Team 102

CS5500

Final Project - Report

Summary of Project Goals

Over the course of the semester we were tasked with enhancing the existing messaging system, Prattle. The given code contained a bare-bones system that was able to connect clients, via Chatter, to the server, Prattle, and broadcast messages from one client to all other connected clients. Prior to taking on the assignment of enhancing the system, the team set expectations regarding system functionality, quality assurance, and teamwork.

As a group, we deliberated the given sprint expectations and stretch goals to ultimately formulate our own system functionality goals. We determined a set of primary system functionality goals for this project:

1. Add the notion of users and groups to the system
2. Enable messaging between users and groups
3. Enable a duping functionality
4. Queue messages for offline users
5. Add message history and persistence enhancements

From an environmental standpoint, our goal was to host the Prattle server on an Amazon Web Services (AWS) cloud instance where it would run continuously. Our team also decided to integrate a MySQL database instance to provide data storage for the server. Throughout the project duration, we agreed upon additional stretch goals, such as parental controls, searching for messages, and recalling messages, once we were comfortable that the main system functionalities were met and were performing adequately.

Given the scope of the project and the complexity of the messaging system, quality assurance was an essential activity to verify system functionality. The goal of our test suite was to achieve at least 85% branch coverage and at least 50% condition coverage. To meet these percentage marks we added test classes for the legacy code and for all the newly added functionalities. The team also set an expectation of annotating all methods with Javadoc comments and additional explanations when necessary.

Teamwork is a crucial component to any group project. To keep everyone accountable and involved, we set a goal to meet in person as a group at least once per week over the course of the project. Our intention was to meet once in the beginning of the sprint to assign tasks and talk about sprint

expectations and then to meet once at the end of the sprint to merge features into the master branch. We also encouraged individual daily updates, questions, comments, and general team communication on Slack during the days that we did not meet. In terms of project organization, our goal was to update JIRA regularly to track task completion and to use smart commits when committing any changes to the repository.

Overview of Project Results

System Functionality

1. Create, read, update, and delete (CRUD) operations for users and groups
    - A user can register a username (if not already registered) with our server. They just need to provide a username and a password which is then encrypted and stored in the database. The user can then login at any point in the future with the registered credentials. They may also choose to update their password or delete their account altogether. Similar operations are available for group management. A user may create a group and add members to it. They may also update the group with new members or delete the group.

2. Private messaging between users
    - In sprint 2, we added the functionality for private messaging between two client users. A user can send a direct message to another user by specifying the message type, recipient name, and message text. The message type in this case would be PVT (private).
3. Ability to send a group message from a user to a group
    - A user can also send a group message to an existing group of users even if they are not a part of that group. Here too, the user must specify the message type (“GRP”) and recipient group name along with the message text.

4. Message persistence through use of MySQL database

- We integrated a MySQL database to add the functionality to store all messages sent to the server, which would allow us to search for messages. Each message is stored in the database with its original message text, timestamp, sender's username, receiver's username, and the message type.

5. Queuing messages for offline users
    - When private or group message is sent to clients that are not connected to the server, these messages get stored in our database and are marked as unsent. When the receiving user logs in, our system queries the database for messages that have not been sent to the receiving user yet. The unsent messages are returned from the database and sent to the user. These messages are then marked as sent.

6. Recall unsent messages
    - If a user sends a message that they did not intend to and the recipient if offline, the user may choose to recall the message. Recalled messages are still stored in the database, however, they would not be displayed in any message search query that a user requests.
7. Duplicating (dup'ing) a message targeting a specific user or group
    - An "agency" user has been created that can request access to wiretap incoming and outgoing messages of targeted users or a single group. This user cannot perform any other functionality (read-only access).

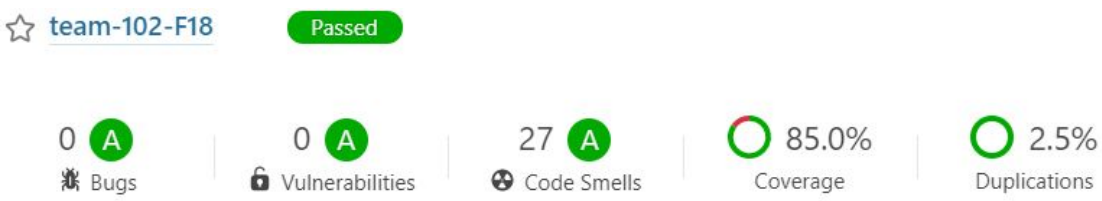
8. Message history and searching
    - Users can query their own message history with another user by specifying the other user's username and the number of messages to retrieve. The search message feature allows users to search all messages by specifying a type of message (outgoing or incoming), a targeted user (or all users), and a time frame.

9. Parental Controls
    - A user may choose to turn parental controls on or off while they are online. If they set parental controls to "on", any messages containing inappropriate words will be flagged for abusive content. The sending user will receive a warning message from the system. The inappropriate message will also be marked as inappropriate in our database.

10. Wrap IP addresses of the sender and receiver with each message
    - For each message sent, our system wraps the sender's and receiver's IP addresses with the text of the message upon storing it in the database. A user's IP address is updated every time they log in.

Code Quality

The code was thoroughly tested using Java's JUnit test framework (version 5). A modular testing model was followed as each method was tested as its own test case. We were able to achieve our goal of 85% branch coverage with our testing framework. Tests were also written keeping conditional coverage in mind. We exceeded the conditional coverage criteria by achieving 78% conditional coverage. Code duplication was kept below the 3% mark, as literal and string constants were used rather than hard coding. The code focused on minimizing bugs and system vulnerabilities. The system code has been heavily documented using Javadoc comments to facilitate future hand over of the project.



```java
/**
 * sends the message to active client threads with a particular user name
 *
 * @param message       the message to be sent
 * @param recipientName the name of the recipient the message has to be sent
 */
public static void directMessage(Message message, String recipientName, boolean checkUser) {
        UserCrud uc = new UserCrudImpl();
        int flag = 0;
        // Loop through all of our active threads
        for (ClientRunnable tt : active) {
```

Development Process

Our team met at least once per week for every sprint for task setup and feature merging. Each team member was responsible for their own modules and streams. During feature merging, we had a staging branch for each sprint. All team members merged their code with the staging branch which was eventually merged to master.

Communication regarding design, issues, questions, comments and daily updates happened over Slack. The work was divided amongst team members by the team's mutual agreement. All team members were responsible for writing unit test cases for their modules and made sure the unit test cases covered the 85% branch coverage and 50% conditional coverage. This made the process of merging to master much easier.

Our application is hosted in an AWS server that is accessible 24x7. After each deployment, each team member is responsible for testing their module in order to make sure the functionality is working on the AWS server. We are also using a MySQL database for managing data. The database is also available 24x7 and can handle high volumes of messages, users, and group.

Challenges and Solutions

1. We had difficulty scheduling in-person meetings. We resolved this by communicating heavily over slack. All team members were available on slack throughout the day in case any issues arose.
2. We had a task collaboration issue during Sprint 2 where multiple group members worked on the same feature which caused some duplication of work. We resolved this issue by consistently using JIRA for task tracking during Sprint 3.

3. At times we had difficulty in debugging issues and errors in Jenkins. We understand that Jenkins should not be used for debugging an issue. To debug our issues in a Linux environment we used our AWS server and replicated the same docker environment in our local systems to test how we can mock our database calls.
4. In Sprint 1 we were struggling with reaching the 85% code coverage mark. To resolve this issue, we used the Chatter application as a dependency in our application. This allowed us to reach the required coverage.

What did not work

1. We developed a feature that created a clean database instance for each test using the Mariadb4j library, to prevent tests from changing the deployed system while still being able to test the details of database interactions. This was successfully tested on our local and deploy machines, but due to a specific incompatibility with Alpine (which is used in the automated testing server), we could not merge this feature with the master branch during the project timeframe.
2. Initially we had planned to use text files for managing users and groups. However, this did not work for us since we were unable to modify a text file that was present in a jar during runtime.

Retrospective

As a team we have developed a product which is easy to manage and maintain. We used best coding practices including Javadocs comments, unit test cases, usage of properties files, and design patterns.

Parts of the project the team enjoyed

1. Usage of Continuous Integration and Deployment
2. Learning how to use JIRA for task allocation
3. Learning how java sockets operate
4. Learning how to write JUnit test cases

Parts of the project the team did not enjoy

1. Some of the requirements were ambiguous and it was left up to the team members to implement what they think the functionality should be. We would rather prefer a concrete set of requirements.

2. Considering the time frame for each sprint, we would have preferred if there was an application code walk session beforehand. This would have made sure all team members understood what the application did before writing the test cases and code.
3. Sprint 1 had many issues including most teams using the same port for testing the application which resulted in failure of test cases. The solution to this issue should have been discussed beforehand and would have saved everyone a lot of time and effort.

The project had many high and low points. The most important factor which resulted from this project was teamwork. We learned how to work together as a development team and by the end of sprint 2, the development process was much smoother for our team. Each member of our team contributed equally and, in the end, we were able to produce a maintainable, quality product. Our team has a few recommendations that we believe can help make this course an even better experience in the future:

1. Team members should be allowed to create or modify jobs in Jenkins.
2. Slack and Jenkins integration should be done by team members rather that producing a webhook URL and sharing it with Alex. In the end, we do not know how the integration is happening behind the scene.
3. There should be documentation on how to replicate the environment and Jenkins pipeline on your local system. It will help all teams to debug their issues and fix them locally.