WeeGee von Michael Nunes Jacobs für LF 6

Technische Dokumentation
und
Bestandsaufnahme

Verwendete Technologien:

Apache Commons Library 3 um Strings zu parsen
JavaFX für die GUI (zweite Version)
JSwing und WindowBuilder für die GUI (erste Version)
TCP und Telnet für die Kommunikation
Java für das komplette Programm
Css für JavaFX Oberfläche

Wie funktioniert das Programm?

Serverside:
Zuerst muss der Server gestartet werden. Dieser initialisiert einen Server der einen Thread startet der kontinuierlich alle einkommenden Verbindungen akzeptiert, die Clienten in eine Liste anhängt die dann von einem ServerWorker verwaltet werden.
Dieser ServerWorker verwaltet die Einzelverbindungen und parst die einkommenden Nachrichten in einem Thread. Je nachdem was am Anfang einer einkommenden Nachricht steht wird in dem Switch Case der passende Fall gefunden. Wenn nichts gefunden wird verwirft er die Nachricht. Die Cases sind in dem Interface ChatCommands.java dokumentiert.

```
while ((line = reader.readLine()) != null) {
    String[] tokens = StringUtils.split(line);
    if (tokens != null && tokens.length > 0) {
        String cmd = tokens[0];
        switch (cmd.toLowerCase()) {
        case (TERMINATE):
            handleLogoff();
            break;
        case (LOGOFF):
            handleLogoff();
            break;
        case (LOGIN):
            handleLogin(outputStream, tokens);
            break;
        case (MESSAGE):
            String[] tokensMessage = StringUtils.split(line, null, 3);
            handleMessages(tokensMessage);
            break;
        case (JOINGROUP):
            String[] tokensGroup = StringUtils.split(line, null, 3);
            handleGroupChatJoin(tokensGroup);
            break;
        case (LEAVEGROUP):
            handleGroupChatLeave(tokens);
            break;
        default:
            String msg = "unknown " + cmd + "\n";
            outputStream.write(msg.getBytes());
            break;
        }
    }
}
clientSocket.close();
```

```
public interface ChatCommands {
    static final String TERMINATE = "exit";
    static final String LOGIN ="login";
    static final String LOGOFF = "logoff";
    static final String MESSAGE = "msg";
    static final String JOINGROUP = "join";
    static final String LEAVEGROUP = "leave";
    static final String ONLINE = "online";
    static final String OFFLINE = "offline";
}
```

Es ist wohl offensichtlich was die Funktionen machen. Um zu parsen nutze ich die StringUtils Klasse mit der Methode split um die einkommenden Strings in Arrays zu

stecken. Hier ein Beispiel anhand der handleMessage Methode:

```
private void handleMessages(String[] tokens) throws IOException {
    String receipentOrGroup = tokens[1];
    String msgBody = tokens[2];
    boolean isItAGroupChat = receipentOrGroup.charAt(0) == '@';
    List<ServerWorker> workerList = server.getWorkerList();

    for (ServerWorker worker : workerList) {
        if (isItAGroupChat) {
            if (worker.isMemberOfCurrentTopic(receipentOrGroup)) {
                String msg = "msg " + receipentOrGroup + ": " + "<" + getLogin() + "> " + " " + msgBody + "\n";
                worker.send(msg);
            }
        } else {
            if (receipentOrGroup.equalsIgnoreCase(worker.getLogin())) {
                String msg = " msg " + "<" + this.login + ">" + " " + msgBody + "\n";
                worker.send(msg);
            }
        }
    }
}
```

Wenn eine Nachricht reinkommt wird die Nachricht in drei Teile aufgeteilt. Die Methode splittet bei Leerzeichen im String aber maximal dreimal. Heißt wenn jemand eine lange Nachricht schreibt wird diese Nachricht nicht gesplittet. Daher ist es wichtig vorangehend mit msg zu schreiben gefolgt von einem Empfänger und an dritter Stelle dann der MessageBody. Nun iteriere ich durch die ServerWorker Liste um einen Empfänger zu finden. Wenn der Empfänger mit einem @ beginnt ist es eine Gruppenchat Nachricht und ich iteriere durch die Gruppenchat Liste. Wenn ein Empfänger gefunden wurde ruft der Server die Send funktion auf und sendet sie an alle Clienten die zuhören.

Login der Clienten ist noch hardcoded im Quelltext. Hierzu werde ich noch eine Datenbank aufsetzen.

```
private void handleLogin(OutputStream outputStream, String[] tokens) throws IOException {
    if (tokens.length == 3) {
        String login = tokens[1];
        String password = tokens[2];

        // login abfrage mit Datenbank wird hier sein
        if ((login.equals("guest") && password.equals("guest")
                || login.equals("jeff") && password.equals("jeff"))) {

            String msg = "ok login\n";
            outputStream.write(msg.getBytes());
            outputStream.flush();

            this.login = login;
            System.out.println("successfully logged in: " + this.login);

            List<ServerWorker> workerList = server.getWorkerList();

            // send current user all other online logins
            for (ServerWorker worker : workerList) {
                if (worker.getLogin() != null) {
                    if (!login.equals(worker.getLogin())) {
                        String msg2 = "online " + worker.getLogin() + "\n";
                        send(msg2);
                    }
                } else {
                    System.err.println("User is already signed in.");
                }
            }

            // send other online users current users status
            String onlineMsg = "online " + login + "\n";
            for (ServerWorker worker : workerList) {
                if (!login.equals(worker.getLogin())) {
                    worker.send(onlineMsg);
                }
            }
        } else {
            String msg = "nah Error login\n";
            outputStream.write(msg.getBytes());
            outputStream.flush();
            System.err.println("Login failed for " + login);
        }
    }
}
```

Wenn jemand sich einloggt wird auch an alle anderen eine Nachricht gesendet, dass dieser User online ist. Genauso kriegt jeder User beim einloggen Benachrichtigung darüber welche User alles online sind. Dies ist Notwendig um später eine Userliste anzuzeigen.

Clientside:
Als Einstiegspunkt wählen wir das LoginWindow.java. Als erstes wird ein connect() versucht und wenn dieser erfolgreich war starten wir den Login, der dann den Clienten anwirft. Nun wird die UserListe angezeigt und das LoginFenster ausgeblendet. Diese UserListe meldet sich als neuen Clienten an und schreibt sich in die UserListe rein. Wenn eine Nachricht reinkommt wird eine neue Notification gesendet. Nun zu dem Clienten der Teil jeder GUI ist.
Hier sind zwei ArrayListen die auf einkommende Nachrichten horchen. Einmal UserStatusListener und einen MessageListener. Als Einstiegspunkt ist die Klassenmethode gewählt und setzt mit this die Verbindungsparameter. Wenn das LoginFenster geklappt hat und die connect() Funktion und die Client.Login() funktion true zurückgibt starten wir den readMessage-Thread der permanent auf Nachrichten horcht. Wenn jetzt eine Nachricht reinkommt wir die von handleMessage() bearbeitet. HandleMessage() iteriert über alle angemeldeten MessageListener und stellt sie dem User zu.

Was fehlt jetzt noch?
Zuerst müssen die Nachrichten gespeichert werden können und Logins kreiert werden können. Ich habe zwar im Antrag geschrieben es soll ohne Logins funktionieren aber da habe ich mich dann doch gegen entschieden. Es fehlt noch die komplette Kalenderfunktion und das erstellen von Terminen + vergeben von Aufgaben. Die Desktop-Benachrichtigungen sollten auch gesendet werden, wenn das Chatfenster geschlossen ist und wenn dieses geschlossen ist sollte ein Klick auf die Benachrichtigung einen direkt ins ChatFenster weiterleiten und die Nachricht nochmal anzeigen.
Da JSwing nicht meinen Designansprüchen gerecht wird habe ich vor das Design neu mit JavaFX zu machen. Dieses Design habe ich mit dem SceneBuilder bereits erstellt es muss nur noch die Logik implementiert werden.