

UNIVERSITÃ DE SOUSSE  
ECOLE NATIONALE D'INGÃNIEURS DE SOUSSE

## Rapport de Projet de Fin d'Etudes

PrÃsentÃ en vue de l'obtention du diplÃme d'

**IngÃnieur en GÃnie .....**

Option ...

---

# Conception et dÃveloppement d'une application

---

*RÃalisÃ par:*

PrÃnom NOM

*EncadrÃ par:*

Mr. PrÃnom NOM

To my family and all my beloved.

## Acknowledgements

I would like to express my gratitude towards my university's supervisor, Mr. Taha Ben Salah, and my company's supervisor Mr. Richard Lindberg for supporting me during my internship and for all the knowledge and experience they shared with me. I would also like to thank all my professors for teaching me and helping me achieve my goals.

# Contents

<b>1</b>	<b>State of the art</b>	<b>4</b>
1.1	Formalism . . . . .	7
<b>2</b>	<b>Specifications</b>	<b>8</b>
2.1	Actors . . . . .	8
2.1.1	Internal actors . . . . .	8
2.1.2	External actors . . . . .	9
2.2	Functionalities' overview . . . . .	9
2.2.1	Register . . . . .	11
2.2.2	Login . . . . .	13
2.2.3	Search . . . . .	15
2.2.4	View details . . . . .	17
2.2.5	Review a place . . . . .	19
<b>3</b>	<b>Design &amp; architecture</b>	<b>23</b>
3.1	Physical architecture . . . . .	23
3.2	Logical architecture . . . . .	25
3.3	Modular architecture . . . . .	27
3.4	Modules . . . . .	27
3.4.1	searchAdModule . . . . .	27
3.4.2	OPR app module . . . . .	29
3.4.3	REST API module . . . . .	33

---

<b>4</b>	<b>Implementation</b>	<b>44</b>
4.1	Technology choices . . . . .	44
4.2	Screenshots . . . . .	47
4.2.1	Web interface . . . . .	47
4.2.2	API . . . . .	51
4.3	Deployment . . . . .	53
4.4	Testing . . . . .	55
4.5	Work environment . . . . .	58
<b>5</b>	<b>Conclusion</b>	<b>60</b>
<b>6</b>	<b>References</b>	<b>61</b>

## List of Figures

1.1	screenshot of Tripadvisor . . . . .	4
1.2	screenshot of search results for Trivago . . . . .	5
1.3	screenshot of Momondo . . . . .	5
1.4	example of an expert blog . . . . .	6
2.1	general usecase diagram . . . . .	10
2.2	registration sequence diagram . . . . .	11
2.3	login sequence diagram . . . . .	13
2.4	search sequence diagram . . . . .	15
2.5	view details sequence diagram . . . . .	17
2.6	review a place sequence diagram . . . . .	19
2.7	manager usecase diagram . . . . .	21
2.8	admin usecase diagram . . . . .	22
3.1	The app's physical architecture . . . . .	24
3.2	The app's logical architecture . . . . .	26
3.3	The app's modular architecture . . . . .	27
3.4	searchAd class diagram . . . . .	28
3.5	search sequence diagram . . . . .	30
3.6	view-details sequence diagram . . . . .	31
3.7	review a place sequence diagram . . . . .	32

3.8	interaction overview diagram . . . . .	33
4.1	Chart showing how nginx's user base is growing while apache's is regressing . . . . .	46
4.2	the landing screen of the app . . . . .	47
4.3	search screen . . . . .	48
4.4	showing the location of a place on the map . . . . .	48
4.5	place details . . . . .	49
4.6	Creating a review . . . . .	49
4.7	Thanks screen . . . . .	50
4.8	Showing a review . . . . .	50
4.9	API Login . . . . .	51
4.10	API Login failure . . . . .	51
4.11	API search request . . . . .	52
4.12	API search request . . . . .	52
4.13	Logout API call . . . . .	53
4.14	AWS EC2 allowed inbound traffic . . . . .	53
4.15	AWS EC2 allowed outbound traffic . . . . .	54
4.16	AWS RDS setup . . . . .	54
4.17	NGINX setup . . . . .	55
4.18	unit tests . . . . .	56
4.19	landing page speed test . . . . .	57
4.20	details page speed test . . . . .	57
4.21	facebook page speed test . . . . .	57
4.22	concurrent users test . . . . .	58

## List of Tables

1.1	comparison of tourism platforms . . . . .	6
2.1	Registration description . . . . .	12
2.2	login description . . . . .	14
2.3	search description . . . . .	16
2.4	view-details description . . . . .	18
2.5	review a place description . . . . .	20
4.1	MySQL vs PostgreSQL . . . . .	45



# Abstract

My graduation project consists in creating a web application that contains listings of hotels, restaurants and any place available in the Google places API and allows registered users to create reviews about these places as well as rate these places following some specific criteria like the use of renewable energy and the use of recyclable materials.. The web app also provides a REST API that is being used by a mobile app developed separately. Part of the project also consists of deploying the app to the cloud.

## Keywords

Python - Django - Sustainable Tourism - REST API - Amazon Web Services - Ratings

# Introduction

The internet is full of tourism platforms that list hotels, restaurants and similar tourism related services. To name a few of them we can cite as notable examples: Tripadvisor, Trivago and Kayak. There is literally tens of these platforms, from different countries and in different shapes and colours. However all of the existing platforms only let users write "general" reviews about the listed places, and the visitors can only distinguish between those places based on either the prices or the "general" reviews. It turns out that there's also a market for people who are interested in "sustainable tourism" and don't mind paying more for an eco-friendly vacation. Sustainable tourism is defined as "tourism that respects local people, the traveller, cultural heritage and the environment" by UNESCO, you think of it as "fairtrade" and "organic" but for the travel sector.

One Planet Rating (OPR) was created to fill this void and serve this portion of tourists by listing the same places while focusing on sustainability instead of price. One Planet Rating is a newly founded social startup based in Stockholm, with a portion of its team members working remotely (partially including myself). The startup is in it's very early stages, it's main product/service is the web application described in this report. The company's service isn't released to the public yet. The company thrives to have an impact and aims, with its services, to help achieve some of the United Nations' Sustainable Development Goals (UN SDGs) like:

- UN SDG 8: "Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all".
- UN SDG 15: Protect, restore and promote sustainable use of terrestrial ecosystems, sustainably manage forests, combat desertification, and halt and reverse land degradation and halt biodiversity loss"

- UN SDG 13: "Take urgent action to combat climate change and its impacts by regulating emissions and promoting developments in renewable energy."

OPR is a tourism platform that is sustainability focused, it allows people from around the world to rate and review touristic services based on how much they respect the earth. The long term goal of OPR is to promote the movement of sustainable tourism and incite people to be more responsible during their trips, and eventually help protect the environment and the community. OPR also emphasizes on transparency as it doesn't participate in writing the reviews and ratings, it only provides a playground for all people to express themselves freely.

In this report we'll briefly cover the state of the art and market analysis of the project. In the first chapter we'll go through the requirements and specifications. The second chapter covers the design aspect of the project. The third chapter covers the development and deployment aspects of the project.

## CHAPTER 1

# State of the art

Many online tourism platforms exist today, we can consider these platforms as "indirect competitors" because they offer an alternative functionality to OPR, but what makes the most difference is the target market. Some of these major platforms are Tripadvisor(Fig.1), Trivago(Fig.2) and Momondo(Fig.3). As shown in the following screenshots, all of those platforms focus on the general ratings and the price. They offer very similar functionalities but in different layouts.

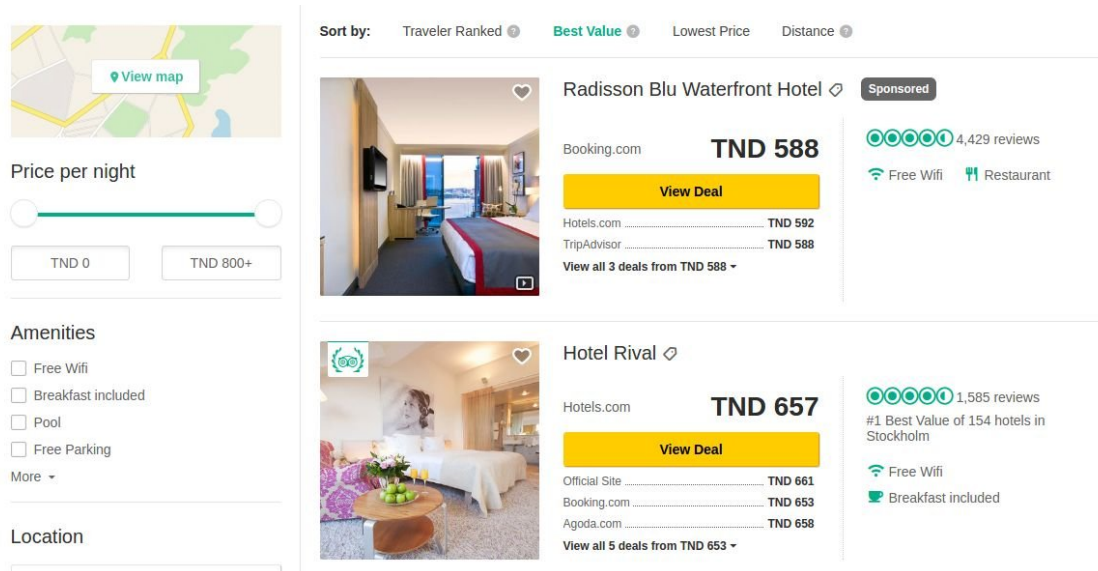


Figure 1.1: screenshot of Tripadvisor

Trip advisor, which is the world leader in the tourism sector provides hotel booking and listing of tourism establishments. The company is based in the United States, and it also offers other related services like tourism focused forums. The platform also allows users to search for and book flight tickets.

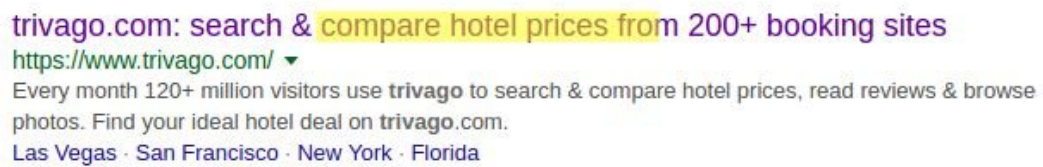


Figure 1.2: screenshot of search results for Trivago

In contrast, the Germany based company, Trivago, only focuses on hotels. Trivago is the leader hotel search engine in Germany. Though trivago only lists hotels, it is owned by Expedia, which owns and operates more than 200 websites that offer diverse services in the tourism and travel sectors.

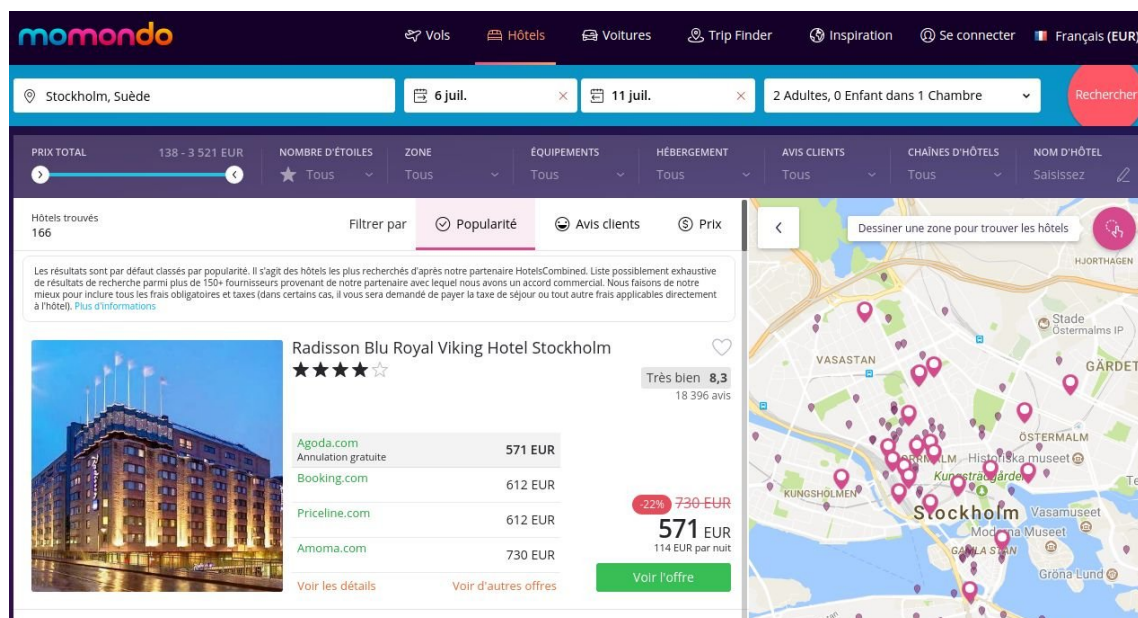


Figure 1.3: screenshot of Momondo

Momondo is also a similar service which, in addition to listing hotels, also provides flight booking and car rental services. Another interesting thing about Momondo is that it's based in Scandinavia, and this makes of it a local competitor.

Another thing to note is that most of these "indirect competitors" are booking websites while OPR is not. OPR's goal is to promote the sustainable places and punish the unsustainable ones.

The following table shows more in detail the difference between OPR and the above-mentioned competitors:

Funtionality	Tripadvisor	Trivago	Momondo	OPR
Search hotels	YES	YES	YES	YES
Search restaurants	YES	NO	NO	YES
Search activities and other places	YES	NO	NO	YES
Search for flights	YES	NO	YES	NO
Search for flights	NO	NO	YES	NO
Booking service	YES	YES	YES	NO
General ratings	YES	YES	YES	YES
Sustainabilityratings	NO	NO	NO	YES

Table 1.1: comparison of tourism platforms

Another category of competitors includes magazines, blogs and expert websites (like shown in Fig.4) that promote some specific sustainable places. Those are also indirect competitors because they rely on experts' advice on their reviews which tend to be not very trustworthy. OPR's content will be completely user generated, because most people believe more in reviews created by peers and want to be part of the content creation. OPR will only be the intermediary between content creators and content consumers, and will not participate in content creation.

## ABOUT TIME .



Figure 1.4: example of an expert blog

## 1.1 Formalism

During the work on this project we used the following formal methods:

### UML

We used UML (Unified Modeling Language) for describing and modeling the specifications of the project. UML is very flexible and versatile. UML is also one of the best choices for modeling because it is very popular and widely used across the globe, which makes it easier to grasp by other people. Most software engineers are probably familiar with it.

### Agile scrum

We used the agile scrum methodology because it is the most convenient method to the project. Every week we have a sprint where we define the goals of that week from the backlog. This methodology turned to be very efficient, especially that the project consists in developing a prototype, which requires continuous thinking and customization. Agile scrum allowed us to customize the project while working on it.

## CHAPTER 2

# Specifications

## 2.1 Actors

### 2.1.1 Internal actors

#### **User**

Anybody who accesses OPR Ratings via App or Web is a User. As opposed to "Member" (see below) this does not require any log-on, or creation of a OPR Profile. A User can search for specific places and read reviews of those places.

#### **Member**

OPR Members are Users who sign up to use OPR's services with a Profile. A User needs to be logged in, to be acting as a Member. A member can do everything a User can do, in addition to the ability of posting their own reviews and ratings.

#### **Manager**

In addition to the permissions that a Member has, an Admin has the permission to Create, Read, Update and Delete (CRUD) Ads, reviews and rating categories.

#### **Admin**

The admin is responsible of managing the users, he has the permission to CRUD users.



### 2.1.2 External actors

#### Google places service

Google places API is called for every search to look for relevant places based on provided location.

The API provides us with :

Basic information (Name, address..), Photo, Google's place\_id (unique identifier)

#### Google maps service

Google Maps API provides us with a map pointing at the specific location based on the location's place\_id we retrieve from Google Places API

#### Gmail service

Google's Gmail is used to send the emails to users.

## 2.2 Functionalities' overview

The main functionality of the platform is to allow members to read and post reviews and ratings about some specific places. Obviously this implies that they need to be able to register and login to the platform. Users should also be able to search for a specific place by a keyword. And since OPR is all about sustainability-related reviews, the platform offers the possibility to rate establishments based on multiple criteria. The platform also has a mobile app, and this implies that the web based app should provide an API to supply the mobile app with the data.

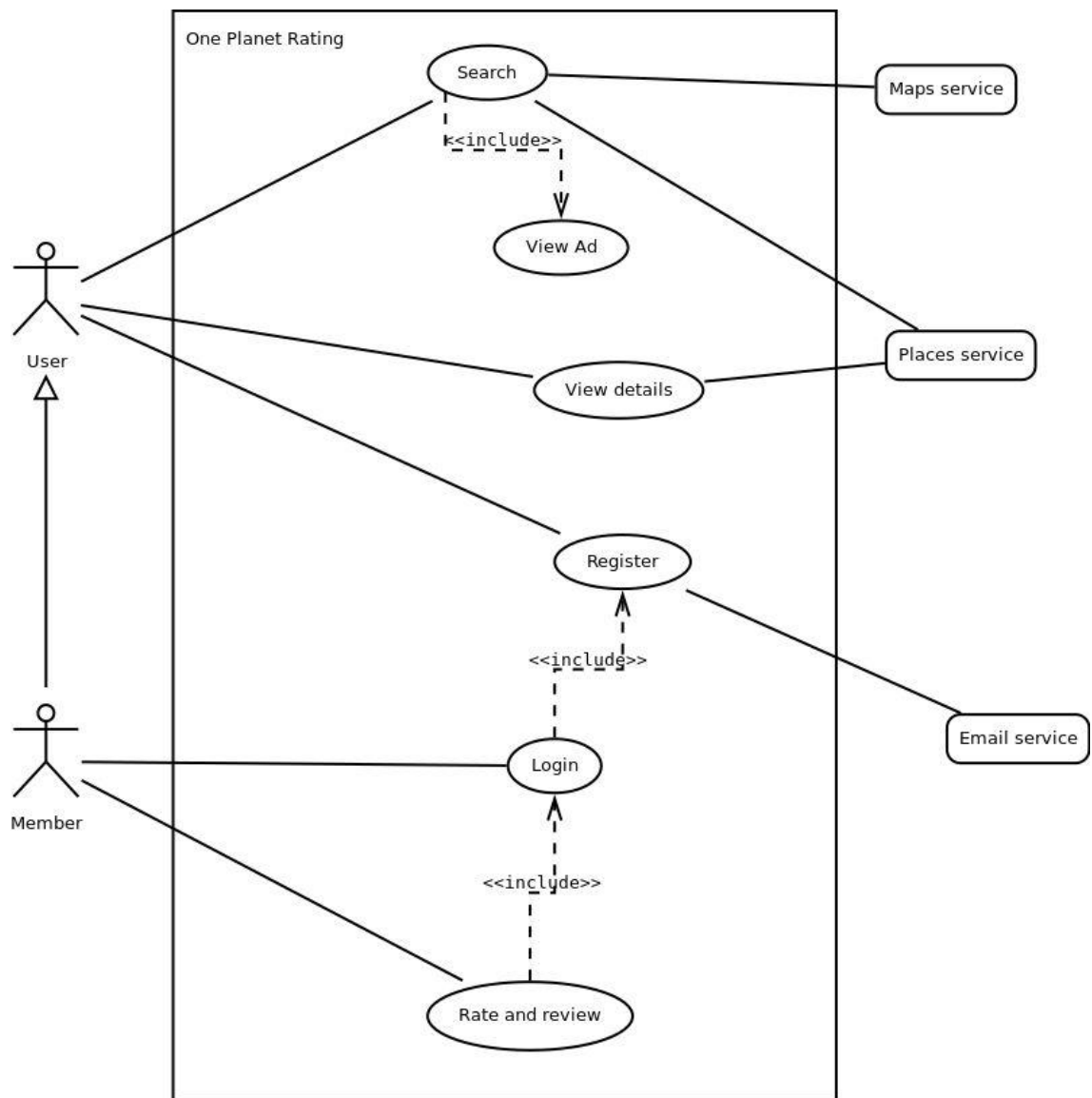


Figure 2.1: general usecase diagram

As shown in the use case diagram, any user can use the app to search and view the details of a place. The search functionality pulls data from Google places service, and shows a map using Google maps service. When a user registers, he becomes a member. During the registration process Gmail service is used to send a confirmation email. The member can then login to his profile and becomes able to rate and review places.

### 2.2.1 Register

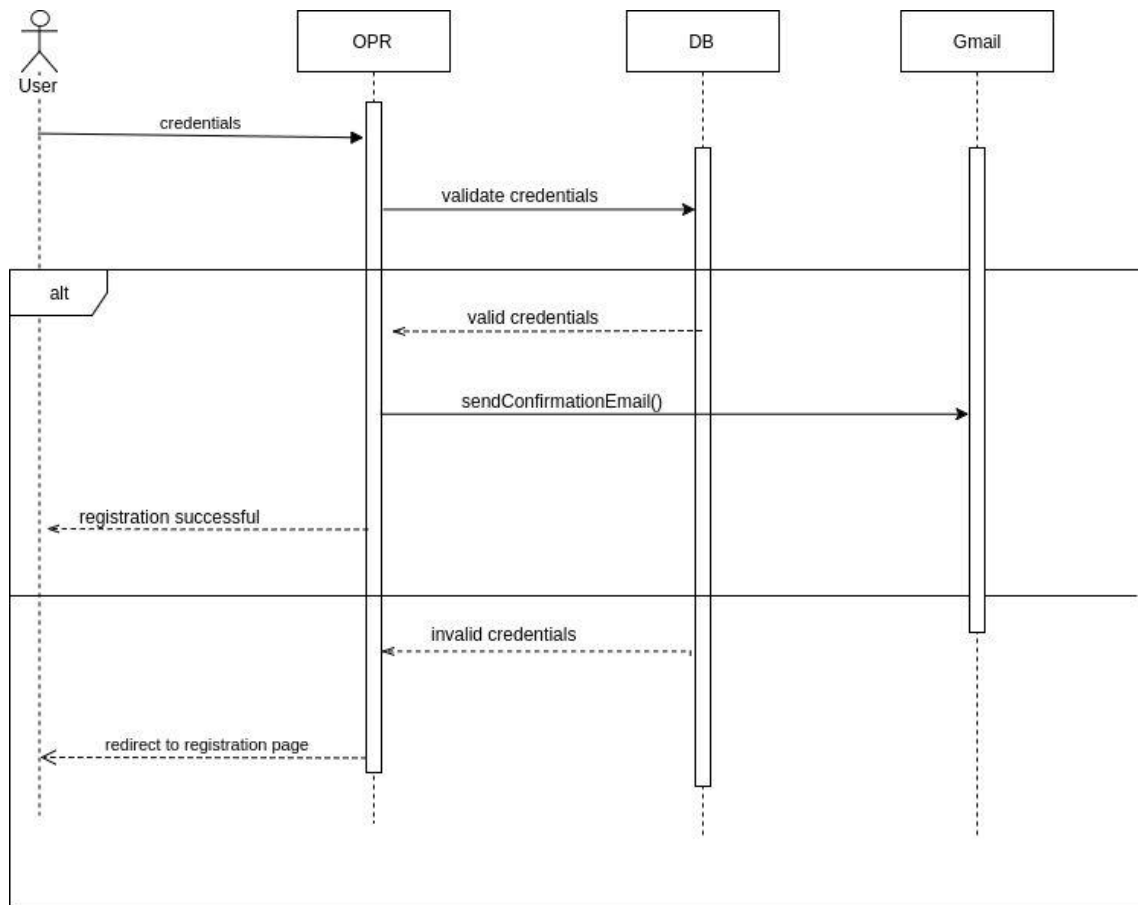


Figure 2.2: registration sequence diagram

Title	Registration
Author	Moetaz Ben Charrada
Version	1.0
Objectives	Allows users to register and create an account
Actors	User - OPR - Gmail
Pre-conditions	The user should be on the registration page The user must not be already registered.
Post-conditions	The user becomes registered (member)
Story	1. The user enters his username 2. The user enters his email 3. The user enters his password 4. The user re-enters his password 5. The user submits the form
Alternative story	The functionality is accessed through the mobile app instead of a browser.
Exceptional story	If the user enters an email that's already in the DB, the user will be prompted to enter a different email. If the user enters a password that's not compliant to the security constraints, he will be prompted to enter a different password

Table 2.1: Registration description

### 2.2.2 Login

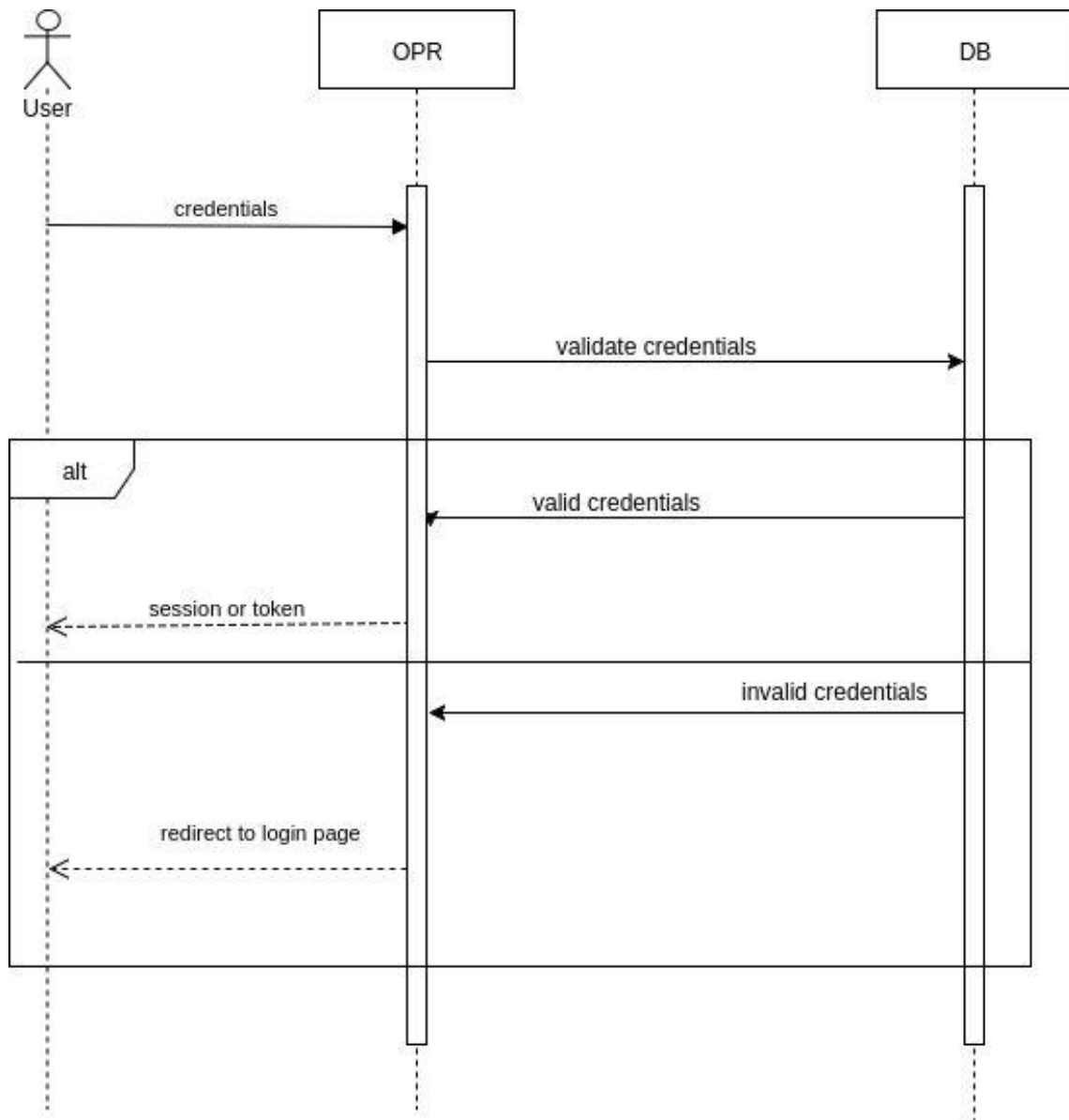


Figure 2.3: login sequence diagram

Title	Login
Author	Moetaz Ben Charrada
Version	1.0
Objectives	Allows users to login and access his account
Actors	User - OPR
Pre-conditions	The user should be a registered member. The user must not be already logged in.
Post-conditions	The user becomes authenticated. The user can then post reviews.
Story	1. The user enters his username 2. The user enters his password 3. The user submits the form
Alternative story	The functionality is accessed through the mobile app instead of a browser. (Through the REST API)
Exceptional story	If the user enters incorrect credentials, he will be prompted to try to login again.

Table 2.2: login description

### 2.2.3 Search

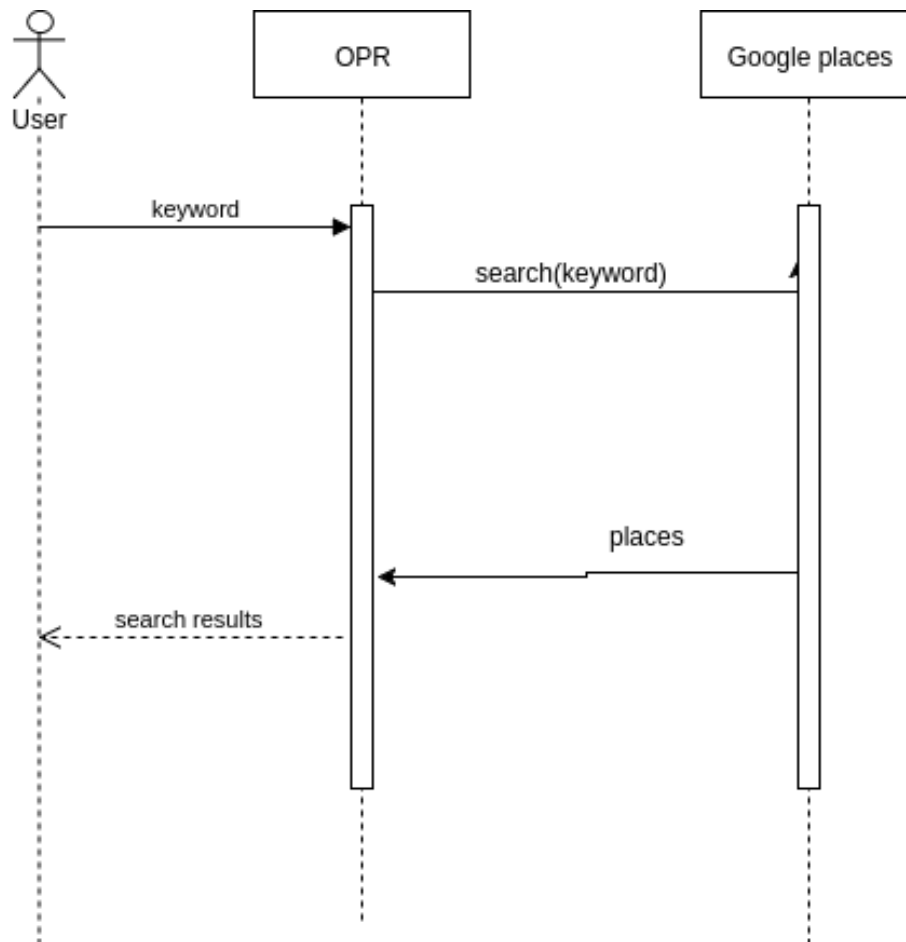


Figure 2.4: search sequence diagram

Title	Search
Author	Moetaz Ben Charrada
Version	1.0
Objectives	Allows users to search for a specific place.
Actors	User - OPR - Places service
Pre-conditions	The user should be on a page where there's a search bar.
Post-conditions	The user finds places related to his keywords, and may click to check the details of that place.
Story	1. The user enters keywords related to the place he's looking for 2. The user selects the type of place 3. The user submits the form
Alternative story	The functionality is accessed through the mobile app instead of a browser. (Through the REST API)

Table 2.3: search description



## 2.2.4 View details

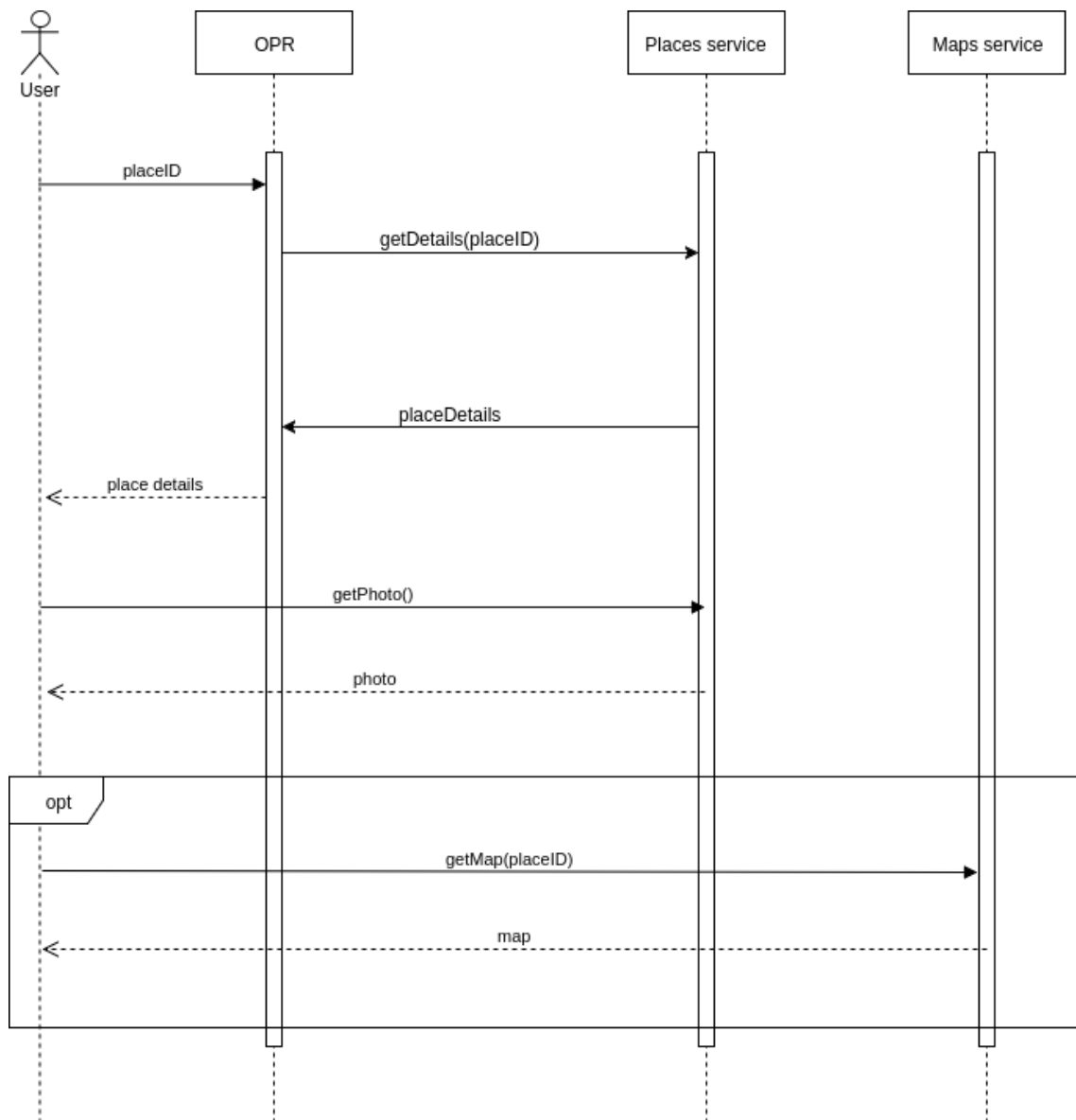


Figure 2.5: view details sequence diagram

Title	View details
Author	Moetaz Ben Charrada
Version	1.0
Objectives	Allows users to view the details and reviews of a specific place
Actors	User - OPR - Places service - Maps service
Pre-conditions	The user should have searched for the place before.
Post-conditions	The user sees the details and reviews of the place and can possibly post a review about it.
Story	<ol style="list-style-type: none"><li>1. The user clicks on one of the places he found in the search view.</li><li>2. OPR app will then get the details of that place from Google places service, get the map from Google maps service.</li><li>3. The data is combined with the local reviews' data and is sent back to the user.</li></ol>
Alternative story	The functionality is accessed through the mobile app instead of a browser. (Through the REST API)

Table 2.4: view-details description

### 2.2.5 Review a place

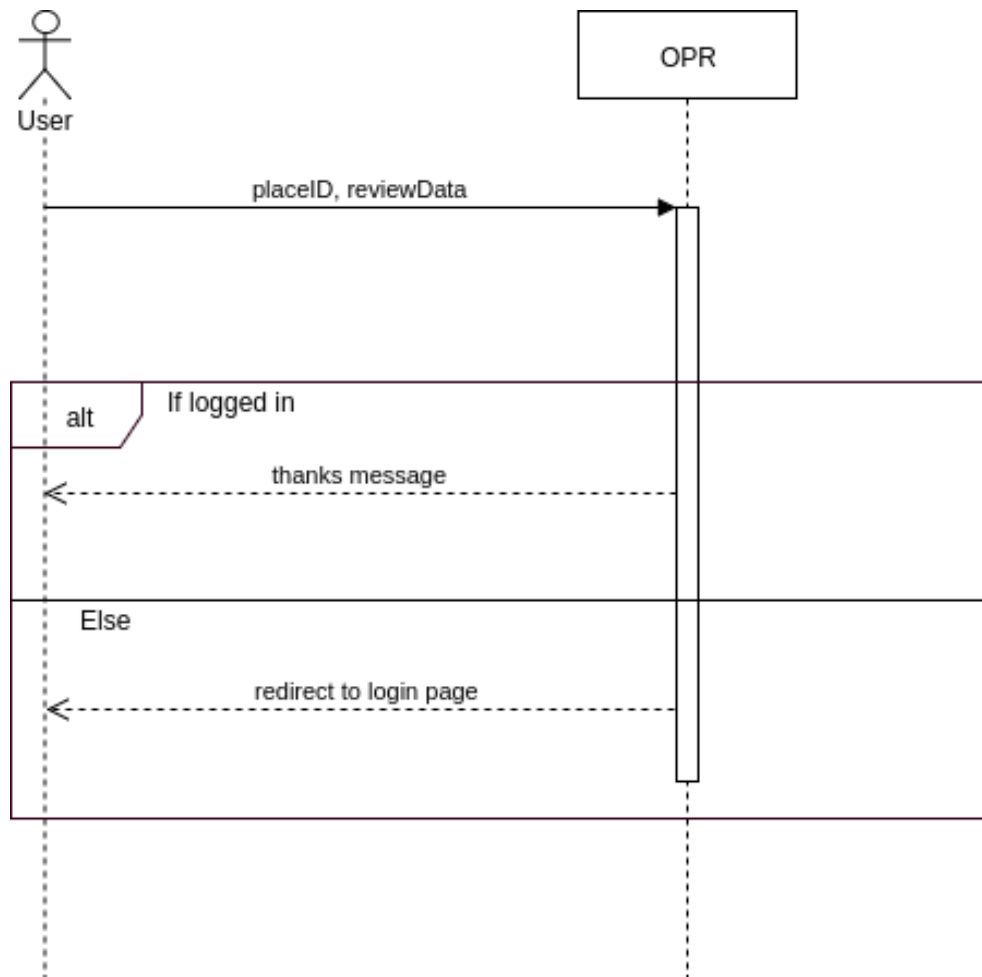


Figure 2.6: review a place sequence diagram

Title	Review a place
Author	Moetaz Ben Charrada
Version	1.0
Objectives	Allows users to post a review about a specific place
Actors	User - OPR
Pre-conditions	The user should be authenticated to be able to post a review.
Post-conditions	The user creates a review. The review becomes visible to the public.
Story	1.The user enters his review data 2.The user submits the form. 3.The review data is saved to the database. 4.A thanks message is shown to the user.
Alternative story	The functionality is accessed through the mobile app instead of a browser. (Through the REST API)
Exceptional story	If the user tries to post a review while he's not authenticated. He is redirected to the login page.

Table 2.5: review a place description

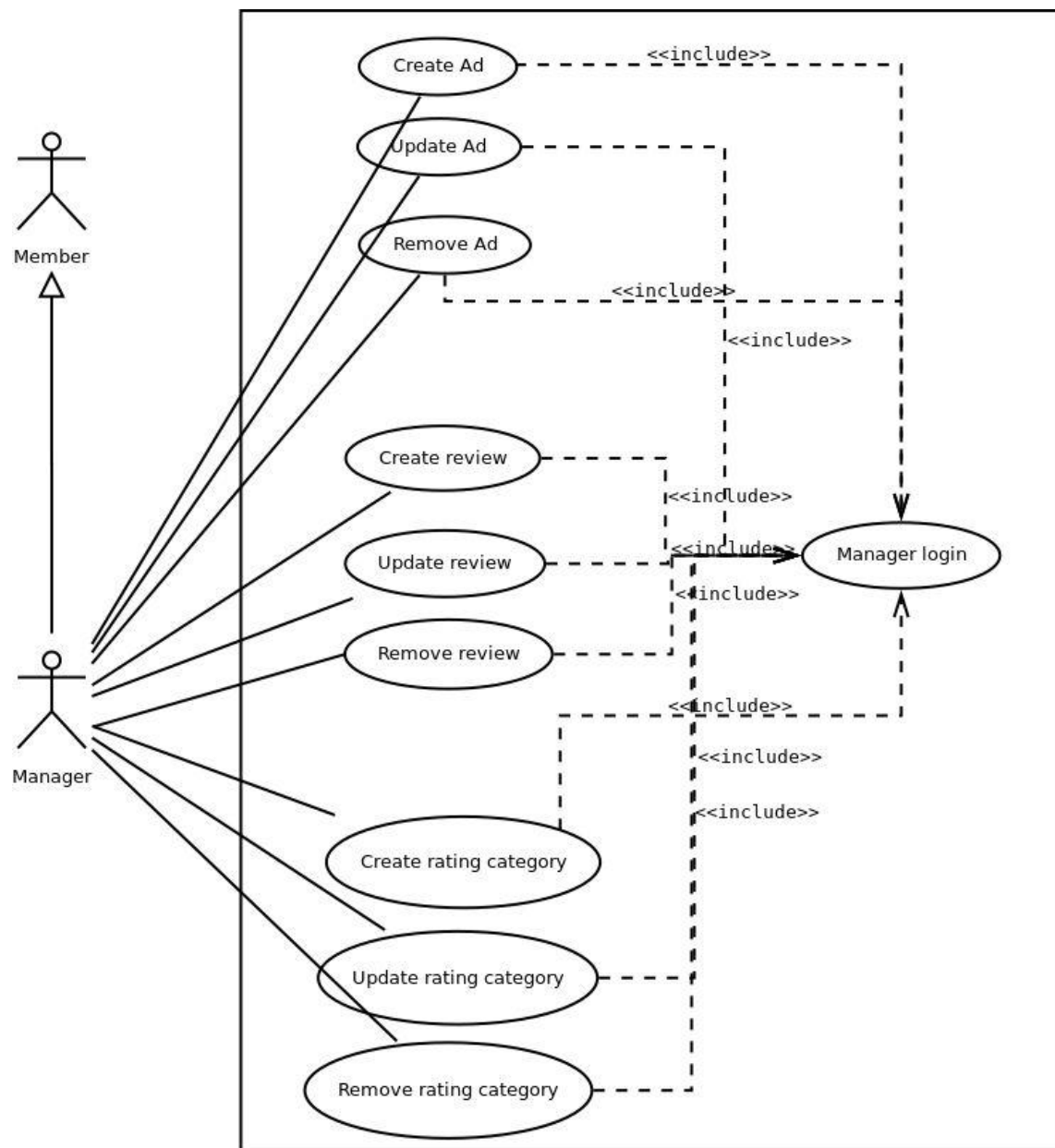


Figure 2.7: manager usecase diagram

A manager, is a member with additional privileges. An admin can create, update and delete: ads, reviews and rating-categories.

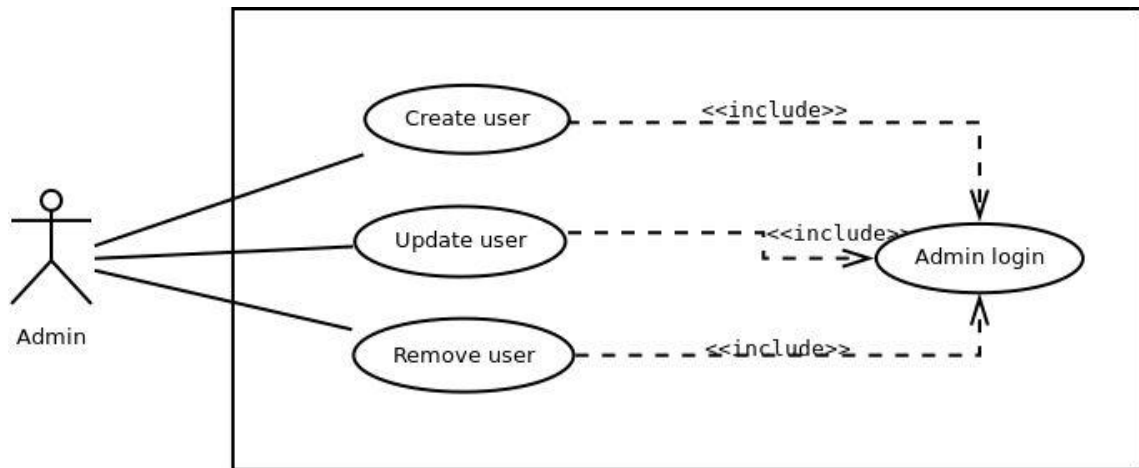


Figure 2.8: admin usecase diagram

## CHAPTER 3

# Design & architecture

In this chapter we'll go through the design of the app from the ground up. Since we're building the app from scratch we'll have to take into consideration both the physical and logical aspects of the app.

As for the physical architecture, the app is designed in a way that prioritizes security and makes it easily scalable.

In the logical side, the app is broken into smaller modules to facilitate maintainability and extensibility. During the design I aimed to make the modules independent to reduce coupling as much as possible, thus offering the possibility to easily change a module in the future, without touching to the other modules.

### 3.1 Physical architecture

The main parts of our physical architecture are: The server hosting the web server and static files, the server hosting the app and the business logic and The DB server.

For both the web server and the app server, we used an amazon EC2 instance running Ubuntu 16.04 LTS. AWS offers great support and scalability features.

The first server hosts the static files of the app (Images, CSS, JS) as well as the web server, in our case NGINX. The web server is configured to listen for the incoming requests, whether from a web browser or the mobile app. If the call is requesting a static file, NGINX will get the file using filesystem access and send it back to the client. If the call is requesting a function to be run in the app, NGINX acts as a

proxy and redirects the call to the app server via HTTP, then gets the response and redirects it to the client.

The second server is hosting the app, and the app server. Our app server is Gunicorn, it's a python WSGI server. That's where all the business logic is hosted and run. When a call only requires computing capabilities, this server will respond without needing any third parties. When the call requires external data, this server is responsible of acquiring that data.

Our sources of data are either Google's APIs or our own DB. If the data needed is from Google, the app just triggers an HTTP request to Google's API.

If, on the other hand, the data requested is on our DB, the app has to access the DB through a TCP connection and get that data. The DB server is an AWS RDS instance hosting a PostgreSQL DB.

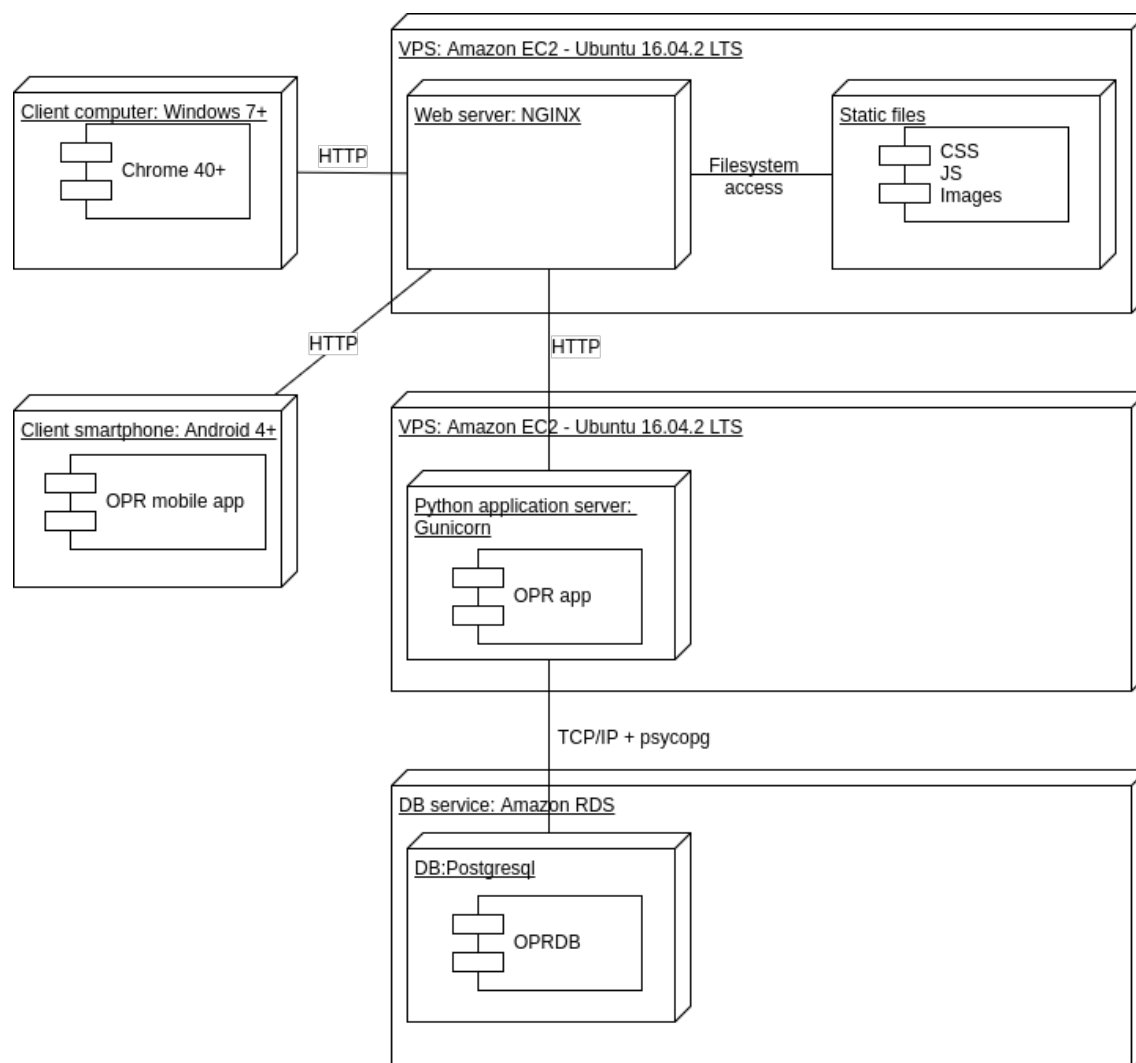


Figure 3.1: The app's physical architecture



## 3.2 Logical architecture

The architecture used is a mix of MVC + 3 Tiers.

### 3 Tiers

3 Tiers was used to separate the data, logic and the client from each other. Each of those three tiers is a separate, standalone entity. This provides more security, because if one of the tiers gets attacked the other tiers remain intact. This also provides some sort of interoperability because we can change one of the tiers without touching the other ones.

### MVC

MVC architecture was used in the server level. MVC architecture facilitates code visibility and maintainability because the architecture is well known and one can easily predict and understand the functionality of each of its components. MVC also has the advantage of loose coupling, which also offers interoperability as explained in the example of 3 Tiers above.

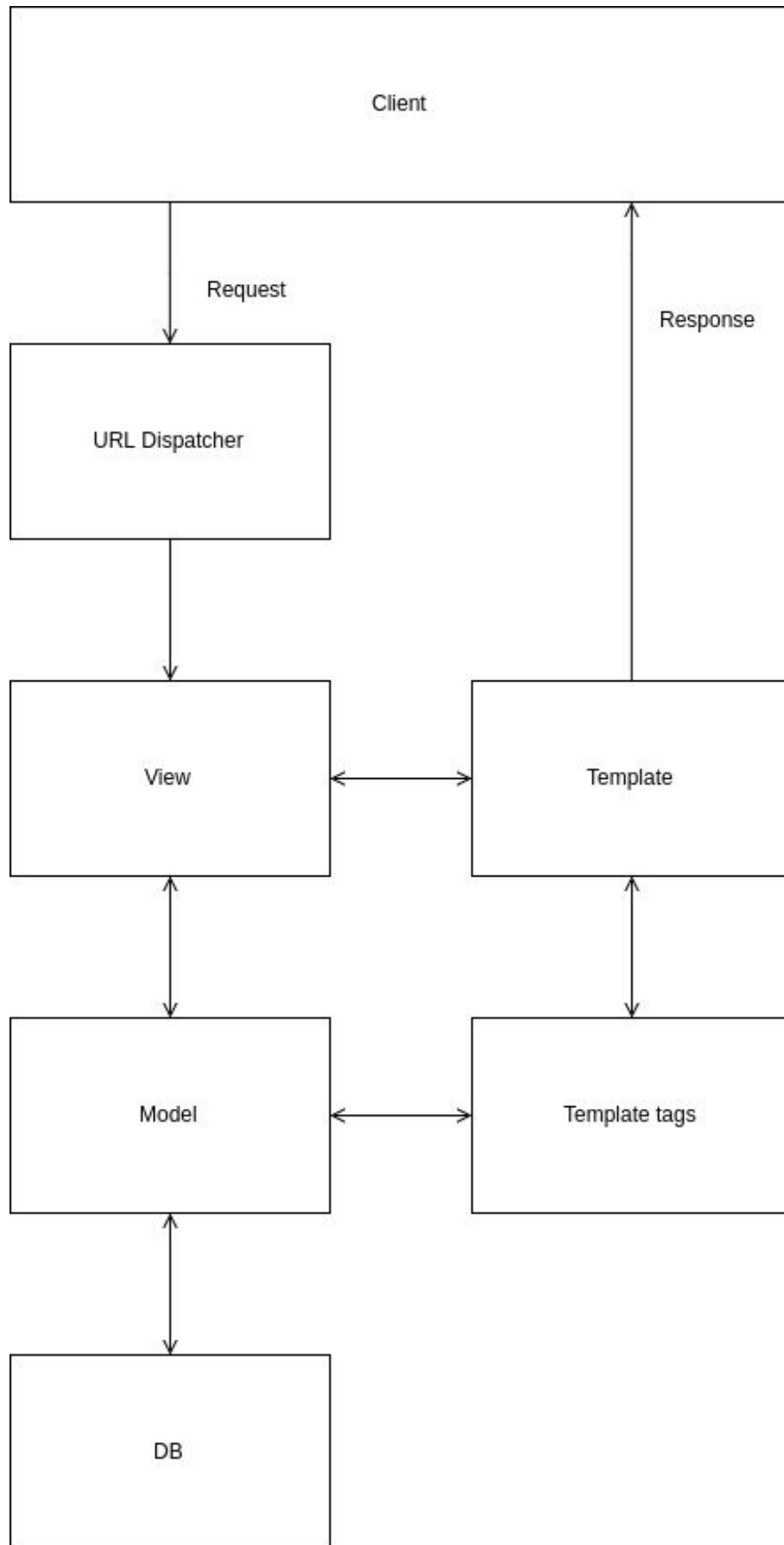


Figure 3.2: The app's logical architecture

### 3.3 Modular architecture

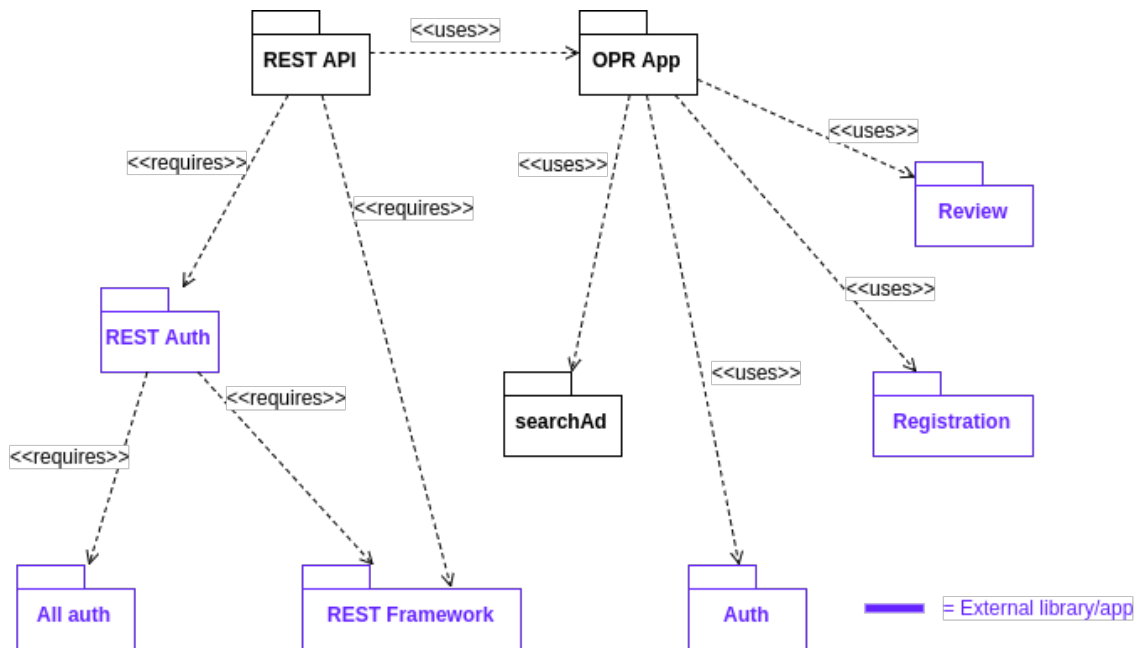


Figure 3.3: The app's modular architecture

During my work on the project I tried my best to reuse existing code instead of re-writing what already exists and is ready to use. Thus Most of the modules shown in the diagram are external libraries. My work consists of creating the following modules: OPR App, REST API and searchAd.

### 3.4 Modules

#### 3.4.1 searchAdModule

This is a standalone library that offers the possibility to create and show ads either from a local source (the user's DB) or from an external source (like Google Adsense).

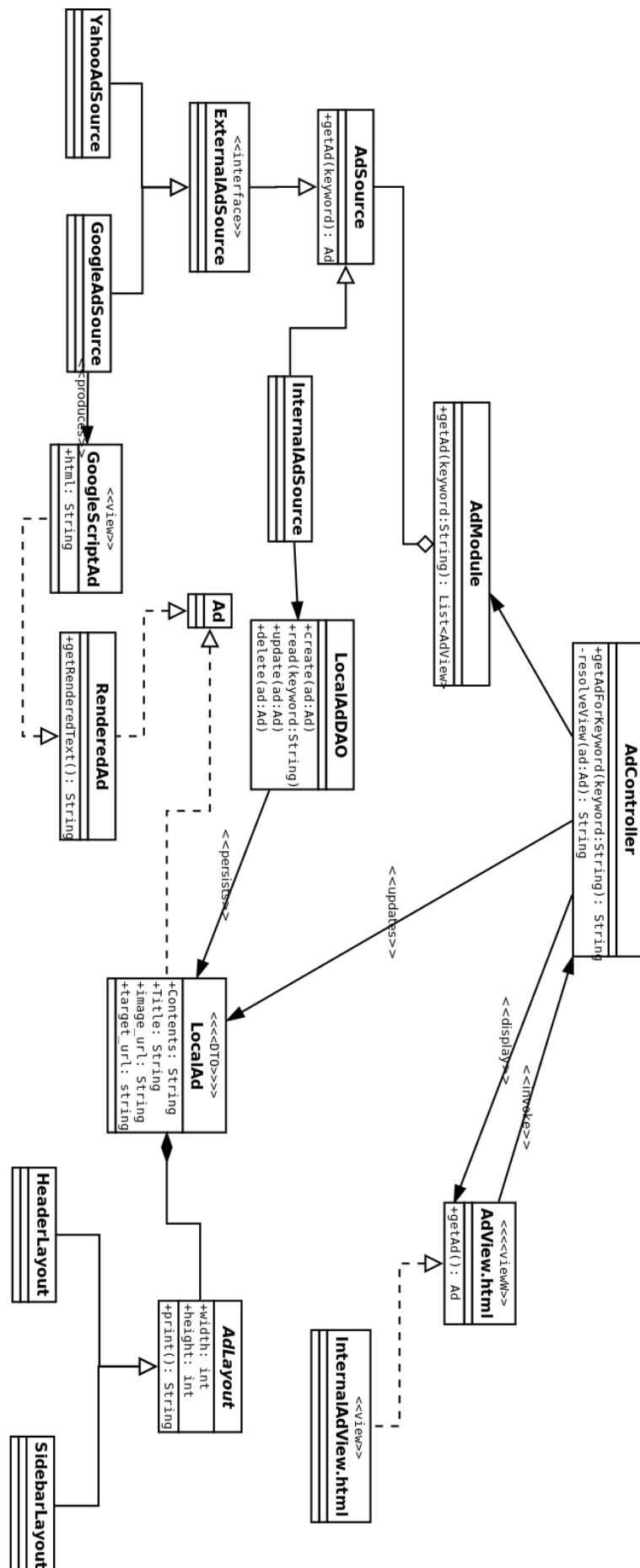


Figure 3.4: searchAd class diagram

The searchAd module is designed following the MVC architecture. The AdController is the entrypoint for the module, it allows the user to get an ad based on a keyword. It also contains a private method `resolveView()` that shows the ad in html format to be directly injected in the view.

The ExternalAdSource allows to get ads from external sources like Google AdWords. It's made in a way to make it easily extensible with other external sources. In fact all we have to do is create a new class inheriting the ExternalAdSource class and containing the url to the ads service.

In case the ads are not pulled from an external source, they're stored in the local database as LocalAd. The LocalAd model contains a title, content, target\_url and image\_url.

### 3.4.2 OPR app module

this is the main app, it contains the business logic for searching and rating places. It's also where most of the other libraries are integrated and linked together.

#### Search

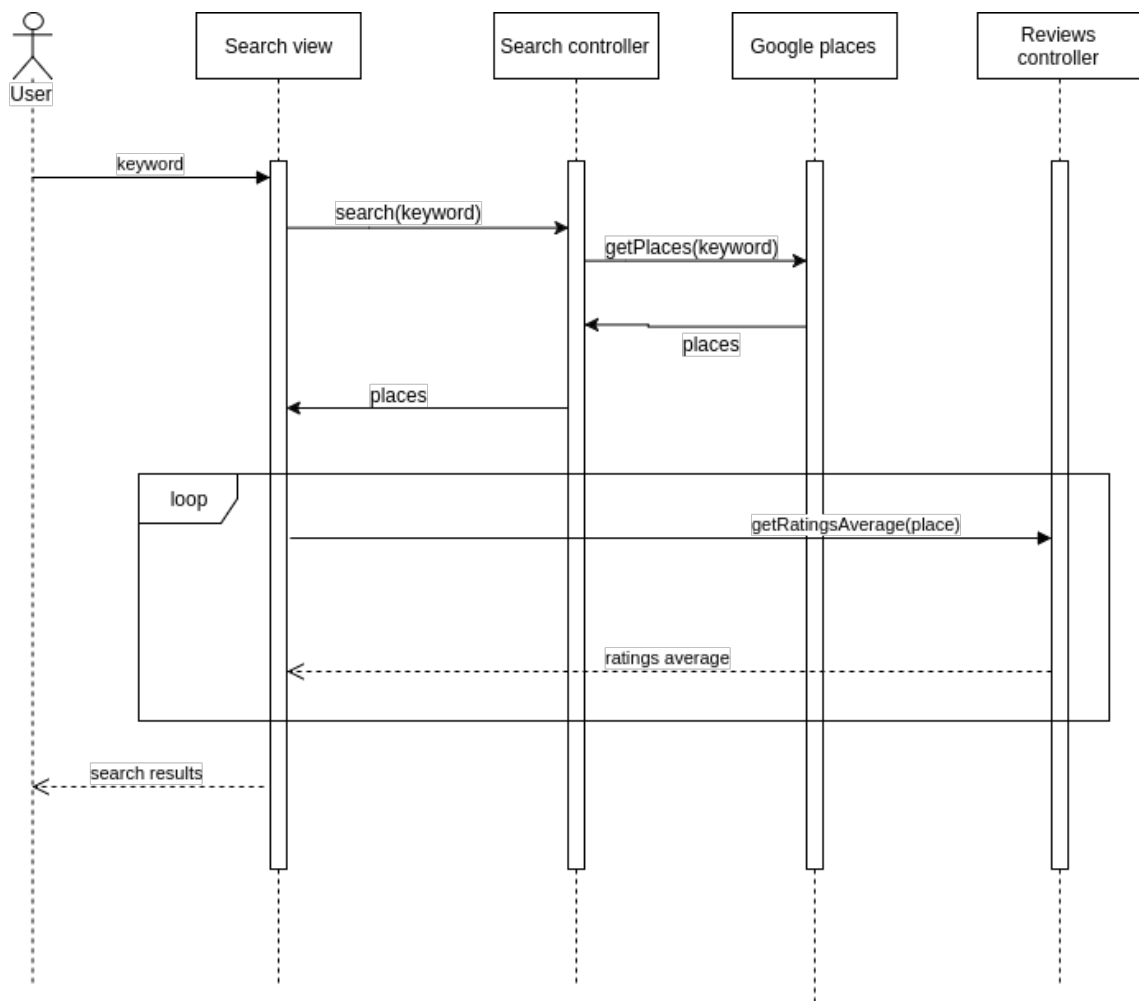


Figure 3.5: search sequence diagram

## View details

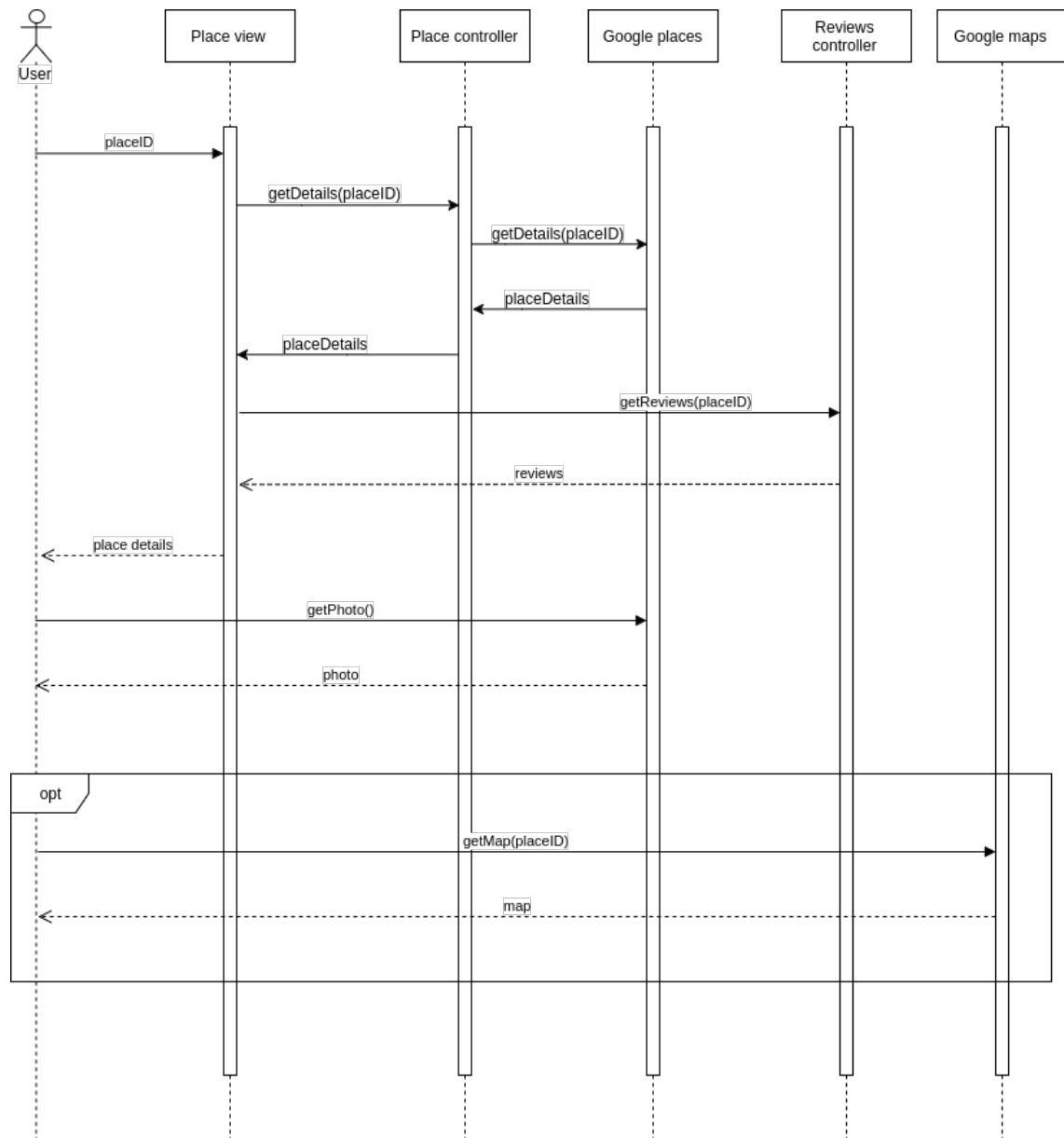


Figure 3.6: view-details sequence diagram

## Review a place

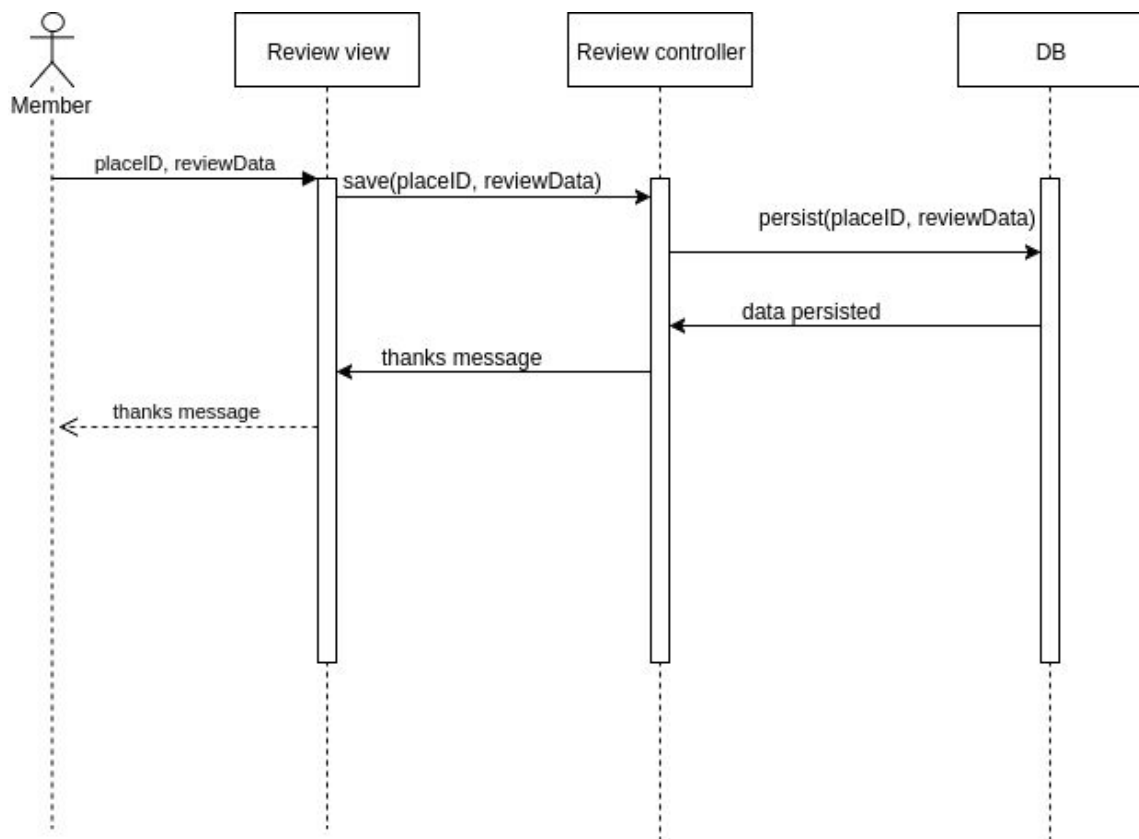


Figure 3.7: review a place sequence diagram

Note that a user can only post a review if he is logged in. The process is explained in the following interaction diagram:



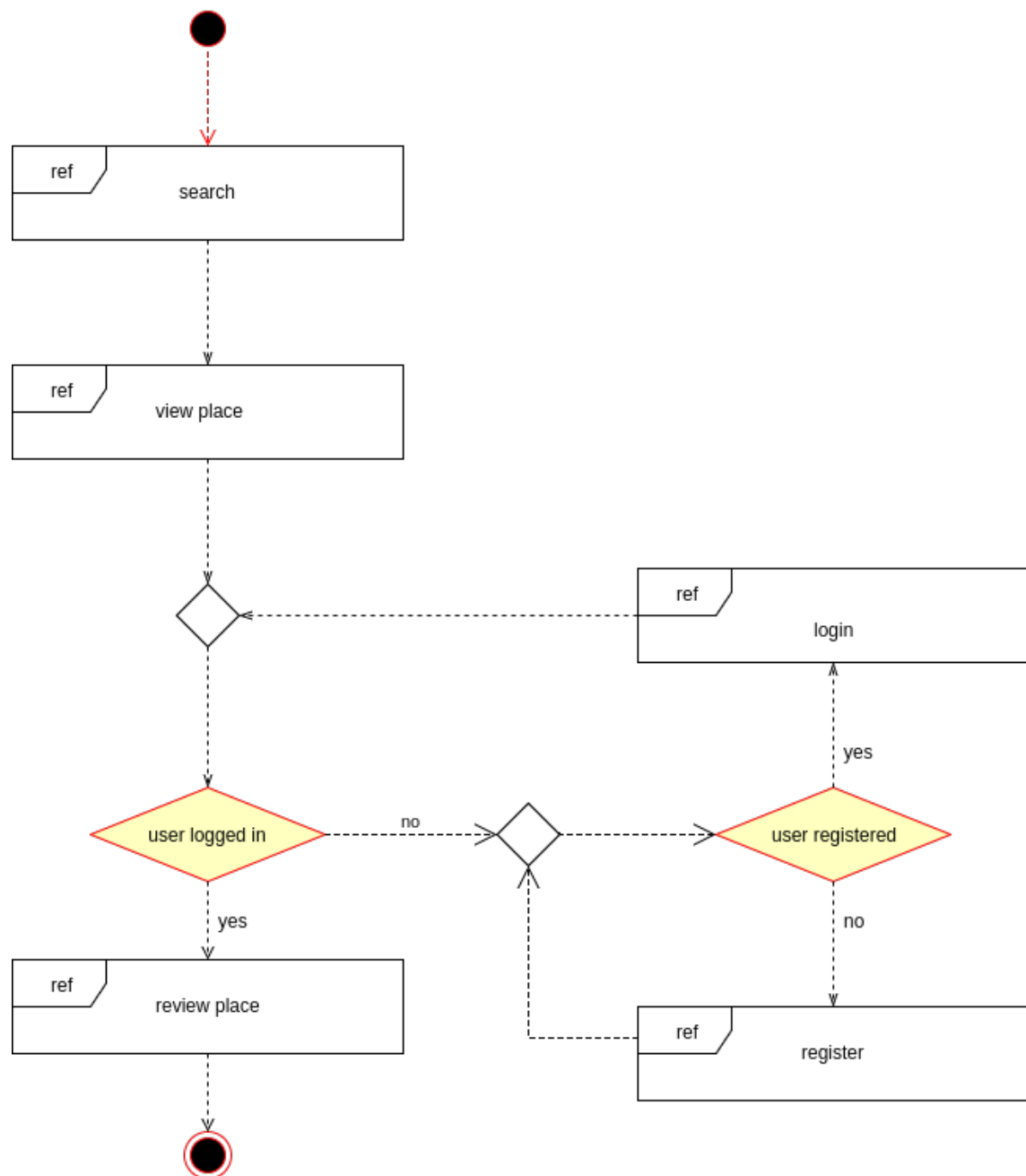


Figure 3.8: interaction overview diagram

### 3.4.3 REST API module

This is a sort of a wrapper for OPR App that provides a REST API interface, to be used by the mobile app.

For the REST API Module, I worked using the backwards approach; I started by writing the documentation of the functionalities first.

The services provided by the API are:

- Login
- Register
- Logout
- Search for places
- Get place details
- Get reviews
- Add review

The functions that require the transfer of critical data were designed to use the HTTP POST method because it's more secure and it is the standard for this type of operations, for instance we used POST for the authentication and registration functions. We also chose HTTP POST for posting reviews, because it contains the authentication token, which is critical data, and it contains the review's text, whose size might exceed the GET data size limit.

As for the other functionalities we used HTTP GET methods. One of the benefits of using GET here is that you can share and bookmark links easier when the data is included in the URL.

### **Login**

Authenticate the user with the system and obtain the `auth_token`

## Request

Method	URL
POST	api/accounts/login/

Type	Params	Values
POST	username	string
POST	password	string

## Response

Status	Response
200	<pre>{   "auth_key": &lt;auth_key&gt; }</pre> <p><b>auth_key</b> (<b>string</b>) - all further API calls must have this key in headerx</p>
401	<pre>{"error": "Incorrect username or password."}</pre>
500	<pre>{"error": "Something went wrong. Please try again later."}</pre>

## Register

Create and add a new user to the app

## Request

Method	URL
POST	api/accounts/register/

Type	Params	Values
POST	username	string
POST	Password	String
POST	email	string

## Response

Status	Response
200	<pre>{   "status": "success" }</pre>
500	<pre>{"error": "Something went wrong. Please try again later."}</pre>

## Logout

Log out the current user

## Request

Method	URL
POST	api/accounts/logout/

Type	Params	Values
POST	auth_key	string

## Response

Status	Response
200	{ "status": "success" }
500	{"error": "Something went wrong. Please try again later."}

## Search for places

Search for hotels, restaurants or activities

## Request

Method	URL
GET	api/search/

Type	Params	Values
GET	type	string
GET	keyword	string

Example:

api/search/?type=hotel&type=restaurant&keyword=stockholm

## Response

Status	Response
200	<p><b>Response will be an object containing the list of places with their id, name, average rating, number of ratings, image.</b></p> <pre>{   "places": [     {       "place_id": "qsdf6r45",       "image":         "https://maps.googleapis.com/maps/api/place/photo?maxheight=1000&amp;photoreference=v2d5qsd",       "name": "Radisson Blu Stockholm",       "avg_rating": "3.7",       "ratings_count": "125"     },     {       "place_id": "fsqd1qs887",       "image":         "https://maps.googleapis.com/maps/api/place/photo?maxheight=1000&amp;photoreference=g88s9d",       "name": "Marriott Stockholm",       "avg_rating": "4.1",       "ratings_count": "354"     }   ],   "places_count": 2 }</pre>
500	<pre>{ "error": "Something went wrong. Please try again later." }</pre>

### Get place details

Get the details of a specific place using its `place_id`

## Request

Method	URL
GET	api/place/

Type	Params	Values
GET	place_id	string

## Response

Status	Response
200	<pre>{   "place_id": "qsdf6r45",   "image":     "https://maps.googleapis.com/maps/api/place/photo?maxheight=     00&amp;photoreference=v2d5qsd",   "name": "Radisson Blu Stockholm",   "avg_ratings": [     "general": "3.7",     "environmental": "3",     "social": "2",     "cultural": "5",     ...   ],   "ratings_breakdown": [     ["Very poor", 0, 0.0],     ["Poor", 0, 0.0],     ["Acceptable", 1, 25.0],     ["Good", 1, 25.0],</pre>

	<pre>["Outstanding", 2, 50.0]] ', "ratings_count": "125", "description":{   "address":"stockholm, sweden",   ... } }</pre>
500	<pre>{"error": "Something went wrong. Please try again later."}</pre>

In the `ratings_breakdown`, `["Outstanding", 2, 50.0]` means there is 2 votes for outstanding, which count for 50% of the total number of votes

In the `ratings_breakdown`, `["Outstanding", 2, 50.0]` means there is 2 votes for outstanding, which count for 50

### Get reviews

Get the reviews of a place using its `place_id`



## Request

Method	URL
GET	api/reviews

Type	Params	Values
GET	place_id	string
GET	page	number

Reviews will be split into pages containing X reviews each.

Example:

api/reviews?place\_id=qsdf6r45&page=1

## Response

Status	Response
200	<pre>{   "place_id": "qsdf6r45",   "page": "1",   "total_reviews_count": "1025",   "reviews_in_page": "50",   "reviews": [     {       "review_id": "12",       "author": {</pre>

	<pre>"User_id": "lmkjsqdf" "name": "John Doe", "image": "http://opr.com/user/1655/image.jpg" }, "date": "May 3, 2017, 2:15 p.m.", "review": {   "title": "clean, solar powered hotel",   "visit_date": "May 1, 2017",   "text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco",   "type": "business",   "ratings": {     "general": "4",     "environmental": "4",     "energy": "5",     "water": "1",     ...   } }, ...</pre>
500	<pre>{"error": "Something went wrong. Please try again later."}</pre>

### Add review

Add a new review to a place

## Request

Method	URL
POST	api/addreview

Type	Params	Values
POST	auth_key	string
POST	user_id	string
POST	place_id	string
POST	visit_date	string
POST	title	string
POST	text	string
POST	type	string
POST	general	number
POST	environmental	number
POST	energy	number
POST	...	

## Response

Status	Response
200	<pre>{   "status": "success" }</pre>
500	<pre>{"error": "Something went wrong. Please try again later."}</pre>

## 4.1 Technology choices

### Frontend

For the front-end I used HTML/CSS/JS with JQuery library because it facilitates basic functionalities like DOM manipulation.

I also used twitter's Bootstrap in order to use it's prebuilt components like the buttons. The app is responsive but has some responsiveness-bugs; I didn't fix these because of time constraints and because the front-end is being redesigned by a professional designer anyway.

### Backend

The programming language used is Python, with the framework Django.

After careful thought I went with this choice because of several factors, including but not limited to:

- Portability: this stack is cross-platform and can run on linux as well as windows
- Lots of tools out of the box: this choice offers:
  - Runtime + Web framework
  - Package manager (PIP)
  - A database to test with (sqlite)
  - A lightweight development server

- Unit testing library
- Integrated ORM
- Lots of libraries and ready-to-use components
- Abundant support and documentation
- Stability and reliability
- Scalability
- Great community

This boils down to creating a prototype quickly with this technology stack; we can get a lot done in little time. Timing is a crucial part of this project, especially in this phase (prototyping).

## Database

While comparing multiple databases, I ended up with two of the most popular databases to compare between: MySQL and PostgreSQL.

The following table shows some differences of how things work in MySQL vs. in PostgreSQL.

	MySQL	PostgreSQL
CREATE INDEX	Entire table is locked for writes	Entire table is locked for writes. But PSQL has "CREATE INDEX CONCURRENTLY" that permits the creation of index without locking the entire table (but it's slower)
Adding new column	Entire table data needs to be rewritten	Instantaneous
Connection model	One thread/connection: Easy to create but hard to monitor and manage	One process/connection: Easier to monitor and manage

Table 4.1: MySQL vs PostgreSQL

I decided to use PostgreSQL as a database because of the previous comparison and because it's well supported by Django.

This choice is in some way influenced by the previous choice.

## Server

I used an Ubuntu 16.04 LTS Virtual Private Server for hosting the project. Ubuntu has a great community and support, is reliable, and the Long Term Support (LTS) version remains maintained for 5 years, so we won't have to worry about upgrading the system for five years.

Ubuntu is based on Debian, which is one of the most stable Linux distributions ever. The provider of the VPS is Amazon Web Service (AWS). I used AWS because it offers a free tier during the first year of use, and because it has extensive capabilities beyond offering a simple VPS. One of the capabilities that interested me the most is the ease of scaling and setting up load balancers for the project.

## Web Server

I had the choice between Apache HTTP Server and NGINX.

Both of them are great web servers and have great communities and support, and both of them are open source as well.

I went with NGINX because it's better performance-wise and is taking over Apache.

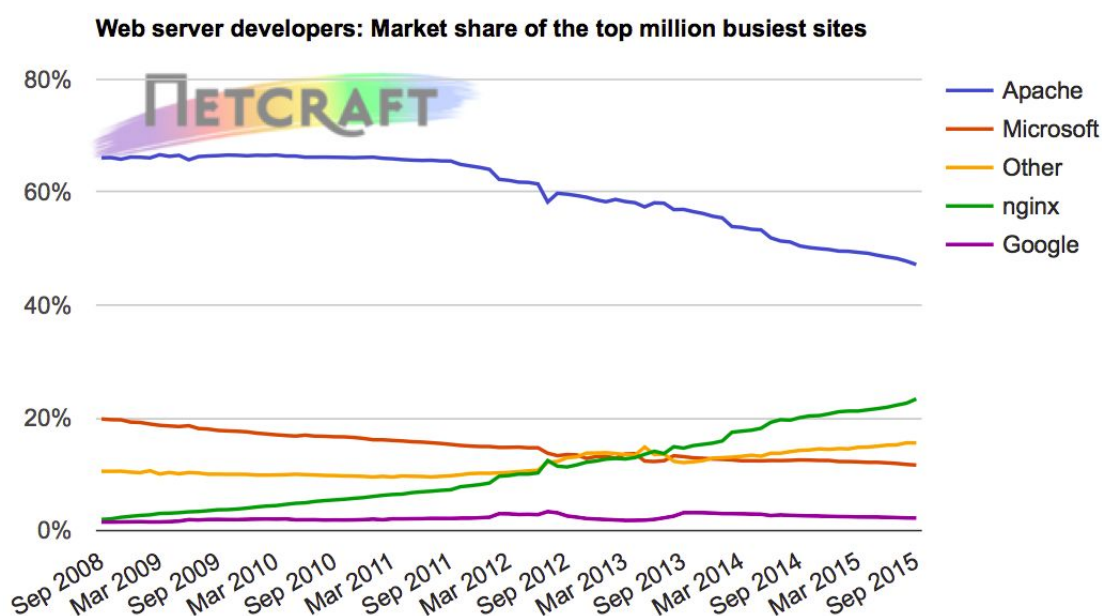


Figure 4.1: Chart showing how nginx's user base is growing while apache's is regressing

## Python Application server

I used Gunicorn as application server because it is the most popular and the best supported server.

## Additional Frameworks/libraries

### Django-rest-framework

Django-rest-framework (DRF) supports the creation of REST APIs on top of Django. This framework is django's standard tool to create REST APIs.

### Django-allauth

This is a library addressing authentication, registration, account management and social authentication

### Django-rest-auth

This library makes the previously mentioned library (allauth) accessible via REST API

### Django-review

This is a library facilitating the creation of reviews and ratings. I found multiple libraries offering reviews-related functionalities. I chose this library because it's kept updated and is backed by a company.

## 4.2 Screenshots

### 4.2.1 Web interface

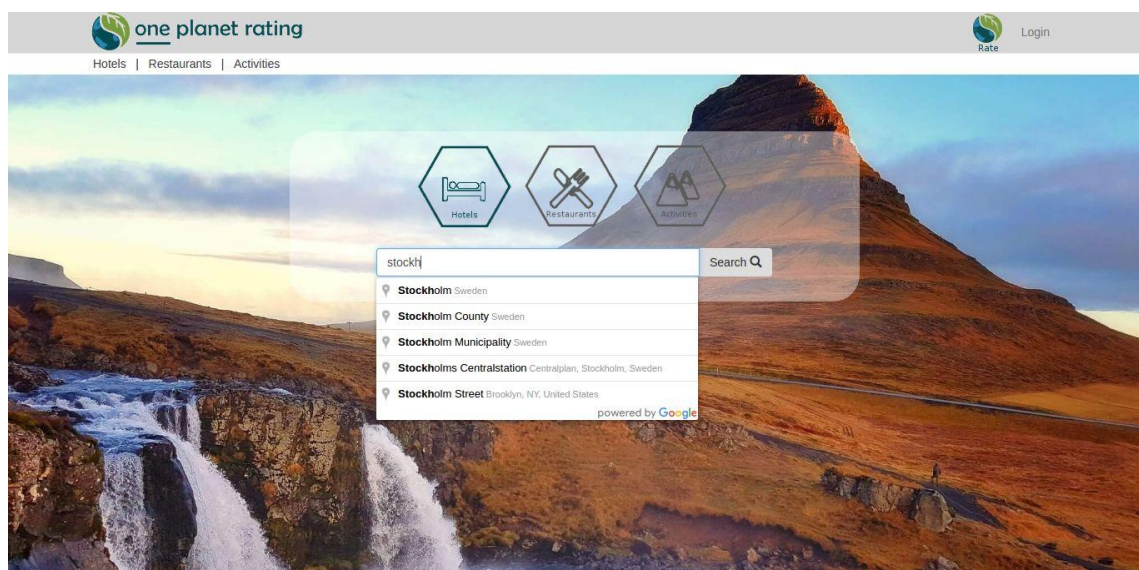


Figure 4.2: the landing screen of the app

When first accessed, the app shows the previous landing page. From there the user is able to search for a place using the search bar.

The user is also to access the login screen from there.

Other functionalities include listing the popular and trending hotels using the links in the top left corner, but these functionalities haven't been implemented yet.

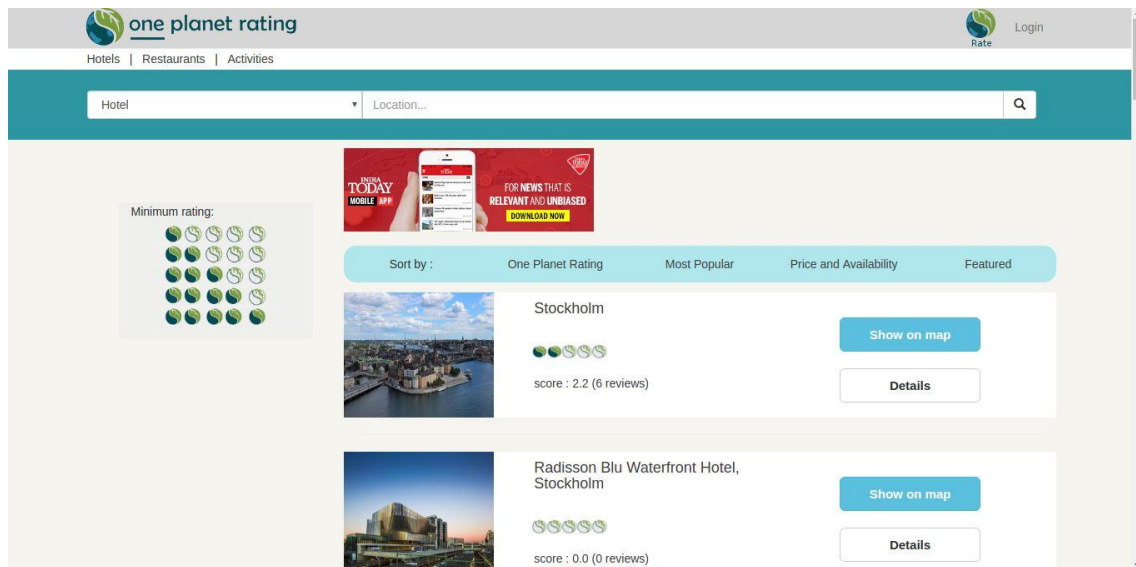


Figure 4.3: search screen

After the landing page, when a user searches for a place he will get redirected to this search screen which includes the results retrieved from Google's places service. The search results include the name and the picture (from google) as well as the number of reviews and the average score from the reviews posted on OPR.

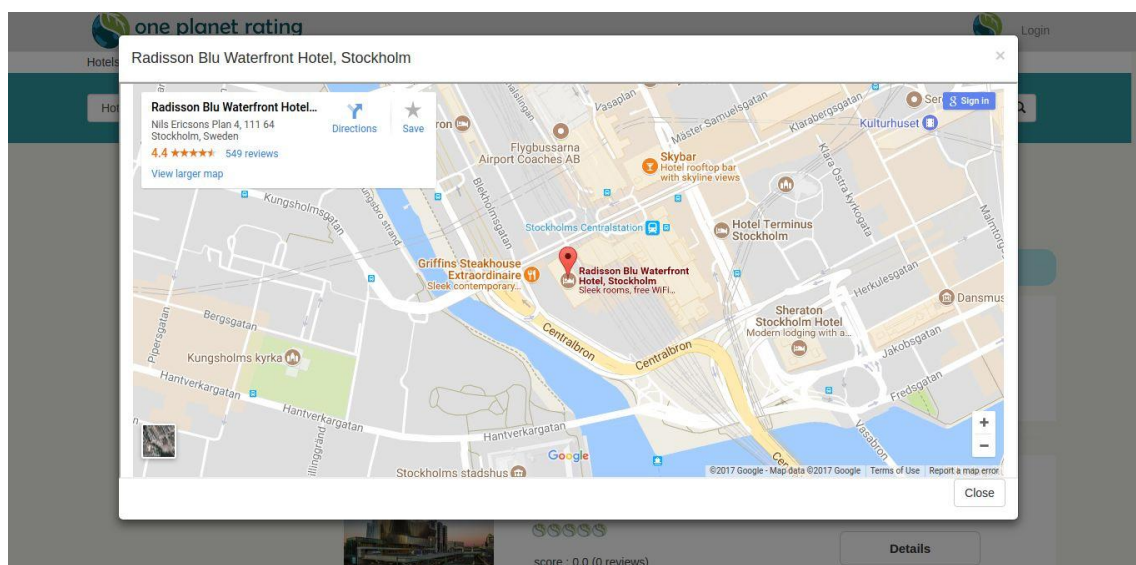


Figure 4.4: showing the location of a place on the map



From the search screen, the user is able to click the "show on map" button to see the selected place on Google maps. The map is shown in a modal view in order for the user not to quit OPR's site.

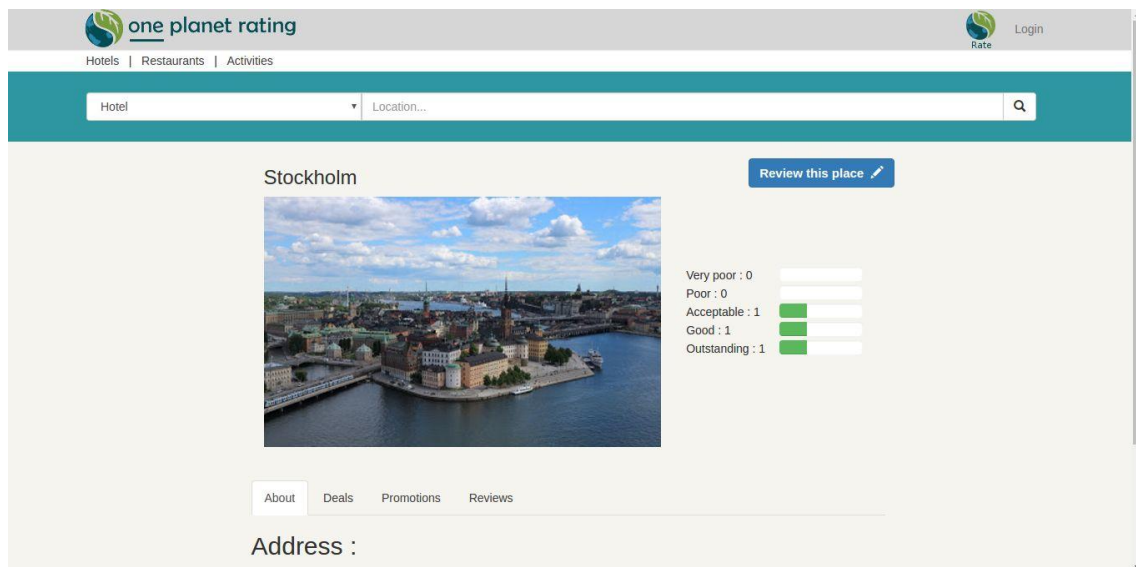


Figure 4.5: place details

When the user selects a place to see its details, he's shown this screen; it contains the name, picture and basic details of the place. This same screen also contains the reviews and a chart showing the numbers and proportions of ratings.

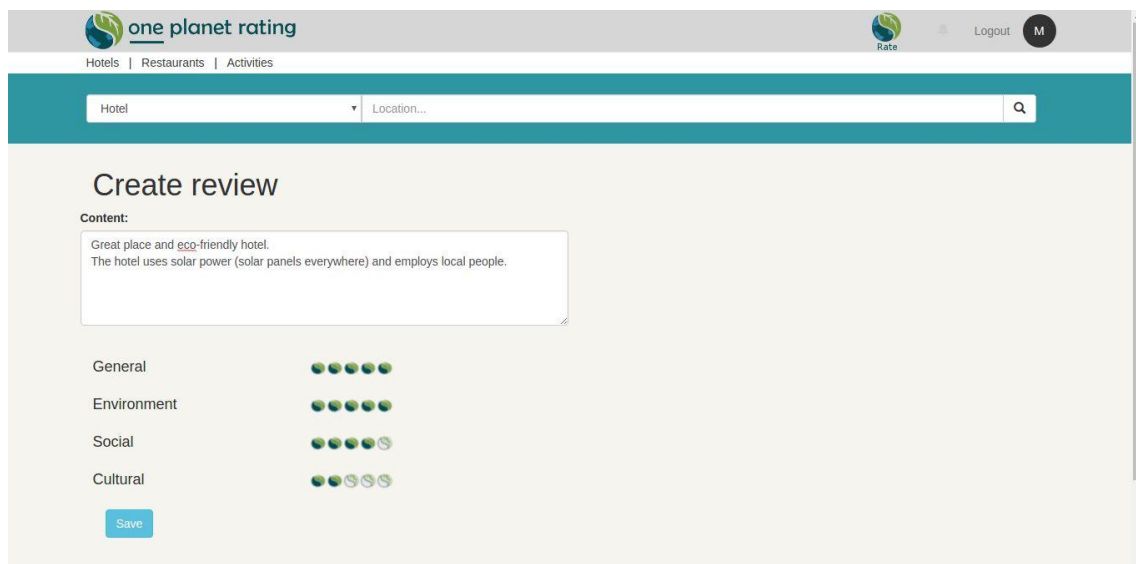


Figure 4.6: Creating a review

When the user chooses to create a review, the app first checks whether he's logged in.

If not, the user is redirected to the login screen, otherwise this form appears. The

form allows the user to enter a text review as well as ratings following specific criteria.

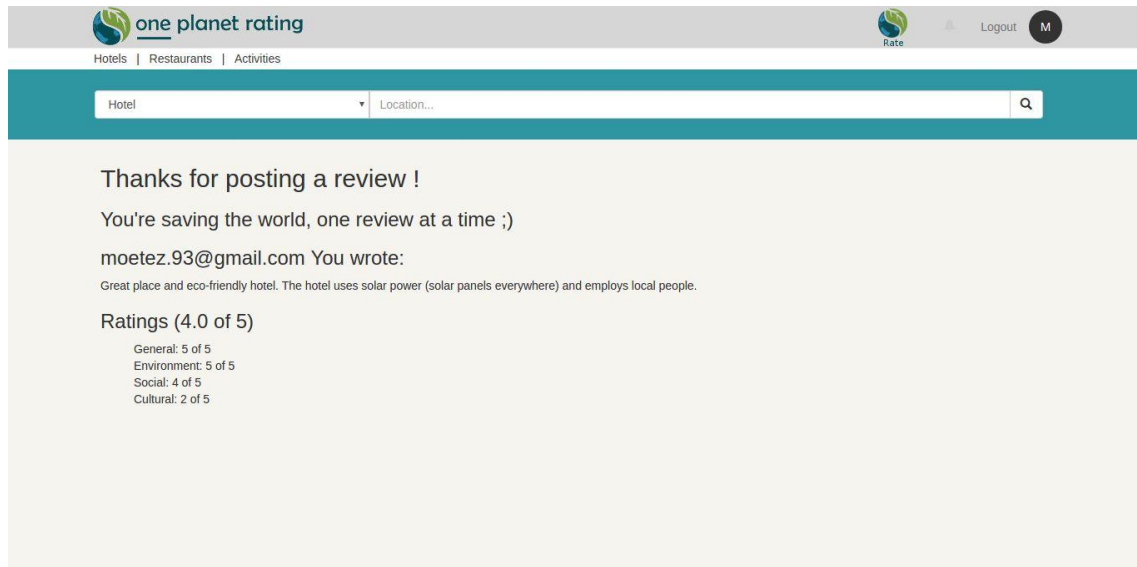


Figure 4.7: Thanks screen

After a member posts a review successfully, he's shown a thanks screen.

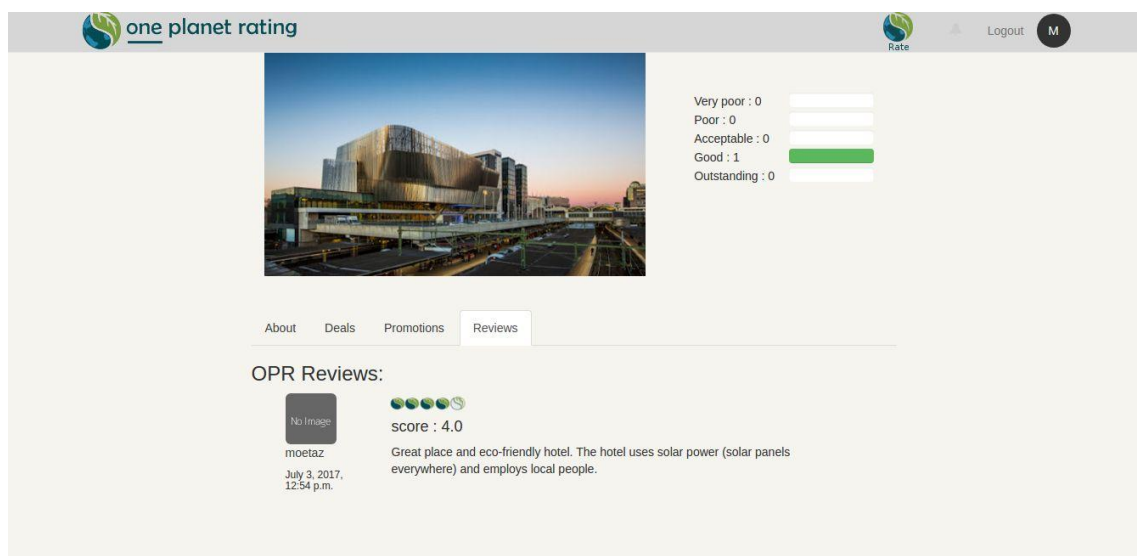


Figure 4.8: Showing a review

After a review is added, it is displayed as in this screenshot. The score, text, user-name and date are currently shown. Other features are being added like the user's photo and the categorized ratings.

## 4.2.2 API

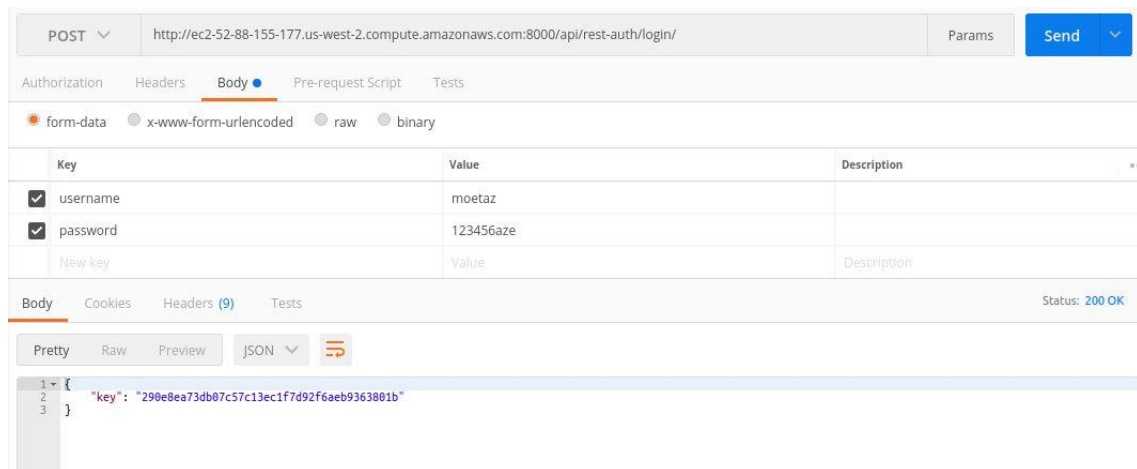


Figure 4.9: API Login

This screenshot shows an example of a login request through the REST API.

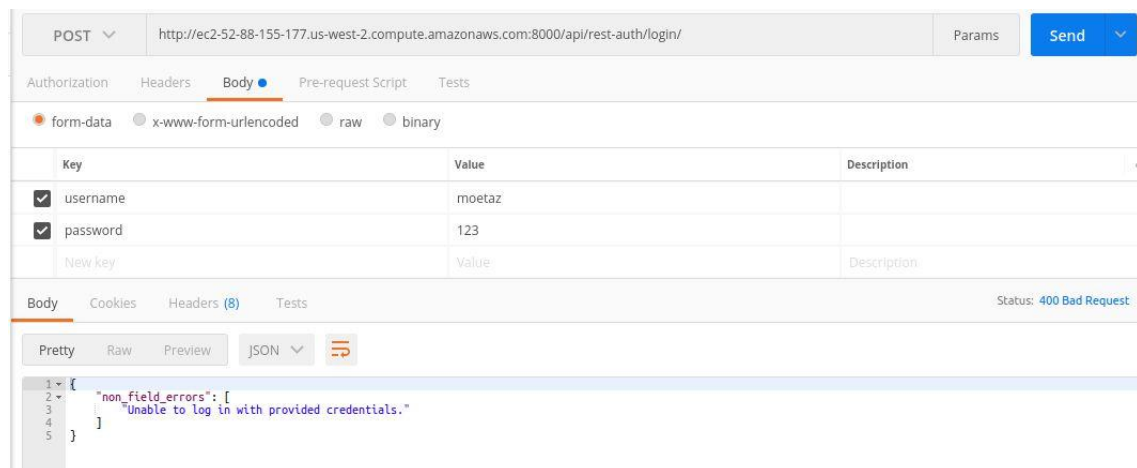


Figure 4.10: API Login failure

In case of failure (wrong credentials), an error message is sent back to the user.

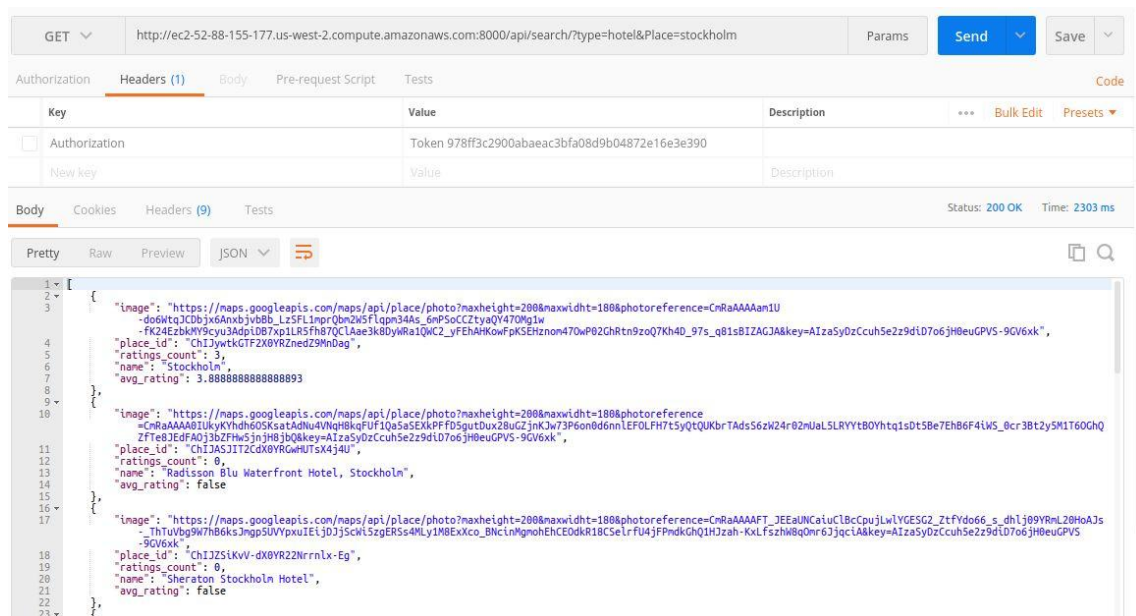


Figure 4.11: API search request

This screenshot shows the response to a search query. The query includes as parameters the type of place (hotel, restaurant..) as well as a keyword to specify the location.

The response contains the name, image, place\_id, ratings count, and the average rating.

The name and image are displayed as is. The ratings count and the average rating, however, are used to display the globes/stars next to each place in the results list. The place\_id is used to get the place's map or the place's details.

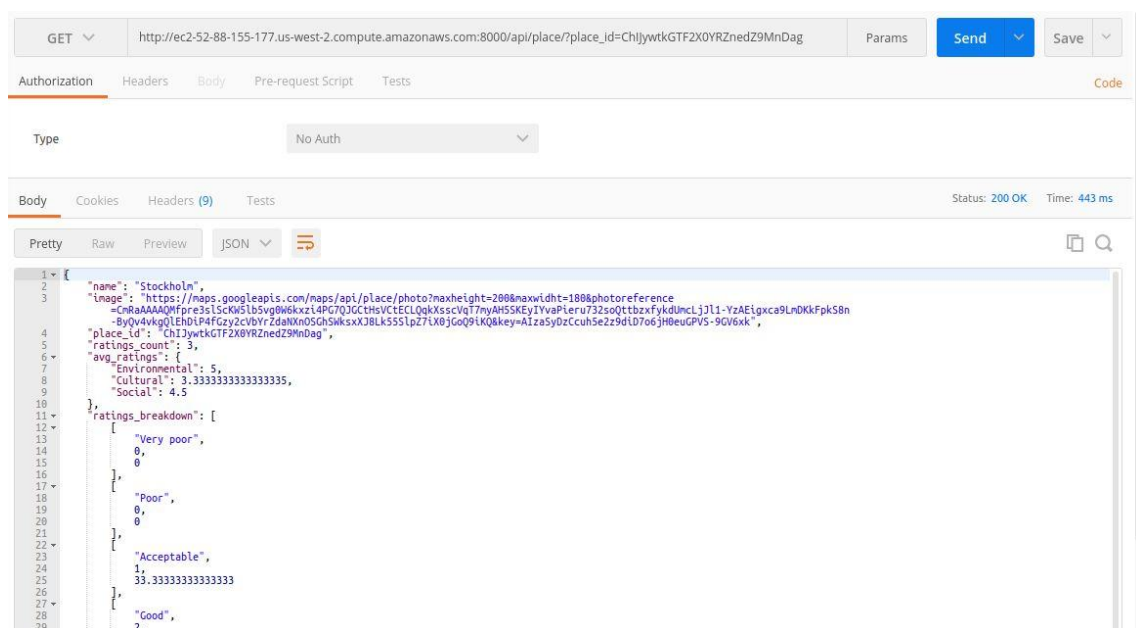


Figure 4.12: API search request

When requesting the details of a place, the request includes the `place_id` of the place. Obviously the response includes the name and image of the place, it also includes the number and averages of ratings. It also contains the ratings breakdown, which consists of the proportions of the ratings arranged by the number of stars ( 0 star = very poor, 1 star = poor...)

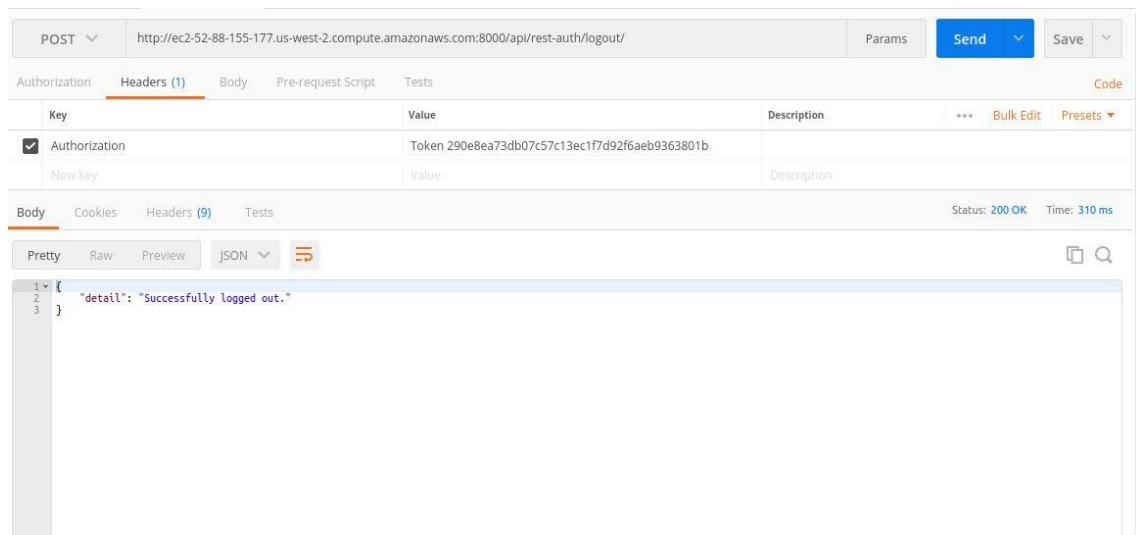


Figure 4.13: Logout API call

This screenshot shows a successful logout call. The authentication is based on an authorization token stored in the DB. After logging out, that token is deleted.

## 4.3 Deployment

### AWS EC2 Security setup

opr sg-ccb737b7 opr-WebServerSecurityGrou... vpc-d3c0f6b4 Enable connection from your IP			
Description	Inbound	Outbound	Tags
Edit			
Type	Protocol	Port Range	Source
HTTP	TCP	80	0.0.0.0/0
HTTP	TCP	80	::/0
PostgreSQL	TCP	5432	0.0.0.0/0
PostgreSQL	TCP	5432	::/0
Custom TCP Rule	TCP	8000	0.0.0.0/0
Custom TCP Rule	TCP	8000	::/0
SSH	TCP	22	0.0.0.0/0
SSH	TCP	22	::/0

Figure 4.14: AWS EC2 allowed inbound traffic

The screenshot shows the AWS Management Console for the 'opr' security group. The 'Outbound' tab is selected, displaying a table of rules. The table has four columns: Type, Protocol, Port Range, and Destination. The rules are as follows:

Type	Protocol	Port Range	Destination
PostgreSQL	TCP	5432	0.0.0.0/0
PostgreSQL	TCP	5432	:::0
All traffic	All	All	0.0.0.0/0
Custom TCP Rule	TCP	8000	0.0.0.0/0
Custom TCP Rule	TCP	8000	:::0
SMTP	TCP	25	0.0.0.0/0
SMTP	TCP	25	:::0

Figure 4.15: AWS EC2 allowed outbound traffic

A problem I encountered during the deployment is to get the server to reply for requests. In the beginning I was even unable to ssh into the VPS. It turned out, for security reasons AWS blocks all incoming and outgoing traffic.

## AWS RDS setup

The screenshot shows the AWS RDS console for the instance 'oprdb'. The instance is a PostgreSQL db.t2.micro in the 'saidalo' availability zone. The endpoint is 'oprdb.cr9mm1pfcs3d.us-west-2.rds.amazonaws.com:5432'. The configuration details are as follows:

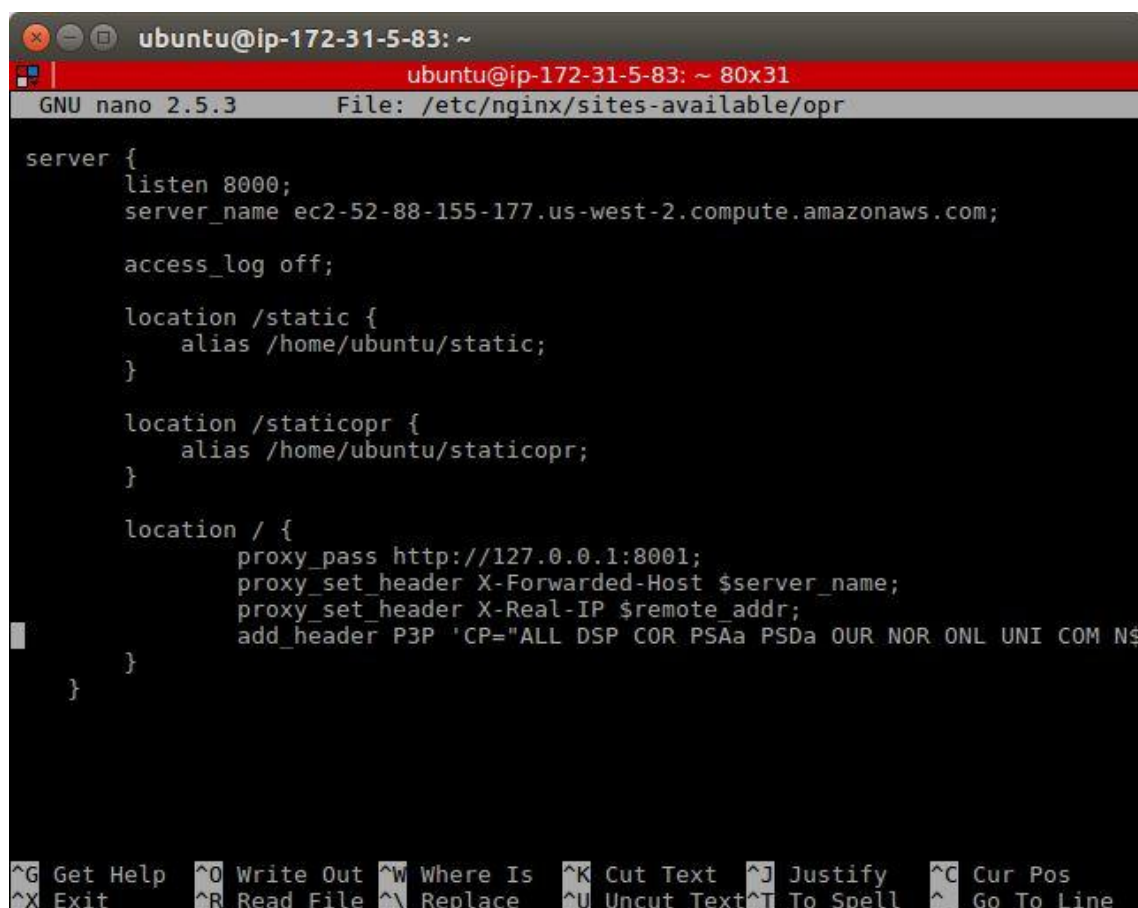
Configuration Details	Security and Network
ARN: arn:aws:rds:us-west-2:336225121527:db:oprdb	Availability Zone: us-west-2a
Engine: PostgreSQL 9.6.2	VPC: saidalo (vpc-d3c0f6b4)
License Model: PostgreSQL License	Subnet Group: default (Complete)
Created Time: May 30, 2017 at 7:07:46 PM UTC+1	Subnets: subnet-a7eba9c0, subnet-430d690a, subnet-7fd12724
DB Name: oprdb	Security Groups: opr-WebServerSecurityGroup-JZACGC3TR8KH (sg-ccb737b7) (active)
Username: oprdb	Publicly Accessible: Yes
Option Group: default:postgres-9-6 (in-sync)	Endpoint: oprdb.cr9mm1pfcs3d.us-west-2.rds.amazonaws.com
Parameter Group: default:postgres9.6 (in-sync)	Port: 5432
Copy Tags To Snapshots: No	Certificate Authority: rds-ca-2015 (Mar 5, 2020)
Resource ID: db-TABRVQ555JBUIWV3FH3Y4BJBIM	

Figure 4.16: AWS RDS setup

I used AWS RDS for hosting the database because security wise it's better approach to host the database separately from the app. Another advantage of AWS RDS is that it automatically creates backups of the DB.

First I hosted a local db for testing purposes then I moved the same DB to RDS.

## NGINX setup



```
ubuntu@ip-172-31-5-83: ~  
ubuntu@ip-172-31-5-83: ~ 80x31  
GNU nano 2.5.3 File: /etc/nginx/sites-available/opr  
  
server {  
    listen 8000;  
    server_name ec2-52-88-155-177.us-west-2.compute.amazonaws.com;  
  
    access_log off;  
  
    location /static {  
        alias /home/ubuntu/static;  
    }  
  
    location /staticopr {  
        alias /home/ubuntu/staticopr;  
    }  
  
    location / {  
        proxy_pass http://127.0.0.1:8001;  
        proxy_set_header X-Forwarded-Host $server_name;  
        proxy_set_header X-Real-IP $remote_addr;  
        add_header P3P 'CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM NS"  
    }  
}
```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos  
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^\_ Go To Line

Figure 4.17: NGINX setup

NGINX is the entrypoint to the app. It's also the web server used to serve the app. NGINX was setup to listen on port 8000 and to serve the static files like the images and css (practically every url pointing to /static). In case of non static files, NGINX redirects the request to gunicorn which is the python server that's serving the app.

## 4.4 Testing

For the development of searchAd module, which is a sort of standalone library, I also created unit tests for it.

The unit tests ensure that the code works correctly, and helps verify that the code didn't break after some changes. This provides better quality and saves more time.



The screenshot shows an IDE window with the following components:

- Top Bar:** opr - [~/Desktop/OP] (estimating..., 98%) Mon Jul 3 7:07 PM super
- Menu Bar:** File Edit View Navigate Code Refactor Run Tools VCS Window Salesforce Help
- Toolbar:** Includes icons for file operations, search, and running tests. A dropdown menu shows 'Test: searchAd.tests'.
- Project Explorer:** Shows the project structure with 'opr' and 'searchAd' folders. The 'tests.py' file is selected.
- Code Editor:** Displays the content of 'searchAd/tests.py'. The code includes imports from Django and searchAd, a 'CreateAdTestCase' class with a 'setUp' method, and several test methods: 'test\_keywords\_count', 'test\_ads\_count', 'test\_ad\_printing', and 'test\_ad\_retrieval'.
- Test Results Panel:** Shows the results of the test run. It indicates that all 4 tests passed with a total time of 97ms. The individual test results are:
  - searchAd.tests.CreateAdTestCase: 97ms
  - test\_ad\_printing: 24ms
  - test\_ad\_retrieval: 27ms
  - test\_ads\_count: 23ms
  - test\_keywords\_count: 23ms
- Output Console:** Displays the execution log, including the command used to run the tests, the start time, the creation of a test database, the test output (including HTML snippets and keyword/ad counts), and the final exit code 0.
- Bottom Bar:** Shows the status of the tests: 'Tests Passed: 4 passed (today 3:35 PM)'. It also includes a status bar with '31:1 LF UTF-8 Git: master'.



I also tested the performance of the app (load speed) using google's Pagespeed Insights, and I got some amazing results:

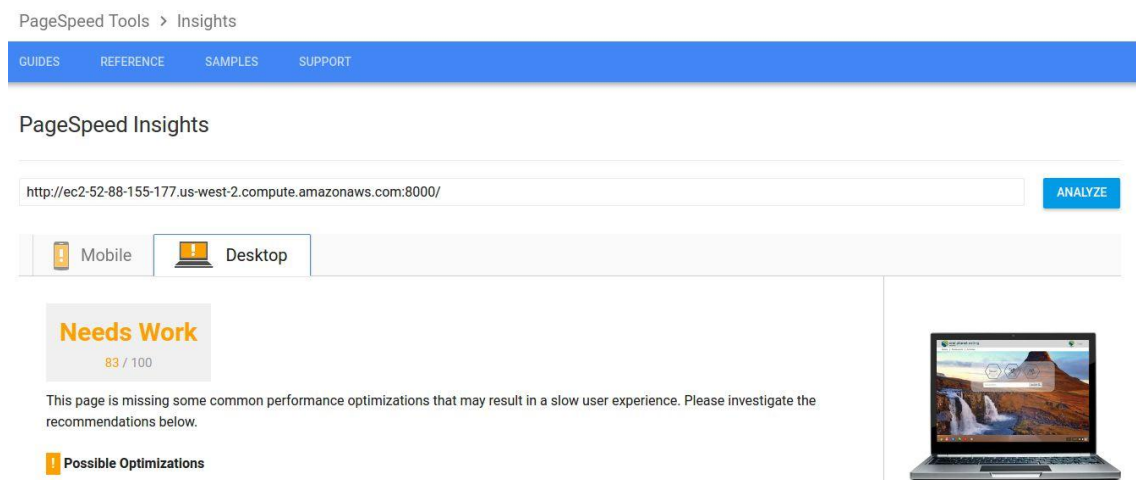


Figure 4.19: landing page speed test

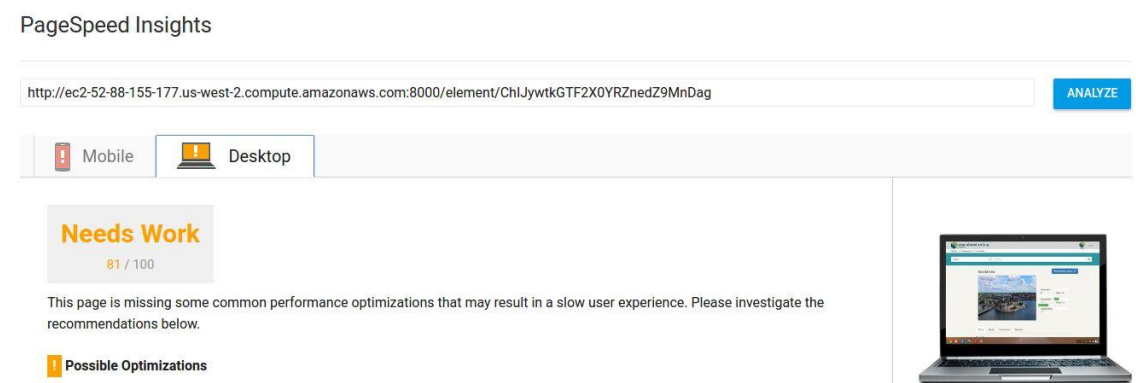


Figure 4.20: details page speed test

Below is included the result for the same test applied to facebook.com for comparison purposes:

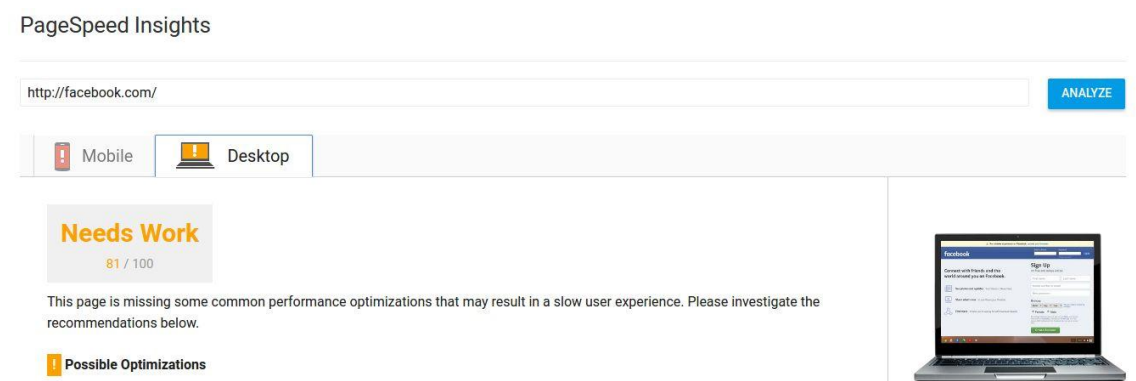


Figure 4.21: facebook page speed test

I also tested the performance of the app using blazemeter. I tested the service with 50 simultaneous virtual users and the results were good.

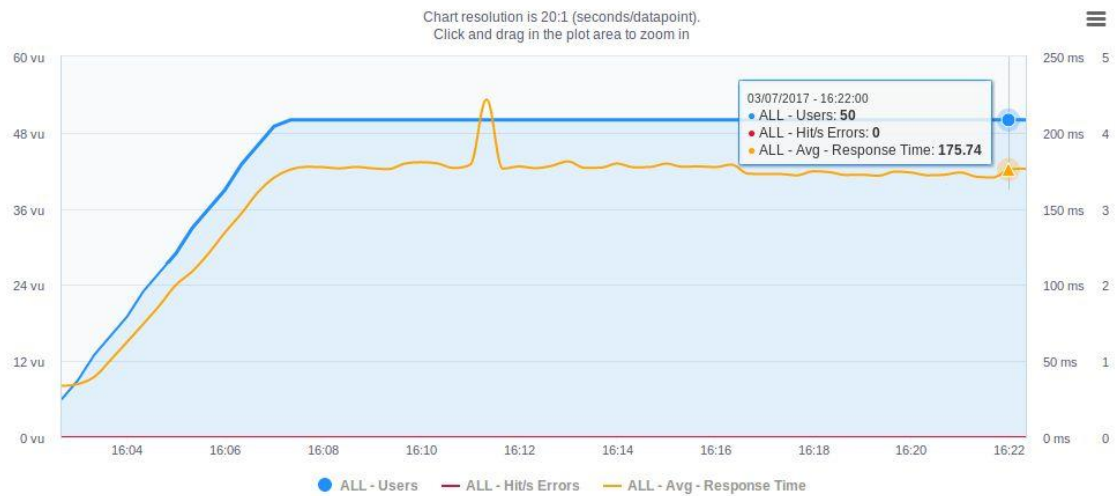


Figure 4.22: concurrent users test

With 50 simultaneous users the response time was 170ms which is okay. Note that the performance is influenced by the performance amazon's server, which is very limited but easily scalable. For the development, I went with a low-performance server instance because of budgetary reasons, but we can scale up easily later when we need to. The used instance is an AWS EC2 t2.micro with the following specs:

- RAM: 1 GB
- vCPUs: 1

## 4.5 Work environment

All of the development was made on my personal laptop which has the following specs:

- RAM: 8GB
- CPU: Intel core i7
- OS: Ubuntu 16.04 LTS

### **PyCharm**

During the development I used the IDE PyCharm. It's developed by JetBrains, the world leader in IDEs development. PyCharm's user interface is very user-friendly and familiar because it's similar to the company's other products.

PyCharm is also extremely powerful. To name a few of its features we can cite: code completion, debugging and integrated version control tools.

### **BitBucket**

For version control we used Git, hosted on BitBucket. Git is the most popular distributed version control system. It's widely used and very powerful. It allows developers to keep all the history of the project, and allows them to collaborate efficiently. BitBucket offers unlimited free private git repositories for teams of up to 5 people.

### **Slack**

Communication was all held on Slack. Slack is becoming the standard in the corporate communication tools and is used by some of the biggest organizations in the world like NASA. Slack allows people to create chat channels, share files, and hold audio and video calls from the web browser. Another major powerful feature of Slack is that it has multiple plugins and external integrations.

### **draw.io & Dia**

The diagrams were made using Dia and draw.io

Both of these tools are open source and very popular. The main difference is that Dia runs locally while draw.io runs on the browser. Both of these tools are powerful, versatile and have ready-to-use UML components.

### **GIMP**

In the beginning of the project I also did some graphic design using GIMP, which is an open source and free alternative to photoshop.

GIMP allows the creation of layers, transparency, filters and many advanced features that were very useful during the design phase.

### **InvisionApp**

I used InvisionApp to create an interactive prototype using the mockups I designed. InvisionApp allows the creation of prototypes from images only. Images become clickable and the behaviour is customizable. The prototype can then be tested in the browser or using a mobile device, and gives a very real experience.

**Chrome developer tools**

Chrome developer tools were used extensively for debugging and tweaking the frontend.

The tools offer several features from simple ones like changing the styling of the DOM and seeing the changes in realtime to more advanced operations like debugging JavaScript code in the browser and monitoring the requests and their responses.

**Postman** Postman was used to test the REST requests for the API.

Postman is a Chrome extension that allows to send custom HTTP requests and preview their responses. In the beginning I used curl, but it turned out Postman is a lot more efficient.

**Unittest** Unittest is a python library for unit testing. It's included with python by default. I used it for testing the searchAd module.

The work done during my internship constitutes the first working prototype of OPR. A substantial effort and time was dedicated to designing and defining how things work because the project is still new and was built from scratch.

Though this is a mere prototype, it's working and is being tested internally and soon to be released to the public. The frontend part of the app is being redesigned, while the backend is now the ground work that the company is building upon.

But before being production ready, the app needs security and penetration testing. Also, now that the project is getting bigger, the project should also take advantage of more sophisticated yet facilitating tools, like continuous integration, continuous deployment and containerization.

The REST API, that is now being used solely for the mobile app, might also turn out useful if the frontend of the web app gets redesigned using a single-page app framework such as React or Angular. This will make the app even easier to maintain as it will loosen the coupling between the frontend and backend.

Other performance improvements can be easily implemented, like using a Content Delivery Network to deliver the static files, but the project is still not big enough and not mature enough to require such optimizations.

The project is easily scalable, especially with the help of AWS which allows the expansion of the infrastructure in a couple of clicks. During the deployment I set up a couple instances and a loadbalancer to test the high-availability of the service and it worked perfectly, but I later removed it for budgetary reasons.

## CHAPTER 6

# References

lead-digital.de

[http://www.lead-digital.de/aktuell/work/arbeiten\\_bei\\_trivago\\_keine\\_hierarchien\\_und\\_15\\_biersorten\\_gratis](http://www.lead-digital.de/aktuell/work/arbeiten_bei_trivago_keine_hierarchien_und_15_biersorten_gratis)

allthingsdistributed.com

[http://www.allthingsdistributed.com/2006/11/working\\_backwards.ht](http://www.allthingsdistributed.com/2006/11/working_backwards.ht)

slideshare.net

<https://www.slideshare.net/anandology/ten-reasons-to-prefer-postgresql-to-mysql>

digitalocean.com

<https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-16-04>

hostingadvice.com

<http://www.hostingadvice.com/how-to/nginx-vs-apache/>

AWS RDS Documentation

<http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html>

AWS EC2 Documentation

<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>