Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Restaurant Swiper

Bachelor's Thesis

Anton Brucherseifer

antonbr@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Roland Schmid, Simon Tanner
Prof. Dr. Roger Wattenhofer

March 17, 2019

Acknowledgements

I thank my supervisors Roland and Simon for their valuable feedback and for giving me ideas that I would not have thought of. Also, I would like to thank Anja, Fabio, Daniel, Eric, David, Johanna, Matthias and Adrian who participated in the usability tests and provided creative inputs. Besides contributing many ideas and style hints, Anja illustrated the beautiful thumb icons that are now visible in the application.

Abstract

Whenever a group of people wants to eat out, a decision where to eat must be made. Depending on the group members' location, there might be many restaurants to choose from. How can hungry people find the perfect restaurant matching their appetite without getting bored by scraping the Internet?

In this thesis we built a web application that can help resolve this issue in an enjoyable way. The application finds information about the users' appetite by showing them pictures of food. By swiping left or right, the users indicate whether they like what they see. From that information, and the user's location, the application finds the most suitable restaurants in the area. Optionally, the user can set the price range, the maximal distance of the restaurant and the time when they would like to eat.

Contents

Chapter 1

Introduction

There are numerous apps and websites providing information about restaurants. However, many people still struggle with finding a place to satisfy their needs.

The problem might be, that there are too many restaurants to choose from. This may result in a lot of tedious work reading restaurant menus and reviews. Another issue is the arising discussion that may lead to a bad atmosphere at the dining table. With Restaurant Swiper, we can even overcompensate for this problem by providing a game-like way to choose the perfect restaurant.

1.1 Related Work

Some example competitors for our application are *Google Maps* [1], *TripAdvisor* [2] and *Yelp* [3]. These systems offer information on many other aspects than restaurants, however, we will focus on restaurants specifically in the following. All of them offer a website and smartphone apps to make their database easily accessible for everyone.

They provide restaurant information such as *price range*, *food category*, *location*, *business hours*, *client reviews* and *ratings*. The *price range* is usually categorized by three or four symbolic values, rather than providing a numerical price range. For instance, on Yelp, a restaurant has one of the following price tags: \$, \$\$, \$\$\$ or \$\$\$\$. A restaurant's *food category* is indicated by a set of "categories" (Yelp) or "cuisines" (TripAdvisor). Those well-known information providers usually show the *location* in form of a map or simply a number that indicates the distance or travel time from the current user location. Google Maps, TripAdvisor and Yelp offer client *reviews* and *ratings* to inform future clients about the quality of a restaurant. The *ratings* of a restaurant are usually shown as an average value. Yelp and TripAdvisor provide methods to sort a set of restaurants by *rank*, which is usually composed of the average rating and the

[1] https://www.google.com/maps [Accessed 25-February-2019]
[2] https://www.yelp.com/ [Accessed 25-February-2019]
[3] https://www.tripadvisor.ch/ [Accessed 25-February-2019]

number of reviews. Furthermore, users can look at pictures of the interior, the food and the staff.

One application that has a lot in common with Restaurant Swiper is *Foodguide* [4]. In spite of having a very similar input method for the user's food preferences, however, their general idea differs from ours. Like our application, Foodguide shows pictures of food such that the users indicate whether they like the meal. The dishes that are shown in the pictures are at the same time the recommendations that Foodguide suggests to their users. Additionally, users have the possibility to see a list of their liked meals. Since the pictures were uploaded and carefully tagged by other users, they contain information on the restaurant where they were taken.

After this introduction, we present a brief overview of this thesis. In Chapter 2, we explain where our application gets all its restaurant data from. This should give a general idea of the opportunities and limitations for our thesis. We describe our approaches on how we tackled some fundamental issues in Chapter 3. In that chapter, we present the overall workflow of our application and provide a deep insight on our recommendation technique. Chapter 4 gives an overview of the technologies that we used to implement the application. An evaluation of the final product is provided in Chapter 5. We will conclude the thesis in Chapter 6 while presenting some personal insights and suggesting possible improvements.

[4] `https://thefoodguide.de/get-the-app-en/` [updated on 11-February-2019]

Chapter 2

Background

2.1 The Yelp Fusion API

All of the three big restaurant databases Google Maps, TripAdvisor and Yelp provide APIs that enable other apps to access those databases. Therefore, we did not have to put together our own database to serve our users with information on restaurants. Using an API also enables our application to be used all over the world instead of just a limited area.

Because of these APIs' legal and technological restrictions, one cannot combine all three of them to create the ultimate meta API. Therefore, we had to limit ourselves to one of them. We chose the Yelp Fusion API since they provide an easy-to-use interface that meets our requirements perfectly.

Yelp is concerned with all kinds of businesses. For our app, we are only interested in businesses with the category `restaurants`. There exist 312 subcategories of restaurants [1], which are of great relevance for this thesis. Our application mainly uses the `businesses/search` endpoint of the Yelp Fusion API. This is the endpoint that serves a list of businesses containing most of their information. To retrieve a list of restaurants, one sends a HTTP GET request to `https://api.yelp.com/v3/businesses/search?`*`query`*. The *query* string is a list of parameters separated by an ampersand (&). It is mandatory to let the API know of a location. For this, one can either use the parameter `location`, or the parameters `latitude` and `longitude`. The `location` parameter takes an address, a city or even a train station as an input, while `latitude` and `longitude` require the exact coordinates as decimal numbers. Additionally, users of the API can add parameters to customize the output, such as:

- `radius`: maximal distance to the restaurant in meters (Yelp's upper limit is 40000.)
- `categories`: list of categories, of which at least one must be served in each restaurant

- `limit`: maximal number of restaurants to be returned (Yelp's upper limit is 50.)
- `sort_by`: order of the list. possibilities: `best_match`, `rating`, `review_count` or `distance`
- `price`: a subset of {\$, \$\$, \$\$\$, \$\$\$\$}
- `open_now`: if true, only return restaurants that are currently opened
- `open_at`: only return restaurants that are opened at the specified time (in Unix time)

The response is a JSON object containing the total number of businesses found, the list of businesses and the region that was searched. Each business contains multiple attribute fields including *rating*, *price*, *set of categories*, *name*, *URL to its Yelp page*, *coordinates*, *image URL* and *distance*.

The following GET request

```
https://api.yelp.com/v3/businesses/search?
        location=zurich stadelhofen&
        categories=restaurants&
        sort_by=distance&
        limit=5
```

retrieves the closest 5 restaurants to the train station “Zürich Stadelhofen”.

Chapter 3

Methods

We will now illustrate the methods that were used for solving different issues. Starting with the usability tests, we will give a general intuition, why we used paper prototypes to design the user interface. In the next section, we explain the origins of the some data (e.g. food pictures) that had to be fetched manually. The Sections 3.3 and 3.4 clarify some fundamental concepts of our application. Finally, we will present our algorithms for presenting appropriate food images and restaurant recommendations.

3.1 Usability Tests

One widely used method in the user-centered design process is *paper prototyping*. It offers the possibility to test a web application, before any code is written. This helps creators to eliminate elementary problems in the very early design process. Furthermore, it allows for quick changes and retesting the product.

For this thesis, we created mock-ups of the application before writing any code. The goals of the usability test with the paper prototypes are:

- clearing uncertainties that rose up in the process of creating the paper prototypes
- getting to know more issues in the overall design

It is important that the users think out loud so that we get an insight of how well they understand what we are trying to achieve. To animate our testing person to talk, we can ask questions like “What do you think you can do from this page?” or “What do you think happens when you click on that button?”

3.2 Data Gathering

Aside from restaurant data, our application also requires many pictures of food to present to our users as well as some similarity measurement to compare two restaurant categories. For both of these features, we got the necessary data from the Yelp Fusion API. For the pictures, we only store their URLs which are later embedded in the swipe page, so clients download them directly from Yelp.

3.2.1 Food Pictures

Our application depends on a large set of food pictures including information about what exactly each picture shows. The solution here was to make many requests to the Yelp Fusion API, since they offer a set of pictures for each of their restaurants. These pictures may have been uploaded by guests or owners of the restaurant. Since we know the restaurant in which the picture was taken, each picture comes with a set of Yelp categories. The only problem left with the pictures is that they do not necessarily show food. Some pictures show the interior, the staff or the menu of the restaurant. For our purpose, the pictures should show food only. Additionally, each of them has to clearly represent the set of categories, that is associated with it. We therefore had to remove the unsuitable pictures manually.

3.2.2 Similarity Between Restaurant Categories

Measuring the similarity of two restaurant categories is used for different purposes:

- delivering food pictures (If we have not stored any pictures of a certain category, we show a similar one.)
- adding similar restaurants to a Yelp API query to obtain a broader set of restaurants
- recommending restaurants that serve similar categories than the ones the user liked

To enable this, we collected data about many restaurants and inferred a similarity measurement from their categories. We say two categories are similar, if there exist many restaurants that serve both of these categories.

Definition 3.1 (similarity value)**.** We define the *similarity* value of two categories cat_1 and cat_2 as

$$similarity(cat_1, cat_2) = 2\frac{|R(cat_1) \cap R(cat_2)|}{|R(cat_1)| + |R(cat_2)|}, \tag{3.1}$$

where $R(cat)$ denotes the set of restaurants that serve the category cat.

3.3 Grouping

Since many people like to have company at the dining table, Restaurant Swiper's main purpose is giving recommendations to groups. To that end, we have implemented a mechanism that enables users to create and join groups. When a group of people decides to eat together, they have to agree on a group name to let the application know, that they belong together. This name should currently not be in use by any other group.

The member that creates the group is considered the group's *master*. All settings for the group, including the location, come from its master. Consequently, the master has the responsibility to submit the correct settings for the group.

The application provides a separate page to create and join groups. Alternatively, group members can share their URL to invite further users to the group. Whenever Restaurant Swiper is opened with a group URL, the user will be added to that group. If it was an invalid URL (i.e. the group name does not exist), an alert is shown, where the user has the chance to create a group with that name.

At what point should the application show the recommendations to the group? For single users, the recommendation page is shown as soon as they have swiped all their pictures. However, in the group case, the application does not know whether all users are done, because new members can join as long as the group exists. Our solution is to leave this issue to the user resp. to the group's master. As soon as a user has finished the swiping procedure, the application indicates that the recommendation is not yet ready. At this point, the master has the possibility to *lock* the group, meaning that no group member can swipe anymore. If the master locks the group while some members have not yet finished swiping, an alert will be shown. The master can then either wait for those members or block them from submitting additional votes. Only if the group is *locked*, the recommended restaurants are shown. After a group has eaten, the group gets deleted on our server and the group name becomes available again. If the time to eat is set, the session will be deleted three hours after the specified time.

3.4 Settings

Users can manually set some parameters to narrow the set of restaurants that they are interested in. The settings of a group can only be edited by its master. For the group's location, the application simply uses the location of the group's master. There exists a separate page on the application, where users can change the price range, the maximal distance and the time to eat. Furthermore, users can check and edit their location.

To optimize compatibility with the Yelp Fusion API, our application adopts Yelp's set of price values {\$, \$\$, \$\$\$, \$\$\$\$}. For each of them, the users can specify whether they want to include it or not. If none are selected, all price levels are assumed adequate.

The *maximal distance* is a number between 0 and 40000 meters describing the upper bound of the distance the users are willing to travel.

Additionally, users can specify the time and date, when they would like to eat. The set of possible restaurants that come into consideration depends heavily on that time, as the application only considers restaurants that are open at the specified time. If the *time to eat* is not set, the application assumes that the users would like to eat now.

Along with those settings, a user can examine the exact *location*, that the application knows. Besides that, the users then can overwrite this location by specifying a string to be passed to Yelp. If they would like to use the automatic locating method again, they can also indicate this here.

3.5 Serving Food Pictures and Recommendations

In this section, we describe what it takes for our application to detect the users' preferences and to compose a list of restaurants. To get optimal user feedback, the application has to decide which pictures the user should see. The selection of images depends on the possible restaurants nearby and what the users and their group members have voted previously. The raw data, that is obtained from the user's swipes is preprocessed in the next step. Lastly, the application assembles a small list of restaurants which will be shown to the user.

3.5.1 Step 0: Choice of the Food Pictures

As mentioned in Section 3.2, each picture is associated with a set of categories. Therefore, it is mostly a question of *which categories* should be shown to the user. The following algorithm selects a picture given a category: First, it searches the database for pictures that contain the desired category and that have not yet been shown to the user. If there are no suitable pictures, the algorithm will look for similar categories while lifting the similarity threshold until it finds some pictures. As soon as the algorithm has a set of pictures, it randomly selects one of them and sends it to the user.

The Initial Pictures

When the application is first opened, it does not know much about the user's food preferences. The application might know some of the user's preferences from the settings, which are stored locally. However there are no information about the user's appetite, since we consider this information volatile. To get to know the user's surroundings, the application does an initial request to the `businesses/search` endpoint of the Yelp Fusion API. This request already contains some filtering. Since Yelp does not only consider restaurants, but businesses in general, we have to tell the API to return restaurants only. All attributes from the settings page (see Section 3.4) are applied as filters for the initial Yelp request. We also set the *maximal number of restaurants to be returned* to 50 which is the maximum we can get out of one Yelp request. On top of that, we set the *sorting order* to `distance`, so we can measure the restaurant density of the user's neighbourhood. Depending on the preferences that the user has submitted in the settings, it might happen that there are no restaurants that meet the requirements. For instance, if the time to eat is at night, there might not be any open restaurants nearby. In that case, the user will see an alert saying that no restaurants were found and to check the settings.

From exploring the user's surroundings, the application can infer *what categories* the user might be interested in and *how much* swiping must be done. To be able to make these measurements, we introduce the *closeness factor f*.

Definition 3.2 (closeness factor)**.** The closeness factor is a number $f \in [0, 1]$, that describes how likely it is for a specific user to go to a restaurant, while only considering the distance between user and restaurant (d_{rest}) and the user's preferred travel distance (d_{pref}).

$$f(d_{rest}, d_{pref}) = \begin{cases} 1 - \frac{d_{rest}}{\min(2d_{pref}, 40000)}, & \text{for } 0 \leq d_{rest} \leq \min(2d_{pref}, 40000) \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

All distances are measured in meters. If no distance was specified in the settings, d_{pref} is set to 5000.

Let *Pics* denote the set of pictures that a user swipes. At the point of checking the user's neighbourhood, the application decides exactly how many pictures the user should swipe. This number $|Pics|$ depends on the *restaurant density* and the *restaurant diversity.* The *restaurant density* of a location is the average of the nearest 50 restaurants' closeness factors. The *restaurant diversity* is the number of categories that occur in those restaurants divided by the total number of categories.

The application is looking for $|Pics|/2$ categories of which the user should see pictures. These categories are randomly chosen with a non-uniform probability distribution $P_{init}(cat)$. On one hand, the probability of category *cat* to be chosen depends on how often it occurs in close restaurants. If the user is a member of a group, categories that have been seen by other group members are less likely to occur in one's own set of initial pictures.

Further Pictures

From the votes of the initial pictures, the application has some knowledge about the user's appetite. Based on the previous swipes, we use another probability distribution $P_{furth}(cat)$ to select further categories to be shown. For categories that the user has liked, P_{furth} increases, while it decreases for categories that the user has disliked. If the algorithm detects that the user has disliked most pictures, it will show some more categories that the user has not yet seen. These categories are then determined by the same algorithm that finds the initial pictures.

3.5.2 Step 1: Data Preprocessing

Input

After a user resp. a group has finished swiping, the application has some raw data that represents their appetite. For each member of the group, the application stores two maps: $likes : categories \to \mathbb{N}$ and $occurrences : categories \to \mathbb{N}$. The number $likes(cat)$ states how many times a user liked category cat, while $occurrences(cat)$ is the total number of times, the user saw the category cat.

Output

The preprocessed data consists of one map per member L_p.

Definition 3.3 (preprocessed category preferences)**.**

$$L_p(cat) = \frac{likes(cat)}{occurrences(cat)} \tag{3.3}$$

It describes the fraction of times, that a user liked the category cat.

3.5.3 Step 2: Recommendation

In the following, we describe how the application assembles a list of restaurants, given the preprocessed data and the settings of the user. As a quick reminder, the settings define the location context as well as the price range and the time, at which the restaurant should be open.

The procedure goes as follows: Initially, the application sends multiple requests with varying parameters to the Yelp Fusion API. We then have a set of possible restaurants, about which we can make some measurements of how well each restaurant matches the users. From that, the recommendation engine outputs three restaurants for the user: the best, the cheapest and the nearest.

Finding Possible Restaurants

To begin with, we require a set of restaurants R_{pot}, that potentially interest our users. We want R_{pot} to be as large as possible, meaning that some of the restaurants might not fulfill all of the user's requirements on price and distance. The number of restaurants $|R_{pot}|$ is highly dependent on the location, radius, time of the day and the diversity of the category preferences. In densely populated areas this number can go up to 100 - 200 restaurants.

In order to come up with this list of restaurants, we make several requests to the Yelp Fusion API. The fixed parameters include `location`, `open_now` resp. `open_at`, `limit`, `radius` and `price`. The location is sent as a string or as coordinates, depending on the input method that the user chose (Section 4.1.2). To get the maximum out of each request, we set the `limit` to 50. The value for `radius` is taken directly from the user input in the settings page. It is to point out that the Yelp Fusion API uses this value as a suggestion only and the actual search radius depends on the density of the area [2]. For the parameter `price`, we include the neighboring price levels as well. For instance, if a user has selected the price levels {\$, \$\$}, we would like to get restaurants having the price tag \$, \$\$ or \$\$\$.

The variations in the parameter values take place in the parameters `categories` and `sort_by`. For each fitting category we make two requests to Yelp: one is sorted by `rating` and one by `distance`. At this point we make use of the previously defined similarity measurement, meaning that we do not only accept exactly the *fitting* categories, but also similar ones. Hence, we actually make Yelp requests for each *class* of similar categories.

Evaluate Performance for Each Restaurant

After we have obtained a sizable list of potential restaurants R_{pot}, we would like to have some kind of measurement, how well they fit to our users. We will now explain how we evaluate the fitting values for the contexts *category preferences*, *distance*, *price range* and *quality*. We measure the fit value for *category preferences* for each user separately, while the others are measured once for the entire group.

To get an idea of how well a restaurant matches the *category preferences* of a user m, we introduce the function fit_c^m. This function maps the user's preprocessed category preferences L_p^m and the category set of a restaurant C_r to a number between 0 and 1.

$$fit_c^m(C_r, L_p^m) = \max_{c_r \in C_r, c_p \in L_p^m} \left(L_p^m(c_p) \cdot \sqrt[5]{similarity(c_r, c_p)} \right) \tag{3.4}$$

i.e. the maximal product of the preprocessed category value $L_p^m(c_p)$ and the similarity between a restaurant category c_r and a user-liked category c_p. To aggregate the numbers fit_c^m of the individual group members m to one single value fit_c, we simply use the minimum:

$$fit_c(C_r, L_p) = \min_{m \in group} (fit_c^m(C_r, L_p^m)) \tag{3.5}$$

Another restaurant property that is crucial to the user is the restaurant's location and specifically its *distance*. For this, we use the closeness factor (see Definition 3.2). This function will always favor closer restaurants, even inside the user's preferred radius d_p. To prevent our algorithm from laying too much weight on the restaurant's location, we set the fit value fit_d to be

$$fit_d(d_r, d_p) = \big(f(d_r, d_p)\big)^{0.3} \tag{3.6}$$

where f is the closeness factor. An alternative way to evaluate the distance would be to fully accept all restaurants inside the radius d_p and only penalize restaurants that are too far away. However, we decided against this, because many users prefer short travelling times.

For users, who have some preferences about the *price* of the restaurant, the recommendation is likely to contain only restaurants inside their desired price range. However there might be restaurants that match all their other preferences perfectly, but are a bit more expensive or a bit cheaper. For that case, we introduce the function fit_p, that computes the price-fit value. The arguments of fit_p are the set of price values accepted by the user $P_p \subset \{1, 2, 3, 4\}$ and the price value of the restaurant $p_r \in \{1, 2, 3, 4\}$.

$$fit_p(p_r, P_p) = 1 - \min_{p_p \in P_p} \frac{abs(p_p - p_r)}{4} \tag{3.7}$$

For computing the fit value for the *quality* context fit_r, we simply normalize the rating r_r of a restaurant:

$$fit_r(r_r) = \frac{r_r}{5} \tag{3.8}$$

The final performance of restaurant r and preferences p is then composed of the product of computed values:

$$fit(r, p) = fit_c(C_r, L_p) \cdot fit_d(d_r, d_p) \cdot fit_p(p_r, P_p) \cdot fit_r(r_r) \tag{3.9}$$

Output

After we have carefully evaluated each restaurant, we would like to have one distinct restaurant for each of the following roles: *best*, *cheapest* and *nearest*. These are the three restaurants that will be shown to the users. Let R_{fit} denote the list of all restaurants sorted by the performance value $fit(r, p)$ in descending order The *best* is simply the first element of R_{fit}. The restaurant with the role *cheapest* is not directly the cheapest. This restaurant is intended to have the cheapest price category that was selected in the settings. If the algorithm cannot find any restaurants with that price category, it will search the next higher price range until it finds a restaurant. If there are multiple restaurants with the most favorable price category, the algorithm will select the one with the highest $fit(r, p)$. The *nearest* restaurant is simply the nearest of the top quarter of the list R_{fit}.

Chapter 4

Implementation

Restaurant Swiper consists of multiple programs. The client software is a web application, that is powered by the Ionic Framework. For hosting this web application we use the built-in Node.js server. To handle server-side tasks such as computing recommendations and building groups, we have implemented a second server in Python. Figure 4.1 shows an overview of the most important classes of the client and the Python server.

4.1 Client

The web interface is built with the Ionic Framework which is a cross-platform toolkit for building mobile and desktop apps [3]. To program Ionic applications the languages TypeScript, HTML and CSS are used. Ionic applications can run on a variety of platforms such as browsers, Android, iOS and others. Since our application is not dependent on any native device features, it can be used with any modern web browser.

Our application consists of five pages that utilize two services or *providers* [4], as they are called in Ionic 3 [1]. The purpose of using services is to separate the GUI from the data access to enable more modular software development. Their tasks are to get the user's location (Section 4.1.2) and to communicate to the Python server (Section 4.1.3).

Next to the providers that have been implemented by us, our application uses some built-in Ionic features. One example is *Ionic Storage*. It is used for keeping information stored on the user's device. This offers the possibility for our users to continue where they left off, in case the application gets closed or they reload the page for some reason. Additionally, it is used for communication between pages and services, ensuring consistent memory across different TypeScript classes.

[1]Restaurant Swiper is built on Ionic 3. The term "service" is used in Ionic 4 [5], which was released during the development of Restaurant Swiper.

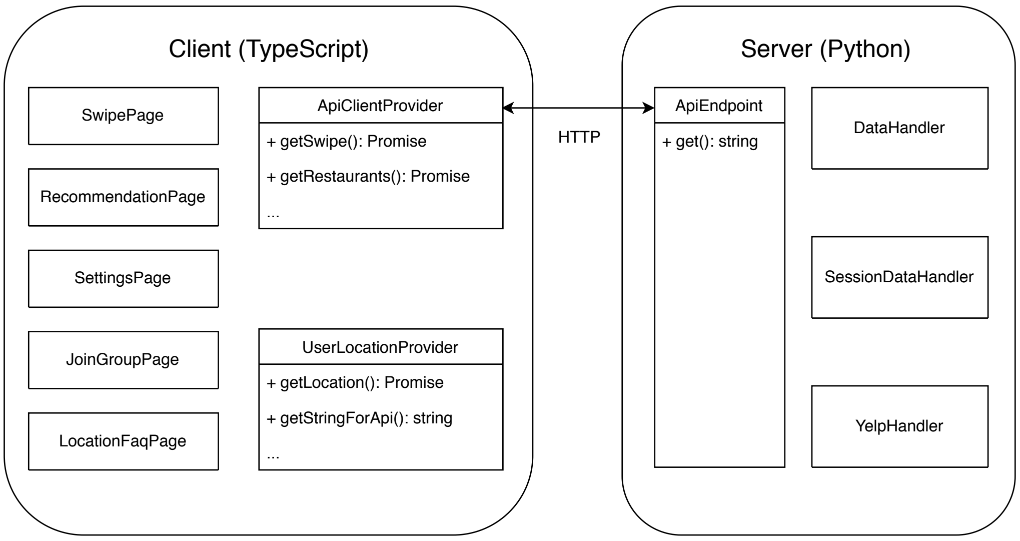


Figure 4.1: Restaurant Swiper's most important classes

4.1.1 Pages

The root page is called `SwipePage` and it is where the user's appetite is determined. The user can either press the thumb buttons or swipe the food picture to the left or to the right. For the thumbs' sliding animation, we use a modified version of animate.css [2]

As soon as a user has swiped all pictures that it got from the server, the `RecommendationPage` is shown. The recommended restaurants are only visible if the user is not in a group, or if the group is locked. Otherwise, the user is told to lock the group resp. to wait for the other users to finish swiping.

The `SettingsPage` provides GUI components for specifying the settings described in Section 3.4. Groups can be created and joined in the `JoinGroupPage`. For users, who have difficulties with the automatic locating method, we provide the `LocationFaqPage` to guide them through the troubleshooting process.

4.1.2 User Location Provider

Modern web browsers offer the functionality to get the location of the client using native device services. Restaurant Swiper uses these features to get to know the location of the user. Nevertheless, it is important to offer the user an alternative to the automatic locating process. Entering the location manually can be useful in several cases:

[2] https://github.com/daneden/animate.css [Accessed 21-February-2019]

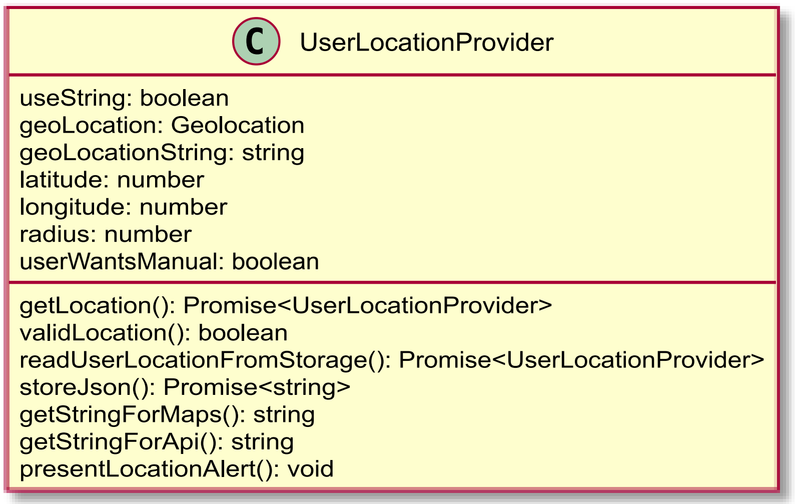

C
UserLocationProvider
useString: boolean
geoLocation: Geolocation
geoLocationString: string
latitude: number
longitude: number
radius: number
userWantsManual: boolean
getLocation(): Promise<UserLocationProvider>
validLocation(): boolean
readUserLocationFromStorage(): Promise<UserLocationProvider>
storeJson(): Promise<string>
getStringForMaps(): string
getStringForApi(): string
presentLocationAlert(): void

Figure 4.2: `UserLocationProvider` class

- Technical issues: Not all devices or browsers offer a location service. Besides that, even modern Android devices have difficulties locating themselves, if Google Location Accuracy is turned off.
- Privacy issues: Some users might not want to turn on location services for privacy reason.
- Current location irrelevant: The current location might not be where the user would like to eat.

In order to function properly, our application *must* know a location to make the initial Yelp request (see Section 3.5.1). For that reason, learning the user's location is one of the app's first tasks when it is opened in a browser. By default, the automatic locating method has the highest priority.

However, if the user has overwritten the automatic location in the settings page, the automatic locating method is skipped. Also, if Restaurant Swiper has no access to the location, another method has to be used. The second priority for getting the user's location is to look in the device's storage for an old location, i.e. from the last use of the application on this device. Whenever interested, the user can check and edit the location in the settings (Section 3.4).

If there is no location stored, the user will be prompted with an alert to enter the location manually (Figure 4.3). Using the location from last time is considered better, since we do not want to bother our users to enter a location all the time. The alert offers the possibilities to enter a location as text and to retry the automatic method. The idea behind the retry button is, that the user resolves the location problems before retrying. If it still did not work, the same

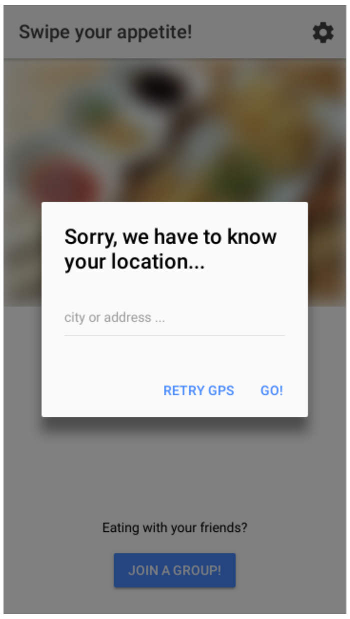

Swipe your appetite!
Sorry, we have to know your location...
city or address ...
RETRY GPS
GO!
Eating with your friends?
JOIN A GROUP!

Figure 4.3: Location Alert

alert reappears with an additional button that navigates to a help page to resolve location issues. With the manual location method, a user can type the name of a city, an address or a train station. A flow chart diagram of the `getLocation()` function can be found in Figure 4.4.

The `UserLocationProvider` class (Figure 4.2) can handle location data from either automatic or manual input. The automatic location is stored as the two numbers `latitude` and `longitude`. A manual location, on the other hand, is stored as a string. If the manual location should be used, the `geolocationString` variable is sent to the Yelp Fusion API in the parameter `location` (see Section 3.2). Otherwise, the location is sent as coordinates (i.e. the parameters `latitude` and `longitude`).

4.1.3 API Client Provider

This provider builds the interface between the front end and the Python server. Initially, the `ApiClientProvider` must get some information from the storage such as the location and the session ID (if it already exists). After that, it can be used for sending some specific requests to the server (see Section 4.2). These functions are asynchronous and return a promise with the response of the server. This means that the component that called the function can define a callback function which is executed as soon as the promise resolves resp. rejects. In addition, the `ApiClientProvider` updates some data in the device storage. The data that is updated at this point contains the `session_id` as well as the

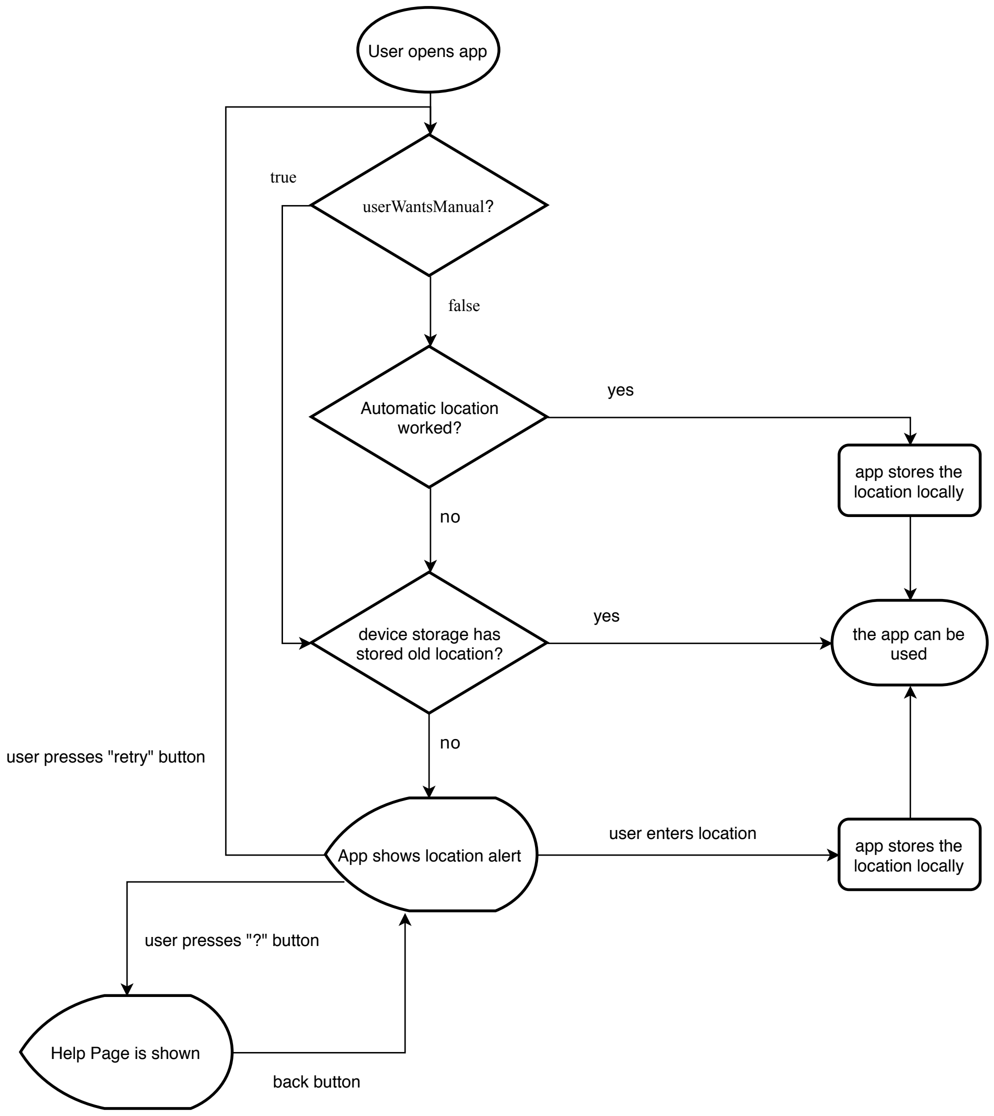


Figure 4.4: `getLocation()`

information whether the client is the master of its group (in the group case).

4.2 Server

As mentioned in the introduction to this chapter, our web application requires two servers to communicate with. Since there was no programming necessary for the web server, we will focus on the Python server in the following. This Python server was specifically written for Restaurant Swiper and is built to handle all requests from the ApiClient provider (Section 4.1.3). As a data format, JSON is used.

Both servers can be accessed over HTTPS. For this purpose, we use ningx as a reverse proxy. This additional step had to be made to enable the automatic locating method on browsers that block the functionality for unencrypted connections. These browsers include Google Chrome [6] and Mozilla Firefox [7].

4.2.1 API Endpoint

`api_endpoint.py` is the main file that is executed to start the server. As the name already suggests, it builds the endpoint between the client and the server. It is powered by Flask to handle HTTP requests and it defines the workflow that is executed on every request from the client. Furthermore, it holds the `SessionDataHandler` object, where the data about all current user and group sessions is stored.

On every request to the API, some initial functions have to be executed, such as getting the right session object from the `SessionDataHandler` and adding some attributes to be sent back to the client. If the client is not recognized, a new session is created (Section 4.2.3) and the `session_id` is added to the response. Therefore. it is important that the client adds its `session_id` to every request. In the first request of a session, the location has to be submitted as well.

The `swipe` endpoint is requested, whenever the Swipe Page is loaded and every time a user swipes a food picture. If the user has not swiped the first picture, the server will respond with the initial food pictures (Section 3.5.1). In the case that the user has swiped, the server receives the set of categories of the swiped image and the vote result (i.e. whether the user liked or disliked the picture). This information gets written to the `SessionDataHandler` object. At this point, the server checks if the user has finished the swiping procedure. If so, the `done_swiping` attribute is set to `True` and added to the response. Otherwise, the server will add some more pictures to the response, if needed.

The endpoints `create_group` and `join_group` handle the grouping mechanism. Creating a group is successful, if the provided `group_name` does not exist, whereas for joining a group, it is vice versa. These endpoints then return explicit feedback indicating the success of the operation. Optionally one can also join a group by opening the application with the group's URL. In that case, any endpoint can be requested.

In order to finally get the list of restaurants that are most suitable for a user resp. a group, the endpoint `recommendation` is used. This is where the `recommend()` function from `YelpHandler` is called. In the group case, this function is only called if it has not yet been called since the settings changed or the group has been unlocked. The reason for this is to make fewer requests to the Yelp Fusion API and to minimize the waiting time for the response on client side. A detailed description of the `recommend()` function is given in Section 3.5.3.

4.2.2 Data Handler

All non-volatile data for the server is stored in `.csv` files. This includes URLs of the food pictures and some statistics on the categories, which is needed for the similarity measurement. For simple access to these files, we introduce the class `DataHandler`. The idea is that the queries provided by `DataHandler` can be called without worrying about data storage. For instance, the function `get_random_picture()` returns a random picture of a given restaurant category resp. of a similar category. Optionally the caller can specify a set of categories to be excluded as well as a set of pictures to be excluded. A *picture* is a Python dictionary containing a URL and a set of categories.

4.2.3 Session Data Handler

The `SessionDataHandler` consists of a list of sessions, a list of group sessions and a `DataHandler` object. It provides many functions that are needed by the `ApiEndpoint` as well as the `YelpHandler`. For instance, this class contains the implementations for getting the right pictures to send to the user (Section 3.5.1). Since we have our own definition of `Session` and `GroupSession`, we will now explain what exactly these classes do.

A session is created, every time a user opens Restaurant Swiper and does not yet have a valid session ID. A session ID is valid if and only if there exists a session in the `SessionDataHandler` with that ID. By default, a session gets deleted after three hours, since we assume that the users have eaten by then. The session keeps track of a user's received pictures and the category preferences.

If multiple users join a group, a `GroupSession` object is stored in addition to each member's `Session` object. The `GroupSession` has a set of session IDs to keep track of its members and is identified by the group name. Since `GroupSession` is a subclass of `Session`, group sessions also get deleted after the group has eaten.

4.2.4 Yelp Handler

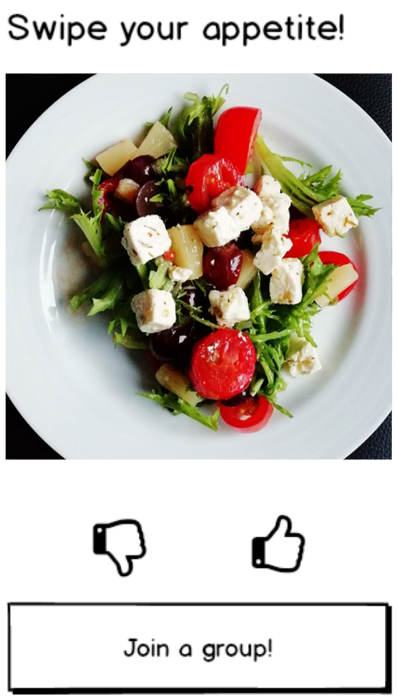
This is where all requests to the Yelp Fusion API (Section 2.1) are constructed and sent. The `YelpHandler` contains several functions such as `get_restaurants_closeby()`, which implements the initial Yelp request described in Section 3.5.1. The `recommend()` function gets a set of possible restaurants (in parallel) and orders them by how well they fit. It returns a small list of restaurants assigning roles to them such as *best*, *cheapest* and *nearest*. If no food pictures were liked, it assumes that the user resp. the group is not hungry and suggests nearby bars.

Chapter 5

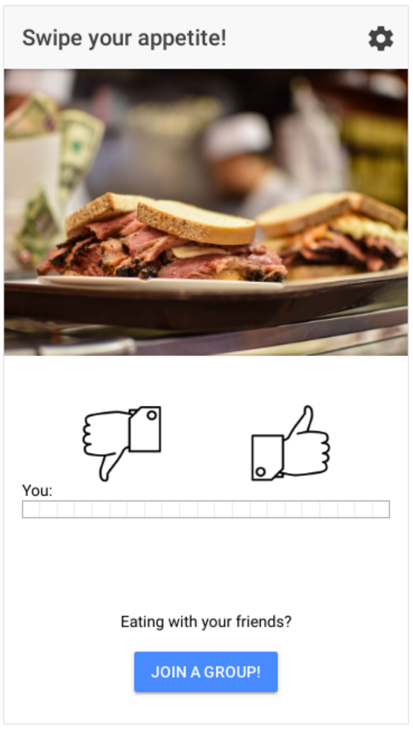
Results

In this thesis, we implemented an application, which is able to recommend restaurants to hungry users, after they have enjoyed a game-like process. The application can run in modern browsers and can therefore be used on a variety of devices (i.e. desktop, tablet, mobile) and operating systems such as Linux, Mac OS, Windows, iOS or Android. With the grouping mechanism, multiple users can form a group to get recommendations based on their current appetite while avoiding arguments about where they should eat.

Despite having invested some time in creating mock-ups and executing usability studies, there are some ways of improvement for the UI. Since Restaurant Swiper is a web application, it can be viewed on different screen sizes. That is why today's web pages put great value on responsiveness. Until this point, our application is optimized for a smartphone in portrait mode, while on wide screens it looks less appealing. Another usability issue is the fact that users might not immediately realize that they should create a group. This feature is extremely valuable because visiting restaurants is usually not done alone. We have not found the perfect compromise for teaching the users how to use the application without bothering them. Instead, the users see a hint at the bottom of the swipe page telling them to invite their friends and from a group (Figure 5.1). From asking some people to use the application, we learned that they usually finish swiping before even realizing that there exists a grouping mechanism.



(a) mockup



(b) result

Figure 5.1: swipe page for a single user

In contrast to the user interface, the server's computations are not directly visible to to the user. One has to test the application multiple times to detect the quality of the restaurant recommendations. Due to spending much time on producing a usable application, there are some flaws in recommendation computation. One example for a non-optimal decision is that the application computes the number of pictures that a user has to swipe before the swiping procedure begins. However, this number should be dependent of the user's votes, such that the application gets a meaningful understanding of the user's appetite. Nevertheless, the usability benefits from this decision because users know how long the swiping is going to take.

Chapter 6

Conclusion

For this thesis, a lot of time was spent on implementing various features to end up with a working application. For every feature, we tried to build a working version as quick as possible so that we could optimize it afterwards. This approach succeeded in the sense that we managed to produce an application that serves its purpose.

6.1 Personal Insights

In spite of having achieved the main goal, our approach might not have been optimal. We might have saved a lot of time if we had planned the code architecture more thoroughly in the beginning. For instance, the communication between client and server is not perfectly abstracted. Adding an element to be sent to the server always involved a great deal of programming. However, we cannot know at this point whether another approach would have been faster.

I am happy with what I achieved with my bachelor thesis. The application works and supports almost all features that were proposed in the task description. Furthermore, the development process was passed quite fluently, without ever getting stuck for multiple days. There were many small issues on the way, but I always managed to make some progress.

6.2 Future Work

By what means can Restaurant Swiper be improved? For instance, one can optimize the user interface as well as the computation of the recommendations. To that end, Chapter 5 gives some ideas of what can be reformed for these features.

One of the Restaurant Swiper's biggest troubles is the quality of the data. For the city of Zurich, the Yelp Fusion API offers data about many restaurants. However, in rural regions, only a small fraction of restaurants have an entry on

Yelp. To solve this issue, one could switch to Google's Places API. For this, one would have to make many small adjustments to Restaurant Swiper, but most concepts should be similar to Yelp.

A small adjustment, that could satisfy many users, lies in the concept of the settings. Instead of indicating a maximum distance, users could then say how much they value the restaurant's distance. Using this input, the application could change the weight of the closeness factor when evaluating the restaurant performances (Section 3.5.3).

Bibliography

[1] List of yelp categories (json file). https://www.yelp.com/developers/documentation/v3/all_category_list/categories.json [Accessed 16-March-2019].

[2] Yelp fusion api documentation: businesses/search. https://www.yelp.com/developers/documentation/v3/business_search [Accessed 16-March-2019].

[3] (2019, Jan.) What is ionic framework? https://ionicframework.com/docs/intro [Accessed 14-March-2019].

[4] Ionic 3 cli documentation: `ionic generate`. https://ionicframework.com/docs/v3/cli/generate/ [Accessed 16-March-2019].

[5] Ionic 4 cli documentation: `ionic generate`. https://ionicframework.com/docs/cli/commands/generate [Accessed 16-March-2019].

[6] P. Kinlan. (2016, Apr.) Geolocation api removed from unsecured origins in chrome 50. https://developers.google.com/web/updates/2016/04/geolocation-on-secure-contexts-only [Accessed 13-March-2019].

[7] (2017, Aug.) Firefox 55 for developers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/55 [Accessed 13-March-2019].