Geo-fencing-based emergency advertising

Gabriele Magazzù
Master's student in Computer Science
Alma Mater Studiorum - University of Bologna
Via Zamboni, 33 - 40126 Bologna
Student ID: 0001102322
Email: gabriele.magazzu@studio.unibo.it

Gabriele Raciti
Master's student in Computer Science
Alma Mater Studiorum - University of Bologna
Via Zamboni, 33 - 40126 Bologna
Student ID: 0001102147
Email: gabriele.raciti2@studio.unibo.it

***Abstract*—A platform designed for the distribution of geolocated alerts related to potential hazardous situations, such as road closures. The system, managed by administrators defining geofences, ensures that alerts reach all users within specified geographical areas. Notably, the platform adapts the alert delivery experience based on the user's mode of transportation (walking or driving). In the former scenario, information is presented visually on the user's device map and in a text notification, while in the latter, the message is conveyed audibly. Leveraging location-based services, context-aware services, and human activity recognition, the platform enhances public safety by tailoring alert experiences to user contexts. The implementation includes a well-developed architecture involving a mobile app, backend, database, and frontend dashboard, orchestrated with Docker and Kubernetes. The system's versatility makes it applicable for various purposes demonstrating its potential for optimizing public safety.**

***Index Terms*—Public Safety, Emergency Services, Hazardous Situations, Geolocated Alerts, Location-Based Services, Context-Aware Services, Human Activity Recognition, Docker, Kubernetes, PostGIS.**

I. Introduction

Mobile device usage is prevalent globally, and modern technologies enable real-time data acquisition from these devices. As a result, it is possible to collect user information. These technologies primarily encompass geolocation and user activity recognition. Despite potential privacy implications, these tools can be used to create applications that serve the interests of public safety.

Throughout history, **Location-based Services** (LBSs) have been recognized as a valuable tool for different purposes. Initially, they found their primary applications in three main areas: military and government industries, **emergency services**, and the commercial sector. Besides the military use of location data, emergency services have turned out to be an important application field. [1]

Nevertheless, **location** is just one example that describes the current situation of the mobile user. The sum of all parameters that are taken into consideration for delivering the user with relevant information are called context, and they are derived and processed by so-called **Context-Aware Services** (CASs). [2]

Another closely connected field of mobile device application focuses on **Human Activity Recognition** (HAR).

Understanding what users are doing in the physical world allows your app to be smarter about how to interact with them. The Activity Recognition API - Google APIs for Android used in this project - is built on top of the sensors available in a device. Device sensors provide insights into what users are currently doing.

The purpose of our project is as follows: to develop a platform including a mobile application for transmitting **geolocated informations** concerning potentially hazardous situations (e.g., road closures). Moreover, the system monitors users' transportation modes (walking or driving) to ensure that the message is delivered in a context-appropriate manner. When in walking mode, information is presented via a text notification and on the user's device map. Conversely, in car mode, the text message is relayed through an audio notification.

Hence, we acknowledge the platform's potential for use and expansion for purposes aimed at enhancing public safety. For instance, it could serve as an emergency service managed by a city's administration to inform its citizens about potential hazards and risks. This approach enables us to notify all those who want to join the project by installing and using the application, ensuring they receive information.

While the usage mentioned above serves as an illustrative example, it undoubtedly presents numerous opportunities for unlocking further potential and exploring different avenues to optimize the system and the platform itself.

II. Project's Architecture

The platform we have built comprises the following key components: a **mobile app**, a **backend**, a **database**, and a **frontend dashboard**. The backend and frontend components are developed within Docker containers, orchestrated seamlessly through the Kubernetes framework.

A. Mobile App Architecture

The mobile app offers the following features:

1) Automatic position detection.
2) Automatic detection of transportation mode (walking or car).
3) Receiving alert messages, which include text and coordinates. When the current mode is "walking," a notification containing the text of the alarm (different based on the

user's location relative to the alarm's location) is shown, and the user's and alert location are displayed on a map. If the current mode is "car," the message is played back audibly.

The application includes two activities, shown in the **Fig. 1**:

- **Auth activity**. A layout where the user can log in or sign up for the platform. This activity will only appear when the user logs in for the first time or after a manual logout on the device, as the application retains the previous authentication session.
- **Main activity**. A layout where a user can:
  - see the email associated with the account used for login;
  - log-out, using the dedicated button;
  - see the geographic map and any alert geofences that have been added, as well as its location.

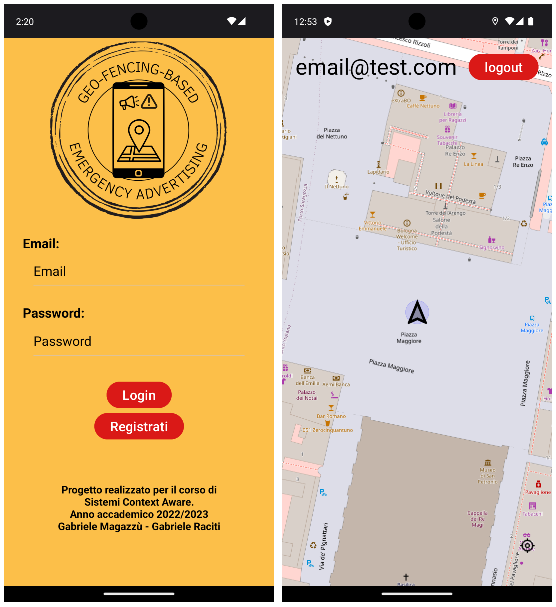

2:20
GEO-FENCING-BASED EMERGENCY ADVERTISING
Email:
Email
Password:
Password
Login
Registrati
Progetto realizzato per il corso di Sistemi Context Aware.
Anno accademico 2022/2023
Gabriele Magazzù - Gabriele Raciti
12:53
email@test.com
logout
Piazza del Nettuno
Piazza Re Enzo
Piazza Maggiore
Via dei Pignattari

Fig. 1. Auth and Main activity.

B. Backend Architecture

The backend receives user locations and checks if they are within specific alarm geofences (added by administrators via frontend), which are defined as polygonal areas. There are three levels of alarms:

1) When a user is inside a geofence.
2) When a user is outside the geofence but within 1 km of its boundaries.
3) When a user is outside the geofence, at a distance ranging from 1 to 2 km from its boundaries.

If any of these conditions are met, an alarm is triggered, and the corresponding notification is sent to the user.

The backend manages data using POSTGIS and spatial queries.

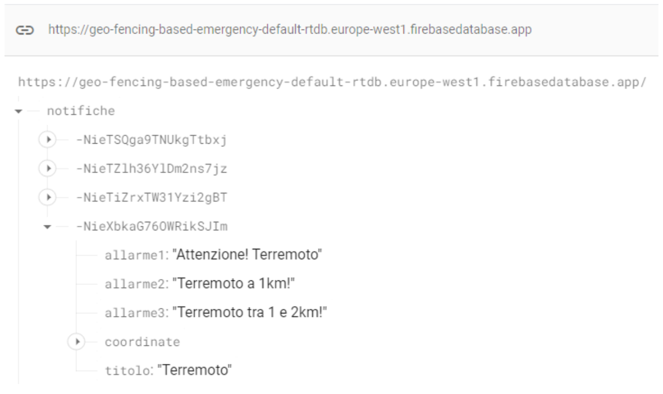
C. Database Architecture

The database is split into a real-time database (Firebase) and an SQL database with the POSTGIS extension.

*1) **Firebase RTDB**:* The real-time database (RTDB) consists of a tree structure with two main branches:

a) Notifications: the branch, if present, contains information regarding the alert geofences inserted in the backend.

Below, we provide the structure of this branch.



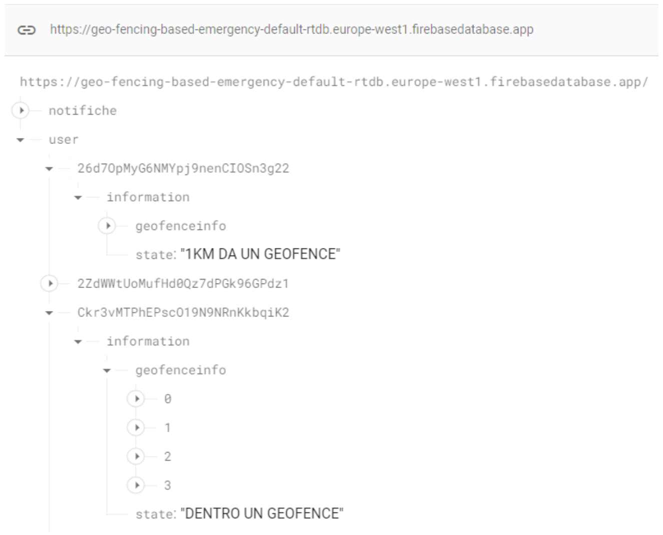
```
https://geo-fencing-based-emergency-default-rtdb.europe-west1.firebasedatabase.app

https://geo-fencing-based-emergency-default-rtdb.europe-west1.firebasedatabase.app/
notifiche
  -NieTSQga9TNUkgTtbxj
  -NieTZlh36YlDm2ns7jz
  -NieTiZrxTW31Yzi2gBT
  -NieXbkaG76OWRikSJIm
    allarme1: "Attenzione! Terremoto"
    allarme2: "Terremoto a 1km!"
    allarme3: "Terremoto tra 1 e 2km!"
    coordinate
    titolo: "Terremoto"
```

Fig. 2. *Notification* branch in RTDB (Firebase).

b) User: the branch containing information about users (geofences involving the user and the global *state* field).

Below, we provide the structure of this branch.



```
https://geo-fencing-based-emergency-default-rtdb.europe-west1.firebasedatabase.app

https://geo-fencing-based-emergency-default-rtdb.europe-west1.firebasedatabase.app/
notifiche
user
  26d7OpMyG6NMYpj9nenCIOSn3g22
    information
      geofenceinfo
      state: "1KM DA UN GEOFENCE"
  2ZdWWtUoMufHd0Qz7dPGk96GPdz1
  Ckr3vMTPhEPscO19N9NRnKkbqiK2
    information
      geofenceinfo
        0
        1
        2
        3
      state: "DENTRO UN GEOFENCE"
```

Fig. 3. *User* branch in RTDB (Firebase).

*2) **Database SQL - POSTGIS**:* The SQL database handles geospatial data, user information, and alarm geofences through the ***emergency-schema***. There are three tables within the schema:

a) edge-information: it contains information about the position of the two edge server nodes.

b) user-information: it contains information about users, namely their location and detected activity (walking or car)..

c) geofence-information: it contains information about alarm geofences, namely their location, title and a timestamp.

D. Frontend Architecture

The frontend is a web dashboard developed using OpenLayers, shown in **Fig. 4**. It is exclusively used by the platform administrator (e.g., the municipality of Bologna). Through it, the administrator can:

1) see user locations as markers on the map, with filtering based on their mode of transportation;
2) create a new alert by specifying the geofence (polygonal area), title and messages to be sent to various user for the three different alert/proximity levels.
3) view the existing geofences, colored based on the number of users currently inside them.
4) clustering user positions using the K-Means algorithm. The number of clusters can be managed in two ways: (i) automatic configuration (the system selects the optimal number of clusters using the elbow method); (ii) manual configuration, with the number of clusters entered by the user through the interface.

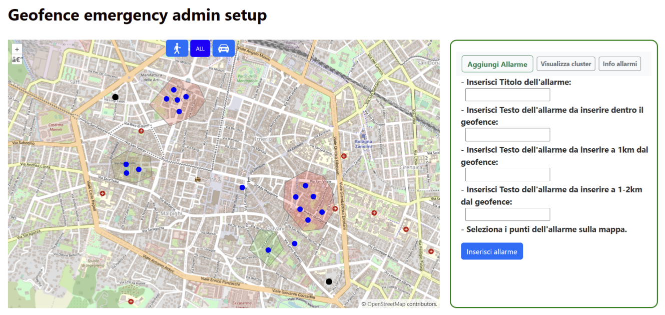

Geofence emergency admin setup
ALL
Aggiungi Allarme
Visualizza cluster
Info allarmi
- Inserisci Titolo dell'allarme:
- Inserisci Testo dell'allarme da inserire dentro il geofence:
- Inserisci Testo dell'allarme da inserire a 1km dal geofence:
- Inserisci Testo dell'allarme da inserire a 1-2km dal geofence:
- Seleziona i punti dell'allarme sulla mappa.
Inserisci allarme
© OpenStreetMap contributors.

Fig. 4. Frontend webpage.

III. Project's Implementation

In this section, we provide a description of the implementation of the main parts of the project.

Our main goal was to implement a platform ready for use by:

- Users with devices running the Android operating system, through a mobile application.
- The platform administrator, through a frontend dashboard accessible via a web browser.

All of this has been achieved through the implementation of features within the architectures described earlier.

A. Mobile App

The mobile app has been developed using native Java, tailored for Android devices. It is designed to run on devices with a minimum SDK version of API 29 and a target SDK version of API 33.

As previously mentioned in the **II-A** section, we structured the mobile application into two activities. These were mainly implemented by us using the drag-and-drop mode on the design interface provided by Android Studio, as well as by modeling the corresponding XML code. We then created the corresponding Java classes, allowing us to implement all the required functionalities.

About the Auth Activity

Regarding the Auth Activity, we needed to implement:

1) Login and sign-up functionalities for the system. Through Firebase authentication, we provided users with the ability to register with an email and password. On the other hand, if the user is already registered on the platform, they can log in with their credentials. Furthermore, if the user had already logged in without logging out before closing the application, the app takes into account the last login by directly opening the main activity without requiring the re-entry of credentials.
2) Requesting and managing access to the necessary permissions to ensure the proper functioning of the application. In particular, two methods were used for this:
   - `initRequestPermissionsLauncher()`: initializes the launcher used to request permissions if the user has not granted the necessary authorizations. If not all permissions have been granted, it displays an alert dialog allowing the user to open the application details settings to make the necessary modifications.
   - `checkAllPermissions()`: checks that all required permissions have been properly granted. If not, it calls the launcher, initialized through the aforementioned method, to display the alert dialog on the screen.

About the Main Activity

The Main Activity serves as the core of the application, handling various crucial aspects such as map management, notification reception and sending location information to the backend.

In the `onCreate` method, a series of initial operations are performed. Among them are the initialization of notification channels, configuration of the `LocationManager` and `FusedLocationProviderClient`, setup of the `Retrofit` service for HTTP requests, initialization of references to the Firebase Realtime Database, preparation of the `MapView`, and initialization of services for activity recognition (walking or car). Finally, methods are defined to be notified when a specific node in the Firebase Realtime Database is modified. These operations lay the essential foundations for the proper functioning of the application at startup.

In the `onStart` method, a `BroadcastReceiver` is registered for alert notifications, and the GPS status is checked.

In particular, the MainActivity operates as follows:

It sends constant updates to the backend via Retrofit, containing the user's username, their position, and the detected activity. This information is obtained by leveraging the APIs for Activity Recognition and the FusedLocationProviderClient service to retrieve location information. When a new geofence is added, the backend inserts it into the Firebase Realtime Database and updates the state of each user, indicating their involvement in the new alarm (inside the geofence, within 1km of the geofence, between 1 and 2km from the geofence). When information about the new geofence is added to the Firebase Realtime Database, the MainActivity is notified through the addChildEventListener method associated with the "notifications" node in Firebase (the node containing the list of present geofences). Within this method, the new geofence is added to a list of existing geofences and is subsequently drawn on the map.

The next update concerns the state of the individual user involved in the geofence. It is notified to the MainActivity through the addValueEventListener method associated with the "users" node in Firebase (the node containing the list of geofences involving the user and their state with respect to each one). With this update, we can determine which geofence involves the user. Once retrieved from the maintained global list of geofences, an alarm notification is created and sent to the involved user. During the notification sending process, a check is performed on the recognized activity (walking or car) to replay the notification via audio if the user is driving.

Finally, the user is also notified of a possible removal of an alarm (which will remove the associated geofence from the map), allowing them to check the current situation on the map and the remaining alarm geofences if any are present.

B. Backend

The backend was developed using the Flask framework. The choice to use Flask was driven by our familiarity with the framework and its lightweight and flexible nature, which allows us to easily integrate libraries and tools tailored to the project's needs. In particular, multiple libraries (specified in the requirements.txt file) were utilized to develop all the required functions.

Our backend operates as follows: after establishing a connection to our PostGIS database using the SQLAlchemy library and to our Firebase Realtime Database using the firebase-admin library, we implemented various routes accessible from the frontend and mobile app.

In particular, the mobile app's interaction with the backend is limited to the "*upload_location*" route. In this route, the user sends values such as username, latitude, longitude of their geographical location, and the detected activity (walking or driving). The backend, after creatinga Point object with the received coordinates, updates the record in the PostGIS database with the obtained information. It then sends geo-spatial queries to the database to calculate the user's position in relation on various geofences present (inside a geofence, within 1km from a geofence, between 1 and 2 km from a geofence). After obtaining the results of these queries, it inserts the geofences for which the user is involved and their respective states (inside, within 1km, between 1 and 2 km) into the Firebase Realtime Database (which will be used by the mobile app to obtain information about own state). The rest of the routes implemented in the backend are accessible from the frontend and involve managing geofences on the map, sending alarms, user clustering, and addressing the remaining project requirements.

We now provide a brief description of the various implemented routes:

add_geofence: route that allows the insertion of a new geofence into the system. It receives parameters from the frontend, including the geofence's title, alarms to be sent to users based on their positions relative to the geofence, and the coordinates defining the geofence. After receiving these parameters, the backend creates a polygon object with the obtained coordinates, generates a unique identifier for the geofence, and inserts it into the PostGIS database table called "geofence-information", along with other geofence-related details. Subsequently, the backend inserts the same identifier and associated geofence information into the Firebase Realtime Database under the "notifications" node, providing a reference to the various geofences accessible from the mobile app. Additionally, through geo-spatial queries executed on the PostGIS database, the backend calculates the users affected by the newly added geofence and their respective states (inside the geofence, within 1km, between 1 and 2km). The recalculated states and the geofences involving each user (if any) are then inserted into the Firebase Realtime Database. This enables users of the mobile app to obtain information about their own state based on these updates.

delete_geofence: route that allows us to delete a geofence. It receives the geofence's ID from the frontend and removes it from both the PostGIS database and the Firebase Realtime Database. Subsequently, using geo-spatial queries, it recalculates the new state for each user and updates it on the Firebase Realtime Database.

get_walking_user_data: This route returns the positions of users walking. Information about each user's recognized activity is stored in our PostGIS database. A query is simply used to obtain the positions of users with a recognized activity equal to "WALKING." It is used by the frontend to display only walking users.

get_car_user_data: route that returns the positions of users driving. Similar to the previous case, a query is used to obtain the positions of users with a recognized activity equal to "CAR," which are subsequently displayed by the frontend.

get_all_user_data: This route returns the positions of all users. A query is directly invoked to retrieve the positions of all users present in the "user-information" table, which are then displayed by the frontend.

get_geofence: route that returns the geofences present on the map. Specifically, it provides the geofence's ID, polygon, title, and the number of users within it (a value later used by the frontend for coloring the geofences based on the number of users inside).

$get\_cluster$: route that returns the cluster to which each user belongs. It receives the requested cluster number from the frontend. After creating a CSV file with user positions obtained through queries from the PostGIS database, it calculates the cluster_id of each considered user using k-means. It then returns the positions of various users and their respective cluster_id (subsequently used by the frontend for different cluster coloring).

$get\_cluster\_elbow$: route that returns the optimal number of clusters to use in k-means using the elbow method. Specifically, the "yellowbrick" library was used, particularly the "KElbowVisualizer" module. It explores a range of k values and automatically determines the best k based on the elbow method.

add_server: supporting route that is used externally to set the positions of the two required edge servers mentioned in the "Additional Components" section of the delivery. Later, the mechanism for automatic activation/deactivation of the POD containing the alarm generation system will be implemented on the node currently closest to the majority of users in the system. The frontend provides the positions of the servers to be added, which are then inserted into the "edge-information" table in our PostGIS database.

get_server: route that allows retrieving the positions of the previously chosen servers. The positions are obtained from the PostGIS database through queries and returned to the frontend, where they are displayed on the map.

Our backend, through communication with the PostGIS database and the Firebase Realtime Database, allowed for excellent management of data obtained from both the mobile app and frontend, integrating them efficiently and coherently.

C. Database

As mentioned earlier, we used two databases for the project: a PostgresSQL database with the PostGIS extension, and a Firebase Realtime Database. Specifically, the PostgresSQL database was implemented as follows.

After creating the "emergency-schema" schema, we proceeded to add the PostGIS extension, an extension that allows us to work with geospatial data. Through SQL commands, we subsequently created three tables: edge-information, geofence-information, and user-information. In particular:

- The edge-information table contains information regarding the location of the two edge servers.
- The geofence-information table contains information about the geofence IDs, their positions, titles, and creation timestamps.
- The user-information table contains information about usernames, user location, and detected activity (walking or car). The various tables will later be populated with information obtained from the frontend (adding or removing geofences) and the mobile app (user positions and detected activities).

We subsequently used a Firebase Realtime Database to communicate with the mobile app. In particular, within the Firebase Realtime Database, there are two parent nodes: notifications and users. Within the notifications node, various sub-nodes are named with the IDs of the various geofences present. Inside each one, there are alarms divided based on proximity to the geofence, the alarm title, and the coordinates of the points defining the polygon.

Inside the users node, there is a node for each user registered with the service, containing information about the various geofences that affect the user with their local state based on the geofence (inside, within 1 km, between 1-2 km). There is also a state node that contains the user's global state relative to the geofence that influences them the most.

These nodes will be modified through the backend when adding or removing a new geofence or updating the position of a user that changes their involvement with the various geofences present.

D. Frontend

The frontend was developed using HTML, CSS, and JavaScript. The "OpenLayers" library was also utilized to create an interactive map and display geo-spatial data. Finally, the Bootstrap framework was used for the visual design, enabling the creation of a clear and intuitive user interface, ensuring the most user-friendly experience possible.

Inside the "index.html" file, all visual components and structures of the website's user interface, such as the map, labels, textboxes, and various buttons, have been defined. On the other hand, the actual frontend logic, along with the corresponding requests to the backend, can be found in the "script.js" file.

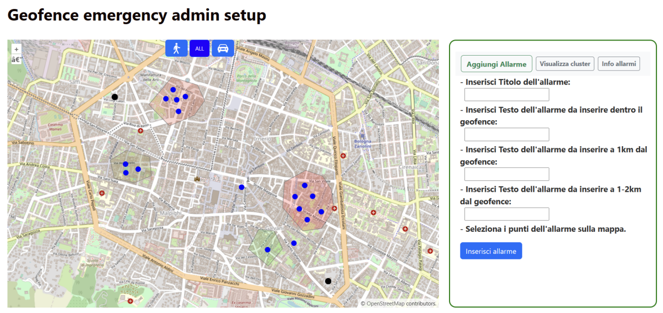

Geofence emergency admin setup
Aggiungi Allarme
Visualizza cluster
Info allarmi
- Inserisci Titolo dell'allarme:
- Inserisci Testo dell'allarme da inserire dentro il geofence:
- Inserisci Testo dell'allarme da inserire a 1km dal geofence:
- Inserisci Testo dell'allarme da inserire a 1-2km dal geofence:
- Seleziona i punti dell'allarme sulla mappa.
Inserisci allarme
ALL
© OpenStreetMap contributors.

Fig. 5. Frontend webpage.

Let's describe the logic present in the "script.js" file briefly. After defining all the necessary variables, we initialized the map with two layers. The first layer is an openStreetMap layer, providing a basic map visualization. The second layer is called "pointsLayer," displaying the points defining the boundaries of a new geofence to be inserted. These points can be added by clicking the left mouse button.

Once this is done, our script calls the functions $loadGeofence$ and $loadServer$, and using the $setInterval$

function, we specify to invoke the loadGeofence function every 3000 ms. This function, which we will delve into shortly, allows us to load the geofences present on the map. It is called every 3000 ms to ensure accurate updating of geofence colors based on the number of currently active users (continuously moving) within them.

Let's describe the three user visualization modes available in our frontend. It is possible to view walking users, users in cars, or all available users. You can switch between these views using the buttons at the top of the map. The change in visualization is achieved through HTTP GET requests to the backend, which will return the positions of the users based on the selected mode. Subsequently, a layer with the corresponding users, represented by blue dots, will be created and added to the map.

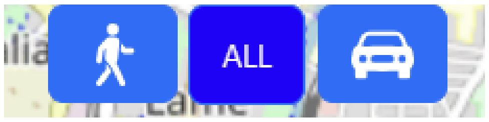
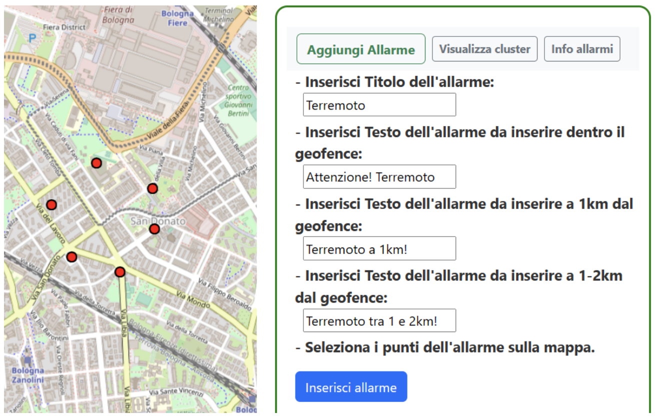


Fig. 6. User Visualization Modes.

Following this, we proceed to describe the various modes within the frontend: Adding an alarm, Viewing clusters, and Alarm Information. It is possible to switch between modes using the menu located on the right side of the interface.

The first mode we will analyze pertains to adding an alarm. Adding an alarm is possible by filling out the form in the right menu, which requires entering the alarm title, the text of the alarm to be sent to users inside the geofence, the text for users within 1km from the geofence, and the text for users between 1 and 2 km from the geofence. Finally, the user must select the points defining the geofence boundaries on the map (at least 3 points). By clicking "Insert Alarm", our geofence will be added to the list of active geofences, displayed on the map, and users from the mobile app within the respective geofence will be notified with the corresponding alarm. Let's see how we have implemented this mode.

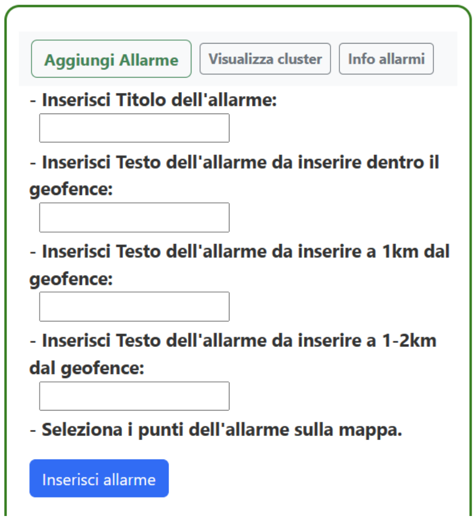


- Inserisci Titolo dell'allarme: Terremoto
- Inserisci Testo dell'allarme da inserire dentro il geofence: Attenzione! Terremoto
- Inserisci Testo dell'allarme da inserire a 1km dal geofence: Terremoto a 1km!
- Inserisci Testo dell'allarme da inserire a 1-2km dal geofence: Terremoto tra 1 e 2km!
- Seleziona i punti dell'allarme sulla mappa.

Fig. 7. Adding Geofence.

Regarding the insertion of the title and the various alarms to be received based on the user's location, we simply used a combination of labels and textboxes from which we later retrieve the filled text to send it to the backend. The insertion of points on the map was implemented using the "map.on('click')" function, allowing us to define behavior upon mouse click. Specifically, within the function, after using a flag to determine the current mode (in this case, "addmode"), we create a "pointFeature" object with the coordinates of the click and subsequently add it to a collection of points chosen for defining the geofence. Additionally, if we are clicking with the Ctrl key pressed, the last point added to the collection will be removed. These points' visualization on the map is possible thanks to the "pointsLayer," which, using an object represented by our selected points as the source, displays these points as red circles.

After clicking the "Insert Alarm" button, all this information will be sent to the backend, and the collection of these points will be emptied, making room for the geofence loaded through the loadGeofence function.



- Inserisci Titolo dell'allarme:
- Inserisci Testo dell'allarme da inserire dentro il geofence:
- Inserisci Testo dell'allarme da inserire a 1km dal geofence:
- Inserisci Testo dell'allarme da inserire a 1-2km dal geofence:
- Seleziona i punti dell'allarme sulla mappa.

Fig. 8. Adding alarm mode.

The $loadGeofence$ function does nothing more than make an HTTP GET request to the backend, specifically to the $get\_geofence$ route. From this route, it receives information about the geofence IDs present, the number of users inside them, the title, and the points defining their boundaries. Subsequently, it generates polygons associated with those points and colors the geofences based on the users inside, using a color normalization technique. This technique involves the product of the "r" and "g" components of the rgba color definition for the normalization defined earlier and 1-normalization, respectively. Information regarding the geofence's color, the defining polygon, ID, number of users, and title are stored within an array of features. Finally, the layer associated with these features is added to the map, making the geofences visible.

The second available mode is the cluster visualization. You can access this mode by clicking the "View Clusters" button in the right menu. Within this mode, we can perform two different actions. We can enter the number of clusters we want and click the "View Clusters" button to display them on the map. Additionally, we can click the "Elbow Method" button to calculate the optimal number of clusters using the elbow method. The result will be inserted into the textbox below. The handling of these operations occurs as follows:

Regarding the optimal number of clusters calculated using the elbow method, the frontend simply sends an HTTP GET request to the backend at the $get\_cluster\_elbow$ route, which will return the numerical value inserted into the textbox below.

Regarding the cluster request, it is made through an HTTP POST request to the backend, sending the requested number of clusters. The backend will return the positions of the users associated with the respective cluster_id. Once the frontend receives this information, it assigns a random color to each cluster_id and draws the various users on the map with the color associated with their cluster_id.

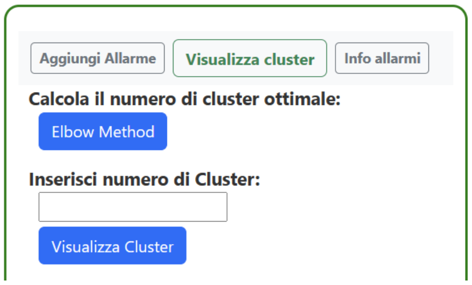


Fig. 9. Cluster Visualization mode.

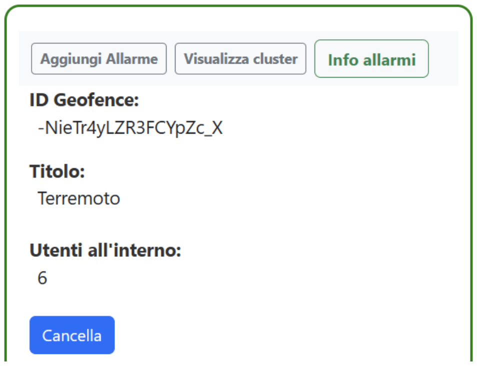
The last available mode is "Alarm Info". With this mode, we can obtain information such as the geofence ID, title, and the number of users inside it by clicking on the respective geofence. It is also possible, once a geofence is selected, delete it. The implementation of this mode was done as follows:

upon mouse movement, using the function $map.on('pointermove')$, we obtain information about the object we are hovering over. If it's a geofence, we store the associated feature object. On a mouse click, if we are in "Alarm Info" mode and are hovering over a geofence, we update the labels in the right menu with the geofence ID, the number of users inside it, and the title. If we click the "Delete" button, an HTTP POST request is sent to the backend with the geofence ID to be deleted. The backend will then proceed to delete it from the PostGIS database and Firebase. Subsequently, the geofence will disappear from the map thanks to the call to $load\_geofence$ function, which will reload all the present geofences.

E. Docker & Kubernetes

For the project, it was required to develop the components of the back-end and front-end within Docker containers orchestrated through the Kubernetes framework. Below, we show how this was achieved.

In particular, the Kubernetes framework was utilized through Minikube. Minikube is a solution that allows running



ID Geofence:
-NieTr4yLZR3FCYpZc_X

Titolo:
Terremoto

Utenti all'interno:
6

Fig. 10. Alarm Info mode.

Kubernetes clusters on a local machine. To use the back-end and front-end components within Docker containers orchestrated by Kubernetes, the following steps were necessary. Firstly, it was essential to create a Dockerfile, a script containing a series of instructions used to build a Docker image for each of the components to be used. Subsequently, .yaml files were created, which are used to define the configuration of the containerized applications and Kubernetes clusters for each component. Specifically, two .yaml files were produced for each component: a deployment file and a service file. The deployment file defines how the application should be deployed and managed, including information about the number of pod replicas, the container image, and other configurations associated with running the application. On the other hand, the service file defines a network service that can be used to expose the application outside the cluster.

Finally, to use these files, we executed some commands from the terminal for the following purposes: start Minikube, build the images associated with each component, and push them to DockerHub; apply the configuration regarding the previously defined deployments and services. Lastly, the "minikube tunnel" command is used to expose the services outside the cluster, and "minikube dashboard" is used to open the Minikube dashboard, providing information about the present deployments, the number of replicas, various services, and their IP addresses to access it.

In this way, it will be possible to access the various services outside the cluster.

F. Distributed Edge System with automatic activation/deactivation

For the implementation of this point, the following approach was followed: Firstly, a Python script was developed to obtain a reference to the two deployments (edge) implementing the backend through the Kubernetes APIs. Subsequently, once the positions of the two simulated edge servers were defined, it retrieved them through a query to the PostGIS database. Afterward, it counted the number of users closer to one server rather than the other using a geo-spatial query. Once this was done, based on the query result, the script proceeded to specify to zero the number of replicas for the deployment with the fewer users nearby, and to 1 the the number of replicas for the other deployment. Thanks to this script, it is then possible to activate the pod containing the alarm generation system on the node currently closer to the majority of users in the system and deactivate the other.

To use this script within Kubernetes, it was necessary to define additional .yaml files. In particular, three new files were defined.

The first one, edgecontrol-deployment.yaml, is a deployment created using the image containing the script made earlier.

The second one, role.yaml, is a file that defines a role within Kubernetes – essentially a set of rules specifying which actions can be performed on which resources within a specific namespace. With this role, we indicate the ability to read, list, monitor, and make partial changes to resources of type "Deployment." This will be used to obtain references to the two deployments and to modify the number of replicas.

Finally, role-binding.yaml is a file that allows associating a role with a specific user. In particular, with this file, we assign the previously created role to the default account.

After creating these files, we proceeded to build the associated images and push them to DockerHub. We then applied the configurations related to the role.yaml file, role-binding.yaml file, and finally to the edgecontrol-deployment.yaml file. Once activated, the deployment will automatically manage the activation/deactivation mechanism of the edge servers.

IV. Conclusion

In conclusion, as requested, a platform for sending geo-localized information related to danger situations has been implemented. Alarm messages reach all users within the geofence and its vicinity, taking into account users' modes of transportation to diversify message reception. The components have been developed within Docker containers and orchestrated using the Kubernetes framework, as specified. Additionally, an automatic activation/deactivation mechanism for the POD containing the alarm generation system on the edge node currently closest to the majority of users in the system has been implemented.

The implemented platform could offer numerous benefits, including a significant improvement in public safety through the timely dissemination of crucial information in dangerous situations. Geo-localization ensures precision in alerting users, ensuring that those in the vicinity of potential danger promptly receive relevant information. Furthermore, diversifying message reception based on users' modes of transportation optimizes the effectiveness of communication, adapting it to each individual's specific needs.

This platform could represent a significant step toward better emergency management, contributing to maintaining safer and well-informed contexts.

References

[1] J. Schiller and A. Voisard, Location-based services. The Morgan Kaufmann series in data management systems. Amsterdam Heidelberg: Elsevier [u.a.], 2004.

[2] U. Bareth, A. Küpper, and B. Freese, Geofencing and Background Tracking: The Next Features in LBSs, in GI-Jahrestagung 2011: Proceedings of the 43rd Annual Conference of the German Computer Science Society, Bonn, Germany, 2011, pp. 205-216.

[3] Y. Asim, M. A. Azam, M. Ehatisham-ul-Haq, U. Naeem, and A. Khalid, 'Context-Aware Human Activity Recognition (CAHAR) in-the-Wild Using Smartphone Accelerometer', IEEE Sensors J., vol. 20, no. 8, pp. 4361–4371, Apr. 2020, doi: 10.1109/JSEN.2020.2964278.

[4] 'Activity Recognition API', Google for Developers. Accessed: Nov. 06, 2023. [Online]. Available: https://developers.google.com/location-context/activity-recognition