

Folie 1

07.07.2019

Food in Progress

Entwicklungsprojekt interaktive Systeme – SS19
Audit 3 - Implementierung

Dozenten: Kristian Fischer, Gerhard Hartmann
Betreuer: Corinna Sofie Klein, Robert Gabriel
Studierende: Kay Ruck, Tristan Schmele

Technology
Arts Sciences
TH Köln

Folie 2

Inhalt

- Implementierung
 - Client
 - Server
- Prozessassessment
- Fazit
- Plakat

2

Client - Retrofit

- Wichtige Bestandteile für einen Retrofit Client:
 - Interface, welche die REST Calls abbildet
 - Data Class, welche der JSON Struktur entspricht
 - Sowie ein JSON-Parser (wie GSON)

```

13 public interface AnzeigenInterface {
14
15     @Headers("Origin: NOTIHA17wV7v2bGM7oT")
16     @GET("anzeigen")
17     Call<List<Example>> getDisplayPosts();
18
19     @Headers({
20         "Origin: NOTIHA17wV7v2bGM7oT",
21         "Content-Type: application/json"
22     })
23     @POST("anzeigen")
24     Call<Data> postDisplayPost(@Body Data data);
25
26 }

```

```

Implementation: com.google.gson.GsonBuilder
Implementation: com.squareup.retrofit2.converter.gson.Gson
Implementation: com.squareup.retrofit2.converter.gson

```

Um Retrofit benutzen zu können, müssen Interface und die Data Class erstellt werden, Desweiteren kann ein JSON Parser implementiert werden, um das Auslesen der Daten zu vereinfachen. Anschließend ist wichtig den REST-Call nach einem Schema aufzubauen. Hierbei muss zuerst ein Retrofit Objekt erstellt werden. An diesen wird die IP und der Port in Form eines Strings gebunden. Nachdem das Retrofit Objekt erstellt wurde (mit build();) wird mittels des Interface und des Objekts einen neuer Endpunkt erstellt (Hierbei erstellen die Annotationen (REST-Aufrufe) einen neuen Endpunkt)

Client - Anfragen

- Retrofit Objekt initialisieren
- Endpunkt erstellen
- Call Objekt initialisieren
- Call durchführen & Callback Objekt erstellen
- onResponse und onFailure definieren

```
private void getMessages() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.4mat.de/v1")
        .addConverterFactory(GsonConverterFactory.create())
        .build();

    MessagesInterface api = retrofit.create(MessagesInterface.class);

    // Aufruf der REST-Methode
    api.getMessages().enqueue(new Callback<List<Message>>() {
        @Override
        public void onResponse(Call<List<Message>> call, Response<List<Message>> response) {
            // Erfolgsbehandlung
            List<Message> messages = response.body();
            // ...
        }
        @Override
        public void onFailure(Call<List<Message>> call, Throwable t) {
            // Fehlerbehandlung
            // ...
        }
    });
}
```

```
private void getMessages() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.4mat.de/v1")
        .addConverterFactory(GsonConverterFactory.create())
        .build();

    MessagesInterface api = retrofit.create(MessagesInterface.class);

    // Aufruf der REST-Methode
    api.getMessages().enqueue(new Callback<List<Message>>() {
        @Override
        public void onResponse(Call<List<Message>> call, Response<List<Message>> response) {
            // Erfolgsbehandlung
            List<Message> messages = response.body();
            // ...
        }
        @Override
        public void onFailure(Call<List<Message>> call, Throwable t) {
            // Fehlerbehandlung
            // ...
        }
    });
}
```

Um Anfragen an den Endpunkt verarbeiten zu können wird ein Call Objekt erstellt, welches den Typparameter des Daten Objektes besitzt und aus dem vorher erstellten Endpunkt, sowie der REST-Call Methode des Interface erstellt wird. Zum Schluss wird das erstellte Call Objekt asynchron abgeschickt. Als Übergabeparameter wird hierbei ein neues Callback Objekt mit dem Typparameter des Call Objektes erstellt. Dieses besitzt die Methoden onResponse(), welche ausgeführt wird wenn eine Response zurück geschickt wird, diese ist unabhängig vom Statuscode. Um nun nur die Erfolge abzufangen wird mit isSuccessul() abgefragt ob der zurückgegebene Statuscode im 200 bzw. 300 Bereich liegt. Die zweite Methode onFailure() wird dann ausgeführt wenn keine Response (Statuscode) zurück kommt.

Server – GET

```

41 router.get('/', (req, res) => {
42   if (req.header('origin') === undefined) {
43     res.statusMessage = 'origin undefined';
44     res.status(500).end();
45   } else {
46     msgClient.subscribe('/anzeige/' + req.header('origin'), message => {
47       [message.statusMessage] =
48         res.statusMessage = message.statusMessage;
49     }
50     res.status(message.status).json({
51       anzeigen: message.results
52     });
53     msgClient.unsubscribe('/anzeige/' + req.header('origin'));
54   });
55   if (req.query.radius === null || req.query.radius < 0) {
56     msgClient.publish('/anzeige/alle', {
57       origin: req.header('origin'),
58       action: 'set',
59       radius: req.query.radius
60     });
61   } else {
62     msgClient.publish('/anzeige/alle', {
63       origin: req.header('origin'),
64       action: 'get'
65     });
66   }
67 });
68 }
69 }
70 }
71 }

```

Anzeigenverwaltung

Gateway - /anzeige

```

40 msgClient.subscribe('/anzeige/' + req.header('origin'), message => {
41   var response = '/anzeige/' + message.origin;
42   return (channel) {
43     case '/anzeige/set':
44       break;
45     case '/anzeige/alle':
46       if (message.radius) {
47         rest_args.parameters = {radius: message.radius};
48         rest_args.headers = {origin: message.origin};
49       }
50       restClient.get(rest_url, rest_args, (data, response) => {
51         msgClient.publish(response, {
52           status: response.statusCode,
53           results: data
54         });
55       });
56     case '/anzeige/err':
57       msgClient.publish(response, {
58         status: 500,
59         results: {
60           error: err
61         }
62       });
63     break;
64   }
65 });

```

Alle Anfragen des Clients gehen in Gateway ein und werden dort auf notwendige Inhalte geprüft. Vollständige Anfragen werden mittels PubSub Messaging weitergesendet. Messages aus /anzeigen werden von Anzeigenverwaltung angenommen und dort in eine Restanfrage an Datenmanagement umgewandelt. Die Antwort hiervon wird per PubSub Messaging wieder an Gateway zurückgeschickt und von dort an den Client.

Server - Matching



- Nach Erstellung einer neuen Anzeige
- Filter nach Entfernung
- Aussortieren mit Tags
- Sortieren nach Bewertung

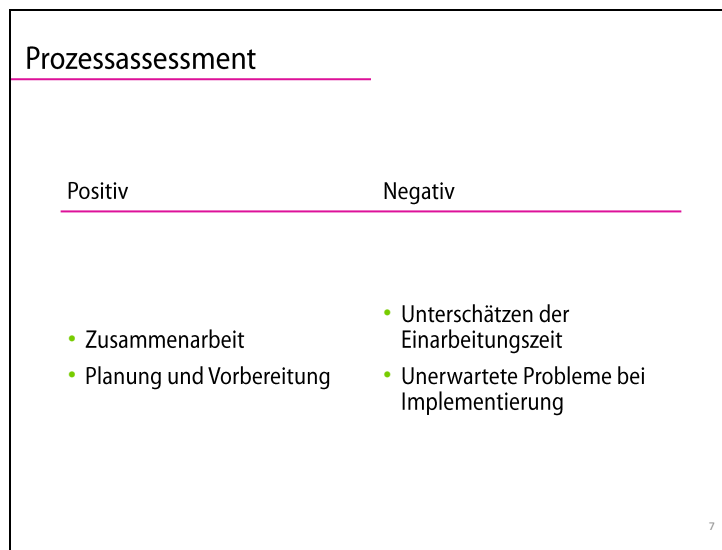
```

127: msgClient.subscribe('matching/monitor', null, function(channel, message) => {
128:   var origin = new GeoPoint(message.standort_latitude, message.standort_longitude);
129:   restClient.get(rest_url, rest_args, (data, response) => {
130:     var possible_matches = [];
131:     data.forEach(user => {
132:       var userLocation = new GeoPoint(user.data.standort_latitude, user.data.standort_longitude);
133:       var userDistance = userLocation.distanceTo(origin, true);
134:       if (userDistance < message.radius && user.id != message.actor) {
135:         possible_matches.push({
136:           id: user.id,
137:           data: user.data,
138:           distance: userDistance
139:         });
140:       }
141:     });
142:   });
143: });
  
```

6

Neu erstellte Anzeigen werden via PubSub Message zusätzlich an den Matching Dienst gesendet. Dieser fragt zunächst unter Angabe eines Standorts alle Benutzer in der Nähe der Anzeige an. Alle hierbei erfragten Benutzer werden dann mit den Tags der neuen Anzeige abgeglichen, um herauszufinden, welche der Benutzer an der Anzeige Interesse zeigen könnten.

Die anschließend übrigen Benutzer werden nach ihrer Bewertung sortiert und der beste Benutzer bekommt einen direkten Hinweis auf die erstellte Anzeige, um eventuell als erster von dieser zu erfahren und sie direkt reservieren zu können.

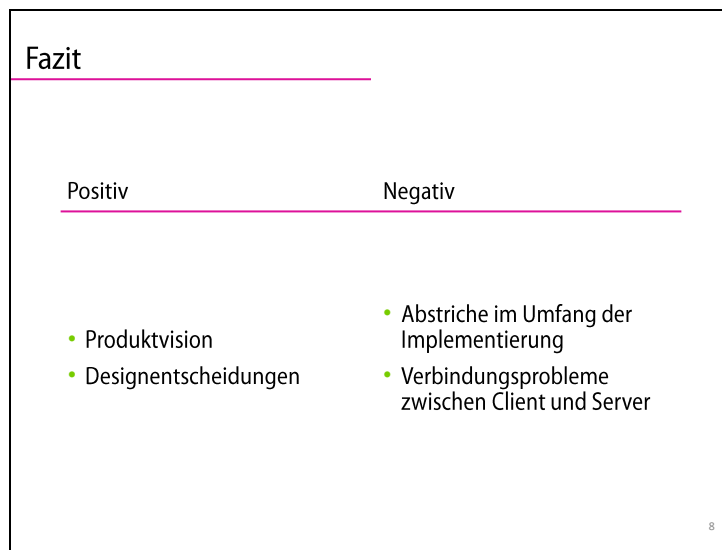


Im Anschluss an die Identifizierung des Problemraums, wurde zuerst ein Projektplan erstellt. In diesem wurden zunächst die notwendigen Schritte und Zwischenziele bis zur Erreichung des ersten Meilensteins, sowie die groben Schritte bis zum Projektabschluss festgehalten.

Durch großzügige Planung konnte der erste Meilenstein mit ausgiebigem Puffer erreicht werden.

Zu Beginn wurde primär im Team an allem zusammengearbeitet und besprochen. Dadurch waren alle Teammitglieder an den wichtigsten Entscheidungen beteiligt und über jedes Artefakt informiert.

In späteren Meilensteinen wurde die benötigte Zeit durch das problemlose Erreichen des ersten Meilensteins falsch eingeschätzt und es kam zu großen Verzögerungen. Um dennoch im Zeitplan bleiben zu können, wurden die weiteren Artefakte weitest gehend im Team verteilt und getrennt voneinander erarbeitet. Durch die Parallelisierung vieler Aufgaben konnte der zweite Meilenstein trotz Verzögerungen erfolgreich abgeschlossen werden. Insbesondere der Arbeitsaufwand des letzten Meilensteins wurde stark unterschätzt. Einzelne Designentscheidungen mussten nachträglich stark überarbeitet werden und die komplett selbstständige Einarbeitung des Teams in Android Studio sowie genutzte libraries führte zu massiven Verzögerungen. Aufgrund fehlender Zeit wurden die Projektziele neu bewertet und der Umfang des geplanten Prototyps reduziert, um die verbleibende Zeit effizienter verplanen zu können. Gerade die Implementierung des Microservice Architekturpatterns kostete deutlich mehr Zeit, als zu Beginn erwartet. Ebenso wurde die mobile Anwendung primär mit einem Fokus auf Funktionalität implementiert, wodurch sie nicht vollständig dem zuvor entwickelten und getesteten Design entspricht.



Zu Beginn des Projekts wurden Ziele und Anforderungen definiert, die für das System relevant sind. Im Laufe des Projekts konnten diese nicht vollständig erfüllt werden, da das System zu umfangreich geplant wurde und die Umsetzung in dem Angesetzten Projektzeitraum nicht möglich war.

Da der Zeitaufwand für die selbstständige Einarbeitung aller Teammitglieder in Android Studio, sowie die Umsetzung des Microservice Architektur-Patterns, unterschätzt wurde, konnten im Laufe des Projekts nicht alle geplanten Funktionen umgesetzt werden. Der entwickelte Prototyp ist in der Lage Angebote zu erstellen und anzuzeigen, während serverintern passende Matches berechnet werden können.

Gerade die Implementierung eines REST-Clients mit Retrofit erwies sich als deutlich komplizierter als erwartet, da sowohl Android als auch Retrofit, nicht ohne Weiteres Verbindungen zu unverschlüsselten Servern zulassen. Ebenso führte die gewählte Datenstruktur gegen Ende des Projektzeitraums zu unerwarteten Problemen bei der Konvertierung von JSON in Java Objekte, da einzelne Datensätze unterschiedliche viele Felder besitzen. Java kann damit nicht umgehen und bricht die Konvertierung ab.

Da viele Funktionen für den Prototypen gekürzt wurden, bietet das System bereits viel Raum für Erweiterungen. Zudem kann in Zukunft noch ein interner Nachrichtendienst implementiert werden, um die Kommunikation zwischen den Benutzern einfacher zu gestalten. Auch die Einführung von gemeinsamen Lagern für Wohngemeinschaften wäre eine denkbare Erweiterung. Mit wachsender Benutzerbasis werden sich in der Domäne noch weitere Gelegenheiten bieten die Anwendung zu erweitern, um allen Ansprüchen gerecht zu werden.



Zukunftsaussicht

Das System, für welches in diesem Projekt ein Prototyp erstellt wurde, soll nach seiner Fertigstellung einige wesentliche Funktionen besitzen, um Lebensmittelverschwendung zu vermeiden.

Nach der Erstellung einer Anzeige sollen noch Benachrichtigungen bei positiven Matches gesendet werden. Zudem sollen Anzeigen nicht nur in einer Liste, sondern auch auf einer Karte zu finden sein.

Erweiternd zur Weitergabe von Lebensmitteln soll in der Lagerungsfunktion ein Überblick über die aktuell privat verfügbaren Lebensmittel geboten werden. Hierbei soll zudem über bald ablaufende Produkte informiert werden, welche dann auf Wunsch direkt in eine Anzeige umgewandelt werden könnten.

Nach diesen Vervollständigungen könnte noch das ursprünglich konzipierte Forum zum Austausch über Wissenswertes zur Lebensmittelabfallvermeidung ergänzt werden. Dieses könnte mit einer internen Messaging-Funktion kombiniert werden, um Absprachen zwischen Benutzern direkt innerhalb des Systems klären zu können.