Università Politecnica delle Marche

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Progettazione e implementazione di un'app Xamarin e di un sistema di back-end per la gestione delle ordinazioni in una pizzeria

Design and implementation of a Xamarin app and a back-end system for the management of the orders in a pizzeria

Relatore

Prof. Domenico Ursino

Candidato

Giacomo Vitali

Anno Accademico 2018-2019

Indice

Elenco delle figure

Elenco dei listati

Introduzione

Nell'ultimo decennio, in Italia e nel resto del mondo, si è assistito ad una diffusione dei dispositivi mobili come strumenti che permettono il supporto e la semplificazione di molteplici attività. Lo smartphone ha, anche, sostituito diverse attività presenti in precedenza, come l'invio dei messaggi al posto delle lettere o la gestione degli appuntamenti al posto di un'agenda cartacea. Per questo motivo, negli ultimi anni, sono state realizzate sempre più applicazioni che hanno rivoluzionato il modo con cui vengono condotte molte attività; si pensi, ad esempio, alle applicazioni di *home banking* che hanno rivoluzionato il nostro rapporto con la banca.

Uno dei campi in cui l'utilizzo di questi dispositivi può portare diversi vantaggi è quello della ristorazione. I cellulari, infatti, vengono ampiamente utilizzati durante il processo di ordinazione, in modo da poter evitare errori derivanti dalla scrittura di un ordine da parte di un cameriere. In più, l'ordine può essere facilmente inviato alla cucina o alla cassa e ai restanti cellulari utilizzati dal personale. Con una spesa iniziale, si possono garantire nel tempo diversi vantaggi a livello economico.

Il passo successivo di questi dispositivi consiste nel farli utilizzare direttamente alla clientela. Si può, infatti, notare in che modo diverse catene, come, ad esempio, *McDonald's*, tramite alcune postazioni fisse, permettano agli utenti di ordinare autonomamente senza recarsi alle casse. Questo ha fatto sì che il personale, avendo una diminuzione del carico di lavoro, e dovendosi occupare soltanto di far pagare i clienti nel caso in cui abbiano scelto il pagamento tramite contanti, ha a disposizione più tempo per fornire un servizio migliore. In più, le lunghe file che erano presenti alle casse sono molto diminuite.

Ulteriori vantaggi riguardano la possibilità di scegliere la pietanza in tutta tranquillità, potendo, in più, vedere, rispetto a diversi menù cartacei, immagini che rappresentano i piatti, divisi in sezioni, con i corrispondenti ingredienti. Il cliente può, anche, decidere, in molti casi, come personalizzare il proprio piatto, aggiungendo o rimuovendo ingredienti e segnalando alla cucina particolari intolleranze o allergie. In qualsiasi momento è possibile vedere il totale dell'ordine e il prezzo del singolo piatto.

Vista la crescita dell'utilizzo di questi dispositivi, anche piccole attività di ristorazione hanno iniziato a comprendere l'utilità e i benefici di questi sistemi innovativi.

Questa tesi è dedicata allo sviluppo e alla realizzazione di un'applicazione mobile che permetta al cliente di ordinare autonomamente al ristorante, senza la necessità di un cameriere. Lo scopo è quello di fornire un'alternativa valida e meno dispendiosa rispetto al classico sistema di ordinazione. In questo modo, lo stesso cliente risparmierà tempo durante l'ordinazione, mentre il proprietario avrà un risparmio sul personale.

Si è, quindi, deciso di sviluppare un'applicazione da installare sui dispositivi presenti sui diversi tavoli del ristorante. Questa scelta viene fatta perchè, diversamente dalle postazioni fisse accennate in precedenza, che non si addicono all'ambiente di un ristorante, il dispositivo mobile permette di raggiungere un buon equilibrio tra benefici e costi, in quanto il prezzo di una singola unità si aggira intorno al centinaio di euro e ha in più la funzionalità di intrattenere la clientela tramite alcuni giochi che possono essere installati al suo interno.

La scelta di usare dei dispositivi mobili permette di inserire, in ciascuno di essi, il numero del tavolo sul quale si trovano, permettendo, anche, la sua modifica durante la serata. In questo modo l'applicazione risulta flessibile e si adatta facilmente alle esigenze del ristorante.

La spesa iniziale derivante dall'acquisto dei telefoni cellulari sarà recuperabile grazie alla diminuzione del personale e alla possibilità di non dover più stampare menù cartacei. Infatti, il menù sarà presente all'interno dell'applicazione e potrà essere modificato in ogni momento. La modifica eseguita verrà visualizzata in modo istantaneo all'interno dei dispositivi.

Si è, anche, deciso di rendere l'applicazione quanto più semplice e comprensibile possibile, in modo da essere adatta a qualsiasi tipo di utenza. È stato, anche, previsto l'inserimento di un pulsante per richiedere aiuto direttamente al personale, nel caso in cui ci fossero problemi o incomprensioni durante l'utilizzo.

Per permettere una fidelizzazione dei clienti, in modo da non perderli e ottenerne di nuovi, si è pensato di inserire una sezione dove l'utente può registrarsi e ottenere, durante i vari ordini, alcuni punti, spendibili per avere degli sconti personalizzati sugli ordini successivi. In questo modo, il cliente sarà incentivato a tornare nel ristorante successivamente, per ottenere altri sconti a lui dedicati.

Riassumendo, le principali funzionalità dell'applicazione riguardano la possibilità di visualizzare il menù, le sezioni e i vari piatti all'interno dell'applicazione, la possibilità di effettuare ordini da inviare direttamente alla cucina e, infine, la possibilità di registrarsi per ottenere futuri sconti su ordini successivi.

La presente tesi, dopo una spiegazione generale sulle app ibride e sul framework *Xamarin*, intende presentare la realizzazione e l'implementazione dell'applicazione descritta in precedenza, soffermandosi sull'analisi dei requisiti, sulla progettazione della base di dati e sull'implementazione delle funzionalità più rilevanti.

La tesi è strutturata nei seguenti capitoli:

- Nel primo capitolo, *Le app ibride*, verranno introdotti i vari tipi di applicazioni, soffermandosi sui vantaggi e sugli svantaggi delle applicazioni ibride. Infine, verranno descritti alcuni tra i framework con cui realizzarle, ovvero *Ionic*, *Cordova*, *React Native*, *Flutter* e *Xamarin*.
- Nel secondo capitolo, *Xamarin*, tratteremo i vari componenti del framework Xamarin, da noi adottato per la realizzazione dell'app, soffermandoci sulla sua

architettura, sui principali *IDE* e su come il codice può essere condiviso tra i vari progetti.

- Nel terzo capitolo, *Analisi dei requisiti*, presenteremo dapprima una descrizione, in linguaggio naturale, delle funzionalità desiderate per l'applicazione, per poi elencare i requisiti funzionali e non funzionali, nonchè i casi d'uso che formalizzano i requisiti funzionali.
- Nel quarto capitolo, *Progettazione*, illustreremo, tramite una mappa, tutte le aree dell'applicazione. Successivamente, descriveremo i vari *Mockup* e i *Diagrammi delle attività* realizzati. Infine, presenteremo la base di dati realizzata nel dettaglio attraverso la progettazione concettuale e logica.
- Nel quinto capitolo, *Implementazione*, tratteremo le varie funzionalità dell'applicazione, per poi illustrare il codice delle sue componenti principali.
- Nel sesto capitolo, *Discussione in merito al lavoro svolto*, analizzeremo il progetto sviluppato, considerando i suoi punti di forza e di debolezza, e confrontandolo con alcuni sistemi simili che si trovano in commercio.
- Nel settimo capitolo, *Conclusioni*, ripercorreremo ciò che viene trattato all'interno della tesi e parleremo di alcuni possibili sviluppi futuri dell'applicazione realizzata.

1

Le app ibride

In questo capitolo introdurremo le app ibride. Prima di tutto illustreremo la differenza tra i vari tipi di applicazioni; in seguito, vedremo i principali framework usati per la loro realizzazione.

1.1 La principale divisione tra le applicazioni

Quando si parla di applicazione mobile si intende un'applicazione software dedicata ai dispositivi di tipo mobile, normalmente realizzata seguendo le linee guida del dispositivo sul quale è installata. Dividendo le app (abbreviazione di applicazione) per sistema operativo e per utilizzo su più piattaforme possiamo classificarle in 3 principali categorie: *App Native, Web App e App Ibride* (Figura 1.1).

Figura 1.1. I diversi tipi di applicazione

1.1.1 App native

Le app native (Figura 1.2) sono quelle applicazioni mobili che vengono sviluppate interamente nel linguaggio del dispositivo per il quale vengono progettate. Ad esempio, nel caso del sistema operativo iOS, un'applicazione nativa sarà scritta in linguaggio *Swift*, mentre, per Android, sarà scritta in *Java* o *Kotlin*.

Queste sono le app che si interfacciano con il sistema operativo nel modo più completo. Infatti, essendo sviluppate per la singola piattaforma, possono interagire completamente con il dispositivo, avendo, ad esempio, facile accesso alla telecamera, alla geolocalizzazione e ad altre funzioni dello smartphone.

I diversi vantaggi che ci sono utilizzando questo tipo di applicazioni sono i seguenti:

- *Utilizzo totale delle capacità del dispositivo*: come abbiamo già detto, infatti, essendo sviluppate ad hoc per un determinato sistema operativo, queste app possono usare tutte le funzionalità del device in modo molto semplice.
- *Ottimizzazione dell'Esperienza Utente*: le app native riescono a rispondere prontamente, e in modo molto intuitivo, a tutte le richieste dell'utente.
- *Distribuzione negli store di appartenenza*: rispetto a qualsiasi altro tipo di app, le app native hanno maggior risalto all'interno dello Store messo a disposizione dal sistema operativo.

Figura 1.2. App native

Ci sono, però, anche dei difetti, legati allo sviluppo di queste app:

- *Devono essere scritte per ogni piattaforma*: bisognerà, quindi, conoscere tutti i linguaggi specifici, uno per ogni sistema operativo per cui si vuole implementare.
- *Single platform affinity*: essendo progettate soltanto per il singolo sistema operativo, queste app funzionano in modo efficiente soltanto sulla singola piattaforma. Pertanto, se si vuole cambiare ambiente di lavoro sarà necessario effettuare un'implementazione ex-novo.

1.1.2 Web app

Una web app (Figura 1.3) è un'applicazione che opera su di un server web e che viene utilizzata attraverso un browser web, a differenza di un'applicazione nativa. Lo scopo delle web app è quello di rendere il contenuto disponibile sui dispositivi mobili. Per il loro uso, però, è necessario avere una connessione Internet. Ciò rende possibile non installare questo tipo di applicazioni, avendo un vantaggio sul risparmio della memoria del dispositivo. I dati degli utenti non verranno salvati nel dispositivo, ma su di un server web, il che potrebbe rallentare l'applicazione.

Le applicazioni web presentano i seguenti vantaggi:

- *Non bisogna installare l'applicazione sullo smartphone* e, quindi, la capacità di memoria del dispositivo non viene appesantita.
- *Sono semplici da scrivere*, in quanto si possono sviluppare usando linguaggi web ormai molto noti, come *HTML*, *CSS* e *JavaScript*.
- *Possono essere supportate da qualsiasi dispositivo* che abbia a disposizione un browser, quindi possono essere considerate multipiattaforma.
- *I tempi di sviluppo sono minori*, in quanto non devono essere sviluppate in versioni differenti per essere eseguite su diversi dispositivi.

Essendo però visualizzate all'interno di un browser hanno dei difetti:

- *Non possono funzionare senza una connessione ad Internet*: nel caso non ci sia connessione, sarà impossibile accedere a questo tipo di app.
- *Sono le più lente* e non consentono performance all'altezza di altri tipi di app.
- *Il processo di fidelizzazione è molto più complicato*: infatti, non essendo presenti sugli store potrebbe esser molto più difficile avere utenti fissi che le utilizzano.
- *L'invio di notifiche push non è possibile*, non essendo tali app installate sul dispositivo sul quale vengono eseguite.

Figura 1.3. Web app

1.1.3 App ibride

Le app ibride uniscono i vantaggi delle web app e delle app native, riducendo i costi e permettendo di implementare l'applicazione un'unica volta rendendola, poi, in grado di operare su più piattaforme contemporaneamente. Questo tipo di applicazione è composto da un componente chiamato *WebView*, che permette di usare tecnologie web per sviluppare l'applicazione e, al contempo, di considerarla come un'app nativa, in modo tale da poter avere i pregi di entrambi i modelli.

I benefici più importanti che si hanno nello sviluppare applicazioni ibride sono i seguenti:

- *Sono più rapide e meno care da sviluppare rispetto alle app native*; infatti non servirà più conoscere i linguaggi di ogni singola piattaforma per creare applicazioni *cross-platform*.
- *Offrono più possibilità di una web app*. Come già detto prima, infatti, possono essere inserite all'interno degli store, aumentando la distribuzione e la fidelizzazione tra gli utenti.
- *La manutenzione è più facile*. Essendo multipiattaforma, basterà un'unica modifica per rendere effettivi i cambiamenti su tutti i dispositivi nei quali vengono installate.
- *Viene generata una sola versione*, indipendentemente dal numero di piattaforme sulle quali verranno eseguite.

Sono presenti, però, anche degli svantaggi:

- *Le performance sono inferiori*. Rispetto alle app native, viene sfruttato maggiormente l'hardware. Questo potrebbe peggiorare la velocità dell'applicazione, ad esempio sui tempi di reazione ad una determinata azione da parte dell'utente che la sta utilizzando.
- *Risultano meno stabili* dal momento che vengono implementate sul singolo sistema operativo, a differenza delle app native.
- *L'implementazione dell'interfaccia utente* è difficile.

1.1.4 Quali scegliere? Pro e contro

Dopo aver visto tutti i pro e i contro di questi diversi tipi di applicazione, possiamo dire che non esiste una scelta migliore.

Tutto dipenderà dagli obiettivi che si vogliono realizzare e dai mezzi e dalle esigenze che si hanno nel momento in cui si decide di creare un'applicazione.

Se, ad esempio, si vuole sviluppare un gioco mobile, occorrerà utilizzare uno sviluppo nativo, in modo da rendere l'app il più veloce ed efficiente possibile. Se, invece, si vuole creare un'applicazione semplice ed accessibile sarà preferibile utilizzare una web app. Se, infine, si vuole adattare un sito web facendolo diventare un'app o si vuole sviluppare velocemente un'applicazione in grado di funzionare su qualsiasi sistema operativo si sceglierà uno sviluppo ibrido.

1.2 Come realizzare le app ibride: i principali framework

Un framework è un'architettura logica di supporto su cui un software può essere progettato e realizzato, facilitando lo sviluppo da parte del programmatore. Questo è definito da un insieme di classi astratte e dalle relazioni tra di esse. Alla sua base, spesso, troviamo sempre librerie utilizzabili con uno o più linguaggi di programmazione, spesso insieme ad un IDE, un debugger o altri strumenti per velocizzarne e facilitarne lo sviluppo.

Di seguito vedremo i principali framework in grado di sviluppare applicazioni ibride. Sul mercato infatti esistono diversi tipi di framework, ciascuno con caratteristiche e vantaggi differenti. Ad esempio, alcuni framework potrebbero non supportare tutti i sistemi operativi. Per questo motivo un programmatore sceglierà un framework in base alla propria esperienza personale.

1.2.1 Ionic

Ionic è un framework utilizzato per lo sviluppo di applicazioni mobile tramite tecnologie Web. Si pone come supporto per la creazione di interfacce grafiche per le app ibride. Per evitare che ogni sviluppatore riparta da zero nella progettazione di una nuova applicazione, Ionic racchiude al proprio interno le best practice per lo sviluppo di interfacce mobili tramite tecnologie web, come *CSS*, *JavaScript* e *HTML* (Figura 1.4).

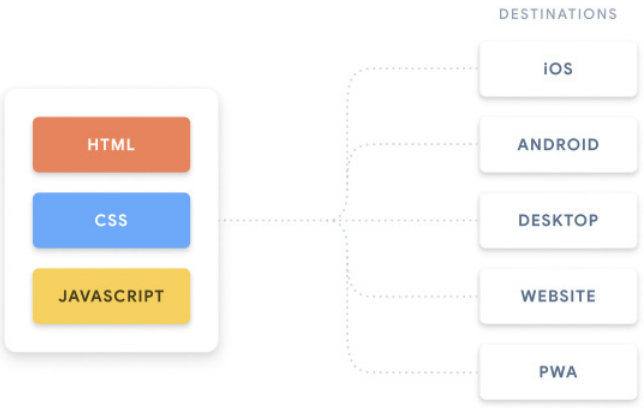


Figura 1.4. Tecnologie di riferimento per creare un'applicazione

Per definire i componenti dell'interfaccia grafica viene utilizzato *AngularJS*. Queste direttive possono essere riutilizzabili ed espandibili in modo da disegnare un'interfaccia grafica nel modo desiderato. L'aspetto grafico di base utilizzato è ispirato ad iOS7, ma può essere facilmente modificato. Dal momento che Ionic ha le sue

radici in Apache Cordova, la creazione di un nuovo progetto Ionic è strettamente correlata ad un progetto Cordova.
Per questo motivo all'interno del progetto si avranno sia file di Cordova, sia file di Ionic. Tra i più importanti troviamo:

- *Il file* `ionic.project`, che contiene la configurazione del progetto Ionic con il quale si andrà a lavorare.
- *La cartella* `scss` che conterrà il codice SASS per la generazione dei CSS.
- *La cartella* `www` che conterrà il codice HTML, JavaScript e CSS utilizzato per sviluppare l'applicazione.

Ionic possiede i seguenti vantaggi:

- *Performance, velocità e reattività del prodotto finale.* Ionic evita completamente l'uso del linguaggio *jQuery* e l'uso di librerie di terze parti per animazioni e transizioni, velocizzando così l'applicazione. In questo modo l'esperienza utente viene migliorata.
- *Interfaccia utente simile ad un'applicazione nativa.* Tutte le funzionalità del framework sono state create per far assomigliare il prodotto finale ad un'applicazione puramente nativa. Inoltre tutte le componenti si vanno perfettamente ad adattare con le altre, senza che il programmatore le debba sistemare nell'interfaccia del dispositivo mobile.

1.2.2 Cordova

Apache Cordova è un framework open-source per lo sviluppo di applicazioni mobile; esso permette di usare tecnologie web standard, come *HTML5*, *CSS3* e *JavaScript* per lo sviluppo di applicazioni *cross-platform*. Le applicazioni vengono eseguite all'interno di alcuni wrapper specifici per ogni piattaforma. Apache Cordova tramite alcune API, garantisce l'accesso alle funzionalità di ogni dispositivo, come sensori, stato, etc.

Ci sono molti componenti nell'architettura di Cordova (Figura 1.5). Questi possono essere raggruppati in tre principali categorie: *web view*, *web app* e *plugins*.

La *web view* caratterizza l'interfaccia utente dell'applicazione. In alcune piattaforme, potrebbe addirittura essere un componente all'interno di un'applicazione Ibrida più grande, che integra la *web view* con componenti dell'applicazione nativa.

Il codice, invece, si trova nella web app. L'applicazione viene implementata come una pagina web che ha al proprio interno *CSS*, *JavaScript*, immagini, file multimediali o altre risorse necessarie. L'app viene eseguita all'interno della *web view* con i wrapper dell'applicazione nativa. Nel progetto avremo un file chiamato `config.xml` che fornirà tutti i parametri che riguardano il funzionamento dell'applicazione, come, ad esempio, l'orientamento dello schermo quando l'applicazione stessa viene aperta.

I *Plugin*, infine, rappresentano una parte essenziale dell'ecosistema di Cordova. Questi forniscono un'interfaccia per far comunicare tra di loro Cordova e i componenti nativi. In questo modo si può richiamare codice nativo attraverso *JavaScript*. All'interno avremo un gruppo di plugin chiamati *Core Plugins*, che permettono all'applicazione di accedere alle funzionalità del dispositivo, come la batteria, la

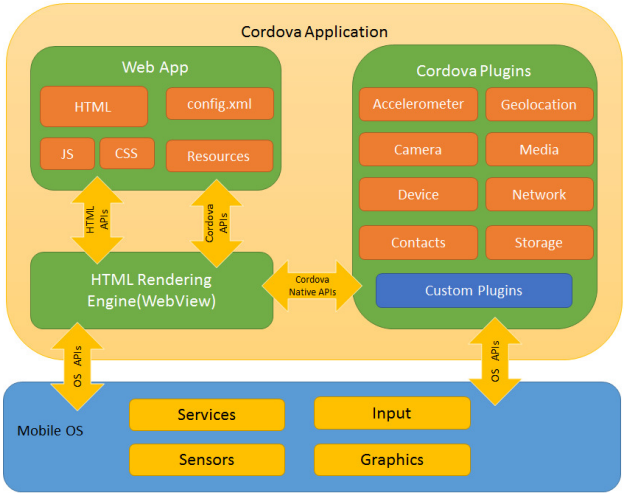


Figura 1.5. Architettura di un'applicazione Cordova

fotocamera, i contatti, etc. In più ci sono dei plugin di terze parti, che permettono di usufruire di alcune funzionalità che potrebbero non esser presenti in tutte le piattaforme.

Infine Cordova fornisce due workflow per creare applicazioni mobili. Questi possono essere usati entrambi per effettuare la stessa cosa, ma forniscono differenti vantaggi:

- *Cross-platform (CLI) workflow*: se si vuole che la propria applicazione funzioni sul numero maggiore possibile di sistemi operativi mobili questa è la scelta più adatta. Questo workflow si concentra sulla CLI di Cordova. Quest'ultima è un pannello di alto livello che permette di creare progetti per più applicazioni alla volta, astraendo la maggior parte delle funzionalità di basso livello. Inoltre la CLI fornisce un'interfaccia per aggiungere plugin alla propria applicazione.
- *Platform-centered workflow*: se si vuole focalizzare un'applicazione su una singola piattaforma e si ha bisogno di fare modifiche di basso livello è consigliato questo approccio. Per esempio, se si vogliono integrare componenti native con componenti web di Cordova questo è il workflow da utilizzare. Con questo approccio, però, è più complicato realizzare app cross-platform, per la mancanza di pannelli che permettano di modificare l'applicazione con funzioni di alto livello; tale mancanza comporta che bisognerebbe creare funzioni differenti per ciascuna piattaforma.

1.2.3 React Native

React Native (Figura 1.6) è un framework open source realizzato da Facebook che consente di sviluppare applicazioni multipiattaforma mobili utilizzando React, una libreria JavaScript per creare interfacce utente. La principale differenza tra React e React Native consiste nel fine dell'applicazione, ovvero:

- *il DOM* nel caso di React

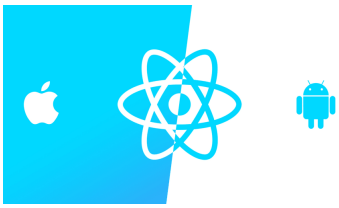


Figura 1.6. Logo di React Native

- *la UI* nativa della piattaforma corrente nel caso di React Native

Con React Native si riesce a sviluppare un'applicazione che utilizza gli stessi componenti di una nativa. In più il programmatore può aggiungere al framework i propri binding nativi. Questa flessibilità consente di avere app ibride in cui una parte è sviluppata nativamente e l'altra in React Native.

Il mapping dei componenti React Native nei componenti nativi viene delegato al framework, ma non c'è nessun tipo di aiuto da parte di React Native su come questi componenti dovrebbero esser accostati per avere un'interfaccia nativa coerente. Questo, quindi, potrebbe essere un problema perchè porta il programmatore a dover essere informato sulle linee guida di iOS e Android prima di progettare un'applicazione ibrida con questo tipo di framework.

In più, alcune funzionalità potrebbero non esser supportate; quindi, il programmatore deve controllare se esistono pacchetti di terzi che le implementano oppure deve implementarsele da solo, il che potrebbe portar via tempo. React Native guadagna molto in termini di performance rispetto alle comuni web app, perchè fa uso di un'interfaccia nativa e non delega alla *web view* il rendering dell'interfaccia utente. Infine, React Native ha un ciclo di sviluppo molto rapido. In questo modo, ad ogni modifica del codice, non è necessario ricompilare l'intera applicazione, ma basterà compilare soltanto la nuova modifica effettuata.

Non essendo un'applicazione interamente nativa, nonostante l'interfaccia utente cerchi di essere il più simile possibile, la performance di un'applicazione React Native risulta minore rispetto a quella di un'app nativa.

1.2.4 Flutter

Flutter è un progetto open source che permette di sviluppare applicazioni in tempi rapidi con il supporto alle interfacce native. È stato sviluppato e progettato da Google nel 2018 e mantenuto da questo insieme alla community. Il suo obiettivo è quello di creare applicazioni tramite una rapida fase di sviluppo, chiamata *hot reload*, che non richiede di ricompilare il codice. Le interfacce utente sono flessibili con un insieme di widget componibili, librerie per animazioni e un'architettura estensibile e stratificata. Flutter, infatti, è composto da due macrostrati:

- *Uno strato* scritto in *C/C++*.
- *Uno strato* scritto in *Dart*, un nuovo e moderno linguaggio orientato agli oggetti, che definisce la maggior parte del suo sistema ed offre ai programmatori un controllo quasi totale sul sistema stesso.

Differentemente da altri framework, Flutter non utilizza *web view* o widget nativi per lo sviluppo di app ibride. Infatti esso sfrutta il proprio motore di rendering ad alte prestazioni per creare i widget, senza passare attraverso quelli del dispositivo (Figura 1.7).

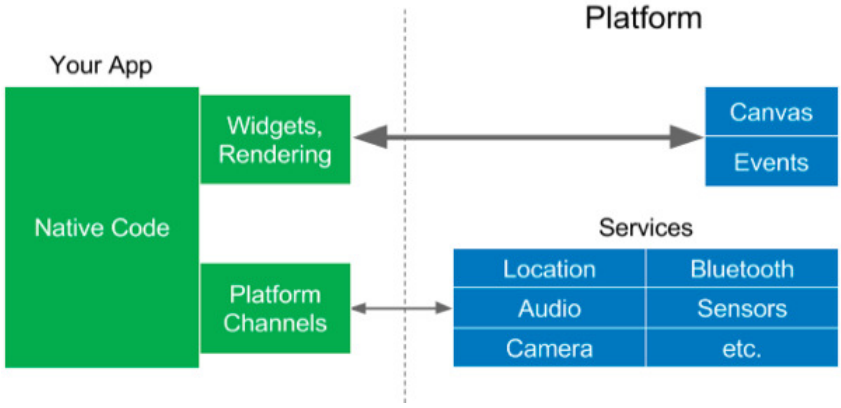


Figura 1.7. Interazione tra un'applicazione Flutter e la piattaforma

Un'altra delle novità di Flutter è la modalità di gestione del layout. Il sistema, infatti, determina la dimensione e la posizione degli elementi sulla base di un insieme di regole. Tradizionalmente, i layout usano un grande numero di regole che possono essere applicate a qualsiasi elemento. Il problema dei layout tradizionali è che le varie regole possono andare in conflitto tra di loro creando problemi sugli elementi da visualizzare. Per risolvere questi problemi Flutter ha deciso che ogni widget deve specificare un proprio layout in modo tale che ciascun layout diventi parte del singolo oggetto da rappresentare.

In più, siccome i widget diventano una parte dell'app, non è necessaria nessuna ulteriore libreria. L'applicazione, quindi, funzionerà anche sui più recenti sistemi operativi, senza bisogno di ulteriori modifiche. Siccome Flutter non utilizza i widget nativi delle piattaforme su cui lavora, questi potrebbero richiedere meno aggiornamenti e modifiche rispetto alle singole piattaforme.

1.2.5 Xamarin

Xamarin (Figura 1.8) è un framework per lo sviluppo di applicazioni mobile cross-platform. Nasce nel 2011 negli USA da un'azienda chiamata *Mono*. In poco tempo raggiunge grandi numeri di sviluppatori, così da farsi notare da *Microsoft*, che decide di acquistare l'azienda nel 2016. Tramite un codice condiviso ad oggetti basato su *C#*, si possono utilizzare gli strumenti Xamarin per scrivere applicazioni per Android, iOS e Windows con interfacce native e codice condiviso.

Per ogni piattaforma Xamarin permette allo sviluppatore di utilizzare gli elementi di un'interfaccia nativa tramite degli specifici tool. Ciò però, presenta lo svantaggio di dover implementare una UI per ciascuna piattaforma; tuttavia, il *code-behind* verrà scritto un'unica volta, andandolo ad implementare *cross-platform*.

Figura 1.8. Logo di Xamarin

Essendo Xamarin il framework da noi utilizzato nella presente tesi, esso verrà descritto in dettaglio nel prossimo capitolo.

2

Xamarin

In questo capitolo approfondiremo Xamarin, vedendo nello specifico le componenti da cui è formato. Successivamente vedremo la sua architettura, i suoi principali ambienti di lavoro e i diversi approcci che permettono la condivisione del codice tra le varie piattaforme.

2.1 Componenti di Xamarin

Xamarin è formato da tre principali componenti, `Xamarin.Forms`, `Xamarin.Android` e `Xamarin.iOS`, tramite i quali è possibile gestire in *C#* tutte le caratteristiche di Android, iOS e Windows Phone, dall'interfaccia utente alle risorse hardware del dispositivo.

Il principale vantaggio di questa suddivisione è che è possibile creare interfacce utente native *cross-platform* in Xamarin.Forms. Tuttavia, se si ha il bisogno di utilizzare librerie specifiche di una determinata piattaforma a livello inferiore, si può utilizzare Xamarin.Android o Xamarin.iOS (Figura 2.1).

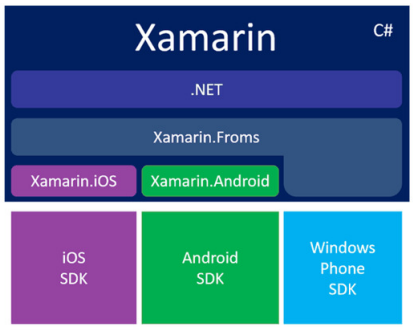


Figura 2.1. Il legame tra i componenti di Xamarin e le SDK native

2.1.1 Xamarin.Forms

Xamarin.Forms è un "*UI toolkit cross-platform*" che permette agli sviluppatori di creare facilmente layout di interfacce native, che possono essere utilizzate in Android, iOS e Windows Phone. Infatti, ponendosi al di sopra di queste tre componenti, Xamarin.Forms consente di avere una piena condivisione del codice tra le varie piattaforme.

Le principali caratteristiche che rendono conveniente questo approccio allo sviluppatore sono:

- *Binding tra proprietà e controlli*, in modo da separare i dati dall'interfaccia utente, migliorando la leggibilità e la manutenzione dell'app.
- *Messagging Center*, ovvero un servizio attraverso il quale diversi componenti e classi possono comunicare tra di loro senza conoscersi.
- *Dependency Service*, tramite la quale le applicazioni Xamarin.Forms possono richiamare funzionalità della piattaforma nativa dal codice condiviso.
- *Possibilità di aggiungere controlli specifici della piattaforma*, ovvero la capacità di inserire in Xamarin.Forms proprietà disponibili soltanto sulle singole piattaforme.

2.1.2 Xamarin.Android

Le applicazioni Xamarin.Android vengono eseguite tramite l'ambiente di esecuzione *Mono*, che permette lo sviluppo di app Android tramite un *wrapping* delle API native in *C#*. Un'applicazione Xamarin.Android, in fase di compilazione, passa da un linguaggio in *C#* ad un linguaggio intermedio (IL: *Intermediate Language*), il quale, solo in tempo di esecuzione, viene trasformato in linguaggio nativo, per esempio quando l'applicazione viene avviata (compilazione *Just-in-Time*). Infine, l'applicazione verrà eseguita comunque sulla macchina virtuale di *Android Runtime (ART)*.

Come si può vedere dall'architettura in Figura 2.2, Xamarin.Android utilizza due wrapper: *Managed Callabel Wrappers (MCW)* e *Android Callable Wrappers (ACW)*.

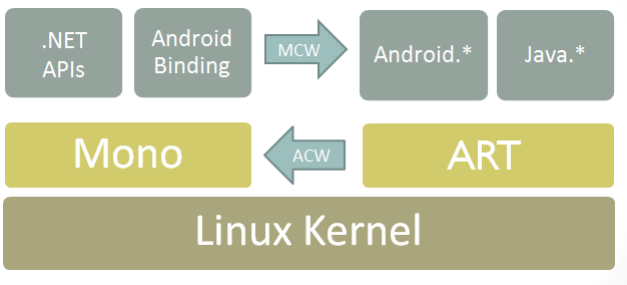


Figura 2.2. Architettura approssimativa di Xamarin.Android

MCW viene usato ogni volta che una porzione di codice *C#* deve richiamare codice nativo Android e fornisce supporto per l'implementazione di interfacce Java. Ogni *MCW* che viene creato include un riferimento globale a Java.

ACW viene usato ogni volta che il runtime di Android deve richiamare una porzione di codice in *C#*. Tramite questo wrapper possono essere implementati metodi virtuali ed interfacce Java.

2.1.3 Xamarin.iOS

Basato sulla libreria proprietaria *MonoTouch*, Xamarin.iOS offre pieno supporto allo sviluppo di app iOS tramite un *wrapping* delle API native in *C#*.

Le applicazioni Xamarin.iOS usano la compilazione *AOT* (*Full ahead of Time*) per compilare il codice nel linguaggio assembly *ARM*. Questa viene eseguita insieme al runtime ad oggetti *C*. Entrambi gli ambienti runtime vengono eseguiti su un kernel di tipo *UNIX* e consentono agli sviluppatori di accedere al sistema nativo sottostante. Una panoramica di quest'architettura è presente nella Figura 2.3.

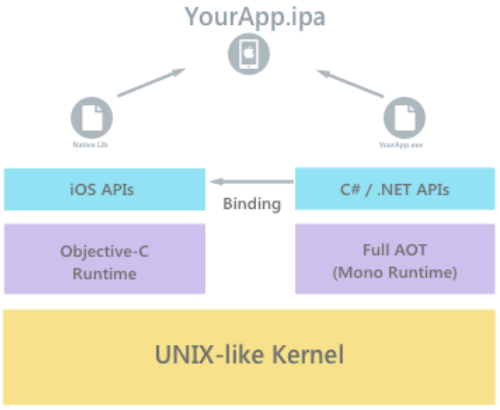


Figura 2.3. Architettura approssimativa di Xamarin.iOS

Viene utilizzato un compilatore *AOT* perchè in iOS è presente una restrizione di sicurezza che impedisce l'esecuzione di codice generato dinamicamente in un dispositivo. Il compilatore *AOT*, invece, produce un file binario iOS nativo, che può essere distribuito nel processore basato su ARM di Apple. Inoltre, rispetto ad un compilatore *Just-in-Time (JIT)*, fornisce una serie di miglioramenti tra cui diverse ottimizzazioni delle prestazioni.

2.2 Architettura di un'applicazione

Un'applicazione è un insieme di componenti che comunicano tra di loro; alcuni di essi sono indispensabili per il corretto funzionamento dell'app. In Xamarin una

classe fondamentale di questo tipo è la classe `Application` (Listato 2.1), la quale permette di configurare le caratteristiche principali dell'applicazione, come la pagina principale, oppure di associare ad un particolare stato del ciclo di vita di un'activity un determinato evento.

```
1   public partial class App : Application
2       {
3           public App()
4           {
5               InitializeComponent();
6
7               MainPage = new MainPage();
8           }
9
10          protected override void OnStart()
11          {
12              // Handle when your app starts
13          }
14
15          protected override void OnSleep()
16          {
17              // Handle when your app sleeps
18          }
19
20          protected override void OnResume()
21          {
22              // Handle when your app resumes
23          }
24      }
```

Listato 2.1. Definizione della classe `Application`

Dalla Figura 2.4 si può, invece, vedere un esempio di architettura di un'applicazione Xamarin. Una parte di codice sarà condivisa con tutti i progetti (*Shared Code*), mentre le restanti, come l'interfaccia utente o le librerie specifiche della piattaforma, apparterranno alle singole piattaforme.

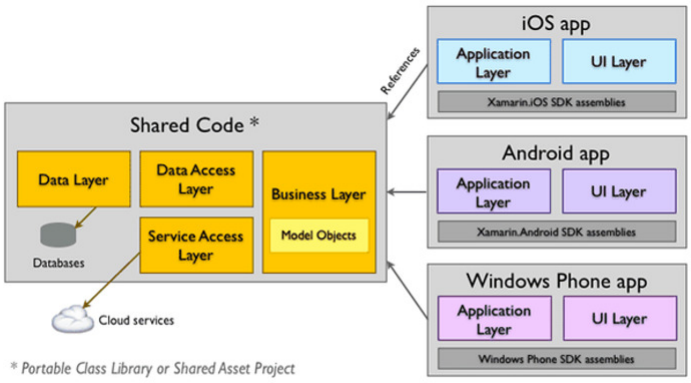


Figura 2.4. Esempio di architettura cross-platform

2.2.1 Page, Layout e View

I componenti principali che vanno a comporre un'applicazione Xamarin sono *Page*, che rappresenta la schermata dell'applicazione in Xamarin, *Layout*, il quale viene

usato per trasformare controlli dell'interfaccia utente in strutture visive, e *View*, che permette la visualizzazione dei *building block* dell'interfaccia utente.

Per quanto riguarda Page, questa deriva dalla classe Xamarin.Forms e solitamente occupa tutto il display del dispositivo. In iOS, nello specifico, andrà a rappresentare un *View Controller*, mentre in Android occuperà la schermata come un'*Activity*. Xamarin.Forms supporta i seguenti tipi di Page (Figura 2.5):

- *ContentPage*: è il tipo di pagina più semplice e comune. Verrà impostata la proprietà `Content` su un singolo oggetto View, che solitamente è un layout.
- *MasterDetailPage*: gestisce due riquadri di informazioni. Generalmente il riquadro principale è una pagina con una lista o un menù, che prenderà la proprietà `Master`. Il secondario, che avrà la proprietà `Detail`, mostrerà l'oggetto selezionato tramite la *Master Page*.
- *NavigationPage*: gestisce la navigazione tramite più pagine usando un'architettura a *stack*. Quando viene usata una NavigationPage, un'istanza della pagina principale viene passata al costruttore dell'oggetto.
- *TabbedPage*: questo tipo di pagina deriva dalla classe astratta `MultiPage` e permette la navigazione attraverso pagine figlie utilizzando le schede.
- *TemplatedPage*: mostra un contenuto a schermo intero ed è la classe di base per `ContentPage`.
- *CarouselPage*: deriva dalla classe astratta `MultiPage` e permette la navigazione attraverso le pagine figlie tramite lo scorrimento (*finger swiping*).

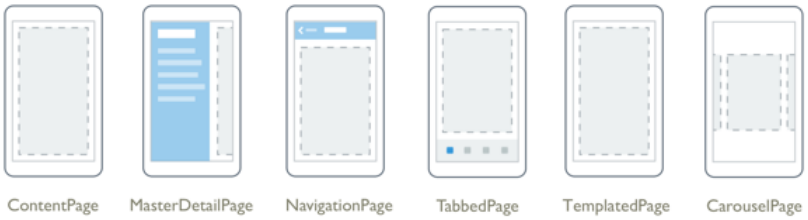


Figura 2.5. Tipi di Page in Xamarin.Forms

I layout in Xamarin.Forms, invece, fungono da contenitori per le View e, a loro volta, per altri layout. Questi possono essere divisi in due categorie: layout con contenuto singolo (*Layouts with Single Content*) e layout con più figli (*Layouts with Multiple Children*).

La prima categoria deriva dalla classe `Layout`; in essa troviamo:

- *ContentView*: contiene un singolo elemento figlio con la proprietà `Content`. Viene principalmente usato come elemento strutturale e funge da classe base per Frame.
- *Frame*: deriva da ContentView e visualizza un bordo intorno al relativo figlio.
- *ScrollView*: tramite questo layout è possibile scorrere il contenuto della pagina quando questo è troppo grande per essere visualizzato tutto insieme sullo schermo.

- *TemplatedView*: mostra il contenuto con un modello di controllo ed è la classe base di ContentView.
- *ContentPresenter*: è un gestore di layout per le visualizzazioni basate su modelli per contrassegnare il punto in cui viene visualizzato il contenuto.

La seconda categoria deriva dalla classe `Layout<View>` e a questa appartengono:

- *StackLayout*: posiziona gli elementi figli in uno stack orizzontalmente o verticalmente, in base alla proprietà `Orientation`.
- *Grid*: posiziona gli elementi figlio in una griglia di righe e colonne tramite delle proprietà impostate dal programmatore.
- *AbsoluteLayout*: posiziona gli elementi figlio in posizioni specifiche rispetto al relativo padre, tramite le proprietà `LayoutBounds` e `LayoutFlags`.
- *RelativeLayout*: posiziona gli elementi figli rispetto al layout stesso o rispetto ad altri elementi che si trovano allo stesso livello.
- *FlexLayout*: definisce delle proprietà che consentono di impilare o incapsulare i vari elementi figli.

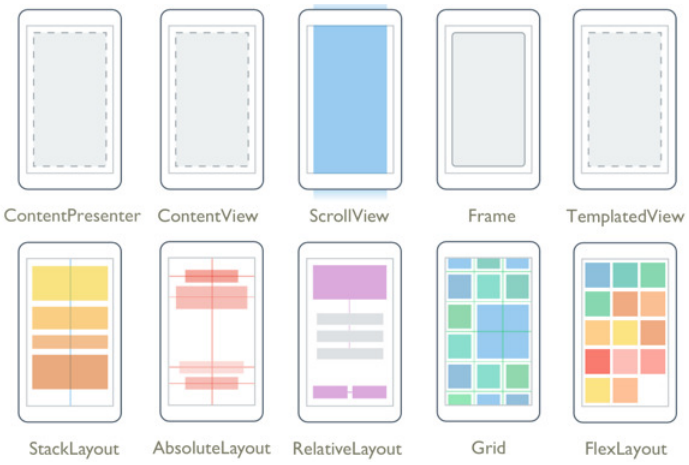


Figura 2.6. Tipi di Layout in Xamarin.Forms

In ultimo le View sono oggetti dell'interfaccia utente come, per esempio, campi di testo o pulsanti, e vengono chiamati comunemente widget. Le View di Xamarin.Forms derivano tutte dalla classe `View` e possono essere divise in diverse categorie: viste per la presentazione (*Views for presentation*), viste che inizializzano comandi (*Views that initiate commands*), viste per configurare parametri (*Views for setting values*), viste per la modifica del testo (*Views for editing text*), viste che indicano attività (*Views to indicate activity*) e viste che mostrano un insieme di oggetti (*Views that display collections*). Alcuni tipi di viste vengono mostrate nella Figura 2.7.

Tra le View più utilizzate troviamo:

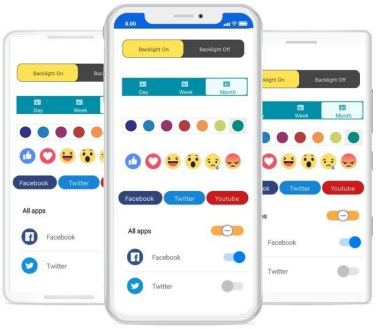


Figura 2.7. Alcuni tipi di View in Xamarin.Forms

- *Label*: mostra una singola linea di testo o blocchi di testo con più linee a formattazione costante o variabile. Il testo può essere impostato tramite la proprietà `Text`.
- *Image*: visualizza una *bitmap*. Questo tipo di immagine deve essere inserito all'interno della cartella del progetto per poter, poi, essere mostrato nell'applicazione.
- *BoxView*: visualizza un rettangolo a tinta unita colorato tramite la proprietà `Color`.
- *Button*: è un oggetto rettangolo che visualizza un testo e genera un evento `Clicked` quando viene premuto; ad esso si possono collegare alcune azioni.
- *Slider*: consente all'utente di specificare un valore tra un minimo ed un massimo, specificati tramite le proprietà `Minimum` e `Maximum`.
- *Switch*: è un oggetto tramite il quale l'utente può attivare o disattivare una determinata opzione. Nel codice un valore booleano verrà impostato a 0 o a 1.
- *DatePicker*: consente all'utente di selezionare una data tramite il selettore specifico della piattaforma, tipicamente un piccolo calendario.
- *Entry*: consente all'utente di immettere o modificare una singola riga di testo. Solitamente viene usato per gestire i Login o per far inserire brevi informazioni all'utente.
- *Editor*: consente di immettere e modificare più righe di testo. Viene tipicamente utilizzato quando si devono gestire grandi quantità di testo, per le quali un Entry non basterebbe.
- *ProgressBar*: usa un'animazione per mostrare che l'applicazione sta procedendo a compiere un'operazione di lunga durata. Viene utilizzata attraverso la proprietà `Progress`.
- *CollectionView*: visualizza un elenco scorrevole di elementi selezionabili, utilizzando diversi Layout. Offre un'alternativa più flessibile ed efficiente di un ListView. L'evento `SelectionChanged` segnala che è stato selezionato un elemento dall'utente.
- *ListView*: visualizza un elenco scorrevole di elementi di dati selezionabili, tramite le proprietà `ItemSource` e `ItemTemplate`. L'evento `ItemSelected` segnala che è stato selezionato un determinato oggetto dall'utente.

2.2.2 Ciclo di vita di un'applicazione

Il ciclo di vita di un'applicazione generica (Figura 2.8) prevede 4 stati principali. Quando l'applicazione viene avviata passerà dallo stato di *Not Running* allo stato di *Running* e rimarrà in questo stato finchè sarà in primo piano. Passerà, invece, a quello di *Pause/Inactive* nel momento in cui il dispositivo viene, ad esempio, bloccato o l'applicazione passa in secondo piano. Invece, se si utilizzerà un'altra app, l'applicazione originaria passerà nello stato di *Backgrounded* per poi, infine, passare a quello di *Not Running*, quando viene chiusa.

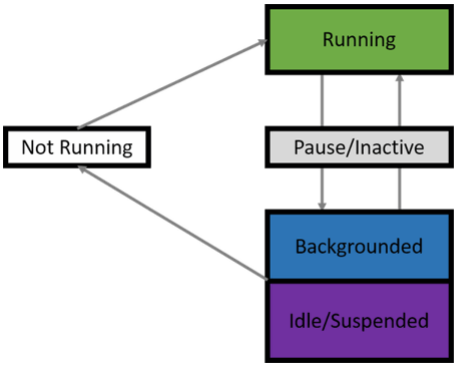


Figura 2.8. Stati in cui si può trovare un'applicazione

Un'applicazione Xamarin, invece, salta uno di questi passaggi, non considerando lo stato *Not Running*. Infatti, non è disponibile alcun metodo per la terminazione dell'applicazione. In circostanze normali, ovvero quando non avvengono interruzioni anomale causate da errori, la terminazione dell'applicazione viene eseguita dallo stato *OnSleep*.

Come mostrato dalla Figura 2.9, il ciclo di vita in Xamarin.Forms è definito attraverso tre stati:

- `OnStart`: viene invocato quando l'app è inizializzata.
- `OnSleep`: viene invocato quando l'app viene sospesa ed è in background.
- `OnResume`: viene invocato quando l'app è nuovamente in foreground dopo essere stata messa in background.

Questi stati sono, anche, metodi virtuali che possono essere trovati all'interno della classe `Application`.

2.2.3 MVVM

Il *Model-View-ViewModel* (MVVM) è un pattern software architetturale che permette di separare la logica dell'applicazione dall'interfaccia utente attraverso l'uso di un controller. Il pattern MVVM viene definito da Microsoft per lo sviluppo di

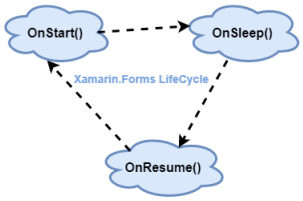


Figura 2.9. Ciclo di vita di un'applicazione Xamarin

applicazioni *Windows Phone* e *Silverlight*, per questo è pienamente supportato da Xamarin.

L'applicazione viene divisa in tre livelli: Model, View e ViewModel.

- *Il Model* è il punto di accesso ai dati ed è costituito da un insieme di classi che permettono di gestire i dati utilizzati nell'applicazione. Tipicamente in Xamarin è una pagina dell'app.
- *La View* è l'interfaccia grafica dell'app e permette all'utente di interagire con i dati.
- *Il ViewModel* serve a far comunicare la view con il model e permette di gestire i dati provenienti dal model per mostrarli nella View, e viceversa.

Come si può notare dalla Figura 2.10, l'utente interagisce con la View che raccoglie i dati e li inoltra al ViewModel. Quest'ultimo effettua un cambio di stato e accede, se necessario, al Model. Allo stesso tempo, il ViewModel aggiorna il suo stato, così da tenere aggiornata anche la View. In questo modo, View e Model vengono disaccoppiati tra di loro, ma comunque rimangono aggiornati su ogni cambiamento.

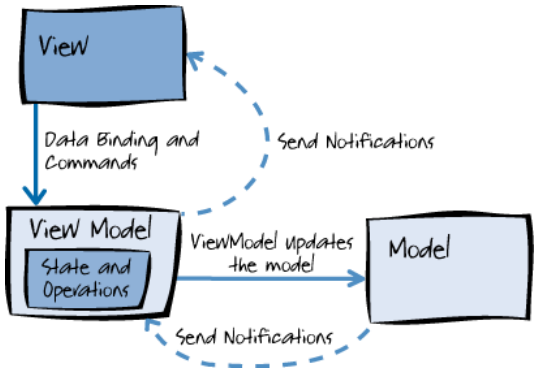


Figura 2.10. Modalità di comunicazione tra i diversi strati

Per capire bene il ruolo del pattern MVVM consideriamo i vari strati di un'applicazione Xamarin (Figura 2.11):

- *Application Layer*: questa parte del codice permette all'applicazione di girare su ogni dispositivo ed ha un'implementazione specifica per ogni piattaforma. Non rientra puramente nel pattern MVVM, ma le varie librerie del pattern forniscono

alcuni elementi per la gestione dell'applicazione. In questo modo si può anche avere del codice cross-platform in grado di controllare la logica dell'app.
- *UI Layer*: è il View layer e sarà specifico per ogni piattaforma.
- *Binding*: il binding tra l'UI layer e l'UI logic layer è dato dal binder. Questo, generalmente, è un misto di codice cross-platform e codice specifico per la singola piattaforma.
- *UI logic layer*: è il ViewModel layer. Esso fornisce la logica per l'UI e altre interazioni con codice cross-platform. Una parte di questa logica è dedicata alla conversione dei dati dagli oggetti del dominio in dati della UI.
- *Business logic layer*: rappresenta il Model layer e contiene i dati, gli oggetti del dominio e la logica di business. Viene implementato tramite codice cross-platform.

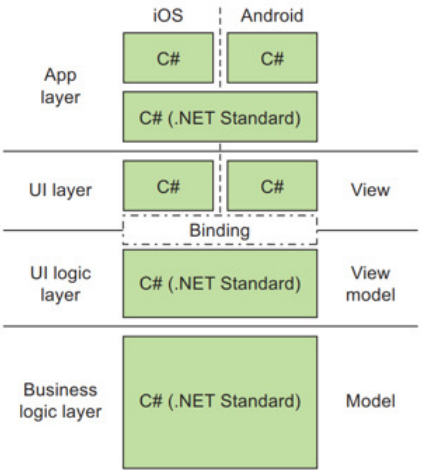


Figura 2.11. Struttura del pattern MVVM

2.3 I principali IDE

Un IDE (*Integrated Development Environment*) è un ambiente di sviluppo integrato che, in fase di programmazione, supporta i programmatori nello sviluppo del codice sorgente di un programma.

Normalmente è uno strumento software con più componenti, tra cui un editor del codice sorgente, un compilatore, un tool di building automatico e un debugger. Un compilatore è un programma che traduce una serie di istruzioni scritte in codice sorgente in istruzioni di un altro linguaggio, chiamato linguaggio oggetto.

Nel caso di applicazioni Xamarin ci sono due ambienti di sviluppo principalmente usati, ovvero Xamarin Studio e Visual Studio.

2.3.1 Xamarin Studio

Xamarin Studio è stato il primo IDE disponibile per la creazione di app Xamarin. Su Xamarin Studio non è possibile creare applicazioni per Windows Phone. Tramite questo IDE, su Mac, si possono sviluppare app sia per Android che per iOS. Su computer Windows, invece, si possono sviluppare solamente applicazioni Android e lo stesso IDE incoraggia gli utenti a spostarsi su Visual Studio.

Appare come un IDE molto semplice da utilizzare, ma la sua interfaccia accessibile nasconde tool molto potenti, che includono molte funzionalità che si possono ritrovare anche in Visual Studio, tra cui:

- *Sintassi evidenziata con diversi colori*, in modo da facilitare il lavoro allo sviluppatore.
- *Tool per la navigazione all'interno del progetto*, che rendono facile spostarsi da un file all'altro.
- *Meccanismi di debug ben sviluppati*, inclusa la visualizzazione del valore corrente quando viene passato il puntatore su una variabile.
- *Integrazione nativa di strumenti di terze parti*, come ad esempio GIT.

Xamarin Studio ha, però, anche degli aspetti negativi:

- *I tasti di scelta rapida*, come, ad esempio, la funzionalità copia-incolla, sono solo nel layout inglese. Gli sviluppatori hanno segnalato questo problema diverse volte all'azienda produttrice dell'IDE.
- *A volte ci sono errori di compilazione*, che costringono il programmatore a dover riavviare l'intero programma.

Successivamente non sono più stati fatti aggiornamenti di Xamarin Studio e tutte le sue funzionalità sono state passate su Visual Studio. Al momento, infatti, Xamarin Studio è stato rimpiazzato con *Xamarin for Visual Studio* in Windows. Su macOS, invece, è ancora in fase di sviluppo, ma nel 2016 è stato rinominato come *Visual Studio for Mac*.

2.3.2 Visual Studio

Microsoft Visual Studio (Figura 2.12), conosciuto più comunemente come Visual Studio, è un IDE sviluppato dalla Microsoft. È un IDE multilinguaggio e al momento supporta la creazione di progetti per varie piattaforme, tra cui anche console o dispositivi mobili, nonchè diversi linguaggi di programmazione tra cui `C#`, `.Net` e `C++`, ma resta incompatibile con `Java`.

Visual Studio integra la tecnologia *IntelliSense*, che permette di correggere errori sintattici, e anche logici, senza il bisogno di dover prima compilare l'applicazione. Tramite il suo compilatore *.NET*, converte il codice sorgente in codice IL (*Intermediate Language*). IL è un linguaggio di basso livello rispetto al codice sorgente, ma ha un livello di astrazione più alto rispetto ai linguaggi macchina.

Le funzionalità più note di Visual Studio sono le seguenti:

- *Controllo ortografia e azioni rapide*: durante la digitazione vengono automaticamente evidenziati potenziali errori, di ortografia o di codice, in tempo reale, e vengono mostrate delle azioni rapide utilizzabili per risolvere l'errore.

Figura 2.12. Logo di Visual Studio 2017

- *Pulizia del codice*: questa funzionalità consente di risolvere problemi del codice prima di lanciare l'applicazione.
- *Casella di ricerca*: essendo Visual Studio un IDE ricco di impostazioni e proprietà, viene messa a disposizione del programmatore una casella di ricerca, in modo da trovare velocemente il contenuto che si cerca.

Al momento Visual Studio è l'IDE più aggiornato per lo sviluppo di applicazioni Xamarin.

2.4 Condivisione del codice

Sin dalla creazione di un progetto Xamarin è possibile scegliere la modalità di condivisione del codice tra i progetti delle varie piattaforme. Per capire l'utilità di questa scelta, riprendendo in esame la Figura 2.4, dividiamo l'applicazione in due macro progetti:

- *Core Project*: questo è il cuore dell'applicazione ed è composto dagli elementi principali, come l'accesso ai dati e la logica di business.
- *Platform-Specific Application Projects*: è composto dai progetti di tutte le piattaforme in cui l'applicazione è supportata. Ogni progetto dovrebbe avere al proprio interno la gestione dell'UI e la gestione delle funzionalità specifiche della singola piattaforma.

Ci sono due possibili modalità di condivisione: SAP e PCL. La scelta su quale utilizzare dipende dagli obiettivi finali dell'applicazione; entrambi hanno *pro e contro* differenti, che potrebbero aiutare il programmatore a scegliere l'approccio giusto al momento dell'implementazione.

2.4.1 SAP

L'approccio SAP (*Shared Asset Project*) fa in modo che il codice sia condiviso tra i progetti durante la compilazione della soluzione (Figura 2.13). Concettualmente, lo Shared Project viene copiato in ogni progetto della soluzione che lo referenzia e viene compilato come parte del progetto.

Il limite di questi progetti, però, è dato dal fatto che i progetti Android e iOS accedono ad una versione del framework .NET differente rispetto a quella usata

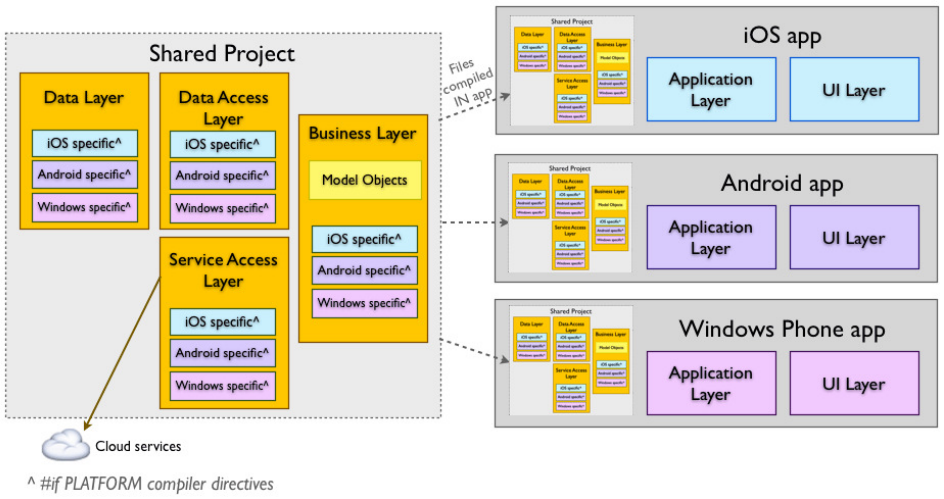


Figura 2.13. Architettura di un'app con approccio SAP

da Windows. Per questo motivo, bisogna differenziare il codice, come, per esempio, l'accesso al database, per ogni piattaforma utilizzata.

Per risolvere questo problema ci viene incontro il compilatore, che supporta alcune direttive C#, come `#if`, `#elif` e `#endif`. Queste direttive vengono integrate con dei simboli che permettono di identificare la piattaforma con la quale stiamo lavorando:

- `__MOBILE__`: utilizzato nei progetti iOS e Android.
- `__IOS__`: definito nel progetto iOS.
- `__ANDROID__`: definito nel progetto Android.
- `__ANDROID_nn__`: definito per Android, dove `nn` rappresenta il numero dell'API che deve eseguire una particolare azione.
- `WINDOWS_PHONE_APP`: definito nel progetto per Windows Phone.
- `WINDOWS_UWP`: utilizzato nei progetti UWP.

Utilizzando simboli e condizioni si possono integrare le funzionalità delle varie piattaforme (Listato 2.2). In questo modo, durante l'esecuzione dell'applicazione, il compilatore saprà in quale progetto deve andare a prendere una determinata parte di codice.

```
1   #if __IOS__
2           // codice specifico per iOS
3   #elif __ANDROID__
4           // codice specifico per Android
5   #elif WINDOWS_PHONE_APP
6           // codice specifico per Windows Phone
7
8   //supporto agli altri progetti
9
10  #endif
```

Listato 2.2. Esempio di un blocco di istruzioni utilizzando l'approccio SAP

2.4.2 PCL

L'approccio PCL (*Portable Class Libraries*) costituisce un approccio alternativo a quello SAP. Come si può vedere dalla Figura 2.14, ogni Platform-Specific Application Project referenzia il PCL, e si possono utilizzare le caratteristiche di una determinata piattaforma attraverso l'utilizzo del pattern *Dependency Injection* (DI). La Dependency Injection offre linee guida per includere le funzionalità specifiche di una piattaforma nella soluzione di riferimento.

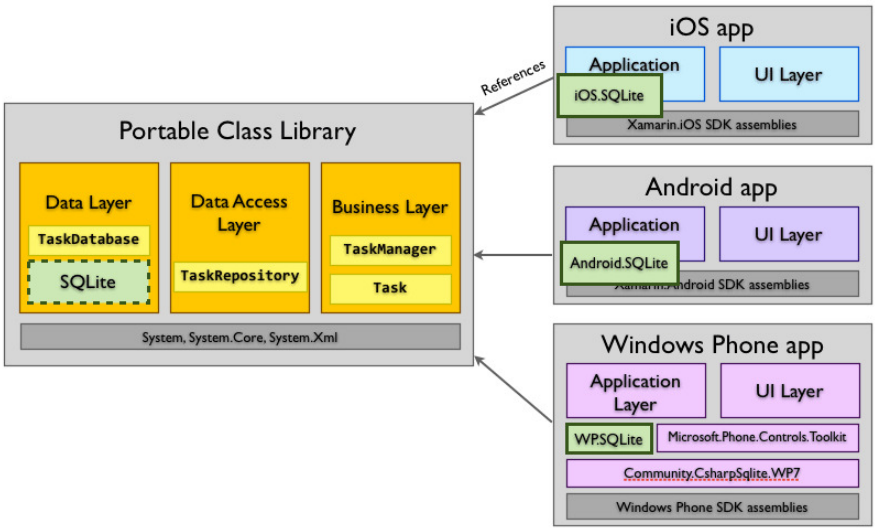


Figura 2.14. Architettura di un'app con approccio PCL

Xamarin offre la classe `DependencyService` per implementare i principi del DI attraverso la creazione di un'interfaccia definita nel PCL, l'implementazione specifica delle interfacce all'interno della singola piattaforma, utilizzando il tag `assembly` (Listato 2.3) e l'invocazione del metodo specifico all'interno del codice condiviso, utilizzando il metodo `Get` (Listato 2.4).

```
1  [assembly: Xamarin.Forms.Dependency(typeof(NomeClasse_PIATTAFORMA))]
```

Listato 2.3. Esempio di utilizzo del tag `assembly` per l'approccio PCL

```
1  DependencyService.Get<INOMECLASSE>.Nome_Metodo
```

Listato 2.4. Esempio di utilizzo del metodo `Get` nell'approccio PCL

Una volta sviluppato il codice condiviso, la compilazione dell'intera soluzione genera una `DLL` che sarà referenziata da tutti i progetti. In questo modo la DLL potrà essere riutilizzata anche in altre soluzioni, cosa non permessa nell'approccio SAP.

3

Analisi dei requisiti

In questo capitolo affronteremo l'analisi dei requisiti utili per l'applicazione da realizzare. Prima di tutto descriveremo il progetto e i suoi requisiti, funzionali e non funzionali. In seguito tratteremo gli attori e i relativi casi d'uso.

3.1 Descrizione del progetto

L'applicazione da sviluppare dovrà consentire agli utenti di un determinato tavolo di effettuare un'ordinazione. Durante l'ordine, l'app dovrà permettere agli utenti di scegliere facilmente il piatto d'interesse, fornendo un'immagine realistica e rappresentativa della pietanza, gli ingredienti che lo costituiscono e la sua descrizione.

Dovrà anche essere possibile inserire ulteriori ingredienti al piatto d'interesse, scegliendo tra le aggiunte specifiche della pietanza suggerite dall'applicazione. Inoltre, l'utente potrà inserire delle note per il piatto, per comunicare allergie, intolleranze o particolari modifiche da effettuare prima di concludere l'ordinazione.

Tramite un'interfaccia intuitiva e pulita, l'app si pone come obiettivo quello di ridurre al minimo le incomprensioni che potrebbero manifestarsi durante l'ordinazione di un pasto, permettendo all'utente, mediante diversi passaggi, di completare e inviare un ordine direttamente alla cucina. Si è, anche, deciso di inserire la possibilità di richiedere aiuto tramite l'app, in caso ci fossero problemi legati all'utilizzo, in modo da facilitarne l'uso.

L'utente potrà registrarsi e autenticarsi all'interno dell'applicazione, così da poter usufruire di alcuni sconti personalizzati, differenti a seconda della spesa fatta, grazie alla presenza dei punti che via via si possono accumulare. All'interno del proprio account è, anche, possibile visualizzare uno storico delle vecchie ordinazioni ancora non pagate, dando all'utente la possibilità di tenerle sotto controllo. Si è deciso di far spendere all'utente automaticamente alcuni dei punti guadagnati negli ordini precedenti durante la fase di pagamento. I punti non spesi resteranno validi per ottenere sconti negli ordini successivi.

Infine, l'utilità maggiore dell'applicazione risiede nel ridurre i tempi di attesa del cliente nel momento dell'ordinazione, non essendoci più il bisogno di un cameriere.

Allo stesso tempo, viene alleggerito il lavoro di questi ultimi, che non dovranno più preoccuparsi di prendere le ordinazioni nei vari tavoli.

3.2 Requisiti funzionali e non funzionali

Dopo una prima descrizione del progetto da realizzare, un'importante fase è quella della raccolta dei requisiti funzionali e non funzionali. Lo scopo fondamentale sarà quello di individuare le caratteristiche che dovranno, poi, essere realizzate all'interno dell'applicazione.

I *requisiti funzionali* sono indipendenti dalla tecnologia, dall'architettura e dal linguaggio di programmazione e descrivono i servizi e il comportamento del sistema quando l'utente interagisce con esso e come il sistema dovrebbe comportarsi.

I *requisiti non funzionali* rappresentano i vincoli e le proprietà di tipo realizzativo relative al sistema, come vincoli di natura temporale, che dovranno essere necessariamente rispettati.

3.2.1 Requisiti funzionali

Le funzionalità principali che vorremmo garantire sono le seguenti:

- *Attività CRUD per i tavoli*: si deve avere la possibilità di modificare il numero del tavolo associato a quel dispositivo.
- *Attività CRUD per l'ordine*: si deve avere la possibilità di creare e visionare l'ordine effettuato.
- *Attività CRUD per le pietanze*: si deve avere la possibilità di creare o cancellare i vari piatti prima di inviarli alla cucina.
- *Attività CRUD per la sessione*: si deve avere la possibilità di chiudere la sessione corrente ed aprirne una nuova.
- *Attività CRUD per l'utente*: si deve avere la possibilità di registrare e visualizzare le informazioni di un utente.
- *Attività CRUD per i punti cliente*: si deve avere la possibilità di visualizzare, inserire, modificare e cancellare i punti che l'utente ha a disposizione.
- *Richiesta aiuto e Invio messaggi al cameriere*: si deve avere la possibilità di richiedere aiuto per l'ordinazione ed inviare messaggi per notificare la volontà di procedere al pagamento.

3.2.2 Requisiti non funzionali

Visto che l'applicazione dovrà essere usata dai clienti del ristorante, abbiamo la necessità di renderla intuitiva e con un design accattivante. Inoltre, dovrà risultare fluida durante l'esecuzione.

I requisiti non funzionali sono i seguenti:

- *Manutenibilità*: si deve avere la possibilità di aggiungere o eliminare funzionalità all'applicazione senza compromettere il funzionamento delle funzionalità già esistenti.

- *Usabilità*: si deve implementare l'applicazione in modo da essere facile da comprendere e semplice da utilizzare per le varie categorie di utenza. Questa, infatti, dovrà essere utilizzata sia dai nativi digitali sia da utenti non esperti.
- *Durata sessione*: si deve implementare il concetto di sessione, la quale inizia dopo aver inserito il numero del tavolo e si conclude quando l'utente avrà completato il pagamento degli ordini effettuati.
- *Database online*: si deve utilizzare un database online in modo da permettere di visualizzare i dati dei clienti, gli ordini effettuati e i messaggi ricevuti su più dispositivi. Inoltre, si deve avere la possibilità di modificare il menù senza il bisogno di aggiornare l'app.

3.3 Attori e casi d'uso

La fase successiva porta all'individuazione degli attori e dei loro casi d'uso. Questi descrivono funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono con esso.

Gli *attori* vengono rappresentati nel diagramma tramite un'icona che rappresenta un uomo stilizzato. Questo identifica un ruolo, ovvero una famiglia di interazioni, coperto da un certo insieme di entità che interagiscono con essa.

I *casi d'uso* rappresentano una funzione o un servizio offerti dal sistema a uno o più attori. La funzione deve essere completa e significativa dal punto di vista degli attori che vi partecipano.

Gli attori che opereranno con l'applicazione sono i seguenti (Figura 3.1):

- *Utente non registrato*: colui che utilizzerà l'app per ordinare senza autenticarsi.
- *Utente registrato*: colui che, autenticandosi, oltre a poter ordinare, avrà la possibilità di ottenere sconti in base ai punti accumulati.
- *Cameriere/Staff*: colui che potrà accedere ad un'area riservata la quale gli permetterà di modificare il numero del tavolo e chiudere la sessione corrente indicando l'effettivo pagamento dell'utente.

Figura 3.1. Elenco degli attori

3.3.1 Diagrammi dei casi d'uso

I diagrammi dei casi d'uso sono usati per modellare il contesto e i requisiti funzionali di un sistema. Essi sono mostrati nelle Figure 3.2-3.4.

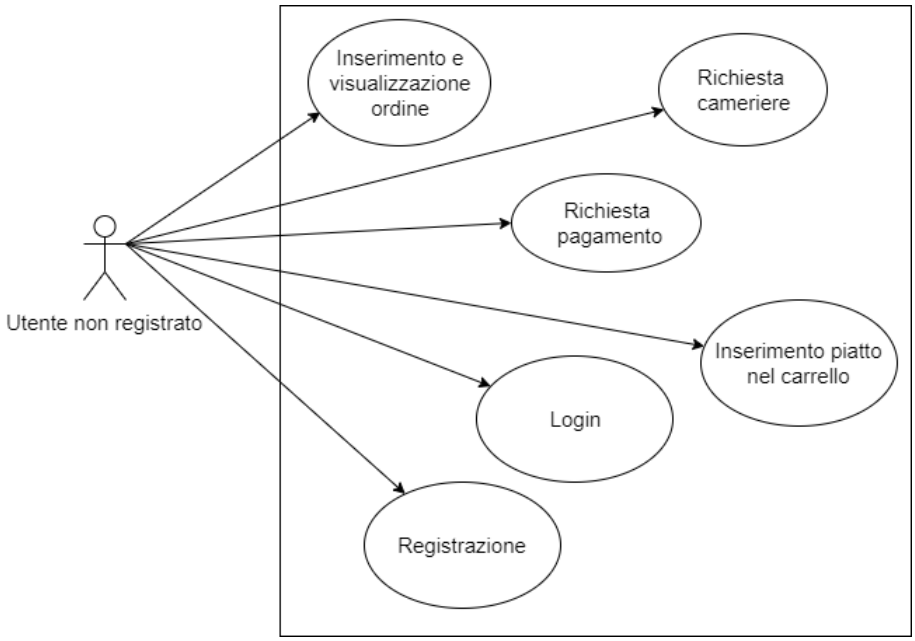


Figura 3.2. Diagramma dei casi d'uso relativo all'utente non registrato

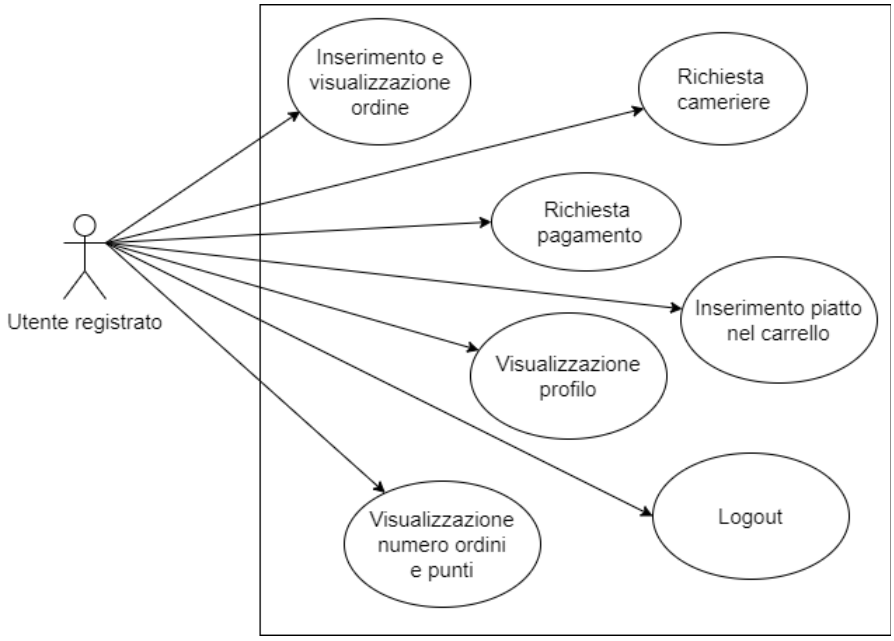


Figura 3.3. Diagramma dei casi d'uso relativo all'utente registrato

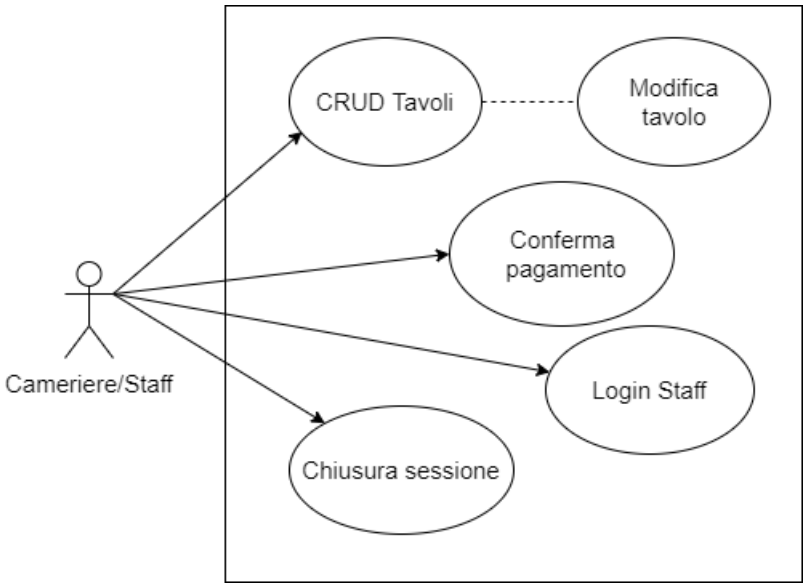


Figura 3.4. Diagramma dei casi d'uso relativo al cameriere/staff

3.3.2 Tabelle dei casi d'uso

In sintesi, i casi d'uso utilizzati nei diagrammi precedenti, che verranno utilizzati nell'applicazione, sono i seguenti:

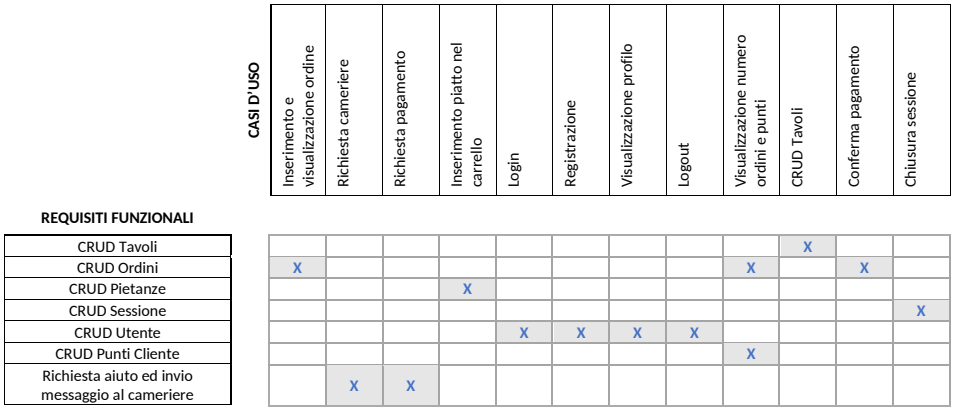
- *inserimento e visualizzazione di un ordine;*
- *richiesta di un cameriere;*
- *richiesta di un pagamento;*
- *inserimento di un piatto nel carrello;*
- *login;*
- *registrazione;*
- *visualizzazione di un profilo;*
- *logout;*
- *visualizzazione del numero degli ordini e dei punti;*
- *CRUD Tavoli;*
- *Modifica di un Tavolo;*
- *Conferma del pagamento;*
- *Login dello Staff;*
- *Chiusura di una sessione.*

3.4 Matrice di mapping

La *matrice di mapping* (o tracciabilità) collega i singoli requisiti ai casi d'uso così da garantire che le funzionalità e le caratteristiche del software siano verificate nella loro completezza e correttezza.

Una tale matrice permette di verificare il grado di copertura dei requisiti e consente di ottimizzare l'applicazione finale. Un requisito coperto da più casi d'uso, ad esempio, suggerisce di eliminare i duplicati che non aggiungono altri valori.

La matrice di mapping dei requisiti relativa all'applicazione viene riportata in Figura 3.5.



REQUISITI FUNZIONALI \ CASI D'USO	Inserimento e visualizzazione ordine	Richiesta cameriere	Richiesta pagamento	Inserimento piatto nel carrello	Login	Registrazione	Visualizzazione profilo	Logout	Visualizzazione numero ordini e punti	CRUD Tavoli	Conferma pagamento	Chiusura sessione
CRUD Tavoli										X		
CRUD Ordini	X								X		X	
CRUD Pietanze				X								
CRUD Sessione												X
CRUD Utente					X	X	X	X				
CRUD Punti Cliente									X			
Richiesta aiuto ed invio messaggio al cameriere		X	X									

Figura 3.5. Matrice di mapping dei requisiti dell'applicazione

4

Progettazione

In questo capitolo tratteremo la progettazione dell'applicazione. Partendo dalla mappa dell'applicazione, che ne illustra il contenuto, e passando per la realizzazione dei mockup, arriveremo alla progettazione concettuale. Dopo aver definito lo schema E/R affronteremo la progettazione logica per arrivare, infine, alla traduzione verso il modello relazionale.

4.1 Mappa dell'applicazione

La mappa dell'applicazione è una rappresentazione visuale della struttura dell'app. Seppur non strettamente necessaria, la comodità della mappa in fase di realizzazione è quella di poter far notare subito allo sviluppatore se l'applicazione è intuitiva e se i passaggi tra le varie fasi sono semplici per l'utente finale. Nella mappa vengono mostrate, tramite un insieme di frecce e blocchi, tutte le attività presenti nell'applicazione.

La Figura 4.1 mostra una schematizzazione delle varie fasi dell'applicazione all'interno di un unico diagramma. Tramite l'utilizzo dei colori, la mappa dell'app viene resa intuitiva e facile da comprendere. Con il colore giallo viene indicata la pagina principale, la quale viene visualizzata la prima volta che viene aperta l'applicazione e, nelle successive volte, quando verrà modificato il numero del tavolo. Tramite alcuni passaggi intermedi, rappresentati con il colore viola, si potrà visualizzare la home page, dalla quale sarà possibile spostarsi all'interno dell'app. Tramite ogni pulsante della navbar sarà possibile compiere diverse azioni, indicate dal colore arancione. Alcune attività verranno completate dopo alcuni passaggi intermedi, quando verrà raggiunto il blocco di colore blu.

Infine, si può notare la duplice funzionalità del blocco "Home", il quale, oltre ad essere una macroarea di riferimento, caratterizzata dal colore verde, svolge la funzione di home page, caratterizzata dal colore giallo. Questo avviene perchè il blocco "Imposta numero tavolo" non viene mostrato ogni volta che viene aperta l'app.

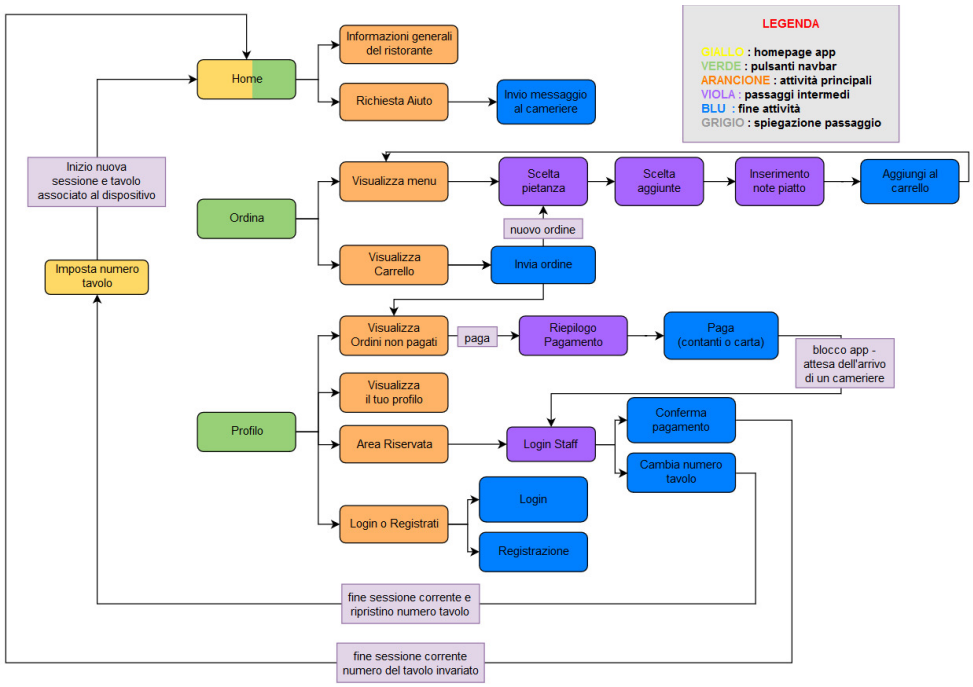


Figura 4.1. Mappa dell'applicazione

4.2 Realizzazione dei Mockup

I *mockup* sono una rappresentazione statica del prodotto finale con il più alto livello di dettaglio e fedeltà possibile. Essi servono a fornire un'idea del progetto finale, rappresentando nel dettaglio i vari contenuti e le funzionalità base dell'applicazione.

Nel nostro caso, verranno proposti i mockup relativi alle attività che si potranno trovare all'interno dell'app, cercando di riprodurre nel miglior modo possibile ciò che poi si andrà, successivamente, ad implementare. Tramite tali rappresentazioni grafiche stilizzate, si è in grado di capire quanto l'app risulti comprensibile e semplice da usare per l'utente finale.

Le Figure 5.1-4.5 mostrano le schermate principali dell'applicazione.

Le Figure 5.5-5.7 illustrano il percorso da effettuare per completare un'ordinazione.

Le Figure 5.9-4.11 mostrano il percorso da effettuare per completare e pagare un ordine.

Le Figure 4.12-4.14 illustrano le schermate tramite cui l'utente può registrarsi e che potrà visualizzare una volta autenticato.

La Figura 5.15 mostra l'Area Riservata, alla quale potranno accedere i camerieri e lo staff del locale.

La Figura 5.8 mostra il riepilogo di tutte le pietanze nel carrello, prima che l'ordine venga inviato alla cucina.

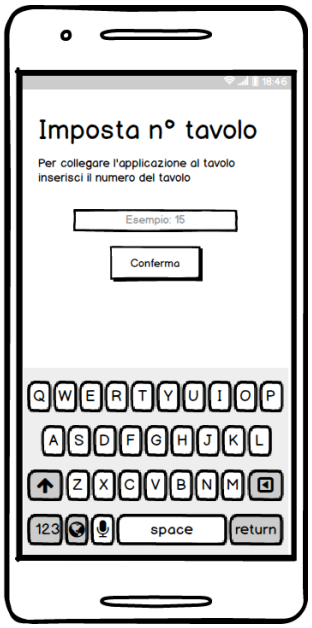


Figura 4.2. Mockup relativo alla scheda "Imposta Tavolo"

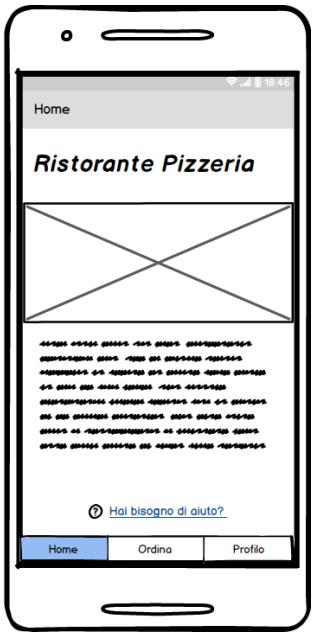


Figura 4.3. Mockup relativo alla scheda "Home"

Figura 4.4. Mockup relativo alla scheda "Ordina"

Figura 4.5. Mockup relativo alla scheda "Profilo"

Le Figure 5.18-5.19 rappresentano una variante rispettivamente della Figura 5.10 e della Figura 4.5, visibili soltanto quando l'utente avrà effettuato il login.

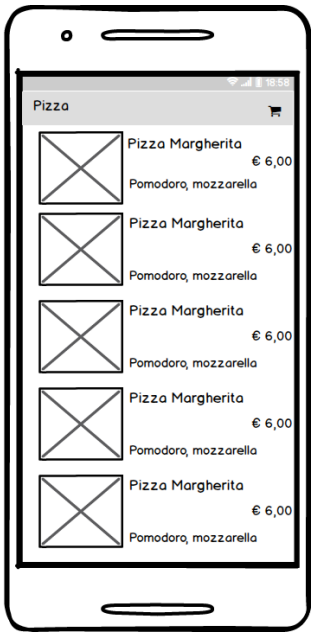


Figura 4.6. Mockup relativo alla scheda "Pietanza"

Figura 4.7. Mockup relativo alla scheda "Aggiunte"

Figura 4.8. Mockup relativo alla scheda "Note"

4.3 Progettazione concettuale

La *progettazione concettuale* ricopre un passaggio fondamentale all'interno dello sviluppo dell'applicazione. Il suo scopo è quello di rappresentare le specifiche informali della realtà di interesse in termini di una descrizione formale e completa, ma indipendente dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati.

In questa fase, infatti, si deve cercare di rappresentare il contenuto informativo della base di dati dell'applicazione, senza preoccuparsi nè delle modalità con le quali tali informazioni verranno codificate, nè dell'efficienza dei programmi che le utilizzeranno.

4.3.1 Diagramma Entità-Relazione

Il *modello Entità-Relazione* è un modello concettuale di dati che fornisce una serie di strutture per descrivere la realtà di interesse in una maniera facile da comprendere.

Il modello E/R ha tre costrutti principali:

- *Entità*: rappresentano classi di oggetti che hanno proprietà comuni ed esistenza autonoma ai fini della realtà di interesse. In uno schema E/R, ogni entità ha un nome che la identifica univocamente e viene rappresentata graficamente tramite un rettangolo.

Figura 4.9. Mockup relativo alla scheda "Comande"

Figura 4.10. Mockup relativo alla scheda "Pagamento"

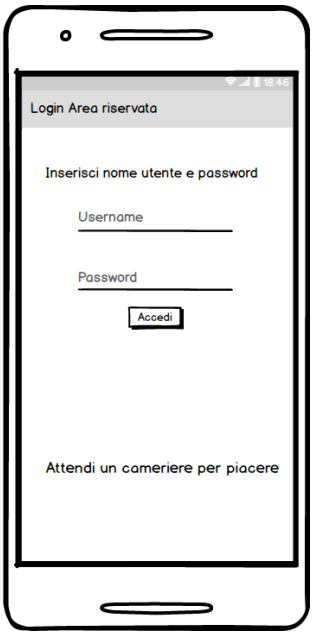


Figura 4.11. Mockup relativo alla scheda "Login Staff"

- *Relazioni o associazioni*: rappresentano legami logici, significativi per l'applicazione di interesse, tra due o più entità. In uno schema E/R, ogni relazione ha un nome che la identifica univocamente e viene rappresentata graficamente mediante un rombo.
- *Attributi*: descrivono le proprietà elementari di entità o relazioni che sono di interesse ai fini dell'applicazione. In uno schema E/R, ogni attributo ha un nome che lo identifica univocamente all'interno dell'oggetto che sta descrivendo.

In Figura 4.19 viene mostrato il diagramma E/R relativo alla nostra base di dati.

La base di dati, al proprio interno, dovrà memorizzare i dati relativi ai piatti che compongono il menù, come il nome, il prezzo, gli ingredienti, il nome dell'immagine, la categoria a cui appartiene il singolo piatto (pizze, primi, secondi, contorni, dolci o bevande) e la posizione in cui vogliamo che esso venga visualizzato all'interno del menù. Ogni pietanza è collegata alla lista delle proprie aggiunte caratterizzata da nome, prezzo e categoria. La cardinalità dell'entità *Voce Menù* è (0,N) in quanto non è obbligatorio che un piatto abbia delle aggiunte, mentre dal lato dell'entità *Aggiunta*, è (1,N) perchè tutte le aggiunte devono avere un piatto a loro associato.

All'interno dell'entità *Sessione tavolo* sono memorizzate tutte le sessioni utilizzate nell'applicazione, insieme al numero del tavolo al quale sono associate. Si è, anche, deciso di memorizzare l'orario di inizio e quello di fine, in modo da sapere se, in un preciso momento, una determinata sessione è terminata o è ancora in corso.

Figura 4.12. Mockup relativo alla scheda "Login Utente"

Figura 4.13. Mockup relativo alla scheda "Registrazione"

Figura 4.14. Mockup relativo alla scheda "Visualizza Informazioni"

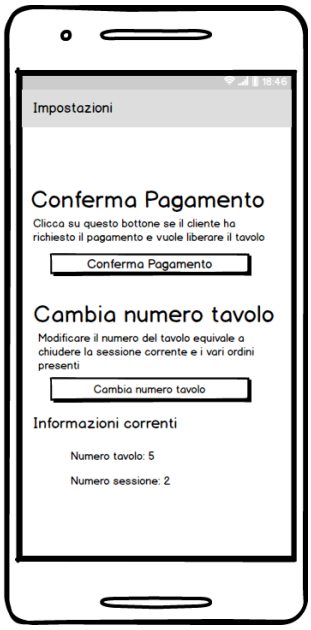


Figura 4.15. Mockup relativo alla scheda "Area Riservata"

Figura 4.16. Mockup relativo alla scheda "Carrello"

Figura 4.17. Mockup relativo alla scheda "Pagamento" con utente registrato

L'entità *Ordine* è collegata con l'entità *Sessione Tavolo* attraverso la relazione *Ordina*, e al suo interno contiene tutti gli ordini effettuati dagli utenti nell'ambito dell'applicazione. L'attributo *Pagato*, presente all'interno dell'entità, è un booleano

Figura 4.18. Mockup relativo alla scheda "Profilo" con utente registrato

che indicherà se il cliente ha pagato l'ordine o se lo pagherà in futuro. Si è deciso di introdurre la ridondanza *Costo Totale* in quanto è un attributo utilizzato molto spesso e sarebbe più dispendioso calcolarlo ogni volta. La cardinalità della relazione *Ordina*, dalla parte dell'entità *Sessione Tavolo*, è (0,N), in quanto una sessione può avere al proprio interno più ordini, mentre, dalla parte dell'entità *Ordine*, la cardinalità è (1,1), in quanto un ordine può essere associato ad un'unica sessione.

L'entità *Utente* è collegata all'entità *Ordine* tramite la relazione *Effettua*; tale relazione contiene, al proprio interno, i dati dell'utente, come e-mail, numero punti, numero ordini e ID. Si è deciso di introdurre la ridondanza *Numero Ordini* perchè sarebbe stato troppo dispendioso ricalcolare tale informazione ogni volta. La cardinalità della relazione *Effettua*, dalla parte dell'entità *Ordine*, è (1,1) in quanto un ordine può essere effettuato da un solo utente, mentre, dalla parte di *Utente*, la cardinalità è (0,N), perchè un utente può effettuare più di un ordine.

L'entità *Notifica* è collegata all'entità *Utente* tramite la relazione *Invia* e, al suo interno, contiene i messaggi inviati dagli utenti al cameriere. La relazione *Invia* ha cardinalità (0,N) dalla parte di *Utente*, in quanto un utente può inviare più di un messaggio, mentre è (1,1) dalla parte di *Notifica*, perchè una notifica è associata ad una sola istanza di *Utente*.

L'entità *Notifica* è collegata all'entità *Sessione Tavolo* tramite la relazione *Associa* e ha cardinalità (1,1) con quest'ultima, in quanto una notifica è collegata con un'unica sessione. Dalla parte opposta, *Sessione Tavolo* ha cardinalità (0,N), in quanto una sessione può inviare più di una notifica.

Infine, l'entità *Staff*, ha al proprio interno, gli attributi ID, nome e password, i quali contengono i dati dello staff del ristorante. Questa entità, rappresentando lo staff del locale, non comunica con il resto del diagramma E/R.

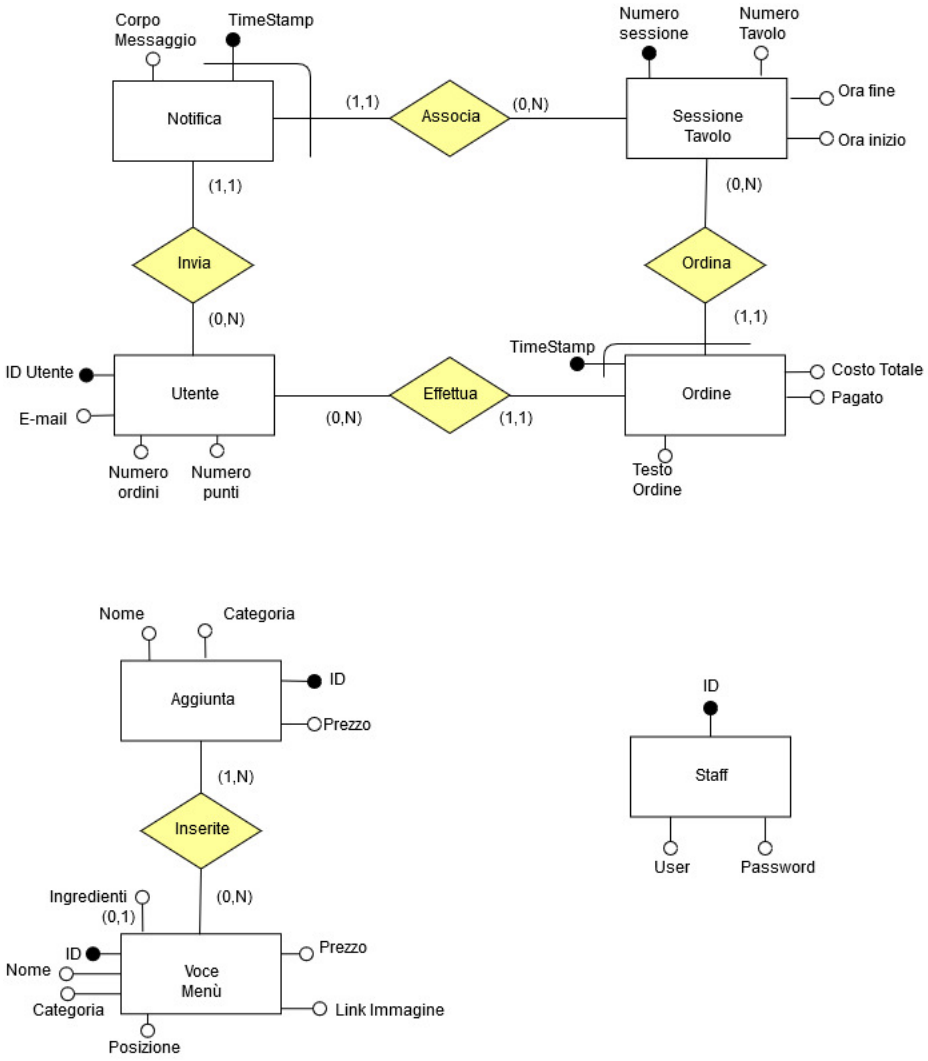


Figura 4.19. Diagramma Entità/Relazione relativo alla base di dati dell'applicazione

4.3.2 Dizionario delle entità e dizionario delle relazioni

Il *dizionario dei dati* serve per facilitare la comprensione di un diagramma E/R e per descrivere le proprietà dei dati che non possono essere espresse direttamente dai costrutti del modello. Il dizionario è composto da due parti:

- *Dizionario delle entità*: descrive le entità dello schema con il nome, una definizione informale in linguaggio naturale, l'elenco degli attributi e i possibili identificatori.
- *Dizionari delle relazioni*: descrive le relazioni con il nome, una loro descrizione informale, l'elenco degli attributi e l'elenco delle entità coinvolte insieme alla loro cardinalità.

Nelle Figure 4.20-4.21 sono rappresentati i dizionari della base di dati.

NOME ENTITÀ	DESCRIZIONE	ATTRIBUTI	IDENTIFICATORE
Aggiunta	Rappresenta l'insieme degli ingredienti che possono essere aggiunti ai piatti del menù	ID (numerico), Nome (stringa), Categoria (stringa), Prezzo (numerico)	ID (numerico)
Notifica	Raccolta dei messaggi inviati dall'utente al personale	TimeStamp (data), Corpo Messaggio (stringa)	TimeStamp (data), Numero Sessione di "Sessione Tavolo"
Ordine	Rappresenta l'ordine effettuato dal cliente	TimeStamp (data), Costo Totale (numerico), Pagato (0,1), Testo Ordine (stringa)	TimeStamp (data), Numero Sessione di "Sessione Tavolo"
Staff	Rappresenta il personale del locale	ID (numerico), User (stringa), Password (stringa)	ID (numerico)
Sessione Tavolo	Rappresenta la sessione effettiva del cliente collegata al numero del tavolo a cui fa riferimento	Numero Sessione (numerico), Numero Tavolo (numerico), Ora inizio (data), Ora fine (data)	Numero Sessione (numerico)
Utente	Rappresenta l'identità dell'utente che ha effettuato la registrazione	ID Utente (stringa), E-mail (stringa), Numero Ordini (numerico), Numero Punti (numerico)	ID Utente (stringa)
Voce Menù	Rappresenta i piatti che il cliente può ordinare	ID (numerico), Nome (stringa), Ingredienti (stringa), Categoria (stringa), Posizione (numerico), Prezzo (numerico), Link Immagine (stringa)	ID (numerico)

Figura 4.20. Dizionario delle Entità relativo alla base di dati dell'applicazione

4.3.3 Vincoli di integrità

Per evitare situazioni che potrebbero portare in errore, vengono introdotti i *vincoli di integrità*; questi denotano proprietà che devono essere soddisfatte dalle istanze che rappresentano informazioni corrette per l'applicazione. In generale, a uno schema di base di dati vengono associati un insieme di vincoli e vengono considerate corrette le istanze che soddisfano tutti i vincoli.

I vincoli di integrità che la nostra applicazione deve soddisfare sono i seguenti:

- *V1*: il campo "Prezzo", relativo all'entità "Voce Menù", deve essere un numero maggiore di 0.

NOME RELATIONSHIP	DESCRIZIONE	ENTITÀ COINVOLTE	ATTRIBUTI
Associa	Associa le notifiche con la sessione che le ha inviate	Notifica (1,1), Sessione Tavolo (0,N)	\\
Effettua	Associa l'ordine all'utente che lo ha richiesto	Ordine (1,1), Utente (0,N)	\\
Inserite	Associa le aggiunte possibili ai piatti del menù	Aggiunta (1,N), Voce Menù (0,N)	\\
Invia	Associa le notifiche all'utente che le ha inviate	Notifica (1,1), Utente (0,N)	\\
Ordina	Associa l'ordine alla sessione che lo ha effettuato	Ordine (1,1), Sessione Tavolo (0,N)	\\

Figura 4.21. Dizionario delle Relazioni relativo alla base di dati dell'applicazione

- *V2*: il campo "Prezzo", relativo all'entità "Aggiunta", deve essere un numero maggiore di 0.
- *V3*: il campo "Costo Totale", relativo all'entità "Ordine", deve essere maggiore di 0.
- *V4*: l'attributo "Pagato", relativo all'entità "Ordine", deve valere 0 (non pagato) o 1 (pagato).
- *V5*: l'attributo "Posizione", relativo all'entità "Voce Menù", deve essere un numero maggiore di 0.
- *V6*: l'attributo "Categoria", relativo all'entità "Voce Menù", e all'entità "Aggiunte" deve essere "Pizze", "Primi", "Secondi", "Contorni", "Dolci" e "Bevande".
- *V7*: l'attributo "Ingredienti", relativo all'entità "Voce Menù", è opzionale.
- *V8*: l'attributo "Numero Sessione", relativo all'entità "Sessione Tavolo", deve essere un numero maggiore di 0.
- *V9*: l'attributo "Numero Tavolo", relativo all'entità "Sessione Tavolo", deve essere un numero maggiore di 0
- *V10*: l'attributo "Ora fine", relativo all'entità "Sessione Tavolo", deve essere maggiore dell'attributo "Ora inizio".
- *V11*: l'attributo "E-mail", relativo all'entità "Utente", deve essere univoco.
- *V12*: l'attributo "Numero ordini", relativo all'entità "Utente", deve essere maggiore o uguale a 0.
- *V13*: l'attributo "Numero punti", relativo all'entità "Utente", deve essere maggiore o uguale a 0.

4.3.4 Elenco degli identificatori principali

La scelta degli *identificatori principali* è essenziale nella traduzione verso il modello relazionale perchè in questo modello le chiavi vengono usate per stabilire legami tra dati presenti in relazioni diverse. Inoltre, i sistemi di gestione di base di dati,

richiedono generalmente di specificare una chiave primaria sulla quale vengono poi costruite delle strutture per il reperimento efficiente dei dati.

Nella Figura 4.22 vengono mostrati gli identificatori principali della base di dati.

NOME ENTITÀ	IDENTIFICATORE
Aggiunta	ID
Notifica	TimeStamp, Numero Sessione
Ordine	TimeStamp, ID Utente
Sessione Tavolo	Numero Sessione
Staff	ID

Figura 4.22. Elenco degli identificatori principali relativo alla base di dati dell'applicazione

4.3.5 Glossario dei termini

Il *glossario dei termini* è molto utile per la comprensione e la precisazione di alcuni termini usati. Al proprio interno, per ogni termine, deve contenere una breve descrizione e dei possibili sinonimi. In questo modo, l'utente finale sarà in grado di leggere correttamente la relazione e il diagramma E/R, senza incappare in equivoci dettati dalla poca chiarezza del contenuto.

In Figura 4.23 è presente il glossario dei termini relativo alla base di dati dell'applicazione.

4.4 Progettazione logica

L'obiettivo della *progettazione logica* è quello di costruire uno schema logico in grado di descrivere, in maniera corretta ed efficiente tutte le informazioni contenute nel diagramma E/R prodotto durante la *progettazione concettuale*. L'efficienza è legata alle prestazioni, anche se non sono sempre valutabili precisamente a livello concettuale e logico.

4.4.1 Traduzione verso il modello relazionale

Una delle fasi della progettazione logica corrisponde a una traduzione tra modelli differenti: a partire da uno schema E/R si costruisce uno schema logico equivalente, ovvero uno schema in grado di rappresentare le stesse informazioni.

TERMINE	DESCRIZIONE	SINONIMO
Categoria	Rappresenta la categoria del piatto e può essere “Pizze”, “Primi”, “Secondi”, “Contorni”, “Dolci”, “Bevande”.	\\
Cliente	Rappresenta l’utente registrato o non registrato, che può effettuare le ordinazioni al tavolo	\\
Corpo del Messaggio	Rappresenta il testo del messaggio inviato automaticamente dall’utente quando effettua delle interazioni, come la richiesta di un cameriere.	\\
Pagato	Rappresenta un booleano (0,1) che indica se il cliente ha pagato gli ordini immediatamente, oppure se pagherà in un diverso momento.	\\
Posizione	È un intero che rappresenta la posizione che avranno i piatti all’interno della lista dei piatti del menù.	\\
Staff	Rappresenta il personale del ristorante, il quale si occupa di settare l’applicazione e sbloccarla quando il cliente ha pagato il conto.	Cameriere, Personale
Testo Ordine	Rappresenta il testo di tutto l’ordine con i nomi dei piatti, i prezzi, le aggiunte scelte e le note.	\\
Utente	Rappresenta l’identità della persona che ha effettuato la registrazione.	\\

Figura 4.23. Glossario dei termini relativo alla base di dati dell’applicazione

In Figura 4.24 è presente la traduzione verso il modello relazionale della base di dati. In tale traduzione si è scelto di non creare delle tabelle per le relazioni e di accorpare il loro contenuto nelle entità, per rendere la base di dati più efficiente.

4.5 Diagramma delle attività

I *diagrammi delle attività*, o *flow chart*, in ambito informatico, sono una rappresentazione grafica delle operazioni da eseguire per l’esecuzione di una determinata azione. Al loro interno, sono presenti un punto di ingresso e un punto di uscita, importanti per capire dove inizia e dove finisce l’azione da effettuare. Infine, le frecce all’interno dei diagrammi sono unidirezionali, in modo da indicare in maniera chiara la direzione che si dovrà seguire. Nelle Figure 4.25-4.27 sono rappresentati i diagrammi delle attività relativi alle seguenti azioni:

- *aggiunta di un piatto nel carrello*;
- *pagamento di un ordine*;
- *cambio del numero del tavolo.*

ENTITÀ/RELATIONSHIP	TRADUZIONE	VINCOLI DI RIFERIMENTO
Aggiunta	AGGIUNTE (ID, Nome, Categoria, Prezzo)	\\
Notifica	MESSAGGI (TimeStamp, Sessione, Tavolo, CorpoMessaggio, E-mail)	Sessione → SessioneTavolo.Sessione Tavolo → SessioneTavolo.Tavolo E-mail → Utente.Email
Ordine	ORDINI (TimeStamp, Sessione, Tavolo, IDUtente, TestoOrdine, Totale, Pagato)	Sessione → SessioneTavolo.Sessione IDUtente → Utente.IDUtente
Sessione Tavolo	SESSIONE (Sessione, Tavolo, DataInizio, DataFine)	\\
Staff	STAFF (ID, User, Password)	\\
Utente	UTENTE (IDUtente, E-mail, Ordini, Punti)	
Voce Menù	MENÙ (ID, Nome, Categoria, Posizione, Ingredienti, Prezzo, LinkImmagine)	\\

Figura 4.24. Traduzione verso il modello relazionale relativa alla base di dati dell'applicazione

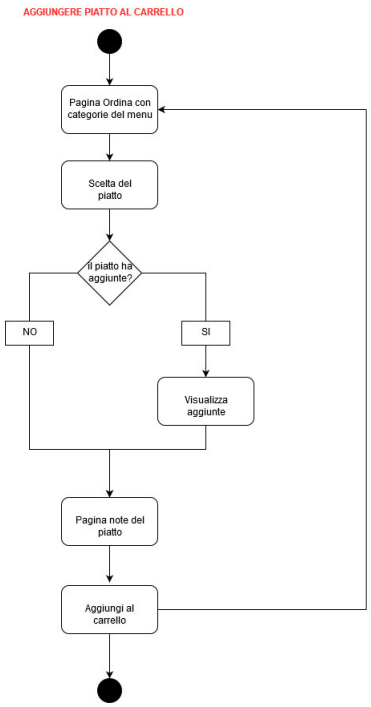


Figura 4.25. Diagramma delle attività relativo all'aggiunta di un piatto nel carrello

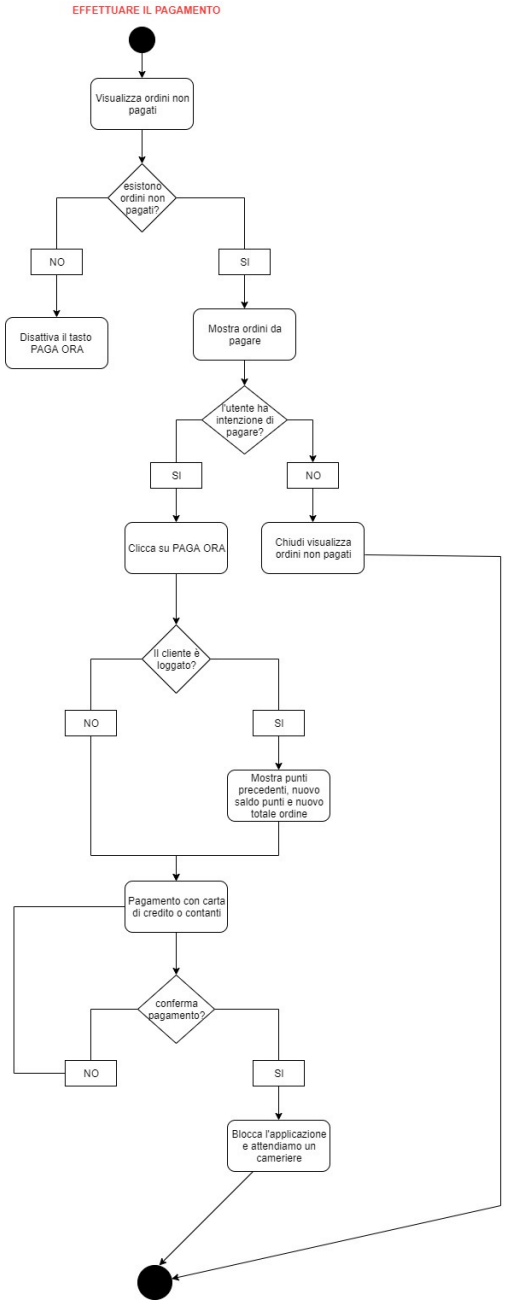


Figura 4.26. Diagramma delle attività relativo al pagamento di un ordine

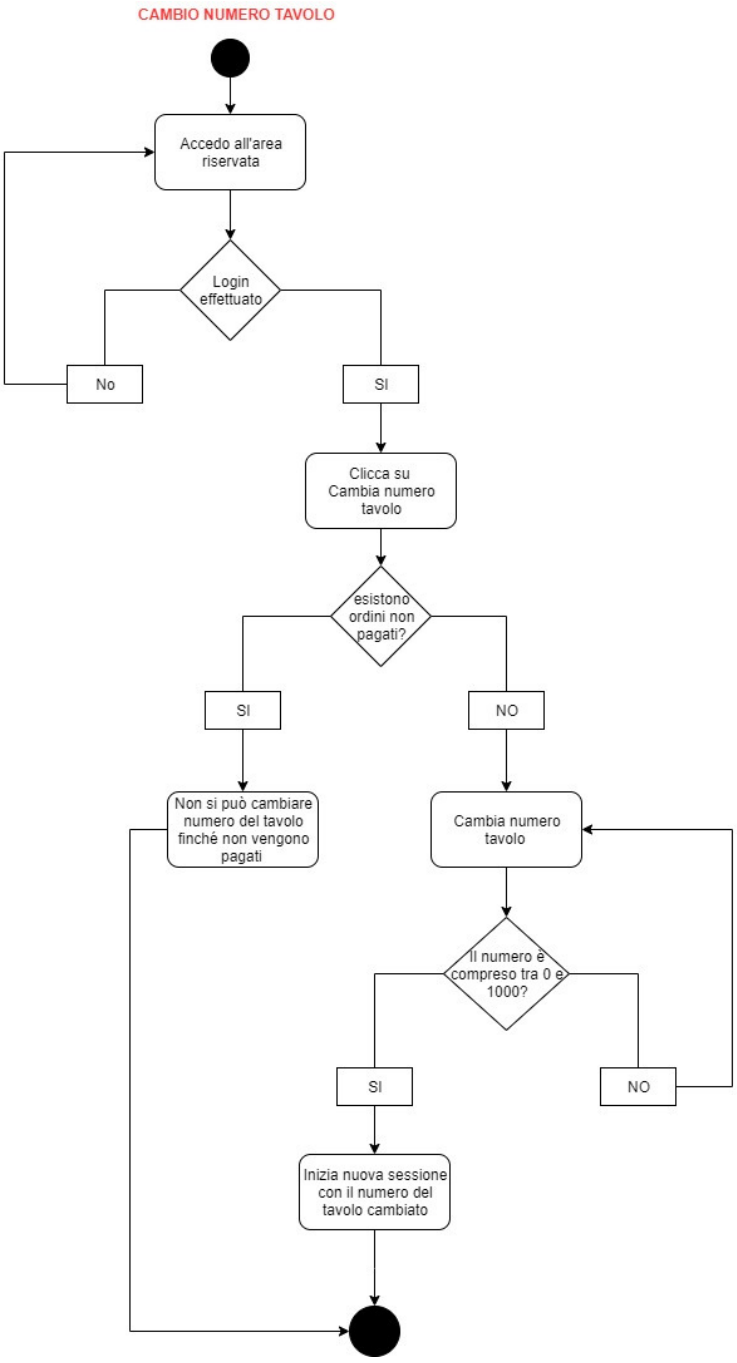


Figura 4.27. Diagramma delle attività relativo al cambio del numero del tavolo

5

Implementazione

In questo capitolo descriveremo l'interfaccia dell'applicazione tramite alcuni screenshot. Successivamente, presenteremo Nuget, Firebase e i pacchetti utilizzati all'interno dell'app. Infine, descriveremo le principali funzionalità implementate nell'applicazione.

5.1 Interfaccia dell'applicazione

La prima volta che viene aperta l'applicazione verrà visualizzata la schermata mostrata in Figura 5.1. Tramite questa pagina, il cameriere potrà configurare l'applicazione inserendo il numero del tavolo, in modo da collegare i futuri ordini dell'utente al tavolo dal quale vengono eseguiti.

L'utente, quando inizierà ad utilizzare l'applicazione, troverà davanti a sè la schermata *Home* (Figura 5.2), visualizzabile soltanto dopo che il cameriere ha configurato l'applicazione. In essa, saranno presenti le informazioni generali del ristorante e il pulsante *Hai bisogno di aiuto?*, utilizzabile per chiamare un cameriere al proprio tavolo.

Tramite la *bottom bar* l'utente potrà navigare attraverso le pagine principali dell'applicazione (*Home*, *Ordina* e *Profilo*). Attraverso la pagina *Profilo* (Figure 5.4 e 5.19), egli avrà la possibilità di visualizzare diverse informazioni dopo essersi registrato (Figura 5.13) o aver effettuato il login (Figura 5.12). L'utente troverà il numero dei punti accumulati e il numero degli ordini effettuati nella sezione *I tuoi dati* e le credenziali di accesso nella sezione *Visualizza il tuo profilo* (Figura 5.14). In più, potrà vedere se sono presenti ordini da pagare nella sezione *Visualizza ordini non pagati* (Figura 5.9).

Attraverso la pagina *Ordina*, mostrata in Figura 5.3, l'utente avrà la possibilità di ordinare il piatto che preferisce, scegliendo prima la categoria e, successivamente, la pietanza (Figura 5.5). Dopo aver selezionato il piatto, egli avrà la possibilità di inserire delle aggiunte (Figura 5.6) e, in caso di allergie o intolleranze particolari, potrà aggiungere delle note da comunicare alla cucina (Figura 5.7).

La pietanza selezionata, dopo aver completato l'ordine, sarà disponibile nel carrello (Figura 5.8). In questa schermata l'utente potrà eliminare un singolo piatto, tramite l'icona del cestino accanto alla pietanza, o svuotare il carrello, tramite il

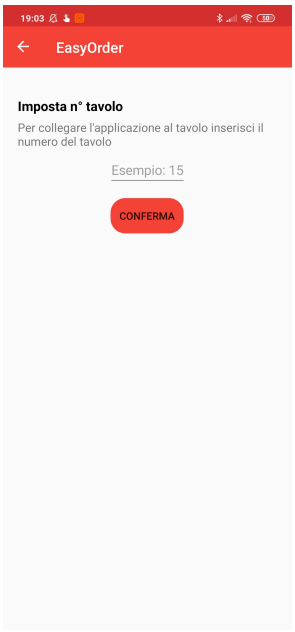


Figura 5.1. Realizzazione della pagina *Imposta Tavolo*

Figura 5.2. Realizzazione della pagina *Home*

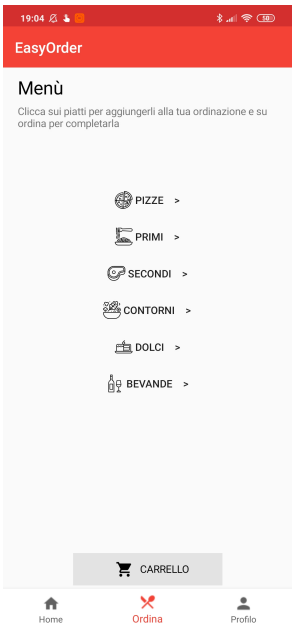


Figura 5.3. Realizzazione della pagina *Ordina*

Figura 5.4. Realizzazione della pagina *Profilo*

pulsante *Cancella carrello*. Infine, una volta completato l'ordine, potrà inviarlo alla

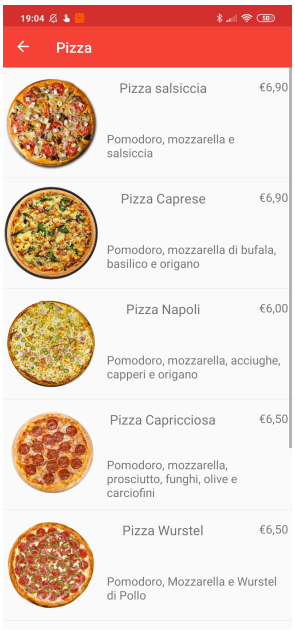


Figura 5.5. Realizzazione della pagina *Pietanza*

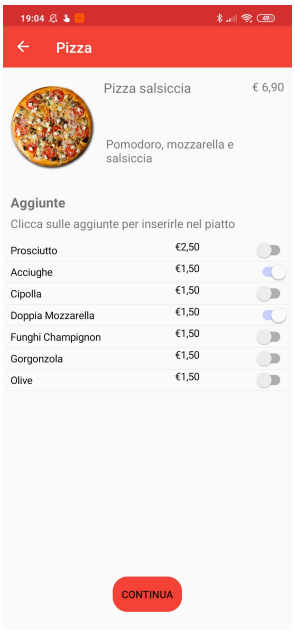


Figura 5.6. Realizzazione della pagina *Aggiunte*

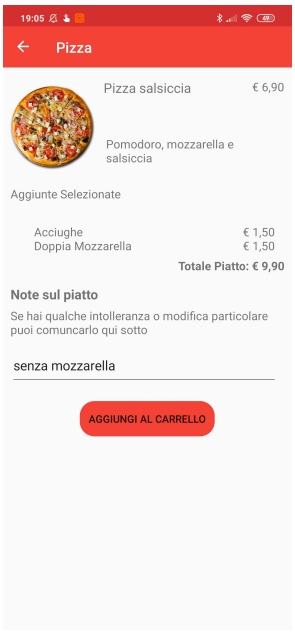


Figura 5.7. Realizzazione della pagina *Note*

cucina, tramite il pulsante *Invia ordine*.

Tutti gli ordini effettuati sono visualizzabili all'interno della pagina *Comande* (Figura 5.9). L'applicazione permette all'utente di compiere più di un ordine prima di effettuare il pagamento. Una volta terminato il pasto, egli potrà cliccare sul pulsante *Paga ora* per saldare il conto.

L'utente, attraverso la procedura di pagamento, avrà la possibilità di scegliere il metodo di pagamento (Figura 5.10). Se effettua il login, avrà in più la possibilità di usufruire di particolari sconti, tramite i punti guadagnati in precedenza (Figura 5.18).

Dopo aver deciso se pagare in contanti o con la carta, l'applicazione rimarrà bloccata nella schermata presente in Figura 5.11, fino all'arrivo di un cameriere. Quest'ultimo, accedendo nell'Area Riservata (Figura 5.15), attraverso la pagina *Impostazioni Staff* (Figura 5.16), avrà la possibilità di confermare il pagamento dell'utente, indicando se questo ha pagato o no (Figura 5.17). Infatti, l'utente potrebbe decidere di saldare il conto successivamente, o abbandonare il locale senza pagare. Anche in questo caso si deve avere la possibilità di sbloccare l'applicazione.

Dopo aver confermato il pagamento, l'applicazione verrà resettata incrementando il numero della sessione. Attraverso l'*Area Riservata*, il cameriere avrà, anche, la possibilità di cambiare il numero del tavolo e potrà visualizzare il numero del tavolo e la sessione collegati attualmente al dispositivo.

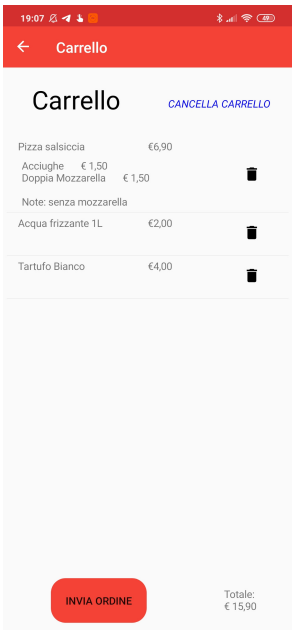


Figura 5.8. Realizzazione della pagina *Carrello*

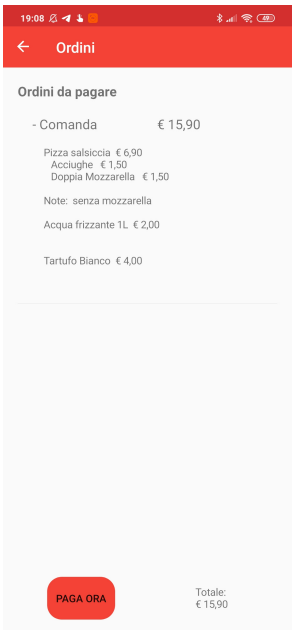


Figura 5.9. Realizzazione della pagina *Comande*

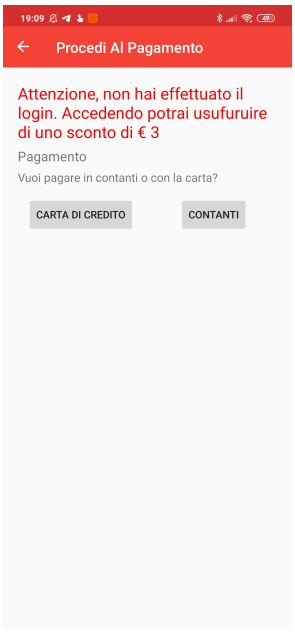


Figura 5.10. Realizzazione della pagina *Pagamento*

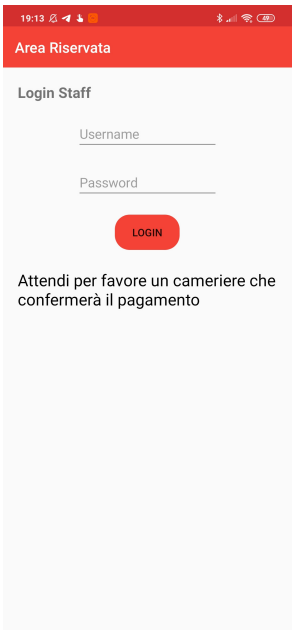


Figura 5.11. Realizzazione della pagina *Login Staff*

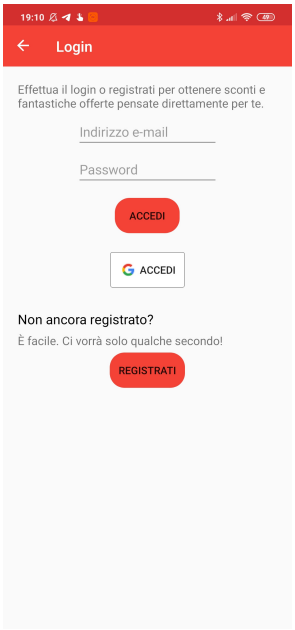


Figura 5.12. Realizzazione della pagina *Login Utente*

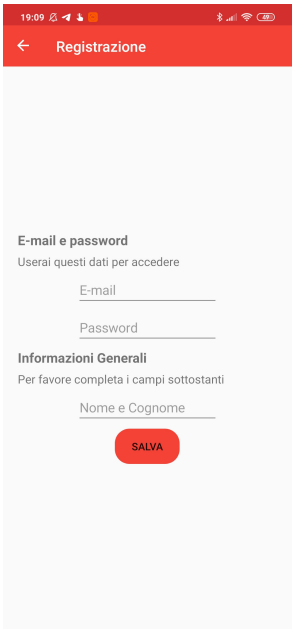


Figura 5.13. Realizzazione della pagina *Registrazione*

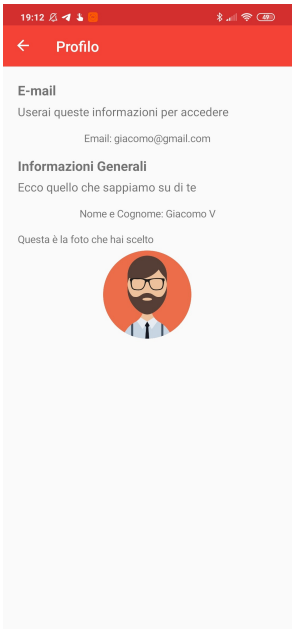


Figura 5.14. Realizzazione della pagina *Visualizza Informazioni*

5.2 Implementazione pacchetti NuGet

Attraverso l'implementazione dei pacchetti NuGet in Xamarin, è possibile utilizzare parti di codice già compilate da altri sviluppatori. Includendoli all'interno di un progetto, e richiamando le loro funzioni in caso di necessità, si può evitare di scrivere ulteriori righe di codice, oltre quelle essenziali. Questi pacchetti possono provenire dalla raccolta pubblica di NuGet o da un'organizzazione di terze parti.

5.2.1 Cos'è NuGet

NuGet è il meccanismo supportato da Microsoft per la condivisione del codice. Esso definisce in che modo vengono creati e utilizzati i pacchetti per *.NET* e fornisce gli strumenti per ciascuno di essi.

Un pacchetto NuGet è un singolo file *ZIP* che contiene codice compilato, alcuni file correlati al codice e un manifesto descrittivo, che include informazioni come il numero di versione del pacchetto.

Un *host*, pubblico o privato che sia, funge da punto di connessione fra gli autori e i consumatori dei pacchetti NuGet. Gli autori compilano i pacchetti e li pubblicano in un host, mentre i consumatori li cercano negli stessi host, scaricandoli e includendoli nei loro progetti. Una volta installate in un progetto, le API dei pacchetti sono disponibili per il resto del codice del progetto (Figura 5.20).

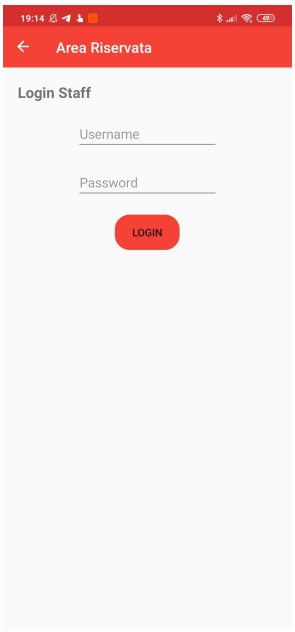


Figura 5.15. Realizzazione della pagina *Area Riservata*

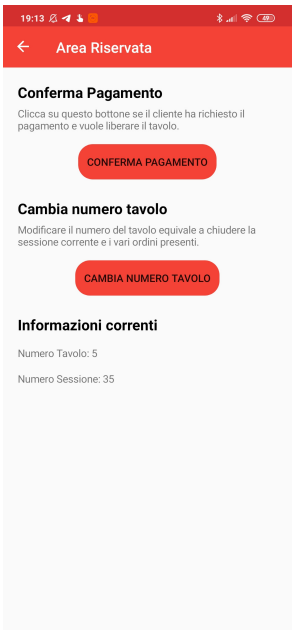


Figura 5.16. Realizzazione della pagina *Impostazioni Staff*

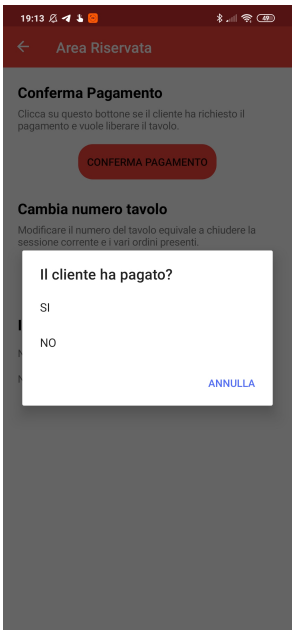


Figura 5.17. Pop-up che permette di impostare il pagamento del cliente

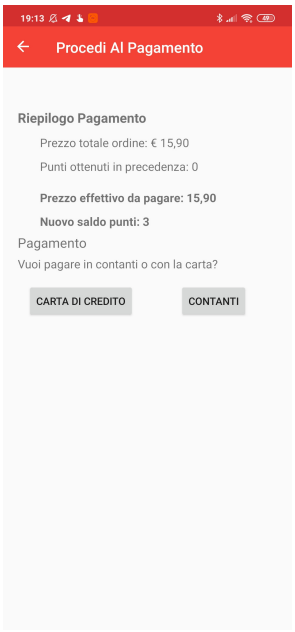


Figura 5.18. Realizzazione della pagina *Pagamento* con utente registrato

Figura 5.19. Realizzazione della pagina *Profilo* con utente registrato

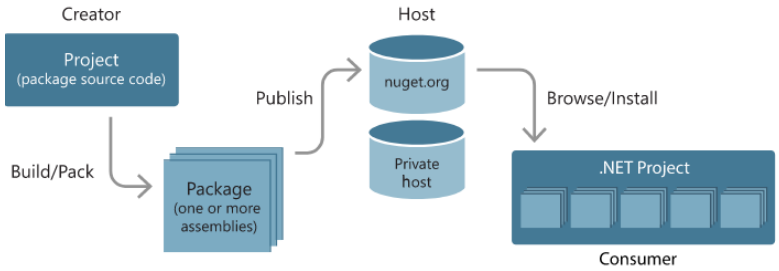


Figura 5.20. Realizzazione di un pacchetto NuGet

Tramite i pacchetti *NuGet*, è possibile riutilizzare facilmente il lavoro di altri utenti. Inoltre, il consumatore dei pacchetti dovrà preoccuparsi soltanto dei pacchetti utilizzati direttamente in un progetto. NuGet stesso si occupa di tutte le dipendenze a livello inferiore.

Invece di associare direttamente gli assembly binari ad un progetto, NuGet gestisce un semplice elenco di riferimento dei pacchetti da cui dipende un progetto. In questo modo, il progetto occupa meno spazio nella memoria del computer e, se questo viene spostato su un'altra macchina, i pacchetti corrispondenti possono essere recuperati tramite il loro codice identificativo.

5.2.2 Firebase Database

Firebase è una piattaforma di Google che permette di costruire con facilità applicazioni, sia web che mobile, grazie agli strumenti e alle API che fornisce. Esso

supporta i principali linguaggi di programmazione, come *Swift*, *Objective-C*, *Java*, *JavaScript*, *C++*, etc.

Firebase Database è un *Database Realtime* che permette di sincronizzare e archiviare dati in tempo reale in tutte le applicazioni connesse. Realtime Database salva i dati all'interno di un albero *JSON* con una coppia chiave-valore. Esso usa la tecnologia *WebSocket* per la sincronizzazione, così che tutte le applicazioni possano ricevere aggiornamenti in tempo reale quando i dati vengono modificati nel server, e viceversa. Esso provvede, anche, al supporto offline, ovvero il client può mostrare lo stesso i dati anche quando non è presente una connessione a Internet, e sincronizzare di nuovo il tutto, quando la connessione torna ad essere presente.

In Figura 5.21 vengono mostrati i principali nodi che compongono il nostro database. Si è deciso di convertire uno ad uno le varie tabelle della progettazione logica e concettuale della Sezione 4.3.

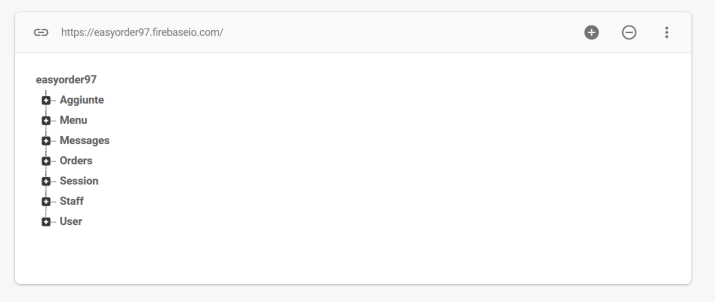


Figura 5.21. Realtime Database di Firebase

Per utilizzare Firebase Database è stato necessario implementare, all'interno del progetto di Visual Studio, i pacchetti NuGet presenti in Figura 5.22. Tramite questi pacchetti, l'implementazione del servizio di Firebase è risultato quello del Listato 5.1, il quale mostra, come esempio, la funzione `GetAllMenu`, che richiede come parametri la categoria della quale si vogliono visualizzare le pietanze. Una volta indicata la categoria, viene creata una lista di oggetti Menù (Listato 5.2) di tutti i piatti presenti in Firebase Database. Uno alla volta, tutti i figli della chiave Menù vengono letti e inseriti nella lista che si sta creando dinamicamente, tramite le funzioni `Select` e `Where`.

```
1          public async Task<List<Objects.Menu>> GetAllMenu(string CategorySel)
2          {
3              return (await firebase
4                .Child("Menu")
5                .OnceAsync<Objects.Menu>()).Select(item => new Objects.Menu
6                {
7                    Key = item.Key,
8                    Category = item.Object.Category,
9                    Ingredients = item.Object.Ingredients,
10                   LinkImage = item.Object.LinkImage,
11                   Name = item.Object.Name,
12                   Position = item.Object.Position,
13                   Price = item.Object.Price
14               }).Where(a => a.Category == CategorySel).ToList();
15         }
```

Listato 5.1. Definizione della funzione `GetAllMenu`

Figura 5.22. Pacchetti NuGet necessari per utilizzare Firebase Database

```
1   public class Menu
2       {
3           public string Key { get; set; }
4           public string Category { get; set; }
5           public string LinkImage { get; set; }
6           public string Name { get; set; }
7           public int Position { get; set; }
8           public double Price { get; set; }
9           public string Ingredients { get; set; }
10      }
```

Listato 5.2. Definizione dell'oggetto `Menù`

Infine, per completezza, nella Figura 5.23 viene mostrato l'esempio di un nodo di tipo Menù. Esso è composto da un nodo interno che rappresenta l'ID del singolo piatto, che ha come figli tutti i valori che deve avere un elemento di tipo Menù; ovvero la categoria, l'immagine, il nome, la posizione e il prezzo.

Figura 5.23. Esempio di nodo contenente le informazioni di un oggetto di tipo Menù

5.2.3 Firebase Auth

Firebase Auth è un altro importante componente di Firebase, che gestisce l'autenticazione degli utenti. Esso permette di memorizzare le informazioni degli utenti

in tutta sicurezza e in modo facile e veloce. Esso gestisce autenticazioni tramite password, numero di telefono o tramite i *federated identity provider*, come Google, Facebook e Twitter.

Per far registrare un utente all'applicazione, è necessario che quest'ultimo fornisca le sue credenziali per autenticarsi. Tali credenziali possono essere l'e-mail e la password dell'utente, oppure un *token OAuth* fornito dai *federated identity provider*. Successivamente, le credenziali vengono inviate al *Firebase Authentication SDK* che, tramite alcuni servizi di back-end, verifica se sono corrette e fornisce una risposta al client.

Dopo una corretta registrazione tramite Firebase, è possibile visualizzare le informazioni di base dell'utente nonchè controllare i suoi accessi tramite la console. In più, può essere usato il *provided authentication token* dell'utente per verificare la sua identità tramite i servizi backend.

Questa componente di Firebase può essere utilizzata nel progetto di Visual Studio utilizzando i pacchetti NuGet presenti in Figura 5.24.

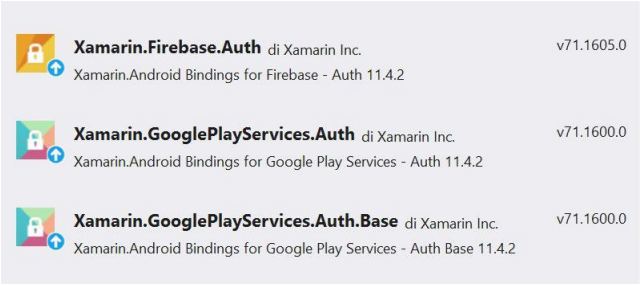


Figura 5.24. Pacchetti NuGet necessari per utilizzare Firebase Auth

Nel Listato 5.3 viene parzialmente mostrata la classe `AuthDroid`, che permette all'utente di registrarsi ed effettuare il login tramite e-mail e password. Tutte le funzioni che comunicano con Firebase devono essere asincrone, in modo da eseguire l'azione in background ed evitare che l'app rimanga bloccata per troppo tempo, rischiando di essere uccisa dal sistema operativo.

La funzione `RegisterWithEmailPassword` richiede come parametri l'e-mail, la password e il nome completo; questi dati vengono inseriti dall'utente al momento della registrazione. Tramite le funzioni fornite dai pacchetti NuGet siamo in grado facilmente di creare un nuovo utente con l'e-mail e la password fornite tramite la funzione `CreateUserWithEmailPasswordAsync`, a cui, poi, colleghiamo il nome completo e un'immagine del profilo. Per fare ciò creiamo un oggetto di tipo `UserProfileChangeRequest` in cui inseriremo i dati dell'utente. Una volta costruito l'oggetto tramite la funzione `AddOnCompleteListener`, che si occupa di creare un listener `OnCompleteListener`, colleghiamo l'oggetto creato all'oggetto `Utente` contenente i dati dell'utente che si è appena registrato. Se l'operazione va a buon fine, l'utente riesce a registrarsi; altrimenti viene visualizzato un messaggio di errore che, nella console di Visual Studio, mostra il problema della mancata registrazione.

La funzione `LoginWithEmailPassword` permette, invece, tramite e-mail e password forniti dall'utente, di effettuare il login all'interno dell'applicazione. Qui viene

creato un oggetto `user` che, mediante e-mail e password dell'utente e tramite la funzione `SignInWithEmailPasswordAsync`, ricava un *token Auth*. Se i dati sono validi viene restituito il token dell'utente che svolgerà correttamente il login.

```
1   public class AuthDroid : IAuth
2       {
3           public async Task<bool> RegisterWithEmailPassword
4           (string email, string password, string fullname)
5           {
6               try {
7                   var user = await FirebaseAuth.GetInstance(MainActivity.app)
8                   .CreateUserWithEmailAndPasswordAsync(email, password);
9                   FirebaseUser Utente =
10                   FirebaseAuth.GetInstance(MainActivity.app).CurrentUser;
11                  UserProfileChangeRequest profileUpdates;
12                  profileUpdates = new UserProfileChangeRequest.Builder()
13                          .SetDisplayName(fullname)
14                          .SetPhotoUri(Android.Net.Uri.Parse(ProfileImageURL))
15                          .Build();
16                  Utente.UpdateProfile(profileUpdates)
17                                  .AddOnCompleteListener(new OnCompleteListener());
18                  return true;
19              }
20              catch (Exception e){
21                  Console.WriteLine("Messaggio di errore: " + e.Message);
22                  return false;
23              }
24          }
25
26          public async Task<string> LoginWithEmailPassword
27          (string email, string password)
28          {
29              try {
30                  var user = await FirebaseAuth.GetInstance(MainActivity.app)
31                  .SignInWithEmailAndPasswordAsync(email, password);
32                  var token = await user.User.GetIdTokenAsync(false);
33                  return token.Token;
34              }
35              catch (FirebaseAuthInvalidUserException e) {
36                  e.PrintStackTrace();
37              }
38          }
39          }
```

Listato 5.3. Definizione della classe `AuthDroid`

5.3 Implementazione delle funzionalità principali

In questa sezione vengono riportate alcune delle funzionalità implementate durante lo sviluppo dell'applicazione. Esse vengono divise nelle diverse sottosezioni.

5.3.1 Ordinazione

La prima soluzione che analizzeremo sarà quella utilizzata per implementare il menù, e, di conseguenza, la suddivisione delle categorie, la visualizzazione delle pietanze, delle aggiunte e, infine, delle note.

La prima classe che prenderemo in esame è la classe `Ordina` (Listato 5.4). È proprio il codice contenuto in questa classe che implementa quanto raffigurato in Figura 5.3. La classe `Ordina` a seconda della categoria selezionata, passerà alla classe `ListaPiatti` una diversa stringa contenente il nome della categoria e farà in modo che venga visualizzata dall'utente l'activity successiva.

```
1   public partial class Ordina : ContentPage
2           {
3           public Ordina() {
4               InitializeComponent();
```

```
5           }
6
7           async void Pizze_Clicked(object sender, EventArgs e) {
8               ListaPiatti.CategorySel = "Pizza";
9               var DescrizionePiatto = new ListaPiatti();
10              await Navigation.PushAsync(DescrizionePiatto);
11          }
12
13          async void Primi_Clicked(object sender, EventArgs e) {
14              ListaPiatti.CategorySel = "Primi";
15              var DescrizionePiatto = new ListaPiatti();
16              await Navigation.PushAsync(DescrizionePiatto);
17          }
18
19          async void Secondi_Clicked(object sender, EventArgs e) {
20              ListaPiatti.CategorySel = "Secondi";
21              var DescrizionePiatto = new ListaPiatti();
22              await Navigation.PushAsync(DescrizionePiatto);
23          }
24
25          async void Contorni_Clicked(object sender, EventArgs e) {
26              ListaPiatti.CategorySel = "Contorni";
27              var DescrizionePiatto = new ListaPiatti();
28              await Navigation.PushAsync(DescrizionePiatto);
29          }
30
31          async void Dolci_Clicked(object sender, EventArgs e) {
32              ListaPiatti.CategorySel = "Dolci";
33              var DescrizionePiatto = new ListaPiatti();
34              await Navigation.PushAsync(DescrizionePiatto);
35          }
36
37          async void Bevande_Clicked(object sender, EventArgs e) {
38              ListaPiatti.CategorySel = "Bevande";
39              var DescrizionePiatto = new ListaPiatti();
40              await Navigation.PushAsync(DescrizionePiatto);
41          }
42      }
```

Listato 5.4. Definizione della classe `Ordina`

Nel Listato 5.5 all'interno della funzione `OnAppearing` (da Riga 3 a Riga 12), viene controllato se la connessione a Internet è presente. In caso di esito negativo viene stampato un errore; altrimenti viene definita la variabile `allMenu`. Questa, richiamando la funzione vista come esempio nel Listato 5.1, scarica da Firebase tutti i piatti appartenenti alla categoria scelta, per poi inserirli nella lista `listaPiatti` che ha il compito di mostrarli all'utente. Successivamente, dopo che quest'ultimo avrà scelto una pietanza, verranno passati all'activity successiva tutti i dati del piatto scelto tramite l'oggetto `item` di tipo `Menu` (Riga 21 - Listato 5.5).

```
1   public partial class ListaPiatti : ContentPage
2           {
3           protected override async void OnAppearing() {
4               base.OnAppearing();
5               if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
6                   await DisplayAlert("Attenzione",
7                   "Non sei connesso ad Internet", "OK");
8               } else {
9                   var allMenu = await firebaseHelper.GetAllMenu(CategorySel);
10                  listaPiatti.ItemsSource = allMenu;
11              }
12          }
13
14          public ListaPiatti() {
15                InitializeComponent ();
16              Title = CategorySel;
17              listaPiatti.ItemSelected += (sender, e) =>
18              listaPiatti.SelectedItem = null;
19              listaPiatti.ItemTapped += async (sender, e) => {
20                  Objects.Menu item = (Objects.Menu)e.Item;
21                  await Navigation.PushAsync(new AggiunteList(item));
22              };
23          }
```

Listato 5.5. Definizione della classe `ListaPiatti`

Nel Listato 5.6 viene gestita la pagina che riguarda le aggiunte (Figura 5.6). Come nel listato precedente, dalla Riga 7 alla Riga 19 viene gestita la connessione

a Internet e vengono inserite nella lista `listViewAgg` le aggiunte relative al piatto selezionato.

Successivamente, tramite la funzione `Switch_Toggled`, vengono gestite le aggiunte selezionate o de-selezionate dall'utente. Se un'aggiunta viene selezionata, viene aggiunta ad un oggetto `agg`, insieme al suo prezzo. Infine, quando l'utente clicca il pulsante *Continua* (Riga 30), vengono passati nella activity successiva le informazioni riguardanti il piatto, insieme ai nomi e ai prezzi delle aggiunte selezionate.

```
1   public partial class AggiunteList : ContentPage
2           {
3           public List<Aggiunte> aggiunteSel { get; set; }
4           public double Totaleaggiunte { get; set; }
5           List<Aggiunte> allAggiunte = new List<Aggiunte>();
6
7           protected override async void OnAppearing()
8           {
9               base.OnAppearing();
10              allAggiunte.Clear();
11              if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
12                  await DisplayAlert("Attenzione",
13                  "Non sei connesso ad Internet", "OK");
14              } else {
15                  allAggiunte = await
16                  firebaseHelper.GetAllAggiunte(ListaPiatti.CategorySel);
17                  listViewAgg.ItemsSource = allAggiunte;
18              }
19          }
20
21          public AggiunteList(Objects.Menu item)
22          {
23              aggiunteSel = new List<Aggiunte>();
24              aggiunteSel.Clear();
25              Totaleaggiunte = 0.0;
26              InitializeComponent();
27              nameLabel.Text = item.Name;
28              priceLabel.Text = string.Format(" {0:0.00}", item.Price);
29              imageLabel.Source = item.LinkImage;
30              Continua.Clicked += async (sender, args) =>
31              {
32                  await Navigation.PushAsync(new Note
33                  (item, aggiunteSel, Totaleaggiunte));
34              };
35          }
36
37          private void Switch_Toggled(object sender, ToggledEventArgs e)
38          {
39              var obj = sender as Switch;
40              var agg = obj?.BindingContext as Aggiunte;
41              if (e.Value == true) {
42                  aggiunteSel.Add(agg);
43                  Totaleaggiunte += (double)agg.Price;
44              } else if (e.Value == false) {
45                  aggiunteSel.Remove(agg);
46                  Totaleaggiunte -= (double)agg.Price;
47              }
48          }
49      }
```

Listato 5.6. Definizione della classe `AggiunteList`

Nel Listato 5.7 è presente la classe `Note`, che consente di ottenere la videata mostrata in Figura 5.7. Tramite tale videata è possibile inserire delle note, che poi verranno collegate al piatto, per comunicare con la cucina, in caso di intolleranze o allergie. Una volta premuto il tasto *Continua* a fine pagina, indipendentemente dal fatto che la nota sia stata inserita o meno, verrà creato l'oggetto che poi verrà inserito nel carrello tramite la funzione `CreaOggetto`, inserita all'interno della classe `ItemCarrello` (Riga 16).

Una volta che il piatto viene inserito nel carrello, l'applicazione eliminerà tutto lo stack di pagine visitate precedentemente, in modo che l'utente non possa tornare indietro e modificare l'ordine, creando problemi nell'applicazione. Questo

viene fatto tramite un `foreach` che, per ogni pagina esistente, chiamerà la funzione `Navigation.RemovePage`.

```
1   public partial class Note : ContentPage
2       {
3               public Note (Objects.Menu item, List<Aggiunte> AggiunteSel = null,
4               double TotaleAggiunte = 0.00)
5               {
6           Title = item.Category;
7                       InitializeComponent ();
8           nameLabel.Text = item.Name;
9           priceLabel.Text = string.Format(" {0:0.00}", item.Price);
10          imageLabel.Source = item.LinkImage;
11          double Totale = item.Price + TotaleAggiunte;
12          Aggiungi.Clicked += (sender, e) =>
13          {
14              string nota = noteEditor.Text;
15              ObjCarrello ItemCarrello = new ObjCarrello();
16              ItemCarrello.CreaOggetto(item, AggiunteSel,
17              nota, TotaleAggiunte);
18              DisplayAlert("", "Hai aggiunto un piatto al carrello",
19              "OK");
20              Navigation.PushAsync(new MainPage(1));
21              var existingPages = Navigation.NavigationStack.ToList();
22              foreach (var page in existingPages)
23              {
24                  Navigation.RemovePage(page);
25              }
26          };
27      }
28  }
```

Listato 5.7. Definizione della classe `Note`

5.3.2 Area Riservata

Per quanto riguarda l'Area Riservata, invece, analizzeremo il codice riguardante la pagina del login destinata ai camerieri, le impostazioni che essi possono compiere all'interno della loro area e, infine, la parte di codice che serve per bloccare l'applicazione quando l'utente deve completare il pagamento.

Nel Listato 5.8 è presente la classe `LoginStaff`, la quale si occupa di gestire il login dei camerieri. Per fare ciò, dopo aver verificato se è presente la connessione a Internet, scaricherà da Firebase tutti gli utenti presenti all'interno del nodo `Staff` del database e li inserirà all'interno della lista `listaStaff`.

La funzione `LoginUtente` (Riga 22), che viene azionata quando viene cliccato il pulsante *Login*, controllerà prima di tutto che l'utente non abbia lasciato i campi vuoti; successivamente, creerà due stringhe, `mUsername` e `mPassword`, che salvano al loro interno lo username e la password inseriti dal cameriere. Infine, la funzione controllerà se all'interno della lista `listaStaff`, creata precedentemente, c'è un utente con username e password uguale a quelli inseriti. In questo caso il login avviene con successo; altrimenti verrà visualizzato sulla schermata un messaggio di errore.

```
1   public partial class LoginStaff : ContentPage
2       {
3       List<Staff> listaStaff = new List<Staff>();
4
5       protected override async void OnAppearing() {
6           base.OnAppearing();
7           listaStaff.Clear();
8           if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
9               await DisplayAlert("Attenzione",
10              "Non sei connesso ad Internet", "OK");
11          } else
12          listaStaff = await firebaseHelper.GetAllStaff();
13      }
14
```

```
15          public LoginStaff(bool pagamento = false) {
16              InitializeComponent();
17              if (pagamento) {
18                  Avviso.IsVisible = true;
19              }
20          }
21
22          public async void LoginUtente(object o, EventArgs e) {
23              if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
24                  await DisplayAlert("Attenzione",
25                  "Non sei connesso ad Internet", "OK");
26              } else {
27                  if (string.IsNullOrEmpty(entryUsername.Text) ||
28                   string.IsNullOrEmpty(entryPassword.Text)) {
29                     await DisplayAlert("Attenzione", "Riempi tutti i campi",
30                     "OK");
31                  } else {
32                     string mUsername = entryUsername.Text;
33                     string mPassword = entryPassword.Text;
34                     if(listaStaff.Any(x => x.Username == mUsername
35                     && x.Password == mPassword)) {
36                         await Navigation.PushAsync(new AreaRiservata());
37                     } else {
38                         await DisplayAlert("Errore",
39                         "I dati inseriti non sono corretti", "OK");
40                     }
41                  }
42              }
43          }
44      }
```

Listato 5.8. Definizione della classe `LoginStaff`

Il Listato 5.9 presenta al proprio interno la classe `AreaRiservata`, mostrata in Figura 5.16. In essa, il cameriere può cambiare il tavolo collegato all'applicazione o confermare il pagamento del cliente.

Nel caso scelga di cambiare il tavolo, l'applicazione, prima di tutto, si assicura che non siano stati richiesti pagamenti. In questo caso, dopo aver inviato un messaggio di conferma, incrementa la sessione, effettua il logout del cliente e resetta il carrello (Righe 12-30).

Nel caso in cui il cameriere scelga, invece, di confermare il pagamento, dovrà, prima di tutto, indicare se il cliente ha pagato o meno. In caso affermativo gli verranno aggiornati i punti, altrimenti questi non verranno modificati.

Successivamente viene invocata la funzione `Finisci` (Riga 53) che si occuperà di aggiornare la sessione ed effettuare il logout del cliente. Essa, inoltre, si occuperà di svuotare il carrello, cancellare lo stack delle pagine visitate in precedenza e rimuovere il blocco dell'app qualora questo sia presente.

```
1   public partial class AreaRiservata : ContentPage
2           {
3           public AreaRiservata() {
4               InitializeComponent();
5               Auth = DependencyService.Get<IAuth>();
6               var Tavolo = Application.Current.Properties["NTavolo"];
7               var Sessione = Application.Current.Properties["NSessione"];
8               Text_Sessione.Text = "Numero Sessione: " + Sessione;
9               Text_Tavolo.Text = "Numero Tavolo: " + Tavolo;
10          }
11
12          private async void CambiaTavolo_Clicked(object sender, EventArgs e) {
13              int block = (int)(Application.Current.Properties["BlockPref"]);
14              if (block == 1) {
15              await DisplayAlert("ATTENZIONE",
16              "Ci risulta che il cliente ha richiesto il pagamento,
17              chiudere prima il pagamento cliccando su CONFERMA PAGAMENTO",
18              "ok");
19              } else {
20              bool answer = await DisplayAlert("ATTENZIONE",
21              "Vuoi veramente chiudere la sessione corrente e cambiare numero tavolo?",
22              "Si", "No");
23                  if (answer) {
24                      await firebaseHelper.UpdateSession();
25                      Auth.Logout();
26                      ListaCarrello.ResettaCarrello();
27                      await Navigation.PushAsync(new ImpostaTavolo());
28                  }
```

```
29              }
30          }
31
32          private async void ConfermaPagamento_Clicked(object sender, EventArgs e) {
33              if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
34                  await DisplayAlert("Attenzione",
35                  "Non sei connesso ad Internet", "OK");
36              } else {
37                  string answer = await DisplayActionSheet
38                  ("Il cliente ha pagato?", "Annulla", null,
39                  "SI", "NO");
40                  if (answer == "SI") {
41                      await firebaseHelper.UpdateOrder();
42                      await firebaseHelper.UpdatePoints
43                      (ProcediAlPagamento.NewPoints);
44                      Console.WriteLine("Cliccato si");
45                      Finisci();
46                  } else if (answer == "NO") {
47                      Console.WriteLine("Cliccato no");
48                      Finisci();
49                  }
50              }
51          }
52
53          public async void Finisci() {
54              if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
55                  await DisplayAlert("Attenzione",
56                  "Non sei connesso ad Internet", "OK");
57              } else {
58                  await firebaseHelper.UpdateSession();
59                  await firebaseHelper.AddSession();
60                  Auth.Logout();
61                  ListaCarrello.ResettaCarrello();
62                  Application.Current.Properties["BlockPref"] = 0;
63                  await Navigation.PushAsync(new MainPage());
64                  var existingPages = Navigation.NavigationStack.ToList();
65                  foreach (var page in existingPages) {
66                      Navigation.RemovePage(page);
67                  }
68              }
69          }
70      }
```

Listato 5.9. Definizione della classe `AreaRiservata`

Per quanto riguarda il blocco dell'app, esso serve per bloccare completamente l'applicazione mostrando la schermata di login dell'*Area Riservata*, dopo che il cliente ha richiesto il pagamento. Per renderlo funzionante, utilizziamo le *Properties* presenti in Visual Studio, inserendo una nuova chiave chiamata *BlockPref*. Se essa ha come valore 0 allora l'applicazione non è bloccata, altrimenti, se ha come valore 1, la pagina principale diventa `LoginStaff`, a cui viene passato il valore `true`. Tutto ciò è mostrato nel Listato 5.10.

Se il blocco app è presente, passando il valore `true` alla funzione `LoginStaff`, viene anche mostrata la scritta "*Attendi per favore un cameriere che confermerà il pagamento*" (Righe 15-20 - Listato 5.8).

```
1     public App()
2           {
3               if (!Application.Current.Properties.ContainsKey("BlockPref")) {
4                   Application.Current.Properties["BlockPref"] = 0;
5               }
6               if (!Application.Current.Properties.ContainsKey("NTavolo")) {
7                   MainPage = new NavigationPage(new ImpostaTavolo());
8               } else if ((int)Application.Current.Properties["BlockPref"] == 1) {
9                   MainPage = new NavigationPage(new LoginStaff(true));
10              } else {
11                  MainPage = new NavigationPage(new MainPage());
12              }
13          }
```

Listato 5.10. Definizione parziale della classe `App`

5.3.3 Sistemi gestione dei punti utente

Si è anche deciso di creare un sistema che possa in qualche modo fidelizzare gli utenti. Proponendo degli sconti, infatti, i clienti sono solitamente più propensi a tornare nel ristorante. Per fare ciò, si è deciso di attribuire un punto, del valore di un euro, ogni cinque euro di spesa. L'utente, quindi, prima di procedere al pagamento, avrà uno sconto pari al numero dei punti ottenuti in precedenza.

Nel Listato 5.11 viene mostrata la funzione che si occupa dell'attribuzione dei punti. Viene, prima di tutto, creata una variabile per memorizzare il nuovo numero di punti. Dopo aver verificato se l'utente è autenticato (Riga 13) e aver verificato il suo numero di punti attuale (Riga 16) viene inizializzata la variabile `NewPoints` a 0.

Successivamente, si verifica se il numero di punti in possesso è maggiore del totale dell'ordine. Se ciò accade, l'utente non dovrà pagare nulla per l'ordine e avrà, come nuovo saldo punti, quello precedente meno il totale dell'ordine attuale. Se, invece, il numero di punti risulta minore del costo dell'ordine, avrà come nuovo totale dell'ordine quello precedente meno i punti ottenuti in precedenza, e come punti la nuova spesa diviso cinque.

```
1   public static int NewPoints { get; set; }
2
3   protected override async void OnAppearing()
4           {
5               base.OnAppearing();
6               allUser.Clear();
7               allUser = await firebaseHelper.GetAllUser();
8
9               if (Connectivity.NetworkAccess != NetworkAccess.Internet) {
10                  await DisplayAlert("Attenzione",
11                  "Non sei connesso ad Internet", "OK");
12                  return;
13              } else if (Auth.IsLoggedIn()) {
14                  User u = allUser.FirstOrDefault(a => a.Email == Auth.Email());
15                  double NewTotalOrder;
16                  int Points = u.Points;
17                  if (Points > Ordini.TotaledaStampare) {
18                      NewTotalOrder = 0;
19                      NewPoints = (int)(Points - Ordini.TotaledaStampare);
20                  } else {
21                      NewTotalOrder = Ordini.TotaledaStampare - Points;
22                      NewPoints = (int)(NewTotalOrder / 5);
23                  }
24              }
25          }
```

Listato 5.11. Definizione del metodo `OnAppearing` della classe `ProcediAlPagamento` per l'attribuzione dei punti

6

Discussione in merito al lavoro svolto

In questo capitolo discuteremo i punti di forza e di debolezza dell'applicazione realizzata e discussa fino ad ora all'interno della tesi. Successivamente, illustreremo le analogie e le differenze rispetto ad altri sistemi simili presenti sul mercato.

6.1 Punti di forza del sistema realizzato

Dopo aver analizzato l'applicativo, si è deciso di elencare i punti di forza del sistema realizzato.

Si può vedere che l'applicazione memorizza e utilizza i dati contenuti in un database online. Facendo uso di *Firebase Realtime Database*, è possibile, infatti, memorizzare le informazioni relative al menù e agli utenti che utilizzano l'applicativo. Grazie a questo, si può modificare facilmente e velocemente il menù, inserendo o modificando delle pietanze, senza dover fornire un'altra versione dell'app.

Allo stesso modo, è anche possibile inserire o modificare delle nuove aggiunte. Al momento, per alcune categorie, come *Secondi* o *Dolci* (Figura 6.1), le aggiunte non sono previste. Se il cameriere o il ristorante decidesse, invece, di inserirle, basterebbe modificare il database online collegato all'applicativo semplicemente collegando le nuove aggiunte alla categoria desiderata. In questo modo, l'applicazione, trovando degli elementi all'interno della categoria, li mostrerà automaticamente al cliente senza dover modificare il codice dell'applicativo.

Un altro punto di forza riguarda l'implementazione del profilo utente. Il cliente, infatti, ha la possibilità di registrarsi tramite e-mail e password, oppure tramite *Google*, all'interno dell'applicativo, grazie alla componente *Firebase Authentication* che si è deciso di implementare. In questo modo, successivamente, potrà effettuare il login e beneficiare di alcuni sconti o offerte a lui dedicati.

Inoltre, all'interno della scheda "*Profilo*", l'utente potrà visualizzare i punti guadagnati fino ad ora e vedere gli ordini completati. In più, egli potrà controllare se ha ordini non pagati, anche in sessioni precedenti rispetto a quella attuale, e sarà libero di pagarli in qualsiasi momento.

Un ulteriore aspetto importante riguarda la stabilità dell'applicazione. Infatti, anche quando questa viene utilizzata su più dispositivi contemporaneamente, rimane stabile. Essendo installata all'interno dei dispositivi situati sui tavoli, l'applicazione

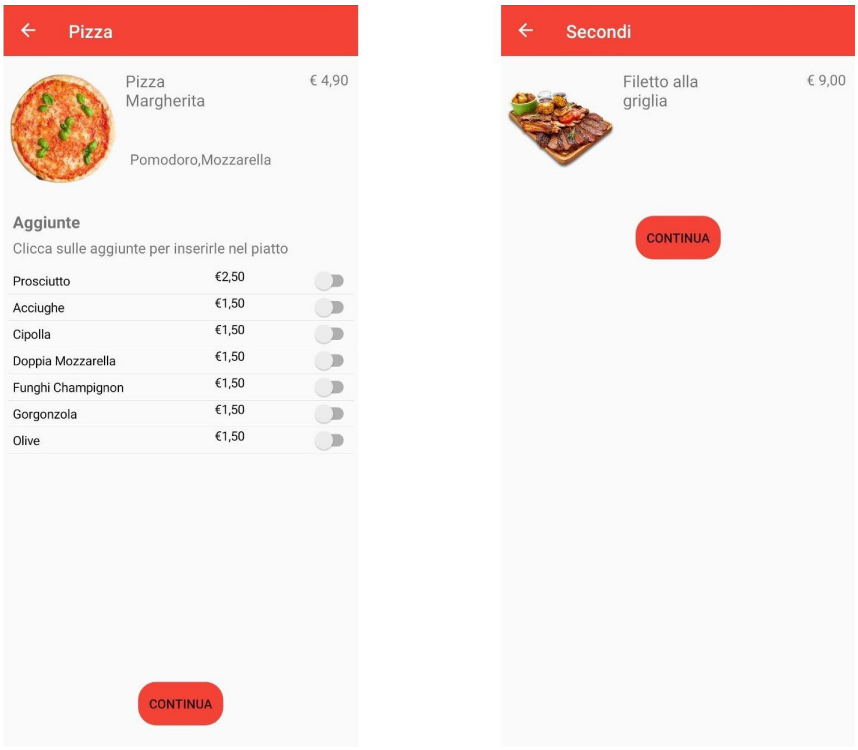


Figura 6.1. Confronto tra sezioni in presenza e in assenza di aggiunte

deve essere in grado di funzionare anche quando viene utilizzata da un elevato numero di utenti.

Infine, l'applicazione resterà completamente bloccata, dopo che il cliente ha richiesto il pagamento, visto che egli non dovrà effettuare altri ordini. Infatti, anche se viene premuto il tasto indietro, l'applicazione non cambierà schermata e il blocco rimarrà presente.

6.2 Punti di debolezza del sistema realizzato

Durante l'analisi dell'applicativo, oltre ai punti di forza, sono stati trovati anche dei punti di debolezza. Questi potranno essere migliorati o riprogettati nel caso in cui si decida di modificare ulteriormente il progetto.

La criticità maggiore è legata al fatto che non è presente un'applicazione o un sito web per il cameriere, dove egli possa vedere gli ordini effettuati dall'utente o i vari messaggi inviati da quest'ultimo. Per visualizzarli, al momento, si dovrà, infatti, accedere direttamente al database. Questo è stato fatto perchè si è ritenuto più opportuno ideare un'applicazione più incentrata sull'utilizzo del cliente.

Un altro punto di debolezza risiede nel fatto che non è possibile modificare il menù del dispositivo all'interno dell'applicazione o attraverso una piattaforma web (Figura 6.2). Non è stato previsto, infatti, uno strumento che permetta di inserire

o modificare le pietanze presenti nel menù. Il cameriere dovrà, quindi, effettuare l'accesso al database presente su *Firebase* per modificare le pietanze. Ciò potrebbe risultare ostico e complicato, rendendo il procedimento lungo rispetto alle esigenze del singolo ristorante. In più, potrebbe mettere a rischio la persistenza dei piatti, nel caso in cui il cameriere faccia un errore durante l'inserimento o la modifica del piatto.

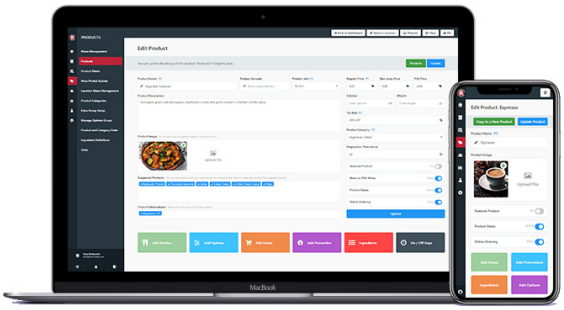


Figura 6.2. Applicazione e sito web che permettono la modifica del menù

Inoltre, non è stato previsto un sistema per il pagamento in-app. Infatti, quando viene richiesto il pagamento, sarà il cameriere che dovrà effettuarlo, recandosi al tavolo. Ciò avviene sia in caso di pagamento con contanti sia nel caso in cui il pagamento avvenga tramite carta. Questa scelta è stata effettuata principalmente perchè implementare un sistema di pagamento sicuro ed efficiente risulta complicato e dispendioso.

Infine, si è pensato che l'applicazione potrebbe essere ampliata aggiungendo delle *notifiche push*, tramite le quali si potrebbe pubblicizzare il ristorante o una particolare offerta presente in quel momento. Inoltre, si potrebbe aggiungere una sezione del menù dedicata al piatto del giorno o a delle particolari pietanze che potrebbero essere in sconto.

6.3 Analogie rispetto ad altri sistemi simili esistenti sul mercato

Sul mercato, esistono diversi applicativi simili alla soluzione realizzata, come, per esempio, *Nu-Menu* (`https://www.nu-menu.co.za/`, *Menu* (`https://menu.app/en/`) oppure *FineDine* (`https://www.finedinemenu.com/`). Si può osservare che le caratteristiche in comune con l'applicativo realizzato sono molteplici.

La prima è quella di avere un menù interattivo, tramite il quale il cliente possa scegliere la pietanza che preferisce, vedendo gli ingredienti e le foto dei singoli piatti, e, infine, inserendo il piatto scelto nel carrello. Come nel nostro applicativo, anche altri sistemi permettono di modificare la pietanza inserendo alcune aggiunte (Figura 6.3).

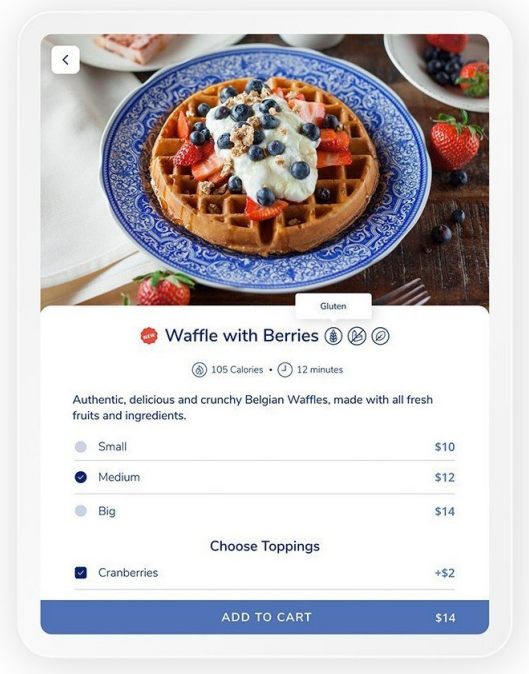


Figura 6.3. Sezione relativa alla scelta del piatto dell'app *FineDine*

Inoltre, è anche possibile, all'interno di diversi sistemi, vedere una foto in alta risoluzione che rappresenti realmente il piatto, insieme ai suoi ingredienti e ai piatti correlati.

Un'altra caratteristica che accomuna i diversi applicativi è la possibilità di registrarsi all'interno dell'applicazione e creare un profilo utente, tramite il quale si favorisce la fidelizzazione. Infatti, si possono trovare diversi sconti personalizzati anche tramite l'utilizzo di punti al momento del pagamento (Figura 6.4). In più, è possibile utilizzare alcuni sconti tramite *coupon*, oppure visualizzare pubblicità che sponsorizzano alcune pietanze presenti nel menù.

6.4 Differenze rispetto ad altri sistemi simili esistenti sul mercato

Analizzando le diverse soluzioni presenti sul mercato, è anche possibile evidenziare alcune differenze con l'applicativo realizzato.

La prima differenza è che la nostra applicazione, per funzionare correttamente, ha un costante bisogno della connessione a Internet. Questo è dovuto al fatto che l'app deve comunicare con il database online, per scaricare ogni volta i dati aggiornati che poi mostrerà all'utente.

I sistemi presenti in commercio, invece, permettono di utilizzare alcune parti dell'applicazione senza l'utilizzo della connessione a Internet, come, per esempio, la

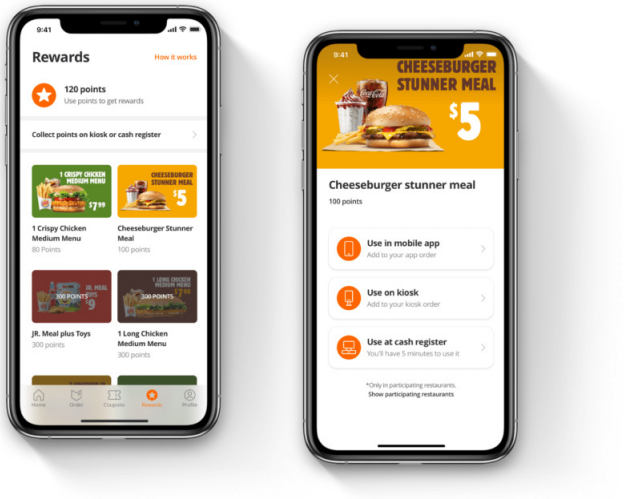


Figura 6.4. Sezione relativa al pagamento dell'app *iMenu*

visualizzazione del menù. Il sistema ha, però, bisogno della connessione per poter inviare gli ordini effettuati alla cucina o per visualizzare alcune offerte speciali.

Inoltre, quasi nessuna delle applicazioni presenti sul mercato permette di inviare messaggi al cameriere. Questo potrebbe essere utile se si hanno alcune comunicazioni urgenti da fare, oppure se l'utente non è in grado di utilizzare correttamente l'applicazione e ha bisogno dell'aiuto del personale.

Invece, una funzionalità non presente nell'applicativo realizzato è quella che permette di recensire il servizio di ristorazione, oppure un singolo piatto. Ciò potrebbe aiutare il ristorante a capire come migliorare, nel caso in cui qualche valutazione non sia positiva.

Si può anche vedere come i diversi sistemi adottino diverse soluzioni per quel che riguarda l'implementazione del numero del tavolo all'interno del dispositivo. Essendo questa informazione di notevole importanza, per far sì che il cameriere consegni le pietanze ordinate al giusto tavolo, si è deciso, nel nostro applicativo, di farlo configurare direttamente al personale prima che il cliente arrivi al tavolo. Altri sistemi, invece, sfruttano, ad esempio, un *codice QR*, posizionato sul tavolo, da inquadrare per configurare l'applicazione, o permettono al cliente stesso di inserire il numero all'interno dell'applicativo prima di iniziare ad ordinare (Figura 6.5).

Infine, un'ultima differenza risiede nel fatto che non abbiamo considerato di sviluppare un'applicazione adatta per il sistema operativo *iOS*. Al giorno d'oggi, si deve fare in modo che, quando un'applicazione viene progettata, sia compatibile con più dispositivi possibili, in modo tale da poter essere utilizzata anche su dispositivi già presenti nel locale e aumentarne l'utilizzo.

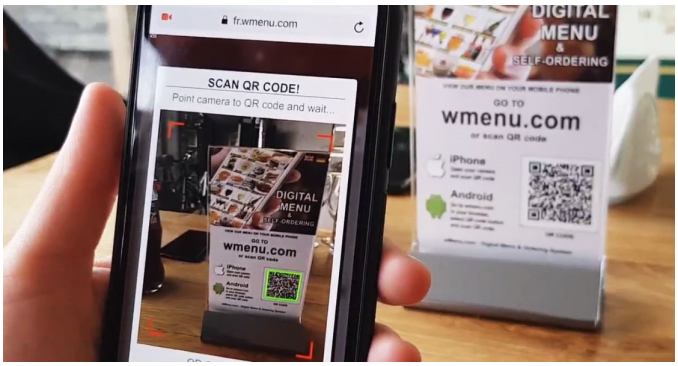


Figura 6.5. Funzionalità del *codice QR* dell'applicazione *Wmenu*

7

Conclusioni

In questa tesi è stata progettata e realizzata un'applicazione mobile per i dispositivi *Android*, implementata tramite il framework *Xamarin*, che permette agli utenti di effettuare ordinazioni al ristorante senza la presenza di un cameriere. L'applicativo implementa un menù interattivo, tramite il quale l'utente può selezionare la pietanza di suo gradimento, personalizzarla e ordinarla. In più, tutte le informazioni visualizzabili all'interno dell'app vengono salvate online, in modo da essere facilmente recuperate dal personale o da altri sistemi.

Inizialmente abbiamo approfondito la differenza tra app native, web app e app ibride, indicando i vari framework presenti sul mercato per realizzare queste ultime. Successivamente, avendo scelto di utilizzare il framework Xamarin, abbiamo studiato i suoi componenti, la sua architettura e la modalità di condivisione del codice, in modo da comprendere pienamente gli strumenti messi a disposizione dal framework.

Si è, poi, proceduto ad un'analisi dei requisiti, la quale ci ha permesso di descrivere il progetto nella sua interezza elencando le caratteristiche che l'applicazione finale avrebbe dovuto implementare. Successivamente, siamo passati all'implementazione, partendo dall'interfaccia dell'applicazione, tramite i *Mockup* e i *Diagrammi delle attività*, per poi proseguire con la progettazione concettuale e logica della base di dati dell'applicativo. Infine, nella fase relativa all'implementazione, abbiamo discusso le principali funzionalità inserite all'interno dell'app, avvalendoci del supporto di *Firebase*, implementato tramite alcuni pacchetti *NuGet*. In ultimo, abbiamo analizzato l'applicativo per trovare i vari punti di forza e di debolezza presenti al suo interno, soffermandoci sulle analogie e sulle differenze rispetto ad altri sistemi simili esistenti sul mercato.

Tramite le ultime analisi effettuate nell'applicazione, abbiamo potuto osservare come questo potrebbe essere migliorato tramite alcune modifiche o alcuni aspetti che potrebbero essere implementati in futuro. Si potrebbe, infatti, utilizzare un sistema di *notifiche push* per pubblicizzare particolari sconti o eventi, in modo da aumentare la fidelizzazione dell'utente. Si potrebbe, inoltre, realizzare un sito web o un'app dedicati al personale, per poter mostrare loro i vari ordini o i vari messaggi inviati dai clienti.

Ringraziamenti

Desidero, innanzitutto, ringraziare tutti coloro che mi hanno aiutato nella realizzazione del progetto e nella stesura di questa tesi di laurea. Ringrazio il mio relatore, il Professore Domenico Ursino, per la sua competenza e per la sua disponibilità e rapidità nel curare e correggere la tesi, capitolo dopo capitolo.

Vorrei ringraziare la mia famiglia, che ha sempre creduto in me e mi ha sempre sostenuto durante questo percorso di studi.

Ringrazio, inoltre, tutti i miei amici, l'Azione Cattolica e tutti i miei compagni di corso, che mi hanno aiutato in questi anni e, in particolare, il mio collega Marco La Gala, con il quale ho condiviso questo progetto.

Riferimenti bibliografici

1. What is a Hybrid Mobile App? https://www.telerik.com/blogs/what-is-a-hybrid-mobile-app-, 2012.
2. App ibride o app native? Questo è il dilemma. https://www.devinterface.com/it/blog/app-ibride-o-app-native-questo-e-il-dilemma, 2015.
3. Architectural overview of Cordova platform - Apache Cordova. https://cordova.apache.org/docs/en/latest/guide/overview/index.html, 2015.
4. Condivisione del codice: Shared Asset Projects. https://www.html.it/pag/60573/condivisione-del-codice-shared-assets-project/, 2016.
5. Pagine di Xamarin.Forms. https://docs.microsoft.com/it-it/xamarin/xamarin-forms/user-interface/controls/pages, 2016.
6. Portable Class Libraries (PCLs). https://www.html.it/pag/60577/portable-class-libraries-pcls/, 2016.
7. Architettura App iOS. https://docs.microsoft.com/it-it/xamarin/ios/internals/architecture, 2017.
8. MVVM e DataBinding. https://www.html.it/pag/62249/mvvm-e-databinding/, 2017.
9. Sviluppare app mobile con React Native. https://www.develer.com/sviluppare-app-mobile-con-react-native/, 2017.
10. What's Revolutionary about Flutter. https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514, 2017.
11. Getting Started. https://facebook.github.io/react-native/docs/getting-started, 2018.
12. Layout di Xamarin.Forms. https://docs.microsoft.com/it-it/xamarin/xamarin-forms/user-interface/controls/layouts, 2018.
13. Matrice di tracciabilità. https://it.wikipedia.org/wiki/Matrice_di_tracciabilit%C3%A0, 2018.
14. Xamarin.Android - Architettura. https://docs.microsoft.com/it-it/xamarin/android/internals/architecture, 2018.
15. Benvenuti all'IDE di Visual Studiog. https://docs.microsoft.com/it-it/visualstudio/get-started/visual-studio-ide?view=vs-2019, 2019.
16. Flutter: nuovo framework per lo sviluppo cross-platform. https://www.html.it/pag/367891/flutter-nuovo-framework-per-lo-sviluppo-cross-platform/, 2019.
17. Framework. https://it.wikipedia.org/wiki/Framework, 2019.
18. Mockup. https://it.wikipedia.org/wiki/Mockup, 2019.
19. Use Case Diagram. https://it.wikipedia.org/wiki/Use_Case_Diagram, 2019.
20. Viste Xamarin.Forms. https://docs.microsoft.com/it-it/xamarin/xamarin-forms/user-interface/controls/views, 2019.

21. What is Apache Cordova? `https://ionicframework.com/resources/articles/what-is-apache-cordova`, 2019.
22. Firebase Authentication. `https://firebase.google.com/docs/auth`, 2020.
23. Firebase Realtime Database. `https://firebase.google.com/docs/database`, 2020.
24. S.Paraboschi P.Atzeni, S.Ceri and R.Torlone. *Basi di dati*. McGraw-Hill, 2006.