TUMBLR MESSENGER - ARCHITECTURAL OVERVIEW by JONATHAN MARBALLI

Time Spent: I spent about 45 mins on the design and writing it out in this document.

INTRODUCTION

The messenger system essentially breaks down into three components:

1. The UI where users can send and receive messages
2. The REST API that takes user inputs and writes/reads from the DB
3. The messenger's database

THE UI

It seems to me that there are two ways you can go in terms of a user interface. The first is an e-mail system where a user has an old-fashioned inbox where messages come in and are prioritized in order of arrival. The second would be more of a Slack style system, where a user has a dashboard that essentially serves as an "Inbox" whose priority parameters is set by the user (i.e.: being able to pick which "channels" / group-chats are more important than others).

Due to the time constraints of this "test" and the fact that the Slack approach seems simpler to me at the moment (I could be wrong), I'm going to opt for a Slack-esque implementation.

When considering the requirements in the test in the context of a Slack-esque system, I think we can make the following assumptions:

- The requirements that each user having a single inbox that can only be accessed by that user will be naturally enforced by requiring the user to login to see their inbox and then by only retrieving messages that they're a sender or receiver of.
- The requirements that a user is able to DM other users and/or groups of users, and cannot carry on more than one DM conversation with the same users essentially allows us to consider DM conversations and GROUP conversations to be the same thing. A DM conversation is just a GROUP conversation that consists of two people. This should allow the DB schema to be simpler.

THE API

The following is a list of potential API endpoints that the UI will call, and a description of what they'd do:

Endpoint	Description	Parameters
/get-messages	Retrieves array of messages (with messageIDs) for a	● userId ● startTime (how far back we'll go) ● endTime (assume if this is null we

	given user within a given timeframe.	mean “now”)
/send-message	Send a message	● Sender (userId) ● Recipients ([user-ids]) ● Message (string)
/delete-message	Deletes a message	● messageId
/delete-conversation	Deletes a conversation *Right now I’m assuming if anyone deletes a conversation it’s removed from all participants’ inboxes	● Sender (userId) ● Recipients ([user-ids])
/get-conversations	Retrieves array of ConversationIds that user is a participant in	● UserId

THE DATABASE

The data required of the API can be generally represented with the following tables:

* => denotes primary key

Table	Attributes
Users	● Username ● UserId*
Conversations	● ConversationId* ● Name (will possibly come into play later if we want to let users name their convos)
UserConversations	● UserConversationId* ● ConversationId ● UserId
Messages	● MessageId* ● ConversationId ● SenderId (UserId) ● Message (string) ● timestamp

Messages-That-Have-Been-Read	• id* • ConversationId • UserId • MessageId

GETTING AN MVP TO MARKET ASAP

I would start by implementing the architecture above and rolling it out to a small number of users to start, perhaps through an A/B test where a handful of users can opt-in. I would also want to make sure that automated testing was implemented from the start so that we can at least have confidence in the fact that it works as expected, eliminating the possibility of obvious bugs to start so that we can focus more on potential scalability issues.

PLATFORMS

I assume this is a web-app. I am somewhat torn between the idea of using Apache/PHP vs Node. The benefits of node in this case is that it's such a client-side-heavy application with a lot of server-side operations, it may reduce code duplication if client and server side lives in the same shared codebase and language. I also think we'll want a relational database as the data will be changing a lot, with every user action essentially.

I think I would also want some sort of local memcache pool in case the service goes down, so that we can at least still provide users with their latest message history.

SCALABILITY ISSUES

A few potential scalability issues that I can think of offhand include:

1. **Q:** What happens when two users send messages to each other at more or less the same time? Do we re-adjust the order in the user's window to match the timestamps, or do we have to handle this more elegantly? Things could get chaotic. / **A:** I think this user experience would be something to discuss with the product team. We want the messages to reflect accurately who said what when, but we also don't want to keep re-shuffling the order of messages that were just sent if users are extra chatty that day.

2. **Q:** If the DB goes down, how do we keep a log of user's messages that are sent (and other actions) in the interim so that they're not just lost? / **A:** I imagine we'd want to set up some sort of queuing system on the client side that messages live in before they're officially written to the DB (which is confirmed by a 200 response from a /send-message call)

3. **Q:** Should we be expecting spikes of users logging in at the same time, like when the workday starts? If so, this could cause the DB to get extremely hammered as users retrieve their message history to render the client. / **A:** This is where the local memcache could come in handy. A local data store means less retrieval of message history onLoad. Another option might be noticing groups of user's data habits, perhaps by keeping track of their login times, and priming a DB cache ahead of time so that the first big /get-messages call doesn't hose the system.