Laborprotokoll

DEZSYS-12: Mobile Chat Application

Systemtechnik
5BHITT 2015/16

Simon Wortha

Version 1.0

Note:

Betreuer: Prof. Borko

Begonnen am 22. April 2016

Beendet am 26. April 2016

Inhaltsverzeichnis

Einführung

Diese Übung soll eine Vertiefung des Wissens für mobile Anwendungen darstellen.

1. Ziele

Das Ziel dieser Übung ist die Entwicklung eines serverbasierten Gruppenchats. Die Applikation soll auf den Entwicklungen der letzten beiden Übungen aufbauen und diese um folgende Funktionalität erweitern:

- Anbindung mit Hilfe eines RESTful Webservice
- Zur Teilnahme beim Gruppenchat ist eine Anmeldung erforderlich (Username/Passwort)
- Abmeldung des Users am Ende der Chatsession
- Verwendung des Observer-Pattern

2. Voraussetzungen

- Grundlagen Java und XML
- Grundlegendes Verständnis über Entwicklungs- und Simulationsumgebungen
- Verständnis von RESTful Webservices
- Grundlegendes Wissen zum Design Pattern "Observer" und dessen Umsetzung

3. Aufgabenstellung

Es ist eine mobile Anwendung zu implementieren, die sich mit Hilfe der Übung DezSysLabor-11 "Mobile Access to Web Services" bei einem Gruppenchat anmeldet. Nach erfolgreicher Anmeldung bekommt der Benutzer alle Meldungen, die in diesem Chat eingehen. Der Benutzer hat ebenso die Moeglichkeit Nachrichten in diesem Chat zu erstellen. Diese Nachricht wird in weiterer Folge an alle Teilnehmer versendet und in der mobilen Anwendung angezeigt. Am Ende muss sich der Benutzer vom Gruppenchat abmelden.

Es ist freigestellt, welche mobile Implementierungsumgebung dafür gewählt wird. Empfohlen wird aber eine Implementierung auf Android.

4. Quellen

"Android Restful Webservice Tutorial – How to call RESTful webservice in Android – Part 3"; Posted By Android Guru on May 27, 2014; online: http://programmerguru.com/android-tutorial/android-restful-webservice-tutorial-how-to-call-restful-webservice-in-android-part-3/

Ergebnisse

5. GitHub

https://github.com/swortha-tgm/DezSys-12

6. Implementierung: Server

Wie in DezSys09, bietet der RestFul Webservice auch hier wieder /login und /register an. [5]

Zu dieser Implementierung mussten allerdings noch Erweiterungen hinzugefügt werden.

Login-Endpoint

Zum Login-Endpoint wurde hier einfach ein Session-Management hinzugefügt.

```
@POST
public Response post(User user) {
	if (user.getEmail() == null)
		return Util.getResponse(Response.Status.BAD_REQUEST, "No email
specified!");

User userFromDb = userRepository.findOne(user.getEmail());
if (userFromDb != null && userFromDb.getPwhash().equals(user.getPwhash())) {
	Map<String, String> response = new HashMap<>();
	UUID uuid = sessionManager.newSession(userFromDb); response.put("sid",
		uuid.toString());
	response.put("code",
		String.valueOf(Response.Status.OK.getStatusCode()));
	response.put("message", "Welcome " + userFromDb.getName());
	return Response.status(Response.Status.OK).cookie(new NewCookie("sid",
	uuid.toString())).entity(response).build();
} else {
	return Util.ge    tResponse(Response.Status.FORBIDDEN, "Invalid account
	data!");

 }

}
```

Mithilfe des SessionManagers wird, nach einer erfolgreichen Anmeldung, eine neue Session gestartet. Die generierte ID wird in dem JSON-Objekt, welches zurück geschickt wird, und als HHTP-Cookie mitgeliefert.

Chat-Endpoint

Der Chat-Endpoint ist unter /chat/<roomid> abrufbar, wobei mittels POST eine Nachricht gesendet werden kann.

Body:

```
{
"content": "Message"
}
```

```
@POST
@Path("/{chatroomid}")
public Response sendMessage(@Context HttpHeaders headers,
@PathParam("chatroomid") String chatroomid, Message sentMessage) {
      String uuid = headers.getCookies().get("sid").getValue();
User user = sessionManager.getUser(uuid);
if (user == null)
      return Util.getResponse(Response.Status.BAD_REQUEST, "No open session
      with this ID. Try reconnecting to the chat.");

String messageContent = sentMessage.getContent().trim();
if (messageContent.isEmpty())
      return Util.getResponse(Response.Status.BAD_REQUEST, "Empty message");

Message message = new Message(chatroomid != null ? chatroomid :
      DEFAULT_CHATROOM, user.getName(), messageContent);

 this.chatroomHandler.sendMessage(message);

return Util.getResponse(Response.Status.OK, "Message sent");

}
```

7. Nachrichtenempfangen

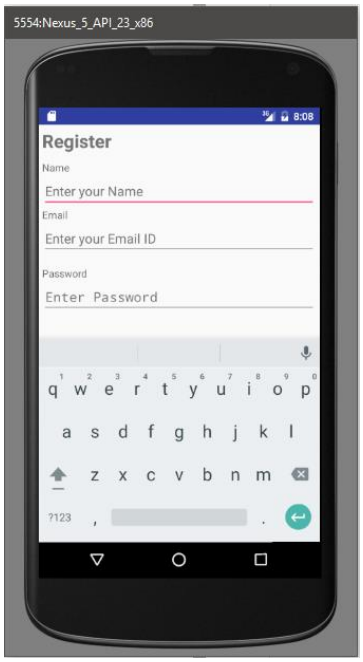
Dies wurde mittels Long Polling gelöst. Der Server hält die Verbindung so lange aufrecht, bis eine neue Nachricht eintrifft. Zum eigentlichen Empfangen der Nachricht, wird ein GET-Request auf /chat/roomid verwendet. [7]

```
@GET @Path("/{chatroomid}")
public Response waitForMessage(@Suspended AsyncResponse asyncResp, @Context
HttpHeaders headers, @DefaultValue("-2") @QueryParam("messageindex")
int messageindex, @PathParam("chatroomid") String chatroomid) {

      String uuid = headers.getCookies().get("sid").getValue();
      User user = sessionManager.getUser(uuid);
      if (user == null)
            return Util.getResponse(Response.Status.BAD_REQUEST, "No open ses
            sion
            with this ID. Try reconnecting to the chat.");
      this.chatroomHandler.waitForMessage(chatroomid != null ? chatroomid :
      DEFAULT_CHATROOM, uuid, asyncResp, messageindex);

      return Response.ok().build();

 }
```

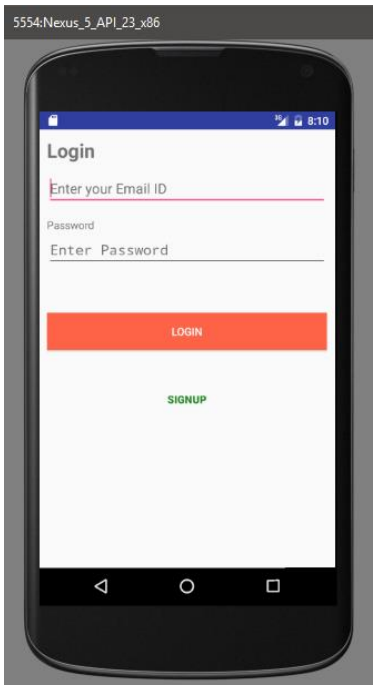
8. Implementierung: Client

Dazu wurde wie in DezSys-11 eine Android App programmiert, welche eine Login bzw. Register Ansicht ermöglicht. Wie diese Activities aussehen kann hier nachgelesen werden: [8]

Register:



Login:



9. Restful Webservice

Der Aufruf des Webservices funktioniert wie auch in DezSys-12 [8], mit der Klasse `Android Asynchronous Http Client`. Dadurch können asynchrone http-Requests versendet werden. (wie es eigentlich der Name schon verrät) Das ist notwendig, damit der GUI-Thread nicht blockiert wird.

Weiters ist es auch hier wieder notwendig festzulegen, dass die App auf das Internet zugreifen darf. Dazu fügt man im `AndroidManifest.xml` folgende Zeile hinzu:

```
<uses-permission android:name="android.permission.INTERNET" />
```

/login bzw. /register wird dann von der zuständigen Activity aufgerufen. Ähnlich wie in DezSys 12. [8]

Hier als Beispiel die RegisterActivity (Ausschnitt):

```
JSONObject params = new JSONObject();
StringEntity entity = null;

try {
	params.put("email", email);
	params.put("name", name); params.put("pwhash", password);
	entity = new StringEntity(params.toString(), "UTF-8");
} catch (JSONException e) {
	Log.e("Register", "Exception occurred", e);
}
if (entity != null)
	CustomRestClient.postJson(this, "register", entity, new
	JsonHttpResponseHandler() {
	@Override
	public void onSuccess(int statusCode, Header[] headers,
	JSONObject response) {

		showProgress(false);
		try {
			showProgress(false);
			mstatusText.setText(response.getString("message"));
			mregisterButton.setEnabled(false);
		} catch (JSONException e) {
			Log.e(TAG, "Failed getting message", e);
		}
	}
	@Override
	public void onFailure(int statusCode, Header[] headers, Throwable
	throwable, JSONObject errorResponse) {
		showProgress(false);
```

```
            try {
                  String errorString = errorResponse.getString("message");
                  showProgress(false);
                  mstatusText.setText(errorString);
} catch (JSONException e) {
Log.e(TAG, "Failed getting error message", e);
}
```

Nachdem sich der Benutzer erfolgreich angemeldet hat, wird dieser zur Chat-Ansicht weitergeleitet. Diese bietet ein Textfeld mit allen Nachrichten, ein Textfeld für die Eingabe einer neuen Nachricht und einen Button zum Senden der Nachricht. In einer Leiste am oberen Bildschirmende wird er Name des Chatrooms angezeigt.

Observer-Pattern

Das Empfangen von Nachrichten wurde mittels Long-Polling und eines Observer-Patterns gelöst.

```
private void checkForUpdates() {
// ..
      CustomRestClient.get(context, "chat/" + chatroomId, params, new
      JsonHttpResponseHandler() {

            @Override
            public void onSuccess(int statusCode, Header[] headers,
            JSONObject response) {

                  try {

                        lastMessageId = response.getInt("id");

                  } catch (JSONException e) {

                        Log.e("JSONException", "Could not get message id", e);

                  } setChanged(); notifyObservers(response);

                  if (polling) checkForUpdates();

            }
```

Feedback

Lessons Learned

- Festigung des Wissen von Android-App Programmierung
- Arbeiten mit Webservices

Probleme

Da wir keine Einführung bekamen und auch keine Zeit im Labor für diese Aufgabe hatten, gab es wiedermal ziemlich viel auf das man selbst draufkommen musste ;)

Zeitaufzeichnung

24. April	@home	10 Stunden
26. April	@home	4 Stunden
Gesamt		14 Stunden

Quellen

[1] "Referenzimplementierung von DezSys09"; Paul Kalauner; online: https://github.com/pkalauner-tgm/dezsys09-java-webservices

[2] "Android Asynchronous Http Client"; James Smith; online: http://loopj.com/android-async-http/

[3] "Android Asynchronous Networking and Image Loading"; Koushik Dutta; online: https://github.com/koush/ion

[4] "AndroidAsync"; Koushik Dutta; online: https://github.com/koush/AndroidAsync

[5] DEZSYS09-Webservices, Simon Wortha
Abrufbar unter: https://github.com/swortha-tgm/DezSys-09
zuletzt abgerufen: 26.04.2016

[6] Android Asynchronous Http Client
Abrufbar unter: http://loopj.com/android-async-http/
zuletzt abgerufen: 26.04.2016

[7] DezSys-12, Paul Kalauner
Abrufbar unter: https://github.com/pkalauner-tgm?tab=repositories
Zuletzt abgerufen: 26.04.2016

[8] DEZSYS11, Simon Wortha
Abrufbar unter: https://github.com/swortha-tgm/DezSys-11
zuletzt abgerufen: 26.04.2016