# Home Zone Analyzer

Context Aware System Project - Track 1

Simone Rinaldi
*id 0001140193*
*simone.rinaldi8@studio.unibo.it*

Valerio Desiati
*id 0001138895*
*valerio.desiati@studio.unibo.it*

## Introduction

This document will cover the development of software focused on spatial data management. The aim of the project is to create a platform capable of providing property purchase suggestions in the city of Bologna, based on user preferences.

User preferences are collected through a questionnaire that allows them to specify their specific needs regarding the proximity and density of various Points of Interest (PoI), such as schools, hospitals, shops, and other essential services.

Once the information is collected, the tool displays on a map not only the properties of interest but also the candidate areas, offering a wide range of details and additional data. This way, users can easily view and compare the information, facilitating a more conscious and informed decision-making process.

## Tool and libraries

List of the technologies (frontend and backend) used for the development of the project:

1) **PostgreSQL**
   PostgreSQL is a relational database management system. It is known for its compliance with SQL standards, extensibility, and robustness. It supports multiple data type, such as JSON data, arrays, etc., and allows the addition of user-defined extensions and functions.
2) **PostGIS**
   PostGIS is a PostgreSQL's extension that adds support for geographic objects, allowing for spatial data management within a relational database. It enables complex geospatial queries execution, such as distance calculations, intersections and more within the database. This makes PostGIS an ideal choice for GIS applications and any project that requires the handling of geographic data.
3) **Mapbox**
   Mapbox is a mapping and geolocation platform that provides APIs for custom maps creation and geospatial applications. Unlike other map providers, Mapbox stands out for its flexibility and the ability to create highly customized maps using vector and raster data.
   Mapbox is used in applications that require a high level of interactivity and customization, such as ride-sharing apps, fitness trackers, and more.
4) **Leaflet**
   Leaflet is a lightweight, open-source JavaScript library for creating interactive maps for the web. It supports a wide range of features, including map layers, markers, pop-ups, and overlays. It is compatible with various map providers like Mapbox and OpenStreetMap, and can be extended with numerous plugins.
5) **Node.js**
   Node.js is an open-source, cross-platform JavaScript runtime that allows JavaScript code execution outside of a browser. Node.js is widely used in server-side application development, RESTful APIs, and microservices. Its ecosystem, supported by the npm package manager, offers thousands of libraries and tools that facilitate rapid application development.
6) **React + TypeScript + Vite**
   React is a JavaScript library developed for building user interfaces. It is component-based and known for its efficiency. TypeScript, a superset of JavaScript, adds static typing, which enhances safety and robustness in React application development by helping to prevent runtime errors. Vite is a build tool that offers an extremely fast development environment with minimal configuration, particularly optimized for React projects.
   React, TypeScript, and Vite provide a powerful stack for front-end development, combining performance and scalability.
7) **Express**
   Express is a Node.js' framework used to create web applications and APIs. Express makes it easy to define routes, middleware, and handlers for HTTP requests. It is widely used for developing web servers and microservices due to its simplicity and the wide availability of middleware that extends its functionality. Express integrates well with various databases, template engines, and other tools, making it an excellent choice for developers who want to build fast and scalable web applications.
8) **Axios**
   Axios is a JavaScript library used to create HTTP requests. It is used in applications like React, Vue, and Angular to send requests to the server and receive JSON data. Axios supports all typical HTTP operations such as `GET`, `POST`, `PUT`, and `DELETE`, and includes features for error handling, request and response interception,

and authentication management.

9) **Flask**
   Flask is a microframework for web development in Python, known for its simplicity and flexibility. It supports extension through external modules for functionalities such as database integration, authentication, and session management.
10) **Pandas**
    Pandas is a Python library for data analysis. It allows for performing complex operations on data, such as merging, transforming, cleaning, and aggregating, efficiently and with intuitive syntax. It is widely used for analyzing large datasets and transforming them into formats suitable for analysis and visualization.
11) **GeoPandas**
    GeoPandas is an extension of the Pandas library, designed to facilitate the manipulation and analysis of geospatial data. GeoPandas introduces new data types for representing geometries such as points, lines, and polygons, and provides functionality for geospatial operations like area calculations, spatial reference system management, and geometry intersections.
12) **SQLAlchemy**
    SQLAlchemy is a Python library that provides an SQL toolkit and an Object-Relational Mapper (ORM) for interacting with databases. With SQLAlchemy, you can define database tables as Python classes and work with instances of these classes instead of writing SQL. This approach allows the changing of the database backend without modifying the code.
13) **ESDA**
    ESDA (Exploratory Spatial Data Analysis) is a tool used to analyze and visualize the spatial distribution of data, identifying patterns, clusters, and spatial anomalies. ESDA can include methods such as Moran's I, which measures spatial autocorrelation, and LISA (Local Indicators of Spatial Association) charts that identify local clusters.
14) **Docker**
    Docker is an open-source platform that allows for the automation of application deployment within containers—isolated environments that include everything needed to run an application. Docker containers are lightweight, portable, and independent of the underlying infrastructure, making them ideal for continuous deployment, collaborative development, and application scalability. Docker facilitates the creation of consistent and reproducible development environments, reducing conflicts between software dependencies.
15) **Kubernetes**
    Kubernetes is an open-source platform for container orchestration. It allows for the management of clusters of virtual or physical machines on which containers run, offering advanced features such as load balancing, autoscaling, and resource management. With Kubernetes, you can automate the application lifecycle, ensuring high availability, error recovery, and optimal workload distribution.

## Design and Implementation

The implementation uses the PERN stack, based on PostgreSQL, Express.js, React, and Node.js, which is widely used for full-stack web application development.

### A. Database

The DBMS chosen for this project is PostgreSQL, used with the PostGIS extension for storing, managing, and utilizing spatial data.
The database consists of 16 tables, of which:

- *apartments* table, contains the specifications of each apartment for sale (cost, location, etc.);
- *neighborhoods* table, contains the list and spatial data of the neighborhoods in the city of Bologna;
- *user_votes* table, contains the user's responses to the questionnaire on Points of Interest (hereinafter referred to as PoI);
- Tables concerning the specifications and spatial data of the examined PoIs, such as: bike racks, electric vehicle charging stations, urban bus stops, green areas, hospitals, libraries, parks, pharmacies, schools, cinemas, and theaters.

### B. Frontend

The frontend part of the web app was developed using React.
Specifically, the implementation is organized into the files described below (only the **essential** files for the operation of the web app will be examined).

*1) Survey.tsx:* It is a React component written in TypeScript that implements a form to collect user preferences regarding various Points of Interest (PoI), allowing users to express the importance of different aspects of the neighborhood, such as the presence of green areas, bus stops, sports facilities, etc. An interface `SurveyProps` is used to define the `onSubmit` property, a callback function that is called upon completing the survey, passing a map of responses (`Map<string, number>`) that associates each PoI type with the score selected by the user.
The component uses a constant array of objects called `questions`, each containing:

- `text`: string representing the question displayed to the user;
- `poi`: identifier representing the PoI type associated with the question.

For each question, the component allows the user to select an importance score on a scale from 0 to 5. When the user completes the survey and presses the submit button, the component calls the `onSubmit` function passed via props, sending the map of responses.

*2) Map.tsx:* This React component implements a map using the Leaflet library, OpenStreetMap tiles, and various plugins. The map is initialized and centered on the city of Bologna and is structured as follows:

- It includes various feature groups to manage elements such as drawn objects, isochrone areas, apartments, neighborhoods, and clusters.
- PoIs are categorized into groups.
- Drawing functionalities are also implemented, allowing users to draw polygons on the map that can be sent to the server to, for example, find apartments within the drawn area.
- Apartments and PoIs are indicated by markers, which are grouped using Leaflet's `MarkerClusterGroup` for better visual management. Each marker has a popup with relevant information such as price, score, neighborhood, and address of the apartments.
- The MapBox Isochrone API is used to create a layer that shows areas reachable within a specified time (as defined by the user) with the specified mode of transportation (walking, cycling, driving).

For the data loading related to apartments and neighborhoods, the component uses Axios. Note how the neighborhoods are color-coded based on a scoring system derived from the data: `D3.js` is used to create color scales that visually represent different scores on the map, helping to differentiate neighborhoods and apartment scores.

Use of React Hooks:

- `useEffect`: manages the lifecycle of loaded data and updates the map when data changes, such as apartments, neighborhoods, and the Isochrone area;
- `useRef`: maintains references to map elements, such as feature groups or marker clusters;
- `useState`: manages state, such as Isochrone coordinates, travel time, transportation mode, and loading status.

*3) App.tsx:* This React component acts as the application entry point. The application is structured to include three main components: `Survey`, `Map` (analyzed above), and `StartingLoading` (it's a loading screen, implemented to display the map only after all data has been loaded). The main purpose is to manage the display of sub-components based on the application's state, particularly whether it is loading, has received data from the survey, or is ready to display a map based on the survey results.

The component uses the `useState` hook to manage two state variables:

- `surveyData`: contains the survey data. Once the survey is completed and the data is submitted, state will be updated with the survey results;
- `loading`: manages the application's loading state, which controls whether the loading component (`StartingLoading`) is displayed.

The `useEffect` hook is used to simulate a loading process, setting a 2-second timeout to simulate an initial loading phase. After the delay, it sets the `loading` state to `false`, thus removing the loading screen and displaying the survey or map depending on the `surveyData` state.

The `handleSurveySubmit` function is passed as a prop to the `Survey` component. It is designed to handle the event when the survey is completed and submitted:

- It accepts `data` of type `Map<string, number>` as an argument, representing the survey results;
- Upon receiving the data, it updates the `surveyData` state with the new data, triggering a re-render to display the `Map` component with the survey data.

### C. Backend

*1) queryRanking.js:* In this JavaScript file, two string constant and exportable variables are defined, which specify two SQL queries, `GET_APARTMENTS_QUERY` and `GET_NEIGHBOURHOOD_RANKING`, designed to rank apartments based on proximity to various PoIs and user-defined preferences from the survey.

The queries calculate scores for each apartment based on distances from PoIs, weighted according to user preferences, and return apartments and neighborhoods sorted by these scores.

The `GET_APARTMENTS_QUERY` query calculates a score for each apartment based on proximity to various PoIs, weighted according to user preferences. The result is a list of apartments ordered by their scores, with lower scores indicating apartments that are optimal according to user preferences. The query is structured as follows:

- Calculation of distances from PoIs (`poi_distances`)
  This CTE (Common Table Expression) calculates the minimum distance from each apartment to the nearest PoI of each type.
  For each type of PoI, the query uses a `CROSS JOIN` or a `LEFT JOIN` combined with spatial functions to calculate the minimum distance between each apartment and the type of PoI.
  For PoIs such as hospitals and bus stops, which require specific conditions (e.g., the apartment must have at least 1 hospital within 500 meters or 2 bus stops within 300 meters), custom distance values (1 or 5) are assigned based on these conditions.
- Calculation of weighted distances (`weighted_distances`)
  This CTE calculates the weighted distance for each apartment-PoI combination by multiplying the distance (in kilometers) by the user's importance rating for that type of PoI. The query joins `poi_distances` with the `user_votes` table to obtain the user's rating for each type of PoI and calculates the weighted distance accordingly.
- Calculation of apartment scores (`apartment_scores`)
  This CTE sums all the weighted distances for each apartment to calculate a total weighted distance score. A lower score indicates that the apartment better meets the user's preferences based on proximity to preferred PoIs.

- Final selection
  The main query selects apartments and their attributes (ID, geometry, price, neighborhood, address) along with their score calculated by the `apartment_scores` CTE. The apartments are then ordered by their total weighted distance score in ascending order.

The `GET_NEIGHBOURHOOD_RANKING` query calculates the apartments' average score within each neighborhood, providing a neighborhoods' ranking based on the average score of the apartments located within them. This ranking can help users choose a neighborhood by providing information on which ones best meet their preferences in terms of apartment locations and nearby PoIs.

The query is structured as follows:

- Calculation of distances from PoIs (`poi_distances`)
  Similar to the `GET_APARTMENTS_QUERY`, this CTE calculates the minimum distances from each apartment to various types of PoIs (the structure and logic are the same, focusing on calculating distances for each apartment to different PoIs).
- Calculation of weighted distances (`weighted_distances`)
  This CTE calculates the weighted distances using the user's ratings for each type of PoI, following the same logic used in `GET_APARTMENTS_QUERY`.
- Calculation of apartment scores (`apartment_scores`)
  Weighted distances are summed for each apartment to calculate a total score, as previously described.
- Apartments with Neighborhoods (`apartments_with_quartieri`)
  At this stage, neighborhood information is added to the apartment data by joining the apartments with the neighborhood table using a spatial relationship provided by PostGIS (`ST_Within`), which checks if an apartment is within the geometry of a neighborhood.
- Neighborhood Ranking
  The main query calculates the average score for each neighborhood by averaging the scores of the apartments within that neighborhood, then orders the neighborhoods by their average score in ascending order, ranking them from most to least desirable.

Both queries rely on spatial operations to calculate distances between apartments and PoIs. Additionally, the queries dynamically adjust scores based on preferences, making the rankings highly personalized.

*2) index.js:* The backend application uses the Express framework to handle `HTTP` requests, along with the `pg` module to connect to a PostgreSQL database.
The application utilizes the `GET_APARTMENTS_QUERY` and `GET_NEIGHBOURHOOD_RANKING` queries declared in the `queryRanking.js` file.
To enable CORS requests `app.use(cors())` is used, allowing the server to respond to requests from different domains, enhancing API access flexibility.

Implemented endpoints:

- `POST /` - Survey votes insertion
  This endpoint receives JSON data representing votes for various types of PoI. Before inserting new votes, the `user_votes` table is cleared of previous votes.
  For each vote, an insertion query is executed using parameterized queries to prevent SQL Injection.

- `GET /apartments` - Retrieve apartment data
  This endpoint executes the `GET_APARTMENTS_QUERY`. If the query is successful, it returns the results in JSON format.
- Endpoint for each type of PoI
  Endpoints for schools, sports facilities, pharmacies, bike racks, green areas, hospitals, libraries, electric charging stations, theaters/cinemas, playgrounds, and bus stops follow a similar structure, each executing a specific query to a corresponding table in the database.
  The data returned is formatted in GeoJSON using the SQL function `ST_AsGeoJSON` applied to the geometries of the objects.
- `GET /quartieri` - Neighborhood ranking
  Executes the `GET_NEIGHBOURHOOD_RANKING` query imported from `queryRanking.js`.
  Returns the results of the query, which may include neighborhood ranking data based on criteria defined in the separate query file.
- `POST /shape/polygon` - Apartments within the polygon
  This endpoint receives a polygon in GeoJSON format and searches for apartments within this polygon. The query uses `ST_Contains` to check if the geometry of the apartments is located within the polygon sent by the client. The results are sent in JSON format, including details such as neighborhood, price, and address of the apartments.
- `POST /isochrone` - Isochrone map
  This endpoint executes a query using the GeoJSON polygon sent by the user to find various PoIs contained within the polygon.
  The query is divided into several subqueries, each responsible for a type of PoI. The results include the type of PoI, the geometry, and other relevant information, returning a set of markers.

*3) app.py:* This backend application, developed with Flask, is designed to perform spatial and statistical analyses on geographic data from the database.
The application includes various endpoints to calculate the Moran's I index, a statistical measure used to assess the degree of spatial autocorrelation among geographic data.
In other terms, it measures how similar (or dissimilar) the values of a variable are to each other in a geographic space based on their location.

Moran's I index is defined as:

$$I = \frac{n}{W} \cdot \frac{\sum_i \sum_j w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_i (x_i - \bar{x})^2}$$

where:

- $n$ is the number of spatial units;
- $W$ is the sum of all spatial weights $w_{ij}$;
- $w_{ij}$ represents the spatial weight between units $i$ and $j$;
- $x_i$ and $x_j$ are the values of the variable for units $i$ and $j$;
- $\bar{x}$ is the mean value of the variable.

Here's how to interpret the results of Moran's I:

- $I > 0$ indicates positive spatial autocorrelation (similar neighbors). This means that nearby geographic units tend to have similar values for the variable being measured;
- $I < 0$ indicates negative spatial autocorrelation (dissimilar neighbors). This means that nearby geographic units tend to have dissimilar values for the variable being measured;
- $I = 0$ suggests the absence of spatial autocorrelation (random distribution). This implies that the values of the variable are randomly distributed across the geographic space, with no discernible pattern of similarity or dissimilarity among neighboring units.

The application was built using:

- Flask: used to create the web application and handle `HTTP` requests;
- GeoPandas: used to work with geographic data in GeoDataFrame format;
- Pandas: used alongside GeoPandas for data manipulation, including non-geographic data;
- SQLAlchemy: used for connecting to the database. It uses the `postgresql+psycopg2` connection string to communicate with the PostGIS database;
- libpysal: used to calculate spatial weights based on KNN (K-Nearest Neighbors);
- esda: used to calculate Moran's I index;
- Shapely: used for geometric functions to determine distances between spatial objects;
- Flask-CORS: used to enable CORS requests, allowing the app to respond to requests from other domains.

Implemented functions:

- `get_engine()`
  The function returns an SQLAlchemy engine for connecting to the PostgreSQL database.
  The connection string specifies the user, password, host, port and database name.
- `load_data()`
  The function loads various spatial data from the database using GeoPandas and the SQLAlchemy engine. Queries defined:
  - `apartments`: selects the prices and geometries of apartments;
  - `neighborhoods`: selects the codes, names, and geometries of neighborhoods;
  - Query to select, individually, the geometries of PoIs.

  Each query is executed and returns a GeoDataFrame for each dataset.
- `calculate_distances(apartments_df, points_gdf)`
  The function calculates the minimum distance between each apartment and the nearest PoI using `nearest_points` to determine the closest PoI to each apartment.
  Distances between apartments and the nearest points are calculated using Shapely.

Implemented endpoints:

- `GET /calculate_morans_i` - Moran's I index calculation
  Calculates the Moran's I index for apartment prices, using distances from PoIs as a spatial correlation factor:
  1) Load apartment and neighborhood data from the database;
  2) Combine all PoIs into a single GeoDataFrame;
  3) Calculate the distance from each apartment to the nearest PoI and add it to the GeoDataFrame;
  4) Perform a spatial join between apartments and neighborhoods to assign each apartment to its respective neighborhood;
  5) Calculate spatial weights using K-Nearest Neighbors;
  6) Compute the Moran's I index by passing apartment prices and spatial weights;
  7) Return the results: Moran's I index, p-value, and z-score.
- `GET /check_connection` - Check database connection
  Actions:
  1) Attempt to connect to the database using the SQLAlchemy engine.
  2) If the connection is successful, immediately close it and return a success message.
  3) If the connection fails, catch the error and return a detailed message.

Finally, the application is started on port 5000.

## Testing and Results

Initially, the tool's interface presents a questionnaire with ten questions to the user, aimed at gathering information about the importance of each Point of Interest (PoI). Fig. 1

Once the survey is completed, the rendering of the map of the city of Bologna proceeds, with different layers applied: Fig. 2

- Neighborhoods;
- Apartment clusters;
- Isochrone map (with default options, so the isochrone map shows the area reachable within a five-minute walk).

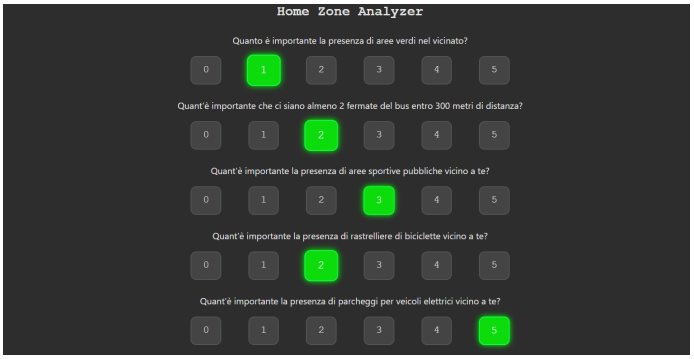


Fig. 1. Questionnaire visualization.

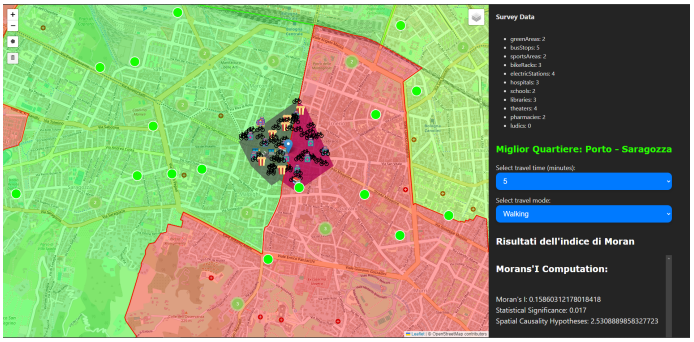


Fig. 2. Map visualization.

On the right side of the screen, there are:

- The survey responses;
- The best neighborhood suggestion, calculated based on the survey;
- Isochrone map settings, customizable by the user.

Scrolling down, the Moran's I index calculation is displayed, specifying Fig. 3:

- Moran's I index;
- Statistical significance;
- Spatial causality hypothesis.

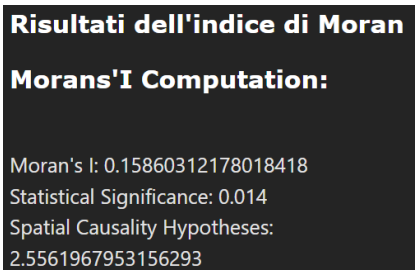


Fig. 3. Moran's I index calculation.

The user can choose which elements to display on the map with the toggle in the upper right corner. Fig. 4

It is also possible for the user to select a marker (of any type, apartment or PoI) to view its details. Fig. 5-6

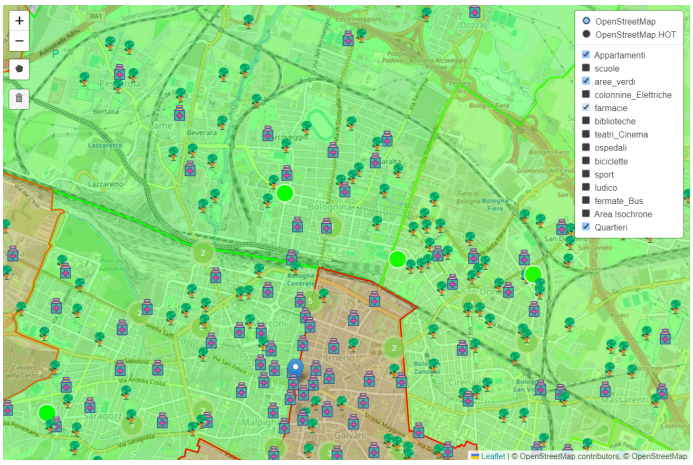


Fig. 4. Choices of elements to display on the map.

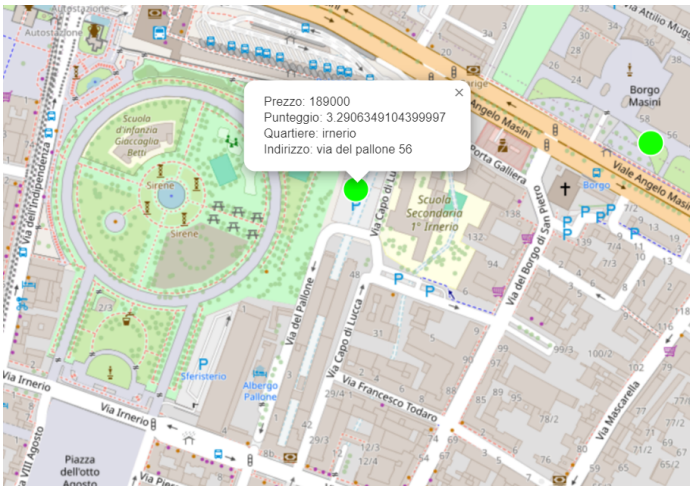


Fig. 5. Viewing apartment details.

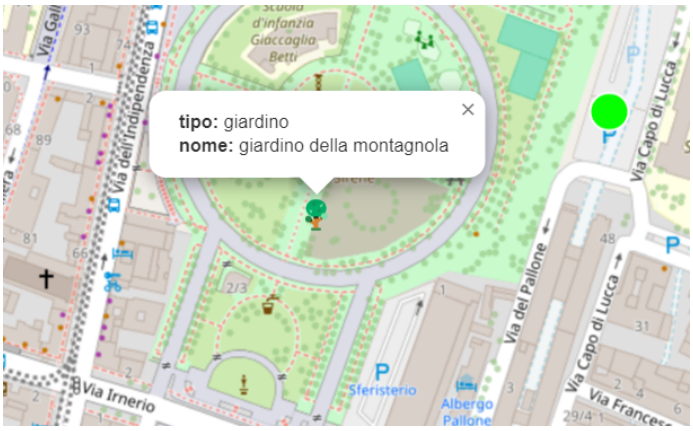


Fig. 6. Viewing PoI details.