Gregory Rogers

Sample code for simple message system.

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

September 6, 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 and Implementation Notes

* Web service providing JSON REST end points that cover requirements.
  + All end points follow REST pattern.
  + All end points only return JSON content (see caveats).
* Web app or mobile client using REST end points
  + Not really my skill set so a true GUI client is not yet implemented.
  + Implementation includes http client tests demonstrate and validate behavior
  + Implementation includes python command line scripts for client functionality.
* Backend database is MongoDB. Mongo performs best when data is organized for most frequent queries. This usually means de-normalization to some extent.
  + 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 duplicate names in the references. That is, the REST end point should not wait for those updates to complete.
  + All de-normalized data should have cron jobs that regularly check for and correct inconsistences.
    - These jobs are not implemented at this time.
    - But then neither is the ability to change a group name!
* Users contain references to the groups they are members of.
  + The exception is the “all” group that all users are implicitly a member of. You can send a message to the “all” group and it will be delivered to every user regardless of their membership in the “all” group.
* 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. Thus you can currently change the status to any valid status.
* 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. This duplication is to take advantage of MongoDB’s ability to scale by sharding the message collection on 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.
  + The python command line script updatemessagestatus.py can do this.
* 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).
  + Sharding is not yet implemented, but the data is designed for sharding.
  + 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
  + Requirements call for mark-down format. Server is agnostic. That’s really a GUI client concern.
* Receiver and Group auto-complete
  + Server provides REST end point to return list of user or groups with the prefix.
    - Not implemented yet
    - This would use a MongoDB query like:
      * db.users.find({loginName:/xyz.\*/},{loginName:1}).sort({loginName:1}).limit(20)
    - where “xyz” is the prefix sent to the REST endpoint and 20 is the maximum number of matches to return.
  + Client would use this information to fill in receiver or group reference in the message.
* Inbox
  + Clients present the inbox by using GET on message/search with query parameters to narrow the returned list by status, date, etc. with paging of results.
  + This allows the client to show all messages of any status, or just those that are “unread” for example.

Quick Implementation Status:

* AWS server up and running
  + No proper build, just eclipse->project->export->to WAR
  + Scp to aws ec2 instance
  + Stop tomcat; replace war; start tomcat
* Executable tests are minimal “crawl” tests (in the crawl, walk, run sense)
  + i.e. not all error scenarios are checked.
  + Everything that should be validated is not.
* unittest from laptop pass against local tomcat server and aws server.
* user creation complete w/tests
* group creation complete w/tests
* user group membership complete w/tests
* message creation and status changing complete w/minimal testing
* message sending functioning
  + demo style – synchronous with the client REST request, not the full backend described above
  + initiated by a test, but not validated
* inbox behavior - message GET with query parameters mostly complete.
  + Todo: have paging based on creation date so that results are stable with asynchronous message creation.
* User name and group name auto completion
  + Not started, see above for design detail
* Background jobs to keep de-normalized data corrected.
  + Not implemented
* Scalable message sending process
  + Not implemented
* Scalable message reading
  + Data model designed to support via MongoDB sharding.
  + Sharding not actually implemented.
* Users are not authenticated.
  + Need to add a Filter and register in the web.xml file.
  + Password should be bcrypted before stored
  + Basic auth header used with client requests – secure with https.
* Database connection properties are hardcoded and should be pulled from properties file or environment.
* 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!
  + 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.
  + An error catching Filter can be put in the request pipeline to ensure that all resource response content is JSON.
* Proper logging not implemented yet.
  + Basic log4j in place, but not use thoroughly enough to help diagnose production problems.
* Input JSON from client is sometimes trusted.
  + E.g when client sends a list of group ids for a user’s memberships, those ids are not validated.

Accessing the example server

* use python scripts
* requires <https://code.google.com/p/httplib2/downloads/detail?name=httplib2-0.8.tar.gz>
* Follow instructions in MessageDemo.docx available in git with the source or from: