Internet Praktikum - HOTEL Square

Technical Documentation

Studienarbeit von Tim Burkert, Christian Hack, Marco Huber, Truong, Robert Königstein

Tag der Einreichung:

1. Gutachten: Prof. Dr. Mühlhäuser
2. Gutachten: Dipl.-Inform. Sebastian Kauschke
3. Gutachten: M.Sc. Christian Meurisch

TECHNISCHE UNIVERSITÄT DARMSTADT

Internet Praktikum - HOTEL Square
Technical Documentation

Vorgelegte Studienarbeit von Tim Burkert, Christian Hack, Marco Huber, Truong, Robert Königstein

1. Gutachten: Prof. Dr. Mühlhäuser
2. Gutachten: Dipl.-Inform. Sebastian Kauschke
3. Gutachten: M.Sc. Christian Meurisch

Tag der Einreichung:

Erklärung zur Studienarbeit

Hiermit versichere ich, die vorliegende Studienarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den August 28, 2017

_____

(Tim Burkert, Christian Hack, Marco Huber, Truong, Robert Königstein)

Liste der noch zu erledigenden Punkte

Contents

List of Figures

List of Tables

1 Introduction

The following technical documentation will give further information for the implementation of the *Internet Praktikum* from the group `HOTEL-Square`. The *Internet Praktikum* aims at getting in touch with current web technologies, which "are becoming the basic building blocks of the next generation of Internet services"[1]. Therefore, state-of-the-art protocols and technologies are used to implement a specific task in a web based context.

1.1 Task

The task of the *Internet Praktikum* in summer term 2017 is to reimplement a clone of the *Foursquare Android* app. The task includes a well defined extend of compulsory features, which the app should support. There is the possibility of extending this compulsory feature list by some bonus features.
In general, the task of the project includes the implementation of a well defined architecture, which comprises a development task for an *Android* app (frontend) and a development task including a server in *NodeJS* (backend). The backend should include a *NoSQL* database and is allowed to query some third-party API-services. In opposite, the frontend is limited to communicate with the *NodeJS*-server via some well-defined API.

The compulsory features of the app are:

- User management:
  - Create new account
  - Login
  - Delete account
  - Reset password
- Venue data:
  - Pull venue data from the backend
- Possible User Actions:
  - Venue Search (at current location and in other location)
  - Venue Search via text input and via predefined categories
  - Search results must be displayed in a list and on a map
- Venue view
  - Venue overview page showing photos and important information about the venue
  - Look at photos that where uploaded to the venue in a photo albu with swiping
  - Add comments with text and photos, like/dislike them and read other's comments
  - Check into venues to show you where there
  - Rate the venues

[1] https://www.tk.informatik.tu-darmstadt.de/de/teaching/sommersemester-2017/internet-praktikum-tk-p4/, 03.07.2017

  - Show the current top visitors of a venue
- Map
  - Show venues on the map, click on them to get to the venue overview
  - Show the user's location on the map
  - Show the user's friends location on the map
- User profile
  - Profile view
  - Profile fields containing the user name, real name, age and city
  - Picture
  - Profile editing for the user's own profile
- Friends
  - Find friends by name/user name, through comments on venues or on the map
  - Friend list showing the user's friends with links to their profile
  - Chat/messaging with the user's friends
- Venue Gamification
  - User with the must "checkins" in a venue is the "king"

Proposed bonus features for the app are:

- User management:
  - Facebook/other login
  - E-Mail confirmation
- Search Results:
  - must be filterable
- Map
  - Live update of the user's friends location
- Friends
  - Privacy measures
- Venue Gamification
  - Any gamification is a bonus

The compulsory features of the server are:

- Implementation of the server in *NodeJS*
- Usage of a *NoSQL* database for storage
- Implementation of a `REST`-ful API for the app-calls (`CRUD` for comments, ratings, pictures and ALL possible acitons)

- Hosting of the server
- Pull venue data from *Google* (or similar service)

Besides implementing all of the above mentioned compulsory features, the following table lists all bonus features, which were further on implemented:

General
Privacy means: "incognito modus" -> if applied, no one can see the user's location and vice versa
Live update of the user's friends location on the map
Filterable search results (opening hours/costs)
E-mail confirmation for registration and password reset (secured by SSL and DKIM)

Frontend
two languages: Deutsch and English
different marker colors on map depending on rating
different accuracy-strategies to optimize accuracy/battery consumption
Button to directly call the venue
Button to open the Website of a venue
Jump to the profile of your friend by clicking on his marker info window
Button to find your own Position on the map

Backend
BDD/TDD -> Chai
CD -> test/static code analysis (Lint, test code coverage)/use of docker containers for deployment
use of *bcrypt* library for secure password hashing
S3 filehosting with *min.io*
Stateless server realized with *JWT* (*JSON Web Token*)
Security measures: SSL for whole API

Table 1.1: Bonus features of HOTELsquare

1.2 General Architecture

To accomplish the given task of implementing a Foursquare clone the general architecture was prescribed in the project definition. As shown in fig. (**??**), a *NodeJS* server should provide an API for the *Android* app. The server itself shall work based on a *NoSQL* database such as *MongoDB*, which is used in the presented project. The server is allowed to fetch venue data from the *Google Places API* and deliver them to the app.

1.2.1 Dedicated Server

To fulfil the given task, it was necessary to have access to a physical or virtual server to run the web server for the backend. We chose to use an already available dedicated server where

[2] Images taken from: https://raw.githubusercontent.com/altairstudios/nodeserver/master/nodeserver-logo.png, http://www.chip.de/ii/1/5/9/3/1/1/1/6/609e955b0a179dab.jpg, https://webassets.mongodb.com/_com_assets/cms/MongoDB-Logo-5c3a7405a85675366beb3a5ec4c032348c390b3f142f5e6dddf1d78e2df5cb5c.png, http://www.ceda.cz/files/logo/google/google_2016/icon_placesapi.png, https://www.webgeoservices.com/wp-content/uploads/2017/02/Google-Places-API.jpg (24.07.2017)

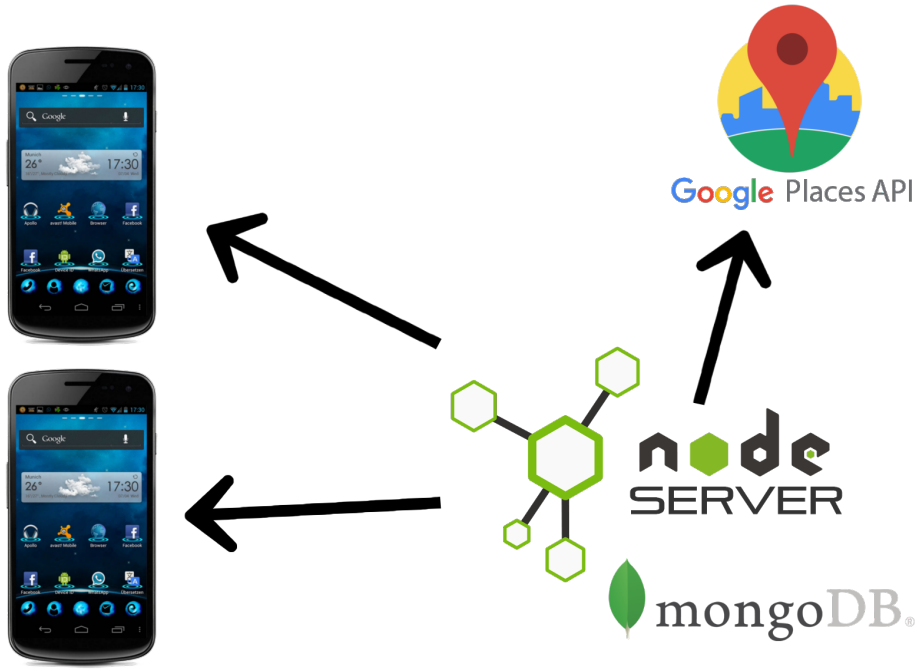

Figure 1.1: The server written in *NodeJS* with a *MongoDB* database pulls venue data from *google places API*. The server provides an API for the *Android* app.[2]

the current status of the web server could always directly be deployed to. Under https://dev.ip.stimi.ovh/, the RESTful API of the web server was available also during the development stages as CD (Continuous Deployment) was the development style of choice for the backend (for more on that, see sec. 3).
The hardware of the server is totally sufficient to handle even a medium amount of accesses to the backend services: 16GB RAM, Core™ i5-750, 2TB HDD, 100 Mbit/s network access speed.

1.2.2 *NodeJS*

NodeJS is an asynchronous, event driven open-source, cross-platform *JavaScript* runtime environment. Within *NodeJS JavaScript* code can be executed on server-side rather than embedding the scripts in the web page content and running them in the client's browser. With *NodeJS* it is possible to build scalable network applications as callbacks are fired for each connection to the server which is a much more scalable than thread-based networking approaches.[3]

1.2.3 *MongoDB*

MongoDB is one of the leading *NoSQL* databases. *NoSQL* databases provide mechanisms for storing and retrieving data without following the approach of having tabular relations like in relational databases. *MongoDB* is an open-source document-oriented database, which means that the data is stored in any kind of documents instead of a tabular model. In this case, user defined schemas are stored in *JSON*-like files. By the use of auto-sharding, it is highly scalable and complex architectures across many data centres are possible.[4]

[3] https://nodejs.org/en/about/, https://en.wikipedia.org/wiki/Node.js (24.07.2017)
[4] https://www.mongodb.com/de, (24.07.2017)

In this project, *mongoose*, an object modelling format for *MongoDB* data is used. It works like an ORM (object-relational mapping) tool making it easy to store data from an object oriented programming language into a relational database (Although *MongoDB* is a *NoSQL* database, *mongoose* facilitates the handling a lot). *mongoose* helps to access the *MongoDB* database in an object oriented manner providing simple `CRUD` methods and to create schemas for it.

1.2.4 *Android*

Android is a wide-spread OS for mobile gadgets like smartphones, tablets, smartwatches, gaming consoles, cars or tv's. The open-source software is based on the *Linux* kernel. On top, there are several software layers abstracting from the hardware as shown in fig. (1.2). Most important for developers, the *Java* API framework provides many state-of-the-art libraries for efficient high level access to most hardware features of the used gadget and a lot of predesigned software libraries for fast coding of apps based on the life cycle of the app.

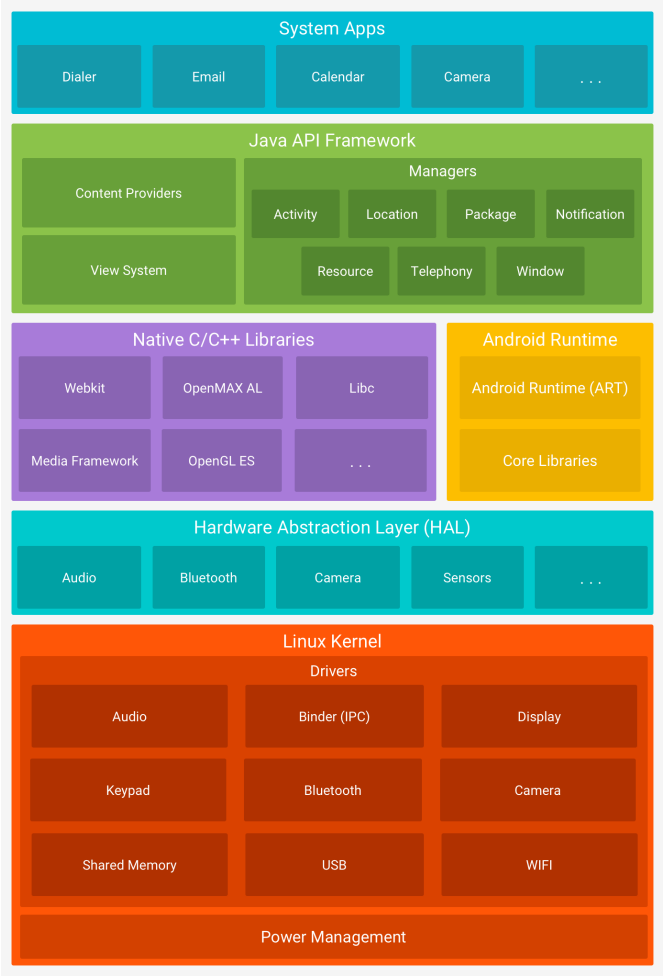

Figure 1.2: Android is based on the *Linux* kernel. Above, a hardware abstraction layer for the sensors, camera, etc., the *Android* runtime with its core libraries and some native *C++* libraries such as *OpenGL* for the graphics are implemented. Based on this, the well known *Java* API framework which provides the libraries accessible for developers helps for programming software in the high level language *Java*.[5]

[5] Image taken from: https://developer.android.com/guide/platform/images/android-stack_2x.png (24.07.2017)

2 Frontend

2.1 Venue Search

Venue Search is one of the main part of the application. When users want to search for their interest, they will experience two steps. The first is fast search and the second is deep search

2.1.1 Fast Search

When user starts the application, the fast search will be firstly displayed. It plays a role as default view of the application. It provides some name suggestions of interest which are written from static strings resource. On it five categories are predefined and each category contains three items (Figure 2.1).

- Food and Drinks: beer, cafe, veggie
- Holiday and Relaxation: beach, castle, zoo
- Services: bank, gas station, car wash
- Shops: supermarket, florist, music
- Infrastructure: airport, harbor, e-charging station

By clicking on one of the suggested name or search bar it will be redirected to Deep Search.

2.1.2 Deep Search

In order to find out venues, a query should be defined. It includes three parts, the first is keyword which can be name of interest selected in fast search or typed text in search bar in deep search, the second are filters, the last one is a page number. There are two types of filters, the location is mandatory and radius, price and open now are optional. The location can be the name of cities, countries or the current GPS value of users. When user types text in search bar, the application will suggest some names of interest that are read from local database. When the app starts for the first time, the keyword suggestions are read from static strings resource then inserted into local database. Additionally, they are also dynamically extracted from types of venues and typed text by the user which has been not located in the local database (Figure 2.2).
If the user doesn't have any input in location filter, the "near me" mode will be enabled. That means searching for all venues around the current location of the user by GPS value. It also plays a role as default search mode when redirecting from fast search to deep search. Otherwise, users have to give desired location name. However, it is not needed to give the complete meaningful location name. The application will take a suggested list of locations from the server then provides for him (Figure 2.3). The page number is the pivot for obtaining corresponding number of venues. It will be clarifyed in the next section. Deep search only processes in the following cases:

1. Submit keyword in search bar
2. Select keyword from suggested names

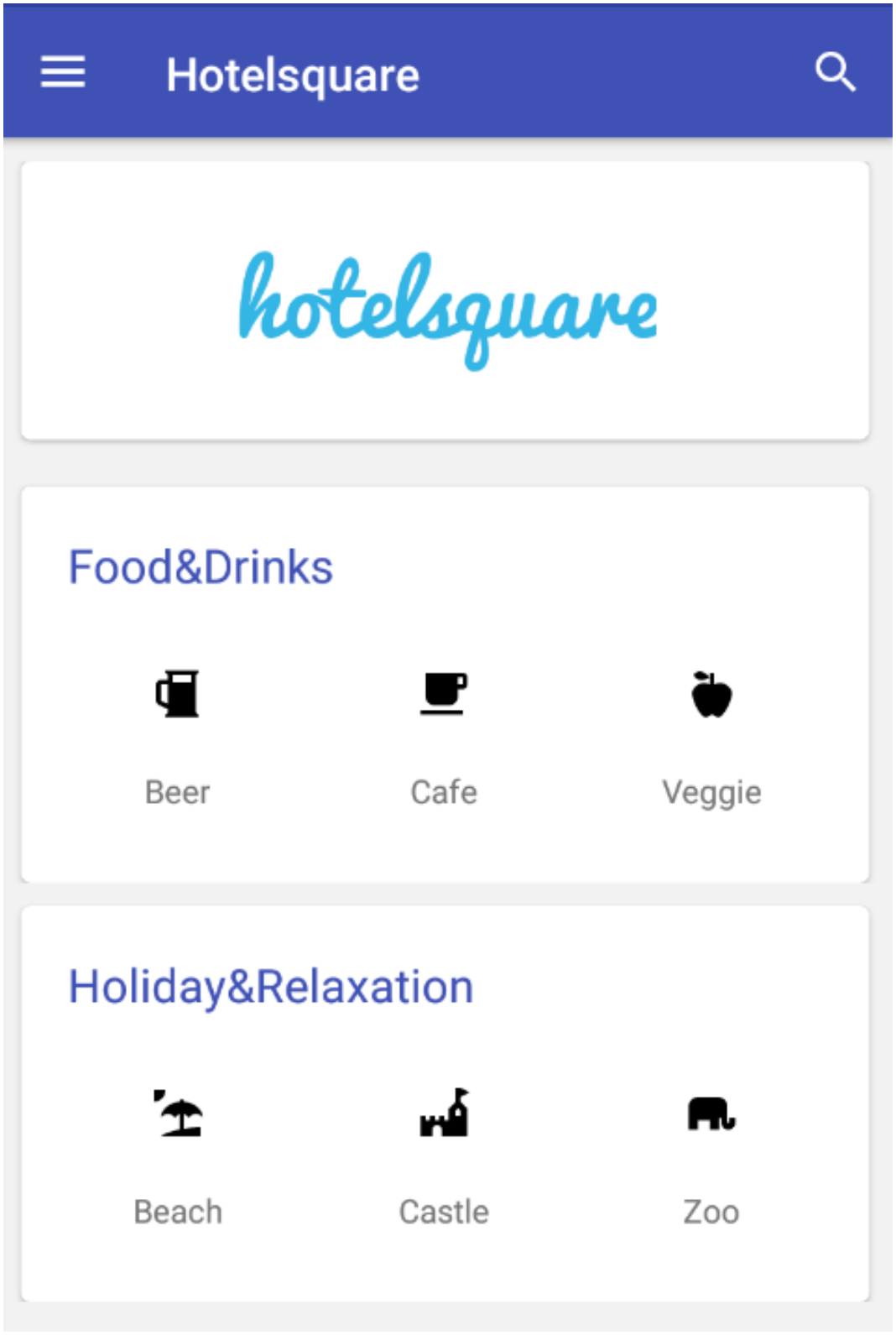

Figure 2.1: Fast Search with suggestion names of interest

3. Select item from suggested locations
4. Change radius values
5. Select and deselect price values(from 1 to 5)
6. Select and deselect open now filter

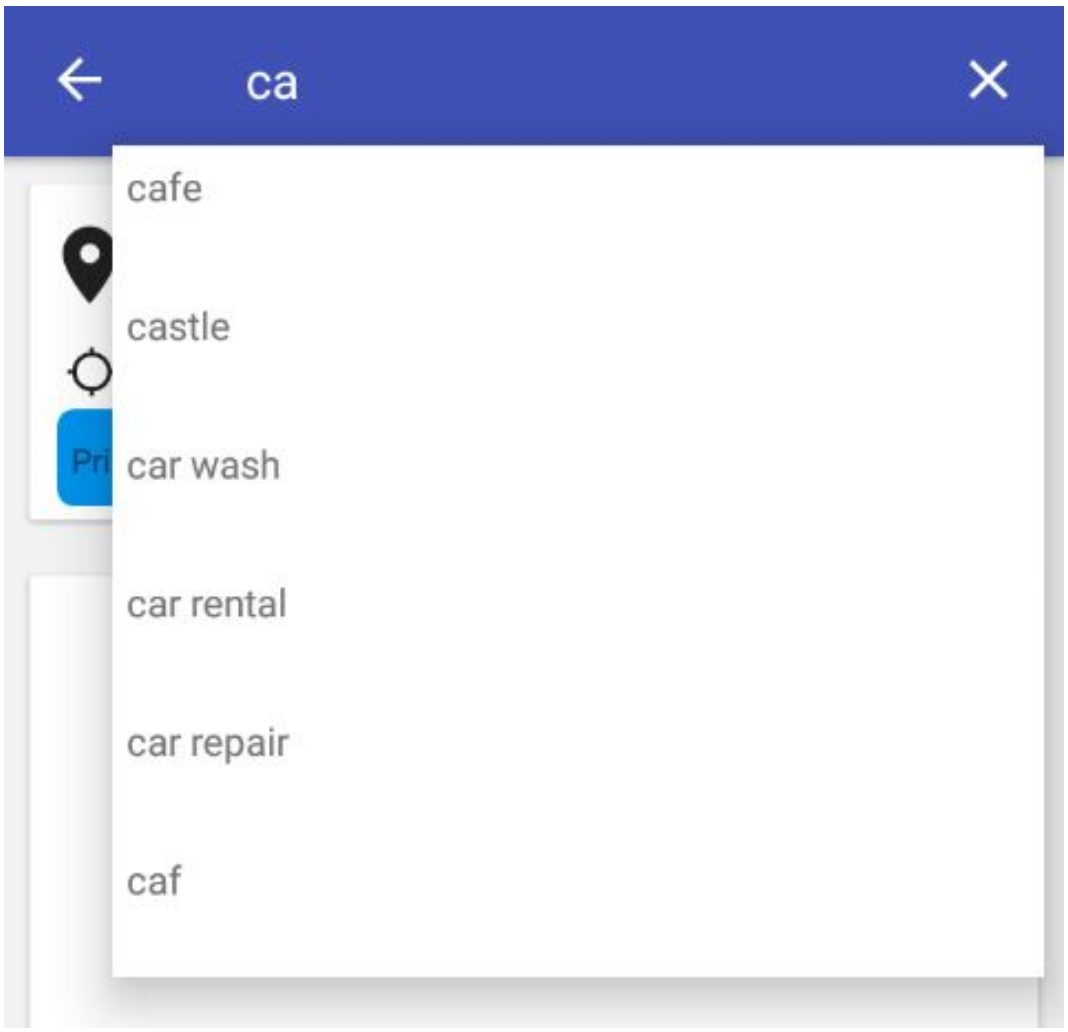

Figure 2.2: Keyword suggestions

In order to avoid sending undesired request to the server, the current keyword will be saved as last keyword query. In case 1 und 2, if the current keyword and the saved last keyword query are the same, the request will be not sent.
Two possible view are provided:

1. Venues in list: For each request are only 10 venues returned. The first reqest starts with the page number 0 to obtain the first 10 venues. As the user scrolls list until a index position which modulo 10 is equals to 9, the page number will be increased by one and the next quest with the same content query will be sent. In this case are only the basic information(venue name, rating and venue image) displayed (Figure 2.4) .
2. Venues on map: it will be described in (2.4)

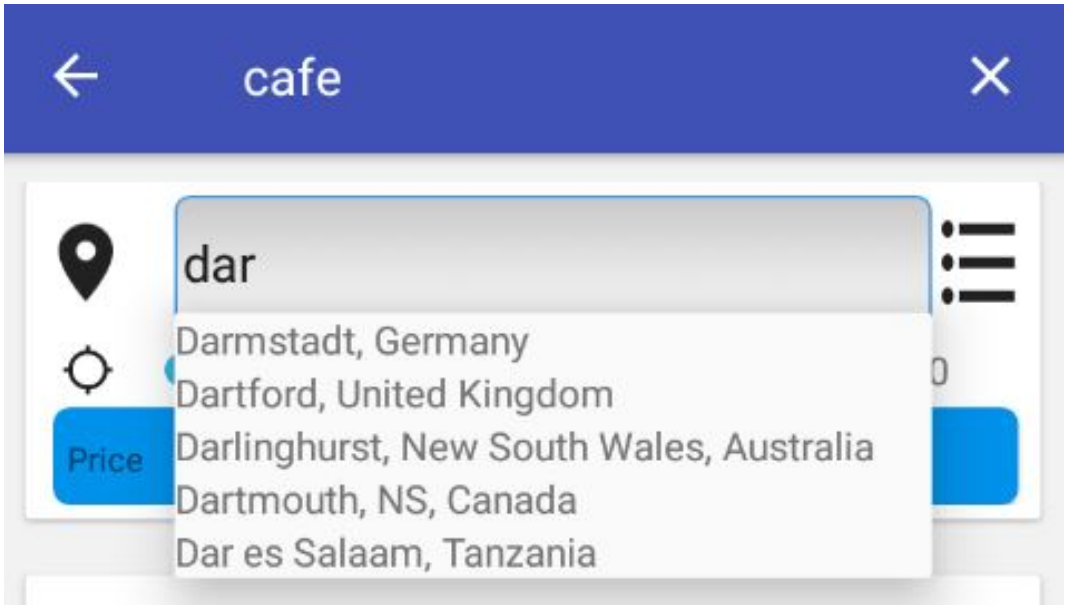

Figure 2.3: Location suggestions

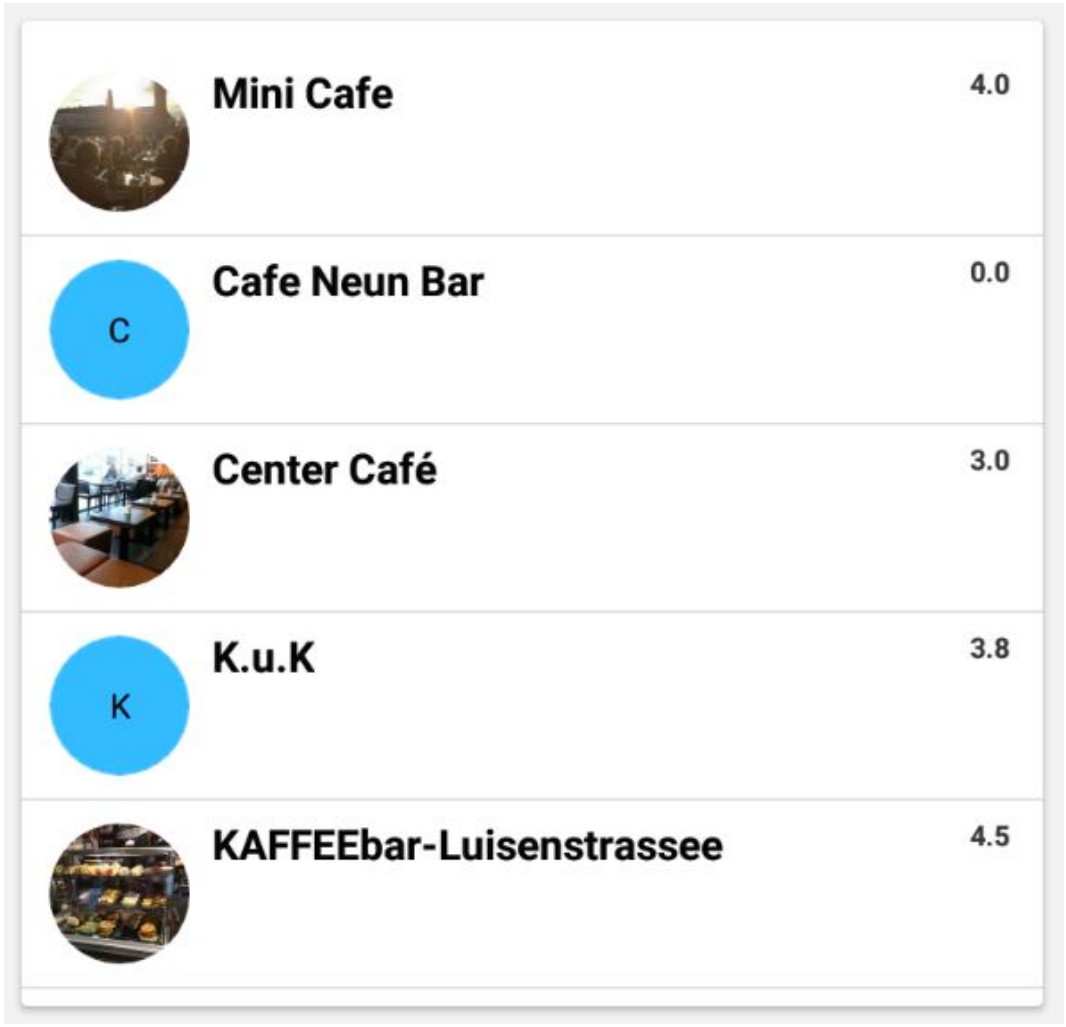

Figure 2.4: Venues in list

2.2 Location Awareness

To provide the possibility to find nearby venues and friends the application needs to be aware of the location of the user and his friends. A background service which tracks the location of the users has been implemented to achieve this. The location data is then provided to the search, the map and the server.

2.2.1 LocationService

The *LocationService* is a simple background service which is started at the start of the app and is stopped when the *MainActivity* is destoryed. The main functionality is to controll the *LocationTracker* which actually tracks the location.

2.2.2 LocationTracker

The *LocationTracker* is implemented as a *LocationListener*. The *GooglePlayService API* is used to obtain the location every interval. The interval is parameterized.
There are two different modes to guarantee a balance between power usage and accuracy. If the user is going to send a search request by starting to search for a venue, the priority of the *GoogleAPI* client is set on high accuracy, otherwise and after the search, the mode is set on balance between accuracy and power usage. Moreover there is a check if the necessary permissions are provided or not. On every change of the location, the location is send as a *LocationEvent* over the *EventBus*.

2.3 Communication via greenrobot.org/EventBus

To communicate between the service and the acticity and fragements the *greenrobot.org/EventBus* is used. There are two different events which are posted on the bus and three different cases:

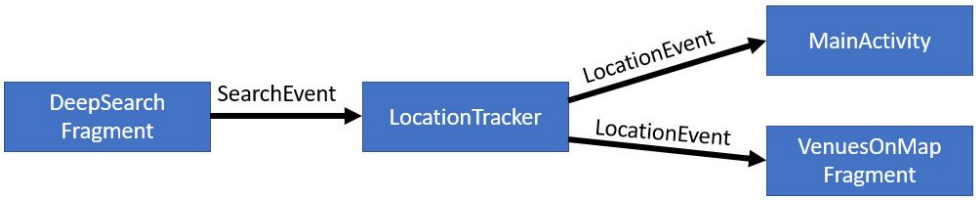

Figure 2.5: *EventBus* graph, schematically shows who sends which event type and who receives it

- The *LocationTracker* subscribes on it to receive the *SearchEvent*, which tells it to change the accuracy
- The *SearchEvent* is posted by the *DeepSearchFragement* everytime its view is created
- The *LocationTracker* posts a *LocationEvent* on the *EventBus*, which contains the Location and is received by the *MainActivity* and the *VenueOnMapFragment*

- The *MainActivity* receives the *LocationEvent*, stores it locally and sends it every 10 seconds (parameterized) to the server, to update the users position data
- The *VenueOnMapFragment* receives the *LocationEvent* to update the location of the user on the map and also the location of his friends if the user is logged in

2.4 The Map

The map is used as a function to show the user graphically where he and his nearby friends are and also the searched venues. The functionality of the map in inside the *VenuesOnMapFragment*, which is part of the *MainActivity*. The map itself is provided by *GoogleMaps*.
There are three different kinds of markers shown on the map. If the user clicks on any marker, a button shows up, which allows him to change to *GoogleMaps* and gets the route to the chosen marker.

2.4.1 Marker: Venue

The venue markers locate the search results on the map. They differ in color, depending on the rating of the venue. Those colors are with the following more or less obvious order:

- **grey**: No rating available / no rating yet
- **red**: rating between 0 and 1
- **orange**: rating between 1 and 2
- **yellow**: rating between 2 and 3
- **lime**: rating between 3 and 4
- **green**: rating between 4 and 5

If the user clicks on one of the markers an infoWindow shows up, which shows on image and tells the name, the exact rating and if it is open right now or not. With a click on the infoWindow the user is redirected to the *VenueDetails*, where he can find more additional information about the venue.

2.4.2 Marker: Friend

The friend markers locate the friends of the user, if he is logged in and has nearby friends. The marker of the friends is similar to the marker of the user, except it is green. The position of his nearby friends are updated everytime, the location of the user changes.
If the user clicks on on of his friends marker, an *infoWindow* shows up with the avatar and name and if given, also the city and age. With a click on the *infoWindow* the user gets to the profile of his friend.

2.4.3 Marker: User

The user marker locate the user on the map. If the user clicks on himself, a *infoWindow* shows up. If the user is not logged in, it shows the default *infoWindow* with the default avatar. If the user is logged in, it shows his own avatar and additonal info.
If he clicks on the *infoWindow* he is redirected to his own profile.

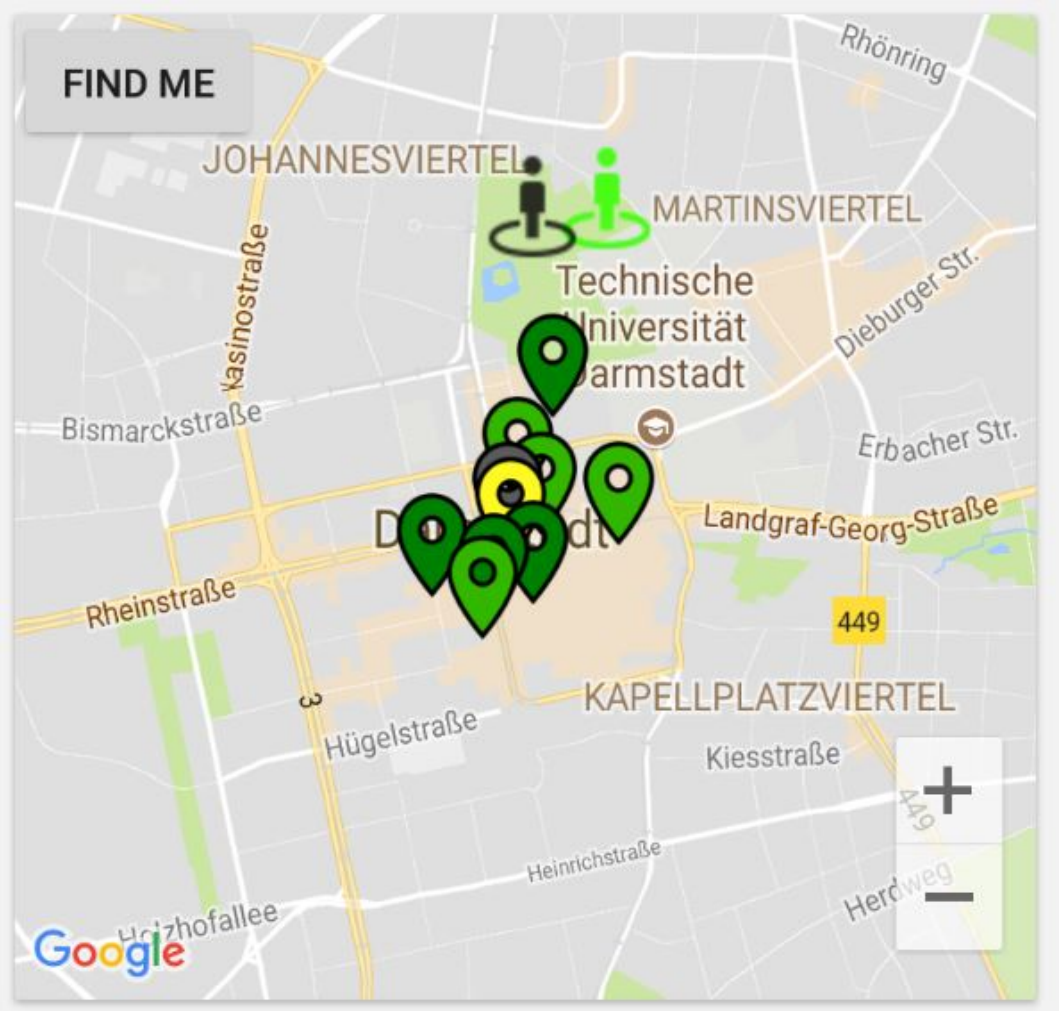

Figure 2.6: Venues, friends and user on map

2.5 Settings

The SettingsFragment mainly provides three functionalities for logged-in users.

- **Language Selection:** The user has the ability to change the language of the app. There are two different languages supported: *Deutsch* and *English*. With a simple spinner he can change the app to his preferred language. Technically the language than is stored in the local storage and the strings are changed depending on that value.
- **Incognito-Mode:** The *Incognito-Mode* provides the user more privacy by not showing his position to anyone especially his nearby friends. The mode ist selected by a CheckBox to toggle the *Incognito-Mode* The information, that the user wants to be not seen by anyone is then send to the server.
- **Delete Profile:** This button provides the user the opportunity to delete his *HOTELsquare* account. If he clicks on it, a dialog is open to make sure he knows what he is doing. If he confirm his choice, his profile is deleted, otherwise nothing happens.

The language selection is also available for user without a *HOTELsquare* account.

3 Backend

The backend of the project shall provide the app with all necessary features like user management, venue data, image handling and a lot more. It was required to implement the backend by creating a *NodeJS* server which is communicating with a *NoSQL* database of choice. For the latter, as mentioned in sec. 1.2, a *MongoDB* database was used im combination with *mongoose* for easy `CRUD` access to it.

3.1 Development Practice

To implement the given task, *Gitlab* was used to manage the git repository for both the frontend and the backend. The reason for choosing *Gitlab* was that it provides build in project management tools such as issue tracking, milestone planning and it supports several integration strategies. Most important, it is easily possible to script a pipeline for a continuous deployment (CD) strategy for the software development. As we used *Docker* containers to build an agile software delivery pipeline to deliver new backend features faster for the frontend team, this could be perfectly done with *Gitlab*.
In general, we built a three staged pipeline running after each push of code to the git repository with an automatic E-Mail notification service for broken pipelines. The steps are:

- Test: running all tests written for the backend, including a code coverage analysis and lint for statical code analysis
- Build: building the backend software after all tests succeeded
- Deploy: deploy software to a new docker container and run it on the server (just for `dev` and `master` branches)

However, development took place on individual branches for the different tasks and the integration of running software (all tests succeeded) could be done by creating a merge request for the `dev` branch, where the current state of functionality was deployed to.

Furthermore, behaviour driven development (BDD) going along with test driven development (TDD) was the development style of choice. After user stories where created, the tests for the backend code providing the required functionality were created in a behavioural style directly representing the thought of user stories. This ensured that most functionality provided to the frontend was working and the routes for accessing the server are provided in a logic manner following the actions of the user. In the next lines, some more information about the just mentioned topics are provided.

3.1.1 Continuous Integration (CI)

Continuous Integration (CI) focusses on the integration of software through validation with unit tests and possibly integration tests. This means that usually there is a test environment (in this project a `dev` space on the server) available, where the newly generated code will be build into. When pushing new code to the repository automatically regression tests are run to check if the integration into the project is working. CI is the first step to automate software development, delivery and deployment.

In this project a CD pipeline is implemented including the necessary steps for CI as mentioned in the next lines.

3.1.2 Continuous Deployment (CD)

Continuous Deployment (CD) is the next step above CI to further improve the development speed and automatically deploy the code. In contrast to continuous delivery, each successful build is directly deployed to production. This results in faster feedback. In this project a CD pipeline based on behavioural tests and build into *Docker* container images is implemented as mentioned above.

3.1.3 Test Driven Development (TDD) & Behaviour Driven Development (BDD)

Test driven development (TDD) is a development style often used in agile environments. Using TDD, the developers or testers create tests before writing the code. This ensures a high coverage of the code by the tests, as in traditional development environments, the code is often not tested exhaustively after its creation. The tests are usually separated into unit tests, integration tests, system tests and acceptance tests, which ensure trust in the software product during all stages of development.
Behaviour driven development (BDD) is another agile style of software development based on user stories. This approach was developed based on TDD. The user stories are used to create a testbench which can be automatically run throughout the whole development process. In combination with TDD, it is possible to always define exhaustive test cases for specific user actions. So the improvement comparing to "just" using TDD is that not just the single components of the software are tested in advance, but also the whole 'action flows' of the end user is stepped through and evaluated.

The testing in this project is realised with the *Chai Assertion Library* for *NodeJS*. In general, that BDD is a slightly different approach than TDD can be seen in the different style of testing instructions in the chosen library:

- BDD:

```
1  chai.should();
2
3  foo.should.be.a('string');
4  foo.should.equal('bar');
5  foo.should.have.lengthOf(3);
6  tea.should.have.property('flavors')
7  .with.lengthOf(3);
```

- TDD:

```
1  var assert = chai.assert;
2
3  assert.typeOf(foo, 'string');
4  assert.equal(foo, 'bar');
5  assert.lengthOf(foo, 3)
6  assert.property(tea, 'flavors');
7  assert.lengthOf(tea.flavors, 3);
```

However, our development was based on the BDD style testing framework of the *Chai Assertion Library*. As an example, a defined user action would be to delete himself in the app. For doing this, the user has to be authenticated. A possible test for this scenario (in the test script, the user has already been defined) looks for example like shown in lst. (3.1).[6]

```
describe('DELETE user', () => {
  it('should delete user if authenticated', (done) => {
    request(server)
      .delete('/users')
      .set('x-auth', peterToken)
      .end((err, res) => {
        res.should.have.status(200);
        User.findById(peter._doc._id, (error, user) => {
          expect(user).to.be.null;
          return done();
        });
      });
  });
});
```

Listing 3.1: Example of BDD testing for a user deleting himself from the database.

3.1.4 Docker

Docker is an open-source software, which isolates applications in virtual OS environments. This has several benefits. It enables the automation of setting up and configuring development environments which makes it easy to work together in a project. Furthermore, *Docker* makes it easy to on the one hand scale a cloud service, as the single images can be replicated as often as needed. On the other hand, containers are easy to shift, which makes the implementation less dependant on the current environment. Finally, it helps creating agile software development pipelines as the build-servers do not have to be manually configured and at the end of a CD pipeline, automatically a *Docker* image can be build.[7]

3.1.5 Lint

Lint is a software tool for statical code analysis. Its purpose is to ensure a readable layout of the code, which is also useful for defining a constant coding style in a project or company, to find not portable constructs like dependencies on special compilers and to highlight dangerous coding style like not initialized variables. It is highly configurable so that the developer can well define the behaviour of the tool. In general, *Lint* is available for most common programming languages.

In this project, *Lint* for *NodeJS* was part of the standard BDD pipeline managed in *Gitlab*, so that good coding style, **JavaDoc**, etc. was ensured. In lst. (3.2), a small extract from the rules defined in the *Lint* configuration file are shown. It can be seen that several issues concerning good coding style, like a valid *JavaDoc* documentation or correct comma spacing are listed. '-error' defines that this feature will lead to an error message when the test pipeline is run.

[6] http://chaijs.com/ (25.07.2017)
[7] https://www.docker.com/what-docker (25.07.2017)

```
rules:
  indent:
    - error
    - 4
  linebreak-style:
    - error
    - unix
  quotes:
    - error
    - single
  valid-jsdoc:
    - error
  require-jsdoc: 1
  prefer-const:
    - error
  comma-spacing: error
```

Listing 3.2: Extract from the rules of the *Lint* configuration file.

3.1.6 Code Coverage

Code coverage is a general criterion of static code analysis. There are several code properties, of which the coverage can be calculated. Below those are for example statement coverage, branch coverage, function coverage, line coverage and many more, which are often derived from the formerly mentioned tests. The statement of how much code is covered by tests does not improve the code directly, but it increases the trust in the proper functioning of it.

For this project, the coverage test was part of the above mentioned CD-pipeline, so that it could be tracked, how much of the backend code was covered by the prior implemented tests. In case of low coverage, some more tests, covering so far not considered effects could be added. Finally, the following coverage ratios could be achieved for the backend:

- statements: 86.33
- branches: 70.14
- functions: 83.15
- lines: 86.34

3.2 Server and Database

For implementing the web server providing a RESTful API to the *Android* app, *NodeJS* was the required *JavaScript* runtime environment. In sec. 1.2.2, some general remarks about *NodeJS* are mentioned. In the project, version 7.10 is used. From version 7.6 on, the `async - await` language construct was introduced to provide a much more readable callback approach than before. This helps for having compact, readable and maintainable software code. Besides many other language features, we made use of this new construct for the prior mentioned reasons. It should still be remarked that the functionality of the software does not improve by the newly

available syntax. In lst. (3.2) and lst. (3.2), a short example of two *NodeJS* statements (both logging 'Hello World' to the console) with the same functionality are shown. To illustrate the real benefit of this new syntax, a more complex example of uploading an avatar to the server is shown in lst. (3.5). It can be clearly seen that this coding style reads very much like sequential code and it supports readability a lot.

```
sleep(100).
  then(() => {
  console.log('Hallo Welt!');
}).
catch(err => {
  // ...
});
```

Listing 3.3: Old version of 'Hello World' code using callbacks.

```
try {
  await sleep(100);
  console.log('Hallo Welt!');
} catch (ex) {
  // ...
}
```

Listing 3.4: New version of 'Hello World' code using callbacks within an async function.

```
async function uploadAvatar(request, response, next) {
  let user = await User.findOne({name: request.authentication.name});
  if(user.avatar) {
    await Image.destroy(user.avatar);
  }
  user.avatar = await Image.upload(request.files.image.path, user,
     user);
  user = await user.save();
  response.json(user);
  return next();
}
```

Listing 3.5: Example of more complex function uploading an avatar of a user to the database.

3.2.1 Server structure

In general, the backend is structured into three main parts: models, routes and tests. Furthermore, the actual server file contains the API of the backend. As far as possible, functionality is clustered in the models, as this reduces the extend of code, possible locations of errors and it improves readability and maintainability of the code.
However, out of the models created with *mongoose*, as stated before, the actual *JSON*-like files for data storage are created for the *MongoDB*. In lst. (3.6) an example of how the 'Schema' of the *venue* model looks like. The models created for this project are the following:

- `chat`: The model contains the participants of the chat, which are references to the respective user objects in the database. In general the chat model would be usable for chat groups as well.
- `comments`: The model contains the author, the object which it is assigned to (could be a any model containing text or image comments), some further own comments and some more minor information. To implement a generic comment model, *mongoose* discriminators are used. With this construct, a '`kind`' parameter is added to the 'Schema'. This parameter

has to be defined as the model's discriminator key beforehand: . Afterwards, it is possible to create different model classes for the different comment types which add further functionality to the inherited generic functionality from a common base class.

- `image`: The model contains a `uuid` (unique 128Bit ID), which is used to store the image with *minio*. Furthermore, the object, to which the image is assigned (could be a comment or a venue for example), the date of creation, the author and the location are stored.
- `geocoderesult`: The `geocoderesult` model is used to store the results of the query of the *Google Places API*[8]. Besides the actual venue object, the query time and the used keyword are stored, so that old information could be updated from time to time.
- `message`: A `message` represents a message in the chat. Therefore the chat ID, the sender and a time stamp are part of the model next to the actual text.
- `searchrequest`: This model represents a search query from the app to the backend and helps to reduce the queries to the *Google Places API* and the *Foursquare API*. It contains the location, where the user is searching for a keyword, the keyword and the query time.
- `user`: A user can register by adding a display name, an e-mail address and a password. The first two parameters are used as primary keys for the database and are therefore 'unique'. To store the name of the user, a second 'name parameter' is used, the actual 'name' of the user. This is derived automatically by creating the lower case representation of the 'displayName', which makes the handling and queries easier. There are few requirements for for the password strength and the length of the user name, which are checked on server-side. In case of an error a descriptive error message is sent back.
  Besides, it is possible that a user has friends, which are users themselves and friend requests. Further important model parameters are for example the gender, the last location of the user, an incognito flag (this prevents the user from being shown in location based queries from other users; this is done on server-side) and an avatar of the user, which can be uploaded through the frontend.
- `venue`: The `venue` model contains some parameters for storing venue information received by the *Google Places API* and the *Foursquare API*. Below the *Google Places API* information are the `place_id`, `opening_hours` and a rating. From the *Foursquare API*, the `phone number`, `price range`, `tags` and `photos` are retrieved. For this project it was necessary to add further parameters for comments, which can be added by the users of the app and check-ins of the users.

```
1  const VenueSchema = new Schema({
2    name: String,
3    place_id: String,
4    reference: String,
5    photo_reference: String,
6    types: [String],
7    location: {
8      'type': {type: String, default: 'Point'},
```

[8] https://developers.google.com/places/?hl=de (28.07.2017)

```
9       coordinates: {type: [Number], default: [0, 0]}
10    },
11    images: [{
12      type: Schema.Types.ObjectId,
13      ref: 'Image'
14    }],
15    details_loaded: {
16      type: Boolean,
17      default: false
18    },
19    price: {
20      type: Number,
21      default: 0
22    },
23    foursquare_id: String,
24    opening_hours: {
25      periods: [{
26        close: [{
27          day: Number,
28          time: String
29        }],
30        open: [{
31          day: Number,
32          time: String
33        }]
34      }]
35    },
36    check_ins: [{
37      user: {
38        type: Schema.Types.ObjectId,
39        ref: 'User'
40      },
41      count: {
42        type: Number,
43        default: 0
44      },
45      last: {
46        type: Date,
47        default: Date.now()
48      }
49    }],
50    utc_offset: Number,
51    website: String,
52    vicinity: String,
53    formattedAddress: String,
54    phone_number: Number,
```

```
55      icon_url: String,
56      rating_google: Number,
57      comments: [{
58        kind: String,
59        item: {
60        type: Schema.Types.ObjectId,
61        refPath: 'comments.kind'
62      },
63        created_at: Date
64      }]
65    });
```

Listing 3.6: Example of a *mongoose* 'Schema'. The model for a venue is shown, which contains some properties which are fetched by the *Google Places API* like the `opening_hours` or the `place_id`. On top, own properties like the ckeck-ins or text or image comments of users are added.

Within the model files, the actual 'Schemas' from *mongoose* defining the data properties, are encapsulated with classes, which in most cases already contain the `CRUD`- and further basic functionality based on the model data. This makes the access for the API very easy as the main functionality does not need to be implemented several times if it is needed in different routes. Despite, the routes provide the the functionality for the backend API. They are directly connected to the API in the server file. Within the routes, either functionality from the objects from the database are called or some functionality is directly implemented. The backend contains the following routes:

- `chat`: Provides functionality to create a new chat, reply to a message, get a full conversation and get all conversations.
- `comment`: Provides methods for receiving all comments from an object, adding text or image comments and liking or disliking them.
- `friend`: Provides routes for getting a list of all friends of a user and a list of all near by friends.
- `image`: Implements methods for getting an image and its information.
- `session`: Supplies a method to post a session to a user.
- `user`: Provides `REST` access to the user model, handles the users friends and friend requests, the avatar, the user's own data and some more minor functionality.
- `venue`: Provides methods to query for venues, to get all comments from a venue and to check in a venue.

After all, it can be seen that all data related functionality is packend within the model files, wheras the routes following the behaviour of the app user like described in sec. 3.1.3.

The actual server was implemented with the restify framework, which is further described in sec. 3.3. Besides including all required models, routes and further libraries or documents, the main server file defines the API of the backend. Therefore, the above described routes are

added to the *restify* server. For logging purposes, the *bunyan* logger is set up and connected to the restify server. In lst. (3.7), the example API for all venue related routes are shown.

```
1 // Venue
2 server.get('venues/:id', venue.getVenue);
3 server.put('venues/:id/checkin', auth, venue.checkin);
4 server.get('venues/:id/comments', comment.getComments(Venue));
5 server.get('venues/:id/comments/:page', comment.getComments(Venue));
6
7 server.post('venues/:id/comments/text', auth,
     comment.textComment(Venue));
8 server.post('venues/:id/comments/image', auth,
     comment.imageComment(Venue));
```

Listing 3.7: All venue realted REST routes defining parts of the API of the server.

It can be noticed that just few routes require an authentication of the user. This shows that a large part of the API is user independent and can be used in the frontend without being logged in. For each incoming request to the server, it is automatically checked whether a authentication header is part of the request. If this is the case, a middleware, basically just a filter, checks whether the *JSON* web token is valid and otherwise returns an error status and message to the sender. If the token is valid, the user started a session and the request is directly lead through to the actual route.

In general, the implemented API is a RESTful API. This means in short that all provided routes are one of the four major HTTP methods POST (post data to the server), PUT (change data on the server), DELETE (delete data on the server) and GET (get data from the server). The naming of the routes should directly follow its purpose. In this project we followed the name conventions from www.restapitutorials.com[9].

3.3 3rd Party Libraries

To achieve having a running server providing all the necessary functionality needed for running a *Foursquare* clone app without too much development time, it was necessary to include some 3rd party libraries providing different features to the backend. In the following, a list of all used libraries and their purposes is given:

- `bunyan`: *Bunyan* is a simple *JSON* logging library for *NodeJS*. This helps both during the development and production phase, as full error messages can be seen and server accesses evaluated.[10]
- `bunyan-logstash`: Adds the logstash UDP stream for the bunyan cloud logger.[11]
- `restify-bunyan-logger`: Adds the bunyan logger to the *restify* web service framework.[12]

[9] http://www.restapitutorial.com/lessons/restfulresourcenaming.html, (28.07.2017)
[10] https://www.npmjs.com/package/bunyan (26.07.2017)
[11] https://www.npmjs.com/package/bunyan-logstash (26.07.2017)
[12] https://www.npmjs.com/package/restify-bunyan-logger (26.07.2017)

- `restify-errors`: *NodeJS* package for easy usage of *HTTP* and *REST* errors. The package provides constructors to create error objects for all common errors. [13]

- `restify`: *restify* is a *NodeJS* web service framework optimized for `RESTful` APIs. This framework already includes a basic server, which is the basis for this project. All routes are added to the server in order to make the API accessible for the app.[14]

- `bcrypt`: *bcrypt* is one of the most secure hashing libraries. In the project it is used for hashing the user's password as shown in lst. (3.8). It is possible to pass a so called 'salt-factor' to the function, which defines the amount of added characters to the password before hashing. This helps increasing the entropy of the input string.[15]

```
1    bcrypt.hash(self.password, SALT\_WORK\_FACTOR).then((hash)
        => {
2      self.password = hash;
3        return next();
4      }, (err) => {
5        return next(new Error(err));
6    });
```

Listing 3.8: *bcrypt* for hashing the users password.

- `mongoose`: As already mentioned in sec. 3, to access the *MongoDB* from the backend, *mongoose* is used. Some more information about *mongoose* is given in sec. 1.2.3. [16]

- `exif`: The *exif* package is used to extract metadata from images. The *Exif* format is used by most digital gadgets which create images and is used to store metadata like the gps coordinates or the creation time.[17]

```
1  static \_getExifInformation(path) {
2    return new Promise((resolve, reject) => {
3      new ExifImage({
4        image: path
5      }, (err, data) => {
6        if (err)
7          reject(err);
8        else
9          resolve(data);
10     });
11   });
12 }
```

Listing 3.9: Extraction of *Exif* metadata from an image.

[13] https://www.npmjs.com/package/restify-errors (26.07.2017)
[14] http://restify.com/ (28.07.2017)
[15] https://www.npmjs.com/package/bcrypt (26.07.2017)
[16] http://mongoosejs.com/ (26.07.2017)
[17] https://www.npmjs.com/package/exif (26.07.2017)

- `urlsafe-base64`: The library takes an input Buffer, which in case is the image ID. This is then enccoded in `base64` and return as a *URL*. This makes it easy to store the image using *minio*. [18]

```
1   const baseFileName = URLSafeBase64.encode(img.uuid);
```

Listing 3.10: This package enables encoding an ID created with `uuid` in `base64` as a *URL*. This name then can further be used to save the image with `minio`.

- `googleplaces`: This package is used to facilitate the access to the *Google Places API*.[19]
- `node-foursquare`: This package is a wrapper for the *Foursquare API* and therefore facilitates the handling of the *Foursquare API* requests.[20]
- `json-web-token`: *JSON* web tokens are compact, *URL*-safe *JSON* based access tokens, which can be used to verify claims. It consists of three parts: the header (defines the token type, and the encryption method), the payload (this is a *JSON* object, which describes the claims) and the signature (the signature is normed as *JSON Web Signature, JWS* in RFC 7515). All parts are `base64` encoded and separated by a point. In this project, the payload contains user information so that the sender can directly be authenticated and is known without putting its name to the body or parameter list of the request. [21]
- `lodash`: *lodash* introduces easy to use lambda functions to *JavaScript*. This makes working with arrays, numbers, objects, strings, etc. easy, fast, readable and maintainable. The library provides modular methods for iterating, manipulating and creating composite functions.[22]

```
1   this.comments = \_.reverse(\_.sortBy(this.comments,
      'created\_at'));
```

Listing 3.11: Example usage of *lodash* functionality. The shown line of code is extracted from the 'addComment' method. By using this package, in this single line all comments of the current object, where the comments are added, are sorted and the order gets reversed. It is much more readable and compact than coding this by hand.

- `minio`: *minio* is a self-hosted, distributed, free alternative to S3 storage for large data as objects. In this project it is used for saving all images which are uploaded from the users of the app or which are fetched from *Google* data. *minio* works like a hash map, which keeps access times very low.[23]
- `node-geocoder`: The *node-geocoder* is used to geocode location names to locations in latitude/longitude representation and reverse. This is necessary for example for easy testing, as exact coordinates of places are not needed to be known and for the transfer if in the app the search in another place than the current location is used. So basically, the package

[18] https://www.npmjs.com/package/urlsafe-base64 (26.07.2017)
[19] https://www.npmjs.com/package/googleplaces (26.07.2017)
[20] https://www.npmjs.com/package/node-foursquare (15.08.2017)
[21] https://www.npmjs.com/package/json-web-token (26.07.2017)
[22] https://lodash.com/ (26.07.2017)
[23] https://www.minio.io/ (26.07.2017)

for example would just return the latitude and longitude of Darmstadt if it is queried with Darmstadt as parameter.[24]

- `restify`: *restify* is a *NodeJS* web service framework for creating RESTful APIs. The framework provides a basic server, to which the self-defined routes are added.[25]
- `sharp`: This is the image scaling package used in this project. By providing and storing the fetched images in different sizes, a lot of bandwidth can be saved, as the image quality can be adapted to the use case.[26]

```
1    const buffers = await Promise.all([
2      sharp(path)
3        .resize(200, 200)
4        .max()
5        .toFormat('jpeg')
6        .toBuffer(),
7      sharp(path)
8        .resize(500, 500)
9        .max()
10       .toFormat('jpeg')
11       .toBuffer(),
12     sharp(path)
13       .resize(1920, 1080)
14       .max()
15       .toFormat('jpeg')
16       .toBuffer()
17   ]);
18   const small = buffers[0];
19   const middle = buffers[1];
20   const large = buffers[2];
```

Listing 3.12: In the above example from the backend an extract from the 'upload' method for images is shown. The image is scaled to three different sizes and those are then lateron stored in the S3 storage.

- `uuid`: With *uuid* it is possible to create a 128Bit random number as a unique ID (following RFC4122). As the ID is 128Bit large, there is nearly no chance of having two times the same ID at the same moment of time.[27]

Furthermore, during development, further libraries for testing (`chai`[28]), static code analysis (`eslint`[29]), test image generation (`js-image-generator`[30]), etc. have been used.

[24] https://www.npmjs.com/package/node-geocoder (26.07.2017)
[25] http://restify.com/ (26.07.2017)
[26] http://sharp.dimens.io/en/stable/ (26.07.2017)
[27] https://www.npmjs.com/package/uuid (26.07.2017)
[28] http://chaijs.com/ (28.07.2017)
[29] http://eslint.org/ (28.07.2017)
[30] https://www.npmjs.com/package/js-image-generator (28.07.2017)