iMMe

Identity Verification Service

Cohort 2 Group 2

1002189 Li Xingxuan
1002455 Hoong Tian Lerk
1002368 Jiang Jinjing
1002074 Soong Cun Yuan

Table of Contents

Project Description

Our project aims to solve the Know-Your-Customer (KYC) problem by emulating the process which the internet verifies the identity of servers using signed certificates issued by Certificate Authorities. iMMe aims to be a *"Validate Once, Verify Everywhere"* one stop identity verification service where users would have their identity verified once by us and subsequent identity verification can be done through facial recognition using our app. Customers and Businesses can trust our platform to provide accurate identity verification services akin to how clients trust Certificate Authorities to ensure that the identity and public key of servers are authentic and valid. iMMe offers its services to both businesses and consumers by supporting User-User Authentication, User-Device Authentication and User-Web Authentication.

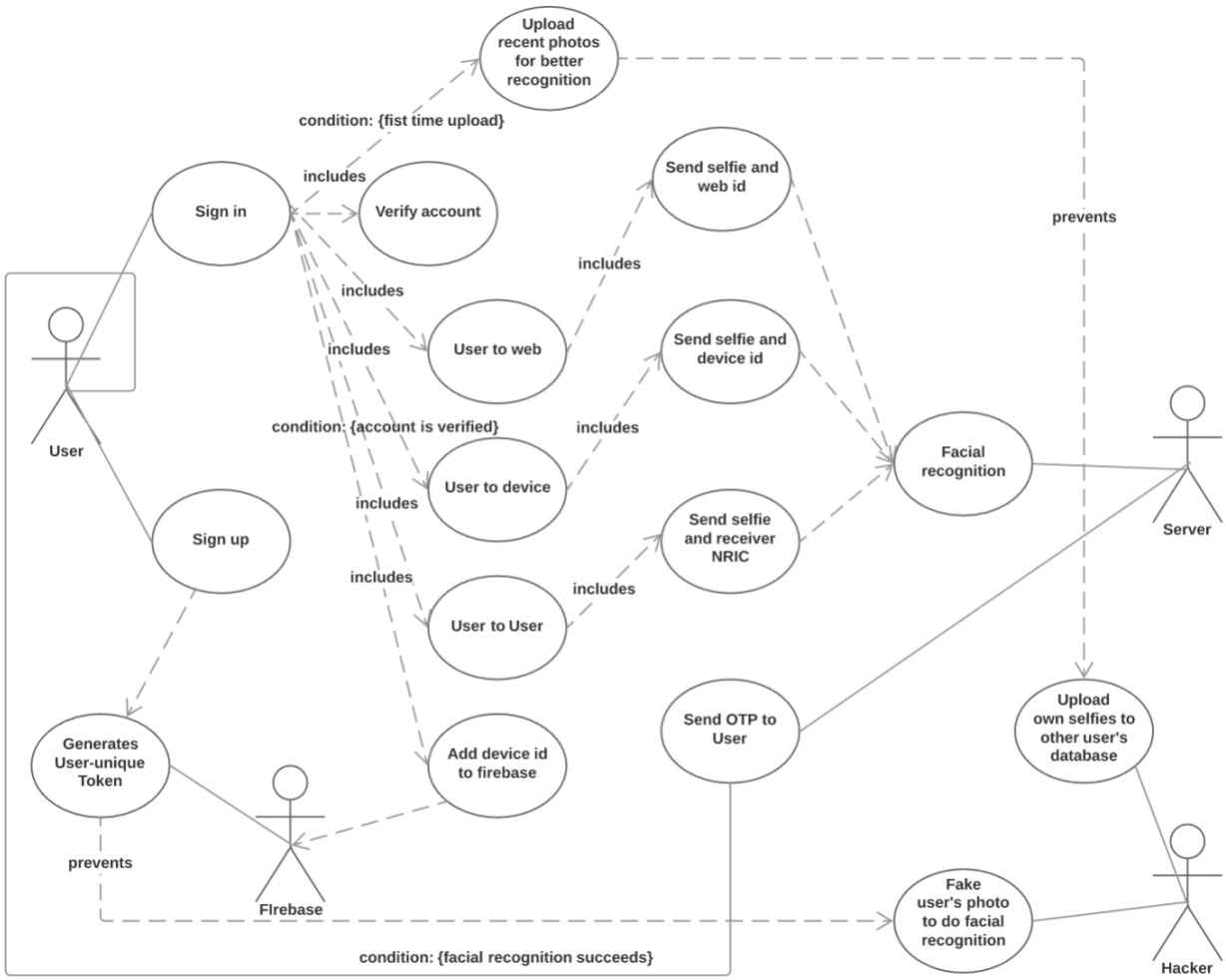
Platforms

The User Interface is developed on the Android platform since most current mobile phones have Near-Field Communication(NFC) and Camera capabilities. Mobile platforms also allow users to utilize our services while on the go.

The backend is developed in Java and Python and is deployed on Google App Engine(GAE). GAE offers automatic scalability and handles the infrastructure and maintenance requirements.

Use Cases

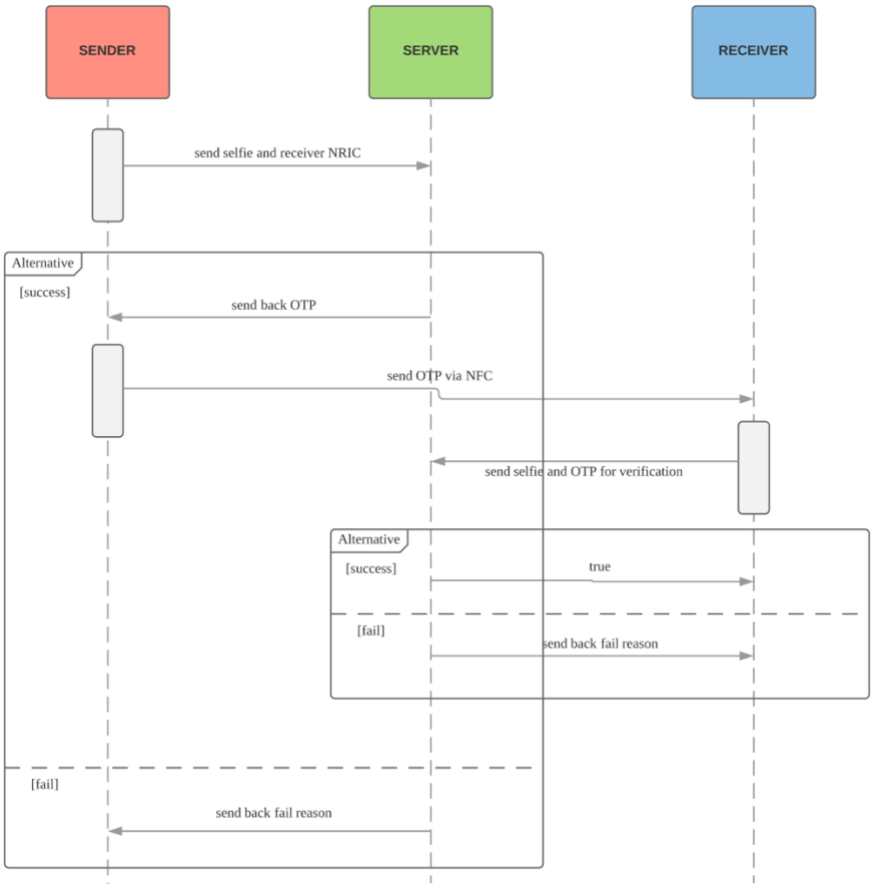
Use case Diagram



Use case: User to User Authentication

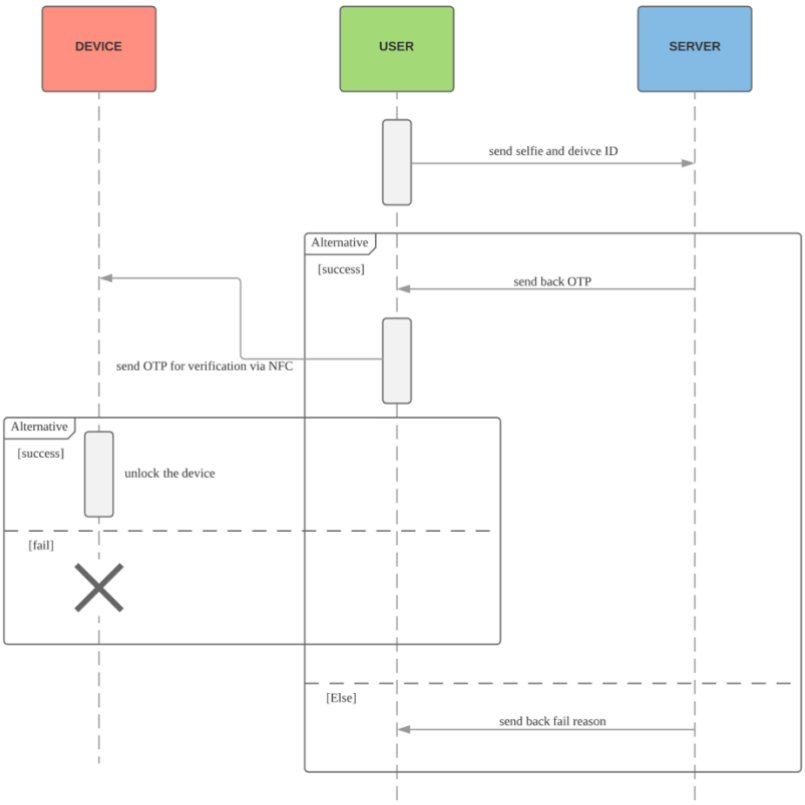
The User to User Authentication is aimed to identify between users. The sender will upload his selfie and receiver's NRIC to our server to both identify himself and notify our server with receiver's information. If the authentication succeeds and the NRIC is valid, our server will send down a time-based OTP to the sender which will be valid for 30 seconds. After that, the sender will pass the OTP to the receiver either with NFC or manually key in. After having the OTP, the receiver will send both his selfie and OTP to our server. Our server will do the authentication again to verify the identity of the receiver.

Our service not only identify the users, but also make sure they are physically in contact.



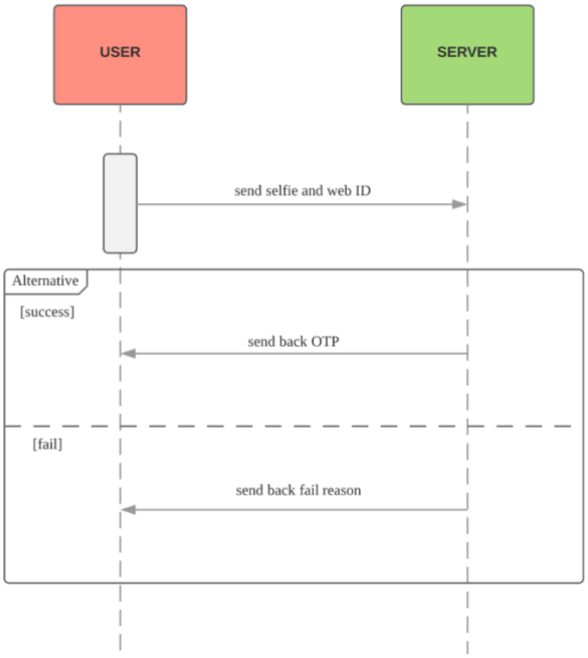
Use case: User to Device Authentication

The User to Device Authentication is aimed to provide identification between user and physical devices. User sends his selfie and device ID to our server for verification. If the authentication succeeds, server will send back a time-based OTP which matches the device ID. After that, send user will send the OTP to the device via NFC, if the OTPs match, the device will be unlocked.



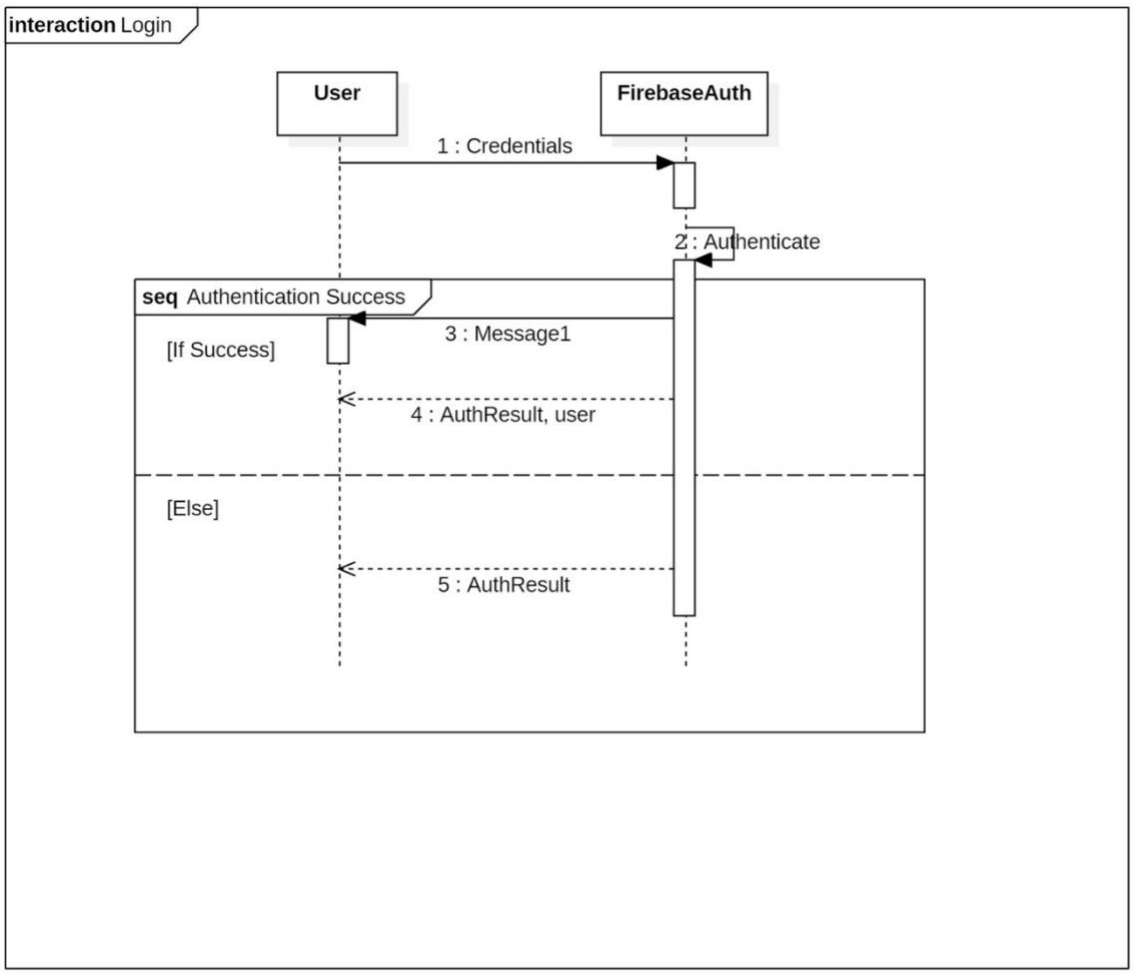
Use case: User to Web Authentication

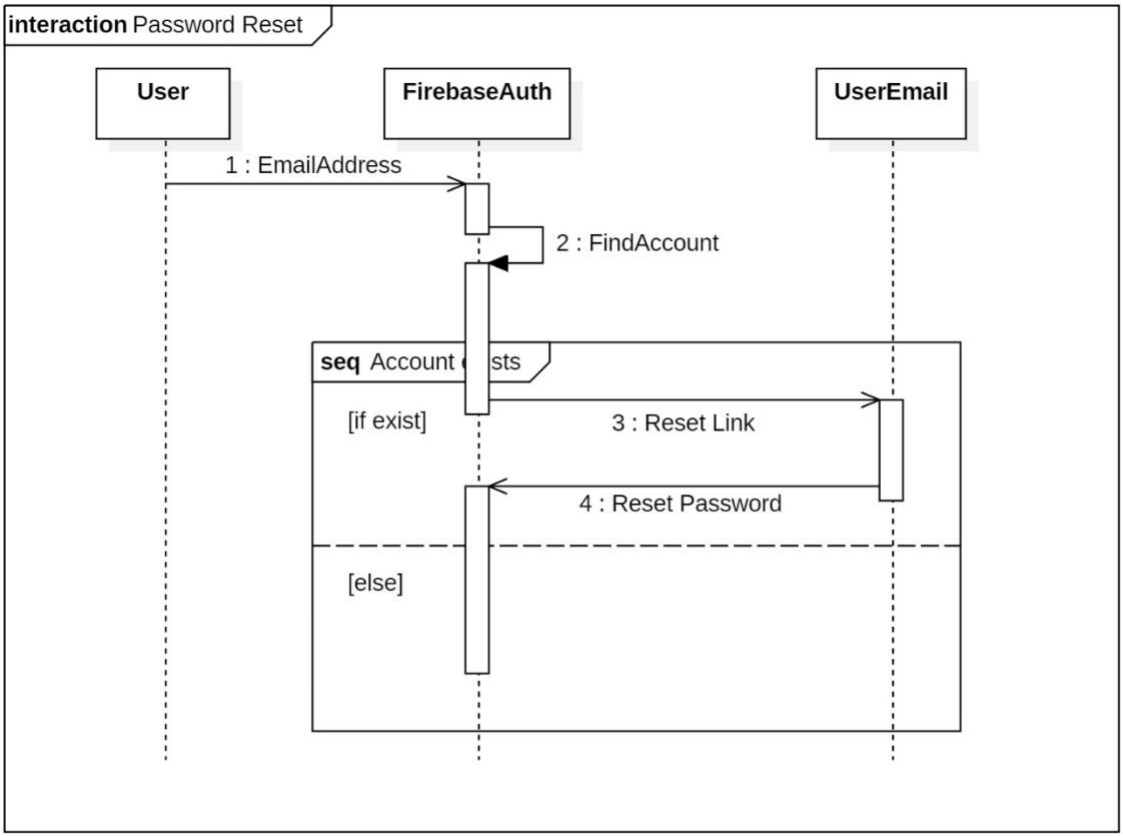
The User to Web Authentication aims to provide an integrated platform for multiple web 2FA services. The user will take a selfie and send the web ID to our server for authentication. If the authentication succeeds, our server will send back the OTP which depends on the 2FA key from the targeted website. One use case could be Banks that require a more secure form of 2FA as the physical token generator currently used by most banks are unable to verify the identity of the user.

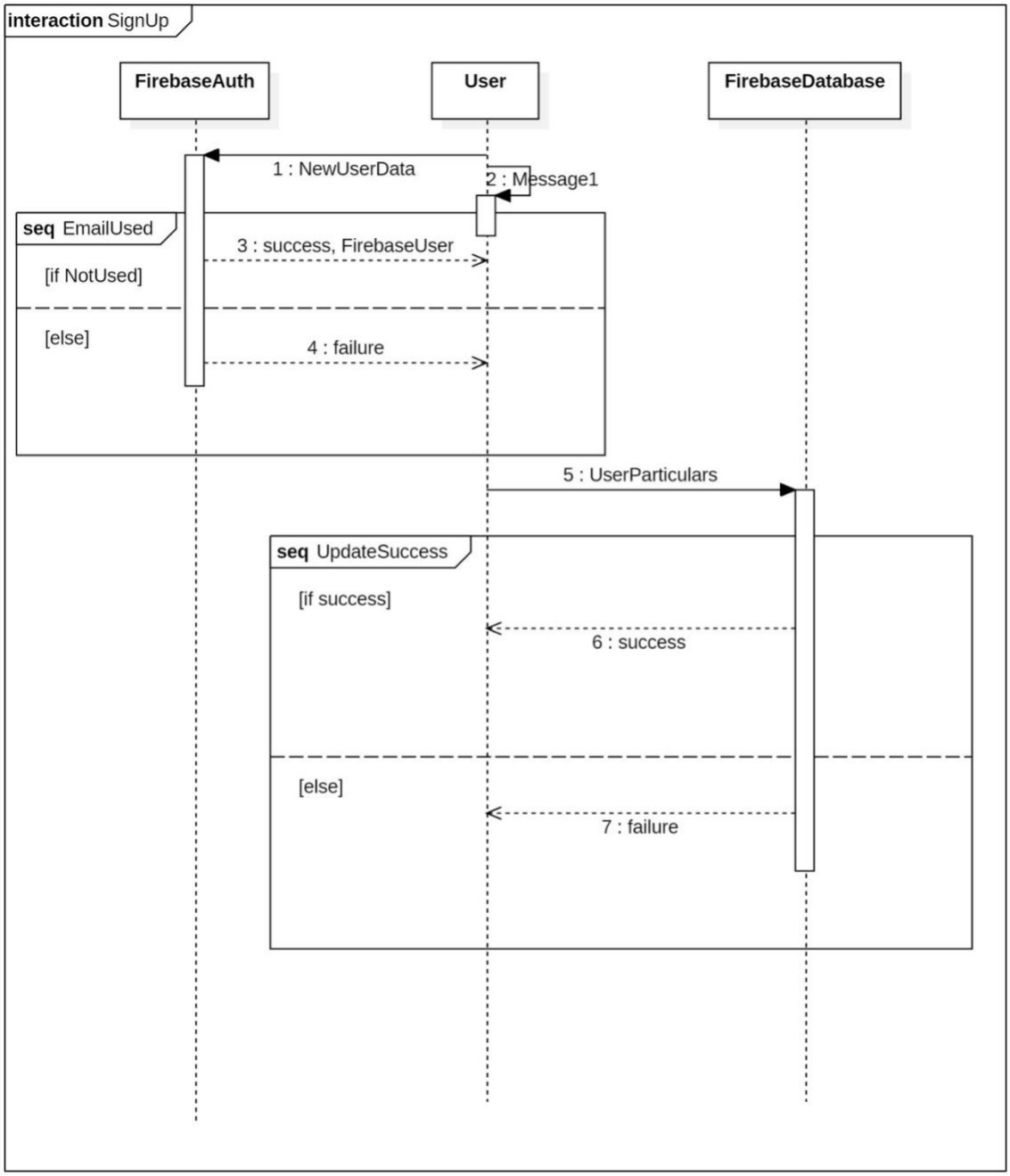


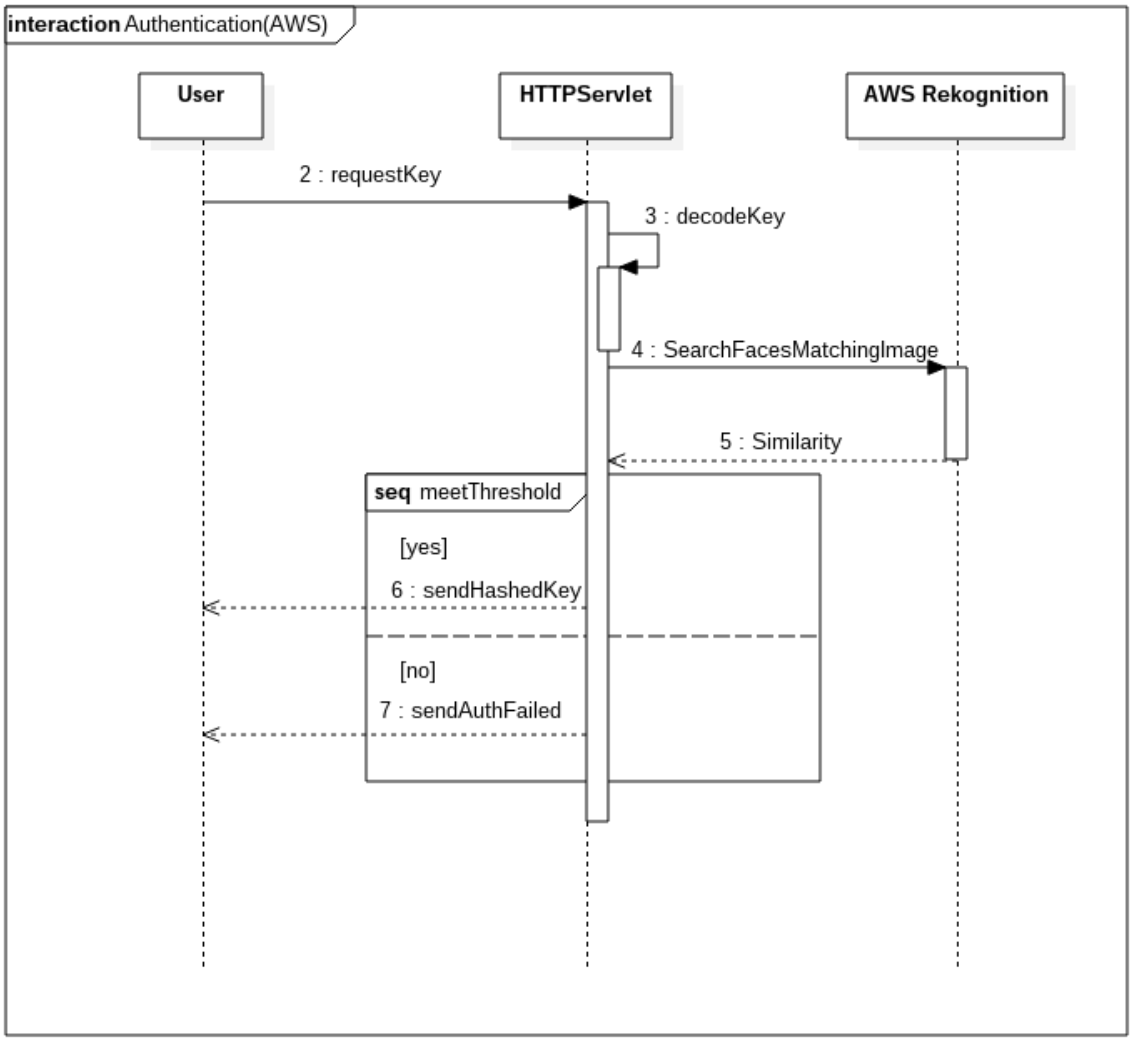
Backend Services

Sequence Diagrams

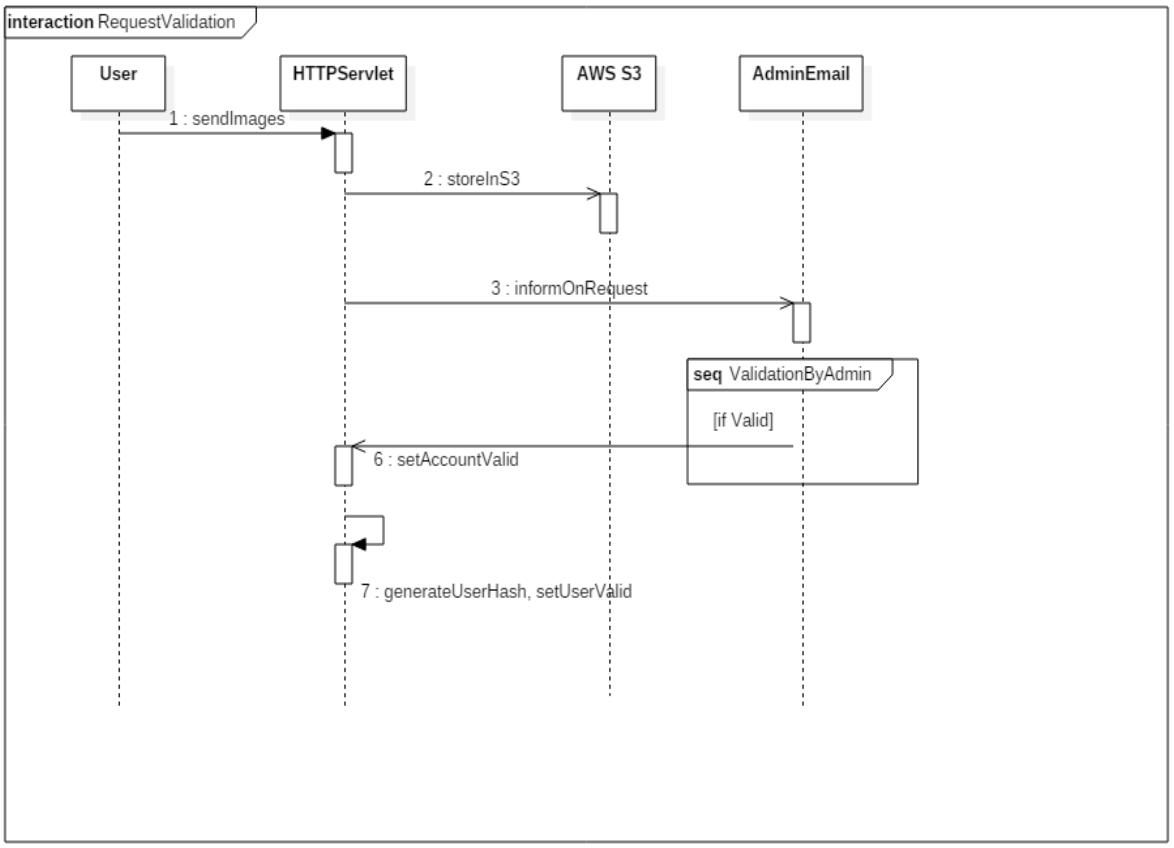


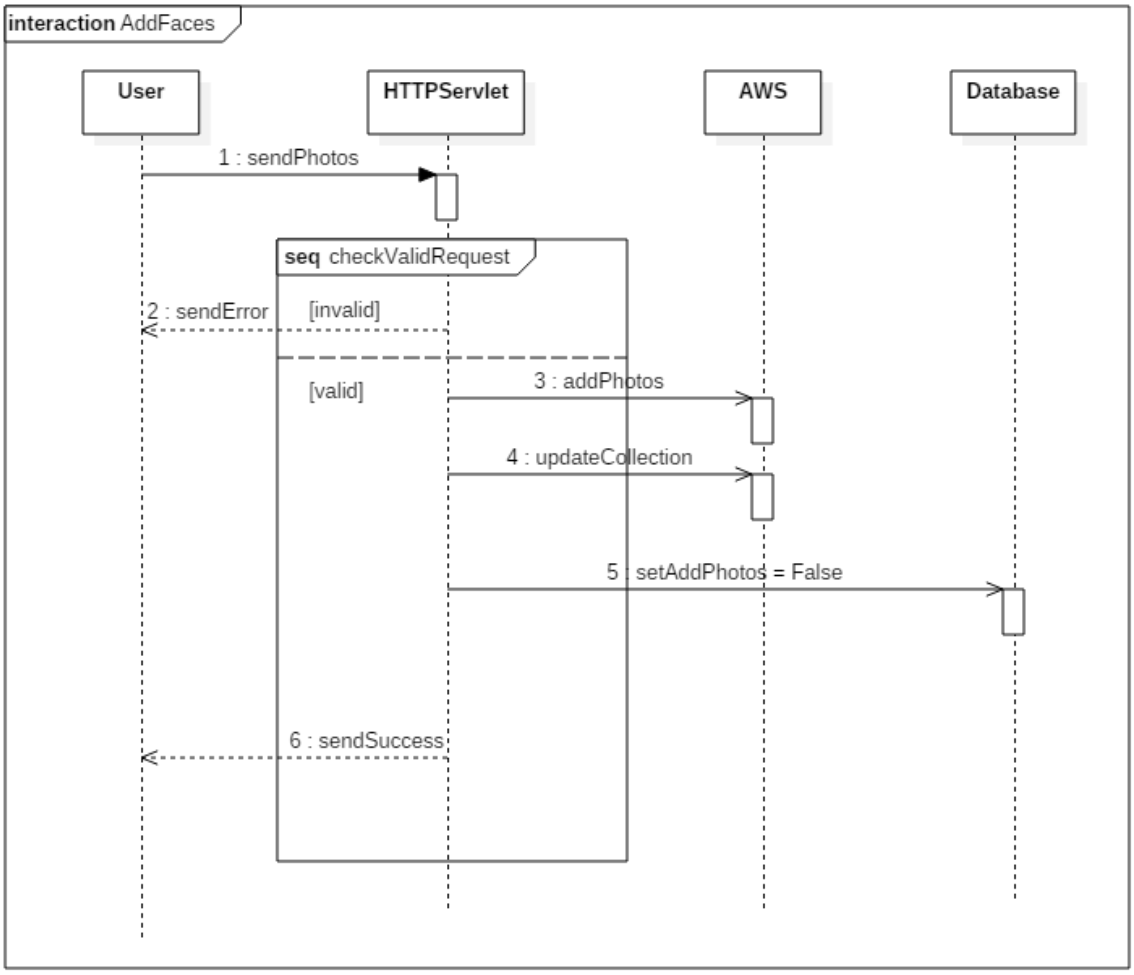

interaction Password Reset
User
FirebaseAuth
UserEmail
1 : EmailAddress
2 : FindAccount
seq Account
sts
[if exist]
3 : Reset Link
4 : Reset Password
[else]


interaction SignUp
FirebaseAuth
User
FirebaseDatabase
1 : NewUserData
2 : Message1
seq EmailUsed
3 : success, FirebaseUser
[if NotUsed]
[else]
4 : failure
5 : UserParticulars
seq UpdateSuccess
[if success]
6 : success
[else]
7 : failure



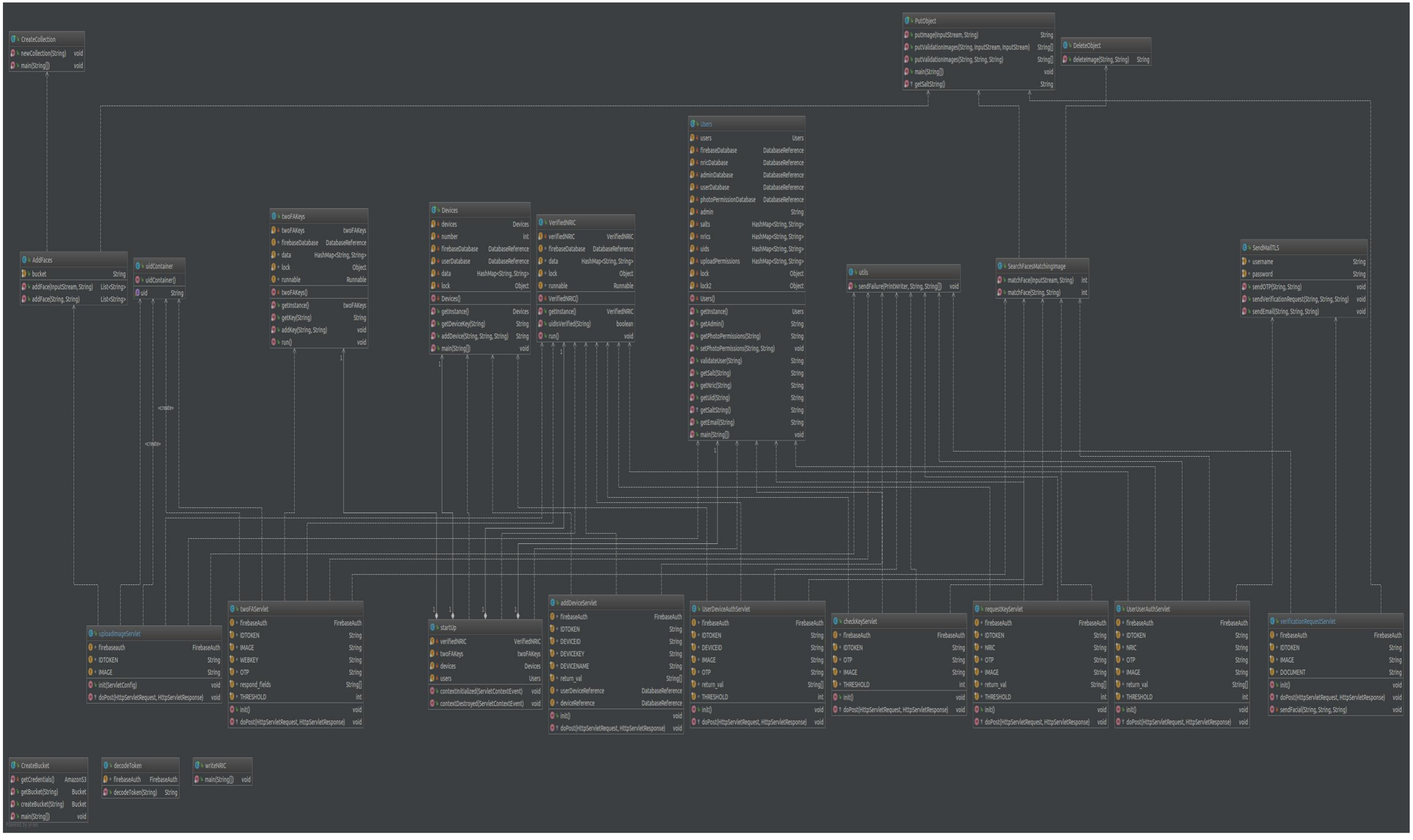
This is the general sequence for the generation of OTPs on the backend for all 3 use cases.


interaction RequestValidation
User
HTTPServlet
AWS S3
AdminEmail
1 : sendImages
2 : storeInS3
3 : informOnRequest
seq ValidationByAdmin
[if Valid]
6 : setAccountValid
7 : generateUserHash, setUserValid



Uses will only be allowed to add faces once per successful validation request. This prevents malicious users who have gained access to the app from uploading their own face to the compromised user's face collection.

Backend Class Diagram



Challenges

Security

The key security requirement for iMMe is to ensure that all requests and responses to and from the service have to be Fresh and Valid.

Freshness

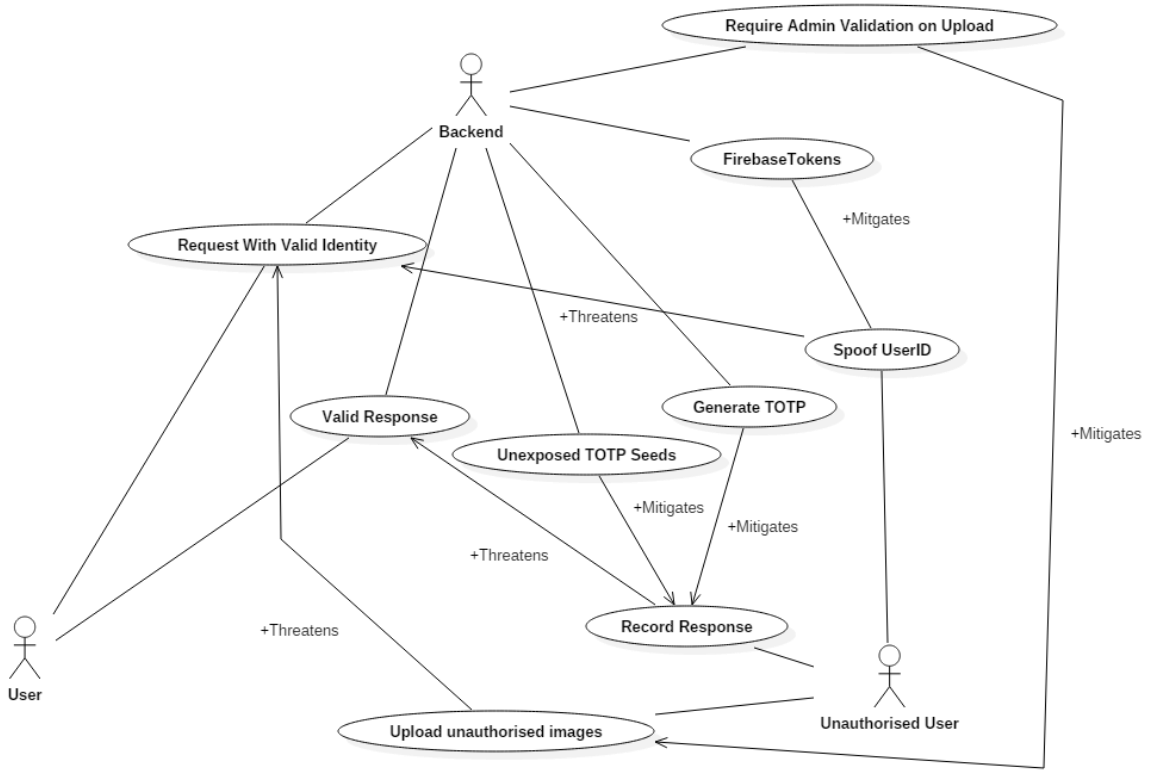
All valid authentication results in a Time Based One Time Password (TOTP) being generated and sent to the user. The TOTPs have a 30 second validity window and are generated on the backend.

All requests by the client have to contain a Firebase token which freshness can be validated on the backend. The UserID is also directly extracted from the token itself.

Valid

Recording attacks are mitigated by not exposing any keys to the clients. When a client is validated, a unique key is generated and stored on the backend. The OTPs are generated based on these keys by the server on each successful verification request. Checking of the TOTPs are also done on the server side.

The facial recognition service by Amazon Web Services(AWS) is sufficiently accurate and valid verification request have a >80% similarity rate.



Unfamiliarity

As we did not have any prior experience deploying backends or working with facial recognition, our group decided to try evaluate and develop on two alternate platforms concurrently. For example, facial recognition was both explored on using native libraries (OpenCV and DLib) and cloud based services (AWS Rekognition). This allows us to have a fallback plan should one platform be deemed unsuitable for our project. While this approach gives us redundancy, it comes at a cost of increased workload and inefficiency as one approach would have to be scrapped eventually.

Concurrency

The backend database is not hosted locally on the server. As our there would be many clients connecting to the server at any one point, we decided to cache the database on the server itself to increase performance. However, this brings in concurrency issues. To mitigate these problems, the local caches are synchronized and hence thread safe. Updates to the database are never cached and are directly written to the database to ensure immediate updates to the database. Updating the local caches by pulling data from the database is done by event listeners provided by the Firebase library to further increase performance by reducing resource requirements.

Integration Challenges

To reduce coupling between the Front and Backends, all requests and response will be done using standardized JSON objects which are natively supported on both platforms. Resilience to invalid requests are done on both the front and backend, (e.g. unfilled fields are both checked on both the front and backend). The backend will send concise failure messages embedded in the JSON response down to the Android app should there be a need to. Reducing coupling allows Front and Backend development to happen independently by adhering through a standardized request/response framework.

There were also integration challenges in utilizing open-source nonstandard libraries as many web servers do not support it out right, hence we need to utilize virtual OS/containerization to allow our application running on nonstandard libraries to be hosted on web servers. For our solution to this please check appendix 1.1

Design Patterns

Singleton

Database caches are singleton instances to ensure that there will be no duplicate connections with the database at any one point and that all threads will access the same database object.

Firebase sessions are also implemented as Singletons to prevent duplicate connections to services.

Testing

Unit Testing

Frontend

The app was distributed to some of our peers in school to collect UI/UX feedback. Some felt that the aesthetics were not pleasing, and the navigation did not have sufficient flow and was not intuitive enough. Following this feedback, we have revamped the entire UI and have improved the aesthetics as well as implemented the Material Design ideology for a better User Experience.

Unit testing was done using Junit on non-UI features like regex checking.

Backend

All Services

All individual backend services are tested using Junit. Whitebox testing was done by sending in multiple invalid requests that would trigger each conditional check in the servlets. Blackbox testing was done by sending in requests and asserting the response by the server.

Facial Recognition

Amazon Rekognition: Multiple true and false images were tested against a bank of faces uploaded onto Amazon Web Services. Results were within expectation (False: not detected, True: above 80% match with 90+% certainty)

Dlib: Library of face and matching photo ID is passed as argument into the test, and unit testing is done with assert true. The faces are then mismatched and assert false. No faces are inserted to test if case returns NULL. A Command line script is created so that the test case can be tested with one click and return timestamp and final report can be saved into a log file. Logfile is timestamped with filename corresponding to time of execution. Runtime is also recorded to record speed of entire process.

Infrastructure

Google Cloud App Engine(backend endpoint): about 100 POST requests were made to various servlets to simulate load. App Engine is able to handle the load with 100% expected results.

Security and Protection

Servlet hosted on App Engine can decode, with 100% accuracy, Javascript Web Tokens issued by firebase to ensure that requests made to the servlets are from legitimate platforms and verified accounts.

System Testing

Time-based One Time Password (TOTP) & Authentication with Lockers

User can unlock a locker by sending a TOTP via NFC. When user authentication is successful, after a request to unlock a registered locker, the user can tap his phone to transmit a TOTP through NFC to cross-check with the TOTP generated by the locker. If the locker stores the same seed as the seed stored in server, it is the correct locker to be unlocked. The two TOTPs will match. Otherwise, the locker will not be unlocked.

Service Interaction Sequence

All backend services are tested by emulating the entire user interaction sequence through Junit.

Front End

Espresso was used to simulate different cases of use cases like login, sign up and updating particulars by simulating user actions.

Robustness Testing

Backend

Account Identity Exclusivity

Multiple request were sent to services to ensure that the validated accounts would not have collisions and that there would only be one account tagged to one NRIC at any one time.

Invalid Requests

All services were tested by sending requests that were invalid (invalid IC format, missing fields, unauthorized NRICs, etc.) and ensuring that all exceptions were handled and the appropriate response was conveyed to the user through the UI.

Malicious Requests

Spoofed tokens and recorded TOTPs were fed to both the clients and servers to ensure that our security protocol would not have these loopholes.

Access to administrative services like validating users were tested by sending recorded admin tokens and by trying to access them using non-admin accounts.

Front End

Monkey testing was done for robustness testing.

Firebase Test Lab and Crashanalytics were used to test the app on various physical android devices hosted in Google's data centers and to collect crash data for bug fixes

Lessons Learnt

1. At the beginning of the project, should set aside time to do a thorough research on the platform and modules to use for the project and have far-sightedness in anticipating the next step, as some components/module may work now, however difficult to integrate with others thereafter. e.g Python nonstandard library with servers.
2. Integrate the various components early, as there may be some interfacing issues that may take a long time to fix.
3. Security should be part of the initial design rather than just an afterthought.
4. An app should provide simple functions that are easy to use as well as opportunities and freedom to use the functions creatively in various situations. For instance, the user-to-user authentication can be used in many scenarios according to the users' needs.
5. It would be good to set a standard android API version if it is a group project, as well as compliances.

Links

Android UI

https://github.com/SanPersie/iMMe

Backend

https://github.com/Chopstickbro/Imme_Backend

Backend (Initial Validation Facial Recognition)

https://github.com/zengersoong/ISTDESCPROJECT

Video

https://sutdapac-my.sharepoint.com/:v:/g/personal/tian_lerkhoong_mymail_sutd_edu_sg/EUqPHoITFBxKgIwFyQzSdC8BObKCA4Qdbjt2lRN-Ocg37A?e=dpu0NN

Appendix

1.1

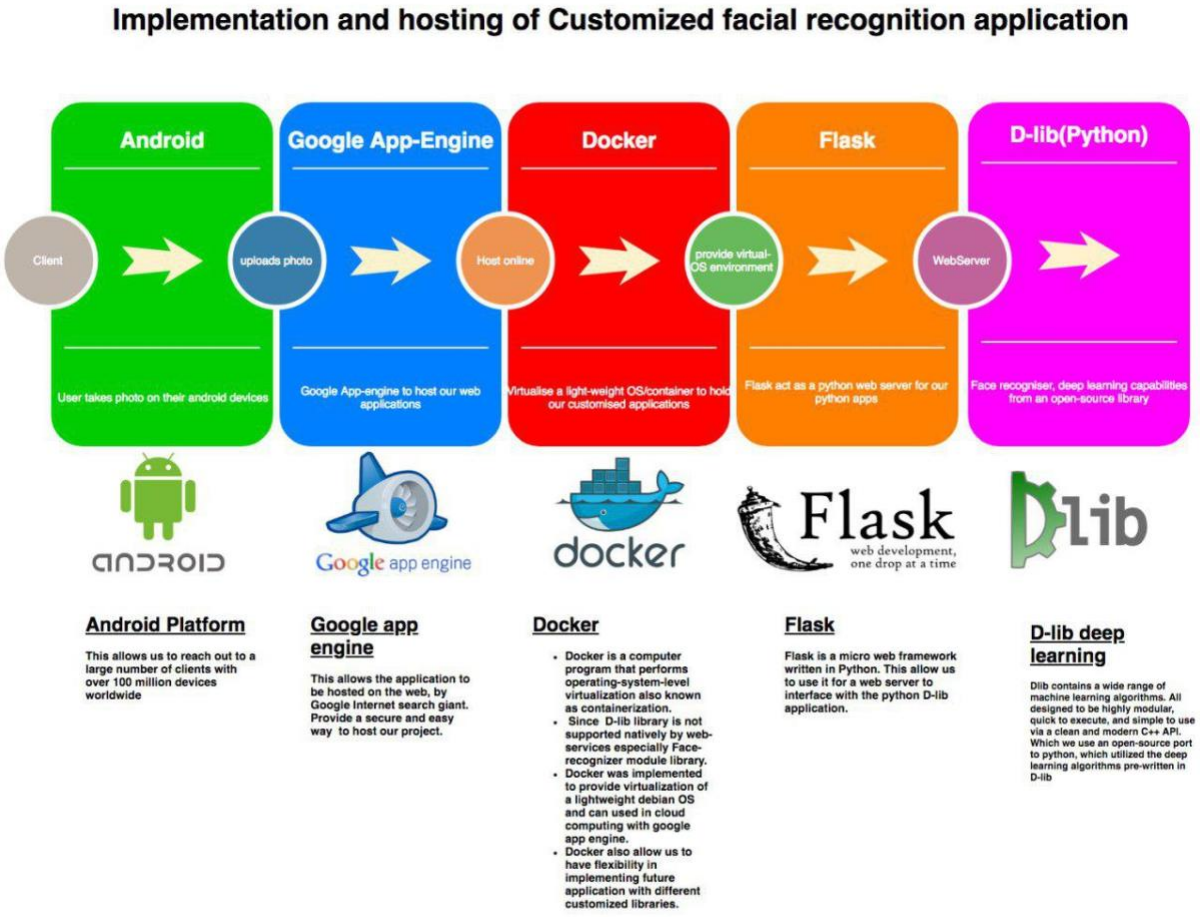


Figure 1Solution to integration problem