CHAT SERVER

This document describes the needs and requirements of a chat server, some are must-haves, while others are described as recommendatins and preferences but not necessarily limited by the technologies and frameworks suggested by this document.

Before requesting this development, we assessed the chance of using one of the following options as a base for a chat server solution: a) open-source projects based on the XMPP protocol, b) SaaS rented chat services, c) traditional backend chats using php polling in order to load information. None of these options satisfies the needs enlisted below, in particular the fact that none of them offers the required control to restrict a recently registered users (upon starting a chat) to be able to see and communicate with only a certain group of selected other users.

Even though this development's scope is a chat server, it is cricial to understand what kinds of applications will be using it.

Some of the chat's requirements are: security, utf8-mb4 (emojis) support, support for text formatting, sending of attachments, online images visualization, status notifications indicating if a message has been sent, received and/or read (inspired in Balckberry's / Whatsapp's / … checkmarks), and most importantly, CONTROL over who is able to see and chat with whom.

In time we will develop web, iOS and Android clients that will connect / use the chat server to be developed as per this document.

TECHNOLOGIES

Programming language

NodeJS, Python o Go are our preferred programming languages because we are familiar with them, they have good performance and low / predictable memory usage in high demand applications + they lend themselves for OOP and modularization.

Architecture

The chat server solution designed to send and receive messages, needs to have a PUB - SUB architecture.

Communication

A Server-Client architecture is required which means that no communications will be directly transmitted between chat participants.

Websockets is considered to be the ideal technology to transmit information, text, files and even audio /video. It is a versatile technology, well documented and supported by all modern browsers, including native iOS and Android applications.

If implemented, websockets need to be used over SSL (wss://).

The chat server must allow users to subscribe to push-notifications, as well as send the author, channel name and a message preview through a push notification when an active websocket with the current user is not available.

Security

Secuity is paramount. Hardcoding and database password exposure in code files must be avoided. Sensible data id's transmision in each message must be minimized and if possible, encrypted / hashed based on what is best on a case by case basis.

Authentification will not be part of the chat server. Another service / server will be in charge of the authentification process, which will be referred to as "auth-server" from now on. The chat server will only check that every operation contains an authentification token provided by "auth-server".

Implementation of the Signal protocol https://signal.org/blog/advanced-ratcheting/ is not an option due to the required Server – Client architecture and storage scheme.

Web API

The chat server needs to consume at least one web API and it must implement and provide at least 4 simple web API's for other services / servers to consume.

API Consumption:

1. The chat server must validate authentication tokens by consuming a web API provided by "auth-server". The chat server will send the token to be validated along with a string that identifies the chat server as the one consuming the service. The consumed API will respond an integer value (either 0 or 1) depending on the validation's success.

API Implementation / Providing:

1. Operations on a chat: a) create a chat, b) inactivate a chat, c) add and remove participant(s). This API will receive the chat data, the operation to be performed, the required parameters (i.e. participants to be removed) as well as an authorization token, and it will respond with an integer value (either 0 or 1) depending on the operation's success.

2. Reception of attachment files: this API will receive the chat's id (that the attachment file was sent through), the file's name (including it's extension), the file's content (in binary code) and an authorization token.
3. Full chat download: this API will receive a chat id and an authentication token. It will respond with the full chat trancription in jSon format.
4. Search: this API will receive the search string, an authehtication token and the chat_id as an optional argument. It will respond with the messages that match the the search string in jSon format, including the chat_id, date and time of each message, ...

Data Storage

The chat server scalability is very important, therefore message storage must be a separate and replaceable module. In a production environment, the chat server and the database would be idealy hosted in different servers. MongoDB is suggested because of its read and write high demand performance, as well as it's good reputation handling up to 100 million documents in a single collection. As mentioned before, utf8-mb4 (emojis) support is required.

Below is a first information storing scheme that we think could serve as a discussion starting point. Ideas and recommendations are welcome for discusion.

Due to it's good cost-benefit balance, we think it might be a good idea to store attached files (sent through the chat server) in an AWS S3 server. We are considering if the chat server should be resoponsible for sending the files for storage, or if it should be done by a different application (perhaps in a 2nd step(?)). Should the chat server be in charge of this operation it would need to calculate a unique string (i.e. a SHA 256 string) to identify each file and it will then save the file using that KEY. On the first stage functionality, the chat server will save the URI of the attached file stored in S3 as a regular message.

```
channel
--------
channel_id
channel_name
channel_date_time
user_id
channel_status

subscription
-------------
subscription_id
channel_id
user_id
subscription_user_is_admin
subscription_date_time
subscription_status

message
--------
message_id
channel_id
user_id
message_delivery *
message_type
message_metadata **
message_date_time
message_content
message_status

download
---------
download_id
download_sha256
download_type
download_size
download_path
download_date_time
download_status
```

where:

*message_delivery is an array that stores the user_id as the array index and the delivery status as its value: 0 = not sent; 1 = sent; 2: received; 3: read

** message_metadata is an array that stores specific attachments' metadata,. i.e. for an attached image it could store the file's name, the extension, resolution, a thumbnail path, etc.

Before starting to code, the data storage scheme needs to be discussed and agreed upon.

Code Formatting

The source code must be commented at the beginning of each file and at the beginning of each class and function (depending on the coding language used). All code must follow a logical order and it must also be separated in as many files as it makes sense, preferrable one file per class, function or module according to the coding language's best practices. The names of functions, variables, etc. must be descriptive.

Resources Usage / Optimization

The chat server's memory usage has to be optimized in order for an AWS EC2 instance (C4.xlarge, 4 vCPUs, 7.5 GB RAM family) to be able to support at least 200,000 simultaneous connected users.

Massive data copying technicques are to be avoided, i.e. having the same informatoin on different variables at the same time, information passed between classes / functions / modules / etc. must be clear and efficient, etc.

Channel creation (chat creation)

The chat server has to prevent message sending between users whenever "auth-server" didn't either allow or initiate the process. Once a channel has been started including "N" participants, any participant can send and receive all future messages in that channel, however none of the participants can either send messages or initiate a chat only with a subset of those participants unless "auth-server" has previously started a channel that only includes those participants. In other words, participants can only communicate through existing channels. Example:

Given chat participants A, B, C, D, E and the following event cronology:

a) "auth-server" starts a new channel including A and B. As of that moment, both A and B can send messages to each other at any given moment through that particular channel.

b) “auth-server” starts a new channel including C and D. Same as above, both C and D can send messagges to each other at any given time through that channel

c) “auth-server” starts a new channel including A, B, C, D and E. Again, all participants can send and receive messages through that channel at any given time. However none of the participants is allowed to start a new chat (channel) with just one or more of the participants just because they already have a common channel, (i.e. A cannot start a channel with C; B cannot start a channel with A and C, etc.) unless “auth-server” has previously created that particular channel.

Message statuses

The chat server must keep record of the different statuses that a message goes through: a) sent by the sender; b) received by the recipient(s); c) read by the recipient(s).

It must also be able to communicate those message statuses to all participants of each chat.

User statuses

The chat server must have a method to determine the a user’s presence status, i.e. if the user is a) Connected; b) Absent or c) Disconnected.

The chat server will deem a user as connected if a) there is an active websocket with the specific user and b) the user has had activity in the last 5 minutes (i.e. sending a message, reading a message, etc.).

The chat server will consider a user as being “Absent” if there is an active websocket with that specific user and b) the user has NOT had any activity in the last 5 minutes.

The chat server will consider a user as being “Disconnected” if there is no active websocket with that specific user.

Participant-writing notifications

The chat server will send a meta-message to all participants connected to a certain channel whenever a user of the channel is writing a message on that channel through the chat application and the chat server receives a meta-message from the chat application indicating that this is happening.

“Meta-message” means a “sign given” that won’t be saved as a message on that conversation.

Token Validation

The chat server must be able to validate security tokens with “auth-server” consuming a web API.

Offline users

Even if a recipient user is offline, message reception must be allowed. In this scenario the message(s) should be delivered whenever the recipient goes online again. As mentioned before, if a recipient user is subscribed to push-notifications, those notifications must be sent immediately (even if the recipient is offline).

Chat transcripts / download

The chat server must provide a web API in order to be able to download the history / conversation of a whole chat (channel). It would get the encripted channel_id along with a security token that it would use to validate the petition. It would then return the messages in chronological order (starting with the newest messages and ending with the oldest ones). The response format has to be a jSon where the index is the date and time of the message (including the milliseconds) concatenated to a dash and the user’s name. The jSon values should be the content of the messages.

Webhooks

The chat server must provide the following webhook:

1. Upon a participant disconnection, the chat server will send a petition with the participant id, an array that includes the active channel_id's of that participant, as well as a token indicating that it was the chat server who made that petition.

Third Party Code / Software

Copying code is allowed (as opposed to trying to reinvent the wheel) as long as: a) it respects copyrights, keeping in mind that this project is going to be used for commercial purposes; b) we don't end with a "spaggethy type of project", i.e., leaving in a single file just 20 lines of a library, then 50 lines of another library and calling that file: "libraries".

Keeping order is paramount. Code should be separated in the way that makes most sense (e.g. in order to avoid having files with more than one programming style) and keeping comments indicating the origin of each piece of code.

Open Source

Use of open source is not only allowed but recommended. Nevertheless, documenting the version and origin of each used project, and respecting the licence files and copyrights is a must.

Pay libraries

Use of code or libraries that have an associated cost (i.e. a one-time only cost like a license, a consumption-based cost/SaaS, etc.) must be approved by us. Before starting any implementation regarding pay code/libraries, submitting to us the required software information, URL, pricing, license, as well as a small explanation justifying the recommendation to use that code is a required first step to evaluate if it makes sense to go ahead with the implementation.