ChatGPT

Guida Tecnica per lo Sviluppo dell'MVP di Lymbo con Firebase (iOS/Swift)

Questa guida fornisce istruzioni dettagliate per integrare **Firebase** nell'app **Lymbo** (iOS in Swift), automatizzando il più possibile il setup e lo sviluppo. È pensata per un sviluppatore junior che utilizza l'editor Cursor, quindi include script da terminale, codice Swift ben commentato e una struttura di progetto chiara. Eviteremo configurazioni manuali via interfaccia grafica ove possibile, utilizzando invece la Firebase CLI e file di configurazione testuali.

1. Configurazione del Progetto Firebase tramite Firebase CLI

Per iniziare, utilizzeremo la **Firebase CLI** (Command Line Interface) per creare e configurare il progetto Firebase e i servizi necessari, senza dover passare dal web. Assicurati di avere **Node.js/npm** installato (serve per installare la CLI). Di seguito i passaggi da seguire:

1. **Installa Firebase CLI**: installa gli strumenti da riga di comando di Firebase globalmente tramite npm.

   ```
   # Installa Firebase CLI tramite npm (necessita Node.js)
   npm install -g firebase-tools
   ```

   Nota: Se preferisci, puoi installare la CLI anche via **Homebrew** su macOS (`brew install firebase-cli`). Dopo l'installazione, verifica con `firebase --version` che tutto funzioni.

2. **Autenticati ed effettua login**: prima di creare un progetto, autentica la CLI con il tuo account Google/Firebase.

   ```
   # Esegui il login nell'account Google (si aprirà il browser per l'autorizzazione)
   firebase login
   ```

3. **Crea un nuovo progetto Firebase**: utilizza la CLI per creare un progetto Firebase per Lymbo. Questo comando genererà un nuovo progetto su Firebase e un corrispondente progetto Google Cloud. Esempio:

   ```
   # Crea un progetto Firebase denominato "Lymbo App" con projectId "lymbo-app"
   firebase projects:create --display-name "Lymbo App" lymbo-app
   ```

   Il parametro `--display-name` assegna un nome leggibile al progetto, mentre l'ultimo argomento `lymbo-app` è l'ID univoco del progetto (che dev'essere univoco globalmente). Puoi

scegliere un ID diverso se preferisci. La CLI provvederà a creare il progetto su Firebase e collegarlo al tuo account [1].

4. **Registra l'app iOS nel progetto Firebase**: una volta creato il contenitore di progetto, registra al suo interno l'app iOS di Lymbo. Usa il comando CLI per creare un'app iOS associata al bundle ID della tua app. Esegui questo nella directory del progetto Xcode (dove c'è ad esempio il file `.xcodeproj`), in modo da poter salvare direttamente il file di configurazione:

```
# Crea un'app iOS nel progetto Firebase (sostituisci com.example.Lymbo
col Bundle ID reale)
firebase apps:create ios --bundle-id com.example.Lymbo --project lymbo-
app
```

   Assicurati di sostituire `com.example.Lymbo` con il **Bundle Identifier** effettivo della tua app (così come impostato in Xcode, es. `org.lymbo.app`). La CLI restituirà un ID dell'app Firebase appena creata.

5. **Ottieni il file di configurazione iOS (GoogleService-Info.plist)**: dopo aver registrato l'app iOS, procurati il file di configurazione necessario all'app (contiene chiavi e ID del progetto). Invece di scaricarlo manualmente dalla console, usiamo la CLI per estrarlo automaticamente [2]:

```
# Scarica il file GoogleService-Info.plist nella directory corrente
firebase apps:sdkconfig ios --project lymbo-app -o GoogleService-
Info.plist
```

   Questo comando recupera la configurazione dell'app iOS appena creata e la salva in un file `GoogleService-Info.plist`. Inserisci questo file nel progetto Xcode di Lymbo (trascina il plist nel progetto, assicurandoti che sia aggiunto ai target dell'app). In alternativa, il comando sopra può essere eseguito direttamente dentro la cartella iOS del progetto, sovrascrivendo il plist se già presente.

6. **Inizializza Firestore e Storage nel progetto (Firebase init)**: ora configuriamo i servizi Firebase che useremo (Firestore, Storage, ecc.) creando i file di configurazione e di regole di sicurezza. La CLI offre il comando di inizializzazione interattiva:

```
firebase init
```

   Durante l'esecuzione di `firebase init`, scegli **Firestore** e **Storage** come servizi da configurare (non selezionare Hosting o altri se non ti servono). Ti verrà chiesto se vuoi configurare le regole di sicurezza e i file indice; conferma le opzioni predefinite. Al termine, la CLI creerà alcuni file importanti nella directory del progetto: `firebase.json`, `.firebaserc`, `firestore.rules`, `storage.rules` (oltre ad eventuali file di esempio per Firestore indexes). Di seguito il ruolo di ciascun file generato:

7. **firebase.json** – definisce quali servizi sono configurati e dove trovare i file di regole. Esempio minimale di `firebase.json` dopo l'init (contenente solo Firestore e Storage):

```
{
  "firestore": {
    "rules": "firestore.rules"
  },
  "storage": {
    "rules": "storage.rules"
  }
}
```

Questo indica alla CLI di usare `firestore.rules` e `storage.rules` come sorgente per le regole di sicurezza di Firestore e Cloud Storage rispettivamente [3] [4].

8. **.firebaserc** – mantiene l'associazione tra la directory di lavoro locale e il progetto Firebase. Contiene l'ID del progetto (o un alias) usato come **default**. Esempio:

```
{
  "projects": {
    "default": "lymbo-app"
  }
}
```

In questo modo tutti i comandi `firebase` eseguiti nella cartella useranno di default il progetto `lymbo-app` senza dover specificare `--project` ogni volta.

9. **firestore.rules** – file contenente le **regole di sicurezza** per Cloud Firestore. In modalità protetta, le regole predefinite potrebbero bloccare tutte le letture/scritture finché non le modifichi. Per l'MVP, possiamo impostare regole semplici che consentano accesso ai soli utenti autenticati. Ad esempio:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Consenti lettura/scrittura di qualsiasi documento solo se
l'utente è autenticato
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Questa regola permette operazioni su qualsiasi documento **solo** se `request.auth != null`, ovvero se l'utente ha effettuato l'autenticazione (Firebase Auth). È una regola generica utile in fase iniziale; in futuro andrà raffinata per concedere accessi più specifici (ad es. solo proprietari del dato, etc.).

10. **storage.rules** – analogamente, definisce le **regole di sicurezza** per Cloud Storage. Possiamo scrivere una regola simile che richiede l'autenticazione per qualsiasi upload/download:

```
rules_version = '2';
service firebase.storage {
```

```
  match /b/{bucket}/o {
    // Consenti accesso ai file solo se l'utente è autenticato
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Anche qui, ogni operazione sullo Storage sarà consentita solo ad utenti loggati. In seguito, potrai restringere ulteriormente le regole (es. solo il proprietario può modificare il proprio file, dimensione massima file, ecc.), ma per l'MVP questa impostazione fornisce un livello base di sicurezza.

11. **Deploy delle regole di Firestore e Storage**: dopo aver personalizzato le regole come sopra, dobbiamo **pubblicarle** sul server Firebase affinché entrino in vigore. Usiamo la CLI per fare il deploy delle sole regole (evitando altri servizi non configurati):

    ```
    # Esegui il deploy solo delle rules di Firestore e Storage sul progetto Firebase
    firebase deploy --only firestore:rules,storage:rules
    ```

    La CLI compilerà e caricherà le regole. Se tutto è corretto, vedrai un output che conferma la pubblicazione delle regole su Firestore e Storage nel progetto remoto [5] [6]. Da questo momento, il database Firestore e il bucket Storage del tuo progetto Firebase seguiranno le regole definite nei file locali.

Nota: I passi sopra (3–7) possono essere automatizzati in uno script shell (ad es. `setup_firebase.sh`) da eseguire una tantum. Tuttavia, eseguirli manualmente una volta aiuta a capire ogni fase. Inoltre, alcune configurazioni (come l'abilitazione di certi provider di autenticazione o la configurazione delle chiavi APNs per le notifiche push) richiedono comunque passaggi sul sito Firebase o su Apple Developer, come vedremo più avanti.

2. Installazione del SDK Firebase nell'app iOS (Swift) e Configurazione Iniziale

Ora che il backend Firebase è pronto, integriamo il SDK di Firebase nell'app iOS di Lymbo e facciamo la configurazione iniziale necessaria.

2.1 Aggiunta dei pacchetti Firebase (Swift Package Manager)

L'app utilizzerà diversi prodotti Firebase (Auth, Firestore, Storage, Messaging), quindi aggiungeremo le relative librerie. Usiamo **Swift Package Manager (SPM)** per gestire le dipendenze Firebase, come raccomandato ufficialmente da Firebase (evitando CocoaPods per automatizzare meglio la configurazione).

- **Aggancio del repository Firebase**: in Xcode, apri il progetto Lymbo. Vai su *File* > *Add Packages...* e inserisci l'URL del repository SPM di Firebase:

```
https://github.com/firebase/firebase-ios-sdk.git
```

Se richiesto, scegli la versione più recente disponibile (di default dovrebbe proporre la latest). Conferma per caricare l'elenco dei pacchetti Firebase disponibili [7].

- **Selezione dei pacchetti Firebase necessari**: Xcode mostrerà una lista di librerie Firebase. Per il nostro caso d'uso, includi i seguenti pacchetti:

- `FirebaseAuth` (Autenticazione email/password)
- `FirebaseFirestore` (Database Cloud Firestore)
- `FirebaseStorage` (Storage di file)
- `FirebaseMessaging` (FCM per notifiche push)
- *(Opzionale ma consigliato)* `FirebaseAnalytics` (per abilitarne il supporto, utile se vuoi statistiche e per alcune funzionalità di FCM). Se non ti serve Analytics, puoi ometterlo.

Seleziona ciascuna di queste librerie e aggiungile al progetto. Xcode scaricherà e integrerà i pacchetti automaticamente.

- **Impostazione del linker flag** `-ObjC`: Alcune librerie Firebase (ad esempio FirebaseAuth) richiedono di aggiungere il flag `-ObjC` ai **Other Linker Flags** del target iOS, affinché siano correttamente linkate categorie Objective-C interne [8]. In Xcode, vai nel *target* dell'app > scheda **Build Settings** > cerca *Other Linker Flags* > aggiungi `-ObjC` (come flag singolo) per il build di Debug e Release. Questo assicura che eventuali simboli di categorie Objective-C all'interno dei framework Firebase vengano inclusi all'esecuzione.

2.2 Configurazione iniziale di Firebase nell'app (GoogleService-Info.plist e FirebaseApp)

Dopo aver aggiunto i framework, bisogna configurare Firebase all'avvio dell'app:

- **Inserisci il file GoogleService-Info.plist**: assicurati che il file `GoogleService-Info.plist` ottenuto in precedenza sia presente nel progetto Xcode (trascinandolo nel Project Navigator) e incluso nel target principale. Questo file contiene l'ID progetto, le API key e altri identificatori non sensibili necessari per inizializzare i servizi Firebase.
  Suggerimento: organizza il file magari in una sottocartella *Config/* nel progetto, per ordine, ma va bene anche a livello root. Verifica che nel *Target Membership* sia spuntato per l'app.

- **Inizializza Firebase nell'** `AppDelegate`: nel metodo `application(_:didFinishLaunchingWithOptions:)` dell'**AppDelegate**, aggiungi il codice per configurare la libreria Firebase:

```
import FirebaseCore  // oltre ad altri import Firebase se usati qui

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    // ...
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
```

```
        // Configura Firebase – deve essere chiamato all'avvio
        FirebaseApp.configure()
        // ... altre inizializzazioni ...
        return true
    }
    // ...
}
```

Il call `FirebaseApp.configure()` carica le impostazioni dal plist GoogleService-Info e prepara l'SDK Firebase [9]. Assicurati di aver importato il modulo `FirebaseCore` (o semplicemente uno qualsiasi dei moduli Firebase, dato che importano FirebaseCore internamente). Dopo questa chiamata, tutti i servizi Firebase (Auth, Firestore, ecc.) sono pronti per essere utilizzati.

- **Abilita i servizi di background necessari**: per usare **Cloud Messaging** (notifiche push), devi abilitare le **Push Notifications** e il **Background Mode** "Remote notifications" nel tuo progetto Xcode. Apri il *target* dell'app > **Signing & Capabilities**:

- Aggiungi la **Capability** "Push Notifications".

- Aggiungi la **Capability** "Background Modes" e seleziona "Remote notifications".
  Questo consente all'app di ricevere notifiche push sia a app aperta che in background. *(Queste configurazioni purtroppo non si possono fare via CLI; bisogna usare Xcode o modificare manualmente l'Entitlements file.)*

- **Chiave APNs per Firebase Cloud Messaging**: per inviare notifiche push ai dispositivi iOS tramite FCM, è necessario fornire a Firebase la chiave APNs (o un certificato) del tuo Apple Developer account. Vai sulla console Firebase, sezione **Project Settings > Cloud Messaging**, e nella configurazione dell'app iOS carica la **APNs Auth Key** (*.p8*) oppure il certificato .p12, insieme al **Team ID** Apple e al **Key ID**. Questo passaggio è manuale e va fatto una sola volta per progetto: permette a Firebase di inoltrare le notifiche APNs ai tuoi utenti iOS. *(Assicurati di avere un Apple Developer account e di aver generato una APNs Auth Key nella sezione Keys; la procedura è documentata nei tutorial Apple/FCM.)*

Una volta completati questi passi, la tua app è connessa al progetto Firebase e i servizi di base (Auth, DB, Storage, Messaging) sono inizializzati. Nella prossima sezione, vedremo come implementare le funzionalità specifiche (login, registrazione, salvataggio dati, upload immagini, notifiche) usando le API Firebase.

3. Implementazione delle Funzionalità Firebase (Auth, Firestore, Storage, Messaging)

In questa sezione svilupperemo le funzionalità chiave dell'app Lymbo utilizzando Firebase. Forniremo esempi di codice Swift riutilizzabile, con funzioni dedicate a ciascun compito, e commenti esplicativi. Si presume che tutte le chiamate Firebase avvengano *dopo* aver fatto `FirebaseApp.configure()` (ad esempio, dopo il login dell'utente nell'app o nelle relative view controller).

3.1 Autenticazione Email/Password (Login e Registrazione)

Per autenticare gli utenti utilizzeremo **Firebase Authentication** con il metodo **Email/Password**. Assicurati di aver **abilitato** questo provider nella console Firebase (Firebase > Authentication > Sign-in methods > Email/Password: Enabled), altrimenti le chiamate di creazione utenti restituiranno errore. (Non c'è un comando CLI per abilitarlo, va fatto manualmente una tantum.)

Di seguito, due funzioni Swift di esempio per registrare un nuovo utente e per autenticare un utente esistente con email e password:

```
import FirebaseAuth

// Funzione per registrare un nuovo utente con email e password
func signUp(email: String, password: String, completion: @escaping (Error?) -
> Void) {
    Auth.auth().createUser(withEmail: email, password: password) {
authResult, error in
        if let error = error {
            // Registrazione fallita (es. formato email errato, password
troppo debole, ecc.)
            print("Errore registrazione: \(error.localizedDescription)")
            completion(error)
        } else {
            // Registrazione avvenuta con successo
            let user = authResult?.user  // Firebase User creato
            print("Utente creato con UID: \(user?.uid ?? "(nil)")")
            completion(nil)
            // Possiamo procedere a creare il profilo utente su Firestore
(vedi funzione dedicata)
        }
    }
}

// Funzione per effettuare il login con email e password
func logIn(email: String, password: String, completion: @escaping (Error?) -
> Void) {
    Auth.auth().signIn(withEmail: email, password: password) { authResult,
error in
        if let error = error {
            // Autenticazione fallita (credenziali errate o utente
inesistente)
            print("Errore login: \(error.localizedDescription)")
            completion(error)
        } else {
            // Login eseguito con successo
            let user = authResult?.user
            print("Login eseguito, utente UID: \(user?.uid ?? "(nil)")")
            completion(nil)
        }
```

```
    }
}
```

Le chiamate `Auth.auth().createUser(withEmail:password:)` e `Auth.auth().signIn(withEmail:password:)` sono fornite dal SDK FirebaseAuth [10] [11]. In caso di successo, `authResult?.user` restituisce un oggetto `User` Firebase contenente informazioni come l'UID univoco dell'utente, l'email, ecc. In caso di errore (ad esempio email già in uso, password troppo corta, mancanza di connessione), viene passato un oggetto `Error` con la descrizione.

Nell'implementazione sopra, utilizziamo un completion handler semplice che ritorna un `Error?` per segnalare se l'operazione è riuscita o meno. In un'app reale, potresti voler restituire direttamente l'oggetto `User` appena creato/loggato (ad esempio usando `Result<User, Error>`). Per semplicità, qui ci limitiamo a segnalare l'errore e stampare l'UID in console.

> **Nota:** È buona pratica, dopo `signUp`, inviare magari una **email di verifica** all'utente (`user.sendEmailVerification`) – ma questo esula dall'MVP se non strettamente richiesto. Inoltre, per completare la gestione Auth, implementa anche la **logout** quando necessario: FirebaseAuth offre `try Auth.auth().signOut()` per disconnettere l'utente (ad esempio su pulsante "Esci").

3.2 Creazione e Aggiornamento del Profilo Utente (Cloud Firestore)

Dopo la registrazione, generalmente l'app Lymbo creerà un **profilo utente** con informazioni aggiuntive (nome, età, bio, preferenze, foto profilo, ecc.). Useremo **Cloud Firestore** per archiviare questi dati strutturati.

Immaginiamo di avere una collezione Firestore chiamata `"users"` dove ogni documento rappresenta il profilo di un utente. L'ID del documento può essere l'UID dell'utente autenticato (in modo da avere corrispondenza 1-1 con Firebase Auth).

Di seguito, una funzione per creare il documento profilo di un nuovo utente e una per aggiornare il profilo esistente:

```
import FirebaseFirestore

// Crea un nuovo profilo utente nella collezione "users" con l'UID
specificato
func createUserProfile(uid: String, name: String, email: String, completion:
@escaping (Error?) -> Void) {
    let db = Firestore.firestore()
    let profileData: [String: Any] = [
        "name": name,
        "email": email,
        "createdAt": FieldValue.serverTimestamp()  // timestamp server-side
    ]
    // Salva il documento del profilo (merge: false, sovrascrive se esiste)
    db.collection("users").document(uid).setData(profileData) { error in
        if let error = error {
            print("Errore creazione profilo: \(error.localizedDescription)")
```

```
        } else {
            print("Profilo utente creato per UID \(uid)")
        }
        completion(error)
    }
}

// Aggiorna alcuni campi del profilo utente esistente
func updateUserProfile(uid: String, fields: [String: Any], completion:
@escaping (Error?) -> Void) {
    let db = Firestore.firestore()
    db.collection("users").document(uid).updateData(fields) { error in
        if let error = error {
            print("Errore aggiornamento profilo: \
(error.localizedDescription)")
        } else {
            print("Profilo utente \(uid) aggiornato con campi: \
(fields.keys)")
        }
        completion(error)
    }
}
```

Spiegazione:
- `Firestore.firestore()` restituisce un riferimento al database.
- `setData` crea (o sovrascrive) il documento con i dati forniti. Usiamo l'UID come ID documento per garantire unicità e facile recupero. Il campo `createdAt` utilizza `FieldValue.serverTimestamp()` per impostare un timestamp dal server Firebase al momento dell'operazione.
- `updateData` aggiorna solo i campi indicati senza toccare gli altri. Può essere usato ad esempio per aggiungere/modificare la bio, il numero di like, l'URL della foto profilo, ecc., passando un dizionario `fields` con solo i campi da cambiare.

Entrambe le funzioni accettano un completion handler con `Error?` per segnalare success/fallimento. In caso di successo l'errore sarà nil.

Esempio d'uso: dopo `signUp`, potresti chiamare `createUserProfile(uid: user.uid, name: nomeInserito, email: user.email)` per creare immediatamente un profilo di base. Oppure mostrare una schermata di completamento profilo e poi salvare.

Per **leggere** i dati di un profilo utente, puoi usare Firestore così:

```
let db = Firestore.firestore()
db.collection("users").document(uid).getDocument { document, error in
    if let document = document, document.exists {
        let data = document.data()
        print("Dati profilo utente:", data ?? "")
        // ad esempio, estrai campi:
        let name = data?["name"] as? String ?? ""
        let email = data?["email"] as? String ?? ""
```

```
    } else {
        print("Profilo utente non trovato o errore: \
(error?.localizedDescription ?? "N/A")")
    }
}
```

In alternativa, puoi utilizzare la lettura **in tempo reale** con `addSnapshotListener` per osservare automaticamente i cambiamenti del profilo (utile se vuoi aggiornare l'UI quando, ad esempio, cambia qualcosa nel profilo), ma per l'MVP una lettura on-demand come sopra è sufficiente.

3.3 Gestione di Likes e Match su Firestore

Lymbo, essendo un'app di incontri, avrà la logica di "like" e "match" tra profili. Possiamo modellarla in Firestore con due collezioni: - **"likes"**: contiene documenti per ogni volta che un utente A mette like a un utente B. - **"matches"**: contiene documenti quando due utenti hanno un like reciproco (A ha likato B *e* B ha likato A).

Un possibile schema per `likes`: ogni documento può avere i campi `"from"` (UID di chi ha messo il like), `"to"` (UID di chi è stato likato), e un timestamp. Per `matches`: ogni documento potrebbe avere un array `"users"` con i due UID coinvolti e un timestamp.

Di seguito una funzione che registra un *like* e controlla se genera un *match*:

```
// Registra un "like" dall'utente userId verso l'utente otherUserId.
// Se anche otherUserId aveva messo like a userId, crea un documento di
"match".
func sendLike(from userId: String, to otherUserId: String, completion:
@escaping (Error?) -> Void) {
    let db = Firestore.firestore()
    // Dati del like da salvare
    let likeData: [String: Any] = [
        "from": userId,
        "to": otherUserId,
        "timestamp": FieldValue.serverTimestamp()
    ]
    // Aggiungi un nuovo documento nella collezione "likes"
    db.collection("likes").addDocument(data: likeData) { error in
        if let error = error {
            print("Errore nell'inviare il like: \
(error.localizedDescription)")
            completion(error)
        } else {
            print("Like inviato da \(userId) a \(otherUserId)")
            // Dopo aver inviato il like, controlla se esiste un like opposto
(otherUserId -> userId)
            let query = db.collection("likes")
                .whereField("from", isEqualTo: otherUserId)
                .whereField("to", isEqualTo: userId)
            query.getDocuments { querySnapshot, err in
                if let err = err {
```

```
                    print("Errore nel verificare il match: \
(err.localizedDescription)")
                } else if let docs = querySnapshot?.documents, !docs.isEmpty
{
                    // Esiste almeno un documento in "likes" che indica
otherUserId ha likato userId: è un match!
                    let matchData: [String: Any] = [
                        "users": [userId, otherUserId],
                        "createdAt": FieldValue.serverTimestamp()
                    ]
                    db.collection("matches").addDocument(data: matchData) {
matchErr in
                        if matchErr == nil {
                            print("  Match creato tra \(userId) e \
(otherUserId)!")
                        }
                    }
                }
                // (Nota: se docs.isEmpty, nessun match per ora)
            }
            completion(nil)
        }
    }
}
```

Cosa fa questa funzione:
- Inserisce un documento nella collezione *likes* con i dati del like.
- Poi esegue una query su *likes* per vedere se l'utente target (`otherUserId`) aveva già messo like all'utente corrente (`userId`). La query filtra i documenti dove `from == otherUserId` e `to == userId`.
- Se la query restituisce uno o più documenti, significa che *anche* l'altro utente aveva ricambiato il like in passato, quindi c'è un **match**. In tal caso, la funzione crea un documento nella collezione *matches* contenente i due utenti. Qui `matchData["users"]` è un array con i due UID, ma potresti anche salvare altri dettagli (es. timestamp, riferimenti ai documenti utente, etc.).
- I `print` e log indicano cosa sta succedendo. In caso di match, si stampa un messaggio di conferma.

Abbiamo scelto di usare `addDocument` per generare automaticamente un ID di documento univoco (utile per likes e matches). Volendo, per *matches* si potrebbe usare un ID deterministicamente combinando i UID (es. `"uidA_uidB"`) per evitare duplicati, ma ciò complica un po' la logica – per l'MVP non è cruciale.

Utilizzo: chiamerai `sendLike(from: currentUserId, to: otherId)` quando l'utente esprime un like nell'interfaccia. La funzione si occupa di tutto. In caso di match, potresti notificare l'utente (es. via push o aggiornando l'UI) che c'è un nuovo match.

Ricorda che per leggere i match di un utente, potresti fare una query su `matches` filtrando l'array `"users"` contenente il suo UID (Firestore permette query con `array-contains`). Oppure mantenere nei documenti utente un elenco di match. Ci sono varie strategie; l'approccio con collezione separata mantiene le cose semplici.

3.4 Upload di File su Firebase Storage (Foto Profilo)

Per gestire le foto (es. foto profilo degli utenti), utilizzeremo **Firebase Cloud Storage**. Ogni utente può caricare un'immagine che associeremo al suo profilo. Un design comune è creare, nel bucket Storage, una cartella per utente (nome = UID) e salvare lì la foto profilo.

Supponiamo di avere un oggetto `UIImage` (ad esempio ottenuto dalla UIImagePickerController o da SwiftUI ImagePicker) da caricare. La funzione seguente mostra come convertire l'immagine in data, caricarla su Storage e ottenere l'URL pubblico di download:

```
import FirebaseStorage

// Carica l'immagine del profilo per l'utente dato su Firebase Storage.
// Salva il file come "profiles/<uid>/photo.jpg". Ritorna l'URL pubblico in
caso di successo.
func uploadProfileImage(userId: String, image: UIImage, completion:
@escaping (Result<URL, Error>) -> Void) {
    let storageRef = Storage.storage().reference()  // riferimento root del
bucket
        .child("profiles/\(userId)/photo.jpg")      // path del file nel
bucket
    // Converti UIImage in JPEG data
    guard let imageData = image.jpegData(compressionQuality: 0.8) else {
        completion(.failure(NSError(domain: "Lymbo", code: -1, userInfo:
[NSLocalizedDescriptionKey: "Impossibile convertire l'immagine"])) )
        return
    }
    // Metadati opzionali (tipo MIME)
    let metadata = StorageMetadata()
    metadata.contentType = "image/jpeg"
    // Carica i dati sul percorso specificato
    storageRef.putData(imageData, metadata: metadata) { metadata, error in
        if let error = error {
            print("Errore upload foto: \(error.localizedDescription)")
            completion(.failure(error))
        } else {
            // Upload riuscito, ottenere l'URL di download
            storageRef.downloadURL { url, urlError in
                if let urlError = urlError {
                    print("Errore ottenendo URL download: \
(urlError.localizedDescription)")
                    completion(.failure(urlError))
                } else if let downloadURL = url {
                    print("Foto profilo caricata. URL: \
(downloadURL.absoluteString)")
                    completion(.success(downloadURL))
                    // (Facoltativo) Salva l'URL nel documento profilo utente
su Firestore
                    let db = Firestore.firestore()
```

```
db.collection("users").document(userId).updateData(["photoURL":
downloadURL.absoluteString]) { err in
                        if err == nil {
                            print("URL foto profilo salvato nel profilo
utente.")
                        }
                    }
                }
            }
        }
    }
}
```

Dettagli:
- Usiamo `Storage.storage().reference().child("profiles/<uid>/photo.jpg")` per definire il percorso del file. Sostituendo `<uid>` con l'userId reale, ogni utente avrà la propria cartella. (Questo non è obbligatorio, ma è una buona organizzazione evitare conflitti di nomi).
- `image.jpegData(compressionQuality: 0.8)` converte l'immagine in un `Data` JPEG con compressione 80%. Puoi scegliere PNG (`pngData()`) se preferisci qualità senza perdita (ma file più grandi).
- `putData` carica i bytes nel Storage. Accetta anche i metadati: qui impostiamo il contentType per chiarezza.
- Nel completion, se non c'è errore, chiamiamo `storageRef.downloadURL` per ottenere l'URL pubblico del file appena caricato. Questo URL potrà essere salvato nel profilo utente e usato per caricare l'immagine nelle UI (è un link HTTPS accessibile pubblicamente se le regole Storage permettono letture pubbliche o se l'utente è autenticato – nel nostro caso la regola richiede auth, ma l'SDK la soddisfa quando l'utente è loggato).
- Infine, in caso di successo, facoltativamente aggiorniamo il documento Firestore dell'utente aggiungendo (o settando) il campo `photoURL` con l'URL ottenuto. Così altri componenti dell'app possono leggere direttamente l'URL dal profilo senza dover interrogare lo Storage ogni volta.

Nel completion handler usiamo un `Result<URL, Error>` per restituire o l'URL o un errore, per comodità d'uso.

Nota di sicurezza: poiché nelle regole Storage abbiamo `allow read, write: if request.auth != null`, l'URL di download ottenuto sopra funzionerà solo se l'utente è autenticato (gli URL Firebase contengono un token di accesso legato all'utente). Per permettere a *chiunque* di vedere le foto profilo senza autenticazione, dovresti impostare le regole di Storage in modo più permissivo per quel path specifico, oppure usare Cloud Storage con file pubblici. Nell'MVP, possiamo accettare che solo utenti loggati vedano le foto degli altri (essendo un dating app ha senso che bisogna essere autenticati per vedere gli altri).

3.5 Ricezione di Notifiche Push (Firebase Cloud Messaging)

Integrare le **notifiche push** con Firebase Cloud Messaging (FCM) richiede sia configurazioni lato app (richiesta permessi, registrazione) che lato server (invio notifiche). Qui ci occupiamo della parte client, ovvero come ricevere notifiche e ottenere i token dispositivo.

Abbiamo già attivato la capability Push Notifications e caricato la chiave APNs su Firebase (sezione 2.2). Inoltre, abbiamo incluso il pacchetto `FirebaseMessaging` nel progetto. Firebase Messaging si occupa di ottenere un **FCM Token** univoco per il dispositivo e di gestire l'interfacciamento con APNs.

Nell'**AppDelegate** implementeremo i metodi delegati necessari:

1. **Richiesta dei permessi all'utente** per le notifiche: subito dopo l'inizializzazione di Firebase (in `didFinishLaunching`), richiediamo l'autorizzazione a inviare notifiche.
2. **Registrazione al servizio di notifiche di iOS (APNs)**: se i permessi sono concessi, registriamo l'app per le notifiche remote con `UIApplication.shared.registerForRemoteNotifications()`. Questo farà sì che iOS contatti APNs e ci fornisca un *device token*.
3. **Comunicare il device token ad FCM**: implementiamo `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)` per passare il token APNs a Firebase Messaging. In questo modo, Firebase lo associa al tuo FCM token internamente [12].
4. **Ottenere il FCM Registration Token**: implementiamo il delegate di `Messaging` per ricevere il token FCM. Possiamo stamparlo o inviarlo al nostro server se faremo invii mirati.
5. **Gestire la ricezione delle notifiche**: implementiamo i metodi di `UNUserNotificationCenterDelegate` per gestire le notifiche in arrivo sia in foreground (`willPresent`) che l'evento di apertura da background (`didReceive response`).

Ecco un possibile AppDelegate con questi componenti (solo le parti rilevanti):

```
import UIKit
import FirebaseCore
import FirebaseMessaging
import UserNotifications

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,
UNUserNotificationCenterDelegate, MessagingDelegate {

    // ... altre funzioni AppDelegate ...

    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Configurazione Firebase
        FirebaseApp.configure()
        // Imposta i delegati per Messaging e notifiche
        Messaging.messaging().delegate = self
        UNUserNotificationCenter.current().delegate = self

        // Richiedi autorizzazione per notifiche push
        UNUserNotificationCenter.current().requestAuthorization(options:
[.alert, .badge, .sound]) { granted, error in
            if granted {
                print("  Permesso notifiche push CONCESSO")
                // Registrazione al servizio APNs (ottenere device token)
```

```
                DispatchQueue.main.async {
                    application.registerForRemoteNotifications()
                }
            } else {
                print("  Permesso notifiche push NEGATO: \
(error?.localizedDescription ?? "unknown")")
            }
        }
        return true
    }

    // Chiamato quando APNs ha registrato con successo il dispositivo e
ottenuto un device token
    func application(_ application: UIApplication,
                     didRegisterForRemoteNotificationsWithDeviceToken
deviceToken: Data) {
        // Passa il token del dispositivo ad FCM
        Messaging.messaging().apnsToken = deviceToken
        // Nota: FCM effettua internamente il mapping token APNs -> token FCM
        // Non c'è bisogno di chiamare Messaging.messaging().token qui, lo
faremo nel delegate
    }

    // MARK: - MessagingDelegate

    // Chiamato quando FCM assegna o aggiorna il token di registrazione per
questo dispositivo
    func messaging(_ messaging: Messaging, didReceiveRegistrationToken
fcmToken: String?) {
        print("FCM Registration Token ricevuto: \(fcmToken ?? "")")
        // Puoi inviare questo token al tuo server per registrare l'utente e
inviare notifiche mirate
        // Ad esempio, salvarlo nel profilo utente Firestore:
        if let uid = Auth.auth().currentUser?.uid, let token = fcmToken {
            let db = Firestore.firestore()
            db.collection("users").document(uid).updateData(["fcmToken":
token])
        }
    }

    // MARK: - UNUserNotificationCenterDelegate

    // Ricezione di una notifica mentre l'app è in foreground
    func userNotificationCenter(_ center: UNUserNotificationCenter,
                                willPresent notification: UNNotification,
                                withCompletionHandler completionHandler:
@escaping (UNNotificationPresentationOptions) -> Void) {
        // Mostra la notifica anche se app è attiva (banner + suono)
        completionHandler([.banner, .sound])
    }
```

```
    // Gestione del tap sull notifica (quando l'app era in background/chiusa
ed è stata aperta tramite notifica)
    func userNotificationCenter(_ center: UNUserNotificationCenter,
                                didReceive response: UNNotificationResponse,
                                withCompletionHandler completionHandler:
@escaping () -> Void) {
        let userInfo = response.notification.request.content.userInfo
        // Puoi esaminare userInfo per capire quale azione intraprendere
        if let matchId = userInfo["matchId"] as? String {
            // Esempio: la notifica riguarda un nuovo match, apri schermata
match
            print("Notifica di match ricevuta, ID match: \(matchId)")
            // TODO: naviga alla schermata dei match se necessario
        }
        completionHandler()
    }
}
```

Commenti al codice:

- Impostiamo `Messaging.messaging().delegate = self` e `UNUserNotificationCenter.current().delegate = self` in didFinishLaunching, in modo che l'AppDelegate riceva gli eventi di messaggistica e di notifiche.
- `requestAuthorization(options:.alert, .badge, .sound)` chiede all'utente il permesso per mostrarle. Il closure ci dà `granted` booleano. Se l'utente consente, chiamiamo `registerForRemoteNotifications()` (sul main thread) per registrare il dispositivo su APNs. Se nega, ci limitiamo a loggare. In produzione, potresti voler gestire il caso negato informando l'utente che le notifiche sono disabilitate.
- `didRegisterForRemoteNotificationsWithDeviceToken`: viene passato un `Data` con il token APNs assegnato. Lo comunichiamo a Firebase tramite `Messaging.messaging().apnsToken = deviceToken`. **Importante:** Firebase di default fa *swizzling* dei metodi AppDelegate per intercettare questo token e fare la stessa cosa automaticamente. Avendo impostato il delegate manualmente, effettuiamo noi il passaggio (il codice sopra è corretto sia che il swizzling sia attivo o meno). In pratica stiamo associando il token APNs con il nostro istanza FCM [12].
- `messaging(_:didReceiveRegistrationToken:)`: qui otteniamo il **FCM token** (una stringa lunga). Questo token identifica in modo univoco l'installazione dell'app presso FCM. Lo stampiamo e, ad esempio, lo salviamo nel profilo utente su Firestore (campo `fcmToken`) – utile se vogliamo inviare notifiche mirate ad uno specifico utente dal nostro server (possiamo recuperare il token dal suo profilo). In un MVP, potresti non avere un tuo server; in tal caso puoi inviare notifiche manualmente dal Firebase Console usando questo token, per test.
- `userNotificationCenter(_:willPresent:withCompletionHandler:)`: questo metodo del delegate UNUserNotificationCenter viene chiamato quando arriva una notifica *mentre l'app è in primo piano*. Di default, le notifiche push **non** mostrano banner/suoni se l'app è aperta, a meno che tu non decida di mostrarle. Qui, chiamiamo `completionHandler([.banner, .sound])` per far comparire il banner di notifica e il suono anche in foreground. Se preferisci silenziare in app aperta, puoi passare un'opzione vuota `.init()` (o chiamare il completionHandler senza opzioni).
- `userNotificationCenter(_:didReceive:withCompletionHandler:)`: chiamato quando l'utente interagisce con la notifica (tap sul banner o sull'avviso quando app era in background, oppure su un'azione custom). Possiamo ispezionare `response.notification.request.content.userInfo` per vedere i dati della notifica. Nell'esempio, immaginiamo che le notifiche di nuovo match includano un campo `matchId` nel payload (chiavi personalizzate che puoi definire quando invii la notifica). Se

presente, potresti navigare l'utente alla schermata del match corrispondente. Dopo aver gestito, chiamiamo sempre `completionHandler()` per permettere al sistema di completare l'elaborazione.

Con questo setup, la tua app è pronta a **ricevere notifiche push**. Puoi testare inviando una notifica di prova dal Firebase Console: vai su Cloud Messaging > New Notification, seleziona la tua app iOS come target e invia (assicurati che l'app sia installata su un dispositivo reale e aperta o in background; le notifiche *non funzionano su simulatore*). In alternativa, puoi utilizzare strumenti come *cURL* o Postman per chiamare l'API HTTP v1 di FCM usando il token dispositivo stampato nei log.

4. Struttura del Progetto e Organizzazione dei File

Per concludere, proponiamo una possibile organizzazione dei file e delle cartelle nel progetto Xcode Lymbo, in linea con quanto implementato:

```
Lymbo/                  (directory radice del progetto Xcode)
├── Firebase/           (configurazioni Firebase)
│   ├── firebase.json
│   ├── .firebaserc
│   ├── firestore.rules
│   └── storage.rules
├── Lymbo.xcodeproj     (file di progetto Xcode)
└── Lymbo/              (source dell'app)
    ├── AppDelegate.swift       (inizializzazione Firebase, Messaging
delegate)
    ├── SceneDelegate.swift     (se app non SwiftUI pura, generato da Xcode)
    ├── Models/
    │   └── User.swift          (modello dati utente, es. struct Codable per
profilo)
    ├── Services/              (layer per comunicare con Firebase)
    │   ├── AuthService.swift       (funzioni di signUp, logIn, signOut)
    │   ├── UserProfileService.swift (funzioni createUserProfile,
updateUserProfile, fetchUserProfile)
    │   ├── MatchService.swift      (funzione sendLike e eventuali fetch di
match)
    │   └── StorageService.swift    (funzione uploadProfileImage, download
image helper)
    ├── Controllers/           (o Views/ se SwiftUI - componenti UI)
    │   ├── LoginViewController.swift   (schermo di login che usa
AuthService)
    │   ├── SignUpViewController.swift  (schermo di registrazione che usa
AuthService e UserProfileService)
    │   ├── ProfileViewController.swift (schermo profilo utente che usa
StorageService per upload foto, ecc.)
    │   └── MatchesViewController.swift (lista dei match, usa MatchService
per caricare)
    └── Resources/
        └── GoogleService-Info.plist    (configurazione Firebase)
```

(La struttura esatta può variare; l'importante è separare concettualmente la logica di Firebase – raccolta nei Services – dalla logica di UI. Ad esempio, `AuthService` *potrebbe essere una classe singleton o un insieme di funzioni statiche per gestire login/registrazione;* `MatchService` *idem per likes e matches. I controller o le view chiamano questi servizi.)*

Nella cartella **Firebase/** teniamo i file di configurazione e regole in modo ordinato. Questi file *non* vanno inclusi nel bundle dell'app, ma vanno versionati nel controllo sorgente per avere traccia delle regole di sicurezza. Il file `firebase.json` e `.firebaserc` servono solo in locale per la CLI.

Puoi creare un file `README.md` nel repository con le istruzioni di setup (simile a questa guida) in modo che altri sviluppatori possano configurare l'ambiente velocemente. Include ad esempio: come installare la CLI, come eseguire i comandi di init, come aggiungere il plist al progetto, ecc. Gran parte di quelle informazioni sono già state descritte sopra.

Nota finale: Abbiamo privilegiato l'automazione e gli script CLI per configurare Firebase. Allo stato attuale, l'unica operazione manuale necessaria dovrebbe essere: - Abilitare Email/Password in Firebase Authentication (via console Firebase). - Caricare la chiave APNs per FCM (via console Firebase). - Abilitare le capability Push/Background in Xcode (una volta, nel progetto).

Tutto il resto – creazione progetto, app iOS, download config, regole di sicurezza – è scriptabile e documentato.

Con questa base, il tuo MVP di Lymbo ha: - Autenticazione utenti funzionante (registrazione/login) [10] [11] - Database Firestore per profili e meccanica di like/match - Storage per le foto profilo - Notifiche push integrate per eventi come nuovi match

Buon sviluppo! Se segui passo passo la guida e utilizzi i codici di esempio come base, dovresti ritrovarti con un progetto iOS pronto e connesso a Firebase, con il minimo sforzo manuale e un massimo di configurazione automatizzata.

---

[1] [2] GitHub - nrikiji/flutter-starter: flutter starter-kit. flutter starter project. upload to store using github actions. Switch between development and production environments and incorporate firebase sdk.
https://github.com/nrikiji/flutter-starter

[3] [4] [5] [6] Blank page after successful Firebase deployment
https://groups.google.com/g/firebase-talk/c/9YJQgEEwdqc

[7] [8] [10] [11] Get Started with Firebase Authentication on Apple Platforms
https://firebase.google.com/docs/auth/ios/start

[9] Add Firebase to your Apple project | Firebase for Apple platforms
https://firebase.google.com/docs/ios/setup

[12] Receive messages using Firebase Cloud Messaging
https://firebase.google.com/docs/cloud-messaging/receive-messages