Łódź University of Technology
Faculty of Electrical, Electronic, Computer and Control Engineering
Institute of Applied Computer Science

Engineering Thesis

Web application to connect users based on the antipodes of their locations

Aplikacja webowa do łączenia użytkowników na podstawie antypodów ich lokalizacji

Mayyar Karkout
204641

Thesis supervisor:
Dr. Eng. Tomasz Kowalski

Łódź, September 2019

Streszczenie

Przedstawiona praca skupia na opracowaniu prototypu i projektowaniu planu aplikacji webowej do połączenia użytkowników na podstawie antypodów ich lokacji. Praca ta prezentuje strukturę takiej aplikacji i omawia związane z tym zagadnienie geolokalizacji i wyjaśnia każdą podjętą decyzję oraz wskazuje alternatywne rozwiązania. Wynik to aplikacja spełniająca wymagania ze skutecznością i dokładnością udowodnioną metodami matematycznymi.

Słowa kluczowe: aplikacja webowa, antypody, geolokalizacja, geokodowanie, serwis społecznościowy

Abstract

The focus of this thesis is to prototype and lay the blueprint of a web application that connects users based on the antipodes of their locations. This thesis presents the structure of said application, discusses relevant geolocation issues and clarifies each decision made as well as indulging alternative solutions. The result is an application which fulfills its purpose with efficiency and accuracy proved using mathematical methods.

Keywords: web application, antipodes, geolocation, geocoding, social networking service

1. Prologue

On a macro scale, our Earth is merely a *pale blue dot*, lost in the vastness of the universe. However, from the perspective of any of its inhabitants, it is probably everywhere they had ever been and everywhere they could ever reach.

"Our planet is a lonely speck in the great enveloping cosmic dark. In our obscurity – in all this vastness – there is no hint that help will come from elsewhere to save us from ourselves. It is up to us. It's been said that astronomy is a humbling, and I might add, a character-building experience. To my mind, there is perhaps no better demonstration of the folly of human conceits than this distant image of our tiny world. To me, it underscores our responsibility to deal more kindly and compassionately with one another and to preserve and cherish that pale blue dot, the only home we've ever known." — Carl Sagan, speech at Cornell University, October 13, 1994 [1]

Until such a time when we will be able to find and inhabit another spot of the galaxy, Earth is all we have and it might be all we will ever have. For all its insignificance on a planetary level, it is big enough that exploring the whole of it is a near impossible task for one human to do in the course of their lifetime. We are a very diverse species, we speak thousands of languages and we come in numerous shapes, sizes, colors and cultures. Yet, most of us spend their whole lives living in one place. We are very diverse, but we are more divided than ever. This age of technology has made us more suspect to partisanship and left us tightly polarized into two camps on most issues, with little room for dialogue.

We have had our technological revolution and life has been improving like never before. Now, it is time to use technology to bring people together, eliminate prejudice and help make Earth a better place on a humanitarian level. This is where engineers come in. Engineers have been the propellers of the advancement in our societies, but it is time to start making use of technical knowledge for the societal and human betterment of the world.

2. Introduction

2.1. Aim

The Internet allows users to connect in a very easy and simple way. Most social applications allow people to connect with people they already know (or may know), random strangers, or people generally living nearby. This means most users never get the chance to interact with people from other countries, cultures, and backgrounds. If we take this fact and flip it on its head, the aim becomes to connect users located as far as possible from each other.

Such a service would be based on identifying the location of a user and linking them with another user located on the other side of the planet. Once such two users are registered in the service, they would be able to communicate, exchange messages and perhaps achieve the daring feat of creating an “Earth Sandwich”: putting two slices of bread on opposite points of the Earth’s surface.

2.2. Challenge

For such a project to develop, the basic elements of any communication application are needed. These include user registration, a fitting database management system and structure, a presentable and functional interface, and a chat and messaging service.

At the core of the project is an algorithm that takes the location of a user as input data to figure out the opposite location to this user as the output. In case there is no land at that position, the algorithm must find the nearest inhabited location.

3. Technologies

Originally, this project was supposed to be implemented using the **Java** programming language with the **Spring** framework and **MySql** serving as the database management system. However, after a fair trial period, Spring proved to be a considerably large and complicated framework for such a project; and Java's inherent unsuitability for web applications' development forced a change of direction. Upon reflection, **Python** and the **Flask** micro web framework became the programming language and web framework pair used. MySql was also replaced by the simpler, lighter (and aptly named) **Sqlite**.

To code and prepare the web elements, **JavaScript** was employed to be the client-side scripting language and basic **HTML** for web templates, as well as the open-source **Bulma** framework as the CSS framework for templates' design.

PyCharm was used as the development environment (IDE) and a number of Python packages were installed. Everything was being kept track of by the **Git** system version control, with the project's repositorium available publicly on **GitHub**.

4. Antipodes

4.1. Definition *[2]*

The Earth is round, but it is not a perfect sphere. Due to flattenings at the poles and bulging around the equator, the shape Earth is closest to being is an oblate spheroid. Which brings us to the mathematician and geographical definitions of antipodes.

- Mathematics: the antipode, or antipodal point of a point on the surface of a sphere is the point which is diametrically opposite to it. *(Fig. 4.1)*
- Geography: the antipode of any point on the surface of the Earth is the point on the Earth's surface diametrically opposite to it.

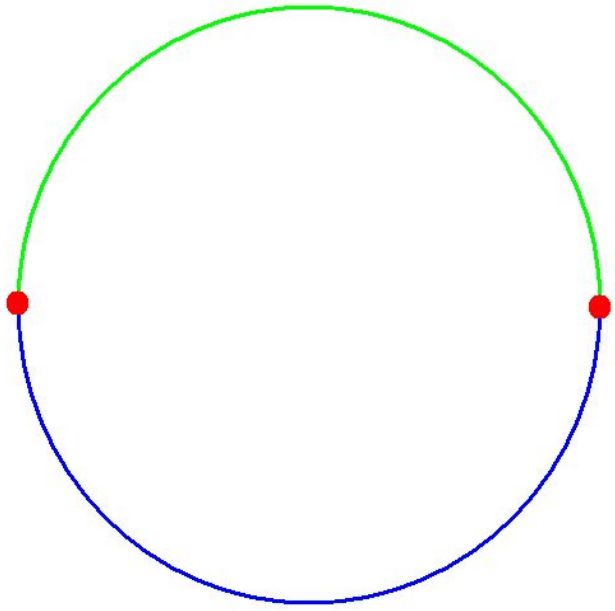


Fig. 4.1. Illustration of antipodes on a circle.

In other words, every spot on the planet forms an antipodal pair with the spot it is farthest from. If points X and Y are points on the Earth's surface, and point Y is the antipode of point X; this means that there is no other point on the Earth's surface that is further from point X than point Y and vice-versa.

4.2. Antipodal Cities

Antipodal cities are cities that are direct antipodes to each other. Those are not common occurrences, because *antipodal land* is scarce. *(Fig. 4.2)* This is down to two factors:

- Two thirds of the Earth's surface being covered with water.
- The amount of land in the Southern Hemisphere being disproportionately low in comparison with the Northern Hemisphere.

Fig. 4.2. Map image: antipodes of each point on Earth's surface. Land antipodes highlighted in orange. [4]

For example, the city of Ulan-Ude, Russia, is the direct antipode of the city of Puerto Natales, Chilé. However, for most cities, the antipode is the ocean. For example, the antipode of Copenhagen, Denmark, is located in the South Pacific Ocean.

4.3. How to Calculate Antipodes

The location of a point on the surface of the Earth is defined by latitude and longitude degrees. The former specifies how far north/south a point is, with the Equator representing 0° latitude . The latter specifies how far east/west a point is, with the Prime Meridian of Greenwich, England representing 0° longitude. The coordinates of the Eiffel Tower, for example, are 48.858239° latitude and 2.294587° longitude, and could be denoted as 48°51'29.7"N 2°17'40.5"E.

Through a few simple steps, we are able to get the antipodal coordinates by having the initial coordinates of a point::

- Change the sign of the latitude.
- Subtract the absolute value of the longitude from 180° and change the sign of the sum.

As per the former example, the antipode of the Eiffel Tower lies at -48.8583701° latitude and -177.7077074° longitude.

5. Application Description

The application built in regards to this project is a web application running on a local-development server.

5.1. Structure

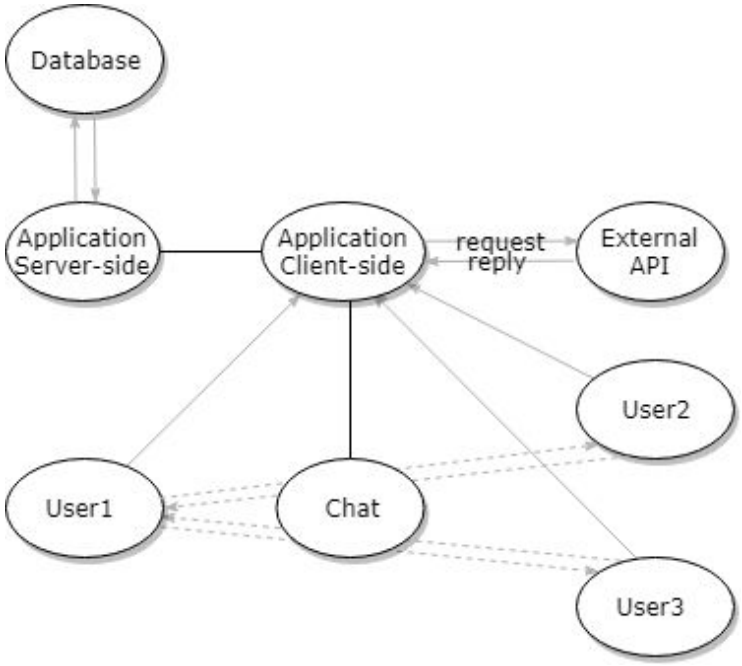


Fig. 5.1. Application structure diagram.

As is the case with web applications, there is a distinction between frontend and backend, as well as a contrast between operations happening client-side and server-side. Structure of the application is shown on figure 5.1. Communication between the application and its database only occur on the server side. On the other hand, API requests and user interactions with the application and each other are handled through the client side of the application. The server side serves as a mediator between the client side and the database. The chat functionality is a part of the client side and it allows only certain users to message each other. Each of these aspects will be discussed in more depth in this chapter.

5.2. Database

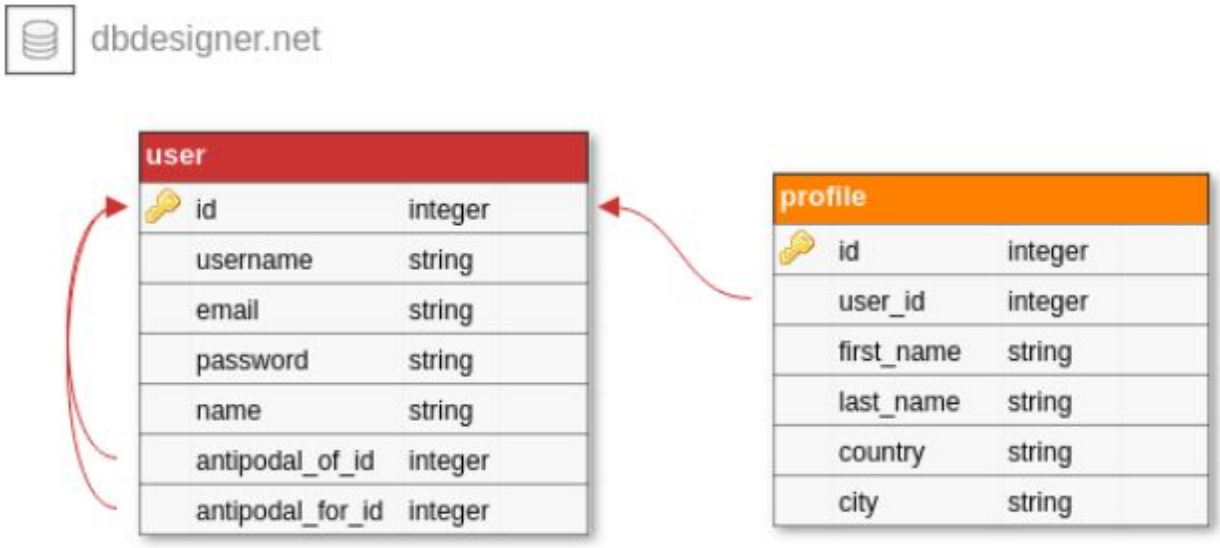


Fig. 5.2. Database structure.

The database consists of two tables, representing users and their profiles. *(Fig.5.2)* Timestamp fields have been cut from the diagram for briefness.

User table:

Aside from the basic *id* (primary key), *username* and *email* fields, there are two additional fields: *antipodal_of_id* and *antipodal_for_id.* Both have foreign key in relation to the *id* field in the same table. These define “one-to-one” relationships between a user and another user.

This means that each user could be related to two different users at the same time: one through the *antipodal_of_id* foreign key and another through the *antipodal_for_id* foreign key. This is because when it comes to cities with no direct antipode, a city could be the antipode of another city without that same city being its antipode.

- Profile table:

For the sake of security and as a good practice, users' details as well as eventual preferences are contained in a separate table to the one containing users' sensitive data such as login and password. In this table, two important fields are the *city* and *country* fields, which are used to connect users. Each profile entry has a foreign key on its *user_id* field, creating a "one-to-one" relationship where each user has one profile and each profile is assigned to one user.

This structure was preferred to other solutions, such as having two user tables, each containing users from two halves of the globe ex. *northern_hemisphere_users* and *southern_hemisphere_users* to optimize performance and sustain it with the scalability of the application.

5.3. Backend

5.3.1. Server Side

The Python scripts that run on the server side of the application as a part of the backend contain configuration files, authentication handling, database querying functions, models and controllers.

- Authentication:

Users are able to create accounts, only one account per email, and log in using their credentials. Passwords are encrypted and stored securely in the database. Logged in users can access pages which allow them to edit their details and chat. Users cannot view or access these pages if they are not logged in nor can they view pages and details belonging to other users. Code regarding signup query is shown in figure 5.3.

```
@auth.route('/signup', methods=['POST'])
def signup_post():
    email = request.form.get('email')
    name = request.form.get('name')
    password = request.form.get('password')

    # check if user already exists in database
    user = User.query.filter_by(email=email).first()
    # redirect to signup page to try again if user with same email is found
    if user:
        return redirect(url_for('auth.signup'))
    # create new user using inputted data; password is encrypted from plaintext
    new_user = User(email=email, name=name, password=generate_password_hash(password, method='sha256'))
    # add new user to database
    new_profile = Profile(user=new_user)
    db.session.add(new_profile)
    db.session.commit()

    return redirect(url_for('auth.login'))
```

Fig. 5.3. Excerpt from the authentication code:user signup fragment.

- Database querying:

This is performed through *POST* requests generated by users' actions on the application. Data from forms is passed on to the *POST* requests, where it is validated through database queries. Any changes to the database are saved when necessary before moving on. *(Fig. 5.4)* The ORM (Object Relational Mapper) used for SQL queries is SQLAlchemy.

```
@main.route('/profile', methods=['POST'])
def profile_post():

    if request.form.get('first_name') is not None and request.form.get('first_name') != "":
        current_user.profile.first_name = request.form.get('first_name')
    if request.form.get('last_name') is not None and request.form.get('last_name') != "":
        current_user.profile.last_name = request.form.get('last_name')
    if request.form.get('country') is not None and request.form.get('country') != "":
        current_user.profile.country = request.form.get('country')
    if request.form.get('city') is not None and request.form.get('city') != "":
        current_user.profile.city = request.form.get('city')

    db.session.commit()

    return render_template('profile.html')
```

Fig. 5.4. Excerpt from the database query code: profile edit POST *request handling.*

- Models

Models as a part of the MVC model are classes stored in an individual file. Each model defines its fields (ids and other fields such as username or country), attributes (whether given field is nullable and/or unique, a field's size and similar details), relationships with other models (one-to-many and similar relationships) as well as settings relating to the modification of an object of the given model in the case of the modification of another model's object (delete when its parent object is deleted and similar behaviors).

This information is then used and translated into migration files, which form corresponding database tables for the models. Part of the profile model is shown on figure 5.5.

```
class Profile(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(255))
    last_name = db.Column(db.String(255))
    country = db.Column(db.String(50))
    city = db.Column(db.String(50))
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    user = db.relationship('User', back_populates='profile')

    def __repr__(self):
        return '<Profile {}>'.format(self.user_id, self.first_name, self.last_name)
```

Fig. 5.5. Excerpt from the models code: profile model.

- Controllers:

Controllers as a part of the MVC model are separated into two types: authentication route controllers and model controllers, kept in different files to preserve SoC (Separation of Concerns). Requests are handled by the controllers and views - discussed in the frontend section - are returned. *(Fig. 5.6)*

```
@main.route('/index')
@main.route('/')
def index():
    return render_template('index.html')


@main.route('/profile')
@login_required
def profile():
    return render_template('profile.html')
```

Fig. 5.6. Excerpt from a controller file: GET request processing.

5.3.2. Client Side

The Javascript scripts that run on the client side of the application as a part of the frontend contain user location detection, processing of certain forms, external API requests, the main friends-finding algorithm and chat scripts.

- User location detection:

The application makes use of the W3C Geolocation API to determine the current location of its users. To explain how this is done, it is necessary to mention a few things about geolocation technology. W3C Geolocation API is a standardization effort by the World Wide Web Consortium in regards to geolocation technology on client-side devices and the retrieval of information concerning their geographical location. It employs four different technologies to retrieve the coordinates of devices in escalating levels of accuracy. These technologies are IP Geolocation, GPS (Global Positioning System), Wi-Fi Positioning and Mobile Network Location (through Cell Tower Triangulation.

When the application is accessed on a telephone device, it is bound to be return a more accurate location because of the availability of those technologies. The user is prompted to accept location detection. Upon confirming, latitude and longitude is received from the external API. In the case of refusal, the user must input their location data by themselves. Implementation shown in figure 5.7.

In later iterations of the application, users will probably have to accept location detection in order to use the application. This would prevent users from faking their locations and minimize undetected malicious use of the application.
Latitude and longitude are loaded into a request to another external API, the Geonames API, which returns more details about the location of the user.

```
function getLocation() {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(showPosition);
    } else {
        x.innerHTML = "Geolocation is not supported by this browser.";
    }
```

Fig. 5.7. Excerpt from client-side script: prompting user for location detection.

- IP Geolocation:

This is the most general and high-level geolocation method. It relies on the nearest public IP address of a device, which means the accuracy of this method is usually limited. In the best-case scenario, it provides information about the city the device is located in.

- Cell Tower Triangulation:

Available on mobiles phones but not on computers, it is a method which relies on radio signals between the phone and several cell towers of the network the phone uses. Cell towers are able to roughly calculate how far the signal-emitting device is through signal strength and round-trip signal time. With three cell towers in radius of the device, it is possible to narrow the location of a device to an area of about two kilometres. [5] *(Fig. 5.8)*

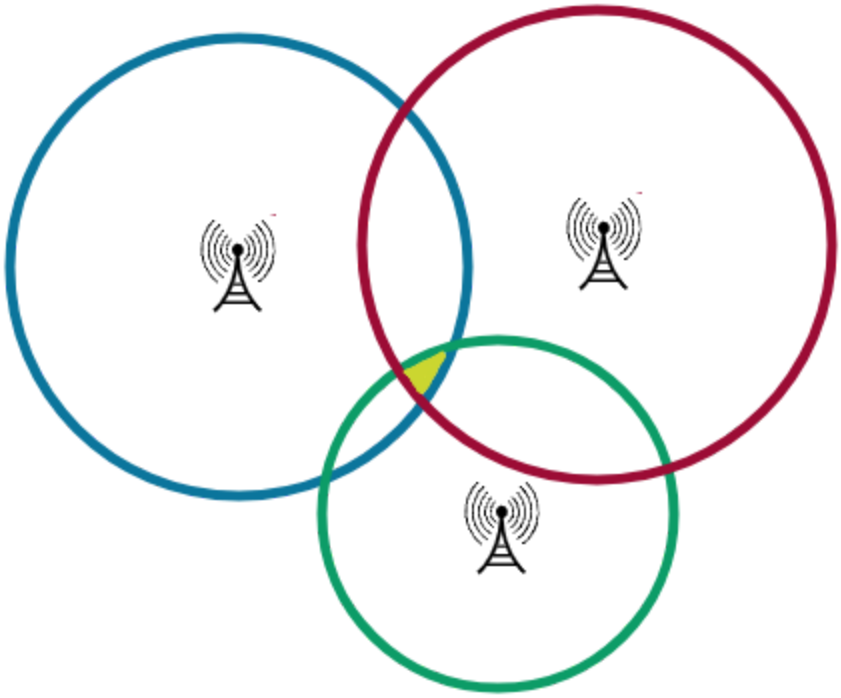


Fig. 5.8. Cell tower triangulation visualization.

- Wi-Fi Positioning:

This method works in a very similar way to cell tower triangulation. Instead of cell towers, position is triangulated on the basis of Wi-Fi networks in range of the device. Due to the surge in Wi-Fi routers in urban areas, positioning is much more accurate in urban areas than in rural ones.

While it is not the most accurate technology, it supplements other technologies in indoor positioning. Indoors, experiments have returned results of around 2.0 to 2.5 metres accuracy. [6]

- GPS (Global Positioning System)

The best scenario for this method is rural, exposed and outdoors areas. GPS satellites determine the location of a device equipped with a GPS chip to a high accuracy. When a device is located indoors, the accuracy of the GPS system suffers a drop in quality because of its reliance on satellite exposure.

This is part of the reason why Wi-Fi Positioning and GPS systems complement each other. Modern GPS receivers are capable to pinpoint locations to up to within thirty centimetres. [7]

- GeoNames API

GeoNames is a geographical database which contains data - among other things - on over 4.8 million populated places. The contents of the database are available for download, free of charge. It has its own set of services, available for free use with reasonable daily quotas. [8] The webservice chosen for use in this application finds the nearest populated place from latitude and longitude. Additionally, requests to the web API could take optional parameters. *Cities* parameter filters the results based on three possible values: ‘cities1000’, ‘cities5000’, or ‘cities15000’. The number in front of ‘cities’ indicates the minimum population of the cities returned by the API request. Example of response to a GeoNames API request in JSON format is shown in figure 5.9.


```
{
  - geonames: [
      - {
            adminCode1: "50",
            lng: "42.11815",
            distance: "13.07079",
            geonameId: 335035,
            toponymName: "Harar",
            countryId: "337996",
            fcl: "P",
            population: 90218,
            countryCode: "ET",
            name: "Harar",
            fclName: "city, village,...",
          - adminCodes1: {
                ISO3166_2: "HA"
            },
            countryName: "Ethiopia",
            fcodeName: "seat of a first-order administrative division",
            adminName1: "Harari",
            lat: "9.31387",
            fcode: "PPLA"
        }
    ]
}
```


Fig. 5.9. Example of response to a GeoNames API request in JSON format.

For the application, cities with a minimum of 15000 inhabitants are used to identify the location of the user.

This decision was made to simplify the process of connecting users based on their locations. Users creating accounts from smaller cities would be assigned to the nearest city with 15000 inhabitants and more, allowing users from less-populated places to also find friends on the application in the early stages of its development.

Assigned to the user's profile from the API response are two parameters: *countryName* which is assigned to the *country* field, and *name* which is assigned to the *city* field. In different countries, different divides of administrative levels exist. Due to differences of language, systems, and administration; the exact definition of *'city'* varies from place to place. Part of the script implementing this is in figure 5.10.

To solve this problem, 'places' found by the API are not defined as being a 'city' or any of the other names for administrative divisions, such as 'county' or 'parish'. Instead, only the name of the place is stated as well as what order of administrative division it is categorized by. For the application, the information of interest is only the name of the *place* and that it has more than 15000 inhabitants.

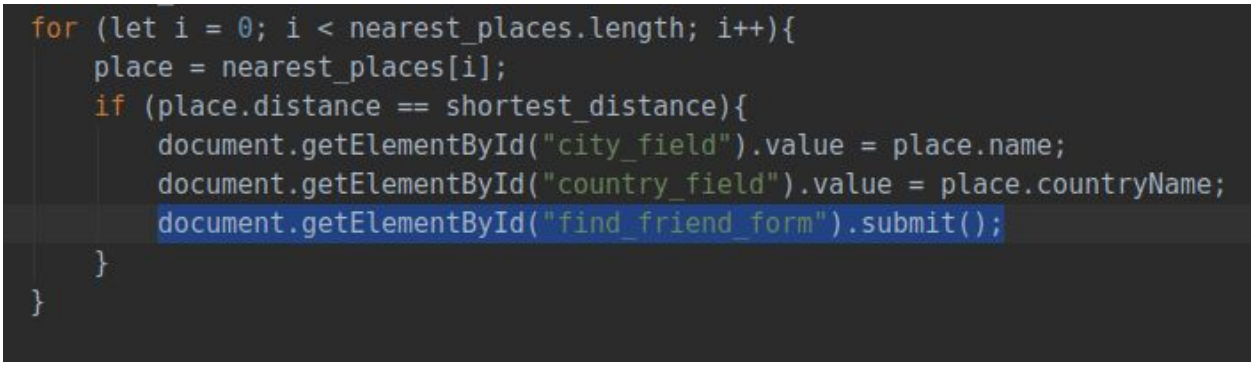
In the application, by using the umbrella term '*city*' instead of the technically-correct term '*administrative division/district*' or the more general term '*place*', it is clearer and simpler to understand from a user's point of view. From the above example, the user's *country* would be 'Ethiopia' and *city* would be 'Harar'.

```
function showPosition(position) {

    const Url = "http://api.geonames.org/findNearbyPlaceNameJSON?lat="
        + position.coords.latitude
        + "&lng="
        + position.coords.longitude
        + "&username=mayar"
        + "&cities=cities15000"

    fetch(Url)
    .then(response => response.json())
    .then(data=>{
        document.getElementById("city_field").value = data.geonames[0].toponymName;
        document.getElementById("country_field").value = data.geonames[0].countryName;
        $("#form3").submit();
    })
}
```

Fig. 5.10. Excerpt from client-side script: external API request and capturing result.

- Forms processing:

On the frontend side, users interact with certain forms to input or trigger data which goes through the application and is processed by the client-side scripts. After form data passes through the client side, the forms are submitted and go to the server side, where the data is handled and changes are written to the database. *(Fig. 5.11)*



```
for (let i = 0; i < nearest_places.length; i++){
    place = nearest_places[i];
    if (place.distance == shortest_distance){
        document.getElementById("city_field").value = place.name;
        document.getElementById("country_field").value = place.countryName;
        document.getElementById("find_friend_form").submit();
    }
}
```

Fig. 5.11. Excerpt from client-side script: example of form processing.

- External API requests:

Requests are generated through a custom function which takes the latitude and longitude of the user's reported location, adds needed parameters to the request and sends it.

In addition to the formerly-discussed 'cities' parameter, the 'username' parameter is essential. In order to make requests to the GeoNames API, one must create an account on their website and use their username as a parameter in the requests each and every time a request is made. This is to identify the source of the request and to maintain the hourly and daily request quotas. The last parameter used is the 'radius' parameter, which sets the radius of the area to be searched in kilometres. *(Fig. 5.12)*
It will be discussed later on in this thesis.

```
function makeApiRequestUrl(lat, lon){
    var Url = "http://api.geonames.org/findNearbyPlaceNameJSON?"
            + "lat="
            + lat
            + "&lng="
            + lon
            + "&username=mayar"
            + "&cities=cities15000"
            + "&radius=300"

    return Url;
}
```

Fig. 5.12. Excerpt from client-side script: API request generating function.

- Friends finding and chat:

When the chat page is accessed, a script which activates a web socket starts running. It listens to events of sent and received messages. The location of the user is taken and used to find out their antipodal location. The algorithm used to find the antipodal city to a user's city will be discussed later in the thesis.

Upon identifying an antipodal city, the application's database is queried to find a user from that city. When such a user is found, they and their antipodal friend become able to communicate with each other in the chat room.

The chat functionality is facilitated through *SocketIO*, a cross-browser JavaScript library for realtime web applications. *(Fig. 5.13)* It is implemented in the application with the usage of *Flask-SocketIO*, an extension for the *Flask* microframework used in the application. [9]

```
var socket = io.connect('http://' + document.domain + ':' + location.port);
  socket.on( 'connect', function() {
    socket.emit( 'my event', {
      data: 'User Connected'
    } )
```

Fig. 5.13. Excerpt from client-side script: connecting to socket.

5.4. Frontend

On the frontend of the application, HTML templates augmented with an external CSS stylesheet pointing to external JavaScript scripts are presented to users.

The current look of the frontend is merely there at the moment to make the application presentable. It is definitely subject to change in compliance with UX (User Experience) best practices.

Frontend is not by far one of the main issues discussed by this thesis and applied in the project, because the subject is distinctly practical and building an aesthetically-pleasing website would considerably veer away from the premise and timeframe of the project.

Templates on the frontend are divided into templates accessible by all users, and templates accessible only by logged in users. *(Fig. 5.14)* All templates extend a 'base' template and share its attributes.

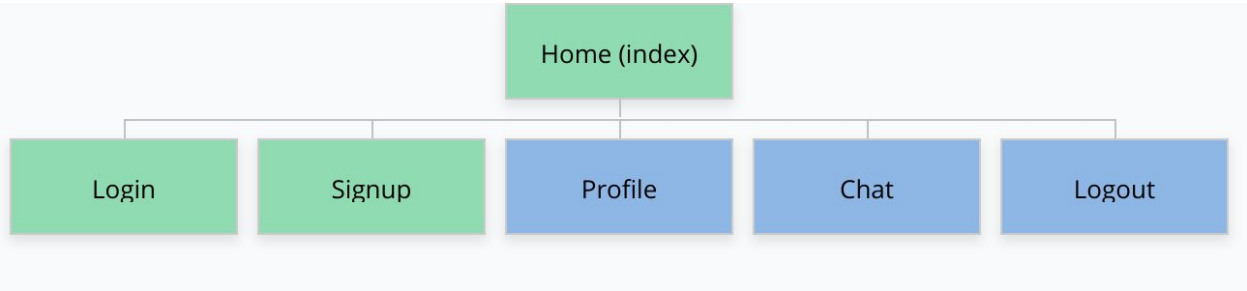


Fig. 5.14. Application sitemap: Green pages don't require login, blue pages require login.

- CSS Stylesheet:

There is no custom CSS stylesheet created for the project. The stylesheet used is from *Bulma*, a free and open-source CSS framework. [10] Screenshots from the application are shown in figures: 5.15, 5.17, 5.18.

It is imported into the application as an external stylesheet. The default color scheme is retained with no changes.

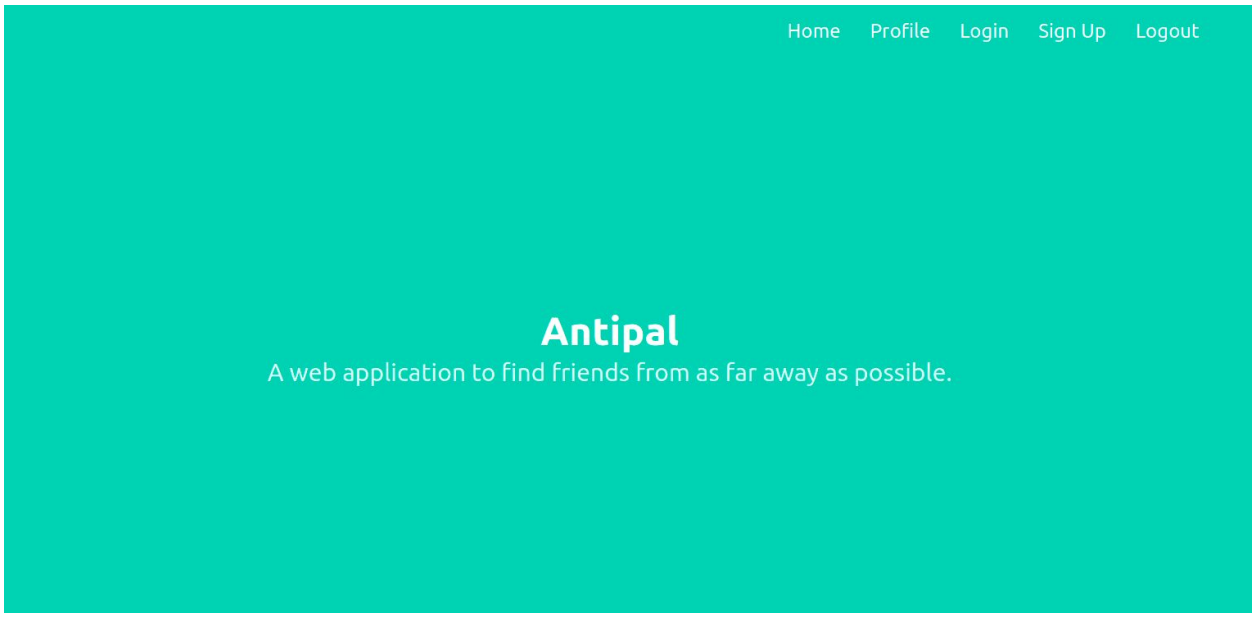


Fig. 5.15. Frontend: screenshot of the homepage of the application.

- Base layout:

The base HTML template used as a layout for the rest of the other templates imports the CSS stylesheet used over the application *(Fig. 5.16)*, links application scripts as well as external Ajax and Jquery scripts, and contains the application's navigation bar.

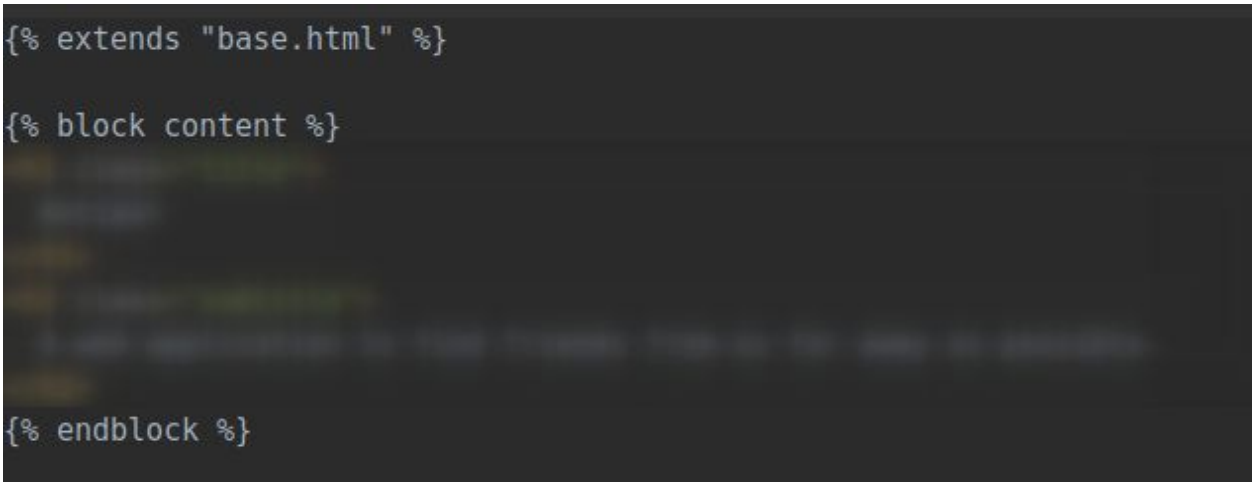


Fig. 5.16. Excerpt from frontend template: extending 'base' template in another template.

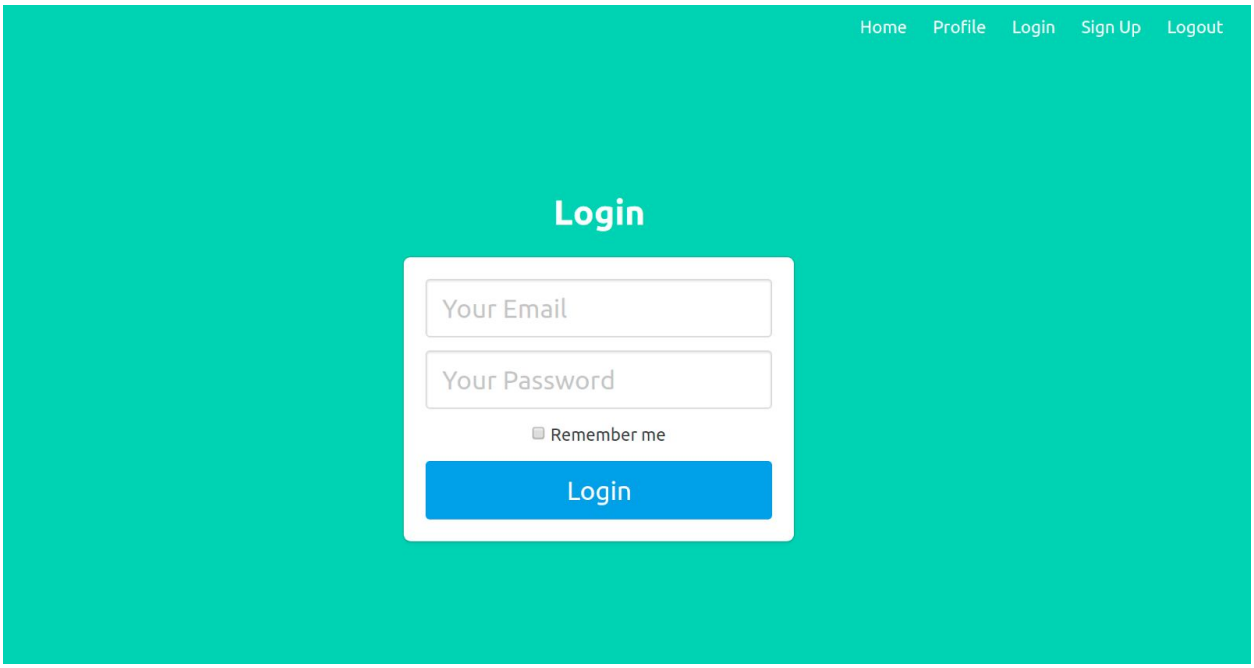


Fig. 5.17. Frontend: screenshot of the 'login' page.

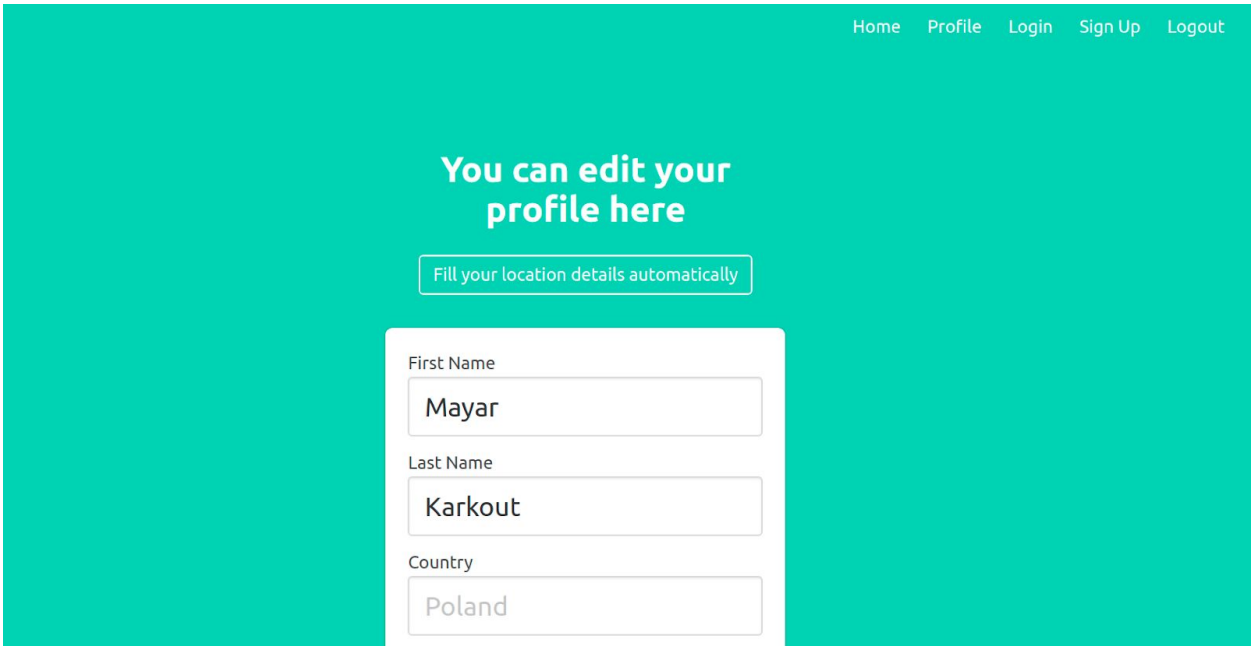


Fig. 5.18 Frontend: screenshot of part of the 'profile' page.

6. Finding Nearest Cities Method

At the heart of the project lies the issue of finding the antipodal friend of a user. As discussed earlier, to calculate the antipode of a location using its coordinates is a simple operation. After doing so, all that is left is to find a user from that location.

It does sound pretty easy, however, there is a small problem. Most antipodes of land locations lie in the water. This comes with its own set of challenges. Namely, how to figure out if said location lies in the water? How to find the nearest land location to a water location?

Both these questions are answered - or at least discussed - in this chapter, but not without raising even more questions. It is a topic which seems uncomplicated on its surface, yet creates more questions the deeper it gets.

6.1. Water or Land?

A scientist's or a geography/satellites engineer's area of work or interest might be to design machinery or carry out research to establish whether a certain point on the surface of the Earth is water or land. From a computer science point of view, an engineer's task would be to figure out a way to extract this information from already existing technologies, solutions, databases, etcetera; then employ it to solve the problem at hand.

Digital maps are compiled using data from satellites and the technology of remote sensing. *Remote sensing* is scanning the earth through satellites or aircraft using recording instruments without making physical contact with what is being investigated.

For normal programmers or aspiring software engineers with no access to such advanced programs or machinery, there are but a few resources or solutions to consider and delve into.

- On Water:

A web API called OnWater [11], built Dockwa Engineers [12], takes the latitude and longitude of a location as input in a request and returns whether this point is on water or not. With no knowledge of the way this API tells the difference between dry land and water, it is difficult to evaluate its accuracy. However, their free plan of 15 *"requests every minute for free. Up to 3 coordinates per bulk request."* would not suffice for this project.

- Elevation:

This approach suggests that water locations are generally below sea-level. Elevation data could be acquired from a service such as Google Maps. A negative elevation would mean that a location is a water location, and a positive elevation would mean that a location is a land location.

There are a few problems with this method. There is a number of places - not a big number, but nevertheless worth noting - on the surface of the planet with a negative elevation which are actually land locations. [13]

Most famously, perhaps, is Death Valley spreading over parts of California and Nevada in the United States. On the other hand, there are water bodies which are situated above sea level. One such example is Lake Titicaca on the border of Peru and Bolivia, with a surface elevation of 3812 m.

With such a margin of inaccuracy, this method is far from perfect. Nevertheless, it is not impossible to employ it and make it work effectively. Such an implementation would rely on creating a database of water bodies above sea level and dry land under sea level.

This is possible because the number of such locations is finite. When such a database is ready (and actively updated), the coordinates of a location would be put against the data to identify if this location exists in the database. If it does, then the location is instantly known as water or land. If not, then elevation would reliably point out whether this location is a water or land one. *(Fig. 6.1)*

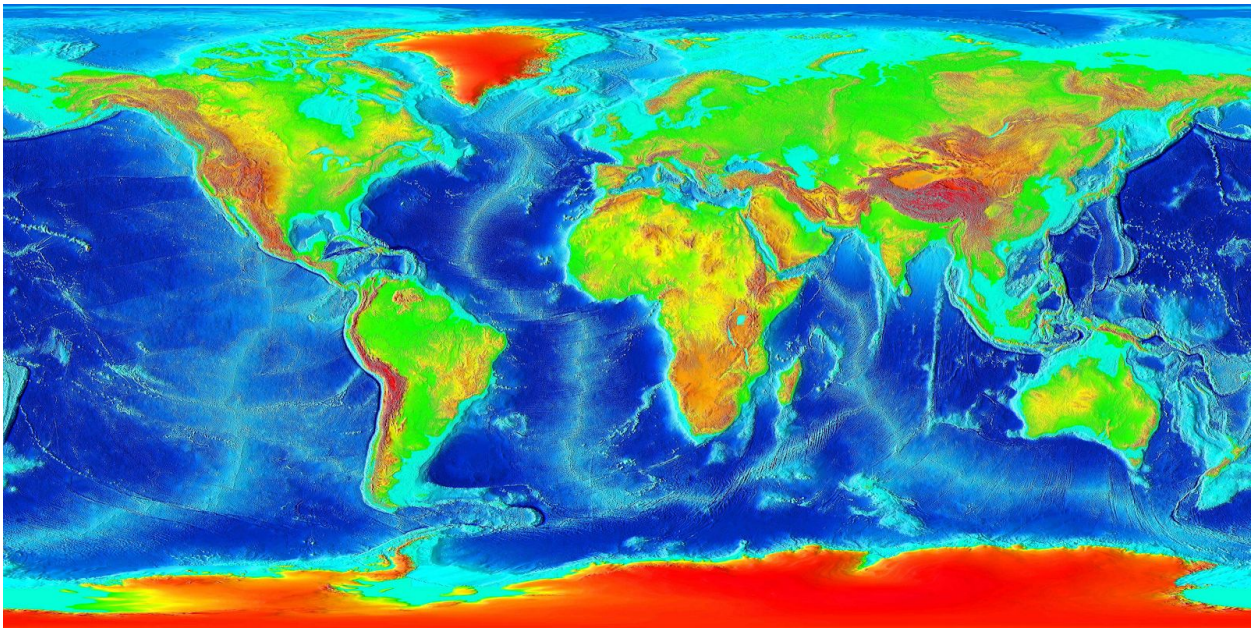


Fig. 6.1. Global land and undersea elevation: warmer colors indicate higher elevations.

- Color detection:

This is a solution which utilizes the same thinking of remote detection. Given a static map, most reliably Google Static Maps [15], it is possible to create a service which receives the coordinates of a location and detects the color of the pixel representing the location on the map. By already knowing which colors on the map represent water, all that would be left to do is to compare the detected pixel's color with the colors representing water. Any other colors would signify that the location is on dry land.

Even though this solution sounds fragile, its accuracy is quite high.

- Reverse geocoding:

Reverse geocoding is a key part of this project and was discussed earlier in this thesis. It is the process of retrieving information - a readable address, mainly - about a location based on its geographical coordinates. Using the aforementioned GeoNames API, when the coordinates fed into the request translate to a water location, the API returns an empty array because it has no address information assigned to water locations. This information is enough for the application to determine the next step.

6.2. Nearest Land

What happens if the antipode of a user's location is a water location? The application is for finding antipodal friends, but if users with no dry-land antipodes can't make use of the application, then its scope would be very limited. Therefore to solve this problem, if a user has no direct inhabited, land antipode; then the application looks for the nearest inhabited (by at least 15000 inhabitants) land and assumes the city at that location as the user's antipode.

The solution employed to originally find the antipode is reverse geocoding is the GeoNames API, and so the task became to work around its limitations. As established earlier, an API request comprising of the coordinates of a water location returns an empty array.

The key here is the *radius* parameter mentioned earlier. If there is a place that fits the criteria in the given radius, then the response from the API returns it in the response array. If there are many places, they are all returned in an array; sorted from the closest place to the farthest. An example of a response to a request using the radius parameter is shown in figure 6.2.

```
{
  - geonames: [
      - {
            adminCode1: "19",
            lng: "76.06035",
            distance: "14.63457",
            geonameId: 1278393,
            toponymName: "Arkalgūd",
            countryId: "1269750",
            fcl: "P",
            population: 15802,
            countryCode: "IN",
            name: "Arkalgūd",
            fclName: "city, village,...",
          - adminCodes1: {
                ISO3166_2: "KA"
            },
            countryName: "India",
            fcodeName: "populated place",
            adminName1: "Karnataka",
            lat: "12.76171",
            fcode: "PPL"
        },
      - {
            adminCode1: "19",
            lng: "76.0962",
            distance: "17.51666",
            geonameId: 1270239,
            toponymName: "Hassan",
            countryId: "1269750",
            fcl: "P",
            population: 128531,
            countryCode: "IN",
            name: "Hassan",
            fclName: "city, village,...",
          - adminCodes1: {
                ISO3166_2: "KA"
            },
            countryName: "India",
            fcodeName: "populated place",
            adminName1: "Karnataka",
            lat: "13.00715",
            fcode: "PPL"
        },
      + {…},
      + {…},
```


Fig. 6.2. Example of response to a GeoNames API request with the radius set to 300km.

For example, if the coordinates of location respond to a water location 120 km away from the closest coastline or water borders of an eligible city and the radius parameter is not set or is set to less than 120, then an empty array is returned by the API. If that is not the case and the radius parameter is set to 120 or more, then the nearest city is returned in the response as well as any cities in the set radius, sorted by distance.

That would be the end of this problem, except for the issue that the radius parameter has an upper limit of 300 km. This presents a new problem to solve with unique conditions. The main and hardest objective of this project has always been to find the farthest city from a user's location and to deal with the problem of water antipodes. The solution chosen and presented is the best method with the conditions and available resources.

On the margin:

The oceanic pole of inaccessibility (the point in the ocean farthest from land) is called "Point Nemo". It is a point in the South Pacific Ocean which lies 2,688 km from the nearest - uninhabited - lands. This puts in context how little 300 km is in comparison to the size of the ocean.

Taking into account the API's limitation, it is possible to design an algorithm which facilitates finding the actual farthest eligible city with maximum accuracy.

Fig. 6.3. Map image on a grid from Google Maps with one sufficient request.

On the figure 6.3, a grid of squares had been applied with each side measuring to about 300 km on the map. The circle on the image shows the radius of an API request, with the centre of the circle being the exact location of coordinates fed to the request. Such a request would return city of Lagos, Nigeria as the nearest city as well as other cities from Nigeria and a city from Benin.

Fig. 6.4. Map image on a grid from Google Maps with one request.

With such a request to the API, the response will be a null array because there is no inhabited land in a radius of 300 km. *(Fig. 6.4)* The next step will be to send four more API requests to check a bigger radius.

Fig. 6.5. Map image on a grid from Google Maps with five requests.

The goal is to cover the biggest area around the original location using the fewest amount of requests. In this case, still, there is no land in sight. *(Fig. 6.5)* The next step is to take the requests to the next level and add four more requests, and keep adding four more requests until an eligible city is found.

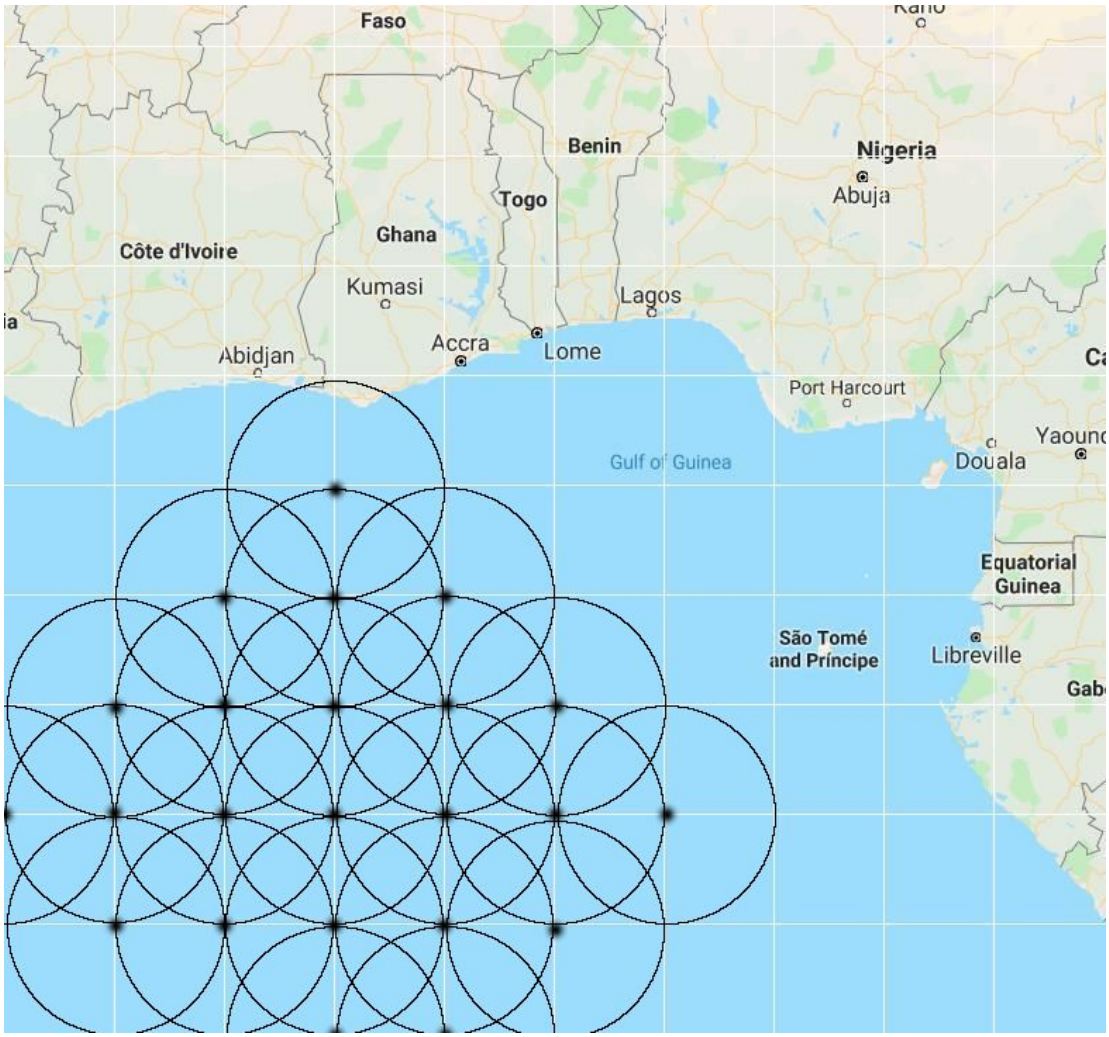


Fig. 6.6. Map image on a grid from Google Maps with twenty five sufficient requests.

On the fourth level of API requests and after twenty five requests, the nearest city to the location is returned: Esim, Ghana; as well as further cities from Ghana and Ivory Coast. *(Fig. 6.6)*

This method isn't perfect. The first level of requests which reflects to just the one request, covers the whole surrounding area with a 100% accuracy, The more levels of requests, the lower the accuracy.

Fig. 6.7. Map image on a grid from Google Map: unsearched area highlighted in yellow.

As demonstrated on the figure 6.7, the area highlighted in yellow isn't searched. If there would be a small island there or the tip of a coast, it would be missed. That wouldn't be a problem if there were no other places located anyway, but if a city was

found near the circumference of the outer circle and another, nearer city is missed then the wrong city would be written to the database.

To deduce the accuracy of this method, the following calculations were carried out:

Given four overlapping circles inside one big circle, it is possible to calculate the area of the big circle not covered by the overlapping circles through a few methods, with the simplest one being the *large square* method. *(Fig. 6.8)*

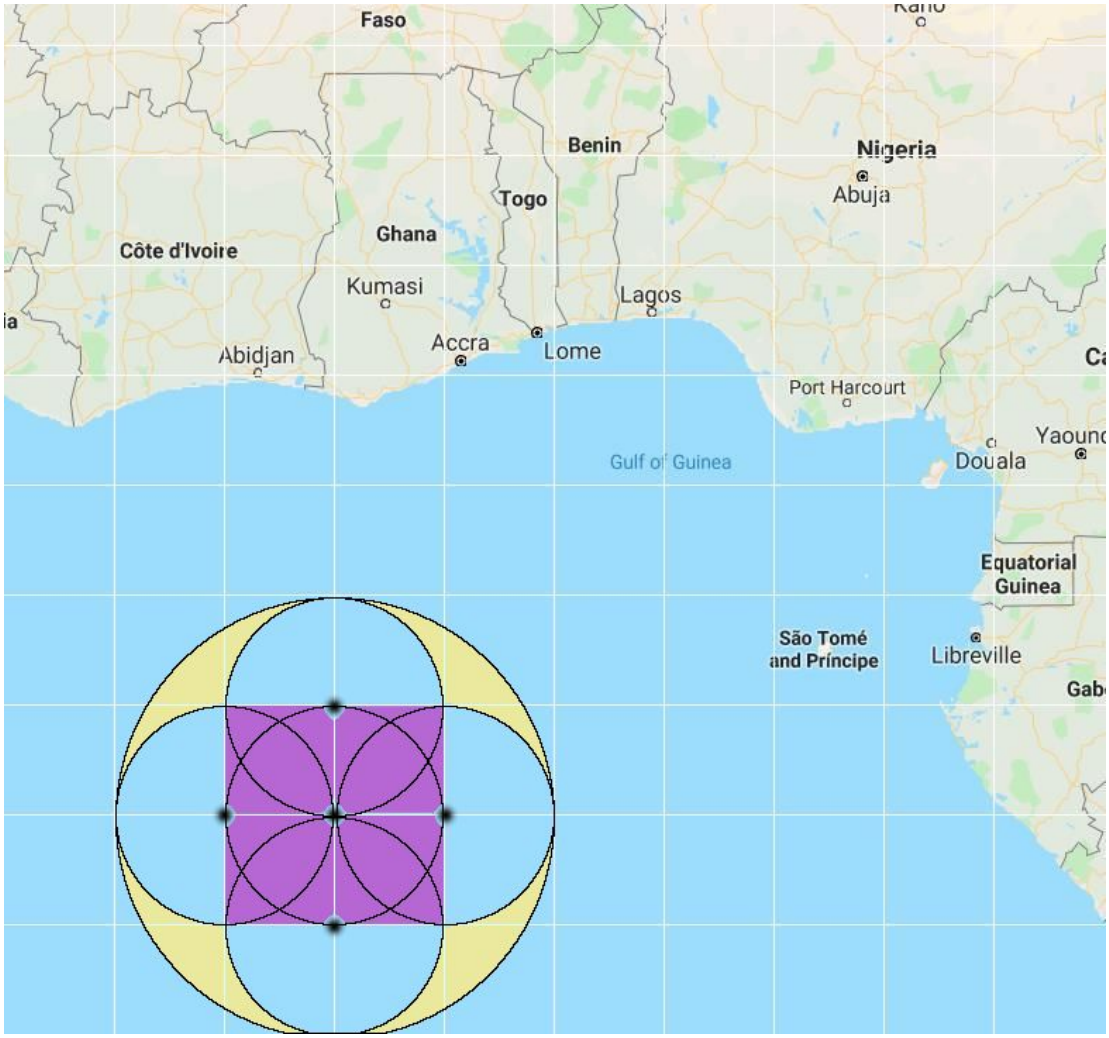


Fig. 6.8. Map image on a grid from Google Map: unsearched area highlighted in yellow.

By connecting the points where two outer circles intersect, a square (highlighted in purple above) covering up the overlapping regions of the circles, which leaves us with a simpler calculation to conduct.

$$Unsearched\ Area\ =\ Whole\ Area\ -\ Searched\ Area$$

$$Unsearched\ Area\ =\ Whole\ Area\ -\ Area\ of\ Square\ -\ Area\ of\ Four\ Semicircles$$

Each of the four semicircles have a diameter that's half as long as the diameter of the bigger circle. The sides of the square are equal to the diameter of the semicircle.

$$Unsearched\ Area\ =\ \pi R^2\ -s^2\ -\ 2\pi r^2$$

Where *R* is the radius of the big circle, *s* is the length of a square's side, and *r* is the radius of a smaller circle.

To take this equation and make it valid for this and following request levels, there is a stable equation that could be presumed. It is assumed that the first level of requests has one circle and its accuracy doesn't need to be calculated.

The second request level has four circles and is calculated using the equation explained earlier:

$$Unsearched\ Area\ =\ \pi R^2\ -s^2\ -\ 2\pi r^2$$

Where $r\ =\ \frac{1}{2}\ R$ and $s = 2r$.

The third request level has eight circles and is calculated through the same method using the following equation:

$$Unsearched\ Area\ =\ \pi R^2\ -3s^2\ -\ 3\pi r^2$$

Where $r\ =\ \frac{1}{3}\ R$ and $s = 2r$.

The fourth request level has twelve circles and is calculated through the same method using the following equation:

$$Unsearched\ Area\ =\ \pi R^2\ -6s^2\ -\ 4\pi r^2$$

Where $r\ =\ \frac{1}{4}\ R$ and $s = 2r$.

There is a pattern here which could be expressed in mathematical terms. The area of the big circle is always the same, but the multiplier of square areas deducted from the big circle's area increases by the request level minus one, and the multiplier of smaller circles area deducted is equal to the request level. Also, the radius of the smaller circles is equal to 1 by request level, whereas the square's side is always equal to double the length of a smaller circle's radius.

$$Unsearched\ Area = \pi R^2 - a_n s^2 - n\pi r^2$$

Where

n = request level

R = radius of big circle

r = radius of small circle

s = length of square's side

And

$$\mathrm{r} = \begin{cases} 0, & \text{if } \mathrm{n} = 1 \\ \frac{1}{n}R, & \text{if } \mathrm{n} > 1 \end{cases}$$

$$\mathrm{s} = 2\mathrm{r}$$

$$\mathrm{a_n} \in A$$

$$\mathrm{A} = \{0, 1, 3, 6, 10, 15, \ldots, \mathrm{a_n}\}$$

$$\mathrm{a_n} = a_{\mathrm{n\text{-}1}} + n$$

Using this equation, the accuracy of this method is estimated by simply calculating what percentage of the whole area is the unsearched area. Here are the results:

Request level	Unsearched % of whole area	Accuracy
1	0%	100%
2	18.16%	81.84%
3	24.24%	75.76%
4	27.25%	72.75%
5	29.07%	70.93%

The results show that the farther a target location is from land, the less reliably its alternate antipode could be found. An almost one-third discrepancy in accuracy is not a small margin of error. This does not mean the solution used in this project was to drop this method and opt for another, because different methods have their disadvantages as well.

There is a simple solution to make sure the accuracy is %100 in all cases:

1. Keep generating requests as specified.
2. When a viable response is returned by the API, save it.
3. Generate the next level of requests and compare the results with the saved response.

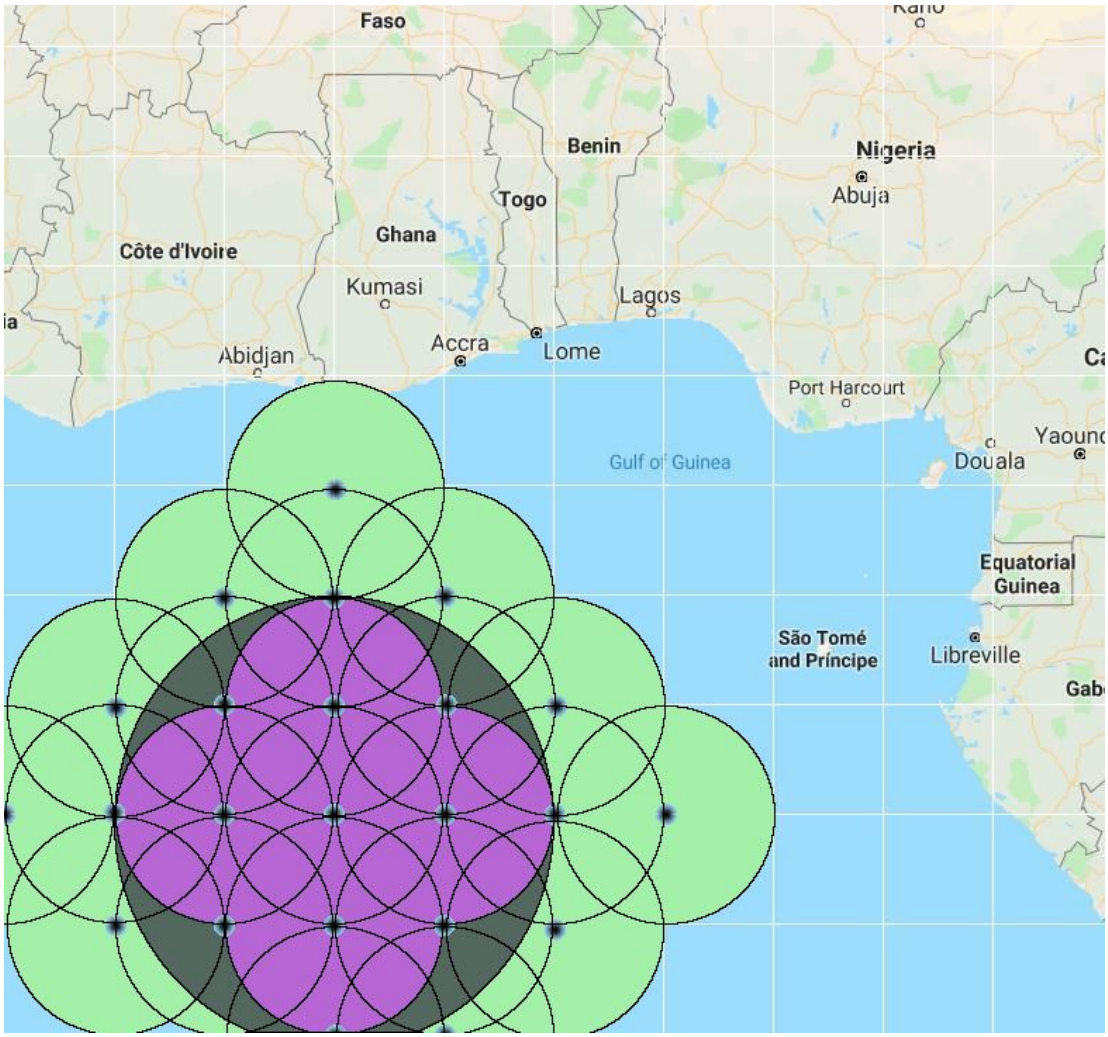


Fig. 6.9. Map image on a grid from Google Map: required area searched by next level of requests highlighted in dark green.

As shown on the figure 6.9, this way, the whole area is searched with %100 accuracy, leaving no room for doubt. It is possible to maintain because the request costs aren't high enough to worry about.

However, if there would be a necessity, here is one way to reduce the amount of requests needed:

Before sending the next batch of requests, check to see whether the found location is in the searched vicinity with 100% efficiency. *(Fig. 6.10)*

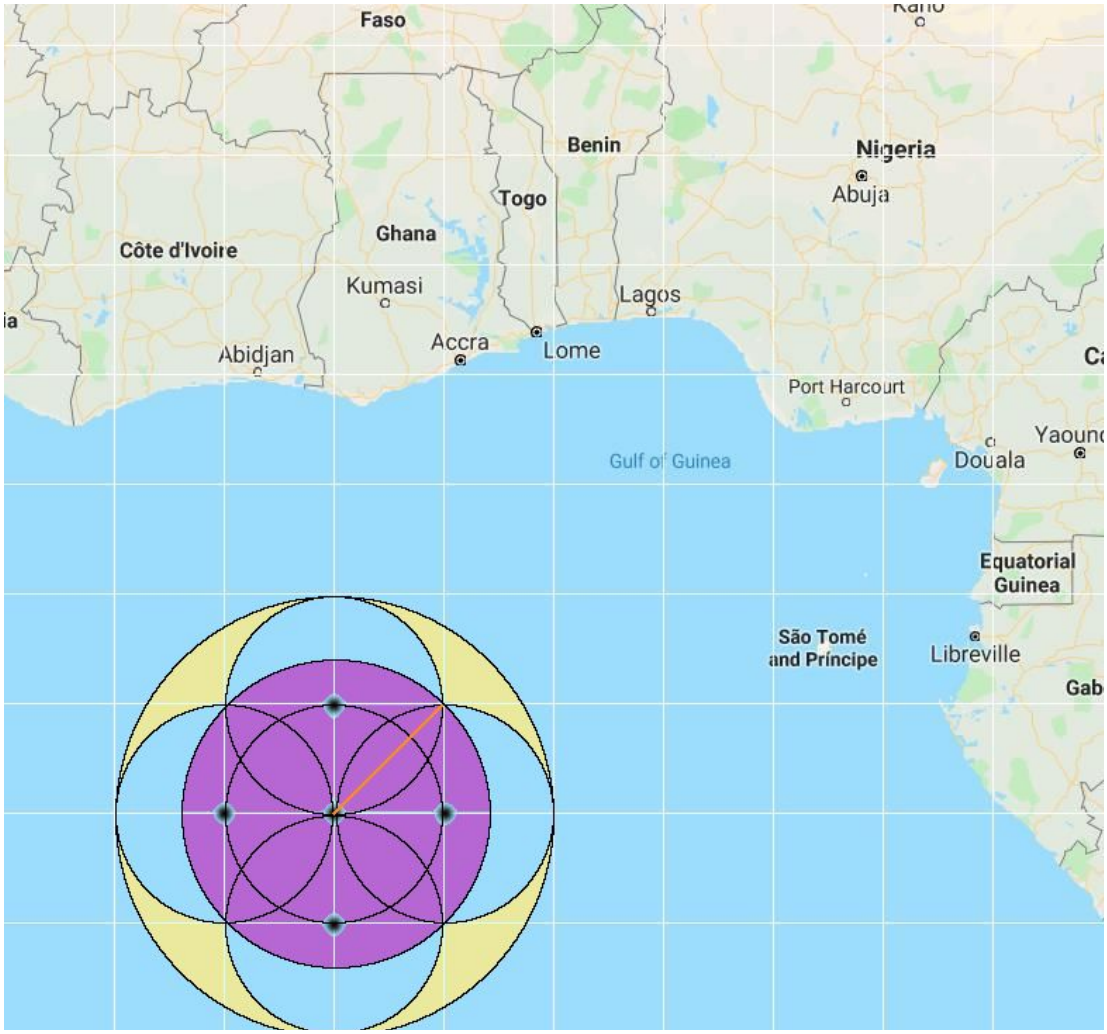


Fig. 6.10. Map image on a grid from Google Map: efficiently searched area highlighted in purple.

The radius of the area where the nearest location is found - if found - is equal to the diagonal of the square with a length side equal to twice the length of a small circle's radius.

$$Area's\ radius\ =\ square's\ diagonal\ =\ \sqrt{s^2\ +\ s^2}$$

If the found location's distance from the original location is equal to or less than the diagonal, then there is no need to send out another batch of requests.

6.3. Algorithm

The algorithm which locates the antipode of a user's location goes through the following steps:

1. Get user's coordinates from browser.
2. Calculate coordinates of the user's antipode.
3. Send API request with those coordinates as parameters, as well as the radius parameter set to 300 and the cities parameter set to 15000.
4. If a city is found at that location and the response is valid, set that city as the user's antipodal city.
5. If not, generate 4 more requests with coordinates around 300 km* east, west, north and south of antipode. Proceed according to the process described earlier until a valid response is found.
6. If a valid response is returned by more than one request, calculate the distance of each valid city from original antipode. Take the nearest city and set it as the user's antipodal city.

* There is a conversion to be made from latitude and longitude coordinates to kilometres. When it comes to latitude, because degrees of latitude are parallel, the conversion from degrees of latitude to kilometres is approximately the same no matter where the location is on Earth.

$$1° \; latitude \approx 111.2 \; km$$

This is not the case when it comes to longitude. Longitude degrees converge at the poles and are at their farthest apart at the equator. This means their lengths

significantly vary - from 111.32 km at the equator to 19.39 km at 80° (north or south) until it reaches zero at the poles. To calculate the length of a longitude degree at a location, it is necessary to have the latitude as well.

$$1° \; longitude \approx cos(latitude \; \tfrac{\pi}{180}) \; km$$

In the algorithm, distances between requests are adjusted in accordance with latitude and longitude.

7. Alternative Solutions

7.1. Cities Database

One solution to solve the water antipode problem would be is to attain access to a database of coastal cities and their coordinates. After recognizing that an antipode is located at a water location, this location is put against the database to find out which city is the closest to this location by measuring the distance between them using their coordinates. This process could be sped up by first finding out the nearest country using a similar database filled with country data, then only comparing the cities of this nearest country to the antipode.

Such data does exist, but it is either behind paywalls, scattered over various different sites, or not constructed in the way needed. For these reasons, this solution wasn’t taken into account for this project.

7.2. Maritime Boundaries

This approach reverses the idea of finding land and is based on establishing the water areas on the surface of the Earth. Through using a database such as Marine Regions [16] and extracting a KML file[1] from it, we are able to visualize the world as polygons extending between coasts. *(Fig. 7.1)*

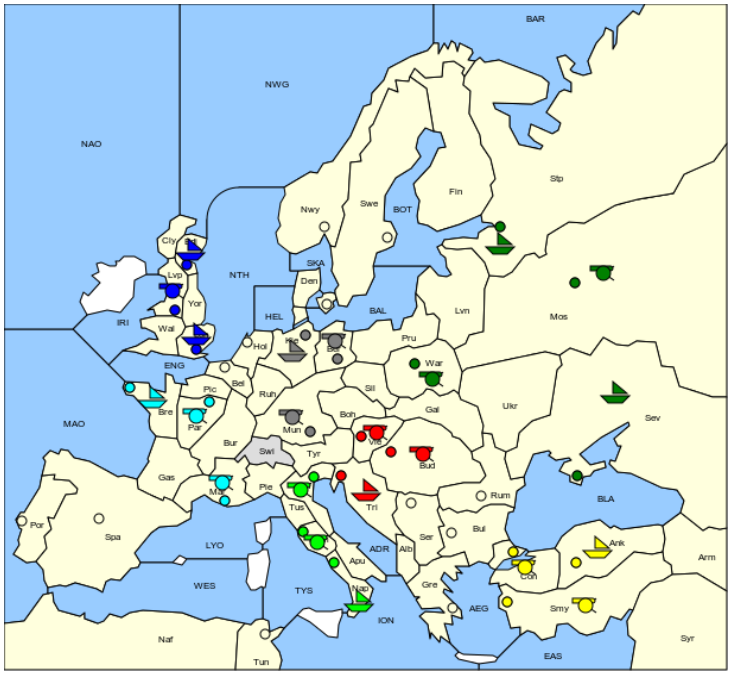


Fig. 7.1. Image from the game Diplomacy [17] demonstrating a rough visualization of the maritime boundaries method.

If the location searched lies within one of the sea polygons, then it is at sea. To find the nearest land location, an algorithm would need to iterate through points at the borders of the polygon and calculate the distance from each point to the antipode in order to find the nearest point.[2]

[1] *KML is a file format used to display geographic data in an Earth browser such as Google Earth. [18]*
[2] *Idea credited to user Jason Fox on StackOverflow. [19]*

This method was not used due to the difficulty of testing and checking whether the database is exhaustive enough, as well as complications that would arise with getting familiar with KML.

7.3. Gray Areas

This is more of an upgrade on the existing solution than an alternative. Using the same method used in this project, it is possible to create a cache filled and constantly updated with data describing already-searched areas, where no results have been found. Such ‘gray areas’ would be ignored when sending requests to the API, thus saving on requests and optimizing performance. *(Fig. 7.2)*

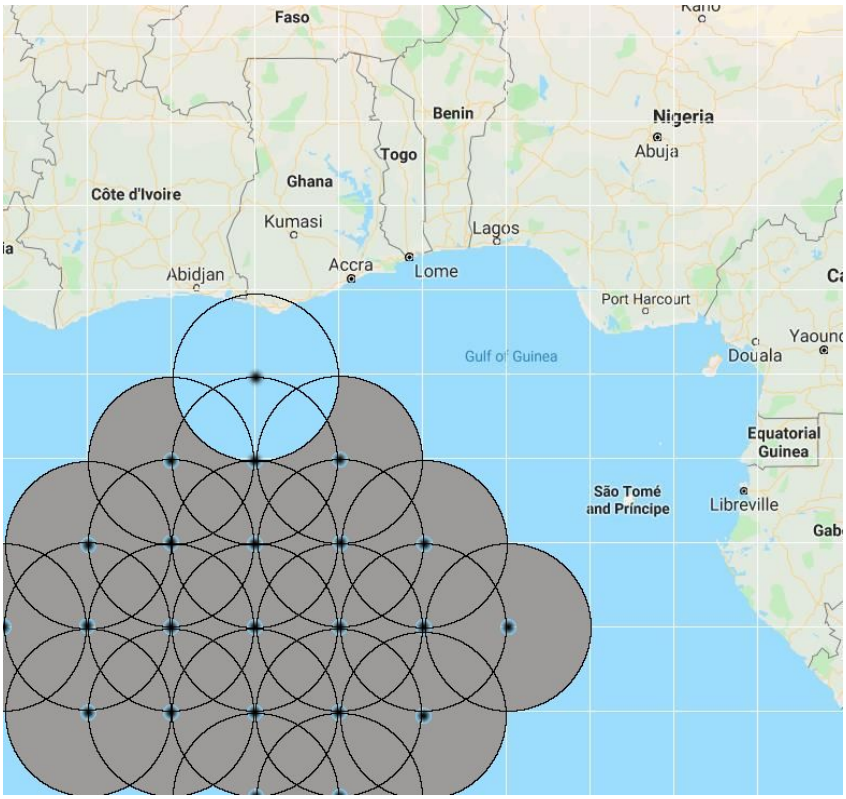


Fig. 7.2. Map image on a grid from Google Map: gray area would be exempt from subsequent requests.

8. Conclusion

Throughout this thesis, various aspects of the discussed problem were exhibited, broken down, explained and solved. Hopefully, readers will agree that the nominated solution is effective and practical based on the arguments presented. The implementation fits the expectations set and achieves the goal of the project.

This project lives on a local development server. The preconditions are there for this to become a public application for public use, but it remains to be seen whether it will see light one day. When it comes to scalability, as discussed, API requests would not become a problem as one user would not be able to change their location so often for it to be problematic for the free API restrictions. The database and application structure is simple enough that even if the database proves to scale badly, a change of database would hardly be costly at all.

Obstacles such as trying out different technologies and getting familiar with them occupied a big chunk of the project's total time, but when the right technologies were established; development proceeded seamlessly. Many challenges stemmed from the initial prospect of creating a chatting web application with a couple of extra additions. Any use of geolocation requires an understanding of the subject and every step deeper into it came with its own set of questions.

Nevertheless, writing this thesis was enriching in knowledge and experience. It shows that one should not judge a subject on a surface level, because sometimes the simplest-sounding subjects prove to be complicated and multilayered.

9. References & Internet Sources

[1] The Big Sky Astronomy Club, http://www.bigskyastroclub.org/pale_blue_dot.html

[2] Sawe, Benjamin Elisha, *What Is An Antipode In Geography?*, World Atlas, 2017.

[3] User Tom Ruen, *Wikimedia Commons* 2013, *https://commons.wikimedia.org/wiki/File:Digon_on_circle.png*

[4] User Citynoise, Unmanipulated image under the Creative Commons Attribution-ShareAlike 2.5 Generic License, Wikimedia Commons, 2007, https://commons.wikimedia.org/wiki/File:Antipodes_LAEA.png

[5] P. Locke, *Cell Tower Triangulation – How it Works*, The Wrongful Convictions Blog, 2012.

[6] S. Moore, *Superaccurate GPS Chips Coming to Smartphones in 2018*, IEEE Spectrum, 2017.

[7] Jekabsons, Kairish, & Zuravlyov, *An Analysis of Wi-Fi Based Indoor Positioning Accuracy*, Scientific Journal of Riga Technical University, 2011.

[8] Unxos GmbH, https://www.geonames.org/,

[9] Socket IO (open source), https://socket.io/

[10] Jeremy Thomas, https://bulma.io/

[11] Dockwa Engineers, https://onwater.io/

[12] Dockwa Engineering, https://engineering.dockwa.com/

[13] Central Intelligence Agency, *Elevations*, The World Factbook, 2019.

[14] National Centers for Environmental Information, https://www.ngdc.noaa.gov/mgg/image/2minrelief.html

[15] Google, https://developers.google.com/maps/documentation/maps-static/intro

[16] Marine Regions, http://www.marineregions.org

[17] Martin Asal, 2019, https://commons.wikimedia.org/wiki/File:Diplomacy.svg

[18] Google, https://developers.google.com/kml/documentation/kml_tut

[19] Jason Fox, 2014, https://stackoverflow.com/questions/22157421/find-land-mass-nearest-to-a-geo-coordinate,

All sources accessed September 2019.

10. Figures Listing