INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

An app for location-based feedback

Anton Borries

Bachelorarbeit

Beginn der Arbeit:	24. April 2017
Abgabe der Arbeit:	26. Juli 2017
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Stefan Conrad

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 26. Juli 2017

Anton Borries

Abstract

Feedbacker is an app which gives users the ability to send location-based feedback. The app is available for Android and iOS. It is using Firebase as its backend.
This thesis presents the app and gives possible use cases. It also explains how Firebase works and in which ways it was used for Feedbacker. A special focus lies on the database function of Firebase as well as the security of the saved data as it handles sensible information in the form of the user's location.
Furthermore, this work also shows the process of developing an App for iOS and Android at the same time. Here, extra focus was given to the process of getting the current user's location. Also, a perspective on how to develop the App further is given.
As the app is not yet published, its success of it needs to be determined at a later time.

Contents

1 Requirements

1.1 Motivation

The original idea was to develop an App that can be used to identify certain areas in cities where users feel afraid or, in a more general term, not safe.
In the process of developing the app the use case was extended to not only give users the ability to mark places as unsafe but also give the option of marking a place as positive. The process lead to the app called Feedbacker.

1.2 User groups

The main targeted user group are individual people. Users have the option to see feedback of other users in order to see how different areas are perceived, or to get ideas what to do/where to go while planning a trip.
City administrations might also have interest in the gathered data to identify urban areas that could be developed further, and to determine what makes certain regions stand out in comparison to other regions.

1.3 User Actions

1.3.1 Sending of Feedback

The user has the option of sending two kinds of Feedback, positive and negative. After the user has sent the feedback it will be saved to the database containing the user's current location, the current time and the kind of feedback.

1.3.2 Editing of Feedback

After sending the Feedback the user has the ability to further specify it.
One of the main points of editing is categorizing the Feedback. The available categories are:

positive	negative
Places to sit	Dark Place
Public/Clean Toilets	Dirty
Disablity Access	Shady Group of People
Public Wi-Fi	Not pedestrian friendly
Place to Eat/Drink	Graffiti
Nice View	

Table 1: Categories

To give additional information about the feedback or the place of the feedback the user

can write a descriptive text. If the user decides to mark the feedback as public, other users are able to see the feedback on a map. Users can also change their feedback from positive to negative (or vice versa), for example when a place was altered or if the user clicked the wrong button when sending his feedback.

1.4 Other Apps & Services

There are other apps and services which have some similarities to Feedbacker. In the following I show similarities and differences to two of those Apps.

Google Maps Google Maps has a lot of information on restaurants, shops and places in general. Users can not only rate and comment on places but Google also the option for users to add new places and enhance information like opening hours of existing places [Meh].
The information is however not as prominent and readily available as it is with Feedbacker, because in Google Maps the feedback of other users is not visible at first glance.

wheelmap.org "Wheelmap is a map for finding wheelchair accessible places. The map works similar to Wikipedia: anyone can contribute and mark public places around the world according to their wheelchair accessibility. The criteria for marking places is based on a simple traffic light system". It distinguishes between 130 types of places [whe].
Similar to Feedbacker, wheelmap.org shows user-submitted ratings of places directly on a map and it is also visible whether a place is rated as good or bad.
With Feedbacker however, users can rate places in more ways than just specialized on one feature such as wheelchair access.

2 User Interface

The user interface consists of a main view and a view where users can edit their Feedback.

2.1 Main View

The app is controlled with a main view that allows selecting from three sub-views with specific functionality. A view to send Feedback (Figure 1), a map view (Figure 2) and a profile view which lists the user's feedback (Figure 3). This is done via interactive buttons and controls familiar to most smartphone users.

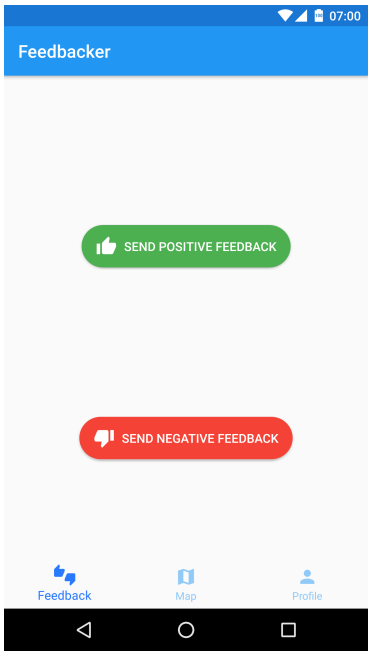


Figure 1: Feedback View Android

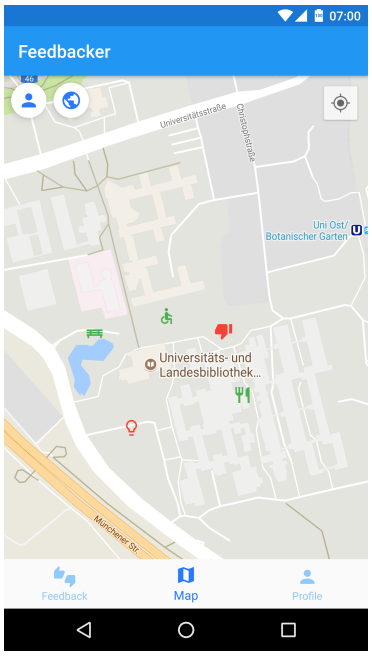


Figure 2: Map View Android

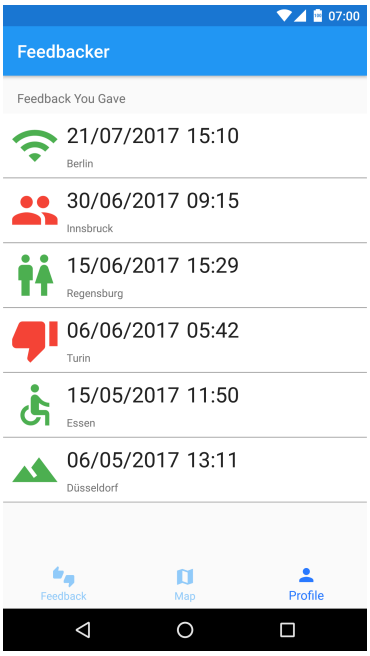


Figure 3: Profile View Android

2.1.1 Feedback View

This only displays two button to send positive and negative feedback, respectively. Clicking on either of these buttons will send a feedback of that type.

2.1.2 Map View

This view displays a map with markers at the locations where feedback has been sent. Clicking on a Marker will open a popup with time and date of the feedback as well as the descriptive text if the feedback's sender provided one. Feedbacks provided by the user itself have a button to edit this feedback.
In the top left corner there are buttons that allow the user to choose whether personal feedbacks and public feedbacks provided by other users should be displayed on the map.

2.1.3 Profile View

This view shows a list of the feedbacks provided by the user. Each feedback item has an icon based on the feedback's category as well as the time, date and city of that specific feedback.

2.2 Feedback Edit View

This view gives the user the ability to edit a feedback. This view can be reached via three ways. By sending a new feedback, via a marker on the map and from the list in the profile view.

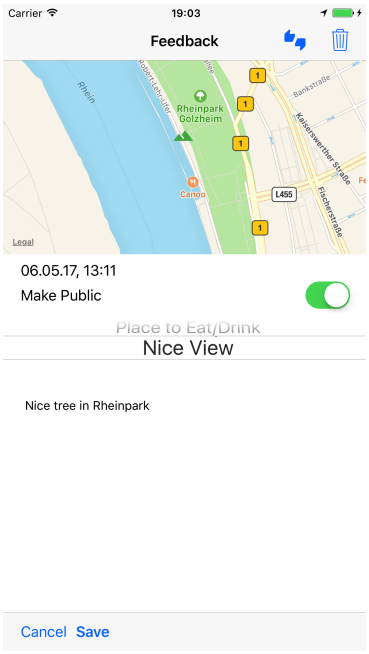


Figure 4: Feedback Edit View iOS

View Components At the top of the layout there is a map marking the location of the feedback to give the user an orientation where his feedback is placed on the map. Directly below the date and time at which the feedback has been sent is shown. These features are common to all types of feedbacks, as they are send directly when the user hits one of the send feedback buttons.
A switch can be used to mark the feedback as public, which will display that feedback for other users on the map. By default the switch is off.
The category can be chosen from a menu. Depending on the kind of feedback (positive/negative), the menu will show the corresponding options listed in Table 1 on page 1. At the bottom of the layout there is a textbox for the user to provide a descriptive text for the feedback.

Toolbar Actions At the upper toolbar the user can switch the kind of the feedback as well as delete the feedback. Both of these actions will show a popup to confirm this action as they delete the currently selected category or the whole feedback, respectively.
At the bottom there is a save and a cancel button. The save button will save all the edits the user has made to his feedback and the cancel button will discard all edits.

3 Firebase

3.1 What is Firebase?

Firebase is a platform of development tools for mobile and web apps.
It was originally founded by James Tamplin and Andrew Lee on September 1, 2011 [cru]. The first beta version was made available on April 21, 2012 [Leh]. Firebase was acquired by Google on October 21, 2014 [cru].
Firebase's products can be separated into tools for development and tools for user growth and monetization. Feedbacker does not implement any features from the second category yet. The following table lists the features provided by Firebase that are used within the Feedbacker app.

Feature	used in Feedbacker
Realtime Database	Yes
Crash Reporting	Yes
Authentication	Yes
Cloud Functions	No
Cloud Storage	No
Hosting	No
Test Lab for Android	No
Performance Monitoring	No

Table 2: Firebase Products

Most of Firebase's products offer support for Android, iOS, Web (in form of JavaScript), C++ and for Unity [pro].

3.2 Why Firebase for Feedbacker

The primary reason for using firebase as the backend service was the ease of use for multiplatform development as well as the quick set-up which does not require to develop code for the server and the database. With Firebase it was easy to develop a first prototype which could then be refined, developed further and made more secure.
Also Firebase has all features which are necessary for Feedbacker, most importantly the database. Thinking ahead, many of possible features (see section 5) can be realized using Firebase.

3.3 Authentication

Feedbacker makes use of Firebase's Anonymous Auth. This has the advantage that the user does not need to sign up with an email or similar form of authentication, skipping the signup/login step while still providing security and ensuring anonymity.

This code checks whether or not the user is currently authenticated. If he is not, he will

```
FirebaseAuth auth = FirebaseAuth.getInstance();
if (auth.getCurrentUser() == null) {
  auth.signInAnonymously();
}
```

Listing 1: User Authentication on Android

get signed in anonymously. Firebase will assign a random, unique UID for the user. Having a form of authentication is important to make the data in Firebase secure, as will be further explained in section 3.5

3.4 Data Structure

The firebase database stores the data in one JSON tree. The different sections within this JSON tree are explained in the following.

Categories All accepted categories are stored in the database to enable a security check (3.5) when writing new feedback to the database.

```
"categories" : {
  "NEG_DARK" : false,
  "NEG_DIRTY" : false,
  ...
  "POS_DISABILITY" : true,
  "POS_EAT_DRINK" : true,
  ...
}
```

Listing 2: Database Object Categories

The allowed categories are stored as keys of a dictionary in the database. As only the keys are required for the security check, the values could be arbitrary. I chose to set them as a boolean value to make clear what kind of feedback the category is representing.

Feedback The most important objects in the database are those representing a single feedback (see the following example).

```
"-KmSCGPr2UfcRHXpEPzo" : {
  "category" : "POS_EAT_DRINK",
  "city" : "Düsseldorf",
  "date" : 1497286119952,
  "details" : "Coffee at Campus Vita",
  "latitude" : 51.1921476,
  "longitude" : 6.793424,
  "positive" : true,
  "published" : false,
  "id" : "-KmSCGPr2UfcRHXpEPzo"
}
```

Listing 3: Database Object Feedback

The title of the object is a unique key, that Firebase generated before initially saving it to the database. "category" specifies the feedback and must be a valid key defined in the category section. The "date" attribute is the UNIX-Timecode of the feedback. "details" stores the optional descriptive text the user used to further specify his feedback. The attributes "latitude" and "longitude" are the coordinates of the feedback. The "positive" attribute shows whether a feedback is positive or negative with true for positive feedbacks and false for negative feedbacks. The "published" attribute determines whether or not the user published his feedback for other users.

User To allow only the creator of a feedback to edit it afterwards it is important to store the connection between each feedback and its creator in the database. The feedbacks are stored in a dedicated section in case more user data has to be sored separately in a future version of Feedbacker.

```
"Vd2dbO95K7fT1PbzFt9ZnbDkLtz2" : {
  "feedback" : {
    "-Kmg08cxhJJ-bFJPWQGZ" : "POS_SIT",
    "-Kmg1KCc81FxAX__Zg_Q" : "NEG_GENERAL",
    "-Kmg1Qwx0e-pD0WPa-2f" : "NEG_DARK",
    "-Kmg1VFcJ_J6biXJDwhc" : "POS_VIEW"
  }
}
```

Listing 4: Database Object User

The title of the section is the users unique UID, generated by Firebase during the authentication process (section 3.3). The feedback section holds keys that reference the unique IDs that are associated with each feedback this user has provided. The corresponding

value details the category of the respective feedback. This allows updating the user interface (UI) in the map view and the profile view when the category changed.

Published All published feedbacks are stored in a section of the database as well.

```
"published" : {
  "-KieONK1zQZueVvSIYIJ" : "POS_DISABILITY",
  "-KieWhBvNKZ7aZJr0YqJ" : "NEG_GRAFFITI",
  "-Klc-1KaHRvryB8LJiTn" : "POS_SIT",
  ...
}
```

Listing 5: Database Object Published

Similar to the user section, the individual feedbacks are stored with their unique ID and category, thus enabling to handle a change of category easily.

3.5 Firebase Security Rules

An important part of every app handling sensitive data, like the user's location in this case, is the security of its database. In Firebase this is accomplished using security rules. They are necessary because, in contrast to a traditional database approach, all read and write operations are initiated by the user instead of the server. Therefore, read and write access as well as validating new data need to be handled by security rules.
The security rules work in a cascading way. This means that if a user has read or write access on one object of the database, he also has access on the children of that data object.

Category Level No rules have been specified for the category section. Firebase then automatically defaults the read and write access to false, which is a reasonable choice here. In the future it might be an option to allow admins write access in order to add new categories.

Feedback Level As the app is built upon the feedbacks it is important to have them secured in the database. This is achieved by the following code.

```
3  "feedback":{
4    "$feedbackID":{
5      ".read" : "auth != null && (data.child('published').val()
         ↪ === true || root.child('users').child(auth.uid)
         ↪ .child('feedback').child($feedbackID).exists())",
6      ".write" : "auth != null && (!data.exists() ||
         ↪ root.child('users').child(auth.uid).child('feedback')
         ↪ .child($feedbackID).exists())",
7      ".validate" : "newData.hasChildren(['latitude', 'longitude',
         ↪ 'positive', 'date', 'category', 'city', 'published',
         ↪ 'details', 'id'])",
8      "latitude" : {".validate" : "newData.isNumber()"},
9      "longitude" : {".validate" : "newData.isNumber()"},
10     "positive" : {".validate" : "newData.isBoolean()"},
11     "date" : {".validate" : "newData.isNumber()"},
12     "category" : {".validate" : "newData.isString() &&
         ↪ root.child('categories/' + newData.val()).exists()"},
13     "city" : {".validate" : "newData.isString()"},
14     "published" : {".validate" : "newData.isBoolean()"},
15     "details" : {".validate" : "newData.isString()"},
16     "id" : {".validate" : "newData.isString() && newData.val()
         ↪ === $feedbackID"},
17     "$other" : {".validate" : false}
18   }
19 },
```

Listing 6: Security Rules Feedback

Line 5 handles who has read access on the Feedback. `"auth != null"` ensures that only users who are authorized (see section 3.3) have the possibility to read the data. `"data.child('published').val() === true"` grants read access if the feedback was published by another user. The rest of the expression checks if the feedback was send by the currently logged in user as `"auth.uid"` returns the uid of the user who makes the request.
The write access is handled similarly. It also checks if the user is authorized and if the request is coming from the author of feedback in case it is edited (Firebase does not distinguish between new writes and edits). For new data that is not yet written in the user's section `"!data.exists()"` allows this data to be saved. Line 7 is responsible for the sanity of new data. Specifically it checks whether a new data set has all the required fields. If it does not have all the fields, the write operation is cancelled and no data is written.
The other lines make sure only data of the correct type can be written in each field. For the category `"root.child('categories/' + newData.val()).exists()"` makes sure the category written is listed in the category section of the database.

To make sure the id saved really is the true id of the feedback `"newData.val() === $feedbackID"` is checked. As a last check `"$other" : {".validate" : false}` is making sure that all other fields are rejected and thus not writing to the database.

Published The security rules for the published section are less complicated.

```
24 "published":{
25   "$publishedID":{
26     ".read" : "auth != null",
27     ".write" : "auth != null && root.child('users').child(auth
       ↪  .uid).child('feedback').child($publishedID).exists()",
28     ".validate" : "newData.isString() &&
       ↪  root.child('categories/' + newData.val()).exists()"
29   }
30 },
```

Listing 7: Security Rules Published

As the creators of the feedbacks in the published section are fine with other users seeing these feedbacks the only condition for read access is a successful authentication. For the write access it checks if the feedbackID is also in *users/uid/feedback* to only allow authors of that feedback to make it public. The validation checks if the data is a string and is an entry in the category section.

Users Protecting the user data is also vital to ensure trust of the users.

```
24 "users":{
25   "$userID":{
26           ".read" : "auth != null && $userID === auth.uid",
27           ".write" : "auth != null  && $userID === auth.uid",
28     "feedback" : {
29       "$feedbackID" : {".validate" : "newData.isString() &&
         ↪  root.child('categories/' + newData.val()).exists()"
30       }
31     },
32     "$other" : {".validate" : false}
33   }
34 }
```

Listing 8: Security Rules User

For read and write access it is checked whether the current user is authenticated in general. Furthermore `"$userID === auth.uid"` checks if the directory the request is made in belongs to the current users, thus making sure that a user can only read and write in his section.
Similar to the published rules (3.5) the validation of new data checks if the value is an entry from the cateogry section (3.4). Like rules for the feedback section data other than a feedback object are rejected and lead to an abortion of the write operation.

3.6 Managing

Firebase offers a web-based console to manage a Firebase project. It can be reached via `console.firebase.google.com` On the main page it shows crash reports and usage numbers of the apps linked to that project. In the Authentication section the allowed forms of authentication can be set up. In case of Feedbacker this only allows anonymous auth.
In the database section the data is displayed. It also offers the possibility to export the data as JSON or importing existing data as JSON. Here there is also the section to edit the security rules (see section 3.5).
Other Firebase features (Table 2, Page 5) can also be configured from the console.

4 Multiplatform Development

One of the goals of this thesis project was to develop an app that supports both iOS and Android. This section describes implementation details specific to the respective operating systems.

4.1 Tools used & supported Versions

Android The Android App is written in Java using Android Studio as the development environment.
The minimum required Version of Android is API Level 16. This minimum Version Number determines which devices can run the App. As of July 6, 2017 this will mean that 98,6% [verb] of all Android Users can download, install and run the App.

iOS The iOS Version was developed in Xcode using Swift. The UI was developed with a storyboard which "is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens" [sto].
Feedbacker on iOS supports all devices running iOS 9 or newer. This corresponds to 97% of all App Store users [vera].

4.2 Implementation of Firebase

4.2.1 Initialization

For both platforms Firebase has to be imported as an additional library. Furthermore the *google-services* file needs to reside in the app's directory. This file can be downloaded after registering the app in the firebase console.

Android Importing Firebase on Android is achieved by adding the relevant Firebase dependencies to the gradle build file. Also the Google Play Services plugin needs to be applied.

```
dependencies{
  compile 'com.google.firebase:firebase-core:10.2.1'
  compile 'com.google.firebase:firebase-database:10.2.1'
  compile 'com.google.firebase:firebase-auth:10.2.1'
  ...
}
apply plugin: 'com.google.gms.google-services'
```

Listing 9: Gradle Dependencies

iOS iOS manages third party SDKs and dependencies with so called pods. They are configured inside a podfile and installed using `pod install` in the command line. This will create a *.xcworkspace* file which has to be used for further development.
After that was done successfully Firebase needs to be initialized inside the App Delegate with `FirebaseApp.configure()`.

4.2.2 Usage

As explained in section 3.4, Firebase saves data in a JSON tree. Each node in this tree can be addressed as a *DatabaseReference*. The following examples show how read and write operations work in Firebase. As the APIs for Android and iOS are similar only the iOS version is shown here.

Read Data Firebase reads data by attaching an observer to a *DatabaseReference*. The example below shows how this was used to mark the user's feedback on the Map (Figure 2).

```
func createPersonalMarkers(){
  let uid : String = (Auth.auth().currentUser?.uid)!
  let personalFeedbackRef : DatabaseReference =
   ↪ FirebaseHelper.ref.child("users").child(uid)
   ↪ .child("feedback")
  //Observe personal Feedback Ref for new Childs
  personalFeedbackRef.observe(.childAdded, with: { snapshot in
    let id : String = snapshot.key
    FirebaseHelper.ref.child("feedback").child(id)
     ↪ .observeSingleEvent(of: .value, with: { (snapshot)
     ↪ in
      //Create Feedback from snapshot and create Annotation
    })
  })
  //Observe personal Feedback Ref for changes
  personalFeedbackRef.observe(.childChanged, with: { snapshot in
    let id : String = snapshot.key
    FirebaseHelper.ref.child("feedback").child(id)
     ↪ .observeSingleEvent(of: .value, with: { (snapshot)
     ↪ in
      //Create Feedback from snapshot and create Annotation
    })
  })
  //Observe personal Feedback Ref for new deleted Childs
  personalFeedbackRef.observe(.childRemoved, with: {snapshot in
    let id : String = snapshot.key
    // Code for deleting the marker
  })
}
```

Listing 10: Read Data

The way the data is structured for Feedbacker (see section 3.4) the user only saves a reference to a feedback in his section of database. This means that when an object was added or changed the corresponding feedback needs to be read as well. This is also achieved by observing a *DatabaseReference* for the selected feedback. It is a one-time-only meaning it only reads the data once and then it is detaching itself from the reference.
When a user toggles to not show his personal feedback (or published feedbacks) on the map the observer on the reference is detached. This system of permanent observers and one time only observers means that the network traffic for the user is kept at a minimum.

Write/Delete Data For both apps a FirebaseHelper class was written to save, delete feedback and create new IDs for Feedback. The following explains how this was achieved.

```
static func saveFeedback(feedback : Feedback){
  let uid : String = (Auth.auth().currentUser?.uid)!
  //Save Feedback in User Section
  ref.child("users").child(uid).child("feedback")
   ↪  .child(feedback.id).setValue(feedback.category)
  //If made public save it in Published
  if(feedback.published){
    ref.child("published").child(feedback.id)
     ↪  .setValue(feedback.category)
  } else{
    ref.child("published").child(feedback.id).removeValue()
  }
  //Save the Feedback itself
  ref.child("feedback").child(feedback.id)
   ↪  .setValue(feedback.getDataAsDict())
}
```

Listing 11: Save Feedback

It is important to note that the feedback is saved in the user directory, the feedback directory, and, if it is published, in the published directory.

```
static func deleteFeedback(feedback : Feedback){
  let uid : String = (Auth.auth().currentUser?.uid)!
  ref.child("feedback").child(feedback.id).removeValue()
  ref.child("published").child(feedback.id).removeValue()
  ref.child("users").child(uid).child("feedback")
   ↪  .child(feedback.id).removeValue()
}
```

Listing 12: Delete Feedback

Similarly, references to the feedback need to be removed from all sections in the database when it is deleted.

```
static func getFeedbackId() -> String {
  return ref.child("feedback").childByAutoId().key
}
```

Listing 13: Generate Feedback Id

The Firebase `childByAutoId()` is used to create IDs for feedbacks. According to the Firebase documentation, this command "generates a new child location using a unique key" [aut]. This means that every new feedback really has a unique ID and cannot be overridden by another user's feedback.

4.3 Location

4.3.1 Location Permission

In order to get access to the user's location data, both platforms demand apps to ask for permission by the user to do so.

Android Up until Android 6.0 (API 23) Permissions were granted on installation. This led to many users not installing certain apps due to uncertainty why apps would need access to services like location.

```
<uses-permission
 ↪ android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission
 ↪ android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Listing 14: Android Manifest Location Permission

If an app wants to use certain features it has to define them in the app's manifest. As Feedbacker is targeting API Level 25 it needs to request permissions also at runtime. If the user already granted the permission nothing happens. If the user has yet to allow location access for the app a dialog is shown, prompting the user to grant location access.

```
public void onRequestPermissionsResult(int requestCode, String[]
 ↪ permissions, int[] grantResults) {
  super.onRequestPermissionsResult(requestCode, permissions,
   ↪ grantResults);
  switch (requestCode) {
    case PERMISSION_LOCATION_REQUEST_CODE: {
      if (grantResults.length > 0 &&
        grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        //Check if App has Hardware Permissions [True because of
         ↪ App Manifest]
          //then start Listening for Location Update
      } else {
        //Permission for Location was Denied
        // --> Switch To Error Fragment
      }
    }
  }
}
```

Listing 15: Location Permission Android

iOS On iOS the Location Manager is in charge of handling location permissions.

```
let locationManager = CLLocationManager()
locationManager.requestAlwaysAuthorization()
locationManager.requestWhenInUseAuthorization()
```

Listing 16: Location Permission iOS

The Location Manager is requesting access to the location of the user. If the user has not granted permission yet or has revoked location access a dialog will prompt the user to give the app access to the location. Furthermore the usage of location and the reasons for it have to be declared in the *Info.plist* file.

Information Property List	Dictionary	(17 items)
Privacy - Location Usage Description	String	Location needed for sending Feedback with Location
Privacy - Location When In Use Usage Description	String	Need to access Location to send Feedback
Privacy - Location Always Usage Description	String	Need to access Location to send Feedback

Figure 5: Location Permission flags in Info.plist

4.3.2 Listening for Location Updates

Android Android's location updates are handled by a GoogleApiClient [gAC] and the FusedLocationProviderAPI [fla].

```
mLocationRequest = new LocationRequest();
//get new Location every 5-10 Seconds
mLocationRequest.setInterval(10000);
mLocationRequest.setFastestInterval(5000);
//Use highest Accuracy Possible
mLocationRequest.setPriority(LocationRequest
 ↪  .PRIORITY_HIGH_ACCURACY);
LocationSettingsRequest.Builder builder = new
 ↪  LocationSettingsRequest.Builder()
      .addLocationRequest(mLocationRequest).setAlwaysShow(true);
builder.build();
LocationServices.FusedLocationApi
 ↪  .requestLocationUpdates(mGoogleApiClient, mLocationRequest,
 ↪  this);
```

Listing 17: Init Location Updates Android

First a location request is generated and configured (in this case with an update interval of 5-10 seconds, highest accuracy possible).

```
@Override
public void onLocationChanged(Location location) {
  mCurrentLocation = location;
  mLastUpdateTime = Calendar.getInstance();
}
```

Listing 18: Location Changed Android

This function is overridden from `com.google.android.gms.location.LocationListener` and is called every time the location changed.
If the location changed, the new location and the current time of the user's device will be saved.This information is accessed when a new feedback is created.

iOS Similar to Android first the location manager from Listing 16 gets configured as desired.

```
locationManager.delegate = self
locationManager.desiredAccuracy =
 ↪ kCLLocationAccuracyNearestTenMeters
locationManager.startUpdatingLocation()
```

Listing 19: Init Location Manager iOS

Also similar to Android a function is overridden to handle the callback when the location changed.

```
func locationManager(_ manager: CLLocationManager,
 ↪ didUpdateLocations locations: [CLLocation]) {
  let location = locations[0]
  coordinate = location.coordinate
  date = location.timestamp
}
```

Listing 20: Location Changed iOS

This function is implemented from `CLLocationManagerDelegate`. Like in Listing 18 the new location and time are saved and accessed when creating a new feedback.

4.4 Internationalization

In order to give users the best experience possible it is best practice in App development to offer the App in as many Languages as possible. Both platforms Android and iOS achieve this by storing the strings to be displayed in the user interface in special language files.
The system then automatically chooses which string to pick and display to the user.
At the moment of releasing this thesis, Feedbacker supports English and German.

Android On Android the strings are saved in *values/strings.xml* for the standard translation (english) and in *values-de/strings.xml* for german. Additional translations would be saved in a folder with the respective country code. This system also allows having different resource values like margins in layout for different regions.

```
<string name="cat_neg_dark">Dark Place</string>
<string name="cat_neg_dirty">Dirty</string>
```

Listing 21: Strings Android English

The strings are identified by a key that is common to all languages implemented.

```
<string name="cat_neg_dark">Dunkler Ort</string>
<string name="cat_neg_dirty">Dreckig</string>
```

Listing 22: Strings Android German

That is, the German strings all have the same key as the English counterparts with only the value changing according to the translation.

iOS iOS also utilizes special files to support different languages. The languages wanted have to be declared in the info section of the project settings.
This will then create **.string* files for the storyboard selected.

```
/* Class = "UITabBarItem"; title = "Map"; ObjectID =
 ↪  "cPa-gy-q4n"; */
"cPa-gy-q4n.title" = "Map";
/* Class = "UITabBarItem"; title = "Profile"; ObjectID =
 ↪  "kUv-uF-GHa"; */
"kUv-uF-GHa.title" = "Profile";
```

Listing 23: Strings iOS English

Similar to Android all UI-Strings have a common key and the value according to the language.

```
/* Class = "UITabBarItem"; title = "Map"; ObjectID =
 ↪  "cPa-gy-q4n"; */
"cPa-gy-q4n.title" = "Karte";
/* Class = "UITabBarItem"; title = "Profile"; ObjectID =
 ↪  "kUv-uF-GHa"; */
"kUv-uF-GHa.title" = "Profil";
```

Listing 24: Strings iOS German

The app will automatically display the UI with the language selected by the user in the system settings.
Localized strings required outside of the UI are saved in *localized.string* similar to the storyboard and UI relevant files they get sub level files for each language. These Strings can be accessed project-wide with `NSLocalizedString("cat_neg_dark", comment: NEG_DARK)`.

5 Future of the App

At the time of finishing this thesis, the app is functioning in the way described previously. However, there is still room for improvement both in the implementation and in the features of the app. In the following I list some possible improvements.

5.1 Implementation enhancements

Cloud Functions With the current version of the app, writing to and deleting from the database have to be done inside the app for all three database sections (see section 4.2.2). This can lead to problems if an operation fails in one of the sections. A more elegant solution would be to perform only one operation in the app and handle the event accordingly inside Firebase's Cloud Functions (Table 2, Page 5).

Error Handling At the moment not all errors are caught and handled accordingly. For example, when a user turns off the location services on his device, the Android version of the app will currently show an error screen. Ideally, a future version of the app would handle all errors in a similar way to ensure users the best possible experience.

5.2 Feature Enhancements

Pictures In order to make the feedbacks more detailed users could be given the possibility to upload a picture with their feedback. Especially for categories such as nice view or dirty places pictures could portray that feedback better than a text.
Firebase Storage would be an option to implement this.

Filtering in Maps At the moment it is possible to display the user's own feedbacks or public ones (or both). Future versions could offer more filtering options, like positive/negative or even category-specific (for example to only show places with a positive feedback with the "place to eat/drink" category when searching for a place to have dinner in an unknown city).

5.3 New Features

Social The anonymity of users is a left-over that is due to the original idea of developing an app that can be used to identify dangerous places.
A possibility might be to give the app a more social character. Social networks like Facebook, Instagram and Twitter belong to the most popular apps in the world. Concerning Feedbacker itself possible features could be commenting on feedback or following users to get notified when they send new feedback.
This could give businesses like shops and restaurants a possibility to contact people sending positive or negative feedback about their place directly in order to improve its perception.

Web Platform To visualize feedbacks outside of the app a website could be offered to allow people to display the feedbacks on a bigger screen.
This web platform could also be used by administrators to add categories.

6 Conclusion

This work documents the process of developing a multiplatform, location-based app called Feedbacker.

Development Process The development process showed that Firebase is a great tool not only for developing an app with both Android and iOS support in general, but also for achieving this quickly and efficiently.
Especially when working with database-focused apps like Feedbacker Firebase can play out its strengths. The simplicity of While there are still many things that could be improved in the future, especially regarding the user interface in the iOS version, this work showed that it is possible to develop functioning prototypes for two platforms in a relatively short amount of time.

Future The future of the app depends heavily on its success after being released in the Apple App Store and Google Play Store respectively.
The success should not only be measured by the number of installations but especially by usage numbers. As Feedbacker is a crowdsourcing app it relies on active users that send feedback regularly. It is also important that the group of active users is as big as possible. The value of the app might increase further if it is used by many people in

the same region, thus resulting in a large density of feedbacks in that region. This effect might be achieved by investing in advertising the app in selected cities and communities.

References

[aut] *FirebaseDatabase Framework Reference | Firebase*. https://firebase.google.com/docs/reference/ios/firebasedatabase/api/reference/Classes/FIRDatabaseReference#/c:objc(cs)FIRDatabaseReference(im)childByAutoId. – Accessed: 2017-07-15

[cru] *Firebase | crunchbase*. https://www.crunchbase.com/organization/firebase#/entity. – Accessed: 2017-07-10

[fla] *FusedLocationProviderApi | Google APIs for Android | Google Developers*. https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi. – Accessed: 2017-07-23

[gAC] *GoogleApiClient | Google APIs for Android | Google Developers*. https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient. – Accessed: 2017-07-23

[Leh] LEHENBAUER, Michael: *The Firebase Blog: Developers, meet Firebase!* https://firebase.googleblog.com/2012/04/developers-meet-firebase.html. – Accessed: 2017-07-10

[Meh] MEHTA, Nirav: *More ways to share your street smarts in Google Maps*. https://www.blog.google/products/maps/more-ways-to-share-your-street-smarts/. – Accessed: 2017-07-17

[pro] *Firebase*. https://firebase.google.com/products/. – Accessed: 2017-07-10

[sto] *Storyboard*. https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html

[vera] *App Store - Support - Apple Developer*. https://developer.apple.com/support/app-store/. – Accessed: 2017-07-11

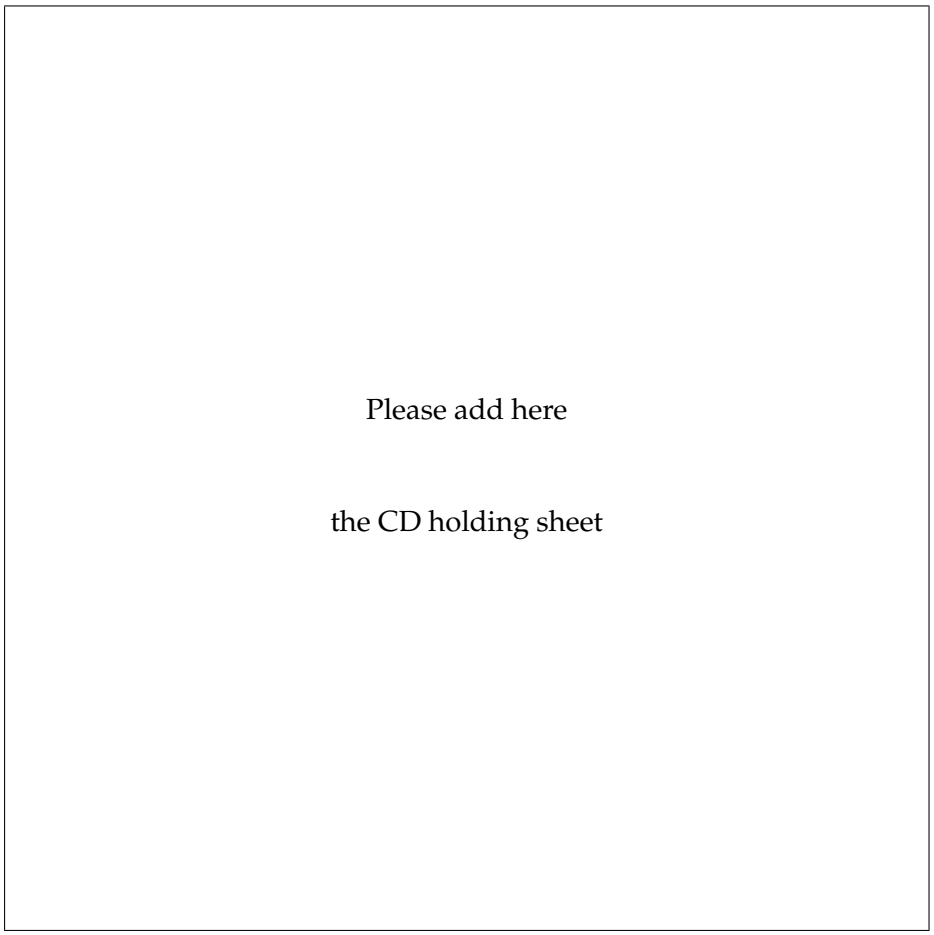
[verb] *Dashboards | Android Developers*. https://developer.android.com/about/dashboards/index.html. – Accessed: 2017-07-11

[whe] *FAQ - News von Wheelmap.org*. https://news.wheelmap.org/en/faq/. – Accessed: 2017-07-15

List of Figures

List of Tables

List of Listings



This CD contains:

- A *pdf* Version of this bachelor thesis
- All LaTeX and graphic files that have been used, as well as the corresponding scripts
- The Android Studio Project of Feedbacker
- The Xcode Project of Feedbacker
- An APK of Feedbacker to install on Android Devices

Everything on this CD can also be found at `http://tuatara.cs.uni-duesseldorf.de/borries/anhang_BA`