Bingo

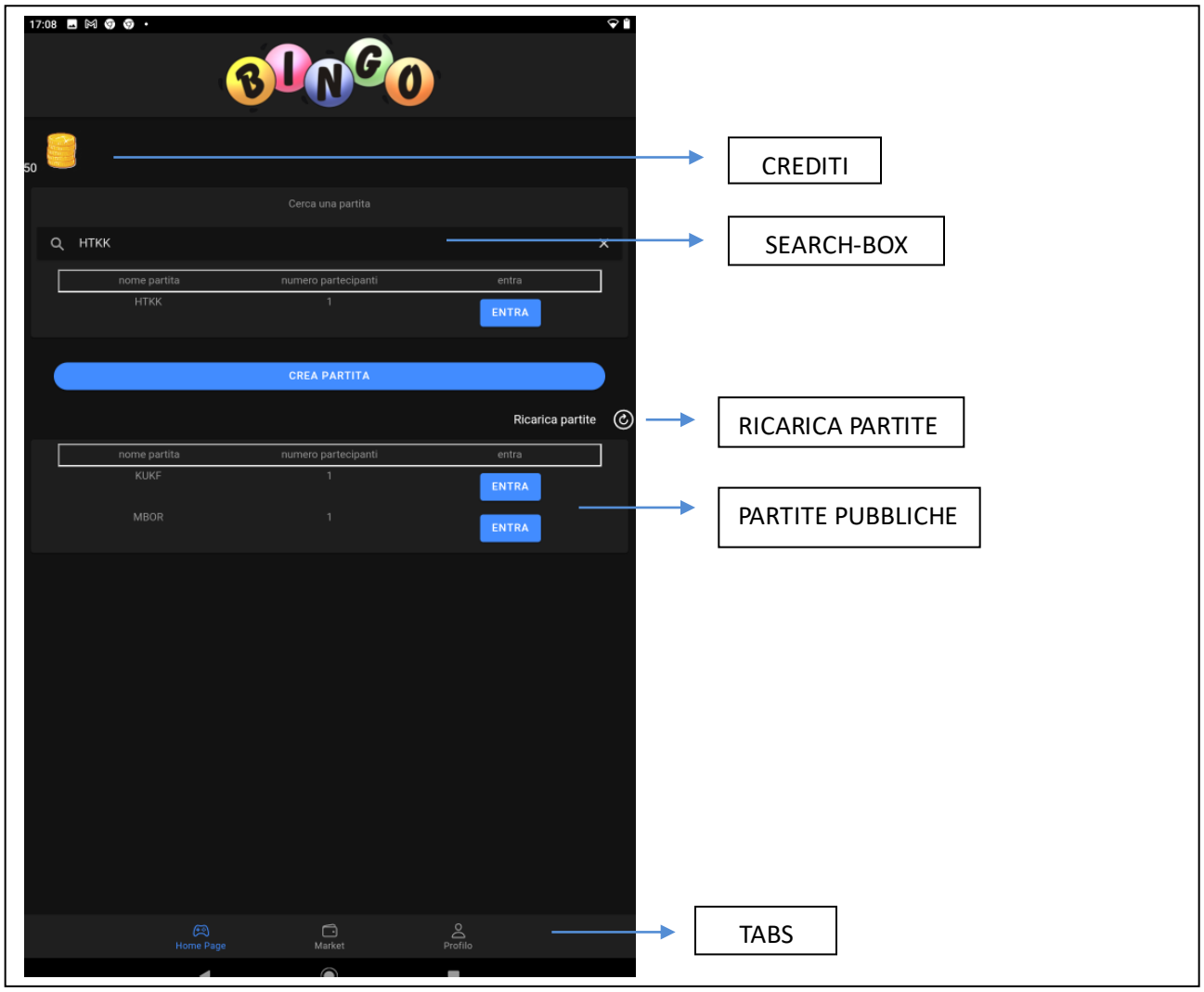
Laboratorio di applicazioni mobili

A.c.: 2021/2022

Alsina Gabriel Riccardo Num.Matricola: 890823 gabriel.alsina@studio.unibo.it	Carboni Carlotta Num. Matricola: 893185 carlotta.carboni2@studio.unibo.it

Manuale utente

La prima volta che si apre l'app, dopo averla scaricata, si aprirà su una pagina che permette di registrarsi o di loggarsi. Per registrarsi verranno richiesti dei dati, se non vengono inseriti tutti, o se l'username scelto è già in uso comparirà un alert per segnalare il problema (controlli equivalenti vengono fatti anche per il login). Una volta registrati o loggati, si verrà reindirizzati alla home page.



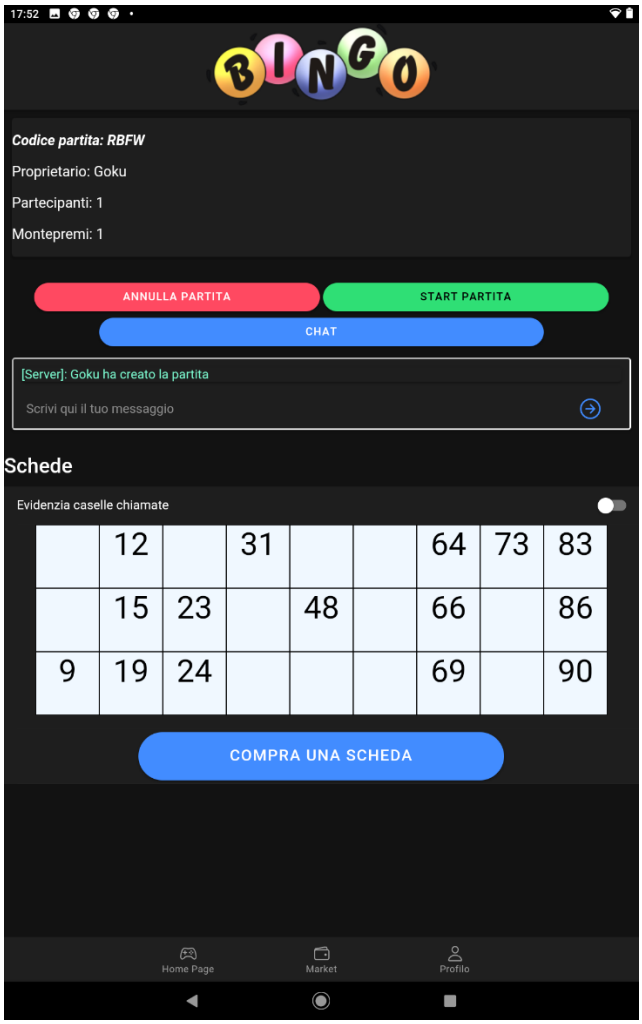
In alto a sinistra si trovano i crediti, necessari per poter giocare. Nel search-box è possibile inserire il codice di una partita: questo box è l'unico modo per poter accedere alle partite private, ma si

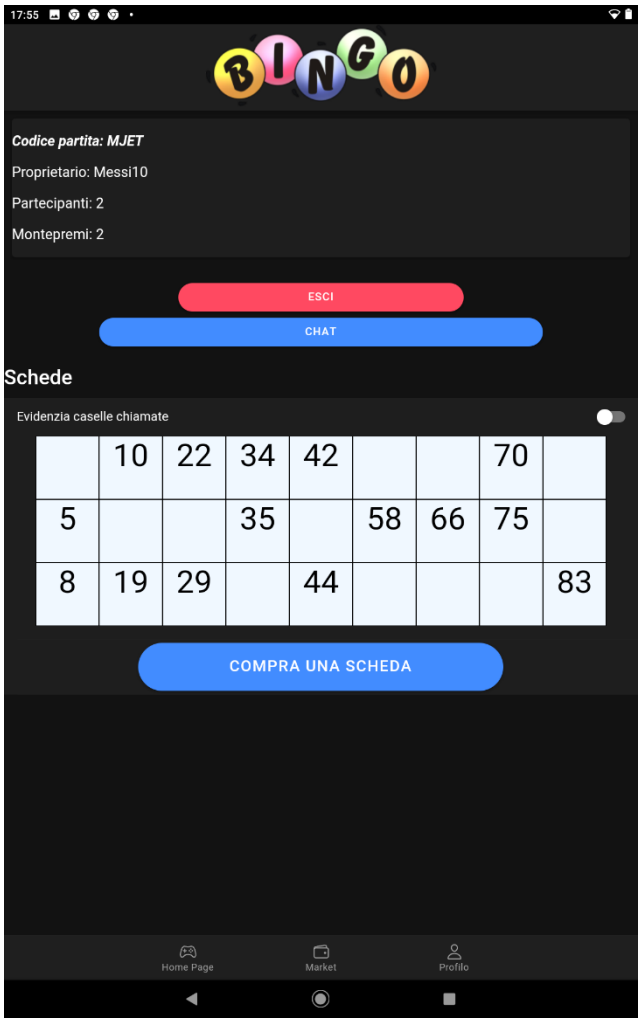
può anche utilizzare per cercare partite pubbliche. Le partite pubbliche attive sono comunque tutte visibili nella tabella sottostante.

Per entrare in una partita basta cliccare sul bottone entra, mentre per crearla basterà cliccare su crea partita, in entrambi i casi è necessario avere almeno un credito, ovvero il costo di una scheda. Se non si hanno abbastanza crediti si verrà avvisati tramite un alert.

Per aggiornare le partite pubbliche è presente un bottone che ne permette l'aggiornamento, non venendo aggiornate in real-time ad ogni nuova partita sono presenti dei controlli che nel caso si provasse ad entrare in una partita già iniziata si verrà bloccati e invitati a cliccare sul bottone per poter aggiornare lo stato delle partite. (La scelta di non aggiornare le partite in real-team è stata presa immaginando un elevato numero di utenti che creano/startano/cancellano le partite e visualizzare una tabella "dinamica" che continua a cambiare renderebbe difficile la selezione di una partita e probabilmente sarebbe anche fastidioso da vedere.)

Per creare una partita, dopo aver cliccato su crea partita, si dovrà scegliere il tipo di partita: pubblica o privata, la differenza tra le due tipologie è che quella pubblica sarà visibile a tutti, mentre la privata solo a chi è a conoscenza del codice della partita.
Eseguita la scelta entriamo nell'istanza della partita.





In alto si trovano i dati della partita: codice, proprietario, partecipanti e montepremi (che è in crediti), questi ultimi due vengono aggiornati con l'entrata/uscita dei giocatori e con l'acquisto di nuove cartelle.

I bottoni per gestire la partita sono diversi nel caso non si sia il proprietario, resta però un bottone in comune: quello per la visualizzazione della chat. Nel caso si sia il proprietario la chat inizia aperta perché è qui che si verrà avvisati su chi entra ed esce dalla partita.

Nel pre-partita il proprietario e i giocatori potranno ancora uscire dalla partita, venendo rimborsati dei crediti spesi, tramite i rispettivi bottoni di uscita o tramite i tabs. Se a uscire dalla partita è il proprietario la partita verrà cancellata e tutti i giocatori verranno reindirizzati alla home, avvisati da un alert che spiega il motivo dell'uscita; mentre se a uscire è un giocatore verrà semplicemente ridotto il numero dei giocatori e aggiornato il montepremi. Se i giocatori escono a partita iniziata non verranno rimborsati, mentre se, a partita iniziata, è il proprietario ad uscire i giocatori verranno rimborsati dei crediti.

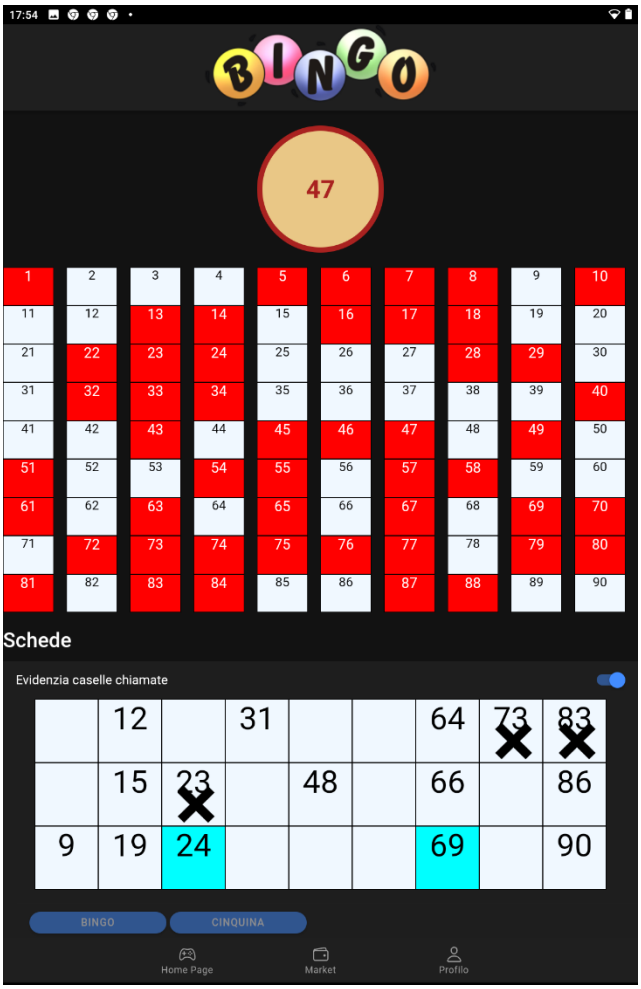
(Se a rimanere in partita è solo il proprietario la partita non verrà annullata, lo consideriamo un premio di consolazione dopo che tutti hanno abbandonato la sua partita.)

Quando si entra/si crea una partita viene generata e comprata una scheda, ma tramite al bottone compra una scheda se ne potranno comprare altre fino ad un massimo di tre per giocatore. Il bottone "compra scheda" è visibile finché non inizia la partita o finché non si comprano 3 schede, anche se non si hanno crediti per comprare altre schede il bottone non viene disabilitato, ma in assenza di sufficienti crediti, cliccandoci, comparirà un alert che permetterà di acquisire 10 crediti senza abbandonare la partita (è comunque possibile rifiutare l'acquisto dei 10 crediti).

Sopra alle schede è presente un toggle: "Evidenzia caselle chiamate", attivandolo quando un numero presente nella cartella verrà estratto questo cambierà colore. ATTENZIONE: il fatto che la cella si illumini non significa sia stato segnato, il cambio colore serve solo per facilitare la verifica dei numeri estratti, bisognerà sempre poi cliccare sulla cella!

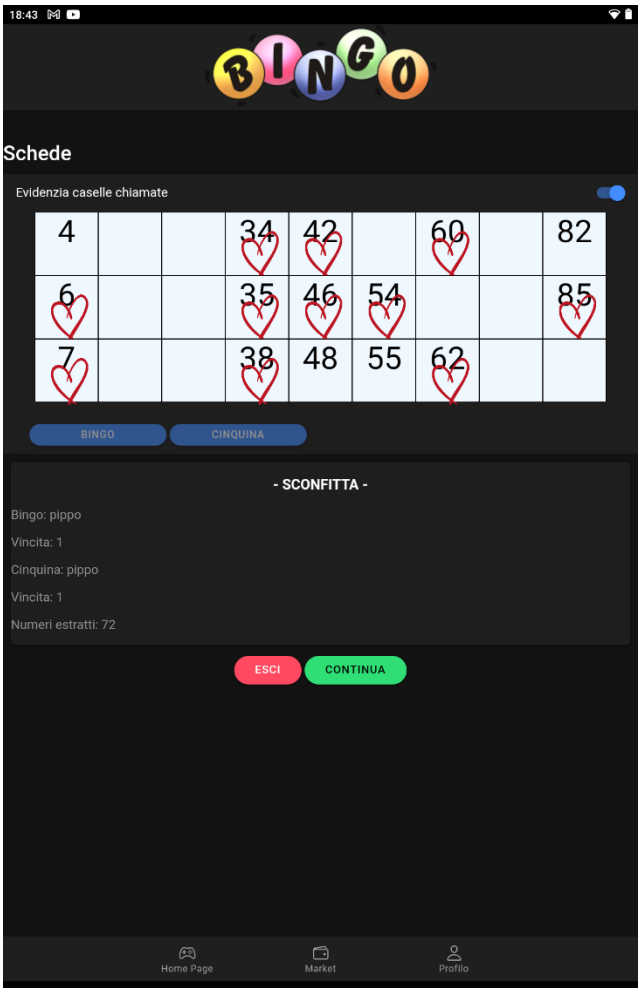
Per poter iniziare una partita il proprietario deve cliccare su start partita, per poterla iniziare devono esserci almeno 2 giocatori, se si clicca senza che il numero minimo sia rispettato il proprietario verrà avvisato da un alert che ricorderà il numero minimo di partecipanti necessari.

Una volta che viene fatta iniziare una partita comparirà il tabellone e inizierà l'estrazione dei numeri, mentre scompariranno la chat, i dati della partita e la possibilità di comprare una nuova scheda e non potranno più entrare in partita nuovi giocatori.



I bottoni per il bingo e la cinquina sono di default disabilitati e si abilitano solo quando sono presenti le condizioni per la vincita: nel caso della cinquina bisogna avere **segnato** tutta una riga, mentre per il bingo bisogna aver **segnato** tutta la cartella. ATTENZIONE: bisogna SEGNARE, non basta che i numeri vengano estratti, finché i numeri non sono segnati (le celle non sono segnabili finché il loro numero non viene estratto) non si abiliteranno i bottoni. Se qualcuno fa cinquina verrà segnalato con un fumetto con scritto "cinquina" (basterà cliccarci sopra per farlo sparire) e ovviamente non si potrà più fare cinquina.

Mentre se qualcuno fa bingo la partita finisce, il tabellone scompare e comparirà un riepilogo della partita: con riportato chi ha vinto, cosa, quanto e con quanti numeri estratti è stato fatto bingo.



Non spariranno le schede per permettere agli utenti di continuare a guardarle e dire "daiiii, me ne mancava solo uno".

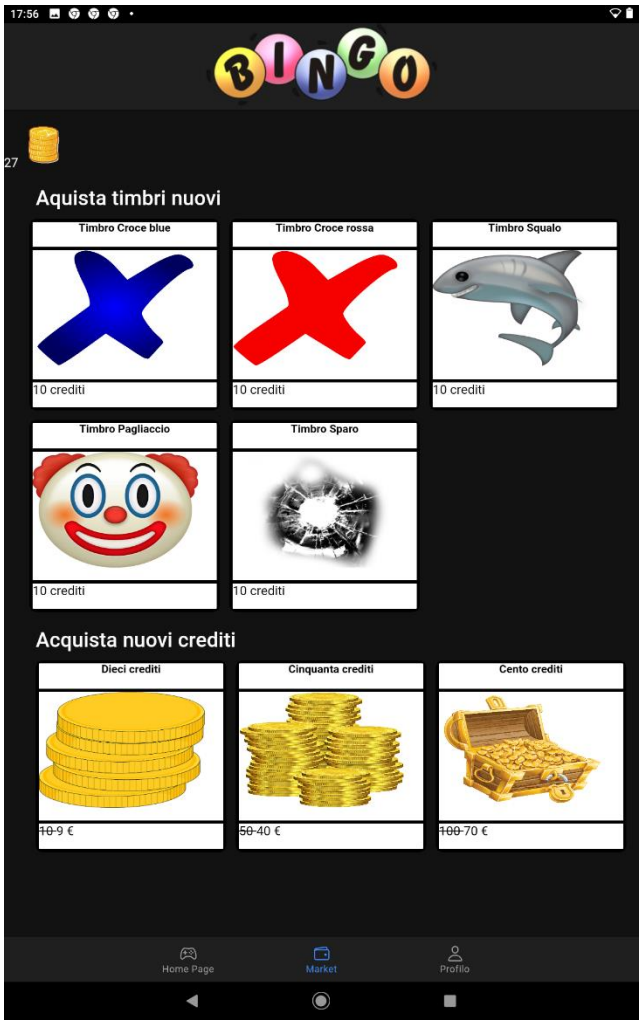
Finita la partita sarà possibile uscire o cliccare su continua per giocare un'altra partita con le stesse persone. Se il proprietario clicca su esci non sarà possibile continuare la partita e si verrà reindirizzati alla home page insieme a un alert che informerà del motivo della chiusura della partita. Una volta che il proprietario fa finire la partita questa verrà cancellata dal db.

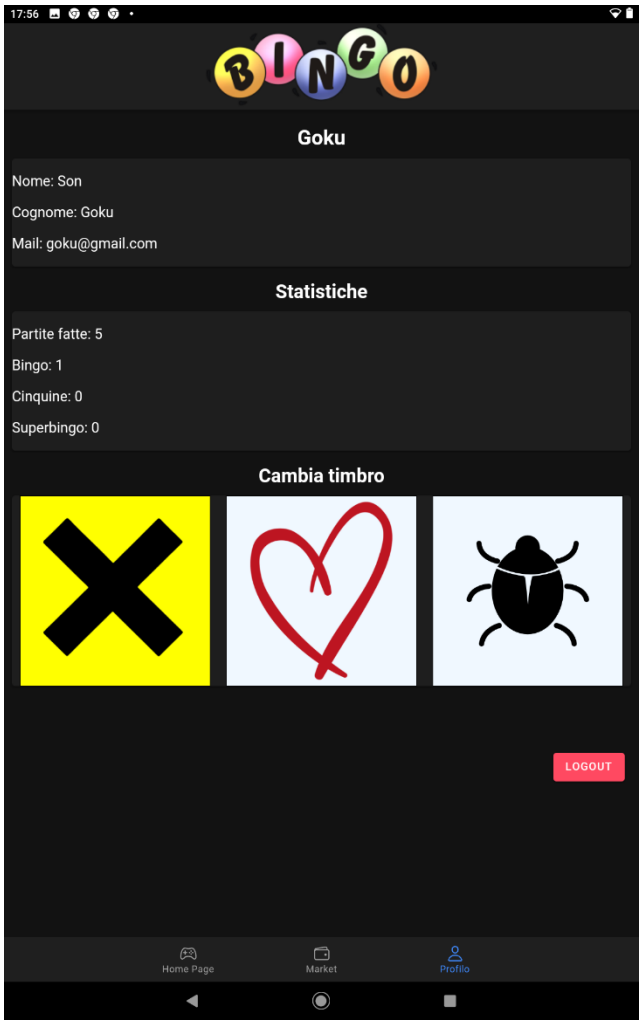
Il bottone “continua” funzionerà per i giocatori non proprietari solo dopo che il proprietario ci avrà cliccato.

Altra funzionalità della nostra app è il Market. Il market è raggiungibile tramite il tabs Market in basso al centro, oppure cliccando l’immagine dei soldi affianco i crediti nella home page.
In questa pagina è possibile comprare nuovi timbri o crediti, i timbri si comprano con i crediti mentre i crediti si comprano con i “soldi”.

L’ultimo tabs è quello che indirizza al profilo, in questa pagina è possibile visualizzare i propri dati e le statistiche di gioco: partite fatte, quante volte abbiamo fatto bingo, quante volte cinquina e quante volte il super bingo, che è sempre bingo, ma per farlo non devono essere stati estratti più di 50 numeri ed essendo così difficile, nel caso lo si riesca a fare, al montepremi del bingo vengo aggiunti di default 100 crediti. Sempre qui è possibile cambiare il proprio timbro, il timbro che attualmente è in uso è quello evidenziato di giallo.

Come ultima cosa in questa pagina è possibile effettuare il logout.





Tecnologie usate

Per la realizzazione di questo progetto abbiamo deciso di creare un'app ibrida sviluppata con Angular e Ionic.
La motivazione principale che ci ha fatto sviluppare un'app ibrida con questo framework è il fatto che è Cross-platform: Ionic consente agli sviluppatori di creare app eseguibili su più piattaforme, tra cui iOS, Android e Web, con un'unica base di codice.
Un altro punto a favore per Angular è il fatto che utilizzi il pattern architetturale Model-view-controller (MVC), Angular implementa questo pattern tramite la sua struttura a componenti, dove ogni componente funge da modello, vista e controller allo stesso tempo, gestendo i propri dati e la propria logica. Ogni componente viene sviluppato come un elemento separato e autonomo. Questi componenti possono essere riutilizzati in diverse parti dell'applicazione, rendendo il codice più organizzato, modulare e facile da gestire.

Come database abbiamo utilizzato Firebase, perché facile da implementare e utilizzare, non è necessario configurare server o gestire la scalabilità, e in più è un Real-time database, ovvero un database NoSQL in tempo reale, che significa che i dati vengono automaticamente sincronizzati su tutti gli elementi in ascolto. Questo rende facile costruire app che supportano la collaborazione in tempo reale tra gli utenti.
Infatti, Firebase offre due metodi per prendere i dati dal database: il metodo get(child()) che serve per prendere i dati dal DB in quel momento, come se fosse uno screenshot e il metodo onValue, con questo metodo i dati richiesti vengono restituiti ogni volta che vengono modificati, in questo modo, per esempio, riusciamo a leggere i numeri appena estratti dal bossolo.
Abbiamo 4 tipi di oggetti: chat, partita, timbri, users.
In chat ogni partita è composta da un array di messaggi, l'id dei messaggi vengono assegnati in automatico da Firebase, e una volta finita una partita l'istanza relativa a una partita viene cancellata.
Ogni partita è composta:

- Codice
- "iniziata": variabile boolean che indica quando una partita è iniziata o finita che serve per gestire l'accesso da parte di nuovi giocatori alla partita
- numPartecipanti
- proprietario
- pubblica: variabile boolean che se settata a false vuol dire che l'accesso a questa è possibile solo se si è a conoscenza del codice
- serveOnline: variabile boolean che quando viene settata a false, tramite il metodo chechkServer(), avvisa i giocatori dell'uscita del server e la fine della partita
- datiPartita: è un oggetto che a sua volta contiene:
    - bingo: variabile boolean di default false, che viene settata a true quando qualcuno fa bingo
    - cinquina: variabile boolean di default false, che viene settata a true quando qualcuno fa cinquina
    - montepremi: variabile che viene aggiornata con l'entrata e l'uscita dei giocatori

    - numeriEstratti: variabile che indica quanti numeri sono stati estratti, viene incrementata ad ogni nuova estrazione
    - premioBingo: viene calcolato in base al numero delle cartelle acquistate
    - premioCinquina: viene calcolato in base al numero delle cartelle acquisite
    - ultimoNumero: variabile dove viene salvato il numero estratto, che viene letto dai giocatori tramite il metodo ascoltaNumero()

Anche per l'oggetto partita, una volta che questa finisce viene cancellata.

Ogni timbro è composto da un id, un numero primo, da un nome e un url che serve per prendere l'immagine.

Sempre in Users sono presenti:

- codiceTimbri: id del timbro selezionato
- cognome
- crediti
- mail
- nome
- password
- timbri: calcolato come moltiplicazione degli id dei timbri che si hanno acquistato, essendo tutti numeri primi il risultato è un numero unico, questo ci permette di risalire ai timbri senza doverli memorizzare in un array, ma in un'unica variabile
- username
- stats: oggetto contenente le statistiche
    - bingo
    - cinquine
    - partiteFatte
    - superBingo

Algoritmi

Mostrare tutto il codice in un documento è impossibile, quindi abbiamo deciso di spiegare le parti che abbiamo ritenuto più interessanti, ovvero:

- Funzionamento di una partita ad alto livello
- Bossolo
- Generatore schede

Funzionamento di una partita ad alto livello

Un qualsiasi giocatore che abbia almeno un credito può creare una stanza per giocare.
Il proprietario è l'unico che comunica con Bossolo-service, che, come vedremo in seguito, si occupa dell'estrazione dei numeri e di comunicarlo al database.
Il proprietario durante la partita viene trattato come un giocatore normale, prendendo i numeri dal DB e non dal bossolo.

Le entità fondamentali per la partita sono la **page Partita**, che si occupa di gestire la pagina della partita e mettere in scena tutti i component (come la chat, il tabellone, le schede ecc), il **component tabellone** che si occupa di mostrare e colorare i numeri che sono usciti, e il **component schede** che si occupa di contenere più **component scheda**.

Page partita: è la pagina che vediamo durante la partita, il primo compito che svolge è mostrare la pagina al giocatore, e a seconda che il giocatore sia il proprietario oppure no, mostra pagine diverse, aggiungendo al proprietario i tasti per eliminare o far partire la partita. Inoltre, questa pagina rimane sempre in ascolto sulle modifiche nel database attraverso vari Subscription:

checkSub: permette di mettere in ascolto i giocatori per essere avvisati nel caso in cui il server abbandoni la partita.

bingoSub: ascolta se qualcuno ha fatto bingo, quando succede attiva la schermata di vittoria mostrando i vari vincitori di bingo e cinquina e le loro vincite.

cinquinaSub: ascolta se qualcuno ha fatto cinquina, quando succede mostra una immagine che sembra l'urlo di un fumetto con scritto cinquina, cliccandoci viene tolto.

aggiornaDatiSub: serve per rimanere aggiornato per il numero di partecipanti e il montepremi, quando inizia la partita si spegne.

schede/a: è quello che gestisce tutte i componet scheda, ovvero la cartella di gioco, questa distinzione fa si che una scheda si occupi solo di ascoltare i numeri che escono, tramite il metodo ascoltaNumero(), e cambiare lo stato della casella (la casella è un quadrato di una scheda, questa può essere: vuota, numero, estratta, segnata). La scheda si occupa di controllare se ha fatto cinquina o bingo e notificarlo a schede con un evento, quando succede schede abilita il tasto cinquina e/o bingo. Inoltre, schede si occupa di comprare nuove schede e abilitare gli "aiuti" per il giocatore.

tabellone: questo component si occupa di ascoltare il numero estratto dal database e mostrarlo, inoltre mostra il tabellone di gioco (ovvero tutti i numeri) e li colora di rosso man mano che questi vengono estratti.

Tutti questi metodi che ascoltano il DB sono dentro il servizio partita-db.service.ts, che si occupa di tutte le chiamate al DB durante la partita, tra cui tutte le chiamate al DB con onValue che ritorna un Observable .

Bossolo

Il bossolo nel bingo reale è l'estrattore di palline, è il contenitore di tutte le palline con le quali si gioca in una partita di bingo. Il bossolo gira per mescolare le palline fino a che, ad un certo momento, si ferma e viene fuori una pallina.
Per simulare questo comportamento abbiamo creato il servizio Bossolo-service.ts
È strutturato con un timer che chiama il metodo estrazione(), questo metodo serve per estrarre un numero con il metodo estraiNumero() e poi lo comunica al DB, cambiando il valore di ultimoNumero, e incrementa il numero dei numeri estratti.

L'estrazione di un numero funziona in questo modo:

- Abbiamo la variabile bossolo, che è un array di numeri che devono ancora essere estratti, all'inizio ovviamente sarà pieno quindi avrà i numeri da 1 a 90.
- Estrae un numero randomico R da 0 alla lunghezza di bossolo. Estraiamo il numero che si trova in bossolo[R], ritornandolo e togliendolo dall'array bossolo.

Il timer che chiama estrazione va avanti fino a quando non viene chiamato stopTimer() oppure finiscono i numeri dentro bossolo.

Generatore schede

Questa parte non è fondamentale per la comprensione della struttura del progetto, ma ci sembra comunque interessante mostrare come abbiamo deciso di generare le schede.

Per la creazione delle schede abbiamo creato un servizio: generatore-scheda.ts.
La generazione di una cartella la possiamo dividere in 3 punti:

- Prendere i numeri;
- Sistemare le caselle bianche
- Ritornare i valori

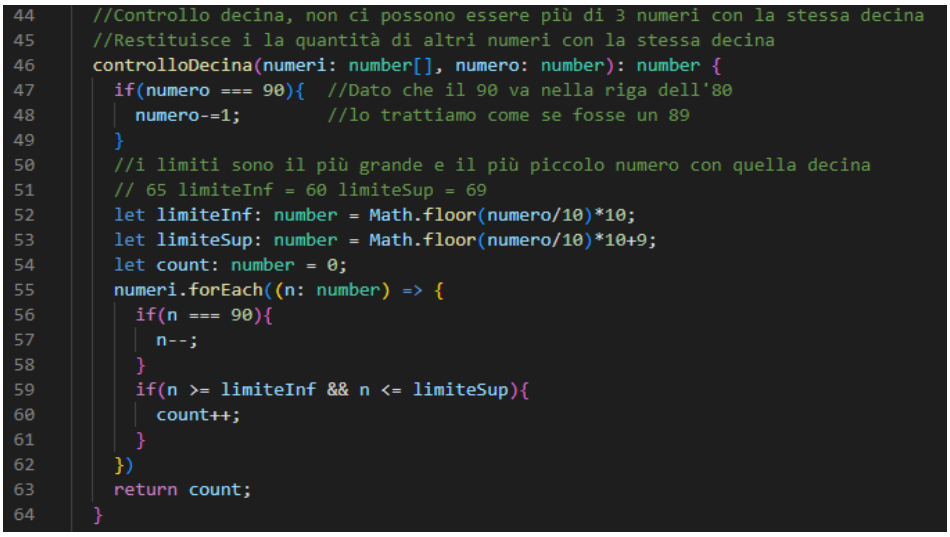
Prendere i numeri:
I numeri della cartella vengono estratti in maniera casuale ma ci sono delle regole:

- Non ci possono essere due numeri uguali (banale)
- Non ci possono essere più di 3 numeri con la stessa decina, inoltre, il 90 fa parte della decina dell'80.

Quindi, per ogni numero estratto, facciamo controlli su queste due regole.

```
//Estrazione dei numeri
getNumeriCasuali(): number[]{
  let numeri: number[] = [Math.floor(Math.random() * (90) + 1)]; //L'array deve partire già con un numero in memoria
  for(let i=0; i<14; i++){
    let n = Math.floor(Math.random() * 90 + 1);
    if(this.controlloPresenza(numeri, n) && this.controlloDecina(numeri, n) < 3){
      numeri.push(n);
    } else {
      console.log("BLOKKA", n)
      i--;
    }
  }
  return this.ordinaNumeri(numeri);
}
```

Per la prima regola è facile, basta controllare che il numero generato non sia già presente nell'array numeri tramite il metodo controlloPresenza().
La seconda è un attimo più complicata:



```
//Controllo decina, non ci possono essere più di 3 numeri con la stessa decina
//Restituisce i la quantità di altri numeri con la stessa decina
controlloDecina(numeri: number[], numero: number): number {
  if(numero === 90){  //Dato che il 90 va nella riga dell'80
    numero-=1;        //lo trattiamo come se fosse un 89
  }
  //i limiti sono il più grande e il più piccolo numero con quella decina
  // 65 limiteInf = 60 limiteSup = 69
  let limiteInf: number = Math.floor(numero/10)*10;
  let limiteSup: number = Math.floor(numero/10)*10+9;
  let count: number = 0;
  numeri.forEach((n: number) => {
    if(n === 90){
      n--;
    }
    if(n >= limiteInf && n <= limiteSup){
      count++;
    }
  })
  return count;
}
```

Innanzitutto, dato che il 90 lo dobbiamo trattare come se facesse parte della decina del 8, lo decrementiamo di uno . Calcoliamo quelli che abbiamo denominato limiteInferiore e limiteSuperiore, i limiti sono il più piccolo e il più grande numero appartenenti alla decina presa in considerazione (es. 65: limiteInf = 60 e limiteSup = 69), vengono calcolati facendo l'arrotondamento di numero/10 e lo moltiplichiamo per 10, per esempio:
65 / 10 = 6.5 troncato diventa 6 per 10 fa 60, il limite superiore è lo stesso ma più 9.

Successivamente scorriamo tutti i numeri che abbiamo già salvato e controllo quanti numeri fanno parte dell'intervallo [limiteInf, limiteSup], se un numero ne fa parte incremento una variabile count, alla fine del metodo ritorno count. Se count è maggiore di 3 non posso aggiungere il numero calcolato all'array di numeri.
Finito il metodo richiamo il metodo per avere tutti i numeri in ordine.

Una volta ottenuti tutti i numeri parte la seconda parte dell'algoritmo, ovvero sistemare le caselle bianche, per farlo dividiamo questo problema in due sottoproblemi:

- Inserire le caselle bianche
- Ordinare le caselle nella cartella.

Per inserire le caselle bianche utilizziamo il metodo aggiungiCaselleVuote()

```
87  //Aggiungi gli zeri
88  aggiungiCaselleVuote(numeri: number[]): number[]{
89    let indexNumeri = 0;
90    let count = 3;
91    let numeriConZero: number[] = [];
92    //Scorro tutti i numeri
93    numeri.forEach((n:number, index: number) => {
94      if(n===90){    //Se è 90 lo aggiungiamo, sicuramente è l'ultimo numero
95        numeriConZero.push(90);
96      } else {
97        //Controllo se sto guardando un altra decina (indexNumeri)
98        if(Math.floor(n/10) !== indexNumeri){
99            let zeri = count + (3 * (Math.floor(n/10) - indexNumeri - 1))
100           for(let i=zeri; i>=1; i--){
101             numeriConZero.push(0);
102           }
103           numeriConZero.push(n);
104           indexNumeri = Math.floor(n/10);
105           count=2;
106       } else {
107         numeriConZero.push(n);
108         count--;
109       }
110     }
111   })
112
113   //Aggiunta zeri finali
114   if(numeriConZero.length !== 27){
115     let zeriFinali = 27 - numeriConZero.length;
116     for(let i = 0; i < zeriFinali; i++){
117       numeriConZero.push(0)
118     }
119   }
120   return numeriConZero;
121 }
```

Le variabili fondamentali di questo metodo sono tre:

- *indexNumeri*: che sarebbe la decina che stiamo controllando;
- *count*: il numero di caselle vuote che dobbiamo aggiungere;
- *numeriConZero*: che è l'array dove salviamo tutti i 27 numeri.

Scorriamo tutti i numeri, facciamo un controllo sulla decina per verificare se sia la stessa di *indexNumeri*, se è la stessa aggiungo il numero a *numeriConZero* e decremento *count*, se invece è diversa dobbiamo aggiungere delle caselle vuote che rappresentiamo con zero. Il numero di zeri da aggiungere per questa decina è il valore di count più 3 * (la decina che stiamo controllando - indexNumeri - 1), questo perché si potrebbe saltare una decina, per capire meglio facciamo un esempio:

l'ultimo numero che abbiamo aggiunto è il 22, quindi *indexNumeri* sarà 2, lo aggiungiamo e decrementiamo count, il prossimo numero che aggiungiamo sarà 30, fa parte della 3° decina, quindi dobbiamo aggiungere degli zeri, aggiungiamo il valore di *count* + (3 * (3 - 2 - 1)) quindi solo il valore count, questa formula serve se un prossimo numero salta la decina, facciamo l'esempio ora che il prossimo numero da aggiungere sia 56, fa parte della 5° decina, e aggiungiamo *count* + (3 * (5 - 3 - 1)), questa formula aggiunge gli zeri per la 4° che non c'è.

I controlli appena spiegati vengono eseguiti su tutti gli elementi dell'array numeri tranne sul numero 90 (nel caso ci sia), perché questo andrà sicuramente infondo ed è inutile quindi farci dei controlli (ricordiamo che l'array numeri è anche già stato ordinato).

Dopo aver aggiunto gli zeri aggiungiamo il numero che stiamo controllando e aggiorniamo *indexNumeri* e *count.*
Alla fine, aggiungiamo degli zeri in fondo, se ce ne mancano, fino ad arrivare a 27 numeri.

Passeremo da avere un array [2, 4, 14, 21, 27, 28, 38, 41, 46, 50, 57, 71, 78, 79, 81] ad avere un array [2, 4, 0, 14, 0, 0, 21, 27, 28, 38, 0, 0, 41, 46, 0, 50, 57, 0, 0, 0, 0, 71, 78, 79, 81, 0, 0].

Ora dovremmo sistemarli, perché fatto così la scheda ci rimane così.

2	14	21	38	41	50		71	81
4		27		46	57		78	
		28					79	

Ogni riga deve avere 5 caselle con numeri (e di conseguenza 4 caselle bianche). Per sistemare la caselle bianche usiamo il metodo sistemaBianche().

[services/generatore-cartella.service.ts righe: 155-209]

Prima di tutto sistemiamo i numeri dell'array in una matrice identica alla cartella sopra e contiamo il numero di caselle vuote nell'ultima riga.
Partiamo a sistemare la cartella sempre dall'ultima riga, mentre la scelta della colonna da cui partire viene fatta casualmente, per rendere la creazione della cartella il più casuale possibile. Se l'ultima riga nell'indice appena estratto ha zero proviamo a scambiarlo con il numero della riga sopra (matrice[1]), se anche quello è zero proviamo con la prima, se anche quella è vuota aggiungiamo un giro al for, perché questo è andato a vuoto e quindi la nostra riga ha ancora lo stesso numero di caselle vuote.
Alla fine di questo pezzo, avremo sistemato l'ultima riga e avremo la cartella così:

2	14	21	38	41	50		71	
4		27					78	
		28		46	57		79	81

Facciamo la stessa cosa ma per la seconda riga scambiandola solo con la prima, alla fine di tutto avremo una cartella completa.

2	14	21		41			71	
4		27	38		50		78	
		28		46	57		79	81

Ultimo passaggio di questo servizio sarà estrarre i dati, per ritornare i dati al component che ha richiamato il servizio: ritorniamo un array con i numeri e gli 0 nell'ordine precedentemente fatto, solo i numeri e le tre cinquine estratte dalla matrice.