Programmazione di dispositivi Mobili - 19/07/2024

studenti: Michele Metta e Pietro Sangermano

Cos'è?

- Un'applicazione per Android basata sulla Musica e in particolare sull'utilizzo dell'API di Spotify, per calcolare i punteggi delle squadre degli utenti.
- In particolare, l'applicazione permetterà ai giocatori di potersi registrare e sfidare 1 vs 1 in partite utilizzando le proprie carte rappresentanti artisti o brani.
- Inoltre, l'applicazione permetterà ai giocatori di potersi registrare, inviare richieste d'amicizia ad altri utenti, scambiare carte con amici e creare i propri mazzi di carte.

Situazione attuale

01 L’app è originale per l’ambito musicale, in quanto le applicazioni esistenti che le somigliano non sono molte e in particolare non fanno riferimento alla musica ma bensì si riferiscono solo ed esclusivamente allo sport.

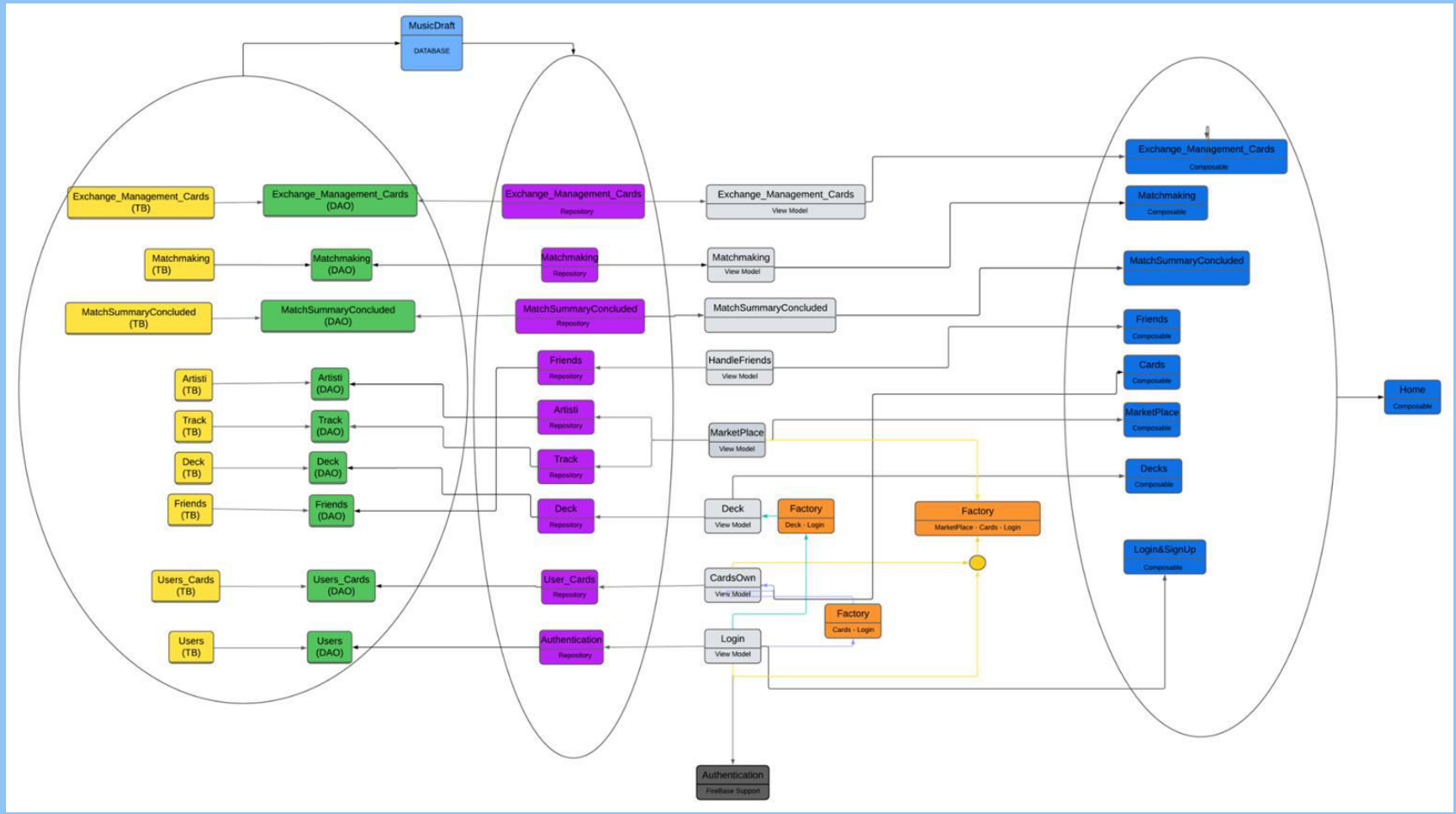
02 Dopo una ricerca sui principali stores, abbiamo notato che l’ambito musicale sia trattato per lo più con app che permettono esclusivamente la riproduzione di brani senza permettere all’utente di poter effettivamente mettere in mostra la sua conoscenza musicale e quindi confrontarsi con altri utenti.

03 La nostra app non mira a risolvere un problema specifico ma bensì ha uno scopo più ricreativo e quindi potenzialmente divertente e intrigante per gli intenditori di musica.

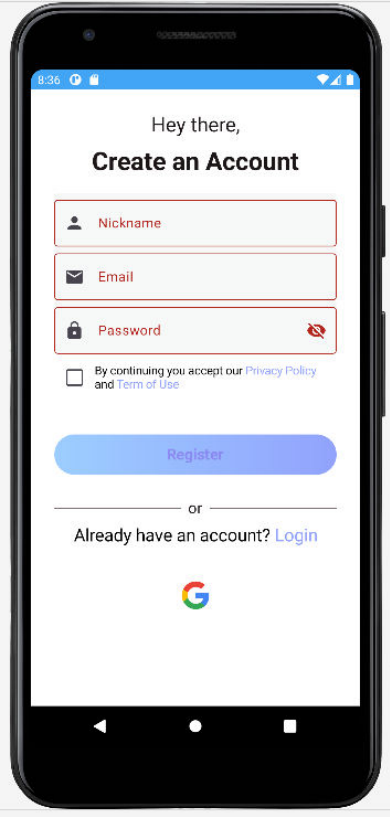
Obiettivi del Progetto

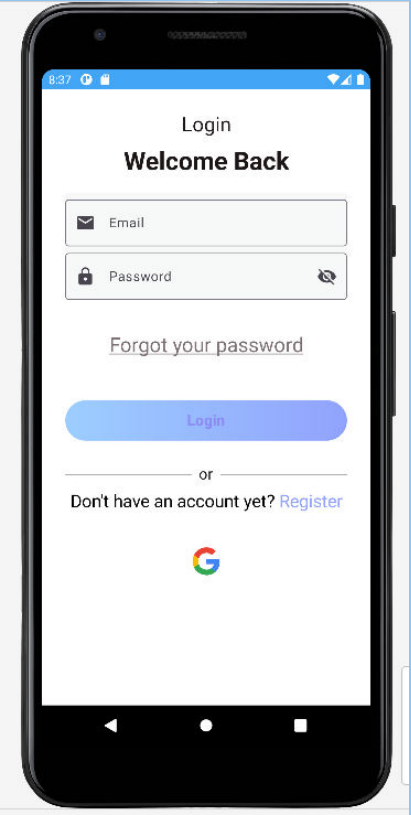
- L’interfaccia grafica di tutta l’applicazione sarà realizzata con Jetpack Compose.
- Verranno estrapolate dalla Spotify API i dati riguardanti brani e artisti.
- Implementazione di differenti View-models come livello intermedio tra interfaccia e repository.
- Utilizzo dei live data (state e flows) per permettere la ricezione e gestione dei dati tra i diversi layers (UI, ViewModel e Repository).
- Implementazione del repository e implementazione della persistenza dei dati e della gestione del matchmaking (per permettere agli utenti di sfidarsi tra loro) tramite l’utilizzo di Room.
- Gestione di un sistema di base di compravendita di carte da un marketplace e di scambio di carte fra giocatori (viene utilizzato Room e flow oltre a strutture come viewModel e Repository).
- Creazione di un sistema di mazzi con cui i giocatori possono sfidarsi tra loro.
- Utilizzo di Room per la persistenza dei dati.

Design


MusicDraft
DATABASE
Exchange_Management_Cards (TB)
Exchange_Management_Cards (DAO)
Exchange_Management_Cards
Repository
Exchange_Management_Cards
View Model
Exchange_Management_Cards
Composable
Matchmaking (TB)
Matchmaking (DAO)
Matchmaking
Repository
Matchmaking
View Model
Matchmaking
Composable
MatchSummaryConcluded (TB)
MatchSummaryConcluded (DAO)
MatchSummaryConcluded
Repository
MatchSummaryConcluded
MatchSummaryConcluded
Artisti (TB)
Artisti (DAO)
Friends
Repository
HandleFriends
View Model
Friends
Composable
Track (TB)
Track (DAO)
Artisti
Repository
Cards
Composable
Deck (TB)
Deck (DAO)
Track
Repository
MarketPlace
View Model
MarketPlace
Composable
Friends (TB)
Friends (DAO)
Deck
Repository
Deck
View Model
Factory
Deck - Login
Factory
MarketPlace - Cards - Login
Decks
Composable
Users_Cards (TB)
Users_Cards (DAO)
User_Cards
Repository
CardsOwn
View Model
Login&SignUp
Composable
Factory
Cards - Login
Users (TB)
Users (DAO)
Authentication
Repository
Login
View Model
Home
Composable
Authentication
FireBase Support

Registrazione e Login





- Nel schermata di **registrazione** l'utente potrà riempire i campi necessari che saranno validati in tempo reale.
- Una volta riempiti i campi e cliccato sulla checkbox per accettare i termini e condizioni allora il button 'Register' diventerà cliccabile.
- Appena l'utente cliccherà su 'Register' verranno eseguiti dei controlli in modo tale da garantire l'univocità sia del nickname e sia dell'e-email inseriti.
- Per il **login** l'utente potrà riempire i campi necessari e solo dopo button 'Login' sarà cliccabile.
- Abbiamo integrato all'interno dell'app il servizio esterno di **Firebase** per permettere all'utente di eseguire sia la registrazione che il login tramite un'account Google. Quindi una volta selezionato l'account, il campo nickname ed email saranno riempiti automaticamente.

User - composable Home

- Appena l'utente entrerà nell'app potrà visualizzare la schermata **'Home'** nella quale potrà vedere il suo **nickname**, i **points** accumulati fino a quel momento e il **rank**, ovvero la forza complessiva del suo insieme di carte.
- Il **rank** è calcolato come valore medio sulla popolarità di tutte le carte dell'utente.
- Nella parte centrale dello schermo potrà visualizzare un **grafico** (scrollabile in orizzontale) che riassume i points guadagnati dal giocatore corrente negli ultimi 50 giorni.
- Nella parte finale della schermata invece, potrà visualizzare la **lista** (scrollabile verticalmente) dei suoi amici (nickname e e-mail) e se questi sono online oppure offline.
- Cliccando sulle 3 barre in alto a sinistra potrà visualizzare e selezionare le diverse sezioni principali dell'app **(ModalNavigationDrawer)**. Il menù laterale potrà essere aperto o chiuso anche attraverso uno swap **(attributo gestuenabled del ModalNavigationDrawer)**.

User - tabella - dao


```kotlin
/**
 * Entità che rappresenta un utente nel sistema.
 *
 * @property id Identificativo univoco dell'utente nel database (auto-generato).
 * @property email Indirizzo email dell'utente.
 * @property nickname Nickname dell'utente.
 * @property isOnline Flag che indica se l'utente è attualmente online.
 * @property points Punteggio o punti accumulati dall'utente.
 */
micmetta
@Entity
data class User(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    var email: String,
    var nickname: String,
    var isOnline: Boolean,
    var points: Int
)
```

efficienza maggiore durante la ricerca nella schermata 'Friends'

```kotlin
/**
 * Ottiene gli utenti filtrati per nickname.
 *
 * @param nickname Nickname da cercare.
 */
micmetta
@Query("SELECT * FROM User WHERE nickname LIKE '%' || :nickname || '%'")
fun getAllUsersFilterNickname(nickname: String): Flow<List<User>?>

/**
 * Inserisce un nuovo utente nel database.
 *
 * @param user Utente da inserire.
 */
micmetta
@Insert
suspend fun insertUser(user: User)
```

```kotlin
/**
 * Verifica se esiste un utente con un dato indirizzo email nel database.
 *
 * @param email Indirizzo email da cercare.
 * @return True se esiste un utente con l'indirizzo email specificato, altrimenti False.
 */
micmetta
@Query("SELECT EXISTS(SELECT 1 FROM User WHERE email = :email)")
suspend fun doesUserExistWithEmail(email: String): Boolean

/**
 * Verifica se esiste un utente con un dato nickname nel database.
 *
 * @param nickname Nickname da cercare.
 * @return True se esiste un utente con il nickname specificato, altrimenti False.
 */
micmetta
@Query("SELECT EXISTS(SELECT 1 FROM User WHERE nickname = :nickname)")
suspend fun doesUserExistWithNickname(nickname: String): Boolean
```

AuthRepository


```
val users: List<User>? = null
val userLoggedInfo: MutableStateFlow<User?> = MutableStateFlow( value: null)
```

```
// sottoscrizione alla variabile "userLoggedInfo" sempre del repository, in questo modo
// non appena "repository.userLoggedInfo" cambierà, automaticamente cambierà anche "userLoggedInfo" del LoginViewModel:
var userLoggedInfo =  authRepository.userLoggedInfo
```

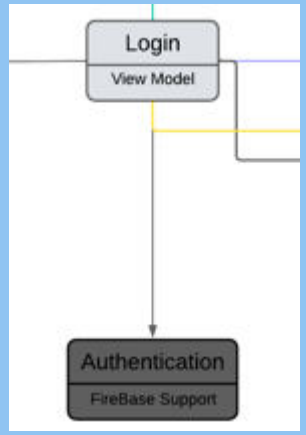
```
/**
 * Ottiene le informazioni dell'utente dal database locale per l'email specificata.
 *
 * @param email Email dell'utente di cui ottenere le informazioni.
 */
micmetta *
fun getUserByEmail(email: String) {
    Log.i( tag: "AuthRepository",  msg: "email: ${email}")
    viewModel.viewModelScope.launch { this: CoroutineScope
        withContext(Dispatchers.IO) { this: CoroutineScope
            // mi faccio restituire dalla funzione "getUserByEmail(email)" del DAO
            // il flow:
            val userFlow = dao.getUserByEmail(email)
            userFlow.collect { user ->
                userLoggedInfo.value = user
            }
        }
        Log.i( tag: "AuthRepository",  msg: "info utente prese dal DB updated: ${userLoggedInfo.value}")
    }
}
```

```
/**
 * Ottiene le informazioni dell'utente dal repository utilizzando l'email memorizzata in 'emailUserLog'.
 */
micmetta
fun getUserByEmail(){
    authRepository.getUserByEmail(emailUserLog.value)
}
```

- Non appena il **composable 'Home'** invocherà il metodo 'getUserByEmail' del 'LoginViewModel' sopra, il valore di userLoggedInfo verrà aggiornato e poichè il composable è legato alla variabile 'userLoggedInfo' del viewModel otterrà l'aggiornamento e quindi potrà mostrare i dati dell'utente (ad esempio il suo nickname e i suoi points) sull'interfaccia non appena questi saranno disponibili.

```
val infoUserCurrent by loginViewModel.userLoggedInfo.collectAsState(initial = null)
```

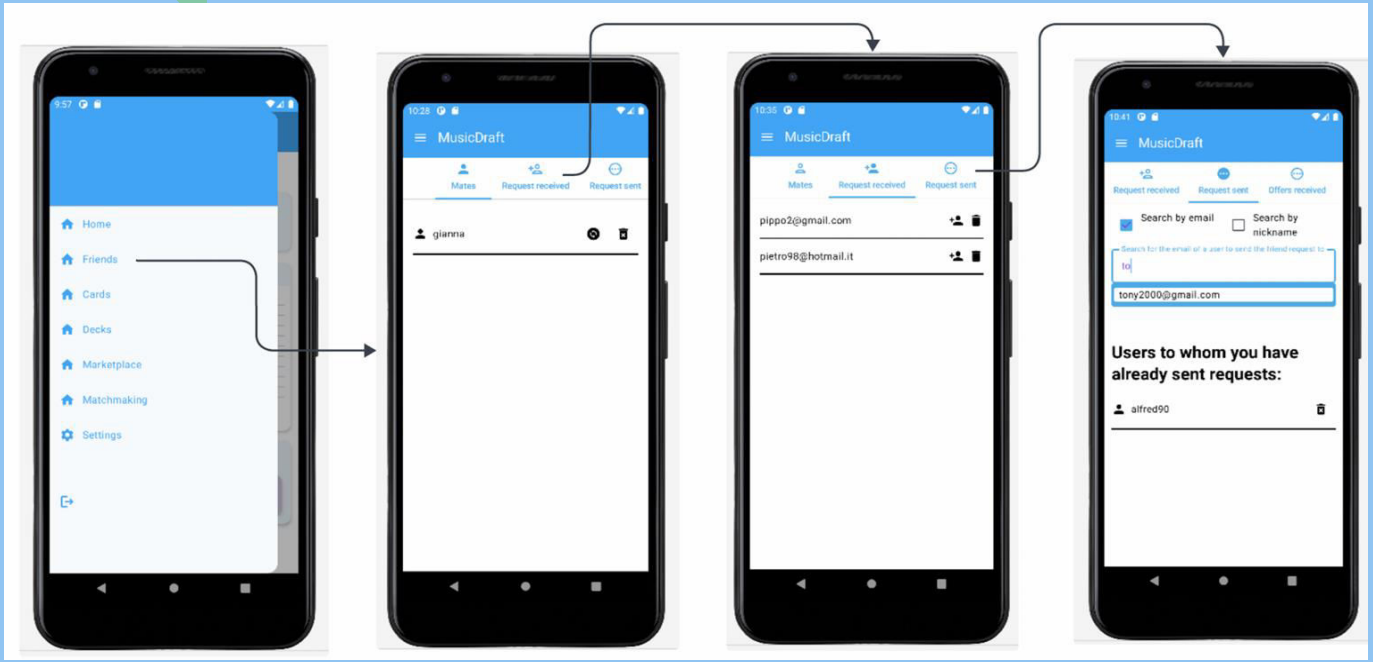
LoginViewModel



Contiene funzioni per:

1) La gestione degli eventi che riguardano l'inserimento dei caratteri da parte dell'utente nelle schermate di registrazione e login e la validazione di tutti i campi.
2) Collegamento al servizio di **Firebase** per la **creazione di un nuovo utente**, per il **logout**, per la **session management** (un utente se non esegue il logout quando aprirà l'app la volta successiva si troverà direttamente nella schermata 'Home'), **modifica password** (tramite e-mail).
3) Aggiungere e sottrarre points agli utenti.

Friends 1 - Richieste di amicizia



- Nella sezione ‘Friends’ l’utente potrà visualizzare la sua lista amici, visualizzare le richieste d’amicizia ricevute oppure inviare una richiesta d’amicizia ad un qualche utente eseguendo una ricerca per email o per nickname.
- E’ stata utilizzata la **‘ScrollableTabRow’** nel composable principale di Friends in modo tale da rendere scrollabili i **‘TabItems’** poiché data la loro quantità (5) il testo al loro interno sarebbe stato schiacciato
- La richiesta di un utente potrà essere eseguita sia per email che per nickname.
- Una volta inviata una richiesta ad un utente, il nickname di quest’ultimo apparirà in basso. (Utilizzo della MutableStateFlow<List<User>).

- Per motivi di efficienza **mentre l’utente digita** l’email (o nickname) nella barra di ricerca, automaticamente verrà eseguita una query **(richiamata tramite l’HandleFriendsViewModel)** che filtrerà gli utente presenti nel DB e automaticamente, sempre rispettando il pattern architetturale MVVM, verranno mostrati subito sotto la barra di ricerca le e-mails degli utenti già registrati nel DB.

Friends 1 - tabella - dao - repository - ViewModel


```kotlin
/**
 * Classe entità che rappresenta la gestione delle relazioni tra amici memorizzata nel database Room.
 *
 * @property id Chiave primaria generata automaticamente da Room per ogni voce della gestione relazioni.
 * @property email1 Email della prima persona coinvolta nella relazione.
 * @property email2 Email della seconda persona coinvolta nella relazione.
 * @property state Stato della relazione tra le due persone.
 */
micmetta
@Entity
data class HandleFriends(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val email1: String,
    val email2: String,
    val state: String
)
```

```kotlin
/**
 * Accetta una richiesta di amicizia specificata nel database, impostando lo stato come specificato.
 *
 * @param email1 Email dell'utente che ha ricevuto la richiesta di amicizia.
 * @param email2 Email dell'utente che ha inviato la richiesta di amicizia.
 * @param state Nuovo stato da assegnare alla relazione di amicizia ('accepted').
 micmetta
 */    @Query("UPDATE HandleFriends SET state = :state WHERE email1 = :email1 AND email2 = :email2")
suspend fun acceptRequest(email1:String, email2: String, state: String)

/**
 * Ottiene tutte le relazioni di amicizia attive di un utente specificato, in stato 'accepted'.
 *
 * @param email_user Email dell'utente di cui recuperare gli amici.
 * @return Flow di lista di HandleFriends rappresentanti gli amici dell'utente.
 micmetta
 */    @Query("SELECT * FROM HandleFriends WHERE (email1 = :email_user OR email2 = :email_user) AND state = 'accepted'")
fun getAllFriendsByUser(email_user: String): Flow<List<HandleFriends>>

/**
 * Ottiene tutte le richieste di amicizia pendenti di un utente specificato, in stato 'pending'.
 *
 * @param email_user Email dell'utente di cui recuperare le richieste pendenti.
 * @return Flow di lista di HandleFriends rappresentanti le richieste pendenti dell'utente.
 micmetta
 */    @Query("SELECT * FROM HandleFriends WHERE (email1 = :email_user OR email2 = :email_user) AND state = 'pending'")
fun getAllPendingRequestByUser(email_user: String): Flow<List<HandleFriends>>
```

Pattern **MVVM**:

- Il **Dao** esegue le queries nel DB utilizzando l'istanza della tabella Friends e restituisce gli eventuali flows.
- Il **Repository** invoca i metodi del Dao all'interno delle coroutines e quando serve aggiorna i flows.
- Il **viewModel** invoca i metodi del repository e si registra ai flows del repository.
- Il **composable** si registra ai flows del viewModel per catturare gli aggiornamenti e mostrarli sull'interfaccia .

Friends - repository - ViewModel

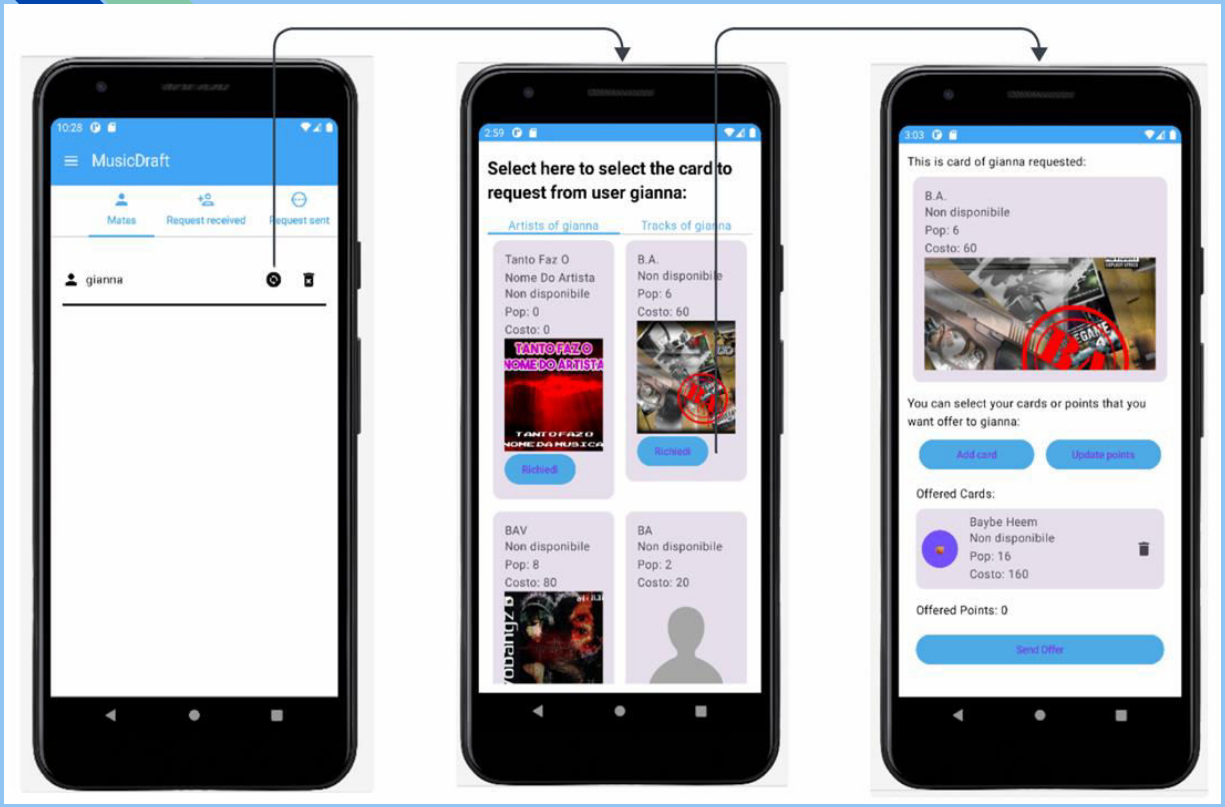

```kotlin
/**
 * Accetta una richiesta di amicizia nel database locale.
 *
 * @param email1 Email del primo utente coinvolto nella richiesta di amicizia.
 * @param email2 Email del secondo utente coinvolto nella richiesta di amicizia.
 */
micmetta
fun acceptRequest(email1: String, email2: String) {
    handleFriendsViewModel.viewModelScope.launch { this: CoroutineScope
        withContext(Dispatchers.IO) { this: CoroutineScope
            HandleFriendsdao.acceptRequest(email1, email2, state: "accepted")
        }
    }
}
```

```kotlin
/**
 * Accetta una richiesta di amicizia specificata nel database, impostando lo stato come specificato.
 *
 * @param email1 Email dell'utente che ha ricevuto la richiesta di amicizia.
 * @param email2 Email dell'utente che ha inviato la richiesta di amicizia.
 * @param state Nuovo stato da assegnare alla relazione di amicizia ('accepted').
 micmetta
 */    @Query("UPDATE HandleFriends SET state = :state WHERE email1 = :email1 AND email2 = :email2")
suspend fun acceptRequest(email1:String, email2: String, state: String)

/**
 * Ottiene tutte le relazioni di amicizia attive di un utente specificato, in stato 'accepted'.
 *
 * @param email_user Email dell'utente di cui recuperare gli amici.
 * @return Flow di lista di HandleFriends rappresentanti gli amici dell'utente.
 micmetta
 */    @Query("SELECT * FROM HandleFriends WHERE (email1 = :email_user OR email2 = :email_user) AND state = 'accepted'")
fun getAllFriendsByUser(email_user: String): Flow<List<HandleFriends>>

/**
 * Ottiene tutte le richieste di amicizia pendenti di un utente specificato, in stato 'pending'.
 *
 * @param email_user Email dell'utente di cui recuperare le richieste pendenti.
 * @return Flow di lista di HandleFriends rappresentanti le richieste pendenti dell'utente.
 micmetta
 */    @Query("SELECT * FROM HandleFriends WHERE (email1 = :email_user OR email2 = :email_user) AND state = 'pending'")
fun getAllPendingRequestByUser(email_user: String): Flow<List<HandleFriends>>
```

```kotlin
/**
 * Accetta una richiesta di amicizia.
 *
 * @param email1 Email dell'utente che ha inviato la richiesta.
 * @param email2 Email dell'utente che accetta la richiesta.
 */
micmetta
fun acceptRequest(email1: String, email2: String) {
    handleFriendsRepository.acceptRequest(email1, email2)
}
```

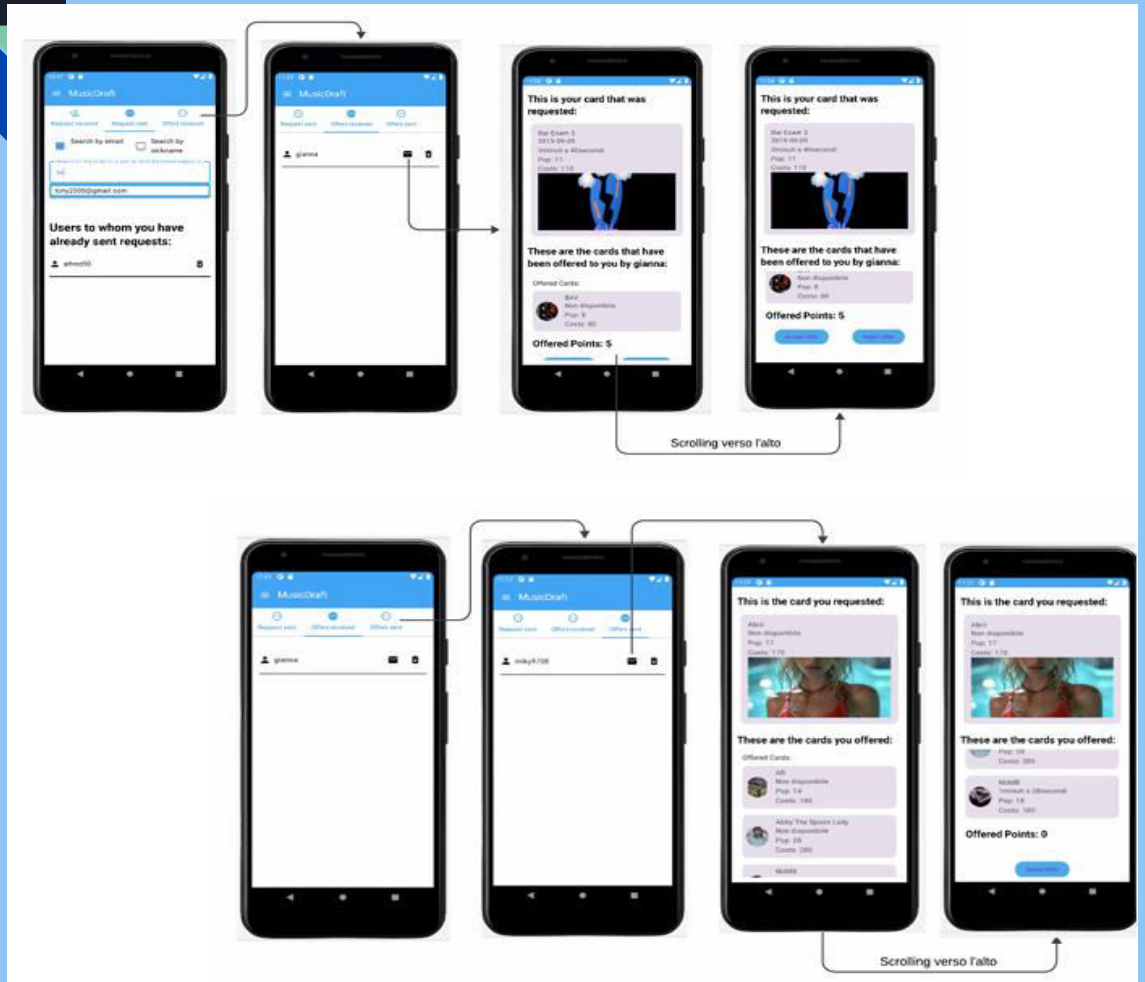
```kotlin
/**
 * Ottiene tutti gli amici di un utente.
 *
 * @param email_user Email dell'utente di cui ottenere gli amici.
 */
micmetta
fun getAllFriendsByUser(email_user: String) {
    handleFriendsRepository.getAllFriendsByUser(email_user)
}

/**
 * Ottiene tutte le richieste di amicizia in sospeso per un utente.
 *
 * @param email_user Email dell'utente di cui ottenere le richieste in sospeso.
 */
micmetta
fun getAllPendingRequestByUser(email_user: String) {
    handleFriendsRepository.getAllPendingRequestByUser(email_user)
}
```

Friends 2 - scambi carte tramite amici



- Nella sezione ‘Friends’ l’utente potrà inviare un offerta di scambio ad un altro giocatore selezionando prima la carta che vuole richiedere al suo amico e poi selezionando, qualora lo volesse, le sue carte e i points da offrire all’amico.
- La carta richiesta sarà 1 mentre le carte che il giocatore corrente potrà offrire potranno essere maggiori o uguale a 0.
- nell’offerta potranno essere inseriti un certo quantitativo di points (anche 0).
- Una volta preparato l’offerta l’utente potrà cliccare su ‘Send Offer’ in modo tale da inviare l’offerta all’amico.

Friends 3 - visualizzazione offerte inviate/ricevute



- Sempre nella sezione 'Friends' l'utente potrà **visualizzare le offerte di scambio carte che ha ricevuto** (come mostrato nelle prime quattro schermate a sinistra) oppure **visualizzare le richieste di scambio che egli stesso ha inviato ad un qualche suo amico** (come mostrato nelle successive 4 schermate).

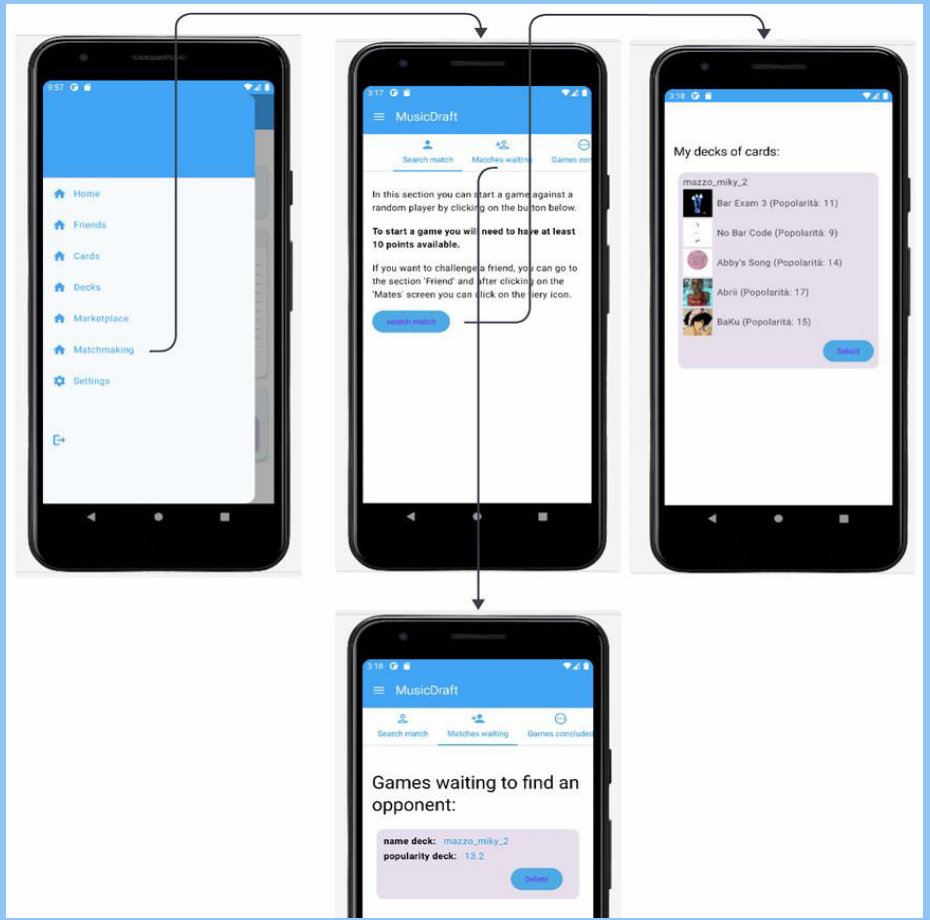
Friends 2-3
tabella - dao - repository - ViewModel

- Per **implementare lo scambio delle carte abbiamo definito una tabella, un dao, un repository e un viewModel differente** rispetto a quelli utilizzati per la gestione delle richieste di amicizia.
- Questo perchè in questo modo siamo riusciti a **modularizzare meglio l'applicazione** in modo tale da rendere più semplice non solo il **debugging** ma anche **l'implementazione di eventuali features future**.
- I collegamenti delle diverse componenti mostrate sopra sono state implementate in maniera simile a quello fatto per la gestione delle amicizie mostrata precedentemente.

Matchmaking (1)

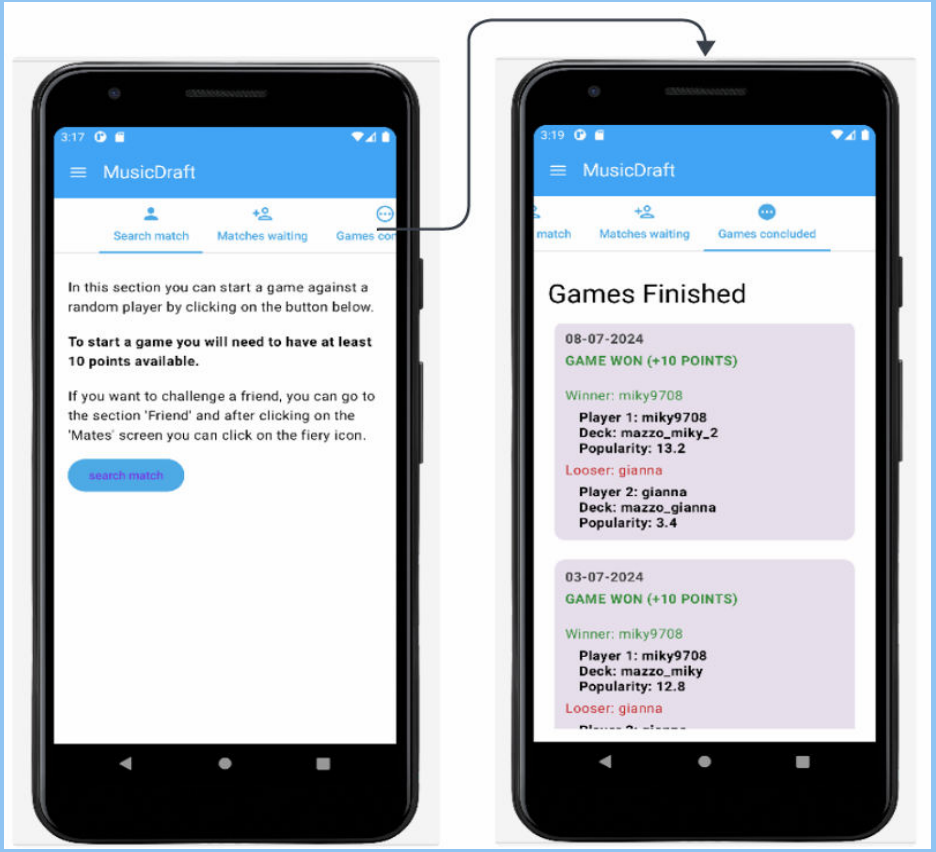
- Nella sezione **‘Matchmaking’** l’utente potrà cliccare sulla schermata ‘search match’ per sfidare un altro utente scegliendo uno dei suoi deck con il quale sfidare l’avversario. Qualora il sistema non riesca a trovare un utente immediatamente disponibile allora l’utente corrente verrà messo in attesa nel DB (sezione **‘Matches waiting’**).

- Utilizzo della ScrollableTabRow.

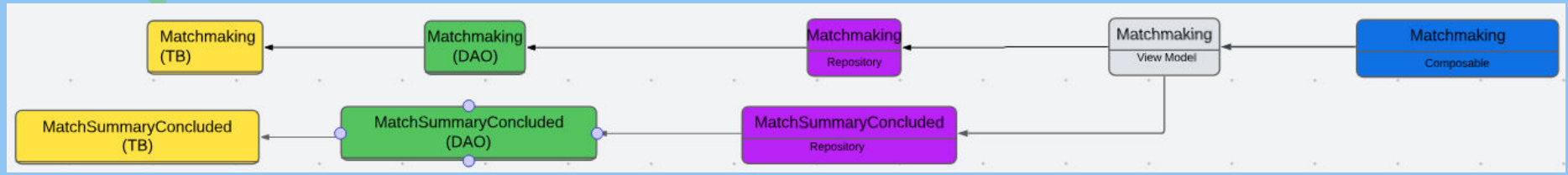


Matchmaking (2)

- L’utente inoltre, potrà cliccare sulla sotto-sezione chiamata **‘Games concluded’** per visualizzare il riepilogo delle partite che ha già giocato e che sono registrate nel DB.



Matchmaking (1-2)
tabella - dao - repository - ViewModel



- Per implementare le funzionalità del matchmaking sono state utilizzate le componenti mostrati sopra.
- **Composable principale unico** nel quale però sono stati definiti i diversi composable delle differenti sezioni di **'Matchmaking'**.
- **ViewModel unico** nel quale però sono stati definiti tutti i metodo necessari per implementare le differenti sezioni di **'Matchmaking'**.
- In particolare, il **repository 'Matchmaking'** fa riferimento al dao e alla tabelle che servono per eseguire la ricerca di una partita e all'eventuale attesa di un avversario (Matchmaking (1)), mentre il **repository 'MatchSummaryConcluded'** fa riferimento al dao e alla tabella che servono per memorizzare e gestire i risultati delle partite mostrati nella sezione 'Games concluded' (Matchmaking (2)).
- Migliore separazione delle funzionalità.
- I collegamenti delle diverse componenti mostrate sopra sono state implementate in maniera simile a quello fatto per le schermate di Friends.

Tabelle - Matchmaking - MatchSummaryConcluded

```kotlin
/**
 * Tabella del database che conterrà tutte le informazioni riguardanti i due giocatori per il quale
 * il sistema ha trovato un match.
 *
 */
micmetta
@Entity
data class Matchmaking(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    var nickname1: String,
    var nickname2: String,
    var nameDeckU1: String,
    var nameDeckU2: String,
    var popularityDeckU1: Float,
    var popularityDeckU2: Float
)
```

```kotlin
/**
 * Tabella del database che conterrà tutte le informazioni riguardanti i riepiloghi delle partite giocate dall'utente loggato.
 */
micmetta
@Entity
data class MatchSummaryConcluded(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    var dataGame: String,
    var nickWinner: String,
    var nickname1: String,
    var nickname2: String,
    var nameDeckU1: String,
    var nameDeckU2: String,
    var popularityDeckU1: Float,
    var popularityDeckU2: Float,
)
```

Matchmaking - Repository e Dao

```kotlin
/**
 * Aggiorna la variabile 'matching' tramite l'invocazione della funzione getAllMatchesWaiting del MatchmakingDao.
 * In questo modo si riesce ad ottenere l'aggiornamento di tutte le partite (con un max di 100) nelle quali
 * ci sono utenti che sono in attesa di trovare un avversario.
 *
 * @param nicknameUserCurrent Nickname dell'utente per il quale si vuole eseguire l'aggiornamento.
 */
fun getAllMatchesWaiting(nicknameUserCurrent:String) {
    viewModel.viewModelScope.launch { this: CoroutineScope
        withContext(Dispatchers.IO) { this: CoroutineScope
            val match = dao.getAllMatchesWaiting(nicknameUserCurrent)
            match.collect { matchFound ->
                matching.value = matchFound
                Log.i( tag: "MatchmakingRepository",  msg: "matching.value: ${matching.value}")
            }
        }
    }
}

/**
 * Aggiorna la variabile 'matchesConcludedByCurrentUser' tramite l'invocazione della funzione getAllMatchesWaitingBy
 * In questo modo si riesce ad ottenere l'aggiornamento di tutte le partite nelle quali l'utente loggato è in attesa
 *
 * @param nickname Nickname dell'utente per il quale si vuole eseguire l'aggiornamento.
 */
fun getAllMatchesWaitingByNickname(nickname: String) {
    viewModel.viewModelScope.launch { this: CoroutineScope
        withContext(Dispatchers.IO) { this: CoroutineScope
            val match = dao.getAllMatchesWaitingByNickname(nickname)
            match.collect { matchWait ->
                matchesWait.value = matchWait
                Log.i( tag: "MatchmakingRepository",  msg: "matchesWait.value: ${matchesWait.value}")
            }
        }
    }
}
```

```kotlin
@Dao
interface MatchmakingDao {

    /**
     * Inserisce un nuovo oggetto nella tabella Matchmaking presente nel database.
     *
     * @param matchmaking Oggetto da inserire come nuova riga nella tabella.
     */
    @Insert
    suspend fun insertNewMatch(matchmaking: Matchmaking)

    /**
     * Elimina una riga della tabella Matchmaking presente nel database.
     *
     * @param id Identificatore della riga da eliminare.
     */
    @Query("DELETE FROM Matchmaking WHERE id = :id")
    suspend fun deleteMatch(id: Int)

    /**
     * Restituisce un flow di tipo List<Matchmaking> al quale il repository chiamato MatchmakingRepository è collegato pe
     * l'aggiornamento di tutte le partite (con un max di 100) nelle quali ci sono utenti che sono in attesa di trovare u
     *
     * @param nicknameUserCurrent Nickname dell'utente per il quale si vuole eseguire l'aggiornamento.
     * @return Flow che emette una lista di Matchmaking.
     */
    @Query("SELECT * FROM Matchmaking WHERE nickname1 != :nicknameUserCurrent AND nickname2 = '' LIMIT 100")
    fun getAllMatchesWaiting(nicknameUserCurrent:String): Flow<List<Matchmaking>>

    /**
     * Restituisce un flow di tipo List<Matchmaking> al quale il repository chiamato MatchmakingRepository è collegato pe
     * l'aggiornamento di tutte le partite nelle quali l'utente loggato è in attesa.
     *
     * @param nickname Nickname dell'utente per il quale si vuole eseguire l'aggiornamento.
     * @return Flow che emette una lista di Matchmaking.
     */
    @Query("SELECT * FROM Matchmaking WHERE (nickname1 = :nickname)")
    fun getAllMatchesWaitingByNickname(nickname: String): Flow<List<Matchmaking>>
}
```

MatchmakingViewModel e Composable ‘SearchMatch’ - Matchmaking

```kotlin
/**
 * Inserisce un nuovo oggetto nella tabella Matchmaking richiamando la funzione insertNewMatch di MatchmakingReposi
 *
 * @param matchmaking Oggetto da inserire come nuova riga nella tabella.
 */
fun insertNewMatch(matchmaking: Matchmaking){
    matchmakingRepository.insertNewMatch(matchmaking)
}

/**
 * Inserisce un nuovo oggetto nella tabella MatchSummaryConcluded richiamando la funzione insertNewSummaryMatch di
 *
 * @param matchSummaryConcluded Oggetto da inserire come nuova riga nella tabella.
 */
fun insertNewSummaryMatch(matchSummaryConcluded: MatchSummaryConcluded){
    matchSummaryConcludedRepository.insertNewSummaryMatch(matchSummaryConcluded)
}

/**
 * Elimina una riga della tabella Matchmaking richiamando la funzione deleteMatch di MatchmakingRepository.
 *
 * @param id Identificatore della riga da eliminare.
 */
fun deleteMatch(id: Int){
    matchmakingRepository.deleteMatch(id)
}

/**
 * Aggiorna la variabile 'matching' legata al -MutableStateFlow<List<Matchmaking>?>- di 'MatchmakingRepository'.
 * In questo modo si riesce ad ottenere l'aggiornamento di tutte le partite (con un max di 100) nelle quali
 * ci sono utenti che sono in attesa di trovare un avversario.
 *
 * @param nicknameUserCurrent Nickname dell'utente per il quale si vuole eseguire l'aggiornamento.
 */
fun getAllMatchesWaiting(nicknameUserCurrent:String) {
    return matchmakingRepository.getAllMatchesWaiting(nicknameUserCurrent)
}
```

```kotlin
fun SearchMatch(navController: NavController, matchmakingViewModel: MatchmakingViewModel, decksViewMo  12  16  294
    NUM_POINTS_MIN: Int // numero minimo di points richiesti per giocare un game
){
    decksViewModel.init()
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    // finestre di dialogo:
    var showDialogSelectDeck = remember { mutableStateOf( value: false) }
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
        //horizontalAlignment = Alignment.CenterHorizontally,
        //verticalArrangement = Arrangement.spacedBy(16.dp)
    ) { this: ColumnScope
        Spacer(modifier = Modifier.height(16.dp))
        Text(
            text = "In this section you can start a game against a random player by clicking on the button below. \n",
            style = MaterialTheme.typography.bodyLarge,
            textAlign = TextAlign.Start
        )
        Text(
            text = "To start a game you will need to have at least ${NUM_POINTS_MIN} points available.\n",
            style = MaterialTheme.typography.bodyLarge.copy(fontWeight = FontWeight.Bold, color = Color.Black)
        )
        Text(
            text = "If you want to challenge a friend, you can go to the section 'Friend' and after clicking on the 'Mate
            style = MaterialTheme.typography.bodyLarge
        )
        Button(
            onClick = {
                /* Azione quando verrà cliccato il button */

                // 1) Scelta mazzo di gioco:
                navController.navigate( route: "selectDeck")
            },
            modifier = Modifier.padding(top = 16.dp)
        ) { this: RowScope
            Text(text = "search match")
        }
    }
}
```

MatchSummaryConcludedRepository - ConcludeMatch (1)

```
    /**
     * Aggiorna la variabile 'matchesConcludedByCurrentUser' tramite l'invocazione della funzione getAllGamesConludedByNickname del MatchSummaryConcludedDao.
     * In questo modo si riesce ad ottenere l'aggiornamento di tutte le partite giocate dall'utente loggato.
     *
     * @param nickname Nickname dell'utente per il quale si vuole eseguire l'aggiornamento.
     */
    fun getAllGamesConludedByNickname(nickname: String){
        viewModel.viewModelScope.launch { this: CoroutineScope
            withContext(Dispatchers.IO) { this: CoroutineScope
                val matches = dao.getAllGamesConludedByNickname(nickname)
                matches.collect { response ->
                    matchesConcludedByCurrentUser.value = response
                    Log.i( tag: "MatchSummaryConcludedRepository",  msg: "matchesConcludedByCurrentUser.value: ${matchesConcludedByCurrentUser.value}")
                }
            }
        }
    }
}
```

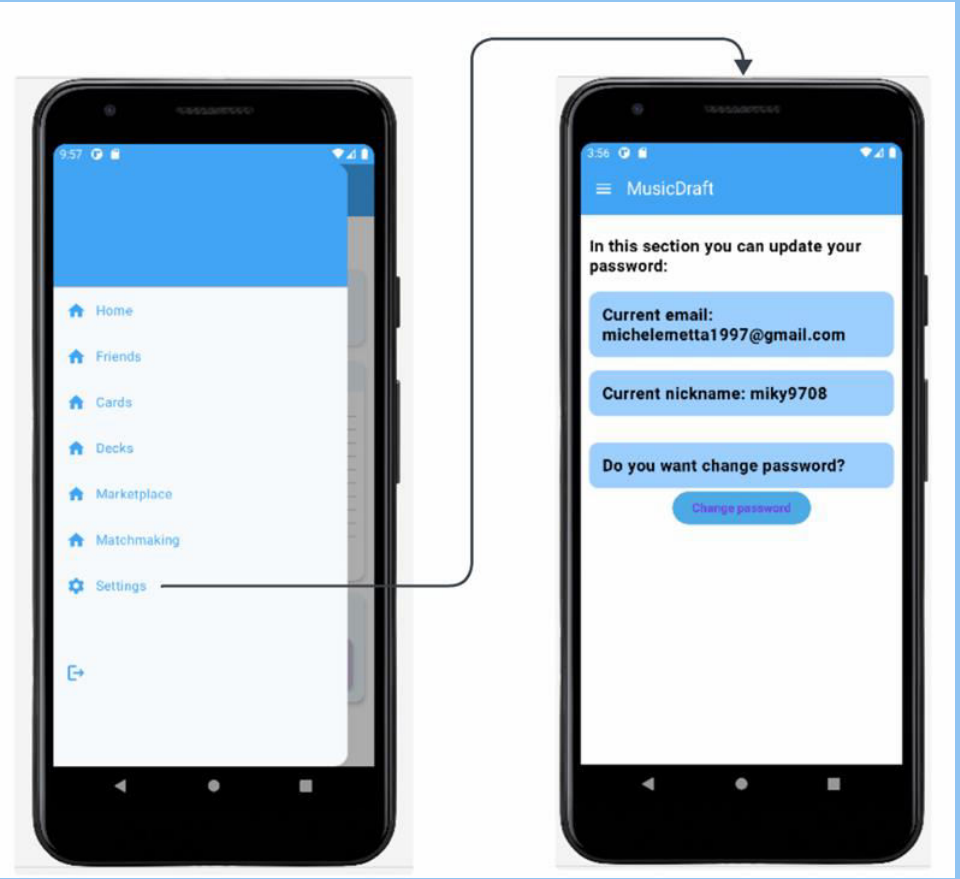
Composable ‘Games concluded’ - Matchmaking

```
@Composable                                                                    ⚠ 12  ⚠ 16  ✓ 294
fun GamesConcluded(navController: NavController, matchmakingViewModel: MatchmakingViewModel, decksViewModel: DeckViewModel, loginViewModel: LoginViewModel,
    NUM_POINTS_MIN: Int
) {
    // Ottieni le informazioni sull'utente loggato
    val infoUserCurrent by loginViewModel.userLoggedInfo.collectAsState(initial = null)

    // Ottieni le partite concluse dall'utente corrente
    val matchesConcludedByCurrentUser by matchmakingViewModel.matchesConcludedByCurrentUser.collectAsState( initial: null)

    // Carica le partite concluse dell'utente corrente
    infoUserCurrent?.let { it: User
        if (matchesConcludedByCurrentUser == null) {
            matchmakingViewModel.getAllGamesConludedByNickname(it.nickname)
        }
    }

    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxSize()
    ) { this: ColumnScope
        Spacer(modifier = Modifier.height(16.dp))
        Text( text: "Games Finished", style = MaterialTheme.typography.headlineLarge)
        Spacer(modifier = Modifier.height(16.dp))

        if (matchesConcludedByCurrentUser.isNullOrEmpty()) {
            Text(
                 text: "You haven't played any games yet.",
                style = MaterialTheme.typography.bodyLarge,
                modifier = Modifier.padding(16.dp)
            )
        } else {
            LazyColumn(
                modifier = Modifier.weight(1f)
            ) { this: LazyListScope
                items(matchesConcludedByCurrentUser!!) { this: LazyItemScope  match ->
                    GameCard(match, matchmakingViewModel, infoUserCurrent, NUM_POINTS_MIN)
                    Spacer(modifier = Modifier.height(8.dp))
                }
            }
        }
```

Settings

- L’utente può cliccare sulla sezione **‘Settings’** tramite la quale potrà visualizzare la sua **email**, il suo **nickname** e volendo **potrà modificare la password**. In quest’ultimo caso è stato utilizzato il **servizio offerto da Firebase** che permetterà all’utente di ricevere via mail il link sul quale cliccare per poter impostare correttamente la nuova password.

- Quando l’utente clicca su ‘**Change password’**, Il composable **‘Settings’** genera un evento che verrà gestito dal **‘LoginViewModel’** il quale si preoccuperà di portare la navigazione sul composable **‘ForgotPassword’** nel quale l’utente potrà digitare l’email di recupero e confermare.



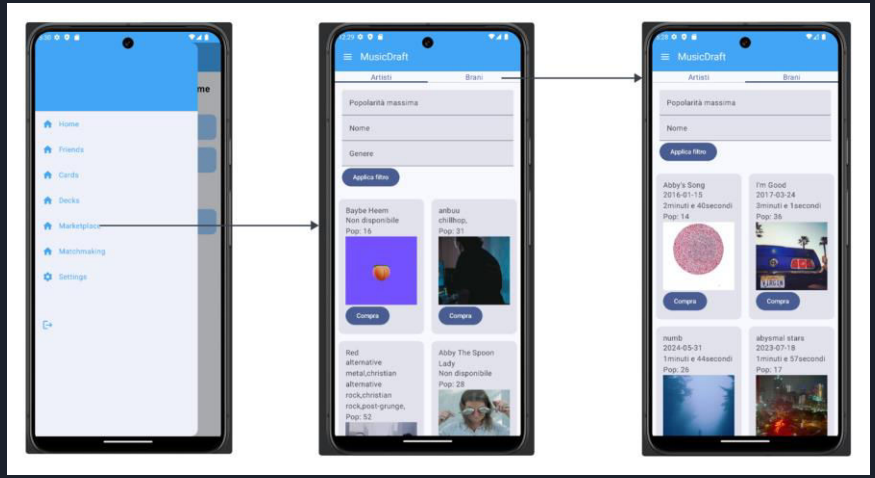
Artisti e Brani

- La nostra app estrae informazioni per artisti e brani dall'Api di spotify.
- Inizialmente si era pensato di estrarle utilizzato Spotify SDK ma purtroppo non è stato possibile per 2 motivi principalmente:
    - Per utilizzare la spotify SDK bisogna autenticarsi nell'app di spotify in modo da poter accedere ai dati e questo significava sia inserire un requisito per l'utente per l'installazione dell'app ma soprattutto un dispendio di batteria molto più oneroso del previsto poiché Spotify deve rimanere sempre in background (il che è abbastanza inutile visto che c'ho che volevamo era solo estrarre delle informazioni solo allo start dell'app).
    - Spotify SDK è scritto in java, è in beta, non viene aggiornata da un anno e a quanto viene riportato su stackoverflow in molti thread i tutorial per il collegamento al SDK sono deprecati.
- Abbiamo quindi prima cercato un wrapper kotlin per la spotify API in modo da integrarla attivamente nel codice ma purtroppo non estraeva le info che ci servivano quindi abbiamo optato per creare 2 piccoli servizi REST che estressero tutto quello che riguardava artisti o brani.
- P.S. è stato estratto un file Json dei primi artisti e brani restituiti dai servizi e usati per popolare solo per la prima volta il nostro DB (come si può notare dalla coroutine che lancia le 2 funzioni di poulate)

```
companion object {
    // marking the instance as volatile to ensure atomic access to the variable
    @Volatile
    private var INSTANCE: MusicDraftDatabase? = null

    /**
     * Ottiene un'istanza del database MusicDraft. Se il database non è stato ancora creato,
     * viene creato e popolato con dati predefiniti.
     *
     * @param context Contesto dell'applicazione.
     * @return Istanza del database MusicDraft.
     */
    fun getDatabase(context: Context): MusicDraftDatabase {
        return INSTANCE ?: synchronized( lock: this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                MusicDraftDatabase::class.java,
                 name: "musicDraftDB"
            ).addCallback(object : RoomDatabase.Callback() {
                override fun onCreate(db: SupportSQLiteDatabase) {
                    super.onCreate(db)
                    INSTANCE?.let { database ->
                        CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
                            populateArtisti(database.artistDao()!!)
                            populateTracks(database.trackDao()!!)
                        }
                    }
                }
```

MarketPlace

- Nella schermata di marketplace, l'utente potrà acquistare carte di brani e artisti.
- La schermata è divisa in 2 tab un per ognuna delle 2 categorie
- In ogni tab è possibile cercare una carta specifica:
  - negli artisti e possibile fare la ricerca per popolarità, nome e genere prodotto dall'artista
  - nei brani per popolarità e nome



MarketPlace - Tabelle

```kotlin
/**
 * Rappresenta l'entità degli artisti nel database Room.
 *
 * @param identifier Identificatore primario auto-generato.
 * @param id Identificatore dell'artista.
 * @param genere Genere dell'artista.
 * @param immagine URL dell'immagine dell'artista.
 * @param nome Nome dell'artista.
 * @param popolarita Livello di popolarità dell'artista.
 */
@Entity
data class Artisti(
    @PrimaryKey(autoGenerate = true) val identifier: Int,
    val id: String,
    val genere: String,
    val immagine: String,
    val nome: String,
    val popolarita: Int,

){}
```

```kotlin
/**
 * Rappresenta un brano musicale nel sistema.
 *
 * @property identifier Identificatore univoco del brano (auto-generato).
 * @property id ID del brano, univoco all'interno del sistema musicale.
 * @property anno_pubblicazione Anno di pubblicazione del brano.
 * @property durata Durata del brano nel formato HH:MM:SS.
 * @property immagine URL dell'immagine associata al brano.
 * @property nome Nome del brano.
 * @property popolarita Livello di popolarità del brano, rappresentato da un valore intero.
 */
@Entity
data class Track(
    @PrimaryKey(autoGenerate = true) val identifier: Int,
    val id: String,
    val anno_pubblicazione: String,
    val durata: String,
    val immagine: String,
    var nome: String,
    val popolarita: Int,

) {
}
```

MarketPlace - Dao

```kotlin
interface ArtistiDao {
    /**
     * Ottiene tutti gli artisti ordinati per popolarità in modo asincrono tramite Flow.
     */
    @Query("SELECT * FROM Artisti")
    fun getAllArtisti(): Flow<List<Artisti>>

    /**
     * Inserisce un singolo artista in modo asincrono.
     */
    @Insert
    suspend fun insertArtist(artista: Artisti)

    /**
     * Inserisce una lista di artisti in modo asincrono.
     */
    @Insert
    suspend fun insertAllArtist(artisti : Array<Artisti>)

    /**
     * Cancella un artista in modo asincrono.
     */
    @Delete
    suspend fun deleteArtist(artista: Artisti)

    /**
     * Ottiene tutti gli artisti ordinati per popolarità in modo ascendente tramite Flow.
     */
    @Query("SELECT * FROM Artisti ORDER BY popolarita ASC")
    fun getArtistiOrderedByPop(): Flow<List<Artisti>>

    /**
     * Ottiene tutti gli artisti ordinati per nome in modo ascendente tramite Flow.
     */
    @Query("SELECT * FROM Artisti ORDER BY nome ASC")
    fun getArtistiOrderedByNome(): Flow<List<Artisti>>

    /**
     * Ottiene gli artisti con una popolarità massima specificata in modo ascendente tramite Flow.
     */
    @Query("SELECT * FROM Artisti WHERE popolarita <= :maxPopolarita ORDER BY popolarita ASC")
    fun getArtistiWithMaxPop(maxPopolarita: Int): Flow<List<Artisti>>
```

```kotlin
@Dao
interface TrackDao {

    /**
     * Ottiene tutti i brani musicali presenti nel database.
     *
     * @return Flow che emette una lista di Track ogni volta che ci sono aggiornamenti nel database.
     */
    @Query("SELECT * FROM Track")
    fun getAllTracks(): Flow<List<Track>>

    /**
     * Inserisce un nuovo brano musicale nel database.
     *
     * @param track Il brano da inserire nel database.
     */
    @Insert
    suspend fun insertTrack(track: Track)

    /**
     * Inserisce una lista di brani musicali nel database.
     *
     * @param tracks Array di brani musicali da inserire nel database.
     */
    @Insert
    suspend fun insertAllTracks(tracks: Array<Track>)

    /**
     * Elimina un brano musicale dal database.
     *
     * @param track Il brano da eliminare dal database.
     */
    @Delete
    suspend fun deleteTrack(track: Track)

    /**
     * Ottiene tutti i brani musicali ordinati per popolarità crescente.
     *
     * @return Flow che emette una lista di Track ordinate per popolarità.
     */
    @Query("SELECT * FROM Track ORDER BY popolarita ASC")
```

MarketPlace - Repository

- Viene riportata solo l' artist Repository ma il trackrepository è praticamente speculare

```
class ArtistRepository(val viewModel: MarketplaceViewModel, val dao: ArtistiDao) {

    /** Lista degli artisti inizializzata a null. */
    val _artisti:List<Artisti>? = null

    /** Flusso che contiene la lista degli artisti. */
    val artisti: MutableStateFlow<List<Artisti>?> = MutableStateFlow(_artisti)


    /**
     * Elimina un artista dal database.
     *
     * @param artista L'artista da eliminare.
     */
    fun delete_artista(artista: Artisti){
        viewModel.viewModelScope.launch { this: CoroutineScope
            dao.deleteArtist(artista)
        }
    }

    /**
     * Ottiene un artista per ID dal database e aggiorna il flusso [artisti].
     *
     * @param id L'ID dell'artista.
     * @return La lista degli artisti con l'ID specificato.
     */
    suspend fun getArtistbyId(id:String): List<Artisti>? {
        viewModel.viewModelScope.launch { this: CoroutineScope
            val result = dao.getallArtistbyId(id)
            result.collect{ response->
                artisti.value = response
            }
        }
        return artisti.value
    }
```

MarketPlace - ViewModel e Composable

- Si vuole riportare 2 esempi nel marketplaceViewModel
- Nel primo si vuole mettere in evidenza l'implementazione di uno dei filtri per la tab artisti (quella track è speculare).
- Ciò su cui si vuole mettere l'accento è l'utilizzo di _filteredArtistiValue il quale è un mutableStateFlow
- Difatti questo viene utilizzato per mantenere aggiornato in real time l'elenco delle carte da mostrare nel composable.
- Per fare ciò viene utilizzata la funzione collect asState sia nel caso in cui abbiamo effettivamente filtrato sia nel caso non lo avessimo fatto e in quel caso il collect viene applicato alla lista di tutti gli artisti disponibili nel marketplace

```
fun applyArtistFilter(popThreshold: Int?, nameQuery: String?, genreQuery: String?) {
    if (popThreshold == null && nameQuery.isNullOrEmpty() && genreQuery.isNullOrEmpty()) {
        clearFilteredArtisti()
        return
    }

    val filteredArtisti = allartist.value!!.filter { artist ->
        val popFilter = popThreshold?.let { artist.popolarita <= it } ?: true
        val nameFilter = nameQuery?.let { artist.nome.contains(it, ignoreCase = true) } ?: true
        val genreFilter = genreQuery?.let { artist.genere.contains(it, ignoreCase = true) } ?: true
        popFilter && nameFilter && genreFilter ^filter
    }

    _filteredArtisti.value = filteredArtisti
}
```

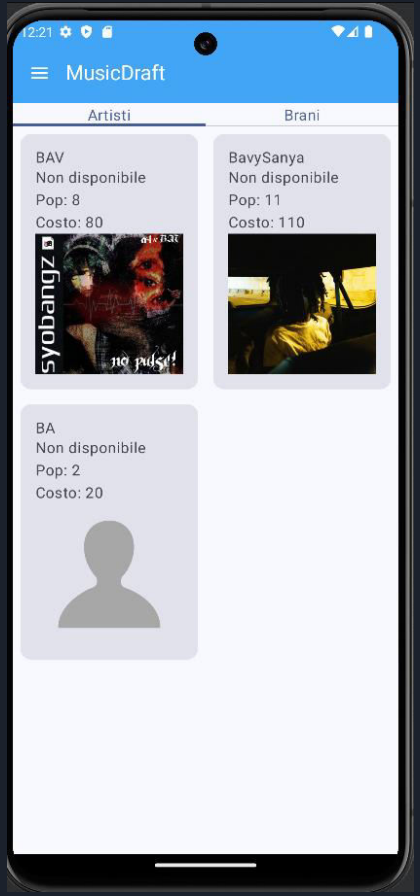
```
// Ottieni la lista degli artisti e delle tracce in base alla scheda selezionata e ai filtri applicati
val artisti by if (!(popThreshold.isNullOrEmpty() && nameQuery.isNullOrEmpty() && genreQuery.isNullOrEmpty())) {
    viewModel._filteredArtisti.collectAsState(emptyList())
} else {
    viewModel.clearFilteredArtisti()
    viewModel.allartist.collectAsState(emptyList())


}
```

MarketPlace - ViewModel

```
fun compra_artisti(artista: Artisti) {
    viewModelScope.launch { this: CoroutineScope
        val email = loginViewModel.userLoggedInfo.value!!.email
        val points = loginViewModel.userLoggedInfo.value!!.points


        // Aggiorna le liste di artisti filtrati e le carte acquistate
        val size = _filteredArtisti.value?.size

        ownRepo.getArtCardsforUser(email)

            if (points > artista.popolarita * 10) {
                if (size == null) {
                    val currentFilteredList = allartist.value
                    // Aggiorna le liste di artisti filtrati e le carte acquistate
                    val updatedFilteredList = currentFilteredList!!.toMutableList().apply { this: MutableList<Artisti>
                        remove(artista)
                    }
                    allartist.value = updatedFilteredList
                    artistRepo.delete_artista(artista)
                    cardsViewModel.insertArtistToUser(artista, email)
                    _message.value = "Artista Comprato"
                } else {
                    val currentFilteredList = _filteredArtisti.value
                    val updatedFilteredList = currentFilteredList!!.toMutableList().apply { this: MutableList<Artisti>
                        remove(artista)
                    }
                    _filteredArtisti.value = updatedFilteredList
                    cardsViewModel.insertArtistToUser(artista, email)
                    artistRepo.delete_artista(artista)
                    _message.value = "Artista Comprato"

                }
            } else {
                _message.value = "Not enough points"
            }
        }
    }
```

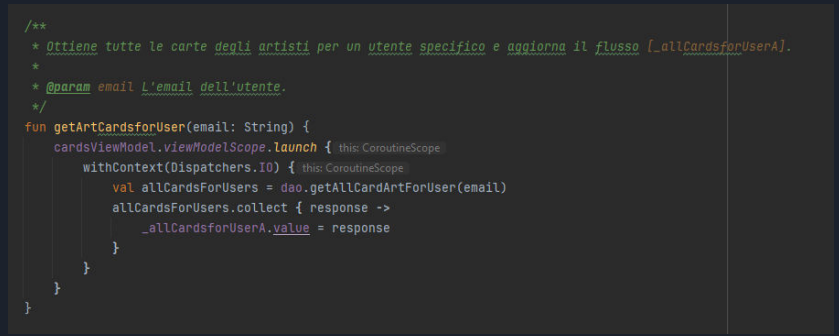
Cards

- La schermata Cards non è altro che una vista per tenere traccia di tutte le carte possedute da un utente
- La schermata è organizzata simile al marketplace in modo da dare continuity nelle scelte grafiche
- Anche qui si è mantenuta la divisione fra artisti e brani



Cards - Repository

- Su Dao,Tabelle e Repository si è voluto sorvolare poichè l'organizzazione è la medesima cambiano solo query e richiami a funzioni cis i cuole soffermare solo su una cosa (che poi viene ripetuta in tutte le repository): l'utilizzo delle coroutines
- Ogni funzione creata nei repository fanno utilizzo di coroutine per richiamare le funzioni creati nel Dao e farle runnare su un thread diverso dal main in modo da non bloccare quest'ultimo
- si fa notare come anche qui si fa uso dei flow, utilizzati come struttura di ritorno per l e funzioni nel Dao (nel caso specifico è un Flow<List<User_Cards_Artisti>>



```
/**
 * Ottiene tutte le carte degli artisti per un utente specifico e aggiorna il flusso [_allCardsforUserA].
 *
 * @param email L'email dell'utente.
 */
fun getArtCardsforUser(email: String) {
    cardsViewModel.viewModelScope.launch { this: CoroutineScope
        withContext(Dispatchers.IO) { this: CoroutineScope
            val allCardsForUsers = dao.getAllCardArtForUser(email)
            allCardsForUsers.collect { response ->
                _allCardsforUserA.value = response
            }
        }
    }
}
```

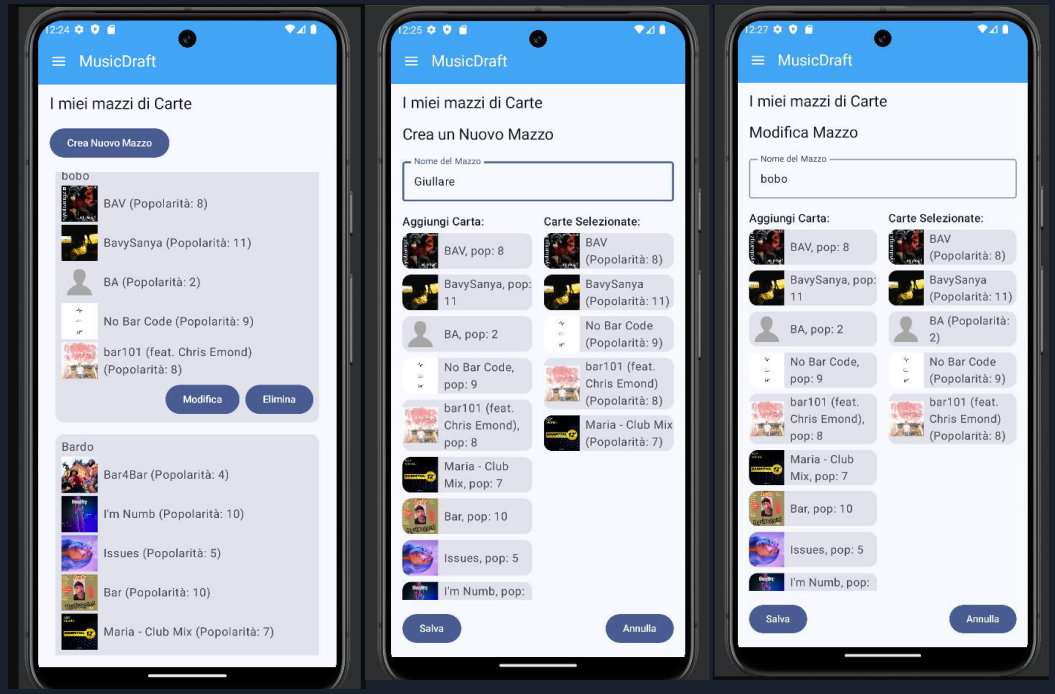
Cards - ViewModel e Visualizzazione

```
/**
 * Inserisce una carta traccia per l'utente e aggiorna i punti.
 *
 * @param track La traccia da aggiungere.
 * @param email L'email dell'utente.
 */
fun insertTrackToUser(track:Track, email:String){
    val totalPoint = loginViewModel.userLoggedInfo.value?.points?.minus((track.popolarita*10))
    if (totalPoint != null) {
        if(totalPoint >= 0) {
            val card = User_Cards_Track( identifire: 0,track.id,track.anno_pubblicazione,track.durata,track.immagine,tracl
            ownArtistRepo.insertUserCardTrack(card)
            ownArtistRepo.updatePoints(totalPoint,email)
            val market = ownArtistRepo.getAllOnMarketCardsT()
            market?.forEach { elem->
                if(elem.id_carta == track.id){
                    val gain = loginViewModel.userLoggedInfo.value?.points?.plus((elem.popolarita))
                    if (gain != null) {
                        ownArtistRepo.updatePoints(gain,elem.email)
                        ownArtistRepo.updateMarketNotStateT(email,elem.id_carta)
                    }
                }

            }

            updateAcquiredAddT(card)
        }
    }
}
```

```
@Composable
fun BraniScreen(brani: List<User_Cards_Track>, viewModel: CardsViewModel) {
    // Visualizza una griglia di carte per i brani
    LazyVerticalGrid(columns = GridCells.Fixed( count: 2)) { this: LazyGridScope
        // Itera attraverso i brani e visualizza una carta per ciascuno
        items(brani.size) { this: LazyGridItemScope  index ->
            BranoCard(brani[index])
        }
    }
}

/**
 * Composable per la visualizzazione di una carta di un brano.
 * @param brano Oggetto [Track] rappresentante il brano da visualizzare.
 * @param height Modificatore per la altezza della carta.
 */
@Composable
fun BranoCard(brano: User_Cards_Track,) {
    // Carta contenente le informazioni del brano
    Card(modifier = Modifier.padding(8.dp)) { this: ColumnScope
        Column(modifier = Modifier.padding(16.dp)) { this: ColumnScope
            Text(text = brano.nome, style = MaterialTheme.typography.bodyLarge)
            Text(text = brano.anno_pubblicazione)
            Text(text = brano.durata)
            Text(text = "Pop: ${brano.popolarita}")
            Text(text = "Costo: ${brano.popolarita*10}")
            // Immagine del brano con dimensioni specificate
            Image(
                painter = rememberAsyncImagePainter(brano.immagine),
                contentDescription = null,
                modifier = Modifier
                    .height(150.dp)
                    .fillMaxWidth(),
                contentScale = ContentScale.Crop
            )

        }
    }
}
```

Deck

- Nella schermata deck vi è tutta la gestione dei mazzi di carte: creazione, modifica,visualizzazione e cancellazione
- Deck si compone di 2 schermate:
  - Una di visualizzazione dei mazzi, che una volta creati possono essere modificati o cancellati
  - Una di creazione/modifica dei mazzi



Deck - Tabelle e DAO

- Mentre sul Dao, oltre alle solite query non c'è molto da aggiungere, mi vorrei soffermare sulla scelta fatta per la creazione del Deck.
- Partendo dal fatto che è un deck può essere composto solo da carte distinte e che queste devono essere esattamente 5. per rappresentare questa cardinalità si fatto associare ad ogni mazzo 5 righe di tabelle che condividono la maggior parte delle info tranne per la carta associata.

```kotlin
@Entity
data class Deck (

    @PrimaryKey(autoGenerate = true) val identifier: Int,
    val nome_mazzo:String,
    val carte_associate: String,
    val email:String,
    val popolarita: Float
```

```kotlin
/**
* Elimina tutti i mazzi di carte con un determinato nome e appartenenti a un utente specifico in modo asincrono.
*
* @param nomeMazzoDaEliminare Il nome del mazzo di carte da eliminare.
* @param email L'email dell'utente proprietario dei mazzi.
*/
@Query("DELETE FROM deck WHERE nome_mazzo = :nomeMazzoDaEliminare AND email =:email")
suspend fun deleteMazziByNome(nomeMazzoDaEliminare: String,email: String)

/**
 * Ottiene i nomi distinti dei mazzi di carte di un utente specifico in modo asincrono tramite Flow.
 *
 * @param email L'email dell'utente di cui ottenere i nomi dei mazzi.
 * @return Flow che emette una lista di nomi distinti dei mazzi di carte dell'utente specificato.
 */
@Query("SELECT DISTINCT nome_mazzo FROM Deck WHERE email= :email")
fun getDistinctDeckNames(email:String): Flow<List<String>>


/**
 * Ottiene le carte associate a un mazzo di carte specifico di un utente in modo asincrono tramite Flow.
 *
 * @param nomeMazzo Il nome del mazzo di carte di cui ottenere le carte.
 * @param email L'email dell'utente proprietario del mazzo di carte.
 * @return Flow che emette una lista di carte associate al mazzo specificato.
 */
@Query("SELECT carte_associate FROM Deck WHERE nome_mazzo = :nomeMazzo AND email = :email")
fun getDecksByNomeMazzoAndEmail(nomeMazzo: String, email: String): Flow<List<String>>

/**
 * Ottiene la popolarità dei mazzi di carte con un determinato nome e appartenenti a un utente specifico in modo asin
 *
 * @param nomeMazzo Il nome del mazzo di carte di cui ottenere la popolarità.
 * @param email L'email dell'utente proprietario del mazzo di carte.
 * @return Flow che emette la popolarità del mazzo di carte specificato.
 */
@Query("SELECT popolarita FROM Deck WHERE nome_mazzo = :nomeMazzo AND email = :email GROUP BY popolarita")
fun getDecksPop(nomeMazzo: String, email: String): Flow<Float>
```

Deck - Tabelle e DAO

- Mentre sul Dao, oltre alle solite query non c'è molto da aggiungere, mi vorrei soffermare sulla scelta fatta per la creazione del Deck.
- Partendo dal fatto che è un deck può essere composto solo da carte distinte e che queste devono essere esattamente 5. per rappresentare questa cardinalità si fatto associare ad ogni mazzo 5 righe di tabelle che condividono la maggior parte delle info tranne per la carta associata.

```kotlin
@Entity
data class Deck (

    @PrimaryKey(autoGenerate = true) val identifier: Int,
    val nome_mazzo:String,
    val carte_associate: String,
    val email:String,
    val popolarita: Float
```

```kotlin
/**
* Elimina tutti i mazzi di carte con un determinato nome e appartenenti a un utente specifico in modo asincrono.
*
* @param nomeMazzoDaEliminare Il nome del mazzo di carte da eliminare.
* @param email L'email dell'utente proprietario dei mazzi.
*/
@Query("DELETE FROM deck WHERE nome_mazzo = :nomeMazzoDaEliminare AND email =:email")
suspend fun deleteMazziByNome(nomeMazzoDaEliminare: String,email: String)

/**
 * Ottiene i nomi distinti dei mazzi di carte di un utente specifico in modo asincrono tramite Flow.
 *
 * @param email L'email dell'utente di cui ottenere i nomi dei mazzi.
 * @return Flow che emette una lista di nomi distinti dei mazzi di carte dell'utente specificato.
 */
@Query("SELECT DISTINCT nome_mazzo FROM Deck WHERE email= :email")
fun getDistinctDeckNames(email:String): Flow<List<String>>


/**
 * Ottiene le carte associate a un mazzo di carte specifico di un utente in modo asincrono tramite Flow.
 *
 * @param nomeMazzo Il nome del mazzo di carte di cui ottenere le carte.
 * @param email L'email dell'utente proprietario del mazzo di carte.
 * @return Flow che emette una lista di carte associate al mazzo specificato.
 */
@Query("SELECT carte_associate FROM Deck WHERE nome_mazzo = :nomeMazzo AND email = :email")
fun getDecksByNomeMazzoAndEmail(nomeMazzo: String, email: String): Flow<List<String>>

/**
 * Ottiene la popolarità dei mazzi di carte con un determinato nome e appartenenti a un utente specifico in modo asin
 *
 * @param nomeMazzo Il nome del mazzo di carte di cui ottenere la popolarità.
 * @param email L'email dell'utente proprietario del mazzo di carte.
 * @return Flow che emette la popolarità del mazzo di carte specificato.
 */
@Query("SELECT popolarita FROM Deck WHERE nome_mazzo = :nomeMazzo AND email = :email GROUP BY popolarita")
fun getDecksPop(nomeMazzo: String, email: String): Flow<Float>
```

Deck - ViewModel

```
fun toggleCardSelection(
    card: Cards,
    reqSentCurrentUser: List<ExchangeManagementCards>?,
    reqReceivedCurrentUser: List<ExchangeManagementCards>?
) {
    val currentSelectedCards = _selectedCards.value?.toMutableList() ?: mutableListOf()

    val cardId = card.id_carta

    val isCardInAnReceivedOffer = reqReceivedCurrentUser?.any { reicevdOffer ->
        reicevdOffer.idRequiredCard == cardId
    }
    val isCardInAnSentOffer = reqSentCurrentUser?.any{ sentOffer ->
        sentOffer.listOfferedCards.any { offerdCard ->
            offerdCard == cardId
        }
    }

    if (false) {
        _message.value = "This card is owned by another deck"
    }
    ////////////////////////////////////////////////////////////////////////////
    else if(isCardInAnReceivedOffer == true){
        _message.value = "This card was requested from you in some offer you received!\n\n" +
                         "Before adding it to a deck you must cancel all offers received in
    }
    else if(isCardInAnSentOffer == true){
        _message.value = "You offered this card in some offer you sent!\n\n" +
                         "Before adding it to a deck you must delete all offers sent in whic
    }
    ////////////////////////////////////////////////////////////////////////////
    else {
    if (currentSelectedCards.contains(card)) {
        currentSelectedCards.remove(card)
        } else {
        currentSelectedCards.add(card)
        }
    _selectedCards.value = currentSelectedCards
    }
}
```

```
fun salvaMazzo() {
    val email = loginViewModel.userLoggedInfo.value!!.email
    val names = deckRepository.namesDecks?.value
    val currentDeck = selectedDeck.value
    val currentCards = selectedCards.value
    if (currentDeck != null) {
        // Verifica se il nome del mazzo è unico
        mazzi.value!!.forEach { elem ->

            for(n in names!!){
                if(compareDeckNames(n,deckName.value)){
                    _message.value = "Deck name already exist"
                    return
                }
            }
        }
        if(!(distinctCards(currentCards))){
            _message.value = "You had to choose 5 distinct cards"
            return
        }
        // Calcolo della popolarità media delle carte selezionate
        var pop: Float = 0F
        for (i in selectedCards.value) {
            pop += i.popolarita
        }
        val temp: MutableList<Deck> = mutableListOf()
        for (i in selectedCards.value) {
            val m = Deck( identifier: 0, _deckName.value, i.id_carta, email,  popolarita: pop / 5)
            temp.add(m)
        }
        deckRepository.insertAllNewDeck(temp)

        mazzi.value!!.add(Mazzo(deckName.value, _selectedCards.value))

        // Reset degli stati
        annullaModifica()
        _selectedCards.value = emptyList()
        _message.value = "Deck Saved"
    }
}
```

Deck - Visualizzazione

```
val message by viewModel.message.collectAsState()
if (message != null) {
    AlertDialog(
        onDismissRequest = { viewModel.clearMessage() },
        title = { Text(text = "Alert") },
        text = { Text(text = message!!) },
        confirmButton = {
            Button(onClick = { viewModel.clearMessage() }) { this: RowScope
                Text( text: "OK")
            }
        },
        //text = { Text(message ?: "") }
    )
}
```

```
        // Colonna per "Carte Selezionate"
        LazyColumn(
            modifier = Modifier
                .weight(1f)
                .padding(start = 8.dp)
        ) { this: LazyListScope
            item { this: LazyItemScope
                Text( text: "Selected Cards:", style = MaterialTheme.typography.titleMedium)
            }

            items(selectedCards) { this: LazyItemScope card ->
                Card(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(vertical = 4.dp)
                        .clickable { viewModel.toggleCardSelection(
                            card,
                            reqSentCurrentUser,
                            reqReceivedCurrentUser
                        ) }
                ) { this: ColumnScope
                    Row(
                        verticalAlignment = Alignment.CenterVertically,
                        modifier = Modifier.fillMaxWidth()
                    ) { this: RowScope
                        Image(
                            painter = rememberImagePainter(data = card.immagine),
                            contentDescription = null,
                            modifier = Modifier.size(50.dp)
                        )
                        Spacer(modifier = Modifier.width(8.dp))
                        Text( text: "${card.nome_carta} (Pop: ${card.popolarita})")
                    }
                }
            }
        }
    }
```

Testing

- Abbiamo effettuato degli unit test per testare alcune funzioni di utility
- Nel caso specifico prendiamo ad esempio 2 funzioni che usiamo nel DeckViewModel:
  - compareDeckNames: compare i nomi del deck che si sta creando con quelli esistenti in modo da non creare 2 mazzi con nome identico.
  - distinctCards: controlla se le carte componenti del mazzo sono tutte diverse l'una dall'altra e se sono esattamente 5.

```
fun compareDeckNames(deckName1:String, deckName2: String):Boolean{

    if(deckName1.replace("\\s".toRegex(), replacement: "").equals(deckName2.replace("\\s".toRegex(), replacement: ""))){
        return true
    }else{
        return false
    }
}
fun distinctCards(cards:List<DeckViewModel.Cards>):Boolean{
    if(cards.distinctBy{ it.id_carta }.size<5 || cards.distinctBy{ it.id_carta }.size>5){
        return false
    }else{
        return true
    }
}
```

Testing

- Abbiamo effettuato degli unit test per testare alcune funzioni di utility
- Nel caso specifico prendiamo ad esempio 2 funzioni che usiamo nel DeckViewModel:
  - compareDeckNames: compare i nomi del deck che si sta creando con quelli esistenti in modo da non creare 2 mazzi con nome identico.
  - distinctCards: controlla se le carte componenti del mazzo sono tutte diverse l'una dall'altra e se sono esattamente 5.

```
fun compareDeckNames(deckName1:String, deckName2: String):Boolean{

    if(deckName1.replace("\\s".toRegex(), replacement: "").equals(deckName2.replace("\\s".toRegex(), replacement: ""))){
        return true
    }else{
        return false
    }
}
fun distinctCards(cards:List<DeckViewModel.Cards>):Boolean{
    if(cards.distinctBy{ it.id_carta }.size<5 || cards.distinctBy{ it.id_carta }.size>5){
        return false
    }else{
        return true
    }
}
```

Testing(2)

```
class UtilTest {
    @Test
    fun testDeckNamesLowcaseUppercase() {
        val deckname1 = "deck1"
        val deckname2 = "DECK1"
        val res = compareDeckNames(deckname1,deckname2)
        assertEquals( expected: false,res)
    }
    @Test
    fun testDeckNamesDifferent() {
        val deckname1 = "deCk1"
        val deckname2 = "deck1"
        val res = compareDeckNames(deckname1,deckname2)
        assertEquals( expected: false,res)
    }
    @Test
    fun testDeckNamesEquals() {
        val deckname1 = "deck1"
        val deckname2 = "deck1"
        val res = compareDeckNames(deckname1,deckname2)
        assertEquals( expected: true,res)
    }
    @Test
    fun testDeckNamesDifferent2() {
        val deckname1 = " deck1"
        val deckname2 = "deck1"
        val res = compareDeckNames(deckname1,deckname2)
        assertEquals( expected: true,res)
    }

    @Test
    fun testDeckNamesDifferent3() {
        val deckname1 = " deck 1"
        val deckname2 = "deck  1"
        val res = compareDeckNames(deckname1,deckname2)
        assertEquals( expected: true,res)
    }
```

Testing(2)

```kotlin
/**
 * Verifica che una lista di carte distinte nel mazzo sia considerata valida.
 */
@Test
fun testDistinctCards() {


    val card1 = DeckViewModel.Cards( id_carta: "0", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card2 = DeckViewModel.Cards( id_carta: "1", nome_carta: "J", immagine: "j.png", popolarita: 11)
    val card3 = DeckViewModel.Cards( id_carta: "2", nome_carta: "Ji", immagine: "k.png", popolarita: 12)
    val card4 = DeckViewModel.Cards( id_carta: "3", nome_carta: "Jim", immagine: "l.png", popolarita: 13)
    val card5 = DeckViewModel.Cards( id_carta: "4", nome_carta: "Jimm", immagine: "z.png", popolarita: 14)

    val cards:List<DeckViewModel.Cards> = listOf(card1,card2,card3,card4,card5)

    val res = distinctCards(cards)
    assertEquals( expected: true,res)
}

/**
 * Verifica che una lista di carte nel mazzo con una coppia di carte uguali non sia considerata valida.
 */
@Test
fun testDistinctCardsWith1Equal() {


    val card1 = DeckViewModel.Cards( id_carta: "0", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card2 = DeckViewModel.Cards( id_carta: "0", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card3 = DeckViewModel.Cards( id_carta: "2", nome_carta: "Ji", immagine: "k.png", popolarita: 12)
    val card4 = DeckViewModel.Cards( id_carta: "3", nome_carta: "Jim", immagine: "l.png", popolarita: 13)
    val card5 = DeckViewModel.Cards( id_carta: "4", nome_carta: "Jimm", immagine: "z.png", popolarita: 14)

    val cards:List<DeckViewModel.Cards> = listOf(card1,card2,card3,card4,card5)

    val res = distinctCards(cards)
    assertEquals( expected: false,res)
}
```

```kotlin
/**
 * Verifica che una lista di carte nel mazzo con meno di cinque carte non sia considerata valida.
 */
@Test
fun testDistinctCardsWithLessCards() {


    val card1 = DeckViewModel.Cards( id_carta: "0", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card2 = DeckViewModel.Cards( id_carta: "1", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card3 = DeckViewModel.Cards( id_carta: "2", nome_carta: "Ji", immagine: "k.png", popolarita: 12)
    val card4 = DeckViewModel.Cards( id_carta: "3", nome_carta: "Jim", immagine: "l.png", popolarita: 13)

    val cards:List<DeckViewModel.Cards> = listOf(card1,card2,card3,card4)

    val res = distinctCards(cards)
    assertEquals( expected: false,res)
}

/**
 * Verifica che una lista di carte nel mazzo con più di cinque carte non sia considerata valida.
 */
@Test
fun testDistinctCardsWithMoreCards() {


    val card1 = DeckViewModel.Cards( id_carta: "0", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card2 = DeckViewModel.Cards( id_carta: "1", nome_carta: "Jimmy", immagine: "i.png", popolarita: 10)
    val card3 = DeckViewModel.Cards( id_carta: "2", nome_carta: "Ji", immagine: "k.png", popolarita: 12)
    val card4 = DeckViewModel.Cards( id_carta: "3", nome_carta: "Jim", immagine: "l.png", popolarita: 13)
    val card5 = DeckViewModel.Cards( id_carta: "4", nome_carta: "Jimm", immagine: "z.png", popolarita: 14)
    val card6 = DeckViewModel.Cards( id_carta: "5", nome_carta: "JimmyOh", immagine: "x.png", popolarita: 15)


    val cards:List<DeckViewModel.Cards> = listOf(card1,card2,card3,card4,card5,card6)

    val res = distinctCards(cards)
    assertEquals( expected: false,res)
}
```

Possibili miglioramenti..

- **Espansione delle Funzionalità Social**: potenziamento delle funzionalità social per una maggiore interazione tra gli utenti. (ad es. ‘Scambio di messaggi’)
- **Funzionalità Avanzate di Matchmaking**: miglioramenti al sistema di matchmaking per offrire un'esperienza di gioco più equilibrata e divertente.
- **Notifiche Push**: aggiunta di notifiche push per avvisare gli utenti di eventi rilevanti, come nuove carte disponibili o sfide ricevute.
- **Sincronizzazione dei Dati**: i dati relativi alle carte e ai mazzi saranno sincronizzati con un server esterno per garantire la disponibilità su più dispositivi contemporaneamente.
- **Servizi di Amicizia e Matchmaking**: integrazione con server per la gestione delle amicizie, la compravendita delle carte e il matchmaking.

GRAZIE PER L'ATTENZIONE