Summary of Project Goals

Our initial requirements document, stored under the requirements.md file under Documents/product, is the initial translation of the project backlog given by the professor. Because we were unable to accomplish all tasks given in the document, we prioritized the most important features of the project during our Sprint goals, which, in total, became the realistic manifestation of our project goals.

We first focused on the persistence end of the requirements document, using a MongoDB database to store essential information such as users, groups, invitations, and messages. In addition, in compliance with government regulations, messages would be stored forever even after a user sets it for deletion, only setting the message visibility for users to be false. As far as functionality, our highest priority goals have always been ensuring that messages between users and groups are transmitted correctly, which we were able to successfully implement by the third sprint. This was built on top of group creation and bidirectional invitation functions that were necessary for sending group messages in the first place. Although we did not have a login that tied to Husky and LinkedIn, we created our own register and login functions. Afterwards, we implemented a queue command for getting messages after logout and a command for getting a user's message history. With these tasks finished, we created a private message functionality, where the message would be encrypted server-side to ensure a secure set of messages.

Result Overview

Project goals consist of four sprint goals. The whole project goal which could be derived from product requirements is too large to be completed in only four sprints. Still, at the end of the semester, all important features are achieved and every stretch we promise is finished. The accomplished results suggest that the whole project is successful. Though the sprint goal will not be repeated, each component will be discussed and then statistics used to evaluate.

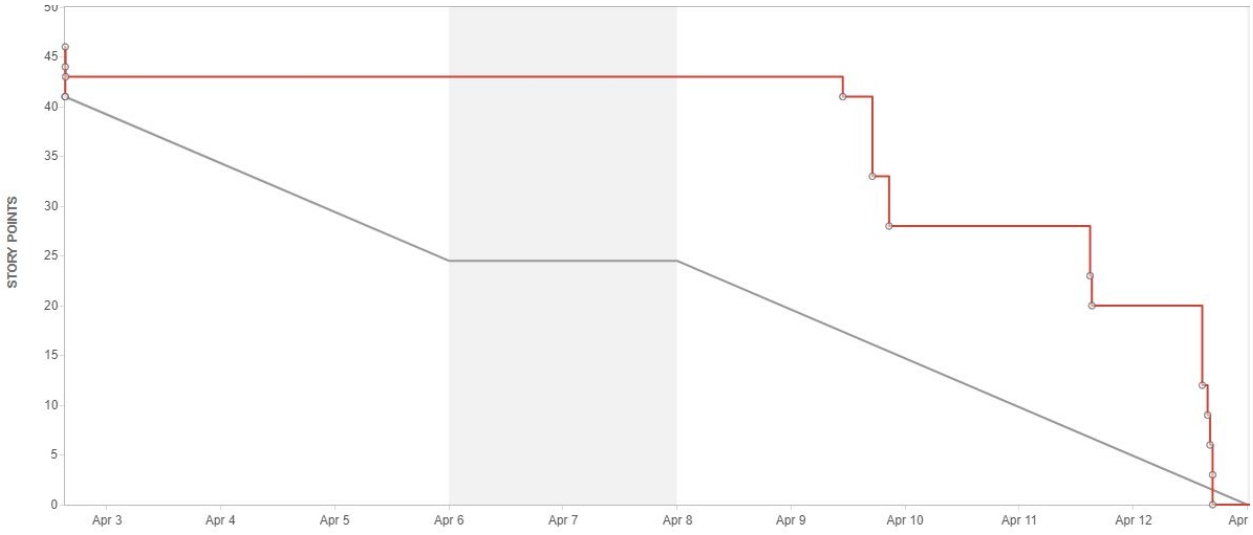
The first part is the database, in which we chose to use MongoDB for persistence. Database work started from the second sprint and the last refactoring was done at the last sprint. In the second sprint, the database was set up with a different configuration of test, local or product. The DAO layer was implemented with POJO for all basic objects like User and Group. In the third sprint, DAO and POJO supported Message. Invitation was supported in the third sprint.

The second part was the server-side implementation. Legacy code was able to work in the first sprint and the server was uploaded to the AWS instance in the second sprint. By the end of the third sprint, broadcast, user message and group message were implemented. Finally, the system supported invitation, message queue, and message encryption as part of its attractive features in last sprint.

The last part was client communication with server. This part of work started in third sprint. The client's framework was refactored in third sprint, using JSON as the main way to exchange information. However, removing all legacy code of Message was finally finished in the last sprint. Basic operations for sending messages was finished in third sprint. In the last sprint,

the client was checked completely to be able to collaborate with server side with all associated functionalities.

In each sprint, we finished enough story points and the following chart showing story point completion over time looks consistent (orange line), showing that all of us did sizeable work over each day of the sprint.



We passed all quality gate check and removed all code smells. With the help of Mockito, testing focused on its and only its functionality and achieved high coverage. It also followed the high-quality requirements of security like password being stored as a hash with a secure algorithm, and RSA encryption of private messages used a 3072 bits key.

Team Development Process

Looking at what we accomplished each sprint shows a lot about how we grew as a team. Some things that remained true across all sprints - as a team, we had great communication. We each knew what everyone else was working on, we took the time to have phone calls daily for stand up calls, and we clarified and offered advice about how things currently work or they should work. We were consistent about asking questions when we were stuck, working on certain parts of the code where others may have more expertise.

How did we grow as a team? Let's consider what did not work so well for us early on. We had just received the product backlog. We had a multitude of questions early on about the product, being confused by sometimes-conflicting requirements. As a result, we didn't take initiative early on in trying to clarify what we needed to do and set goals for ourselves. When we first got around to working on persistence (the DAO layer and Services layer), we found that by splitting the project into layers when no layer was complete, it was hard to develop without being tightly coupled to each other's' work. This was something that didn't work for sure, but it had great benefits for us later on. Though it was tough to code like this, we argue that it was necessary, since it wasn't sensible for multiple people to have to deal with the same task - for example, figuring out how to connect to Mongo, and coming up with different ways of doing so (it was truly a *separation of concerns*, one might say). Our good design for both the code and the database schema future-proofed the development of our product. In later sprints, we picked and scheduled our tasks so no one was blocking anyone else - at this point, there was very little architectural change needed - we each were able to make features and do work across all layers - the POJOs, the DAOs, the Services, ClientRunnable & Prattle, and clientside - to get the feature

working. As most features were independent, no one was blocked, and as the layers had already been designed, no one had to build new layers. Because the layers had been established (and because we continued to refactor our own code when there was a better way to do things), we performed extremely well in the last sprints, when it came to delivering and demoing features. As a team, we had a rough start, with seemingly little to show for it, but the initial work we did set the groundwork for future, enabling us to become more efficient with each sprint.

Project Feedback

The project, as a whole, was a refreshing experience. Jiangyi, Raghav, and Emma had worked with Scrum before but this project was more heavily dependant on legacy code than we were used to, which was new. The best parts of the project were, surprisingly, not its coding aspects. All four of our team members already knew how to code fairly well and did not run into any difficulties where we were completely lost on what to do; however, this was under the assumption that our tasks were clearly cut under JIRA tickets. In other words, because we were able to decide clearly what our goals were beforehand, we were able to pick up the tickets that we knew best about. In a similar case, SonarQube gave us good advice throughout the four sprints about what is and is not considered good practice in Java, allowing us to glean into what we eventually came to see as a cleaner, more elegant code.

One particular grievance we had about the legacy code was that it came untested; even in a situation where the code is being handed off from one team to another, it does not make much sense to have almost completely untested legacy code that the receiver team must test. Another aspect that was rather unusual was that we were supposed to be solely responsible for the server-side of the application but the client-side needed heavy refactoring in order to support even basic functions such as sending direct messages from user to user. The client-side is written in such a way that it sends out a pure String that stores its information as substrings, which does not make sense in modern server-client architecture. We had to refactor this in order to send JSON strings as a starting condition for our work to begin. Each function to be implemented on the server-side had to also be implemented on the client-side so we are not entirely certain why

we were provided only with the server code to begin with or told that we were to work only the server-side.