Wednesday, June 14, 2023

Communere

Health-Care System

(Use Case Based)

\*\* This is the most high-level architecture of the system (most of the details are ignored. Some technical details are explained in some steps just for clarification of flows) \*\*

* Sign-up :
  + User:
    - OTP verification (otp api in UMS)
    - Face Recognition (recognize api in FRS) if its needed
    - Getting user information (SignUp api in UMS)
    - Generating JWT token or Opaque as a session for the user
  + Doctor:
    - OTP verification (otp api in UMS)
    - Face Recognition (recognize api in FRS) if its needed
    - Getting information (SignUp api in DMS) (Waiting for CS-TOOL to approve the identity of the doctor since its crucial)
    - Generating JWT token or Opaque as a session for the doctor
    - Call GetSignUpRequests (DMS) from CS-TOOL (customer support tool)
    - Call Finalize api (DMS) from CS-TOOL to approve the identity of the doctor (since accepting sb as a doctor is sth crucial its better to authorize them more accurate and manually but automation is also possible in this part (calling some third party government apis to check identity of the doctor))
* Sign-in :
  + Call authorize api in AA and getting the token (All apis should be passed to LB first if then it will call AA as a middleware then we will reverse proxy the request to the next service)
* Picking Doctors :
  + User:
    - User calls GetDoctors api in DMS (response is a filter based on his/her needs)
    - User calls GetDoctor api in DMS and will get the doctor with details
    - User calls RequestDoctor api in DMS
  + Doctor:
    - Doctor calls GetRequests in DMS
    - Doctor calls UpdateRequest in DMS (which is accept or reject)
    - Doctor can start looking to user vital signs
    - Doctor and user can start a chat together
* Chat :
  + User or doctor creates a chat
  + A web socket will be created for them
  + WSMS is responsible for the rest (Will write to and read from Cassandra)
* Vital-Signs :
  + User will connect the device via Bluetooth to his/her app
  + User will see vital signs data offline in his/her app
* Vital-Signs (Real-Time) :
  + User calls TrackVitalSigns in HMS
  + App will receive some json time-based rules to send the vital-signs to the server
  + HMS calls VSR and WSMS to create a web socket
  + App will send vital signs over the web socket to VSR
  + VSR will be responsible for the vital signs and etc
* Vital signs (use cases) :
  + Vital signs are crucial so we need to be highly available on that so we need to have both VSR (ONLINE) and VSR (OFFLINE) logics
    - VSR ON (read) :
      * WSMS creates a WebSocket
      * VSR reads the vital signs
    - VSR OFF (read) :
      * VSR receives the data with email or sms (the performance here wouldn’t be a bottle neck since we are setting the rules (e.g : every 1m or with every big change in a vital sign) in UMS before start receiving the data from user)
    - VSR (write) :
      * VSR decides to send it to whom (e.g : if its a big change in heart rate it will send it to EMS over the queue)
      * The logic of all vital signs are here so nobody else in the system would be responsible for that. others will only subscribe to the queue to receive a command from VSR and do it
* Others:
  + There are so many other use cases for the app which doesn’t need so many explanation since they are simple and most of them are http-based
  + DMS (profile CRUD, user-doctors CRUD)
  + UMS (profile CRUD)
  + HMS (vital signs CRUD, health state CRUD, health care schedules CRUD)
  + EMS (emergency-providers CRUD (internal from CS-Tool))

Health Care System

(Structure based (micro-service based))

* UMS:
  + UMS is responsible for every request which is directly about the user
  + SignUp api (can be divided to more than one step based on policies)
  + User profile CRUD (profile photo, bio, personal details)
* DMS:
  + DMS is responsible for every request about the doctor
  + Get and filter doctors
  + DoctorRequest CRUD
* EMS:
  + EMS is responsible for every emergency request which user can ask for ambulance, emergency doctor and etc manually and also we can call them in the system based on vital signs
  + SendEmergencyProvider (sync) (ambulance, doctor, friend)
  + SendEmergencyProvider (async) (ambulance, doctor, friend) from VSR
  + EmergencyProviders CRUD (from cs-tool)
  + UserEmergencyProviders CRUD
* HMS:
  + HMS is responsible for every request about user health
  + UserVitals CRUD
  + UserVitalRule CRUD
  + UserSchedules CRUD
  + VitalTest CRUD (increases the rate and accuracy of receiving VitalSigns)
  + VitalTestTrack (called by doctor)
* PNMS:
  + PNMS is responsible for every notification request which is sent over the SNS
  + All of services mainly EMS, UMS, DMS can send notification over PNMS
  + Which are triggered on : (signUp completion, doctor request, every emergency event for both user and doctor and so many other use cases)
* CAS:
  + CAS is responsible for every chat request and presence checking request
  + Open/Close chat, Send/Receive messages, group messages
  + NotifyPresence for small list of friends based on some algorithms which is sent over WebSocket and will be tracked in Cassandra
  + Write and read identification are same as sending message over the ws
* WSMS:
  + WSMS is responsible for handling websocket servers and users engaged with it
  + Open/Close WebSocket which is used by both CAS and VSR and all websocket services
* KYCS:
  + KYCS is a reference for all identifications and KYC flows for the user and doctor
  + SignUp request will check completion of KYC flows of user
  + All flows which need higher permissions which doesn’t have JWT token
* AA:
  + All requests will be authorized here
  + All AA replicas will keep their sessions in a Redis KV store
  + User authorization and authentication data and history will be in AA’s postgres db

Health Care System

(Performance concerns)

Most of the performance concerns are in CAS and VSR which are realTime and all other apis and read and writes doesn’t need too much effort but they are simply scalable e.g :

Read and write doctors, users, emergency providers and all of their details and profiles can easily scale considering the flow :

1\_request comes in AA and it authorizes it with its Redis and its in memory and AA is also replicable so nothing is faster than this till now

2\_request comes to load\_balancer to balance the load and check the api permissions to see that it can forward (reverse proxy) the request

3\_ request comes to DMS, UMS, EMS, HMS which are replicable so they also can easily scale

4\_ the request comes into the database layer or it will send internal requests for internal requests we use GRPC which is 7-10 times faster and for db it is totally fine with these type of apis but still we can scale it with optimizing the database, using repmgr postgres, replicating postgres with master slave setup for faster reads and even replicating the same master slave cluster for more availability which makes it possible to write to db in master failure cases even before counseling flow and also we can use indexes and normalize the database schema for optimizing it.

Domain models and bounded contexts are shown clear in the schema design

But from SOA point of view I divided the micro-services based on their domains

And the main of idea of DDD is to keep the micro-services and models isolated from technical and conceptual point of view which are totally isolated in the system that I described UMS for User Domain, DMS for doctor, EMS for any emergency needed in the system, HMS for every vitalSign rule and health profile and ills, CAS for chats which is reused for both cs-tool and user chats which is only depends on WSMS which is normal because WSMS is one of the main parts of a chat and only decoupled to be able to reuse it for VSR then everything is isolated, replicable and reusable

TDD :

First we need to design all of the details from up to down and whenever we are clear with the design and we want to start the implementation we can use TDD which means we need to implement from down to top e.g :

We start to write repository or any DAL design pattern for accessing to db

For every class and its method we write some unit tests:

e.g

Type UserRepository interface {

Read(id int)

}

Read\_Test() {

db, mock, err := sqlmock.New()

if err != nil {

t.Fatalf("an error '%s' happened", err)

}

defer db.Close()

mock.ExpectQuery("SELECT \* FROM users").WithArgs(1)

myDB := NewRepository(db)

if \_, err := myDB.Find(1); err != nil {

t.Errorf("something went wrong: %s", err.Error())

}

}

now we are sure about first layer

Then we go to the 2nd layer e.g : service layer

We write it and test it now 2 layers are working correct

We continue the process to provide the apis for the client and other micro-services after writing the last tests which are integration tests and we call the apis directly.

Last words:

There are so many other design-level details which we can describe so I think this would be enough since we can come up with a book with continuing this

So please consider it as really summarized and high-level idea  
  
Thanks!