Gregory Rogers

Sample code for simple message system.

This work is for a pre-interview code challenge and is a work-in-progress.

September 2013

Requirements:

COMPANYX could use an internal messaging system for site-wide announcements, for instructors to send messages to their class, and for one-to-one communication between users.  Design and implement as many of the following features as you feel comfortable (and have time for), include any notes that document the design issues you discovered and how you chose to address them, and don’t forget some tests.  If you don’t have time to implement all the features but believe your design would support them, feel free to just explain how you would go about implementing them given more time.

1. Allow sending of messages between individual users, identified by the unique key of their record in the system.  Messages are in markdown format, and auto-complete of recipients would be a bonus.
2. Allow a user to view their inbox, read messages, (automatically) mark messages as read, and delete messages.
3. Allow sending a broadcast message to all users.  Keep in mind that there could be millions of users.
4. Allow sending a message to a group of users.  Groups can be large (over 100,000 users) and are stored by having each user record list all the groups it’s a member of.  Group membership varies over time and a message should be received only by the users who were members of the destination group at the time the message was sent.

Overall Design

* Web service providing JSON REST end points that cover requirements.
  + All end points follow REST pattern.
  + All end points only return JSON content.
* HTTP client tests that exercise and validate the web service.
* Web app or mobile client using REST end points – not implemented.
* Backend database via MongoDB. Mongo performs best when data is de-normalized and organized for most frequent queries.
  + Example: User contains list of Group references where a reference is {id, name}. Name is de-normalized and references require updating when Group name changes.
  + When a large number of de-normalized references are expected to exist, best practice is to have background jobs that update the references.
  + All de-normalized data should have cron jobs that regularly check for and correct inconsistences.
* Users contain references to the groups they are members of.
* A message contains references to the sender, optional receiver and optional group as well as content and status. At least one of receiver and group must be provided.
* Messages that are not in draft status cannot be changed other than the status.
  + Caveat: this status state machine should be managed and is not.
* Messages are “sent” by cloning the original message for each receiver. The cloned message status is “unread” and saved with the receiver referencing the specific user.
* Message sending is initiated by POSTing a message with status “send” or PUTting the message to update the status to “send”.
  + In a production scenario, a background process would pick up messages in this status and perform the send by cloning the message.
  + In this demo code there is a simple implementation of sending that is called in-line with the POST or PUT resource request.
* A client app is responsible for setting the status of each message to “read” or “deleted”, etc. as appropriate.
* Message scale and performance is achieved via sharding the MongoDB message collection on receiver (i.e. the objectId of the user who owns the message).
  + Message data model to support this.
  + Sharding not implemented
  + All messages are independent, thus any number of “send” servers can be used to clone a source message to each of the intended receivers.
* Message format
  + Server is agnostic.
* Receiver and Group auto-complete
  + Server provides REST end point to return list of user or groups with the prefix.
    - Not implemented yet
  + Client would use this information to fill in receiver or group reference in the message.

Current implementation caveats

1. database connection properties hardcoded – should come from properties file or environment
2. there are Morphia / MongoDB scenarios that can cause the singleton datastore to be unusable. The BaseDAO methods need wrapped to detect and create new singletons for Mongo, Morphia and the datastore. This has been the case with previous versions of M&M; I’m not sure about the new version just yet.
3. Resources are throwing exceptions. These should all be caught and dealt with accordingly. For example by at least returning an appropriate HTTP response code rather than 500!
   1. Since the REST interface is trying to “always” speak JSON, the appropriate thing to do is have an appropriate status code and have a JSON result that provides details.
4. Users don’t have passwords! Use bcrypt or equivalent to safely store.
5. Server should only accept https connections for logged in users.
6. Input JSON from client is sometimes trusted.
   1. When client sends a list of group ids for a user’s memberships, those ids are not validated.
   2. Code has //TODO: to spot some of these.
7. Demo includes a class to send messages to receivers.
   1. In production system this would be done by background process that can queue up any number of message request and farm them out to lots of servers for processing.
   2. Demo class does not handle errors so message could get stuck in the “sending” status and forever be ignored.
8. Auto-completion not implemented yet
   1. REST end point for expanding prefix to a list of users
   2. Client would be expected to call this end point.
   3. Server side does not expand receiver names on the message.