Matchup Firebase Data Structure

General Guidelines

Guidelines are derived from Firebase recommendations (https://firebase.google.com/docs/database/web/structure-data).

1. Data should be in a structure that is easy to be aggregated and filtered.
2. As there are no JOIN queries, data that should be grouped together should be accessed easily. If it's a small amount of non-repetitive data - then by nesting; otherwise - using a foreign key.
3. Unless stated otherwise, all of the collections should have a "last_updated" and "created_at" date fields (and thus, for readability purposes, those are not specified below).
4. **We should avoid nesting data as much as possible.**
5. **Conversations should be E2E encrypted.**

Technologyא

Options available in Firebase (see https://firebase.google.com/docs/database/rtdb-vs-firestore)

1. Cloud Firestore: is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.
2. Realtime Database is Firebase's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in realtime.

Decision:

1. **General database**: Firestore.
2. **Messaging-only database**: Firebase Realtime Database.

Warning: Firebase Realtime Database support up to 200k connections.

Data Structure

invite

Propose: Contains registration invites.
Data Structure:

```
{
     "id": number,
     "code": string, // the invite code
     "sender_id": number, // the matcher who sent the invite, can be zero
for system-generated.
     "expiry_date": date
}
```

Indexes: code.
Notes: after using an invite, this entry should be removed.

matchers

Propose: Contains profile information about the matchers.
Data Structure:

```
{
    "id": number,
    // Basic information
    "full_name": string,
    "phone_number": string,
    "gender": number, // enum { Male, Female }
    "birth_year": number,

      // Foreign keys to other data
     "represented_singles": represented_single[]
}
```

Indexes: phone_number.
Notes: This collection should be tied to the Firebase Authentication user so that there's a bijection function from the Firebase Authentication user id to the matcher id. See

- https://firebase.google.com/docs/auth/web/phone-auth
- https://stackoverflow.com/questions/49538158/how-to-add-custom-profile-details-to-the-user-in-firebase

Nested type – represented_single:

```
{
     "id": number, // the id from the single_profile collection
     "approved": number,
}
```

Indexes: status, id + status.

singles

Propose: Contains basic information about singles. This collection is a bijection from single humans to entities, so John Doe will have one and only one entity here, even if he is being suggested for a match by multiple matchers. For the various profiles that a single has, see “single_profiles”.

Data Structure:

```
{
     "id": number,
      // Basic info
     "phone_number": string,
     "associated_profiles": associated_profiles[],

     // Matching data
     "proposed_singles": number[], // ids of singles (from the singles
collection)
     “Proposed_singles_data”: proposed_single[], // array of rejected
proposed singles
     "last_proposed_match": date,
}
```

Indexes: phone_number.

Nested type – associated_profiles:

```
{
     "id": number, // the id field from the single_profiles, associated
with this entry
     "matcher_id": number, // the id field from the matchers collection.
}
```

Indexes: matcher_id.

Nested type – proposed_single:

```
{
     "id": number, // the id field from the single_profiles, associated
with this entry
     "reason": number, // enum { MatcherDeclined, TimeRanOutForMatchers,
TimeRanOutForSingles, SingleDeclined }
     "rejected_id": number, // matcher id or single id of the person who
were rejected
     "rejector_id": number, // matcher id or single id of the person who
rejected
    “Rejected_reason”: number, // the rejected reason, enum { … } ,
```

```
default 0 if this won’t been done by singles or not specified
}
```

Indexes: matcher_id.

single_profiles

Propose: Contains profile information about the singles. This collection contains a “matcher biased profile” of the single. Thus, we might one human single, with multiple entries in this table, one for each matcher. For example, Jane Doe is a matcher. She registered John as a single and thus John Doe now has an entity here. Then, Yuval comes and registers as a matcher as well – and wants to add John Doe too. Thus, John Doe – a single human – has two entities here. The collection that actually has bijection relationship between single humans and entities (1 single has 1 entry) is the “singles” collection.

Data Structure:

```
{
     "id": number,
     "phone_number": string,

     // Basic details
     "full_name": string,
     "gender": number, // enum { Male, Female }
     "birth_year": number, // we get an age from the user, calculate
"date('Y') - age" and store it
     "location": number, // location id (from the "locations" collection).
     "workplace": string,

     // Extra info
     "extra_info": string, // extra textual input that the matcher tells
about the single
     "extra_info_recorded": string, // path to blob storage that contains
the recording

     // Algorithm factors. Numbers from 1 to 10 except said otherwise
     "activeness_factor": number, // "what are you doing at nights" page
     "religion_factor": number, // "your way of life"; enum { ... }
     "political_factor": number, // "who you voted for"
     "body_structure_factor": number,
     "attractiveness_factor": number,
     "personality_traits": number[], // array of personality_trait ids.
     "unspoken_traits": number[], // array of unspoken_traits ids.
```

```
      // Data that is being entered by the single, in the Web interface
    "profile_pictures": profile_picture[],
    "height": number, // shouldn't be decimal (save 170 for 1.70 etc.).
    "looking_for_gender": number[], // array of applicable genders ([0],
[1] or [0, 1])
    "looking_personality_traits": number[], // ids of personality_traits
ids.
}
```

Nested type – profile_picture:

```
{
      "url": string, // path to the profile picture blob storage URL
      "width": number,
      "height": number,
      "order": number, // the position of the image. 0 is the primary.
}
```

locations

Propose: Contains a list of available locations that are permitted to use the app (for the selection in the profile creation).
Data Structure:

```
{
     "id": number,
     "key": string
}
```

Notes:

- No need to created_at and last_updated here.
- The key is used as a localization key in the app.

personality_traits & unspoken_trraits

Propose: Two collections, each has a list of available traits.
Data Structure:

```
{
     "id": number,
     "key": string
}
```

Notes:

- No need to created_at and last_updated here.
- The key is used as a localization key in the app.

phone_verification

Propose: Contains phone verification requests.
Data Structure:

```
{
     "id": number,
     "phone_number": string,
     "code": string
     "attempts": number,
     "last_attempt": number,
     "created_at": date,
}
```

Indexes: phone_number, phone number + code.
Notes: We shall use Firebase native phone verification API if possible, instead of implementing this feature directly.

active_proposals

Propose: A collection that contains the current active matching proposals.
Data Structure:

```
{
    "id": number,
    "lhs_single": number, // The first single id, from the singles
collection
    "rhs_single": number, // The second single id, from the singles
collection
    "lhs_matcher": number, // The first single id, from the singles
collection
    "rhs_matcher": number, // The second single id, from the singles
collection,

    "lhs_current_state": number, // enum {Proposal, Discussion, Verify,
Sign-Off}
    "rhs_current_state": number, // enum {Proposal, Discussion, Verify,
Sign-Off}
```

```
    "conversation_id": number, // The conversation id. This is a Firebase
Realtime Database entry!
    "expiry_time": number,
}
```

Indexes:

1. lhs_single
2. rhs_single
3. lhs_matcher
4. rhs_matcher
5. lhs_current_state
6. rhs_current_state
7. expiry_time

Notes:

1. The algorithm part will fill this table, and take care of removing old proposals.
2. We have four phases:
    a. Proposal: match is shown as a proposal to the matcher.
    b. Discussion: matchers are in a discussion using a chat window. To show the chat window, both matchers should be in a state >= “Discussion”.
    c. Verify: matcher verified that she agrees to proceed and deliver the proposal to the single.
    d. Sign-Off: The single approved the match proposal.

    Movement should be made as follows:
    e. Proposal → Discussion: **matcher** agreed to the proposal.
    f. Discussion → Verify: **matcher** asked to proceed and send the proposal to the single.
    g. Verify → Sign-Off: **single** agrees to the match.

    Transition events:
    1. When both matchers switched from “Proposal” to “Discussion”, a notification should be sent to inform both that they got an approved match so they can discuss it.
    2. When both matchers verified (lhs_state == Verify && rhs_state == Verify) then a WhatsApp message should be sent to the singles to propose the match to them.
    3. When both singles verified, a WhatsApp message should be sent to both singles, with the phone number of the other single.

conversations

Propose: A collection that contains information about the conversation between two matchers.
Data Structure:

```
{
    "id": number,
    "lhs_single": number, // The first single id, from the singles
collection
    "rhs_single": number, // The second single id, from the singles
collection
    "lhs_matcher": number, // The first single id, from the singles
collection
    "rhs_matcher": number, // The second single id, from the singles
collection,
     "Messages": message[], // A nested message entries array
     "expiry_time": da
}
```

Notes:

1. This is a Firebase Realtime Database collection.
2. The singles and matchers entries are ids from Firestore.

Nested type – message:

Data Structure:

```
{
     "id": number,
    "message_type": number, // enum { Text, Recording, Picture }
     "sender_id": number, // matcher id
    "content": string, // Encrypted value
}
```

Notes:

1. The messages must be E2E encrypted using asymmetric encryption.
2. If the message type is recording or image, then the “content” will be a URL to a blob storage that contains the image / recording.

Feature Flags (FF)

These are additional feature flags that should be allowed with Firebase Configurrations module.

Key	Type	Default	Description
registration_open	bool	1	Determine if new users can be register (regardless of invitations).

invite_only_registration	bool	1	Determine if new users can be register without invites.
proposal_expiry_date	number	24 hours	Determine how much time it takes for a matching proposal to be expired.

Metrics

Key	Description
issued_invites	The number of invites issued by users
used_invites	The number of actually used invites