Fachhochschule Nordwestschweiz
Hochschule für Wirtschaft

Dokumentation Projekt 2:
Project option 2: Implement a client

Projektmitglieder:
Severin Nyffenegger
Kevin Bernet

Dozent:
Bradley Richards

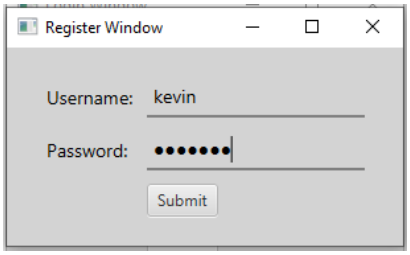
Olten, 26.11.2023

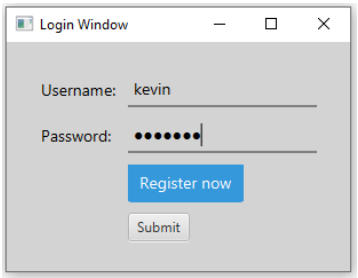
Programm starten und Erklärung:

Nachdem man das Projekt von GitHub geklont hat, sind unsere Files unter folgendem Pfad auffindbar:
src/main/java/chatroom/server/FX
Damit das Programm problemlos funktioniert, muss Java21 installiert und ausgewählt sein.
Wenn man das Programm nur mit einem Lokalen Server ausführen will muss zuerst das Java-File «Server» unter:
src/main/java/chatroom/server
gestartet werden. Nachdem der lokale Server gestartet wurde, kann im FX Ordner das Java-File: **ChatApp** gestartet werden. Somit wird unsere Applikation gestartet. Das Programm startet auch ohne, dass der Lokale Server läuft, jedoch wird im Programm nichts funktionieren.
Man kann jedoch die Serveraddresse unter dem Tab Server ändern auf: javaprojects.ch:50001. Dann funktioniert es auch ohne den lokalen Server.
Die Serveraddresse kann auch vor dem Start des Programms in der Klasse: Model in Zeile 17 und 18 wie im Bild rechts gezeigt geändert werden. Das GUI sieht wie folgt aus:

```
public Model() {
    this.serverAddress = "javaprojects.ch";
    this.serverPort = 50001;
    this.chats = new HashMap<>();
    this.userChats = new ArrayList<>();
}
```

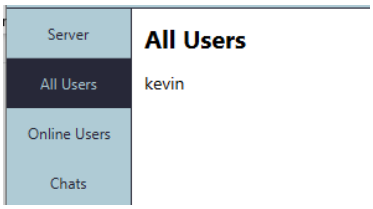
Oben rechts kann man sich über den Login-Button einloggen oder wenn der User noch nicht registriert ist, kann man sich registrieren. Nachdem man eingeloggt ist, wird er in einen Logout-Button verwandelt. Wenn man das Programm schliesst, ohne sich vorher auszuloggen, loggt sich das Programm für den User aus.



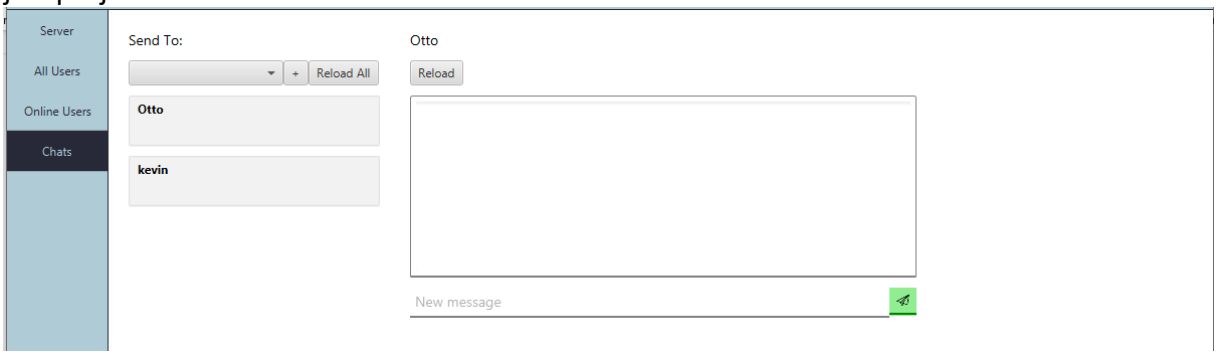


Auf der linken Seite können mittels der Navigationsliste die unterschiedlichen Tabs geöffnet werden. Unter dem Tab Server kann man die Serveraddresse anpassen und Pingen. Wenn die Serveraddresse mit dem gestarteten lokalen Server übereinstimmt oder mit dem von Bradley zur Verfügung gestellten Server, bekommt man einen positiven Alert, sonst einen negativen Alert, dass der Server nicht gepingt werden konnte.

Die beiden nächsten Tabs sind genau gleich aufgebaut, der einzige unterschied ist, dass im All User, wie der Name sagt, alle Registrierten User anzeigt. Im gegensatz zeigt der Online Users nur die User an, die gerade eingeloggt sind und Nachrichten empfangen können. Diese beiden Tabs hätten wir in einer echten Applikation nicht reingenommen, die haben wir fürs Testen benutzt.



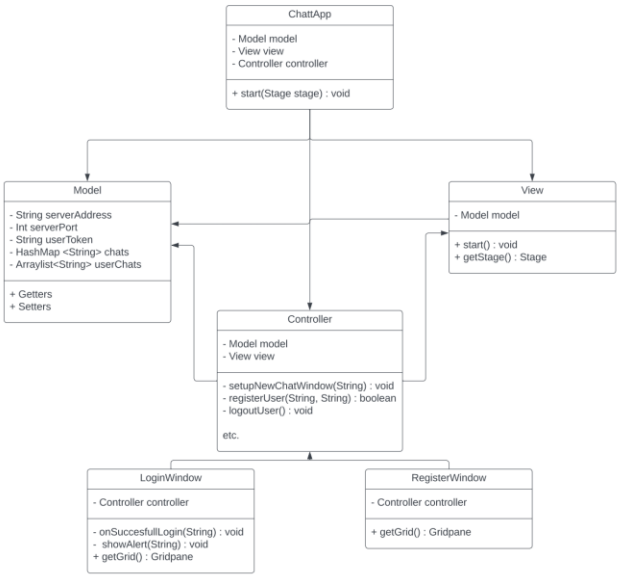
Im untersten Tab Chats wirds spannender. Dafür haben wir uns für den Test mit dem javaprojects.ch:50001 Server verbunden. Der Tab sieht so aus:



Auf der Linken Seite werden in der ComboBox nur die User angezeigt, die online sind. Nachdem einUser darin ausgewählt ist, kann ein neuer Chat mittles dem Button (+) geöffnet werden. Wenn man dann auf den Button unten mit dem Namen drückt (im Bild oben Otto oder Kevin) wird rechts der Chat angezeigt. Wenn ein User hinzugefügt wird, wird er aus der ComboBox gelöscht, dass nicht ein zweiter Chat mit dem User geöffnet werden kann.
Mit dem Button (Reoad All) können alle eingehenden Chats empfangen werden. Wenn bereits ein Chat entesteht, werden die Nachrichten direkt dem Chat hinzugefügt. Um sie Anzuzeigen, muss nochmals auf den Chatbutton mit dem Namen gedrückt werden. Wenn noch kein Chat besteht der eingehenden Nachricht wird ein neuer Chat eröffnet.
Auf der Seite wird oben der Name angezeigt, mit wem gechattet wird. Untendran werden die aus- und eingehenden nachrichten angezeigt. Zuunterst können neue Nachrichten erfasst und versendet werden.

Wie unser Programm aufgebaut ist:

Wir haben uns für die klassische MVC-Architektur entschieden, welche im Ordner FX abgelegt ist. In der Klasse Model speichern wir unseren Default Server, sowie alle Chat Nachrichten und die User-tokens. Der Server kann entweder im Quell-Code angepasst werden oder im GUI. In der Klasse View haben wir alle unseren GUI-Komponenten gespeichert. In der Klasse Controller reagieren wir auf die Events, welche auf dem GUI passieren. Ebenfalls rufen wir aus dieser Klasse zwei weiter Klassen auf, das LoginWindow und das RegisterWindow. Dies ermöglicht uns eine klare getrennte Sichtweise auf unser Programm. Untenstehend sind die Wichtigsten Methoden aufgelistet.



Post Server Anfragen:

```
String loginEndpoint = "http://" + model.getServerAddress() + ":" + model.getServerPort() + "/user/login";

try {
    URL url = new URL(loginEndpoint);
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("POST");
    connection.setRequestProperty("Content-Type", "application/json");

    String jsonInputString = "{\"username\": \"" + username + "\", \"password\": \"" + password + "\"}";
    connection.setDoOutput(true);

    connection.getOutputStream().write(jsonInputString.getBytes( charsetName: "UTF-8"));
```

Unsere Server Anfragen funktionieren immer mit dem gleichen Prinzip, wir geben in der Methode der User-Name, das User-Passwort oder der Token als Parameter. Als nächstes werden mit den Hinterlegen Serverdaten unsere URL aufgebaut, dies geschieht mit loginEndpoint. Danach erstellen wir mittels url.openConnection() eine Verbindung zum Server, zugleich legen wir fest was es für eine Anfrage ist (Post oder Get) und welchen Datenaustausch wir wollen, in unserem Fall ist dies JSON. Der nächste Schritt ist die Nachricht an den Server auszubauen, dazu werden unsere Parameter der Methode verwendet, in diesem Fall ist dies der User-Name und Passwort und jsonInputString erstellt. Mit getOutputStream().write() und getBites() wandeln wir unseren String in lesbare Bites um und senden dies Zugleich dem Server. Im Login Fall brauchen wir vom Server eine Antwort, welche den Token beinhaltet. Diesen Token geben wir auch als return in der Login-Methode zurück. Mittels getResponseCode() können wir überprüfen, ob unser Login Erfolgreich war. Falls dies so ist, können wir mittels StringBuilder und BufferReader die Antwort des Servers auslesen und unseren Token als String zurückgeben. Genauer gesagt bauen wir einen BufferedReader auf, um den InputStream vom Server zu lesen, jedoch nur so lange bis es keine Zeilen mit null gibt. Wir wandeln die Response-

```
int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_OK) {
    StringBuilder response = new StringBuilder();
    try (var reader = new BufferedReader(new InputStreamReader(connection.getInputStream()))) {
        String line;
        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
    }
    JSONObject json = new JSONObject(response.toString());
    return json.getString( key: "token");
```

String in ein JSON-Objekt um, um danach nur den Token im Return Statement in einen String umzuwandeln. Falls wir keinen Token als Antwort erhalten, geben wir null zurück.

Im Falle vom Registrieren eines Unsers, den Server Pingen oder eine Nachricht zu senden brauchen wir lediglich die Nachricht true, um zu überprüfen, ob es geklappt hat. Dies machen wir mit der Methode getResponseCode(), welche eingebaut in Java ist. Mit einem simplen if-Statement können wir mittels HttpURLConnection.HTTP_OK überprüfen, ob die Antwort 200 ist. Falls dies so ist, können wir in der Methode den Wert true zurückgeben, falls nicht geben wir false zurück.

```
int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_OK) {
    System.out.println("Message sent successfully!");
    return true;
} else {
    System.out.println("Message sending failed. HTTP Response Code: " + responseCode);
    return false;
}
```

Get Server Anfragen:

Diese Methoden sind im Grunde gleich aufgebaut, wie unsere Post-Methoden, nur wir die Anfrage Methode auf «GET» angepasst. Wir überprüfen wieder Mittels getResponseCode(), ob die Verbindung erfolgreich war und lesen im Anschluss die Antwort aus. Dies basiert auf dem selben Prinzip wie den Token auslesen. Die Methode parseAllUserList() wird nur dazu benötigt aus der Antwort eine brauchbare Liste zu erhalten. Die zwei Methoden werden getrennt, um eine bessere übersicht im Code zu haben. Das gleiche Prinzip wenden wir auch, um die Online-User auszulesen, nur ersetzen wir json.getJSONArray(„users“) von users auf „online“ um. Dies ermöglicht uns alle User auszulesen welche Online oder Registriert sind, um falls notwendig Debugs zu machen. In einem wirklichen Projekt wären diese zwei Methoden nicht sehr sinnvoll.

```
String serverEndpoint = "http://" + model.getServerAddress() + ":" + model.getServerPort() + "/users/online";

try {
    URL url = new URL(serverEndpoint);
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("GET");

    int responseCode = connection.getResponseCode();

    if (responseCode == HttpURLConnection.HTTP_OK) {
        StringBuilder response = new StringBuilder();
        try (var reader = new BufferedReader(new InputStreamReader(connection.getInputStream()))) {
            String line;
            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
        }
        return parseOnlineUserList(response.toString());
    } else {
        System.out.println("Error: " + responseCode);
        return new ArrayList<>();
    }
} catch (IOException e) {
    e.printStackTrace();
    return new ArrayList<>();
```

```
private List<String> parseAllUserList(String response) {
    try {
        JSONObject json = new JSONObject(response);
        JSONArray onlineUsers = json.getJSONArray( key: "users");

        List<String> userList = new ArrayList<>();
        for (int i = 0; i < onlineUsers.length(); i++) {
            userList.add(onlineUsers.getString(i));
        }
        return userList;
    } catch (JSONException e) {
        e.printStackTrace();
        return new ArrayList<>();
    }
}
```

Methoden

Einzelne Methoden, die nicht selbsterklärend sind, sind direkt im Code mit Kommentaren beschrieben.