Hochschule für Technik und Wirtschaft -HTW

FOODMATCH

Projekt Bericht Embedded und Mobile Datenbank

Sanchez Neme, Carolina - s0544541
Kügler, Alexander - s0544012
Klemm, Felix - s0545009
CE – Sommersemester 2016
Abgabe erfolgte am: 10.07.2016

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Aufgabenstellung

Das Ziel dieses Projektes war die Entwicklung einer Android-App zu einem selbstgewählten Thema.
Die App sollte Datenbank-basiert sein.

2. Inception und Anforderungsanalyse

2.1 Ideefindung

Unsere Idee war, eine App zu programmieren, die Menschen mit dem passenden Essen zusammen bringt. Daher ist der Name „FOODMATCH" entstanden. Die App richtet sich an alle Smartphonenutzer.

Das Prinzip der App besteht darin, dass die Benutzer Bilder von Gerichten angezeigt bekommen. Die Nutzer können dann durch wischen (swipen) entscheiden, ob ihnen das Gericht gefällt oder nicht. Wenn ihnen ein Gericht gefällt, soll die Route zum Restaurant berechnet und dargestellt werden.

Hierfür sind folgende Funktionen notwendig **(Must-Have):**

- Webserver mit Datenbank zur Datensynchronisation
- Umkreissuche
- Gespeicherte Orte
- Filter: Kategorie und Preis

Weitere Funktionen wären wünschenswert **(Nice-To-Have):**

- Vorliebenprofil
- Nutzerbewertungen
- yelp! API – Für mehr Ergebnisse
- Weitere Filter (z.B. Inhaltsstoffe)

2.2 Monetisierung

Einkünfte soll die App hauptsächlich durch Werbung erzielen. Des Weiteren sind Statistikdaten sehr begehrt und können von daher auch entgeltlich angeboten werden.

Die Bewertung der Nutzer spielt hier eine Rolle. Der Standort, sowie der Zeitpunkt der Bewertung und die Bewertung an sich werden geloggt und alle 5 Minuten mit dem Server synchronisiert.

3.Analyse und Design

3.1 User Interface (UI)

In der Abbildung 1 kann man das gewählte Design erkennen, es besteht aus 2 Hauptansichten:

1. Die Gerichtsauswahl
2. Die Routenberechnung

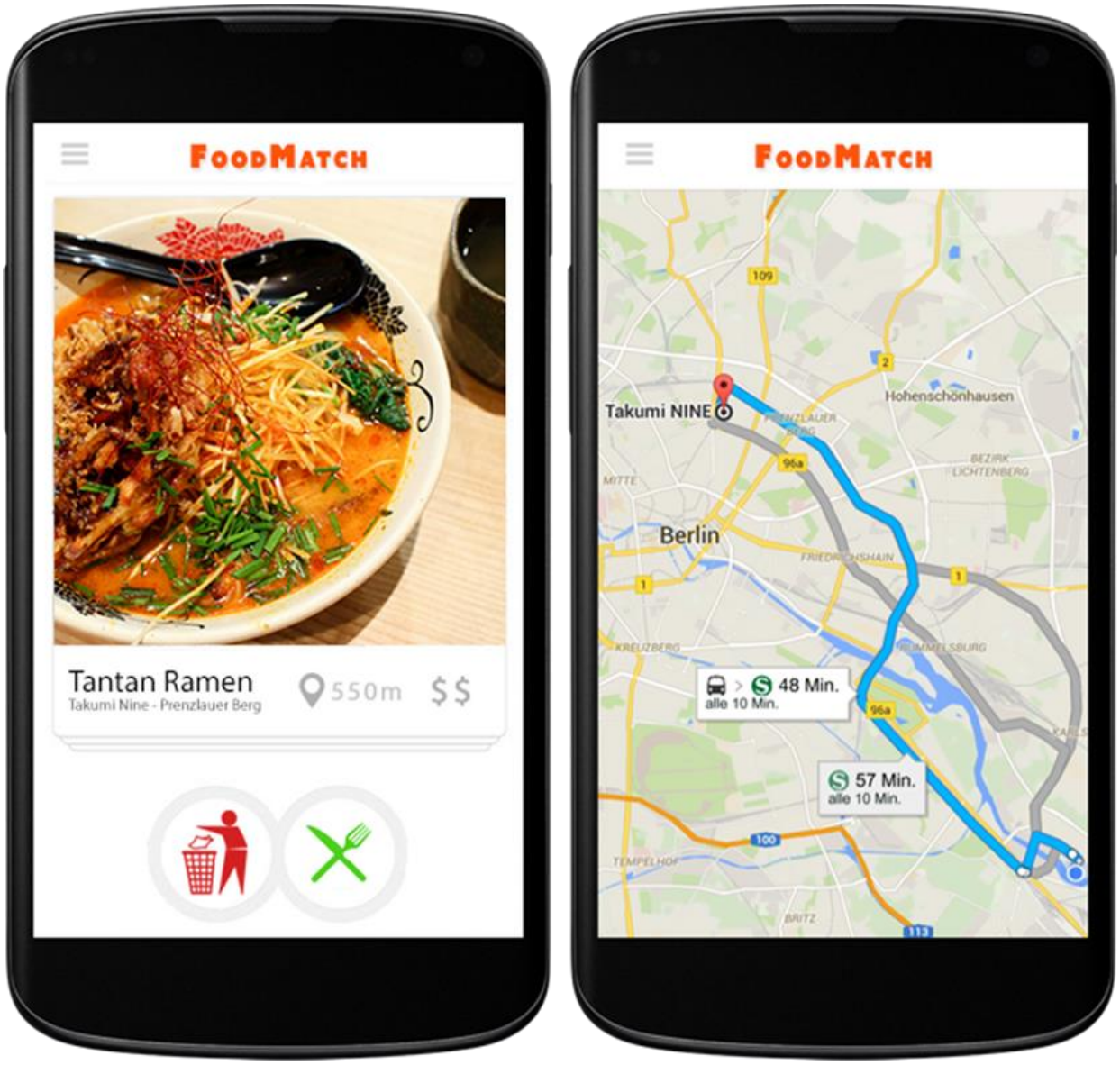

Abbildung 1: Entwurf User Interface

3.2 User Experience (UX)

Die App soll, wie in Abbildung 2 dargestellt, ablaufen. Nach dem Start sollen Die Vorlieben ausgewählt werden. Danach sollen zufällig passende Gerichte dargestellt werden. Hierbei kann der Benutzer entscheiden, ob ihm das Gericht gefällt oder nicht. Schließlich soll, wenn gewünscht die Route anzeigt werden.

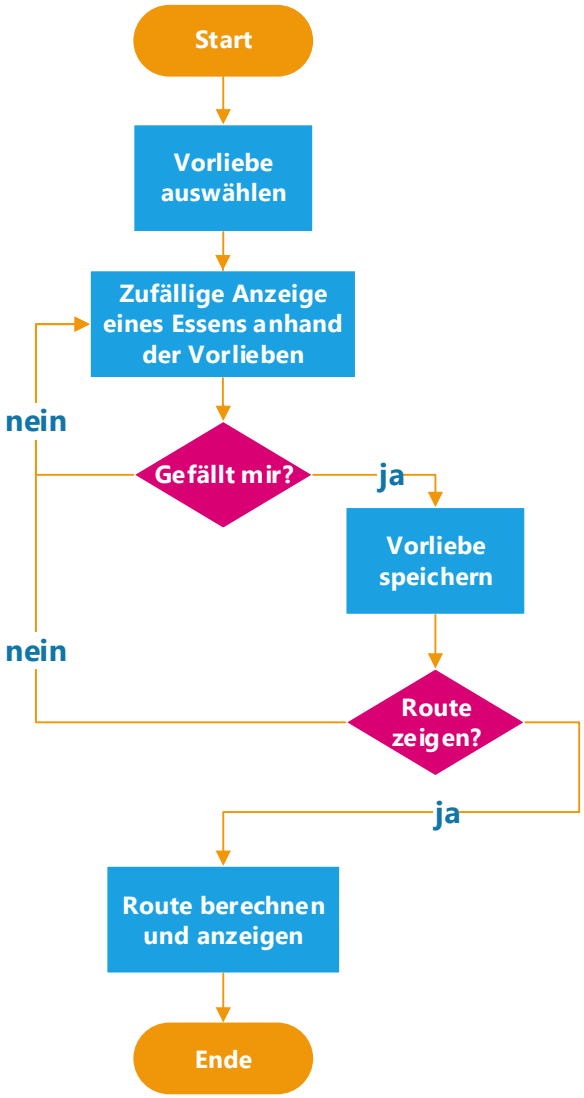

Abbildung 2: Entwurf Ablaufdiagramm

3.3 Datenbankmodell

Als Datenbanktyp ist NoSQL ausgewählt worden. Hierbei steht „No" nicht für „garkein SQL", sondern vielmehr für „not only SQL".

Wir haben uns für NoSQL entschieden, weil es sich besonders gut im App-Bereich eignet. Insbesondere die schnelle und einfache horizontale Skalierbarkeit ist hierfür auschlaggebend.

Weitere Vorteile sind die riesige Open Source Community, die Ausfallsicherheit und die Kostengünstige Skalierbarkeit.

Als Datenbank ist MongoDB definiert worden, da sie am meisten in der NoSQL Szene verbreitet ist.

In der Abbildung 3 kann man die Tabellen der Datenbank sehen.

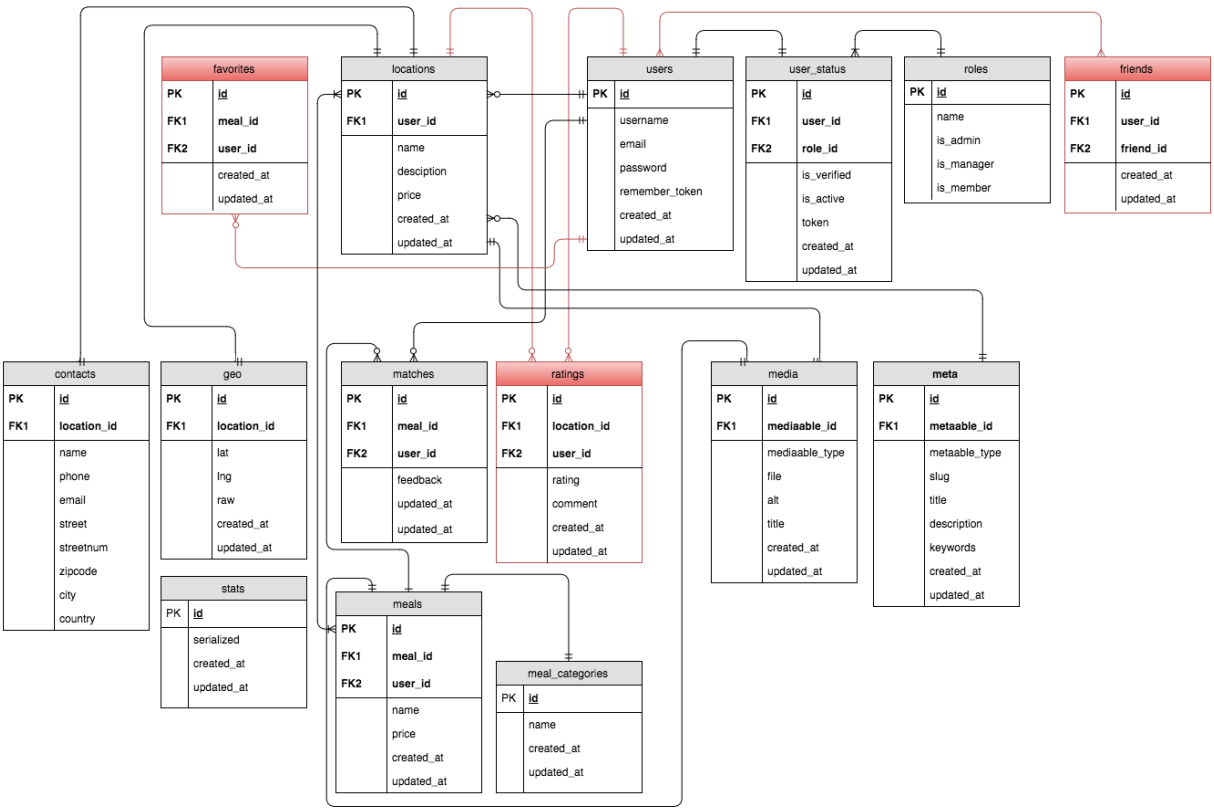

Abbildung 3: Entwurf der Datenbanktabellen.

4. Implementierung

4.1 User Interface (UI)

Das User Interface entspricht der Vorlage, mit eigenem Charakter, der für die App eigens entwickelt wurde. Aus Copyrightgründen darf die Vorlage auch nicht verwendet werden.

Hinzugefügt wurde ein Startbildschirm, der den Nutzer fragt, ob er sofort essen möchte, oder erstmal entdecken möchte, was sich in der Nähe befindet. Ersteres führt dazu, dass sofort nach einem „like" die Google Maps Routenberechnung aufgerufen wird.

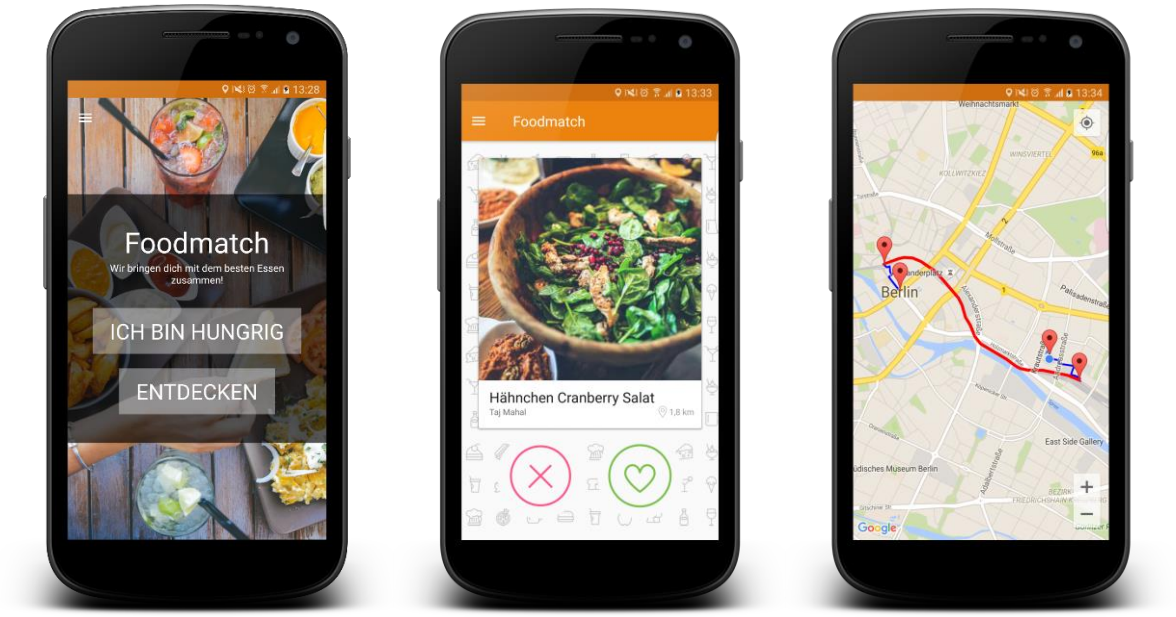

Abbildung 4: Startscreen - MainActivity - MapsActivity

4.1.1 SwipeCards

Die *SwipeCards* Klasse stellt das Hauptmodul der App dar. Die Karten enthalten ein Bild zu dem Gericht und Informationen wie Name, Ort und Distanz. Diese Karten können nach rechts oder links „gewischt" werden, welche Gestik als „liken" oder „disliken" interpretiert wird in der App.

Analog zum Wischen sind auch Buttons vorhanden, die geklickt werden können und die gleiche Funktion aufweisen, wie des Wischens. Diese Methodik wird oft verwendet, um dem Nutzer ein intuitives Verhalten im Sinne der Bedienung anzueignen.

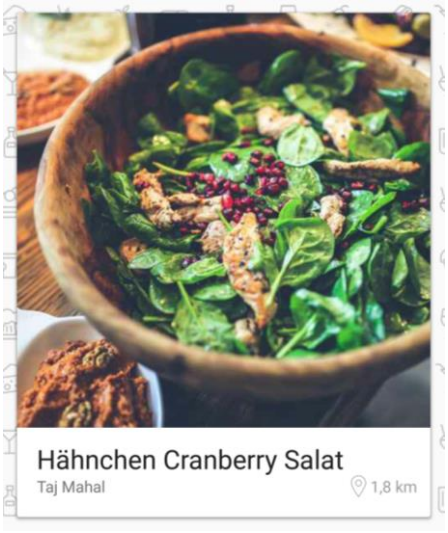

Abbildung 5: Abbildung einer Karte

Wie auf der Abbildung zu sehen ist, enthalten die Karten den Namen des Gerichts, den Namen des Restaurants und eine Angabe der Entfernung in Relation zum aktuellen Standort.

Die Klasse birgt unter anderem noch ein *Listener*, die der *MainActivity* signalisieren kann, dass sie mehr Karten braucht.

4.1.2 Externe Library

Die Funktionalität der wischbaren Karten wird von einer extern eingebundenen Klasse garantiert[1]. Diese Klasse enthält Interfaces, die in *SwipeCards* eingebunden sind, welche für die Auswertung der Benutzerinteraktion zuständig sind. Zudem sorgt diese auch für die intuitiven Animationen und stellt weitere nützliche Funktionen zur Verfügung, wie das Auffüllen des Kartenstapels.

Das Kartendesign entstammt dem *Material Design*. Es wurde darauf geachtet, dass auch ältere Handyversionen mit dem *Material Design* kompatibel sind. Dafür wird die *Support Design Libarary* verwendet nach der Anleitung von der Entwicklerseite[2].

[1] https://github.com/Diolor/Swipecards
[2] https://developer.android.com/training/material/lists-cards.html

4.2 User Experience (UX)

Bei der User Experience gab es Anpassungen zur Planung. Vergleiche Abbildung 2 und Abbildung 6.

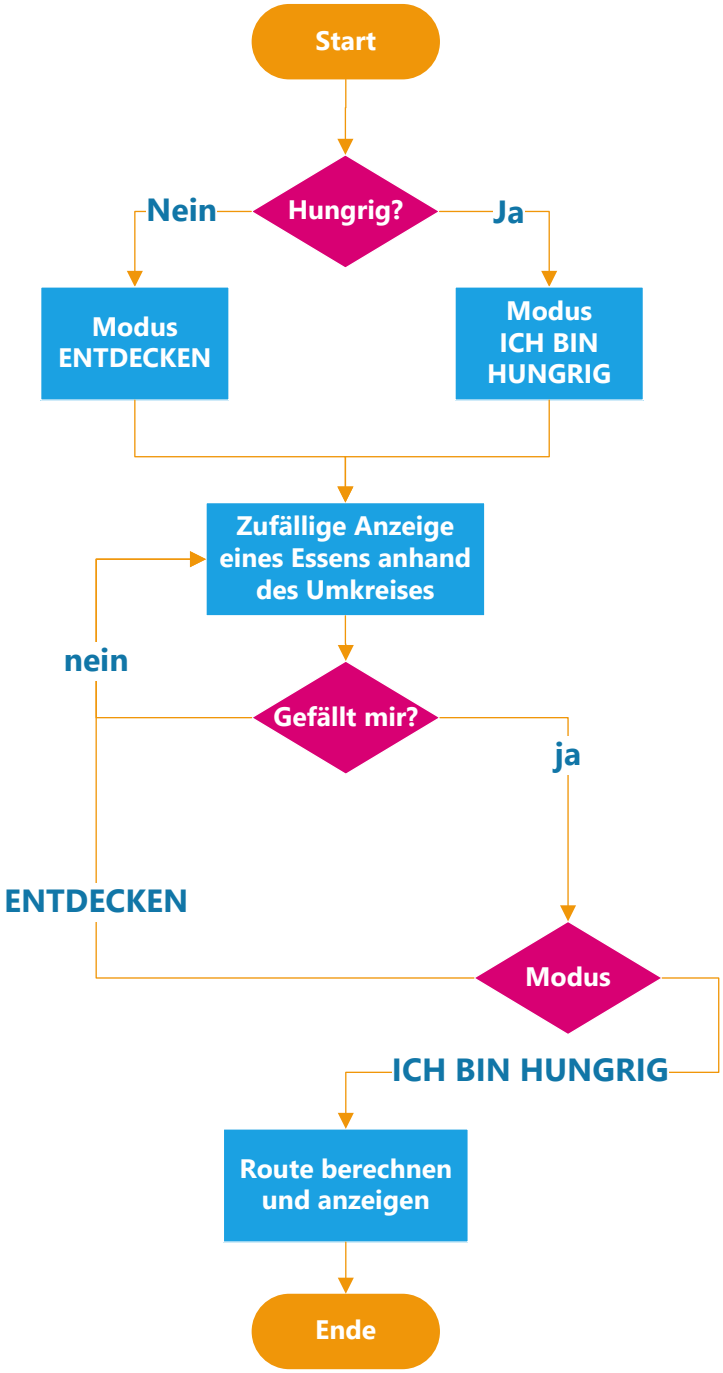

Abbildung 6: Ablaufdiagramm Umsetzung

4.3 Datensynchronisation

Die Datensynchronisation findet asynchron statt, sobald ein Modus (o.g.) ausgewählt ist *oder* die *SwipeCards* Klasse signalisiert, dass diese mehr Karten braucht.

Der aktuelle Standort wird mit dem Google API Service ermittelt und zusammen mit dem Radius an eine URL angehangen, die an den Remote Server adressiert ist.

Während der Wartezeit erscheint ein *Loading Spinner*, der dem Nutzer signalisiert, dass die Daten abgerufen werden.

Die Anfrage liefert ein Ergebnis als JSON[3] Array zurück, welches Restaurant-Daten enthält. Die Restaurants werden mittels eines JSON-Parsers auf Gerichte geprüft und diese werden in die lokale Datenbank gespeichert. Die Datenbank agiert als Puffer zwischen der Datenanfrage und dem Anzeigen der Gerichte. Siehe Abbildung 7.

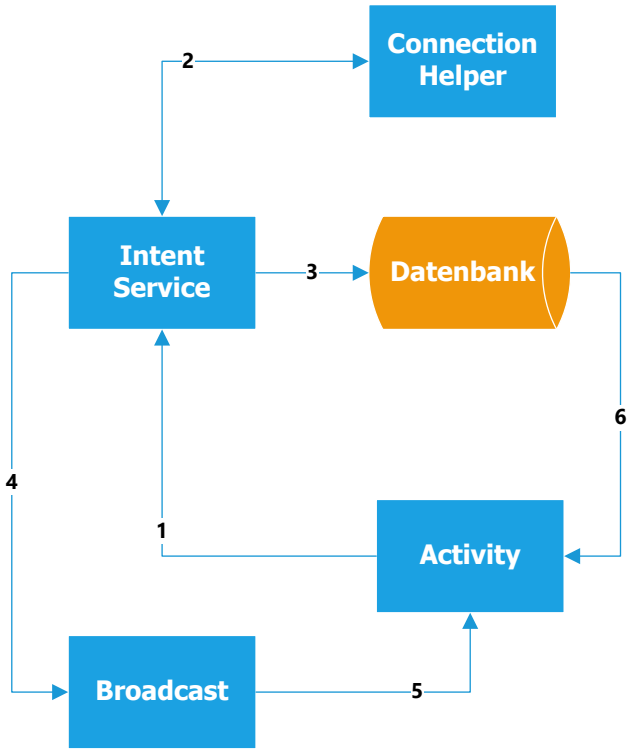

Abbildung 7: Diagramm der Datensynchronisation

Ein *IntentService* wird dabei jedes Mal gestartet, wenn Daten abgerufen werden sollen. Dieser ist nur indirekt mit der *MainActivity* gekoppelt und kann keine Resultate übermitteln. Diesbezüglich fungiert ein *BroadcastReceiver* als Vermittler, die der *MainActivity* eine Statusmeldung zukommen lässt und einen Callback auslöst. Der *BroadcastReceiver* wird von dem *IntentService* aus gestartet.

Die Daten werden dann von der *MainActivity* aus der Datenbank gerufen und den *SwipeCards* übermittelt.

[3] Javascript Object Notation

4.4 Authentifizierung

Jede Anfrage an den Remote Server erfordert einen Schlüssel, der an die URL angehangen wird. Ohne diesen Schlüssel kann keine Kommunikation stattfinden und der Zugriff wird verweigert.

Der Schlüssel wird in der App als interne App-Einstellung gespeichert. Dieser wird bei jeder Anfrage aus dem Einstellungsspeicher geladen.

4.4.1 Erste Authentifizierung

Wird die App das erste Mal gestartet, ist der Schlüssel noch nicht gesetzt. Die App ruft eine spezielle URL zu dem Remote Server auf und sendet einen benutzerdefinierten HTTP-Header. Im Header stehen die Geräte-ID, die Uhrzeit und ein *gehashter* String. Dieser String wird aus der Geräte-ID, der Uhrzeit und einem *Secret Key* zusammengesetzt und *gehasht*. Den *Secret Key* kennen nur die App und der Remote Server.

Der Remote Server überprüft den *Hash*, indem er aus der Geräte-ID, der Uhrzeit und dem *Secret Key* selbstständig einen *Hash* erstellt und diesen mit dem Mitgesendetem vergleicht. Fällt der Vergleich positiv aus, wird ein Schlüssel für die Authentifizierung zurückgesendet, mit welchem die App fortan alle Anfragen authentifiziert.

Dieser Prozess findet auf der Seite der App asynchron über einen *IntentService* statt. Der eigentliche Prozess, um Daten abzurufen ist solange in einer *while*-Schleife gefangen und wird alle 5 Sekunden aktualisiert, um zu ermitteln, ob der Schlüssel schon erhalten wurde. Geschieht dies, wird der eigentliche Prozess fortgesetzt.

Das Aufhalten des Prozesses mit der Schleife stellt kein Problem dar, weil es in einer asynchronen Umgebung geschieht und den *MainActivity Thread* nicht aufhält.

4.5 Webservice

4.5.1 Online Datenbank

Gerade im mobilen App Bereich sind einfach und schnell skalierbare Datenbank wichtig für rasche Erweiterungen bei erhöhtem Nutzeraufkommen. Bei dem Punkt „Skalierbarkeit" stoßen konventionelle sogenannte relationale Datenanken schnell an ihre Grenzen. Diese verfügen über eine feste Struktur bzw. über ein festes Schema.

Genau diese Tatsache wurde in diesem Projekt berücksichtigt. Es kommt hierbei zum Einsatz von MongoDB einem nicht-relationalen Datendanksystem. Diese Datenbank ist dokumentenorientieren und deutlich performanter bei großen Datenmengen, die untereinander verknüpft sind. Zudem verfügt MongoDB über bereits implementierte Features, die beispielsweise bei der benötigten Umkreissuche, eine Menge Arbeit ersparen.

Das Weitern ist dieses Datenbanksystem stark redundant. Es lassen sich schnell und einfach mehrere Datenbanken zusammenschalten, um nahezu eine 100% ige Ausfallsicherheit seitens der Datenbank zu garantieren.

Serverkonfigurationen

Um MongoDB auf dem Server nutzbar zu machen müssen zunächst diverse Programme installiert und konfiguriert werden. MongoDB wird hier in Verbindung mit PHP verwendet. Dazu wird zunächst eine Grundinstallation von dem Datenbanksystem vorausgesetzt und anschließend der PHP-Treiber für MongoDB installiert.

Nachdem diese Schritte erledigt sind ist es möglich das Datenbanksystem über die Konsole zu erreichen: „mongo".

Es werden für die online Datenbank 5 „Collections" benötigt:

- dishes_collections
- categories_collection
- restaurant_Collection
- statistics_collection
- users_collection

4.5.2 Verwaltungsoberfläche

Um Daten in das Datenbanksystem einzupflegen und verwalten zu können wird zusätzlich eine Verwaltungsoberfläche benötigt siehe Abbildung 8 . Diese Vewaltungsoberfläche wird mit dem PHP-Framework „Laravel 5" erstellt. Dabei kommt ein Webserver zum Einsatz.

Sie kann unter folgender Adresse erreicht werden:

http://admin.collective-art.de/
Nutzername: emb
Passwort: google132

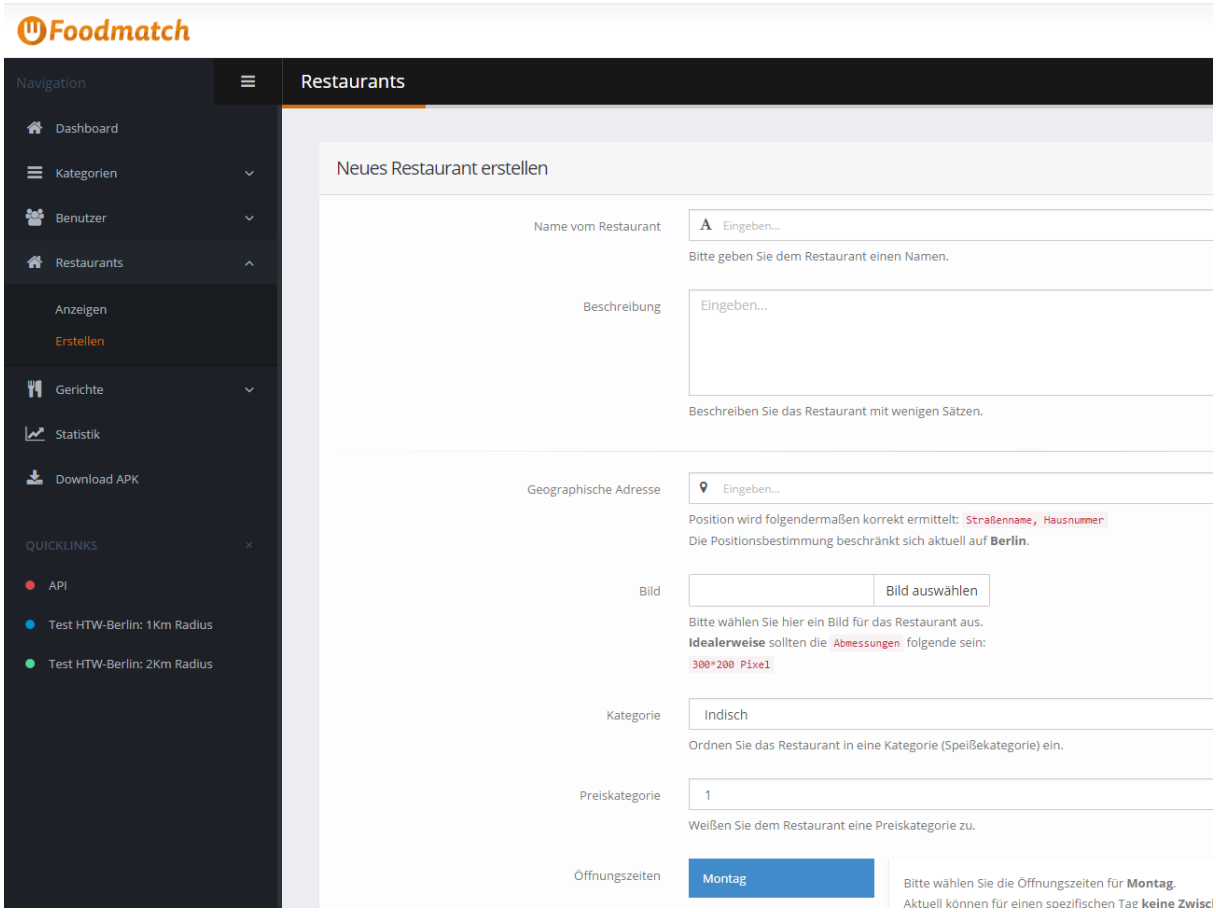

Abbildung 8: Ersteindruck Verwaltungsoberfläche

Dort können Restaurants erstellt, Gerichte angelegt, Kategorien hinzufügt und viele weitere Einstellungen und Änderungen vorgenommen werden. Ein weiterer Vorteil solch einer Oberfläche ist die Analyse vom eigentlichen Nutzerverhalten auf der App. Es können hier Statistiken über einen Spezifischen Nutzer erstellt werden (Vorlieben, Geschmack, etc.). Dies ist möglich, da jedes Handy, welches die App installiert auf dem zentralen Server gespeichert wird (Token, AndroidID). Somit ist jeder App-Nutzer eindeutig mit einer ID versehen.

4.5.3 Schnittstelle

Die eigentliche Schnittstelle befindet sich auf dem gleichen Host. Es wurde hierbei bewusst nicht das bereits bestehende Framework genutzt (Verwaltungsoberfläche) da dieses nicht besonders für API-Schnittstellen geeignet ist (zu viele Klassen).

Aus diesem Grund wird auf ein verwandtes Framework von Laravel zugegriffen: Lumen.

Dieses ist im Vergleich zu anderen API-Frameworks deutlich vorne. Die API ist in unserem Fall absolut „stateless“. Sie liefert bei jedem Zugriff eine reine JSON-Response zurück.

Wird ein Request an diese API gesendet, so wird zunächst geprüft, ob eine Route für diesen spezifischen URL-Zugriff angelegt wurde. Diese „Routen“ werden in einer „routes.php“-Datei verwaltet bzw. angelegt. Dabei wird auch die Art der Anfrage betrachtet (GET, POST, PATCH, PUT, DELETE, usw.)

```php
use Illuminate\Support\Facades\Schema;

$app->get('/', function () use ($app) {

    Schema::collection('restaurants_collection', function($table)
    {
        $table->index(array('geo' => '2dsphere'));
    });

    return $app->version();
});

$app->group(['middleware' => 'api', 'namespace' => 'App\Http\Controllers'], function () use ($app) {

    $app->get('restaurants','RestaurantsController@index');
    $app->get('restaurants/{id}','RestaurantsController@show');
    $app->get('restaurants/{lng}/{lat}/{radius}','RestaurantsController@geo');
    $app->get('restaurants/media/{file}', 'RestaurantsController@media');

    $app->post('statistics','StatisticsController@store');
});

$app->group(['namespace' => 'App\Http\Controllers'], function () use ($app) {

    $app->post('users','UsersController@store');
});
```

Abbildung 9: API Routen

Sicherheit

Damit keine unbefugten Zugriffe auf unsere Daten stattfinden können benutzen wir ein Authentifizierungssystem, welches mit einem eindeutigen Token arbeitet, welcher einem ganz bestimmten Nutzer zugeordnet wird. Erfolgt beispielsweise eine Suche mit Koordinaten und einem Umkreis, so muss stets ein Token mitgeführt werden.

Die getätigten HTTP-Anfragen werden dabei in 2 Gruppen unterteilt:

1. **Zugriff mit Token (Authentifizierung)**

Damit keine unbefugten Zugriffe auf unsere Daten stattfinden können benutzen wir ein Authentifizierungssystem, welches mit einem eindeutigen Token arbeitet, welcher wiederum einem ganz bestimmten Nutzer zugeordnet wird.

Jede Route, die in dieser Gruppe liegt bedingt zuvor ein Durchlauf durch eine sogenannte „Middleware".

Eine „Middleware" steht sozusagen als eine Art „**Schutzpatron" zwischen Anfrage und Weiterleitung der Anfrage an einen Controller.**

In dieser wird stets geprüft, ob der mitgesendete Token auch tatsächlich valide ist (einem Nutzer zugeordnet werden kann).

Ein typischer Zugriff auf die Schnittstelle sieht dabei wie folgt aus:
http://api.collective-art.de/restaurants/13.5264438/52.4569312/2000?pretty&token=n5DUfSC72hPABeEhu89Ex63soJ2oJCQfTxlim8MC6oHVLlrutMa3xDjDursL

http://api.collective-art.de/restaurants/{LONGITUDE}/{LATITUDE}/{RADIUS}?token=TOKEN&offset=OFFSET

Der Token wird somit in jeder Anfrage als URL-Parameter mitgesendet und dann in der „Middleware" geprüft. Der angegebene Offset ist hierbei optional und muss nicht zwingend angegeben werden.

Die gebaute „Middleware" setzt sich aus nachfolgendem Quellcode zusammen.

```php
class BeforeApiAuthMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if(!$request->exists('token')) {

            $response = [
                "status"    => "error",
                "data"      => null,
                "message"   => "token_not_given"
            ];
        } else {

            if(!User::where('token', $request->token)->first() && $request->token != env('MASTER_TOKEN')) {

                $response = [
                    "status"    => "error",
                    "data"      => null,
                    "message"   => "token_not_valid"
                ];
                return response()->json($response);
            }
        }
        return $next($request);
    }
}
```

Abbildung 10: Quellcode Middleware

Nach erfolgreicher Authentifizierung wird die Anfrage an den jeweils zugehörigen „Controller" weitergeleitet.

In einem Controller werden alle Request-Parameter betrachtet und bearbeitet.

a. Restaurants/ Gerichte

Betrachten wir einen „GET" Zugriff mit Geokoordinaten und einem beliebigen Umkreis, um Restaurants im Umkreis mit Gerichten zu finden. Bei diesem Zugriff wird die Anfrage an den „RestaurantController" zu der Methode „geo()" weitergeleitet. Dort wird zunächst unsere dokumentenorientierte Datenbank nach gültigen Ergebnissen für unsere Geokoordinaten mit Umkreis durchsucht. Nachfolgender Quellquellcode zeigt diese stark abstrahierte Funktionalität.

```php
public function geo(Request $request, $lng, $lat, $radius)
{
    // https://docs.mongodb.com/manual/reference/operator/query/near/
    // longitude, latitude

    if($request->has('offset'))
        $offset = (int)$request->offset;
    else
        $offset = 0;

    $restaurants =  Restaurant::where('geo', 'near', array(
        '$geometry' => array('type' => 'Point', 'coordinates' => array((float)$lng, (float)$lat)),
        '$maxDistance' => (int)$radius
    ))->take(5)->skip($offset)->get();
```

Abbildung 11: Abstrahierter Zugriff auf Datenbank

Im weiteren Funktionsverlauf werden die erhaltenen Daten aus der Daten nun für die Ausgabe vorbereitet. Dazu gehört beispielsweise die Sortierung nach Distanz und das eigentliche Berechnen der Distanz von Nutzer und Restaurant.

```php
    if($restaurants->isEmpty()) {

        $response = [
            "status"    => "error",
            "data"      => array(),
            "message"   => "restaurants_empty"
        ];
    } else {

        foreach($restaurants as $restaurant) {

            // eager loading hack
            $restaurant->category;
            foreach($restaurant->dishes as $dish) {
                $dish->statistics;
            }
            $restaurant['distance'] = round($this->distance($restaurant->geo['coordinates'][0], $restaurant->geo['coordinates'][1], $lng, $lat));
        }
        $restaurants = json_decode($restaurants, true);

        usort($restaurants, function($a, $b) {

            $a = $a['distance'];
            $b = $b['distance'];
            if($a == $b) return 0;

            return ($a < $b) ? -1 : 1;
        });

        $response = [
            "status"    => "success",
            "data"      => $restaurants,
            "message"   => null
        ];
    }
    if($request->exists('pretty')) $response = "<pre>".json_encode($response, JSON_PRETTY_PRINT)."</pre>";
    else $response = response()->json($response);

    return $response;
}
```

Abbildung 12: Verarbeitung der Datenbankergebnisse

Man erkennt in der oberen Grafik schon das Ausgabeformat. Es besteht stets aus einem JSON-Objekt in der obersten Stufe.

Eine Anfrage kann sowohl erfolgreich als auch fehlerbehaftet ausgehen. Dafür ist der JSON-Key „**status**" im JSON-Objekt zuständig. Es beinhaltet entweder „**success**" oder „**error**".

Wenn der „status" des JSON-Objektes den Wert „**success**" beinhaltet, dann beinhaltet der JSON-Key „data" ein JSON-Array mit allen gefunden Restaurants.

Sollte es zu einem Fehlerfall kommen, so beschreibt der JSON-Key „message" einen String der in der App ausgewertet wird. (z.B. „restaurants_empty"). Diese Nachrichten dienen uns als lesbare Fehlercodes, die ein Error Handling deutlich simplifizieren.

Nachfolgender Screenshot gibt einen kleinen Einblick in die Struktur der JSON-Response.


```
{
    "status": "success",
    "data": [
        {
            "_id": "577cd9a1e333375e68542371",
            "name": "Restaurante Piazoo",
            "description": "Tolle Italiensische Gerichte zu gigantisch g\u00fcnstigen Preisen. Das Restaurant besticht durc
            "tags": "vegan,glutenfrei,vegetarisch,g\u00fcnstig,ambiente",
            "category_id": "5765d96ae3333740c44ad852",
            "hours": [
                null,
                null,
                [
                    "11:00",
                    "22:00"
                ],
                [
                    "11:00",
                    "22:00"
                ],
                [
                    "11:00",
                    "22:00"
                ],
                [
                    "11:00",
                    "22:00"
                ],
                [
                    "11:00",
                    "22:00"
                ],
                [
                    "11:00",
                    "22:00"
                ]
            ],
            "updated_at": "2016-07-06 10:12:49",
            "created_at": "2016-07-06 10:12:49",
            "geo": {
                "raw": "Mathildenstra\u00dfe 5",
                "type": "Point",
                "coordinates": [
                    13.52652,
                    52.45997
                ],
                "_id": "577cd9a5e333375e68542372"
            },
            "media": {
                "file": "577cd9a1dddba-schwarz.png",
```


Abbildung 13: Einblick JSON-Response

Für weitere Details kann nachfolgende URL aufgerufen werden:

http://api.collective-art.de/restaurants/13.5264438/52.4569312/2000?pretty&token=n5DUfSC72hPABeEhu89Ex63soJ2oJCQfTxlim8MC6oHVLlrutMa3xDjDursL

Es wird in dieser URL der Parameter „pretty" übergeben. Dieser sorgt genau für solche Demonstrationszwecke eine übersichtlichere Darstellung der Elemente.

json_encode($response, JSON_PRETTY_PRINT)

Wichtig: Eine Umkreissuche wird stets auf dem Server durch **MongoDB** vorgenommen – nicht auf dem Device!

a. Statistiken

Ein weiteres Feature, welches in die Schnittstelle integriert wurde, betrifft das Sammeln von Statistiken („like", „dislike", „Koordinaten") zu einem spezifischen Gericht. Es wird zunächst das Klickverhalten in der lokalen Datenbank gespeichert. Nach Ablauf eines Timers wird dieser Datensatz an die Schnittstelle gesendet und direkt danach auch lokal gelöscht.

Der Datensatz wird dabei im POST-REQUEST-BODY im JSON-Format an den Server gesendet.

Nachfolgender Code verarbeitet den Datensatz.

```php
public function store(Request $request)
{
    $response = 0;
    $ratings = json_decode(file_get_contents('php://input'), true);

    $user   = User::where('token', $request->token)->first();

    foreach ($ratings as $key => $value) {

        $data = array();
        $data = array_add($value, 'user_id', $user->id);

        $dish   = Dish::find($value['dish_id']);

        $statistic = $dish->statistics()->orderBy('created_at', 'desc')->where('user_id', $user->id)->where('dish_id', $value['dish_id'])->first();

        if(!empty($statistic)) {

            $datetime1 = new \Carbon\Carbon($data['created_at']);
            $datetime2 = $statistic->created_at;

            $diff = $datetime1->diffInHours($datetime2);

            if($diff >= 2) $dish->statistics()->create($data);
        } else $dish->statistics()->create($data);
    }
    $response = [
        "status"    => "success",
        "data"      => null,
        "message"   => null
    ];
    return response()->json($response);
}
```

Abbildung 14: Statistikdaten speichern

Online wird überprüft, ob einkommende Daten „alt genug" sind, um in der online Datenbank gespeichert zu werden. Mit diesem Timestamp-Prinzip wird garantiert, dass es zu keiner zu großen Datenflut kommt. Nur „likes" und „dislikes" die älter als 2 Stunden sind werden dabei in die online Datenbank gelegt.

Ein Nutzer kann sich somit nach 2 Stunden wieder für ein Gericht entscheiden, obwohl es ihm zuvor nicht gefallen hat.

Zudem werden auch aktuelle Geo-Koordinaten vom Nutzer gespeichert, wenn ihm ein Gericht gefällt oder nicht gefällt. Damit lassen sich Trendkurven erstellen und eine Zielgruppenanalyse durchführen.

2. Zugriff ohne Token (Erstzugriff auf die Schnittstelle)

Die Gruppe beinhaltet lediglich eine Route - http://api.collective-art.de/users

Diese Route ist nur über einen POST-Request zugänglich!

Es fällt auf, dass diese Gruppe keine „Middleware" verwendet – völlig korrekt, da noch kein Token zur Verfügung steht.

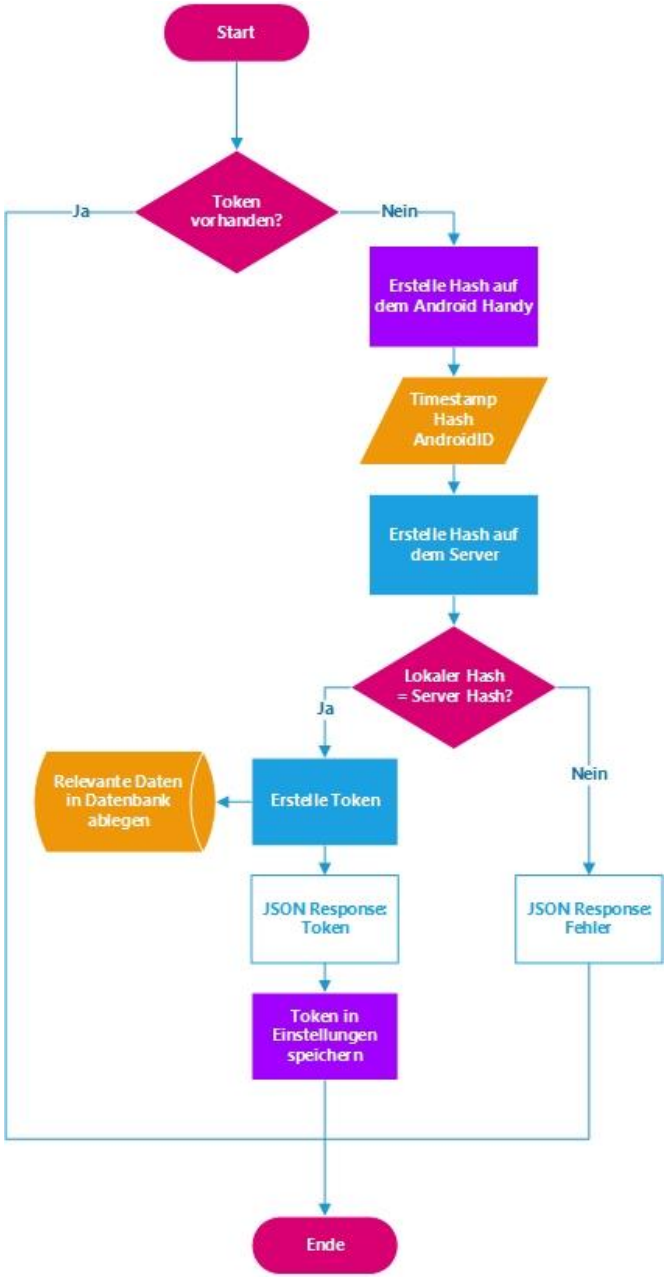

Abbildung 15: Ablaufdiagram Token Generierung

Obiges Ablaufdiagram stellt dar, wie der Token erstellt wird.

Es wird durch den Einblick in die „routes.php"-Datei ersichtlich, dass ein POST-Request an die URL http://api.collective-art.de/users an die Methode „**store**" im „**UsersController**" weitergeleitet und dort verarbeitet wird.

Nachfolgender Screenshot gibt einen Einblick in die eigentliche Generierung des Tokens und die Verifikation des Hashes.

```php
public function store(Request $request)
{
    $secret = '9~3YjmDR=1x>dI~1"Qv3p62m3{u3>9';

    $request_crypted = $_SERVER['HTTP_FM_API_HASH'];
    $server_crypted = sha1($secret."|".$_SERVER['HTTP_FM_API_DEVICE_ID']."|".$_SERVER['HTTP_FM_API_TIMESTAMP']);

    if($request_crypted != $server_crypted) {

        $response = [
            "status"    => "error",
            "data"      => null,
            "message"   => "invalid_checksum"
        ];
    } else {

        $token  = str_random(60);

        $data['device_mac'] = $_SERVER['HTTP_FM_API_DEVICE_ID'];
        $data['token'] = $token;

        if(!$user = User::where('device_mac', $data['device_mac'])->first()) {

            $user = User::create($data);
        } else {

            $user->token = $token;
            $user->save();
        }
        $response = [
            "status"    => "success",
            "data"      => $token,
            "message"   => null
        ];
    }

    return response()->json($response);
}
```

Abbildung 16: Verifikation Hash, Generierung Token

Rein theoretisch wäre es auch ohne App möglich einen Token zu Generierung (von außerhalb – Script). Um genau sowas zu verhindern, kommt ein Sicherheitsfeature zum Einsatz, bei dem sich jeweils ein und derselbe Schlüssel (Secret) auf der App und dem Server befindet. Es werden nun der aktuelle TImestamp auf dem Gerät, die Device-ID und der erwähnte Schlüssel folgendermaßen verschlüsselt:

sha1(SCHLÜSSEL|DEVICE_ID|DEVICE_TIMESTAMP)

Als Verschlüsselungsfunktion wird sha1 benutzt, welche eine 160 Bit Wert zurückliefert.

Dieser Wert wird im POST-Header als Hash-Wert mitgesendet. Das beschriebene Prozedere wird auf dem Server wiederholt und im Anschluss beide Hash-Werte verglichen.

Mit diesem Sicherheitsfeature erreichen wir trotz fehlender SSL-Verschlüsselung eine relativ sichere Umgebung.

Stimmen beide Hash-Werte überein, so wird direkt im Anschluss ein 60-Byte Token generiert. Alle gesammelten Informationen werden inklusive Token in die online Datenbank gelegt und der erstellte Token zurück an das Gerät gesendet.

Der zurückgesendete Token wird letztendlich in den „Preferences" gespeichert. Damit kann nun eine Abfrage der Restaurants erfolgen.

4.5.4 Landing Page

Damit auch potentielle Nutzer dieser App sich ausreichend über den thematischen Inhalt dieser App informieren können wurde folgende „Landing Page" eingerichtet.

http://foodmatch.collective-art.de/

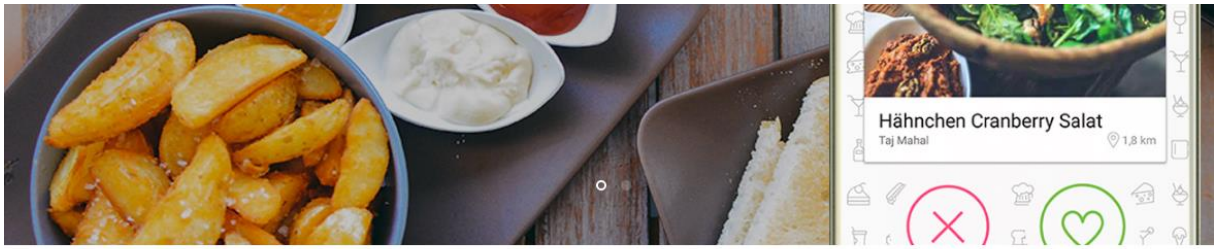

Abbildung 17: Eindruck von der Landing Page

Dort werden Funktionalitäten und Nutzererfahrungen übersichtlich dargestellt. Zudem hat der Nutzer die Möglichkeit sich die APK-Datei direkt auf sein Mobiltelefon zu installieren, ohne den App Store nutzen zu müssen.

Genau aus diesem Grund wurde die Webseite auch für mobile Endgeräte optimiert.

Lizenzen und Rechte

Für alle Gerichte und Webseiten wurden hierbei Grafiken von „PEXELS" verwendet. Alle Grafiken, die bei Uns zum Einsatz kommen sind dabei kommerziell frei nutzbar und bedürfen keiner weiteren Verweise zum Bildeigentümer.

4.6 Lokale Datenbank

4.6.1 Rating Datenbank

Um die Nutzervorlieben speichern zu können, wurde eine lokale SQLite-Datenbank angelegt Diese soll für statistische Zwecke genutzt werden.

Hierfür sind 2 Klassen eingelegt worden „*RatingsContract*" und „*RatingDataSource*".

In der Hilfsklasse „*DatabaseHelper*" werden die SQLite-Datenbanken erzeugt. (sowohl für Rating als auch Gerichte Datenbank).

In der Klasse „*RatingsContract*" werden wichtige Konstanten definiert, die für die Arbeit mit der Datenbank benötigt werden. Wie z.B. Tabellennamen oder die Namen der Spalten.

Die Klasse „*RatingDataSource*" ist das Data Access Object und ist für das Verwalten der Daten zuständig. Sie unterhält die Datenbankverbindung und ist für das Hinzufügen und Auslesen von Datensätzen verantwortlich.

4.6.2 Gerichte Datenbank

Die Gerichte Datenbank trägt alle wichtigen Informationen, welche die *SwipeCards* Klasse braucht. Diese Datenbank ist nur ein Puffer zwischen Datenabfrage und Datenanzeige. Sie wird von daher beim Verlassen der App jedes Mal geleert.

Wichtig zu erwähnen ist, dass während einer Session alle Daten, die vom Remote Server kommen und neu eingetragen werden, einen Statusflag erhalten. Dieser signalisiert, ob die Karte schon verbraucht (*Consumed*) ist. Verbraucht wird die Karte bei der *User Interaction*, sobald der Nutzer die Karte sieht und bewertet.

Daten können nicht überschrieben werden. Es wird auf die ID des Gerichtes geprüft, ob der Datensatz schon vorhanden ist und wenn dies der Fall ist, wird der Neue ignoriert.

4.7 Routenberechnung

Zur Anzeige und Berechnung der Route sind folgende APIs[4] verwendet worden:

- Google Maps Android API
- Google Maps Directions API

[4] Application Programming Interface (Programmierschnittstelle)

4.7.1 Schlüsselgenerierung

Um die Google API überhaupt benutzen zu dürfen, ist es erforderlich einen Schlüssel zu generieren um den Zugang zu ermöglichen.

Um diesen Schlüssel zu erhalten muss man sich unter https://console.developers.google.com/ anmelden, danach müssen folgende Schritte ausgeführt werden:

- Projekt erstellen
- Unter „APIs und Authentifizierung" die benötigten APIs auswählen
- Unter „Zugangsdaten" einen neuen Schlüssel für öffentlichen API-Zugriff erstellen (Der API Schlüssel ist an das Zertifikat[5] gebunden mit dem die App signiert wird)
- SHA1-Fingerabdruck erstellen: im Terminal das Kommando *„keytool -list -alias androiddebugkey -keystore debug.keystore -storepass android -keypass android"* ausführen
- Den erzeugten SHA1-Fingerabdruck und den Packagename des Projektes in die Felder der Website einfügen siehe Abbildung 18

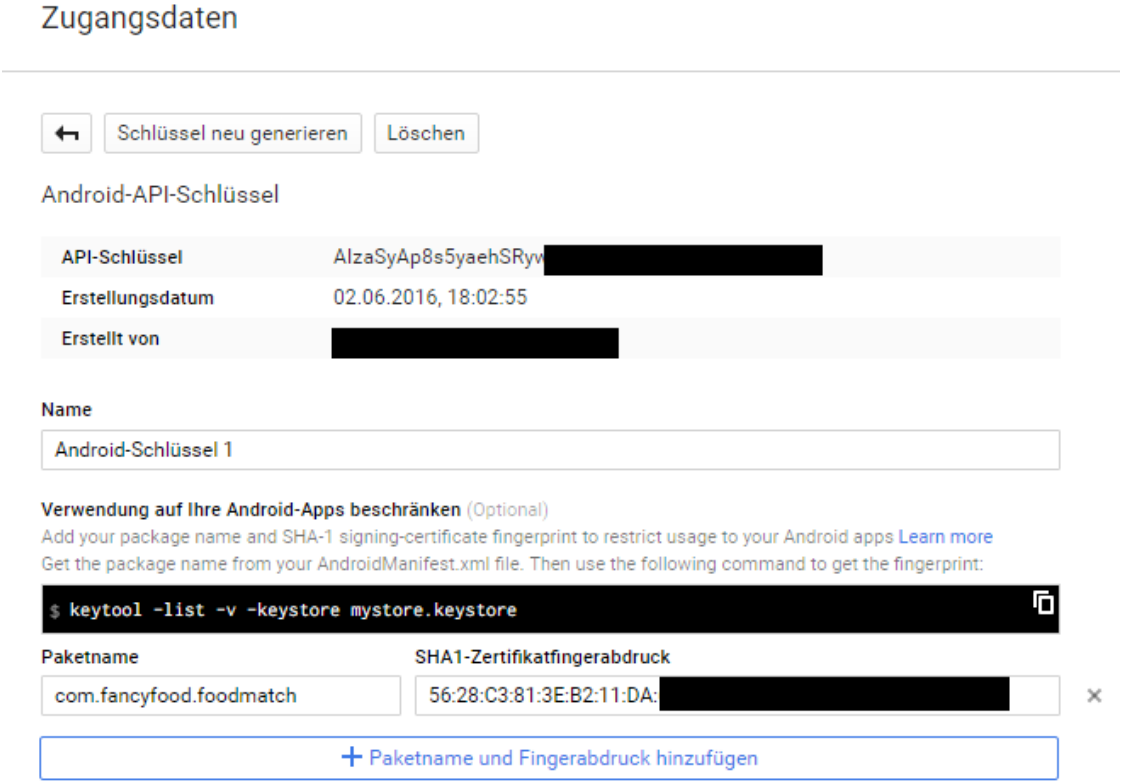

Abbildung 18: Schlüssel für API-Verwendung

4.7.2 Google Maps Android API

Diese API wird für die Darstellung der Karte verwendet, die Verwendung ist kostenlos. Diese API ist ausgewählt worden, da sie eine einfache Möglichkeit bietet um Karten anzuzeigen und auf ihr eigene Markierungen zu setzen.
Die verwendete Klasse heißt „GoogleMap" und stellt alle benötigten Funktionen bereits zur Verfügung, wie z.B Die Schaltfläche „Mein Standort", die es ermöglicht den eigenen Standort

[5] das Zertifikat wird in unserem FOODMATCH Projekt im Android Studio mit „Build"→"Generate Signed APK…" erstellt

als kleine blauen Punkt mit Richtungsangabe auf der Karte darzustellen. So wie die Zoomschaltfläche. (Abbildung 19)

4.7.3 Google Maps Directions API

Für die Bestimmung der Route, wird Google Maps Direction API verwendet. Die Verwendung erfordert wieder einen Schlüssel (Server-Schlüssel) und ist bis 2500 Abfragen pro Tag kostenlos.

Für die Verwendung müssen Anfragen nach dem Schema erstellt werden.

```
https://maps.googleapis.com/maps/api/directions/json?origin=Toronto&destination=Montreal&key= YOUR_API_KEY
```

Da diese Verwendung recht umständlich ist, haben wir uns nach Alternativen umgesehen und die Android-GoogleDirectionLibrary[6] gefunden.

Die Verwendung ist deutlich intuitiver und ist wie folgt:

```
GoogleDirection.withServerKey(String serverKey)
    .from(LatLng origin)
    .to(LatLng destination)
    .execute(DirectionCallback callback);
```

Mithilfe der Android-GoogleDirectionLibrary ist die Routenberechnung umgesetzt worden. Hierfür ist zuerst der aktuelle Standort bestimmt worden und zusammen mit dem Ziel der Android-GoogleDirectionLibrary übergeben worden. Mit den Daten der Antwort, werden auf der Karte Marker an den Eckpunkten sowie Verbindungslinien gezeichnet (vergleiche Abbildung 19: Darstellung der Route auf der Karte

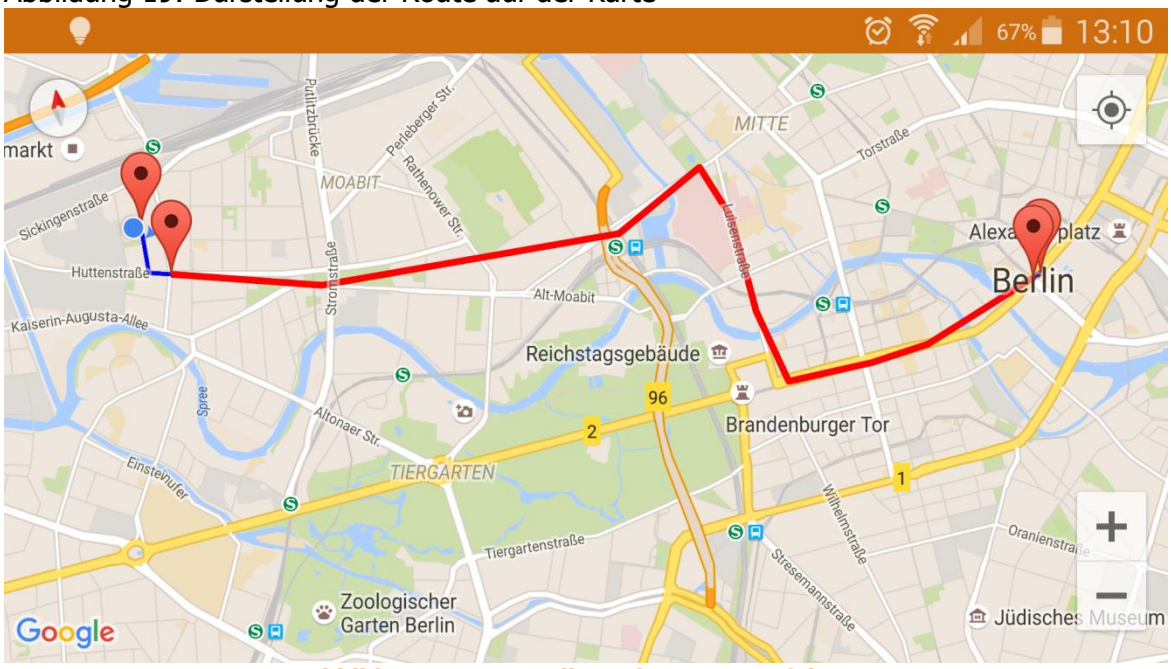

Abbildung 19: Darstellung der Route auf der Karte

[6] https://github.com/akexorcist/Android-GoogleDirectionLibrary

5. Testen und Deployment

5.1 User Interaction Test

Der User Interaction Test verlief teilweise problemlos mit dem *Espresso* Framework. Dieser wird von der offiziellen Android Developer Seite empfohlen.

Das einzige Problem, welches sich erwies, war das „wischen" der Karten, weil es nicht emuliert werden kann. Das Drücken der Knöpfe wiederum, hat funktioniert.

5.2 Instrumentation Test

Für das Testen der MapsActivity ist die Klasse *„ActivityInstrumentationTestCase2*" verwendet worden.

Diese Testart ist gewählt worden, da sie Zugriff auf „Android System Services" bietet. Dies ist notwendig, da für die MapsActivity Zugriff auf den aktuellen Standort zwingend notwendig ist. Der Test läuft daher schon auf dem Zielsystem[7] und nicht in einer Virtuellen Machine, es wird nur die einzelne Activity getestet.

Hierbei haben wir mehrere kleine Tests implementiert, um die Methoden der MapsActivity abprüfen zu können.

5.3 Deployment

Die App kann zurzeit nur von der, schon in Abschnitt 4.5.4 beschriebenen, Landing Page bezogen werden.

[7] Smartphone