Android

Dopo aver definito l'interfaccia grafica, è il momento di scrivere codice C# che la renda operativa. In Xamarin.iOS questo avviene nei `ViewController`, mentre in Xamarin.Android avviene nel codice delle classi `Activity`. Le attività da eseguire sono diverse, quindi il codice verrà scomposto in varie parti e verranno fatte le opportune considerazioni.

Per prima cosa, nel namespace di primo livello inserite questa classe, che ha lo scopo di memorizzare alcune informazioni necessarie all'app come il file, la cartella delle foto e l'immagine in memoria:

```
public static class App
{
    public static File _file;
    public static File _dir;
    public static Bitmap bitmap;
}
```

Android deve sapere qual è l'`Activity` di avvio, quindi questa dev'essere decorata con l'attributo `Activity` in cui la proprietà `MainLauncher` sia `true`:

```
using System;
using System.Collections.Generic;
using Android.App;
using Android.Content;
using Android.Content.PM;
using Android.Graphics;
using Android.OS;
using Android.Provider;
using Android.Widget;
using Java.IO;
using Environment = Android.OS.Environment;
using Uri = Android.Net.Uri;
using SendEmailWithAttachment;
```

```

namespace SendEmailWithAttachment
{
    [Activity(Label = "Send Email With Attachment", MainLauncher = true, Icon = "@drawable/icon")]
    public class MainActivity : Activity
    {
        private ImageView _imageView;
        Uri fileName;
    }
}

```

Deve sempre esserci un'Activity con MainLauncher = true in ogni applicazione.

Icon consente di specificare l'icona che verrà visualizzata nella pagina iniziale di Android (il cosiddetto launcher) mentre Label indica il titolo dell'activity.

Notate la presenza di due campi che serviranno per avere un riferimento all'immagine e al suo nome.

Il punto di ingresso di un'activity è un metodo chiamato OnCreate, nel quale viene eseguito codice di inizializzazione. Nel caso corrente vengono chiamati due metodi, uno che verifica la disponibilità di app per la cattura di foto e uno che crea una sottocartella per memorizzare foto:

```

using System;
using System.Collections.Generic;
using Android.App;
using Android.Content;
using Android.Content.PM;
using Android.Graphics;
using Android.OS;
using Android.Provider;
using Android.Widget;
using Java.IO;
using Environment = Android.OS.Environment;
using Uri = Android.Net.Uri;
using SendEmailWithAttachment;

namespace SendEmailWithAttachment
{
    [Activity(Label = "Send Email With Attachment", MainLauncher = true, Icon = "@drawable/icon")]
    public class MainActivity : Activity
    {
        private ImageView _imageView;
        Uri fileName;
    }
}

```

Notate come vengano ottenuti riferimenti ai widget tramite il metodo `FindViewById` a cui viene passato l'id risorsa che Xamarin.

Android ha mappato in forma di costanti dentro la classe `Resource.Id`. Infine, notate come vengano assegnati due gestori di evento `Click` ai pulsanti con le modalità tipiche di C#. Di seguito i due metodi citati sopra:

```
private void CreateDirectoryForPictures()
{
    App._dir = new File(
        Environment.GetExternalStoragePublicDirectory(
            Environment.DirectoryPictures), "CameraAppDemo");
    if (!App._dir.Exists())
    {
        App._dir.Mkdirs();
    }
}

private bool IsThereAnAppToTakePictures()
{
    Intent intent = new Intent(MediaStore.ActionImageCapture);
    IList<ResolveInfo> availableActivities =
        PackageManager.QueryIntentActivities(intent, PackageManagerFlags.MatchDefaultOnly);
    return availableActivities != null && availableActivities.Count > 0;
}
```

In Xamarin. Android, la gestione di file e cartelle avviene mediante la classe `Java.IO.File`.

Qui viene utilizzata per creare una cartella (`Mkdirs`) all'interno della cartella pubblica delle foto (`Environment.DirectoryPictures`).

L'altro metodo, `IsThereAnAppToTakePictures`, istanzia un nuovo `Intent` di tipo `ActionImageCapture`; il metodo `PackageManager.QueryIntentActivities` verifica quante activity siano disponibili sul sistema (`PackageManagerFlags.MatchDefaultOnly`) per eseguire l'operazione richiesta e restituisce al chiamante un valore vero o falso.



Accesso alla fotocamera e gestione delle immagini

A questo punto si può scrivere il primo gestore di evento Click per il pulsante che attiva la fotocamera. Tale attivazione avviene sempre tramite l'intent visto nel frammento precedente, che però stavolta viene effettivamente lanciato:

```
private void CaptureButton_Click(object sender, EventArgs EventArgs)
{
    Intent intent = new Intent(MediaStore.ActionImageCapture);

    App._file = new File(App._dir, $"IMG_{Guid.NewGuid()}.jpg");
    this.fileName = Uri.FromFile(App._file);

    intent.PutExtra(MediaStore.ExtraOutput, this.fileName);

    StartActivityForResult(intent, 0);
}
```


Come vedete, l'intent che riceve tutte le informazioni del caso, come ad esempio il file di output tramite il metodo PutExtra. Questo avviene indipendentemente dall'applicazione di cattura che verrà usata, così l'utente sarà libero di scegliere ma le informazioni hanno un formato unico.

Il file è rappresentato da un tipo `Android.Net.Uri` e viene costruito tramite il metodo `Uri.FromFile`.

Poiché l'intent viene effettivamente eseguito tramite l'interfaccia di un'app, questa dev'essere avviata col metodo `StartActivityForResult` a cui viene passato l'intent da eseguire. In questo caso particolare, viene anche passato un intero che corrisponde al codice restituito dall'intent in caso di operazione eseguita con successo. Una volta che l'activity viene completata, chiamerà un metodo chiamato `OnActivityResult`, all'interno del quale si può gestire ed elaborare il risultato dell'activity stessa.

In questo caso, viene utilizzato un intent per capire dove debba essere salvato il file (`ActionMediaScannerScanFile`), che poi viene visualizzato nella `ImageView`:

```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
{
    base.OnActivityResult(requestCode, resultCode, data);

    if(requestCode==0 && resultCode==Result.Ok)
    {
        // Make it available in the gallery

        Intent mediaScanIntent = new Intent(Intent.ActionMediaScannerScanFile);
        Uri contentUri = Uri.FromFile(App._file);
        mediaScanIntent.SetData(contentUri);
        SendBroadcast(mediaScanIntent);

        // Display in ImageView. We will resize the bitmap to fit the display
        // Loading the full sized image will consume too much memory
        // and cause the application to crash.

        int height = Resources.DisplayMetrics.HeightPixels;
        int width = _imageView.Height;
        App.bitmap = App._file.Path.LoadAndResizeBitmap(width, height);
        if (App.bitmap != null)
        {
            _imageView.SetImageBitmap(App.bitmap);
            App.bitmap = null;
            // Dispose of the Java side bitmap.
            GC.Collect();
        }
    }
}
```

Tre considerazioni:

- viene rilevata l'altezza del display tramite l'oggetto `DisplayMetrics.HeightPixels`;
- viene utilizzato un [metodo extension](#) chiamato `LoadAndResizeBitmap` per ridimensionare l'immagine catturata per la visualizzazione;
- viene invocata una [garbage collection](#) (`GC.Collect()`) per liberare memoria lato Java. Questo perché un oggetto immagine rappresentato in C# occupa circa 20 Kb ma, dietro le scene, il corrispondente oggetto Java può occupare anche più megabyte.

Il metodo extension `LoadAndResizeBitmap` è definito in una classe separata come questa:

```
using Android.Graphics;

namespace SendEmailWithAttachment
{
    public static class BitmapHelpers
    {
        public static Bitmap LoadAndResizeBitmap(this string fileName, int width, int height)
        {
            // First we get the the dimensions of the file on disk
            BitmapFactory.Options options = new BitmapFactory.Options { InJustDecodeBounds = true };
            BitmapFactory.DecodeFile(fileName, options);

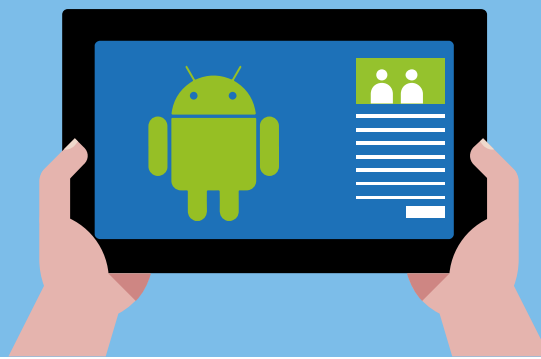
            // Next we calculate the ratio that we need to resize the image by
            // in order to fit the requested dimensions.
            int outHeight = options.OutHeight;
            int outWidth = options.OutWidth;
            int inSampleSize = 1;

            if (outHeight > height || outWidth > width)
            {
                inSampleSize = outWidth > outHeight
                    ? outHeight / height
                    : outWidth / width;
            }

            // Now we will load the image and have BitmapFactory resize it for us.
            options.InSampleSize = inSampleSize;
            options.InJustDecodeBounds = false;
            Bitmap resizedBitmap = BitmapFactory.DecodeFile(fileName, options);

            return resizedBitmap;
        }
    }
}
```

Questo è quanto serve per lanciare l'intent di cattura della foto, elaborarne il risultato e visualizzarlo nell'interfaccia grafica.



API di rete: invio di email

La parte successiva riguarda l'invio di email. Come potete intuire, anche questa operazione avviene tramite un intent istanziato nel gestore dell'evento Click dell'apposito pulsante.

ActionSend è l'intent per l'invio di una comunicazione; che si tratti di un'email è specificato tramite ExtraEmail e un array di indirizzi. E' anche possibile specificare un oggetto (ExtraSubject) e l'allegato tramite ExtraStream. [E' importante che il messaggio abbia il tipo MIME message/rfc822](#). Viene poi avviata un'activity che lascerà all'utente la scelta dell'app per l'invio (Intent.CreateChooser).

[Notate l'utilizzo della classe PackageManager](#) che, tra l'altro, consente di verificare l'esistenza di una intent adatta alla necessità del caso. Tutto quello che serve è stato implementato, ma [prima di avviare l'applicazione è necessario esaminare le proprietà del progetto e in particolare il manifest](#).

```
private void SendButton_Click(object sender, EventArgs e)
{
    if(this.fileName != null)
    {
        var email = new Intent(Intent.ActionSend);

        // Check if at least one activity exists for the specified intent
        if(PackageManager.QueryIntentActivities(email, PackageInfoFlags.MatchDefaultOnly).
Any())
        {
            email.PutExtra(Intent.ExtraEmail, new[] { "someone@email.com" });
            email.PutExtra(Intent.ExtraSubject, "Sample email with attachment");
            email.PutExtra(Intent.ExtraStream, fileName);

            email.SetType("message/rfc822");

            StartActivity(Intent.CreateChooser(email, "Email"));
        }
    }
}
```



Le proprietà del progetto

Facendo doppio click su **Properties in Solution Explorer**, si accederà alle proprietà del progetto. Nella scheda **Application** è possibile definire le impostazioni relative alla versione dell'SDK e di compilazione.

La scheda **Android Options** consente di influenzare la generazione del file .apk, il pacchetto compilato che sarà poi distribuito e pubblicato sul Google Play.

Ma probabilmente la scheda più importante è **Android Manifest**, perché è qui che dovranno essere impostate proprietà che riguardano il comportamento dell'app sul device. La **Figura 3-3** mostra il manifest nel progetto Xamarin.Android.

Qui è possibile selezionare l'icona, specificare il nome del pacchetto per come comparirà nell'elenco di app, specificare se è consentita l'installazione su scheda di memoria (su sistemi che lo consentono) e soprattutto è possibile (e importante) specificare i permessi che l'app deve richiedere per poter eseguire le operazioni.

Nel caso in esame è necessario selezionare i permessi **READ_EXTERNAL_STORAGE** e **WRITE_EXTERNAL_STORAGE** cosicché l'applicazione possa anche scrivere e leggere un'eventuale scheda di memoria. **Non impostare i permessi necessari sarà causa di eccezione a runtime.**

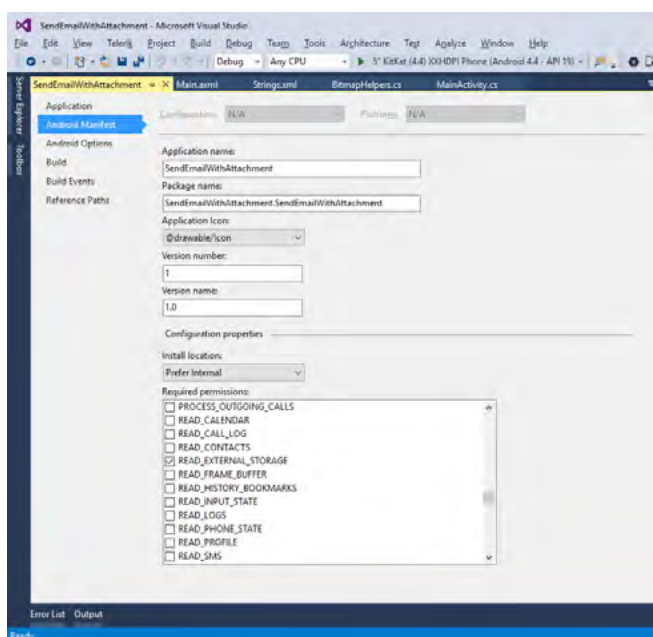


Figura 3-3 Il manifest di Android



Configurazioni di build

Come per altre piattaforme .NET, anche per Xamarin.Android sono disponibili le configurazioni Debug e Release per eseguire la build della soluzione. In entrambi i casi, viene generato un file .apk che rappresenta il pacchetto dell'app, oltre ovviamente ai file di supporto necessari.

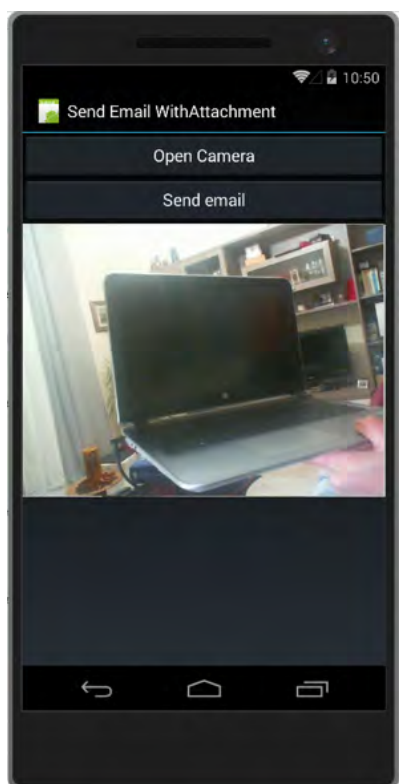


Figura 3-4 L'applicazione eseguita nell'emulatore

Avvio e debug dell'applicazione

E' possibile eseguire l'applicazione (sia in Debug che Release) sia su emulatore che su dispositivo fisico. L'elenco di emulatori disponibili sarà visibile nella barra degli strumenti di Visual Studio, vicino al pulsante di avvio del debug, e varierà a seconda degli SDK che avete installato.

In questo capitolo viene preso in considerazione il Visual Studio Emulator for Android, disponibile tramite l'installer di Visual Studio. Al primo avvio, Visual Studio chiederà il vostro consenso per configurare l'emulatore all'interno di Hyper-V. Nel caso di un dispositivo fisico, è dapprima necessario attivare la modalità sviluppatore. Per farlo è sufficiente andare nelle impostazioni del telefono, poi nelle informazioni e poi toccare per 7 volte il numero di build del sistema operativo. Poi sarà sufficiente collegare il telefono o tablet alla porta USB del PC e l'IDE lo riconoscerà immediatamente. La **Figura 3-4** mostra l'applicazione di esempio in esecuzione nell'emulatore.



Pubblicazione nel Google Play

Al termine della fase di sviluppo, se volete pubblicare la vostra app sul Google Play dovrete eseguire una serie di passaggi che includono la registrazione al servizio (a pagamento), la preparazione di apposite icone e la firma tramite certificato. La [documentazione ufficiale](#) vi guiderà passo per passo verso la pubblicazione.

Risorse di approfondimento

Il portale per gli sviluppatori di Xamarin include tutta la documentazione per sviluppare con Xamarin.Android, dalle basi fino alla pubblicazione passando per l'interfaccia grafica e il ciclo di vita dell'applicazione. Potete consultarla a questo [indirizzo](#).



Riepilogo

Xamarin.Android consente di accedere alle API di del sistema Android utilizzando C# e librerie .NET che fungono da wrapper degli SDK di Java e Google.

In questo modo, e grazie alla strumentazione di Microsoft Visual Studio, è possibile sviluppare app native riutilizzando le proprie competenze. L'IDE consente di gestire tutte le proprietà dell'applicazione, incluse quelle di build e dei metadati, fino alla produzione del pacchetto da distribuire sul Google Play.

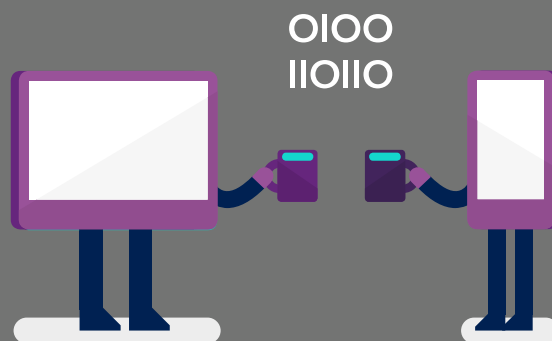
Cap. 4

Introduzione
a Xamarin.Forms



Nei capitoli precedenti sono stati introdotti Xamarin.iOS e Xamarin.Android. Pur essendo piattaforme molto efficaci, il limite maggiore nell'utilizzarle è la grande difficoltà nel condividere il codice per l'interfaccia grafica. Ciò significa che, volendo sviluppare per iOS e Android, è necessario creare due solution diverse che producano pacchetti diversi, scrivendo codice C# che utilizza API diverse e oggetti diversi soprattutto quando si tratta di interfaccia grafica. Inoltre, fino a questo punto lo sviluppo di app per Windows è escluso da Xamarin. E' proprio a questo punto che entra in gioco Xamarin.Forms, uno framework per condividere l'interfaccia grafica e che consente di massimizzare il riutilizzo del codice.

Nota: questo capitolo ha scopo introduttivo, pertanto si faranno spesso rimandi alla **documentazione ufficiale** tramite appositi link.



Xamarin.Forms, tra condivisione di codice e interfacce grafiche

Nei capitoli precedenti avete visto come [Xamarin.iOS](#) e [Xamarin.Android](#) generino progetti autonomi, nei quali l'interfaccia si progetta mediante oggetti tipici delle due piattaforme e con specifica strumentazione.

Da codice C#, poi, si accede alle API native dei sistemi e framework di destinazione. Xamarin.Forms fa un passo in avanti, offrendo uno strato che consente di definire interfacce grafiche una sola 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#. Per fare questo, [Xamarin.Forms mappa i controlli nativi in apposite classi, che sono poi responsabili di disegnare l'elemento visuale appropriato a seconda del sistema su cui l'applicazione sta girando](#). Grazie a questo approccio, in un'unica solution è possibile avere progetti iOS, Android ma anche Universal Windows Platform, Windows Phone 8.1 e Windows 8.1.

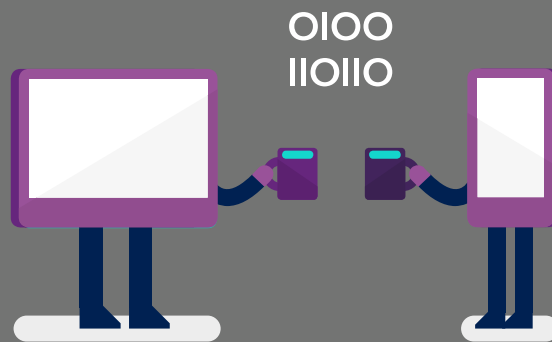
Questo codice condiviso, quindi l'interfaccia grafica e le API per data-binding e il servizio di dipendenza per l'accesso a funzionalità di piattaforma, è contenuto in uno specifico progetto all'interno della solution e può essere, ad oggi, di due tipi: una Portable Class Library (PCL) o uno Shared Project. Nel prossimo capitolo verranno analizzate più in dettaglio le differenze tra i due e verranno fatti anche dei

cenni alle librerie .NET Standard. In questo capitolo, per praticità espositiva, verrà utilizzata la struttura basata su PCL. All'interno del progetto condiviso, lo sviluppatore scriverà tutto il codice che può funzionare con certezza su tutte le piattaforme. Sebbene questo nuovo approccio sia estremamente valido e risolutivo di molti problematiche, poiché finalmente per la prima volta si può scrivere una vera applicazione multi-piattaforma semplicemente condividendo codice, ci sono due 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](#). Per 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 (più avanti verrà comunque spiegato come ovviare al problema con i custom renderer). A partire da Xamarin.Forms 2.3, è anche possibile utilizzare controlli nativi.

Nota: L'elenco completo dei controlli disponibili in Xamarin.Forms è raggiungibile al seguente indirizzo: <https://developer.xamarin.com/guides/xamarin-forms/controls/>

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.

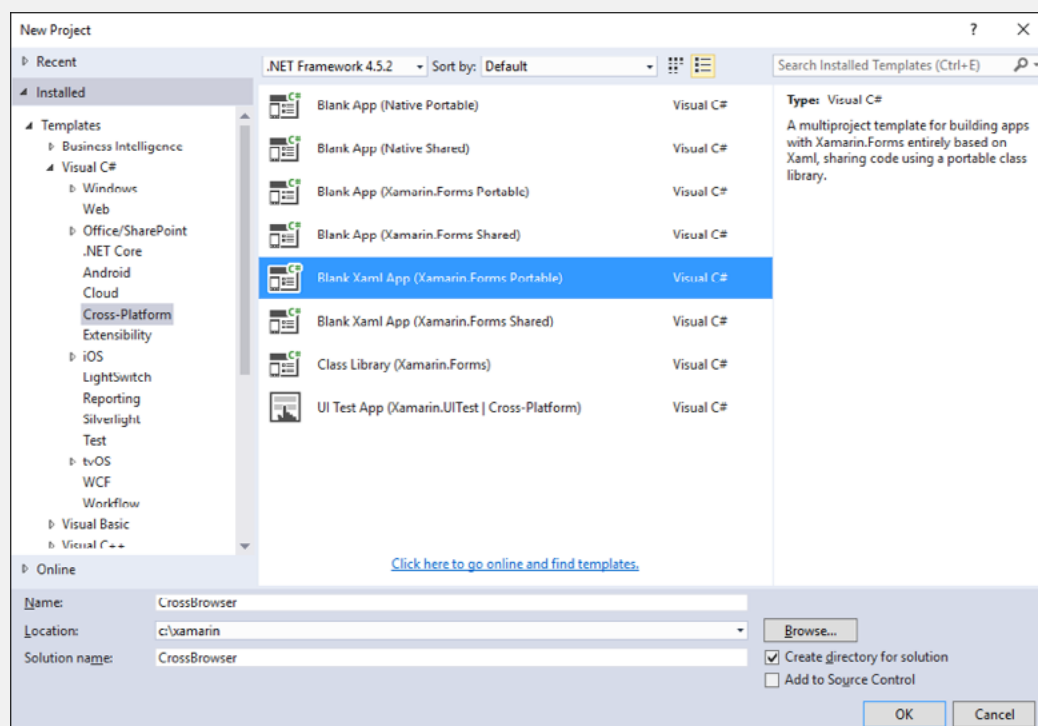
Xamarin.Forms è offerto attraverso l'omonimo pacchetto NuGet che, al momento della redazione di questo capitolo, è la 2.3.3.180. Il modo migliore per prendere confidenza con questa tecnologia è attraverso un esempio.



Creazione di un'applicazione con Xamarin.Forms

Nella finestra New Project di Visual Studio, sotto Visual C#, è disponibile un nodo chiamato **Cross-platform** che contiene una serie di template per progetti basati su Xamarin.Forms.

Figura 4-1 I template di progetto per Xamarin.Forms



Tra i vari template, ce n'è uno chiamato Cross Platform App (Xamarin.Forms or Native). Per creare un'app vera e propria è necessario selezionare questo template e assegnare un nome al progetto, che in questo caso sarà CrossBrowser, quindi click su OK. A questo punto comparirà una ulteriore dialog (vedi **Figura 4-1**) che consentirà di scegliere tra i modelli Blank App e Master Detail; il primo genera un progetto con la sola infrastruttura di partenza, il secondo genera un progetto basato su interfaccia di tipo master-detail e con alcuni dati dimostrativi. Il modello che verrà utilizzato in questo capitolo è il Blank App. La scelta nel riquadro UI Technology è chiaramente Xamarin.Forms, mentre nel riquadro Code Sharing Strategy la scelta ricadrà su Portable Class Library (PCL). Nel capitolo seguente verranno poi approfonditi gli scenari e le strategie di condivisione del codice. A questo punto fate click su OK per creare il progetto. Dopo alcuni secondi, vi verrà richiesto di scegliere la versione minima e massima di Windows 10 su cui l'app per UWP potrà essere eseguita. Il consiglio è quello di lasciare la

selezione suggerita. Ancora, vi verrà richiesto di specificare il Mac a cui connettersi esattamente come visto per Xamarin.iOS. Questo passaggio è ovviamente opzionale. Il numero di progetti generati varierà a seconda di quanti SDK avete installato; nel caso li abbiate tutti, in Solution Explorer vedrete sei progetti (PCL, iOS, Android, UWP, Windows Phone 8.1 e Windows 8.1). Non siete obbligati a tenerli tutti, potete escludere quelli che non vi interessano. Ciascun progetto di piattaforma ha la struttura che ci si aspetterebbe, quindi per i progetti Android e iOS ci sono rispettivamente MainActivity.cs e AppDelegate.cs e così via. Il cuore di questa solution è nel progetto PCL: è, qui infatti, che scriveremo il codice condiviso, incluso quello relativo alla definizione della UI. [Nel progetto di tipo PCL c'è un file chiamato App.xaml e il corrispondente App.xaml.cs.](#) Nel primo potremo specificare risorse (stili, data template) fruibili da tutta l'applicazione mentre nel secondo potremo gestire gli eventi del ciclo di vita e specificare la pagina di avvio, che in questo caso è un oggetto chiamato MainPage.

Listato 4-1

```
using Xamarin.Forms;

namespace CrossBrowser
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

            MainPage = new CrossBrowser.MainPage();
        }

        protected override void OnStart()
        {
            // Handle when your app starts
        }

        protected override void OnSleep()
        {
            // Handle when your app sleeps
        }

        protected override void OnResume()
        {
            // Handle when your app resumes
        }
    }
}
```

Il **Listato 4-1** mostra il codice di App.xaml.cs e della classe App che definisce.

Listato 4-1

```
using Xamarin.Forms;

namespace CrossBrowser
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

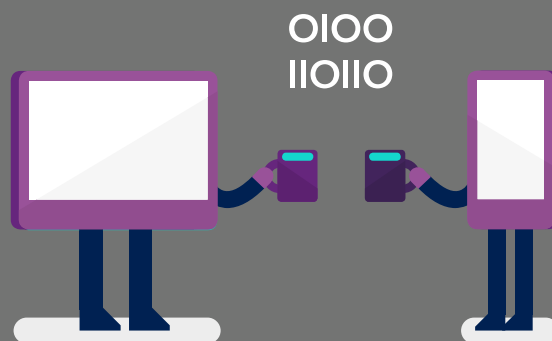
            MainPage = new CrossBrowser.MainPage();
        }

        protected override void OnStart()
        {
            // Handle when your app starts
        }

        protected override void OnSleep()
        {
            // Handle when your app sleeps
        }

        protected override void OnResume()
        {
            // Handle when your app resumes
        }
    }
}
```

Come vedete, vengono gestiti gli eventi del ciclo di vita dell'app con un'unica modalità, che va bene su tutti i sistemi operativi. Nei vari metodi si potrà quindi aggiungere codice da eseguire a seconda dell'evento, come per esempio la memorizzazione dello stato dell'app o dei suoi dati in **OnSleep** e il ripristino dello stato e dei dati in **OnResume**.



Organizzazione dell'interfaccia: pagine e layout

Come in tutte le altre piattaforme, anche in Xamarin.Forms l'interfaccia delle applicazioni è, a più alto livello, **organizzata in pagine**. Xamarin.Forms espone una classe base chiamata Page, da cui derivano una serie di altre pagine, come rappresentato in **Tabella 4-1**.

Tabella 4-1

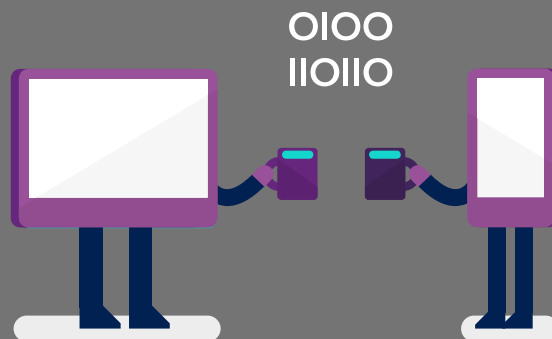
Tipo Pagina	Comportamento
ContentPage	Pagina singola per la visualizzazione di elementi dell'interfaccia. Utilizzabile sia da XAML che da code-behind
TabbedPage	Pagina a schede, che viene popolata con una collection di oggetti Page. Utilizzabile sia da XAML che da code-behind
MasterDetailPage	Pagina con un menu laterale scorrevole (master) e un'area di dettaglio (detail). Il menu è indicato dalla classica icona hamburger
NavigationPage	Oggetto che offre il supporto per la navigazione a stack. E' tipicamente utilizzato da code-behind e ad esso vengono aggiunte le pagine che fanno parte dello stack di navigazione
CarouselPage	Pagina per lo scorrimento di elementi, simile al Panorama di Windows Phone
TemplatedPage	Oggetto per la creazione di pagine personalizzate

Attraverso la `NavigationPage`, `Xamarin.Forms` offre tutta l'infrastruttura necessaria alla **navigazione** modale o a stack tra pagine. All'interno delle pagine, in `Xamarin.Forms`, al pari di piattaforme come WPF e UWP, **l'interfaccia grafica è organizzata in contenitori detti anche panel o layout**. Il loro scopo è quello di consentire il posizionamento e ridimensionamento dinamico della UI. Quelli disponibili sono riepilogati in **Tabella 4-2**.

Tabella 4-2

Tipo Layout	Comportamento
<code>ContentView</code>	Contenitore che, attraverso un unico elemento costituito da contenitori e controlli, rappresenta una gerarchia di elementi
<code>Grid</code>	Contenitore che può essere suddiviso in righe e colonne
<code>StackLayout</code>	Affianca elementi in orizzontale o in verticale
<code>AbsoluteLayout</code>	Contenitore posizionato alle coordinate specificate
<code>RelativeLayout</code>	Contenitore che viene posizionato sulla base di vincoli
<code>ScrollView</code>	Contenitore che consente lo scorrimento degli elementi che contiene
<code>Frame</code>	Contenitore che disegna un bordo intorno all'elemento che contiene

I layout possono essere nidificati tra loro per costruire gerarchie anche complesse. Il concetto di layout è fondamentale in `Xamarin` perchè le pagine possono contenere un unico elemento, che tipicamente è proprio un layout al cui interno è costruita la gerarchia visuale (nota come visual tree).

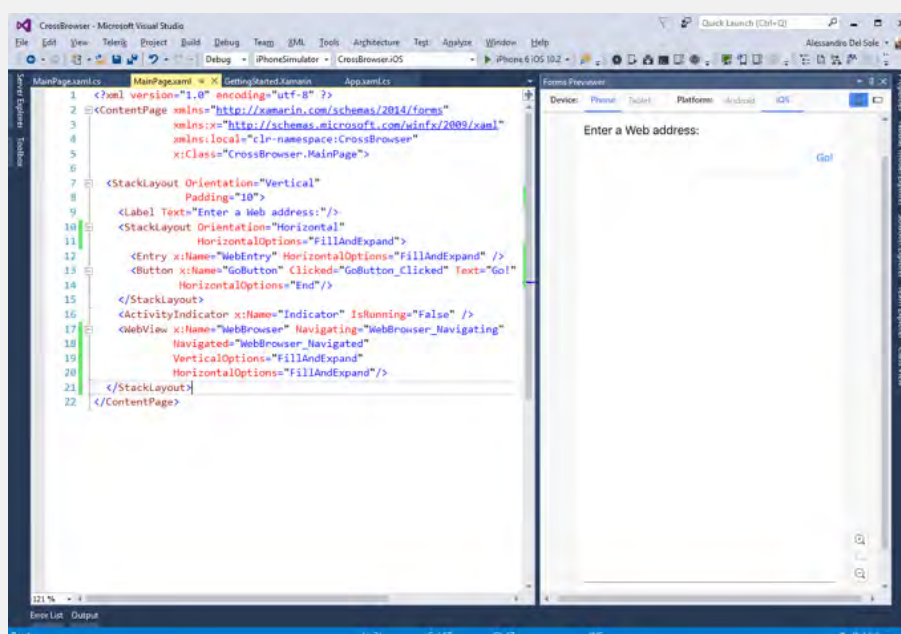


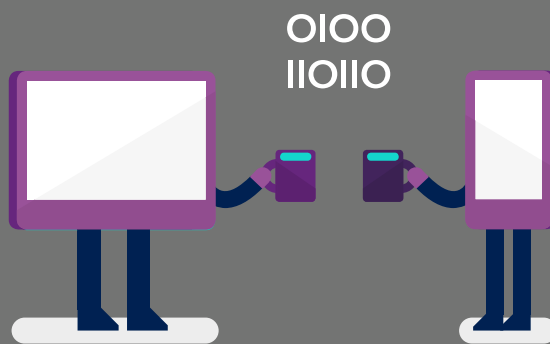
Progettazione dell'interfaccia grafica

L'interfaccia grafica viene progettata normalmente attraverso il linguaggio di markup XAML.

E' bene precisare che si può fare anche in C#, ma XAML è più adatto a rappresentare gerarchie di elementi e separa meglio lo strato visivo da quello operativo. Chiaramente, in alcuni contesti sarà necessario generare elementi visuali a runtime e, in quel caso, verrà fatto da C#. XAML è di derivazione XML e, in Xamarin, rispetta le specifiche XAML 2009. **In XAML, ad ogni elemento corrisponde una classe mentre ad ogni attributo corrispondono proprietà di quella classe.** Per quanto riguarda l'app di esempio, fate doppio click sul file MainPage.xaml. Successivamente, selezionate View, Other Windows, Xamarin.Forms Previewer. Questa finestra ancorabile offre un'anteprima visuale del codice che scrivete e, sebbene non sia un vero e proprio designer, è sicuramente molto utile per avere un'idea del layout. La **Figura 4-2** mostra l'editor di codice XAML e il previewer col risultato che otterremo.

Figura 4-2 – L'editor di codice XAML e il Previewer





API di rete: invio di email

Il **Listato 4-2** mostra come definire una semplice interfaccia grafica che contiene:

- Uno **StackLayout principale** come contenitore di elementi.
- Una **Label** per mostrare testo.
- Uno **StackLayout** nidificato e orientato in orizzontale per l'organizzazione gerarchica di altri elementi.
- Una **Entry** per inserire testo (in questo caso un URL).
- Un **Button** per avviare l'azione di navigazione.
- Un **ActivityIndicator** che mostra un indicatore di attività in corso.
- Un **WebView**, un controllo capace di visualizzare contenuti HTML non solo da siti Web.

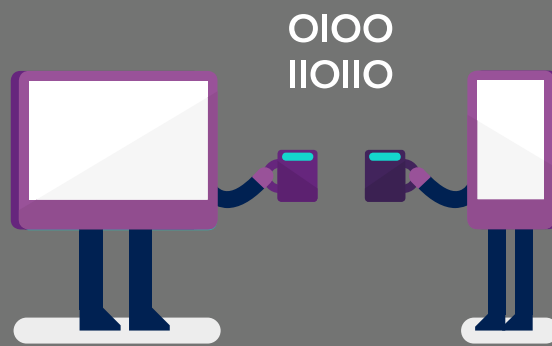
Listato 4-2

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

  <StackLayout Orientation="Vertical"
    Padding="10">
    <Label Text="Enter a Web address:" />
    <StackLayout Orientation="Horizontal" HorizontalOptions="FillAndExpand">
      <Entry x:Name="WebEntry" HorizontalOptions="FillAndExpand" />
      <Button x:Name="GoButton" Clicked="GoButton_Clicked" Text="Go!"
        HorizontalOptions="End" />
    </StackLayout>
    <ActivityIndicator x:Name="Indicator" IsRunning="False" />
    <WebView x:Name="WebBrowser" Navigating="WebBrowser_Navigating" Navigated="WebBrowser_Navigated"
      VerticalOptions="FillAndExpand" HorizontalOptions="FillAndExpand" />
  </StackLayout>

</ContentPage>
```

Alcune note su proprietà che sono disponibili a vari livelli: `x:Name` consente di assegnare un nome agli elementi, utile per interagire con essi da C#; `Padding` stabilisce lo spazio intorno ai controlli presenti in un contenitore ([Margin](#), invece, stabilisce la distanza tra elementi visuali); [HorizontalOptions](#) (e la corrispondente [VerticalOptions](#)) determinano la modalità con cui un elemento occupa lo spazio a disposizione. Nello XAML si possono anche specificare i gestori di evento come [Clicked](#) per il [Button](#) o [Navigating](#) e [Navigated](#) per il [WebView](#). I gestori veri e propri vengono poi implementati nel file di code-behind.



Il code-behind in C#

Per ciascun file `.xaml` esiste, di norma, un corrispondente file `.xaml.cs` definito anche di code-behind, che contiene il codice operativo della pagina e che rende attivi gli elementi visuali. Nel caso dell'esempio corrente, il file di code-behind contiene codice C# che serve ad aprire l'URL specificato dall'utente una volta che viene toccato il pulsante. Il **Listato 4-3** mostra il codice per questo esempio.

Listato 4-3

```
using Xamarin.Forms;

namespace CrossBrowser
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

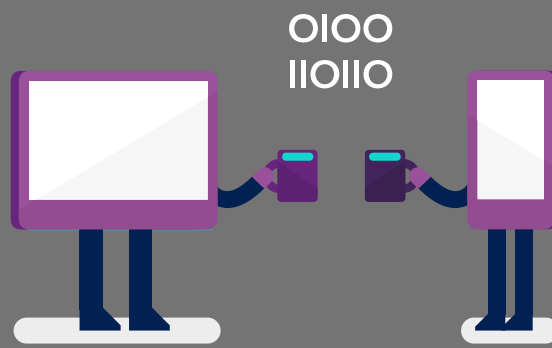
        private async void GoButton_Clicked(object sender, EventArgs e)
        {
            if (!string.IsNullOrEmpty(this.WebEntry.Text) &&
                this.WebEntry.Text.ToLower().StartsWith("http://"))
            {
                this.WebBrowser.Source = this.WebEntry.Text;
            }
            else
            {
                await DisplayAlert("Error", "URL is incorrect", "OK");
            }

            private void WebBrowser_Navigating(object sender, WebNavigatingEventArgs e)
            {
                this.Indicator.IsRunning = true;
            }
            private void WebBrowser_Navigated(object sender, WebNavigatedEventArgs e)
            {
                this.Indicator.IsRunning = false;
            }
        }
    }
}
```

Il codice è semplice: al tocco sul pulsante, rappresentato dall'evento `Clicked`, viene verificato il contenuto della `Entry` e, se valido, assegnato l'URL alla proprietà `Source` del `WebView`. Ciò attiverà l'evento `Navigating`, nel cui gestore viene avviato l'`ActivityIndicator (IsRunning)`, che verrà fermato quando la navigazione è completata (evento `Navigated`).

Per semplicità espositiva, in questo esempio non viene fatto uso del pattern Model-View-ViewModel, che è però raccomandato in scenari reali per separare la logica dall'interfaccia utente.

Nota: per aprire un sito Web è necessario impostare il permesso `INTERNET` nel manifest del progetto Android e `Internet Client` nel manifest del progetto UWP (file `Package.Appxmanifest`).

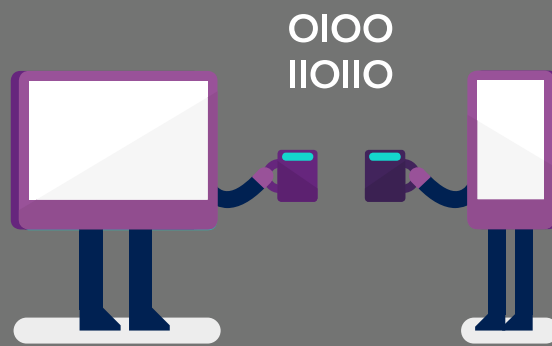


iOS: App Transport Security

A partire da iOS 9, Apple ha introdotto delle restrizioni note col nome di App Transport Security che di default limitano l'accesso a risorse via URL solo tramite protocollo https (TLS 1.2 o superiore). Ciò richiede che vengano effettuate alcune modifiche al file Info.plist. [E' possibile limitare l'accesso solo ad alcuni domini oppure consentire l'accesso anche a siti ritenuti non attendibili](#). Per far sì che il nostro esempio funzioni, in Solution Explorer fate click destro sul file Info.plist, poi Open With e XML Editor. Aggiungete la seguente eccezione:

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsArbitraryLoads</key>
  <true/>
</dict>
```

Fate attenzione al fatto che questa eccezione consente accesso arbitrario a qualunque URL ed è utilizzata in questo capitolo solo a scopo dimostrativo. Per maggiori informazioni sulla configurazione delle eccezioni all'App Transport Security, consultate la [documentazione](#).

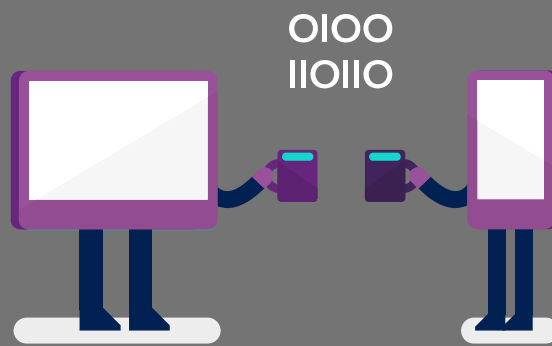


Adattare la UI a seconda del sistema

Ci possono essere situazioni in cui l'interfaccia utente dev'essere adattata al sistema su cui gira, pur mantenendone la condivisione. Per fare questo, Xamarin.Forms offre il metodo [Xamarin.Forms.Device.OnPlatform](#) e il corrispondente elemento XAML `OnPlatform`, che permette di specificare il valore di proprietà a seconda del sistema. Il seguente esempio dimostra come distanziare gli elementi in una pagina in modo diverso a seconda del sistema:

```
<ContentPage.Padding>
  <OnPlatform x:TypeArguments="Thickness" iOS="0, 20, 0, 0" Android="0, 10,
0, 0"
              WinPhone="0, 5, 0, 0"/>
</ContentPage.Padding>
```

Per iOS è sempre bene specificare un padding di 20 per le pagine, in quanto 20 è anche l'altezza di default della barra di stato di questo sistema operativo. [La classe Device espone anche un metodo che si chiama OnIdiom e che consente di sapere se l'app sta girando su uno smartphone, su un tablet o sul desktop \(quest'ultima solo per UWP\).](#)

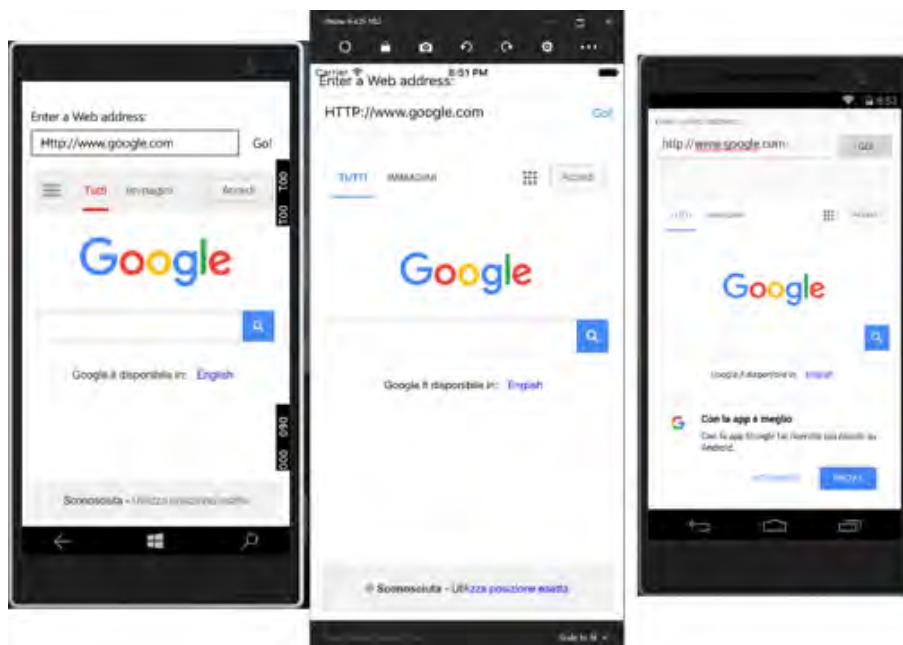


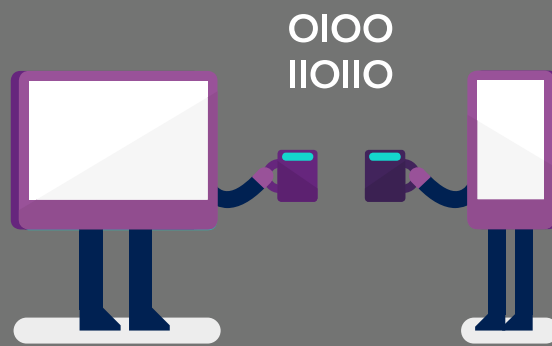
Avvio del debug dell'applicazione

Per avviare un'app scritta con Xamarin.Forms è necessario selezionare il relativo progetto di piattaforma in Solution Explorer come progetto di avvio. Fatto questo, a seconda del tipo di piattaforma dovreste selezionare il dispositivo su cui eseguire l'app, tra uno degli emulatori o un dispositivo fisico. Premendo F5, si avranno a disposizione tutte le funzionalità di debug dell'ambiente.

La **Figura 4-3** mostra l'app di esempio in esecuzione su iOS, Android e Windows 10 Mobile. Quindi, in estrema sintesi, con un'unica base di codice C# e tramite il riutilizzo dell'interfaccia grafica, l'app scritta in Xamarin.Forms è pronta per funzionare su i tre maggiori sistemi operativi.

Figura 4-3 L'app scritta in Xamarin.Forms su tutti i sistemi supportati





Utilizzo di funzionalità specifiche di piattaforma

All'inizio del capitolo si è detto che Xamarin.Forms consiste, tra l'altro, di un layer che espone tutto ciò che è comune ai vari sistemi operativi. Tuttavia, funzionalità come l'accesso ai sensori o quelle telefoniche sono costruite da API completamente diverse tra loro, così come su un sistema possono esserci elementi visuali non disponibili negli altri. Per consentire di avere accesso completo alle API, Xamarin.Forms offre tre possibilità descritte nei prossimi paragrafi.

Utilizzo di Custom Renderer

[Xamarin.Forms disegna gli elementi visuali sfruttando i controlli nativi di ciascun sistema](#), limitandosi però a quelle proprietà disponibili su tutti. Nel caso in cui avessimo bisogno di maggiori personalizzazioni, è possibile utilizzare i controlli nativi in modo diretto ed eseguendo le personalizzazioni di layout attraverso i cosiddetti custom render. [Un custom renderer è una classe che va implementata in ciascun progetto specifico di piattaforma e che consente di specificare proprietà tipiche di quella piattaforma](#). Per esempio, immaginate di voler fare in modo che il contenuto del controllo Entry sia automaticamente selezionato quando lo si tocca. Il layer di Xamarin.Forms non consente questo livello di gestione, per cui si crea dapprima una classe che eredita da Entry nel progetto PCL:

```
using Xamarin.Forms;

namespace CrossBrowser
{
    public class SelectableEntry: Entry
    {
    }
}
```

Nei progetti specifici, poi, si implementa una classe che eredita dal renderer del controllo nativo di interesse, ossia quell'oggetto che Xamarin.Forms chiama dietro le scene per il disegno dell'elemento. In iOS il custom renderer è una classe di questo tipo:

```
using CrossBrowser;
using CrossBrowser.iOS;
using System;
using UIKit;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(SelectableEntry), typeof(SelectableEntryRenderer))]
namespace CrossBrowser.iOS
{
    public class SelectableEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.
Entry> e)
        {
            base.OnElementChanged(e);
            var nativeTextField = Control;
            nativeTextField.EditingDidBegin += (object sender, EventArgs eIos) => {
                nativeTextField.PerformSelector(new ObjCRuntime.Selector("selectAll"),
null, 0.0f);
            };
        }
    }
}
```

Dove l'attributo `assembly` stabilisce che per l'oggetto `SelectableEntry` va usato il renderer `SelectableEntryRenderer`. Nel metodo `OnElementChanged` c'è la logica del comportamento dell'elemento, su cui in questo caso viene invocato il selettore tramite oggetti di Objective-C. In Android il concetto è simile, ma ovviamente il codice sarà quello di Xamarin.Android:

```
using Xamarin.Forms.Platform.Android;
using Xamarin.Forms;
using CrossBrowser;
using CrossBrowser.Droid;

[assembly: ExportRenderer(typeof>SelectableEntry), typeof>SelectableEntryRenderer)]
namespace CrossBrowser.Droid
{
    public class SelectableEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);
            if (e.OldElement == null)
            {
                var nativeEditText = (global::Android.Widget.EditText)Control;
                nativeEditText.SetSelectAllOnFocus(true);
            }
        }
    }
}
```

Infine la versione Universal Windows Platform:

```
using CrossBrowser;
using CrossBrowser.UWP;
using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ExportRenderer(typeof>SelectableEntry), typeof>SelectableEntryRenderer)]
namespace CrossBrowser.UWP
{
    public class SelectableEntryRenderer: EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            Control.SelectAll();
        }
    }
}
```

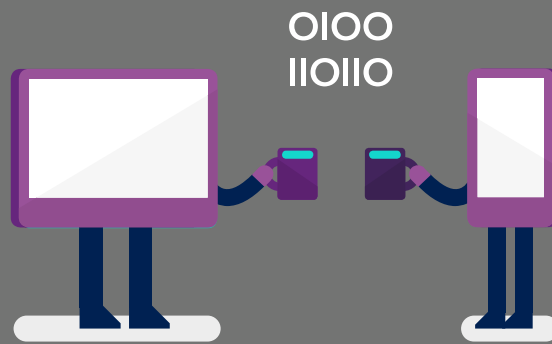
Nota: non è obbligatorio definire una classe che eredita dal controllo, per poi personalizzare il renderer. Un custom renderer può essere anche applicato al controllo offerto da Xamarin (come Entry), ma in quel caso il custom render verrà applicato indistintamente a tutti i controlli di quel tipo.

Nello XAML, poi, verrà utilizzato l'oggetto [SelectableEntry](#) definito nel progetto condiviso dapprima specificando il namespace in cui è definito l'oggetto stesso in questo modo:

```
<ContentPage xmlns:local="clr-namespace:CrossBrowser"
```

E poi utilizzandolo al posto dell'oggetto originario:

```
<local:SelectableEntry x:Name="WebEntry" HorizontalOptions="FillAndExpand" />
```

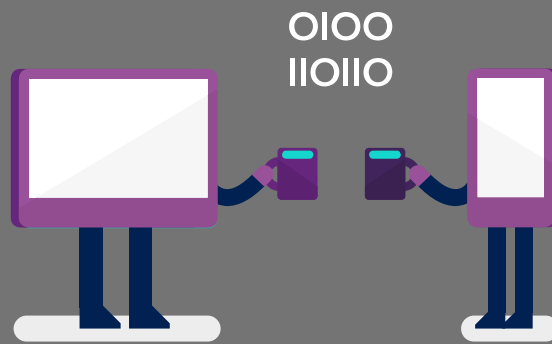


Utilizzo di Dependency Injection

Mentre i custom renderer riguardano gli elementi visuali, l'accesso a funzionalità hardware e software (sensori, fotocamera, file system) avviene attraverso il cosiddetto Dependency Service. In sintesi, nel progetto condiviso si definisce un'interfaccia che stabilisce i membri di una classe che accede a determinate funzionalità e poi nei progetti specifici si fornisce la relativa implementazione, invocata tramite la classe [Xamarin.Forms.Device](#) nel progetto condiviso. Si rimanda alla [documentazione](#) per gli approfondimenti.

Cenni sugli Effects

In alcuni casi si ha necessità di utilizzare controlli nativi, ma senza per forza dover intervenire anche sul loro comportamento, tipicamente quando si vuole solamente modificare alcuni stili estetici. In casi come questo, Xamarin.Forms mette a disposizione i cosiddetti Effects, oggetti derivati dalla classe [PlatformEffect](#), che consentono di ridefinire alcuni stili dei controlli e che possono poi essere consumati nello XAML aggiungendo ciascun Effect alla collection Effects che i controlli espongono. La [documentazione ufficiale](#) spiega in dettaglio come crearli e utilizzarli.



Utilizzo di Librerie NuGet e Plugin

Xamarin consente di scaricare e installare librerie da NuGet, il package manager di Visual Studio. Tra questi pacchetti rientrano i plugin per Xamarin, librerie che forniscono accesso a funzionalità specifiche di piattaforma attraverso delle API unificate fruibili dai progetti condivisi. Il loro nome comincia tipicamente con Xam.Plugin e possono essere cercati attraverso l'interfaccia o la Console di NuGet. Per esempio, il plugin Xam.Plugin. [Connectivity](#) (da installare a livello di solution) [consente di determinare la presenza di connettività senza ricorrere al dependency service](#) semplicemente utilizzando una riga come quella evidenziata in grassetto, che viene aggiunta al codice dell'app dimostrativa:

```
if(Plugin.Connectivity.CrossConnectivity.Current.IsConnected)
{
    if (!string.IsNullOrEmpty(this.WebEntry.Text) &&
        this.WebEntry.Text.ToLower().StartsWith("http://"))
```

Analogamente altri plugin consentono di accedere facilmente a funzionalità specifiche come l'accesso al file system, alla fotocamera, ecc. L'elenco dei plugin ufficiali è disponibile su [GitHub](#).




Riepilogo

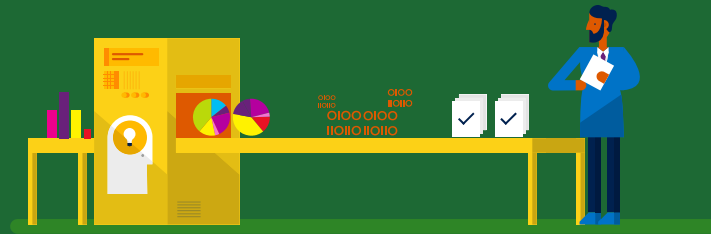
Attraverso un layer di unificazione, Xamarin.Forms consente di creare applicazioni che possano girare su Android, iOS e Windows sfruttando elementi visuali che hanno caratteristiche comuni a tutti i sistemi e massimizzando la condivisione del codice attraverso un progetto condiviso. L'interfaccia grafica è tipicamente definita con lo XAML e resa operativa tramite C#. Laddove si renda necessario accedere a elementi visuali o funzionalità specifiche dei sistemi, si può ricorrere ai custom renderer, al dependency service e ai plugin.

Cap. 5

La condivisione
e il riutilizzo del codice

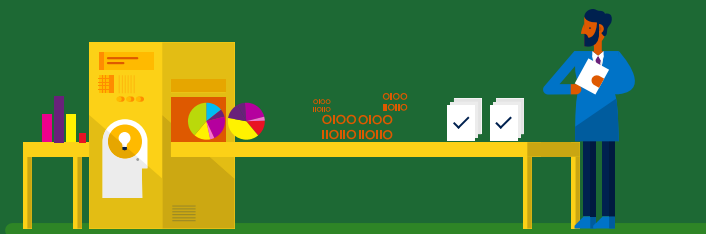


Uno degli aspetti più importanti in Xamarin è la condivisione e il riutilizzo del codice tra più progetti, aspetto che trova sicuramente la sua massima espressione in Xamarin.Forms ma che può riguardare anche Xamarin.iOS e Xamarin.Android. In questo capitolo viene fatta una panoramica delle tecniche di condivisione del codice, con uno sguardo anche alle novità di .NET Standard.



Come condividere e riutilizzare il codice

Le solution Xamarin sono costituite da progetti specifici di piattaforma, sia che si tratti di Xamarin.Forms che di Xamarin.iOS e Xamarin.Android. Ci sono dei casi in cui, e questo è vero in modo particolare per Xamarin.Forms, si può scrivere codice che esegue operazioni che non sono legate alle API dei vari sistemi e che quindi può essere riutilizzato e condiviso. Per fare questo, in Xamarin è possibile, ad oggi, utilizzare due tipologie di progetti: le Portable Class Libraries (PCL) e gli Shared Projects. Delle prime avete avuto un'introduzione nel capitolo precedente, ma in questo verranno portati alcuni importanti approfondimenti.



Le Portable Class Libraries

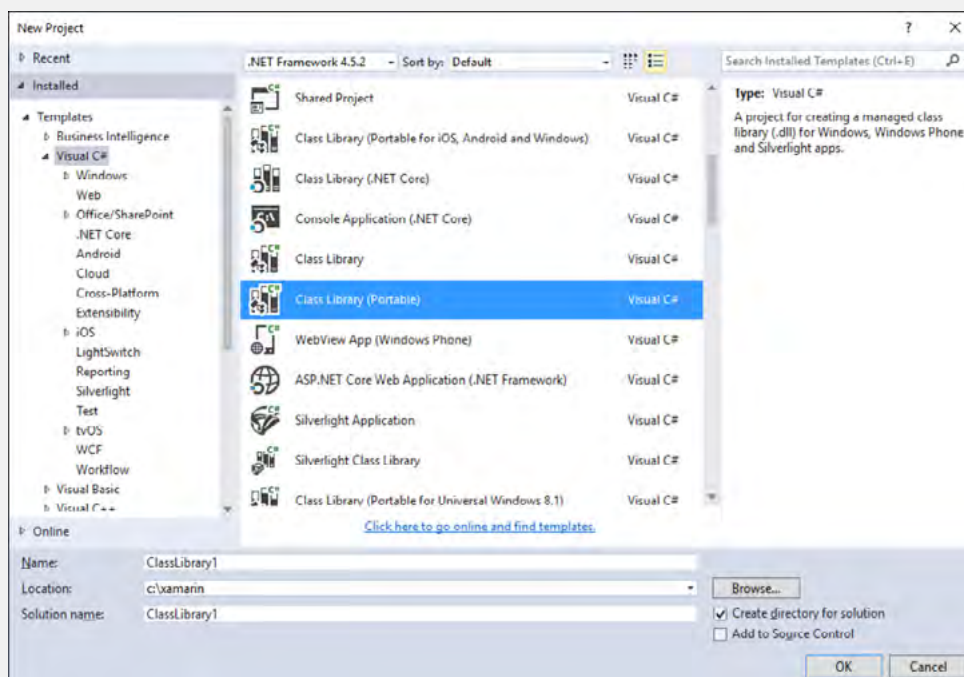
Come il nome lascia chiaramente intendere, le Portable Class Libraries (PCL) sono delle librerie portabili, che possono quindi essere consumate su più piattaforme a condizione che abbiano come target solo il sottoinsieme di API comune e supportato su tutte le piattaforme di destinazione. Non sono assolutamente esclusive di Xamarin e possono essere utilizzate in molti altri contesti applicativi, per esempio per condividere un'architettura basata su Model-View-ViewModel tra piattaforme come WPF e Universal Windows Platform. La prima importante considerazione da fare a questo punto è che le PCL producono un assembly compilato, che può essere distribuito e utilizzato in altri progetti. La seconda considerazione è che le PCL non possono contenere codice specifico di una certa piattaforma, per i seguenti motivi:

- perderebbero la loro caratteristica della portabilità;
- possono contenere solo codice che, con certezza, tutte le piattaforme interessate saranno in grado di eseguire.

Per fare un esempio, [una PCL che vuole essere utilizzata per condividere codice tra WPF e UWP non potrà mai contenere codice che gestisca il GPS di un dispositivo](#), poiché questo non è supportato da WPF e richiede API di Windows 10 a cui WPF non ha accesso. Limitandoci ad un contesto più orientato a Xamarin, oltre al progetto PCL che viene automaticamente incluso nelle solution 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. In questo modo viene favorita anche l'implementazione di pattern come Model-View-ViewModel, Factory, IoC e Dependency Injection (utilizzato col dependency service) e Service Locator. Per creare una PCL adatta a Xamarin in Visual Studio 2015, oltre all'aggiunta ad una solution esistente, si può utilizzare la finestra New Project e il template Class Library (Portable) visibile in **Figura 5-1**.

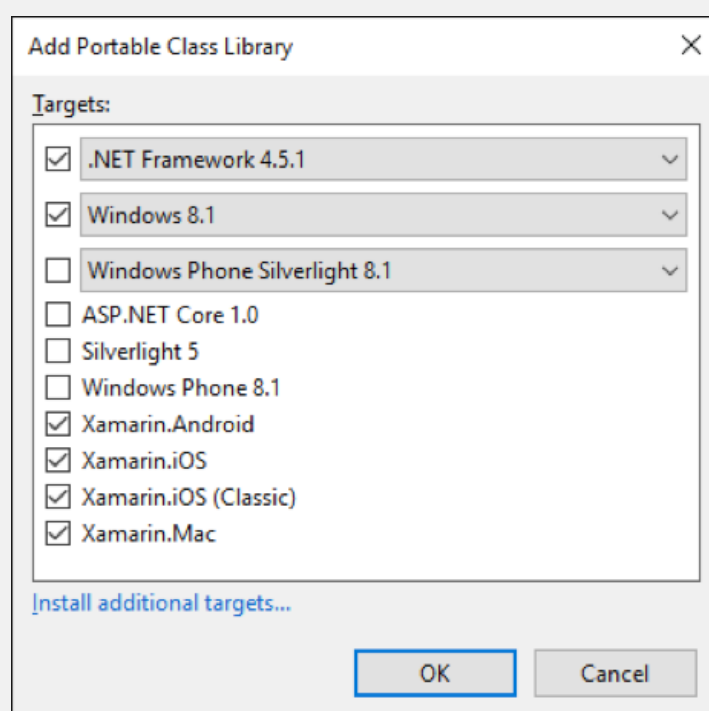
Nota: Per Xamarin.Forms, Visual Studio 2015 include un apposito template per creare una PCL che includa già i target supportati. E' disponibile tramite il template Class Library (Xamarin.Forms) nel nodo Cross-Platform.

Figura 5-1 Creazione di una Portable Class Library



Dopo aver cliccato OK, Visual Studio chiederà di specificare su quali sistemi e piattaforme la libreria dovrà funzionare. La scelta chiaramente dipende esclusivamente dalle vostre esigenze, tenete solo a mente il fatto che, **più piattaforme selezionate, minore sarà il sottoinsieme di API disponibile**. La **Figura 5-2** mostra quali sono i target da selezionare per ottenere una PCL utilizzabile con Xamarin.

Figura 5-2 Specifica dei target per una PCL



Il progetto generato conterrà un file di codice che potrà essere rinominato, eliminato o utilizzato. Il **Listato 5-1** mostra un'implementazione estremamente semplificata di MVVM con un model e un ViewModel che utilizza oggetti supportati da tutte le piattaforme, come i namespace System.Collections.ObjectModel, System.ComponentModel e System.Linq.

Listato 5-1

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;

namespace ClassLibrary1
{
    public class Customer: INotifyPropertyChanged
    {
        private string _companyName;
        private string _address;
        public string CompanyName
        {
            get { return _companyName; }
            set
            { _companyName = value; OnPropertyChanged(); }
        }
        public string Address
        {
            get { return _address; }
            set
            { _address = value; OnPropertyChanged(); }
        }

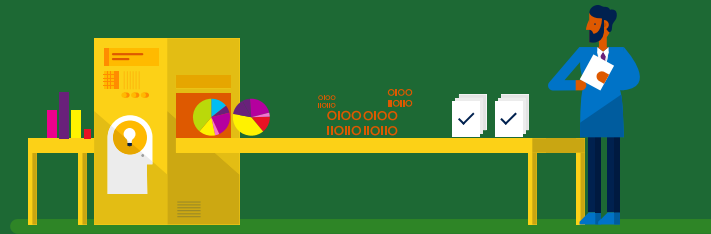
        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    public class CustomersViewModel
    {
        public ObservableCollection<Customer> Customers { get; set; }

        public IEnumerable<Customer> FilteredCustomers(string address)
        {
            var query = from cust in this.Customers
                        where cust.Address == address
                        select cust;
            return query;
        }

        public CustomersViewModel()
        {
            this.Customers = new ObservableCollection<Customer>();
        }
    }
}
```



Il codice di cui al **Listato 5-1** non richiede funzionalità specifiche di piattaforma (come accesso a sensori, file system e fotocamera), bensì utilizza oggetti sicuramente disponibili su tutti i target. Ultima considerazione, ma non per importanza, riguarda il fatto che [una PCL può non solo essere referenziata, ma a sua volta avere riferimenti su altre librerie e può ricevere pacchetti NuGet](#). Tra i vari elementi che possono contenere, le PCL supportano file .xaml per il riutilizzo di elementi dell'interfaccia grafica, immagini ed altri asset. La **documentazione** offre tutti i dettagli necessari.

Gli Shared Projects

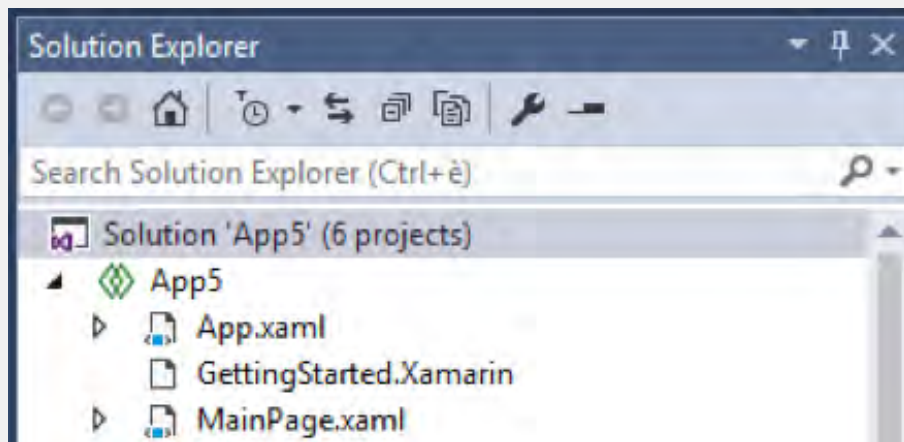
Gli Shared Projects, letteralmente progetti condivisi, hanno ugualmente lo scopo di [favorire e massimizzare il riutilizzo del codice](#) ma differiscono dalle PCL nei seguenti aspetti:

- **non sono autonomi**, nel senso che vivono esclusivamente nella solution a cui appartengono;
- **non producono una .dll compilata**, ma rappresentano più semplicemente un assortimento di file;
- **non possono avere riferimenti ad altre librerie** e non possono ricevere pacchetti NuGet;
- **possono fare uso di codice specifico di piattaforma**, poiché non sono limitati ad uno specifico target;
- con riferimento al punto precedente, [semplificano l'accesso a codice specifico di piattaforma attraverso semplici direttive di pre-compilazione come #if, #elif, #else](#) al contrario delle PCL in cui questo non è possibile e tale codice è demandato ai singoli progetti di piattaforma. Favoriscono inoltre l'uso di classi e metodi parziali e tecniche di class mirroring.

Come per le PCL, esistono dei template di progetto per Xamarin (Forms in particolare) in grado di utilizzare uno Shared Project per la parte di codice condivisa, inclusa l'interfaccia grafica.

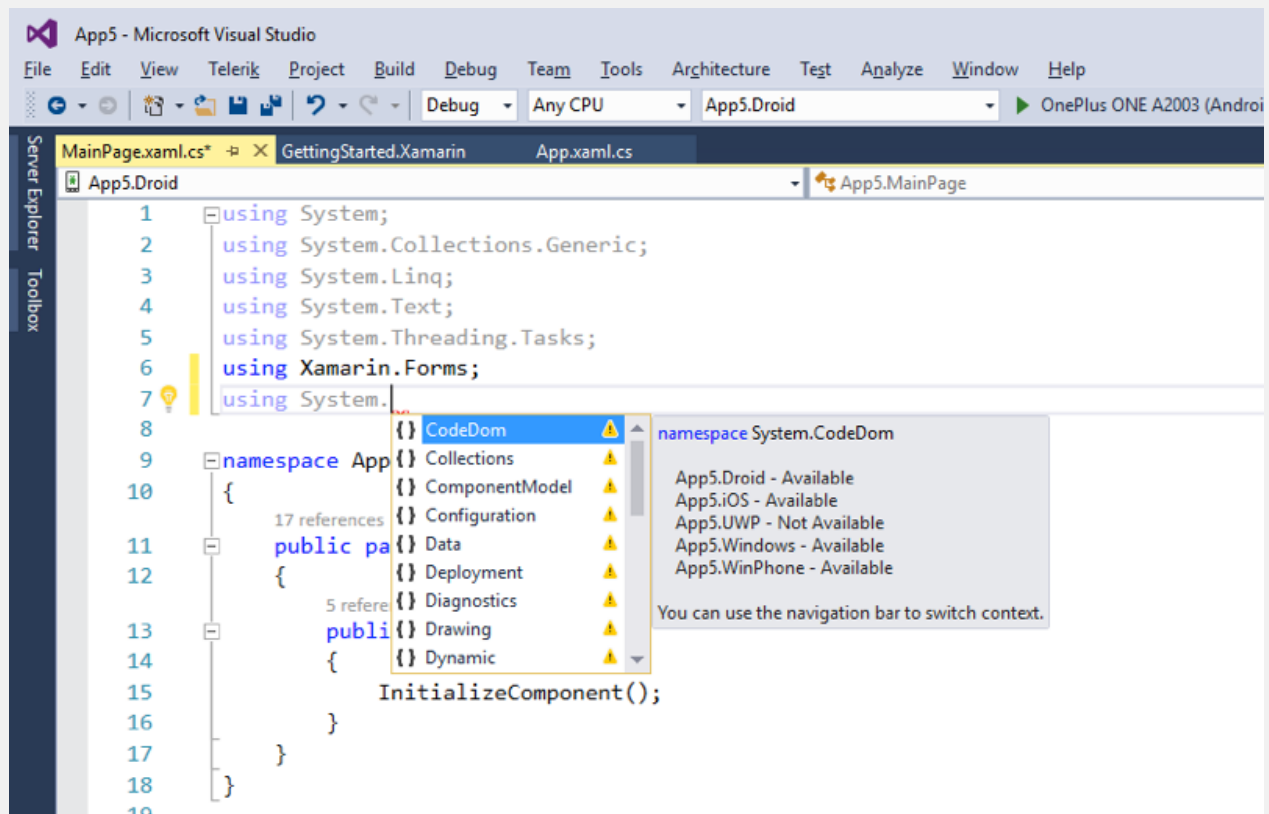
Per creare uno Shared Project si può utilizzare l'omonimo template disponibile nella finestra New Project sotto Visual C#, ma nel caso di Xamarin.Forms ci sono appositi template, tra cui quello chiamato Blank Xaml App (Xamarin.Forms Shared). Dopo la creazione della solution, in Solution Explorer vedrete il nuovo Shared Project come contenitore del codice condiviso, come certamente i file .xaml (vedi **Figura 5-3**).

Figura 5-3 Uno Shared Project per Xamarin.Forms



In questo progetto potrete aggiungere file di codice, file .xaml ma anche un numero pressoché infinito di file e risorse, dal momento che il progetto non viene compilato in modo a sè stante, ma il suo codice e i suoi oggetti vengono poi risolti dal compilatore solo quando viene fatta una build della soluzione. Poiché, come detto, uno Shared Project non può avere riferimenti su altre librerie (ma è chiaramente referenziato dai progetti di piattaforma), è normale non ricordare tutto quanto si ha a disposizione a livello di classi e codice. Fortunatamente, [l'IntelliSense viene in aiuto suggerendo, per esempio, la disponibilità dei namespace sulle varie piattaforme](#) (vedi **Figura 5-4**).

Figura 5-4 L'IntelliSense in uno Shared Project aiuta a capire gli oggetti disponibili

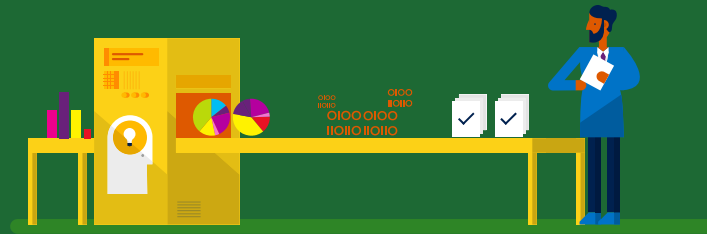


Il vantaggio degli Shared Project è che consentono, tramite direttive di pre-compilazione, di scrivere codice specifico di piattaforma senza doverlo demandare ai progetti ricorrendo al dependency service. Un esempio importante e noto è costituito dal file [SQLite.cs](#), che consente di interagire con database di SQLite, e di cui viene riportato il seguente estratto che mostra l'uso delle direttive `#if..#endif`:

```
#if __IOS__
    static SQLiteConnection ()
    {
        if (_preserveDuringLinkMagic) {
            var ti = new ColumnInfo ();
            ti.Name = "magic";
        }
    }

    /// <summary>
    /// Used to list some code that we want the MonoTouch linker
    /// to see, but that we never want to actually execute.
    /// </summary>
    static bool _preserveDuringLinkMagic;
#endif
//...
#if NETFX_CORE
    SQLite3.SetDirectory(/*temp directory type*/2, Windows.Storage.Appli-
cationData.Current.TemporaryFolder.Path);
#endif
```

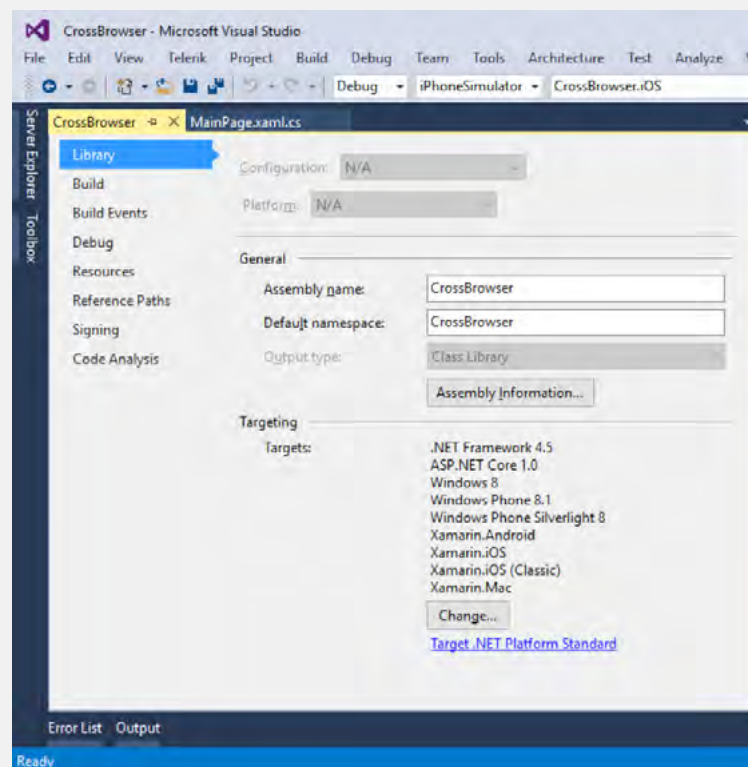
La [documentazione](#) fornisce tutti i dettagli per l'approfondimento.



Panoramica di .NET Standard

Insieme a .NET Core, Microsoft ha introdotto **.NET Standard**, un insieme di specifiche formali per librerie che espongono API unificate per tutte le piattaforme .NET. Sono la naturale evoluzione delle PCL e Microsoft sta lavorando su una roadmap che porterà presto .NET Standard alla versione 2.0. [Sebbene non siano ancora disponibili template di progetti che sfruttino le librerie .NET Standard di default, Xamarin supporta queste librerie.](#) Per attivare questo supporto, è necessario aprire le proprietà della PCL e cliccare su Target .NET Platform Standard, come rappresentato in **Figura 5-5**.

Figura 5-5 Il link per attivare .NET Standard



Prima di attivare .NET Standard in una PCL per Xamarin, occorre dapprima disinstallare il pacchetto Xamarin.Forms tramite NuGet, attivare .NET Standard e poi reinstallare il pacchetto. Per ulteriori informazioni, consultare questo [blog post ufficiale](#).




Riepilogo

Condividere e massimizzare il riutilizzo del codice sono cruciali in Xamarin. Tramite le PCL è possibile generare assembly riutilizzabili che sfruttano un sottoinsieme limitato e comune di API. Tramite gli Shared Project è possibile richiamare più facilmente codice specifico di piattaforma. Tramite .NET Standard è più semplice accedere ad API disponibili su tutte le piattaforme .NET.

Cap. 6

Servizi Azure
Mobile App e Xamarin

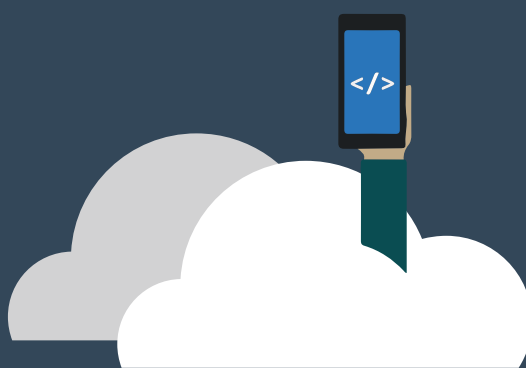


La maggior parte delle app per dispositivi mobili ha necessità di servizi di backend per memorizzare dati, per autenticare gli utenti e per inviare push notifications, oltre alle esigenze degli sviluppatori di monitorare l'utilizzo delle applicazioni stesse, per esempio per finalità di diagnostica o statistica. Le funzionalità sopra citate richiedono infrastrutture server a volte anche complesse, che richiedono tempo, risorse e costi per la loro realizzazione.

Grazie alla piattaforma Azure, la soluzione per il Cloud computing di Microsoft, tutti i necessari servizi di backend sono invece inclusi nell'abbonamento e pronti ad essere utilizzati, con pochissimi passaggi per la loro configurazione. Ciò consente di essere produttivi ed efficaci in brevissimo tempo e con sforzo estremamente contenuto, senza necessità di infrastrutture in casa.

L'insieme di servizi dedicati alle app prende il nome di Azure Mobile App ed in realtà non è limitato all'utilizzo con Xamarin, bensì è disponibile per varie altre piattaforme applicative. I servizi Azure Mobile App possono essere configurati e gestiti sia attraverso il **portale di gestione di Microsoft Azure**, sia attraverso il nuovo Visual Studio Mobile Center. Quest'ultimo semplifica ulteriormente i passaggi per la configurazione dei servizi e verrà preso in considerazione in questo capitolo per illustrare come creare un servizio di backend per memorizzare dei dati. Nel prossimo capitolo, invece, gli Azure Mobile App verranno utilizzati per implementare l'autenticazione degli utenti.

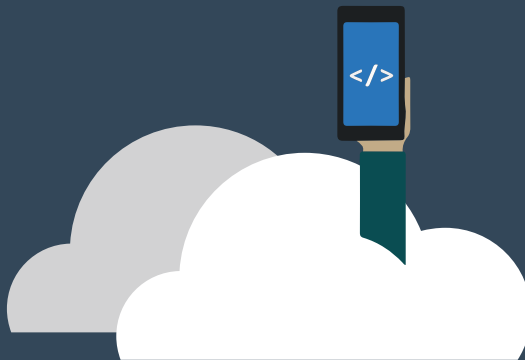
Nota: per poter completare i passaggi descritti in questo e nel prossimo capitolo, è necessario disporre di una sottoscrizione Microsoft Azure. E' possibile attivare una **trial gratuita** della durata di 30 giorni.



Creazione di un progetto di esempio

Per dimostrare come sfruttare compiutamente il servizio di backend di Azure Mobile App, verrà mostrata la creazione di un'app con Xamarin.Forms che consentirà l'inserimento e la visualizzazione di attività da eseguire. L'esempio verrà chiaramente semplificato al fine di focalizzare l'attenzione sui servizi disponibili.

Ciò premesso, create un nuovo progetto di tipo [Blank XAML App](#) (Xamarin.Forms Portable) e assegnate al progetto il nome TaskManager. Ad oggi solo le app per Android e iOS sono supportate dal Visual Studio Mobile Center, quindi i progetti per Universal Windows Platform, Windows Phone 8.1 e Windows 8.1 verranno ignorati. [Accertatevi che il pacchetto NuGet di Xamarin.Forms sia aggiornato all'ultima versione.](#)

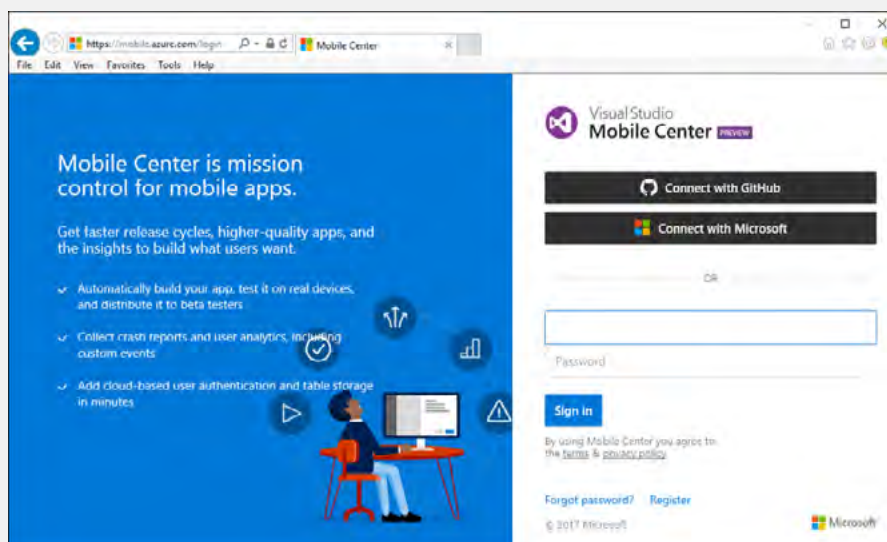


Associazione dell'app al Visual Studio Mobile Center

Prima di mettere mano al codice, è opportuno associare l'applicazione al Visual Studio Mobile Center. Questo servirà anche per ottenere informazioni su alcune librerie che verranno utilizzate. Nel vostro browser, quindi, aprite il **Visual Studio Mobile Center**. Vi verrà chiesto di accedere tramite un account Microsoft o GitHub (vedi **Figura 6-1**). Fate la vostra scelta ed inserite le credenziali. Una volta entrati, fate click su Add new app. Vi verrà richiesto di specificare il nome dell'app, una descrizione (opzionale), il sistema operativo (iOS o Android) e la piattaforma di sviluppo (Objective-C/Swift, Reactive Native, Xamarin).

Inserite Task Manager come nome dell'app, selezionate Android come sistema e Xamarin come piattaforma. In questo capitolo viene fornito un esempio basato su Android in quanto non necessariamente tutti i lettori hanno a disposizione un computer Mac, ma i passaggi sono identici anche per iOS. *E' anche importante sottolineare che dovrete ripetere l'operazione per ciascun sistema qualora vogliate associare i diversi pacchetti della vostra app multi-piattaforma.* Fate click su OK e vedrete la vostra app elencata nella pagina del Mobile Center.

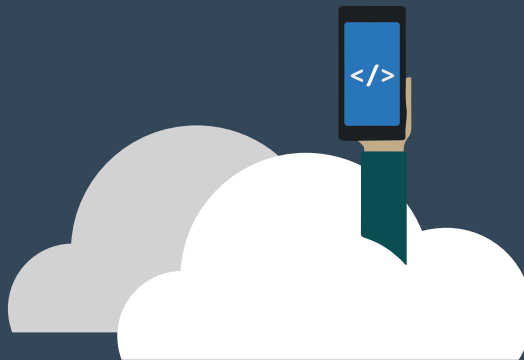
Figura 6-1 – L'accesso al Visual Studio Mobile Center



Una volta entrati, fate click su [Add new app](#). Vi verrà richiesto di specificare il nome dell'app, una descrizione (opzionale), il sistema operativo (iOS o Android) e la piattaforma di sviluppo (Objective-C/Swift, Reactive Native, Xamarin). Inserite Task Manager come nome dell'app, selezionate Android come sistema e Xamarin come piattaforma.

In questo capitolo viene fornito un esempio basato su Android in quanto non necessariamente tutti i lettori hanno a disposizione un computer Mac, ma i passaggi sono identici anche per iOS.

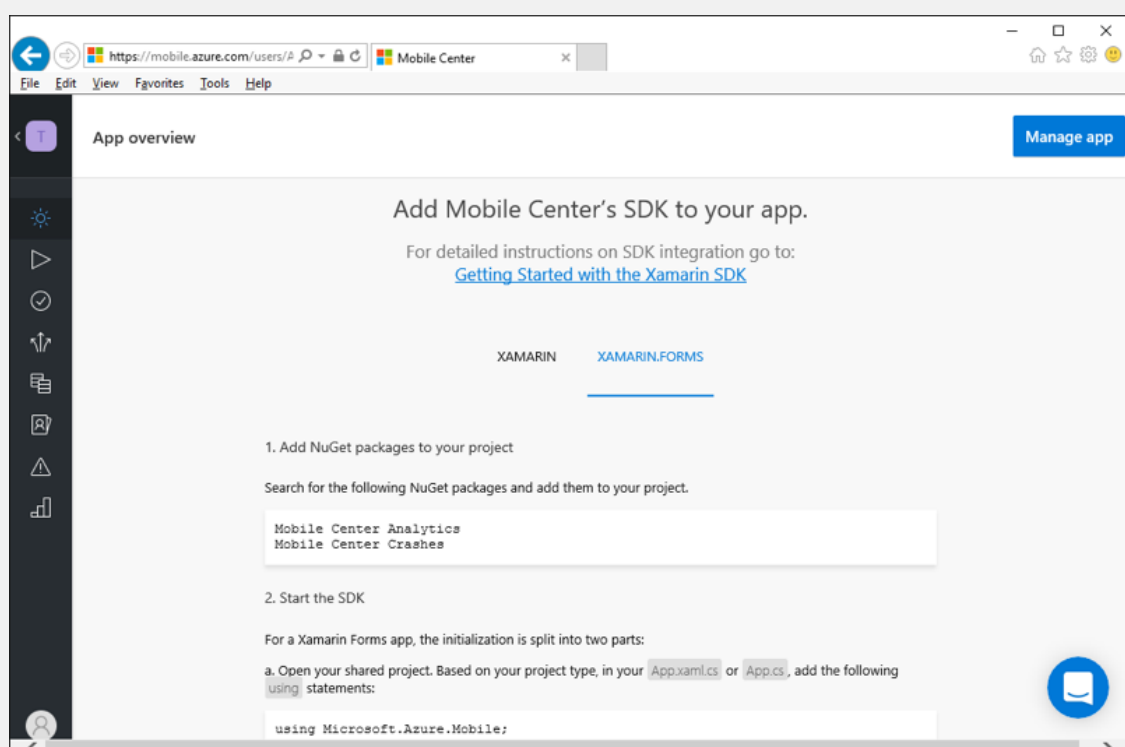
E' anche importante sottolineare che dovrete ripetere l'operazione per ciascun sistema qualora vogliate associare i diversi pacchetti della vostra app multi-piattaforma. Fate click su OK e vedrete la vostra app elencata nella pagina del Mobile Center.



Statistica e Diagnostica: il Mobile Center SDK

Se fate click sul nome dell'app, noterete come il Mobile Center fornirà un elenco di indicazioni per abilitare il Mobile Center SDK nella vostra applicazione, mentre sulla sinistra mostra un elenco di servizi e strumenti che vedremo man mano. [Il Mobile Center SDK consente di attivare strumenti di telemetria per analizzare informazioni](#) come il numero di utenti, le provenienze geografiche, la durata di utilizzo e i dispositivi usati, ma anche informazioni su crash ed eventi. Cliccate sulla scheda Xamarin.Forms (altrimenti Xamarin in caso di app Xamarin.Android), scorrete la pagina e seguite le indicazioni riportate per attivare l'SDK (vedi **Figura 6-2**).

Figura 6-2 Le indicazioni per attivare il Mobile Center SDK

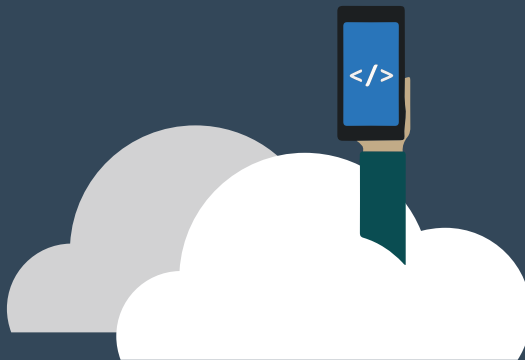


In sintesi è necessario:

- [scaricare e installare due pacchetti NuGet](#) chiamati Microsoft.Azure.Mobile.Analytics e Microsoft.Azure.Mobile.Crashes, che consentiranno di attivare la telemetria per statistica e crash rispettivamente;
- Dopo aver aggiunto le necessarie direttive using, [inserire la seguente riga di codice](#) nel costruttore della classe App dopo la chiamata a InitializeComponent: `MobileCenter.Start(typeof(A-nalytics), typeof(Crashes));`
Il metodo statico `Start` della classe `MobileCenter` consente, in questo modo, di attivare la telemetria per gli oggetti passati come parametri.
- Nel file `MainActivity.cs` del progetto Android, all'interno del metodo `OnCreate` e prima della chiamata a `LoadApplication`, inserire la seguente riga di codice che serve ad attivare la configurazione del Mobile Center:
`MobileCenter.Configure("YOUR-GUID");`

Dove YOUR-GUID rappresenta l'identificatore univoco assegnato all'app ed andrà sostituito con il GUID mostrato nella pagina del Mobile Center.

[Non serve altro, con due righe di codice viene abilitata la telemetria.](#)



Il servizio di backend: Table

Microsoft Azure offre, per il tramite del Mobile Center e degli Azure Mobile App, un comodissimo servizio di backend fruibile tramite tabelle, che le app possono consumare per memorizzare dati. Verrà quindi creata ora una tabella per memorizzare un elenco di attività. Sulla barra a sinistra, cliccate sull'icona con la dicitura Tables, quindi su Create Table. Vi verrà chiesto, a questo punto, di collegare la vostra sottoscrizione Microsoft Azure al Mobile Center. Si tratta semplicemente di selezionare una sottoscrizione attiva dall'apposito elenco. Fatto questo, specificate MyTask come nome della tabella. Sono disponibili le seguenti opzioni:

- **Soft delete:** quando attivata, i dati non verranno cancellati fisicamente dalla tabella ma solo marcati come eliminati ed esclusi dalle query di elenco;
- **Dynamic schema:** fa sì che le colonne vengano generate dinamicamente a seconda dell'oggetto che viene inserito. L'opzione è abilitata di default, ma non è consigliabile in produzione;
- **Per-user data:** questa opzione consente di limitare l'accesso ai dati solo a quelli creati dall'utente corrente. Poiché nel prossimo capitolo verrà implementata l'autenticazione, attivate questa opzione in questa fase.

Nel momento in cui la tabella viene creata, il Mobile Center definisce uno schema costituito dalle colonne e dai tipi rappresentato in **Tabella 6-1**.

Colonna	Tipo
id	String
updatedAt	Date
createdAt	Date
userId	String
deleted	Boolean
version	Version

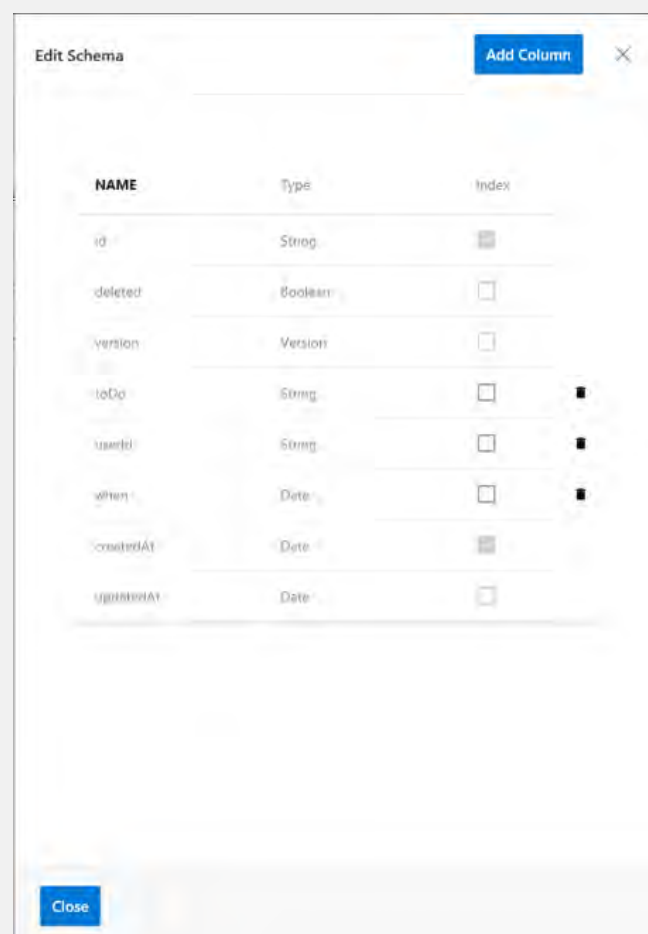
La struttura di base deve chiaramente essere estesa con le informazioni che servono all'app. Per aggiungere o modificare colonne, è sufficiente cliccare su Edit Schema, quindi Add Column e indicare nome e tipo. In questo caso aggiungete due colonne, una chiamata toDo, di tipo String, e una chiamata when di tipo Date. La prima servirà a memorizzare l'attività e la seconda la data in cui andrà eseguita.

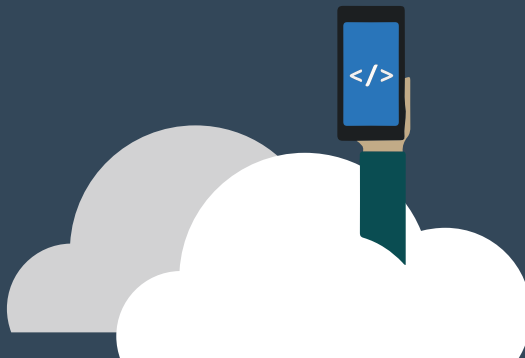
E' fondamentale che i nomi delle colonne inizino con la lettera minuscola per aderire alle specifiche JSON, come meglio descritto successivamente.

La **Figura 6-3** mostra come deve apparire lo schema della tabella.

A creazione avvenuta, il Mobile Center suggerisce un oggetto .NET chiamato IMobileServiceTable per poter interagire con la tabella da codice C#. Verrà ora descritto come utilizzarlo.

Figura 6-3 Gestire gli schemi delle tabelle





C#: memorizzare e gestire dati da codice

L'app necessita, a questo punto, di un modo per poter dialogare da codice C# con la tabella esposta dal servizio Cloud. Per farlo, Microsoft offre una libreria chiamata Microsoft.Azure.Mobile.Client che dovrà essere scaricata da NuGet.

Questa libreria è pensata per lavorare con database di SQLite, sia in Cloud che in locale e, tra l'altro, consente di implementare la sincronizzazione offline dei dati. Ciò significa che la tabella definita in precedenza è basata su SQLite e richiede che, nel codice, ci sia una classe che riproduce la tabella in forma .NET. Poiché le tabelle sono basate su SQLite, gli oggetti che devono lavorare col backend devono rispettare la notazione JSON, su cui la serializzazione SQLite si basa.

E' per questo che i nomi delle colonne iniziano con la lettera minuscola. Non solo, dal punto di vista di C# è necessario rappresentare alcuni tipi di dato in modo che siano comprensibili al serializzatore JSON. Ciò premesso, nel progetto PCL aggiungete una cartella chiamata Model e al suo interno una classe chiamata MyTask.cs. Il codice di questa classe espone tante proprietà quante sono le colonne della tabella (vedi **Listato 6-1**). I nomi delle proprietà seguono la notazione .NET, ma tramite l'attributo JsonProperty il serializzatore saprà a quale colonna della tabella corrispondono.

Listato 6-1

```
using System;
using Newtonsoft.Json;

namespace TaskManager.Model
{
    public class MyTask
    {
        [JsonProperty("id")]
        public string Id { get; set; }
        [JsonProperty("createdAt")]
        public DateTimeOffset CreatedAt { get; set; }
        [JsonProperty("updatedAt")]
        public DateTimeOffset UpdatedAt { get; set; }
        [JsonProperty("version")]
        public byte[] Version { get; set; }
        [JsonProperty("deleted")]
        public bool Deleted { get; set; }
        [JsonProperty("todo")]
        public string Todo { get; set; }
        [JsonProperty("when")]
        public DateTime When { get; set; }

        public MyTask()
        {
            this.Id = Guid.NewGuid().ToString("N");
            this.CreatedAt = DateTimeOffset.Now;
            this.UpdatedAt = DateTimeOffset.Now;
            this.When = DateTime.Now;
        }
    }
}
```

Notate anche come il tipo `Version` sia stato rimappato in `byte[]` e come il `Date` per `createdAt` e `updatedAt` sia stato rimappato in `DateTimeOffset`.

Questo per garantire la compatibilità col formato JSON.

Il passaggio successivo consiste nell'istanziare una classe che si chiama `MobileServiceClient`, e che mette in comunicazione l'app col servizio di backend. Nel file `App.xaml.cs`, aggiungete una proprietà statica di tipo `MobileServiceClient` e modificate il costruttore come indicato nel **Listato 6-2**.

Listato 6-2

```
using Microsoft.Azure.Mobile;
using Microsoft.Azure.Mobile.Analytics;
using Microsoft.Azure.Mobile.Crashes;
using Xamarin.Forms;
using Microsoft.WindowsAzure.MobileServices;

namespace TaskManager
{
    public partial class App : Application
    {
        public static MobileServiceClient AzureClient { get; internal set; }

        public App()
        {
            InitializeComponent();
            MobileCenter.Start(typeof(Analytics), typeof(Crashes));
            AzureClient = new MobileServiceClient("https://your-id.azurewebsites.net/");

            MainPage = new TaskManager.MainPage();
        }

        // ...
    }
}
```

Il costruttore della classe richiede che venga passato l'URL del backend, visibile nel Mobile Center se cliccate sul nome della tabella.

Il passaggio successivo consiste nel definire un cosiddetto `ViewModel`, un oggetto intermedio che si interpone tra interfaccia grafica e dati e che ha il compito di caricare, salvare ed esporre i dati. Nel progetto PCL, aggiungete una cartella chiamata `ViewModel` e un file al suo interno chiamato `MyTaskViewModel.cs`. Poi inserite il codice riportato nel **Listato 6-3**, a cui seguiranno commenti.

Listato 6-3

```
using System.Threading.Tasks;
using TaskManager.Model;
using System.Collections.ObjectModel;
using Microsoft.WindowsAzure.MobileServices;

namespace TaskManager.ViewModel
{
    public class MyTaskViewModel
    {
        public MyTask CurrentTask { get; set; }
        public ObservableCollection<MyTask> MyTasks { get; set; }

        public async Task InitAsync()
        {
            if (this.CurrentTask == null) this.CurrentTask = new MyTask();

            this.MyTasks = await GetTasksAsync();
        }

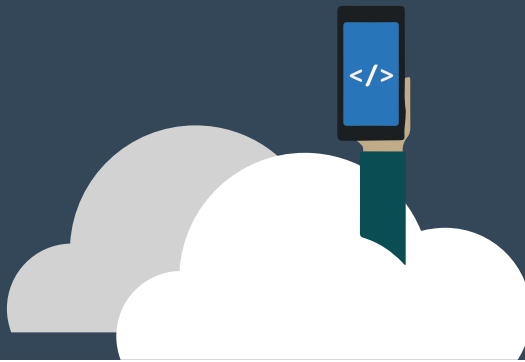
        public async Task SaveTaskAsync(MyTask newTask)
        {
            this.MyTasks.Add(newTask);

            IMobileServiceTable<MyTask> table = App.AzureClient.GetTable<MyTask>();
            await table.InsertAsync(newTask);
        }

        public async Task<ObservableCollection<MyTask>> GetTasksAsync()
        {
            IMobileServiceTable<MyTask> table = App.AzureClient.GetTable<MyTask>();
            var listOfTasks = await table.ToListAsync();

            return new ObservableCollection<MyTask>(listOfTasks);
        }
    }
}
```

La classe [MobileServiceClient](#) espone tutti i metodi che servono per interagire coi dati, tra cui [GetTable](#) per ottenere un riferimento alla tabella e che restituisce un oggetto [IMobileServiceTable<T>](#). Il risultato ottenuto può essere interrogato, per esempio tramite il metodo extension [Where](#), e fatto confluire in una collection con [ToListAsync](#). [MobileServiceClient](#) espone, poi, metodi come [InsertAsync](#) per inserire un oggetto, [UpdateAsync](#) per aggiornarlo e [DeleteAsync](#) per rimuoverlo. Il [ViewModel](#) in questione espone una [ObservableCollection<MyTask>](#), un tipo di collection adatto per il data-binding, e un'istanza di [MyTask](#) che rappresenti un'attività singola. L'inizializzazione avviene nel metodo [InitAsync](#) piuttosto che nel costruttore, in modo che possa essere invocato un altro metodo asincrono.



Predisposizione dell'interfaccia grafica

L'interfaccia grafica dell'app dimostrativa è costituita da markup XAML che consente l'inserimento di un'attività tramite casella di testo e selezione di una data tramite un controllo chiamato `DatePicker`. Inoltre, consente la visualizzazione delle attività esistenti tramite una `ListView`. Viene fatto uso della tecnica del data-binding, per la quale si rimanda alla [documentazione](#) per gli approfondimenti. Il **Listato 6-4** mostra come definire l'interfaccia all'interno del file `MainPage.xaml`.

Listato 6-4

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

    <StackLayout Padding="5">
        <StackLayout BindingContext="{Binding CurrentTask}">
            <ActivityIndicator x:Name="Indicator1" IsRunning="False" IsVisible="False"/>
            <Label Text="Insert task: "/>
            <Entry x:Name="TaskEntry" Text="{Binding todo, Mode=TwoWay}" />

            <DatePicker x:Name="TaskDate" Date="{Binding when, Mode=TwoWay}" />

            <Button x:Name="AddButton" Text="Add task" Clicked="AddButton_Clicked"/>
        </StackLayout>

        <ListView x:Name="TaskListView" ItemsSource="{Binding MyTasks}">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding todo}" Detail="{Binding when}" />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

Nel code-behind, MainPage.xaml.cs, viene inizializzato il ViewModel in un metodo chiamato OnAppearing, che il runtime invoca dopo il costruttore e appena prima che la pagina sia visualizzata. Inoltre viene gestito il click del pulsante, che invoca il metodo di salvataggio dei dati definito nel ViewModel. Il tutto è mostrato nel **Listato 6-5**.

Listato 6-5

```
using System;
using TaskManager.ViewModel;
using Xamarin.Forms;

namespace TaskManager
{
    public partial class MainPage : ContentPage
    {
        private MyTaskViewModel ViewModel { get; set; }

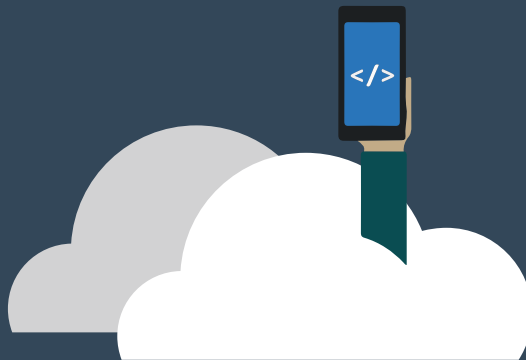
        public MainPage()
        {
            InitializeComponent();
            this.ViewModel = new MyTaskViewModel();
        }

        private void ManageIndicator()
        {
            this.Indicator1.IsVisible = !this.Indicator1.IsVisible;
            this.Indicator1.IsRunning = !this.Indicator1.IsRunning;
        }

        protected async override void OnAppearing()
        {
            base.OnAppearing();

            ManageIndicator();
            await this.ViewModel.InitAsync();
            this.BindingContext = this.ViewModel;
            ManageIndicator();
        }

        private async void AddButton_Clicked(object sender, EventArgs e)
        {
            ManageIndicator();
            await this.ViewModel.SaveTaskAsync(this.ViewModel.CurrentTask);
            this.BindingContext = this.ViewModel;
            ManageIndicator();
        }
    }
}
```

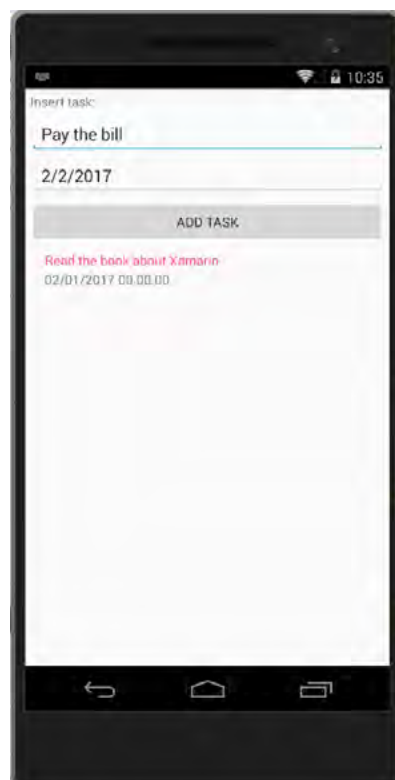


Test dell'applicazione

Indipendentemente dal tipo di device, fisico o emulatore, [l'applicazione di esempio è in grado di accedere al backend su Azure e di leggere e scrivere dati](#). La **Figura 6-4** mostra l'app in esecuzione nell'emulatore di Visual Studio per Android.

Con pochissimi passaggi, quindi, è stato possibile creare un backend per l'applicazione senza ricorrere ad infrastrutture e server proprietari. Il Mobile Center consente anche di visualizzare l'elenco dei dati nelle varie tabelle, semplicemente cliccando sul nome di ciascuna tabella.

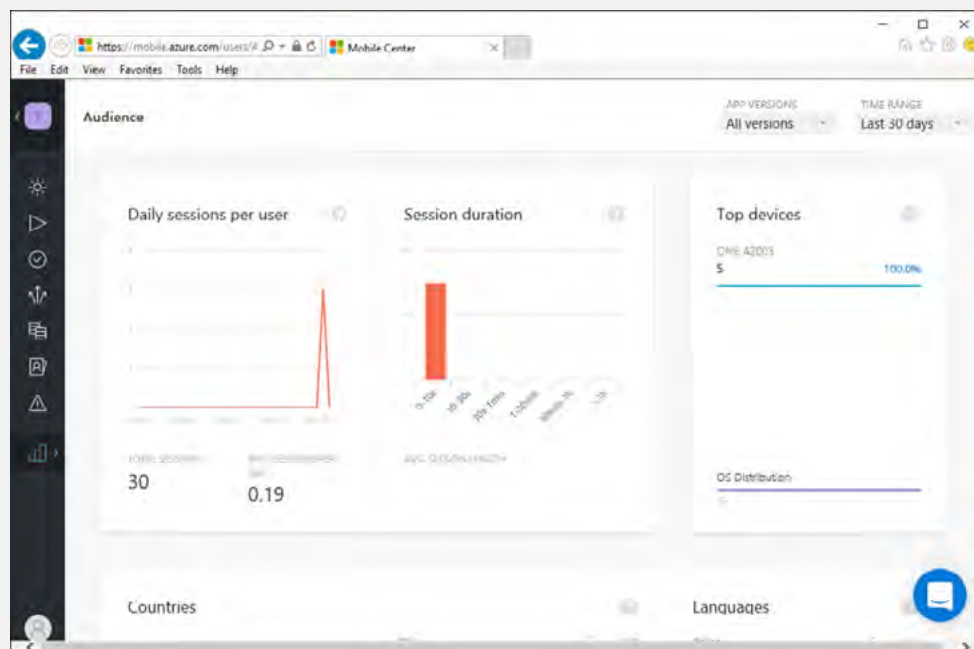
Figura 6-4 L'applicazione interagisce con il backend e le tabelle



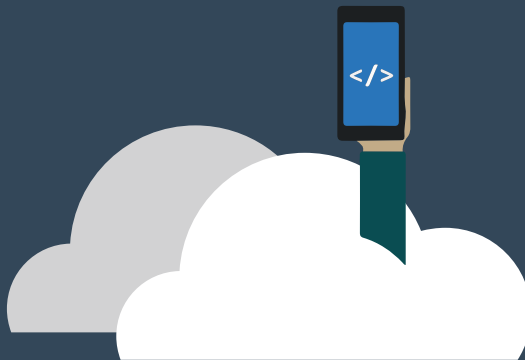
Telemetria nel Mobile Center

Grazie al Mobile Center SDK, è possibile visualizzare informazioni diagnostiche e di statistica sull'uso delle applicazioni. Nel Mobile Center, cliccate dapprima sul pulsante Analytics nella barra a sinistra. E' possibile visualizzare informazioni sull'utenza delle applicazioni, sugli eventi e sul flusso di log. La **Figura 6-5** mostra un esempio.

Figura 6-5 Analisi delle statistiche in telemetria



Tra le numerose informazioni che è possibile visualizzare rientrano gli utenti attivi, quante volte viene usata l'app al giorno da ciascun utente, quanto dura ogni sessione, i dispositivi utilizzati, i Paesi e le lingue di provenienza. Cliccando su Crashes, invece, sarà possibile visualizzare informazioni sui crash che si sono verificati e capire meglio le cause. E' bene ricordare che tutto questo è stato ottenuto con due semplici righe di codice necessarie per l'attivazione del servizio.



Cenni ai vari servizi del Mobile Center

Il Visual Studio Mobile Center offre una serie di servizi non solo legati allo sviluppo ma all'intero ciclo di vita del software. Il servizio denominato Build consente di connettere un repository Git contenente il codice dell'applicazione per automatizzare le build. Il servizio Test consente di definire e lanciare test automatici per l'interfaccia grafica simulando circa 400 dispositivi diversi. Il servizio Distribute consente di distribuire i pacchetti dell'applicazione a gruppi selezionati di persone per finalità di test. In realtà, Microsoft Azure offre anche altri servizi per le applicazioni mobili, tutti raggiungibili dal **portale di gestione**.



Riepilogo

Attraverso la piattaforma Microsoft Azure e il nuovo Visual Studio Mobile Center, definire un servizio di backend per le proprie app è estremamente rapido ed efficiente. Con pochi passaggi è possibile configurare il Mobile SDK per attivare informazioni diagnostiche e statistiche ed è possibile creare tabelle per la memorizzazione di dati che, lato C#, vengono gestiti attraverso la libreria Microsoft.Azure.Mobile.Client. Tra i vari servizi offerti, rientra anche quello dell'autenticazione delle app, discusso nel prossimo capitolo.

Cap. 7

Autenticazione
tramite i servizi Azure
Mobile App

Uno dei requisiti più comuni all'interno di applicazioni per dispositivi mobili è quello di consentire l'accesso a determinate risorse solo agli utenti autenticati e autorizzati.

I servizi di Azure Mobile App, per il tramite del Visual Studio Mobile Center, offrono un modello di programmazione unificato per implementare l'autenticazione tramite diversi provider come [Facebook](#), [Twitter](#), [Microsoft Account](#), [Google+](#) e [Azure Active Directory](#). Certamente è possibile implementare meccanismi di autenticazione personalizzati basati, per esempio, su Web API ma l'obiettivo di questo capitolo è fornire un'introduzione al sistema basato su Azure evidenziando la sua semplicità. Verrà ripresa l'applicazione di esempio creata nel capitolo precedente, che verrà integrata con autenticazione basata su Microsoft Account.

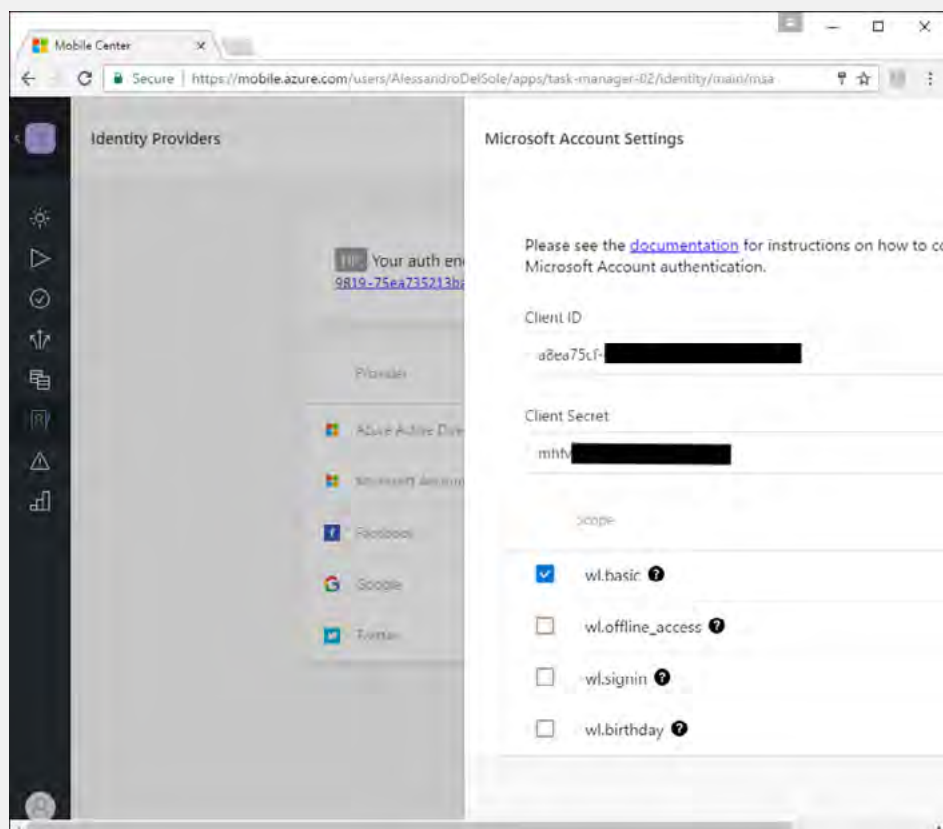
Questa scelta è legata al fatto che non è possibile prevedere se il lettore dispone di account sui social network, mentre certamente ne ha uno a disposizione legato alla sottoscrizione Azure utilizzata. [Vedrete, comunque, che la scelta del provider di autenticazione non influisce sulla scrittura del codice.](#)



Scegliere un provider di autenticazione

Il primo passaggio è quello di scegliere e impostare uno o più provider di autenticazione tramite il [Visual Studio Mobile Center](#). Per farlo, cliccate su Identities. Comparirà l'elenco dei provider disponibili, quindi scegliete Microsoft Account. Come vedete in **Figura 7-1**, sarà necessario specificare alcune informazioni come l'id applicazione, la chiave segreta e le risorse legate all'account Microsoft che l'applicazione potrà leggere (i primi due sono parzialmente oscurati per ragioni di privacy).

Figura 7-1 Abilitare l'autenticazione tramite Microsoft Account

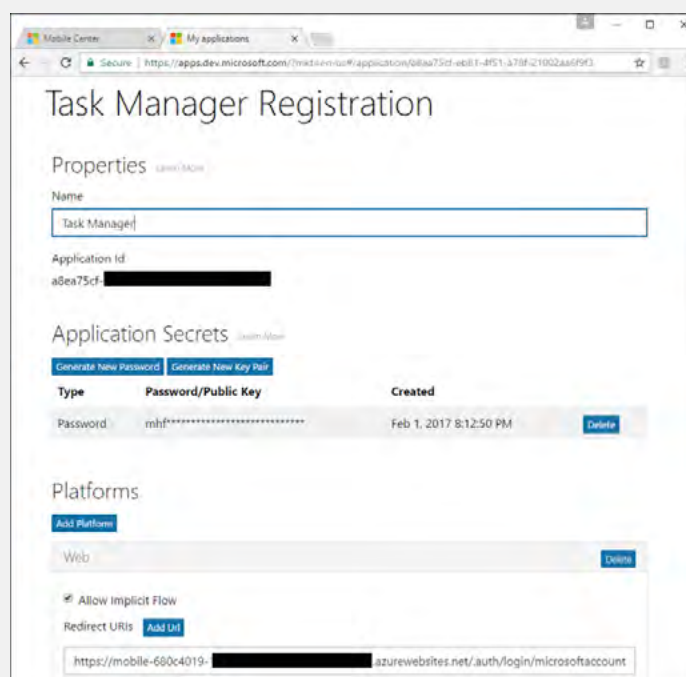




Configurazione del provider e dell'app lato servizio

L'id applicazione e la chiave segreta vengono ottenuti registrando l'applicazione in un'apposito portale, a cui si accede anche tramite il link alla documentazione mostrato nella pagina, chiamato **My Applications**. Questo portale contiene l'elenco delle app che sviluppate e che sono legate al vostro account Microsoft. Quando pronti, cliccate su **Add new app** sotto Converged Applications. Inserite il nome della vostra app, confermate e vedrete poi una pagina in cui sarà presente l'id applicazione. Cliccate poi su **Generate New Password** per ottenere la chiave segreta, di cui dovrete prendere nota perché sarà l'unica volta in cui sarà visibile. Fate poi click su **Add Platform** e inserite l'URL di ritorno che occorre al servizio per sapere cosa fare una volta autenticato l'utente. L'URL è composto nella forma <https://{id-app}.azurewebsites.net/.auth/login/microsoftaccount>, dove {id-app} va sostituito con l'id dell'applicazione contenuto nell'URL che avete utilizzato nel file App.xaml.cs. La **Figura 7-2** mostra un riepilogo.

Figura 7-2 Configurazione dell'app nel portale sviluppatori



Ci sono altre possibili opzioni ma non sono richieste, quindi fate click su [Save](#). Tornate nel Mobile Center e inserite l'id applicazione e la chiave segreta, avendo cura di selezionare anche il permesso wl.basic (vedi **Figura 7-1**) che serve al servizio per poter autenticare l'utente. Per maggiori informazioni sui vari permessi potete passare col puntatore del mouse sull'icona del punto interrogativo.



Impostazione delle autorizzazioni

Il passaggio successivo consiste nel limitare l'accesso a determinate risorse solamente agli utenti autenticati. Tornate quindi nel Mobile Center ed eseguite i seguenti passaggi:

1. [cliccate su Tables](#) ed aprite la tabella MyTask;
2. [cliccate poi sul pulsante identificato dai tre punti sospensivi](#), vicino a Edit Schema;
3. [cliccate su Change Permissions](#). Da qui potrete stabilire se l'esecuzione delle operazioni sui dati deve essere consentita solo ad utenti autenticati.

Per praticità espositiva, nell'esempio viene stabilito che tutte le operazioni sono consentite solamente ad utenti autenticati. Quando pronti, fate click su Save.

Impostazioni di memorizzazione del token

Come detto all'inizio del capitolo precedente, il Visual Studio Mobile Center è attualmente in preview e alcune funzionalità non sono state ancora implementate. Una di queste riguarda l'opzione di memorizzazione del token che dev'essere abilitata affinché l'autenticazione funzioni. E' possibile, però, farlo dal **portale di gestione di Azure**. Nel portale, cliccate sull'id del vostro servizio Azure Mobile App che comparirà nella dashboard. Cliccate poi su Authentication/Authorization, all'interno del gruppo Settings. In basso, sotto Advanced Settings, impostate su On la voce Token Store, quindi salvate le modifiche.



Autenticare gli utenti nell'applicazione

Autenticare gli utenti e consentire loro l'accesso all'app è molto semplice, poiché si tratta semplicemente di invocare il metodo [MobileServiceClient.LoginAsync](#) a cui va passato il contesto grafico in cui viene eseguita l'interfaccia di autenticazione e il provider scelto. Il contesto grafico, però, varia a seconda del sistema operativo su cui gira l'app, per cui la chiamata va implementata per ciascun progetto di piattaforma. Con l'occasione verrà mostrato anche un utilizzo del dependency service di Xamarin.Forms. Ovviamente, nel caso di progetti Xamarin.iOS e Xamarin.Android la chiamata al metodo sarà già specifica e legata al sistema.



Codice condiviso e codice specifico di piattaforma

Nel progetto PCL, create una cartella chiamata Authentication. All'interno aggiungete un'interfaccia chiamata `IAuthentication` come la seguente:

```
using Microsoft.WindowsAzure.MobileServices;
using System.Threading.Tasks;

namespace TaskManager.Authentication
{
    public interface IAuthentication
    {
        Task<MobileServiceUser> LoginAsync(MobileServiceClient client,
                                           MobileServiceAuthProvider provi-
der);
    }
}
```

L'interfaccia stabilisce come dovrà essere implementata la chiamata di login, poi ciascun progetto di piattaforma la implementerà a seconda delle proprie caratteristiche. [Questo metodo personalizzato riceve l'istanza della classe `MobileServiceClient` e il provider di autenticazione](#). Nel progetto Android, aggiungete una classe chiamata `Authentication.cs` che implementa l'interfaccia di cui sopra, il cui contenuto è quello indicato nel **Listato 7-1**.

Listato 7-1

```
using System;
using TaskManager.Authentication;
using Microsoft.WindowsAzure.MobileServices;
using Xamarin.Forms;
using System.Threading.Tasks;
using TaskManager.Droid;

[assembly: Dependency(typeof(Authentication))]
namespace TaskManager.Droid
{
    public class Authentication : IAuthentication
    {
        public async Task<MobileServiceUser> LoginAsync(MobileServiceClient client,
                                                         MobileServiceAuthenticationProvider provider)
        {
            try
            {
                var user = await client.LoginAsync(Forms.Context, provider);
                return user;
            }
            catch (Exception e)
            {
                System.Diagnostics.Debug.WriteLine(e.Message);
            }
            return null;
        }
    }
}
```

Come vedete, viene chiamato il metodo `LoginAsync` sull'istanza di `MobileServiceClient`, a cui viene anche detto che il contesto visuale è corrispondente a quello di `Xamarin.Forms` (`Forms.Context`). **Notate l'attributo `assembly` con `Dependency`, che indica al runtime quale implementazione di `IAuthentication` da utilizzare.** E' possibile specificarne più di una, ma solo una viene utilizzata secondo la logica *last wins*.



Estendere il ViewModel per supportare l'autenticazione

A questo punto occorre fare alcune piccole modifiche al ViewModel, il cui codice completo è contenuto nel **Listato 7-2** a cui seguono i necessari commenti.

Listato 7-2

```
using System.Threading.Tasks;
using TaskManager.Model;
using System.Collections.ObjectModel;
using Xamarin.Forms;
using TaskManager.Authentication;
using Microsoft.WindowsAzure.MobileServices;
using System.Runtime.CompilerServices;
using System.ComponentModel;

namespace TaskManager.ViewModel
{
    public class MyTaskViewModel: INotifyPropertyChanged
    {

        public MyTaskViewModel()
        {
            this.IsUserAuthenticated = false;
        }

        private bool isUserAuthenticated;

        public bool IsUserAuthenticated
        {
            get
            {
                return isUserAuthenticated;
            }

            set
            {
                isUserAuthenticated = value; OnPropertyChanged();
            }
        }
    }
}
```



```

private string userId;

public string UserId
{
    get
    {
        return userId;
    }

    set
    {
        userId = value; OnPropertyChanged();
    }
}

private MyTask currentTask;
public MyTask CurrentTask
{
    get
    {
        return currentTask;
    }

    set
    {
        currentTask = value; OnPropertyChanged();
    }
}

private ObservableCollection<MyTask> myTasks;
public ObservableCollection<MyTask> MyTasks
{
    get
    {
        return myTasks;
    }

    set
    {
        myTasks = value; OnPropertyChanged();
    }
}

public async Task InitAsync()
{
    await LoginAsync();

    if(this.IsUserAuthenticated==true)
    {
        if (this.CurrentTask == null) this.CurrentTask = new MyTask();
        this.MyTasks = await GetTasksAsync();
    }
}

public async Task SaveTaskAsync(MyTask newTask)
{
    if(this.IsUserAuthenticated==true)
    {
        if (string.IsNullOrEmpty(newTask.UserId)) newTask.UserId = this.UserId;
        this.MyTasks.Add(newTask);
        var table = App.AzureClient.GetTable<MyTask>();
        await table.InsertAsync(newTask);
    }
}

```

```

    }
}

public async Task<ObservableCollection<MyTask>> GetTasksAsync()
{
    if (this.IsUserAuthenticated == true)
    {
        var table = App.AzureClient.GetTable<MyTask>();
        var listOfTasks = await table.Where(t => t.UserId == this.UserId).ToListAsync();

        return new ObservableCollection<MyTask>(listOfTasks);
    }
    else
        return null;
}

public async Task LoginAsync()
{
    var authenticator = DependencyService.Get<IAuthentication>();
    var user = await authenticator.LoginAsync(App.AzureClient,
        MobileServiceAuthenticationProvider.MicrosoftAccount);

    if (user == null)
    {
        this.IsUserAuthenticated = false;
        return;
    }
    else
    {
        this.UserId = user.UserId;
        this.IsUserAuthenticated = true;
        return;
    }
}

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

Innanzitutto [viene introdotto un metodo chiamato LoginAsync che, tramite il metodo DependencyService.Get](#), chiama l'implementazione specifica di piattaforma della classe Authentication e su di essa invoca il metodo di autenticazione. Notate che è sufficiente cambiare il valore dell'enumerazione MobileServiceAuthenticationProvider da MicrosoftAccount a uno degli altri per implementare un provider diverso (previa, chiaramente, attivazione del provider lato servizio). Notate come, nel metodo GetTasksAsync, i dati vengano ristretti solamente all'utente corrente mediante metodo Where. [Vengono poi introdotte due nuove proprietà UserId e IsUserAuthenticated](#), la prima in particolare è utile per ricevere il nome dell'utente e che sarà utilizzato nell'interfaccia grafica. Infine, viene implementata l'interfaccia INotifyPropertyChanged che consente di notificare all'interfaccia la presenza di modifiche nei valori delle proprietà.



Modificare l'interfaccia grafica

E' opportuno estendere l'interfaccia grafica con un pulsante di Login e un'etichetta che mostri il nome dell'utente autenticato. Nel file MainPage.xaml, aggiungete le due seguenti righe tra l'ActivityIndicator e la Label esistente:

```
<Label x:Name="UserLabel" Text="{Binding UserId}" />
<Button x:Name="LoginButton" Text="Login" Clicked="LoginButton_Clicked" />
```

Quindi, impostate `IsEnabled="False"` per la Entry e il DatePicker. Nel code-behind, aggiungete questo gestore Click per il pulsante, il cui compito è quello di chiamare il Login dal ViewModel e, se tutto va a buon fine, di abilitare i controlli e collegare i dati:

```
private async void LoginButton_Clicked(object sender, EventArgs e)
{
    await this.ViewModel.LoginAsync();
    if(this.ViewModel.IsUserAuthenticated==true)
    {
        await this.ViewModel.InitAsync();
        this.TaskEntry.IsEnabled = true;
        this.TaskDate.IsEnabled = true;
        this.BindingContext = this.ViewModel;
    }
}
```

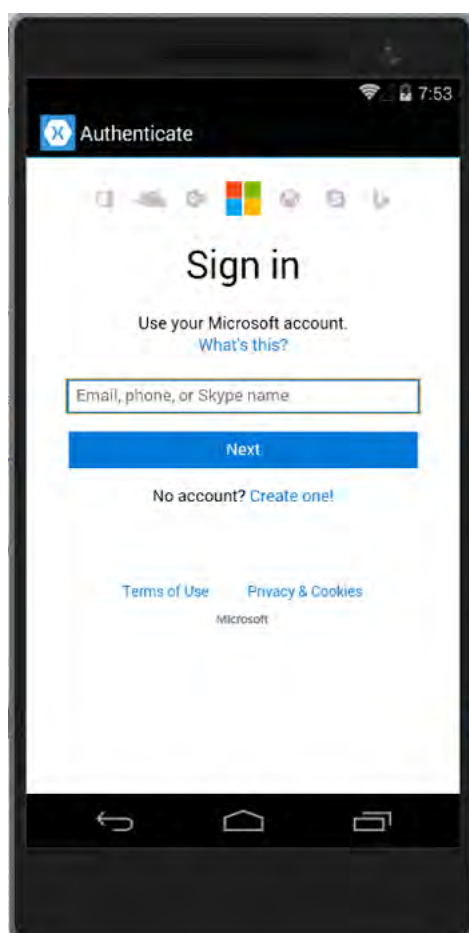
Con codice reale, è opportuno gestire anche l'eccezione `MobileServiceClientException` che si può verificare sia per problemi di login che per tentativi di accesso non autorizzato alle risorse.



Test dell'applicazione

A questo punto è possibile testare l'applicazione e la nuova funzionalità di login. Toccando l'apposito pulsante, comparirà l'interfaccia grafica di login (vedi **Figura 7-3**). [Se la procedura va a buon fine, i controlli dell'interfaccia verranno abilitati per la gestione dei dati.](#)

Figura 7-3 L'interfaccia grafica di autenticazione





Riepilogo

Grazie a Microsoft Azure e ai servizi Mobile App, implementare l'autenticazione basata sui più diffusi provider è estremamente semplice. E' infatti necessario ottenere le informazioni identificative dell'app presso ciascun provider e associarle al Mobile Center. I semplici membri esposti dalla classe `MobileServiceClient` consentono poi di effettuare l'autenticazione con essenzialmente una sola riga di codice.

Biografia

Alessandro Del Sole lavora come Senior .NET Developer e si occupa di sviluppo di applicazioni per dispositivi mobili utilizzando Xamarin e Visual Studio. E' Microsoft Certified Professional e, dal 2008, è Microsoft MVP nella categoria Visual Studio & Development Technologies ed è stato premiato come MVP Of The Year per 5 volte. Autore di numerosi libri ed ebook per sviluppatori, nonché di decine di articoli tecnici per importanti portali tra cui MSDN Magazine e InformIT, è riconosciuto a livello internazionale come esperto di Visual Studio e .NET. Partecipa regolarmente come speaker alle principali conferenze ed eventi italiani ed è Community Leader di Visual Studio Tips & Tricks.

A cura di Alessandro Del Sole
con il supporto tecnico
di Dan Ardelean e **Mahiz Srl**

