Tinder

First step is to take a step back and think of this system in a very logical way. Think about what your users need as features. Think about how your services will be broken down to fill these features. Then think about their individual requirements per service. This allows the design to be more flexible.

- What kind of features will I provide for this app?
- What is the flow of the application?
- Will we be storing profiles?
- SInce it is a dating application, are we going to be storing images in that profile?
    - How many images per user do we want?
        - 5 images per user

To get ideas on how to continue, think about the application flow as a user using the application. Visit the website -> Create a profile (Adding images) -> Accept/Reject people

- From here, we could have an idea for recommending matches as a service
- A follow up question could be, how many users are we designing this for?

- Next, we could think about once two people have matched, we have to note down those matches and do something with them.

- Then, we could set up a direct messaging service between those two people.

Features:

1. Store Profiles
2. Recommend Matches
3. Note Matches
4. Direct Messaging

Storing Images

- Storing images only has one important question in it; how are we going to store images?
- There will be a lot of images. 5 images per user
- We can store images either as a file in a file system or as a blob in a database https://habiletechnologies.com/blog/better-saving-files-database-file-system/

File vs Blob
Databases can offer:

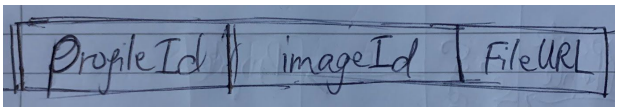
1. Indexing for faster search
    - However it's better to just store the image as a file since we're not going to be searching through the content within the image. It's all going to be 1's and 0's.
2. ACID transaction properties

- For images, we don't require transaction properties since we're not doing any atomic operations on an image

3. Access control
    - It is less secure to store data as a file but with good security practices, like setting up a good/secure internal file structure can reduce security breach from hackers.

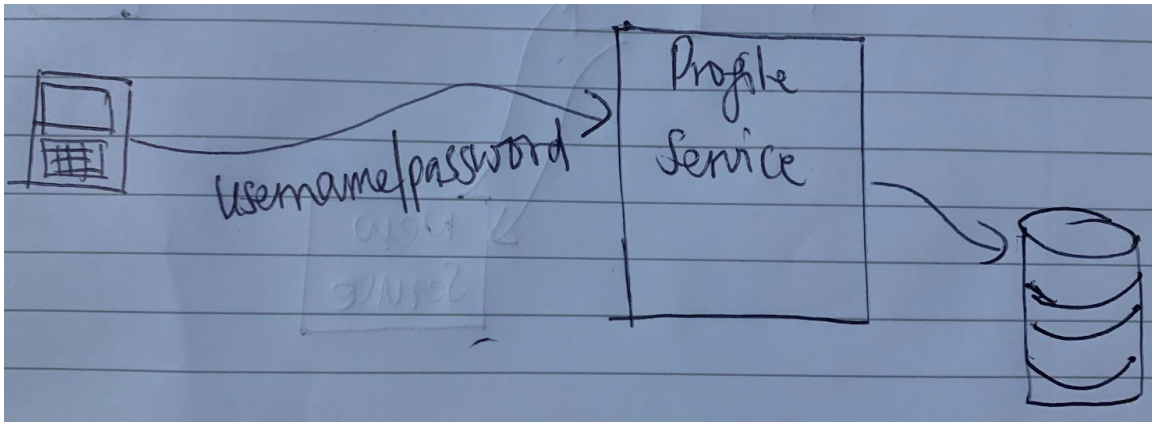
Overall, in this case, it's better to save the images as a file in a filesystem.

1. Files are static so can easily use a CDN (Content Delivery Network) over this. This will enable fast access to images.
2. Our images will be stored in a Distributed File System which is referenced by a file URL saved in a separate database.
    - Each image will have a file URL, imageID and an associated profile ID with it.

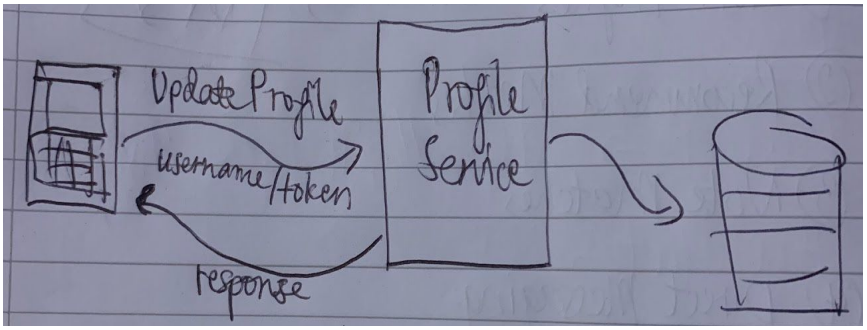


Next, let's illustrate the flow of the application

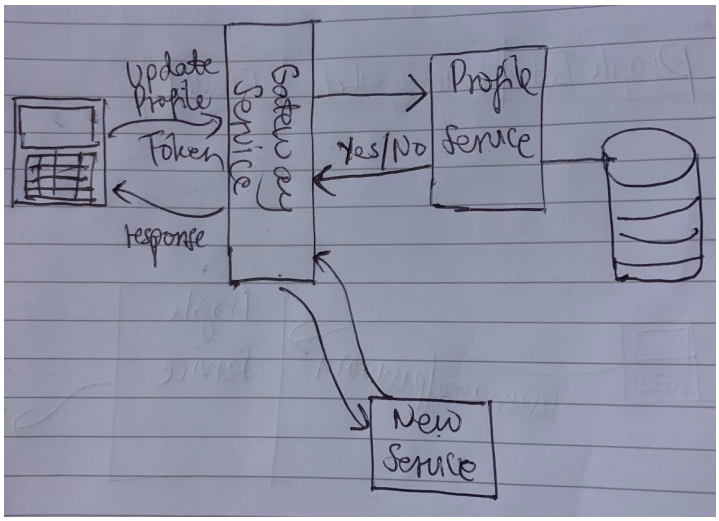
1. We have a client application on the mobile phone.
2. Our user clicks on a button which sends a request to our **Profile** service.
3. It registers itself to our **Profile** service. For example, by registering with a username and password.
4. The **Profile** service stores this data in the database



5. Once a user is stored in the database, the user will then request to update their profile (adding images).
6. For updating the profile, how do we make sure that this is an authenticated request?
    - How do we make sure that the person who is claiming to update their profile is the actual person that the request belongs to.
    - In the **Profile** service, we'll have authentication mechanisms such as sending tokens to authenticate users.

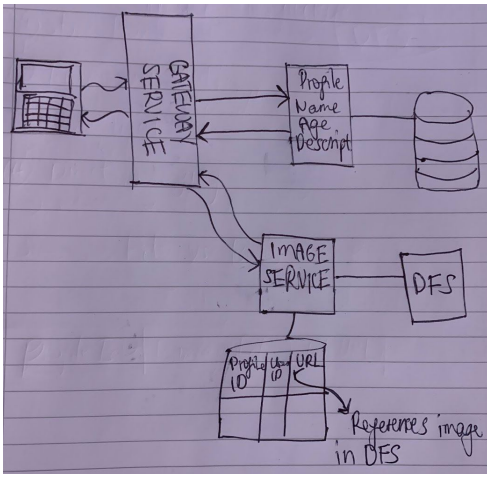


7. The problem is, what if we have an additional service which requires authentication as well?
    - That new service will not have the authentication information so will then have to talk to the **Profile** service every time it receives a request
    - This is a problem because this new service will be relying on the **Profile** service
    - We want to decouple our services as much as we can as it's much easier to manage and scale if services are not dependent/coupled with other services.
    - A solution is setting up a gateway service which acts as an entrypoint for the clients
    - The clients will talk to the gateway. The gateway talks to the **Profile** service, then if the user is authenticated, the gateway will send the response back. In addition, now the new service knows that the user is authenticated through the gateway. It can now send its response without needing to talk to the **Profile** service. It would just send its response to the gateway, then the gateway sends it to the client.



8. By decoupling, what we have done was to remove the need to talk to Profile service for authentication and instead used the gateway instead in case there's other services which require authentication in the future. Another thing is if the new service uses a different messaging protocol, we can easily implement that.
9. Inside our **Profile** service, there is the user's name, description etc.

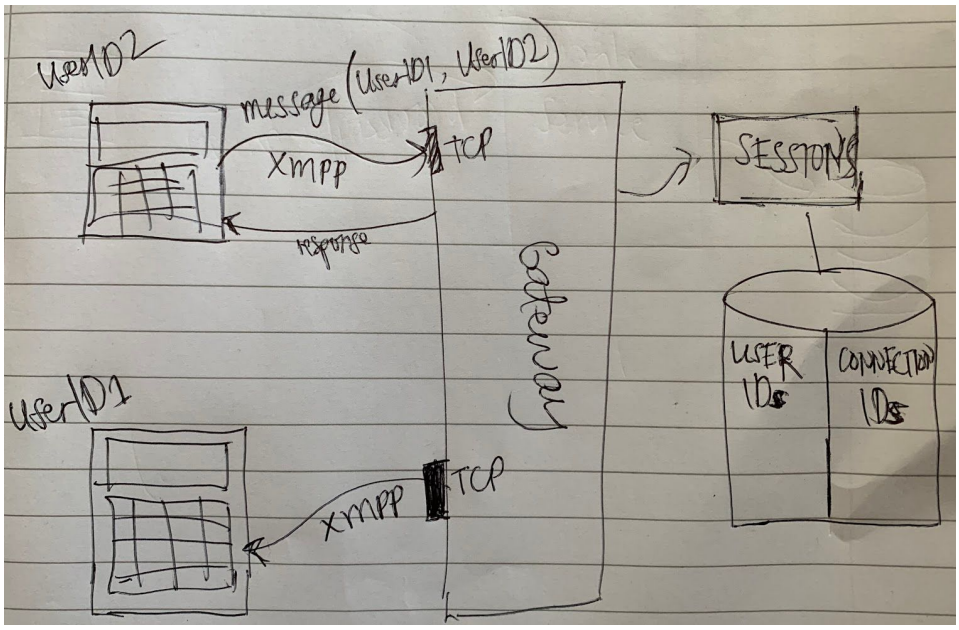
- Do we really want to process our images within our **Profile** service alongside all the other user data?
- Ideally, we would want to store our images in a separate service in case there's another functionality which requires only the images of the user, maybe for machine learning. We don't have to query the entire **Profile** service to get to the image. So the solution is to create a separate **Image** service.
- Another scenario is if we only want the users details, name, age etc for analysis, then transfer that. We don't have to send a response of the entire user details including their images.



Next, we design the **Direct Messaging** feature

- We have to think about how once two people are matched, how is the first client going to communicate with the other client?
- This will mean that User1 will talk to the gateway and say "I want to talk to User2"
- How will User1 talk to User2?
- Usually, we're using **HTTP** protocol as a way of talking between two machines.
- The problem with this way of communication is this is a client-server protocol, which means that the client will always have to request something from the server. The client always has to send a request first, then the server responds to that request. The server doesn't reach out to the client.
- This is not efficient for a chat/messaging feature because in order for User1 to receive its message from User2, it will have to poll the server first; "Hey server, are there any messages for me?".
- We don't want users to poll the server but instead we want the messages to be pushed to the users.
- Instead of a client-server protocol, we could use a peer-to-peer protocol. This is where both the user and the gateway are the same level (peers). If the server needs to send messages to the client, it can.
- A peer-to-peer protocol would be **XMPP** (Extensible Messaging and Presence Control)

- Internally, XMPP will be creating a TCP connection with the gateway. This is because the user will have to create a connection and maintain it.
- With these connections, the gateway can now talk to the clients.
- We need some entity that will store these sessions in order to figure out which connection they are listening to.
- This will be some kind of **session** service that will store all the connectionIDs and then from this, we can connect the two connections of the two users that have matched. This is enough for a direct messaging feature.



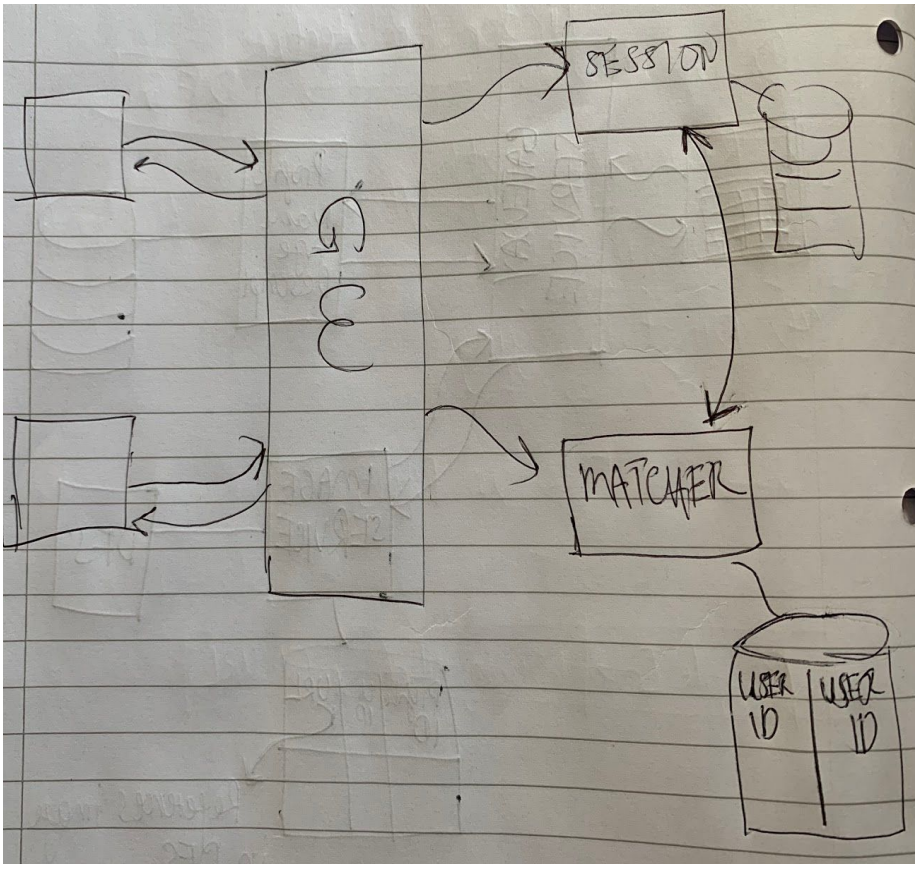
Matching Feature

- For the matching feature, we could save the clients matches on the client's side, if they've matched with someone. However, if the client uninstalls the app, it will lose its data, all the records of that client's matches will be lost.
- Instead, we want this data to be stored on the server side. We'd create a service which will note down all the matches that the client has. This is so that if the client's app is uninstalled, then re-installed, all the data will just get pulled back from the **matcher** service.
- This **matcher** service will keep a table which contains UserID-UserID records for people who have matched. For example, User1matched with User2 and vice versa.
- From here, the **matcher** can tell the **session** service which UserIDs can communicate with each other.

- The flow would be:

User tells the server it has matched with someone -> **Matcher** service tells the **session** service to open up a direct messaging feature between the two users that matched.

- Now if the app is uninstalled then reinstalled, all of your matches will be pulled from the **matcher** service, then make sure you can communicate to your matches from the **session** service, then pull out all your profile data from the **Profile** service. The only data you will lose is the record of who you accepted/rejected, which is not critical data.
- This will just mean that you will get re-recommended the same people that you have already swiped.



Recommendation Feature

- This feature is for how the application will recommend people to you. Which accounts will be shown to you to swipe.
- For recommendation, the core thing to figure out is “who are the users close to me?”
- We can use sex and age prefereces by using indexing. But overall, we recommend using the location.
- Since we’re using location as the main way to partition the users that will be recommended with each other, we could use database sharding to partition our data.
- We could use sharding to store all the data within the same area to be in the same database shard.

- So our recommendation engine might store the userID and their location. This might be updated every hour in case the user has changed locations.
- Then depending on the users location, we redirect all that data to a specific shard
- For example, Shard1 = London, Shard2 = Manchester, Shard3 = Portsmouth.
- With this architecture, only the users within a specific shard will show up on each other's apps. For example, Users in London would only see in their phone the people within London.
- Then we could further filter within the database to only recommend using gender/age preferences.
- Sharding comes with the problem of having a single point of failure. To solve this, we could replicate the shards individually which will persist our data.