NEARBY MARKER IN ANGULAR

1.General Information

This project represents a blend of technology and history, designed to provide users with a dynamic and intuitive way to explore historical events through interactive maps. Utilizing advanced technological tools such as *Angular* for building the user interface and *Leaflet* for map functionalities, the platform allows users to discover historical events tied to specific geographic locations.

The core functionality of the application enables users to select a point on the map and receive an overview of key historical events that occurred in the vicinity of the chosen location. The information includes battles, political changes, cultural events, and notable historical figures, offering users a rich and detailed insight into the past of a particular region. This functionality is powered by the integration of a database containing meticulously organized historical records.

The technological foundation of the project includes *Angular* as the primary frontend framework, ensuring fast and responsive rendering of the interface and providing a seamless user experience. The maps are implemented using *Leaflet*, a well-known open-source library for working with geographically oriented applications, offering flexibility in handling interactive maps, markers, and data layers.

One of the main challenges during development was ensuring the accuracy and relevance of historical data and properly categorizing it to make it easily searchable and understandable for end users. Additionally, optimizing the application's performance to handle large datasets was a key aspect, especially when rendering map layers in real-time.

The goal of this project is not only to provide insights into historical events but also to spark interest in exploring the past through a modern and interactive approach. The application has potential uses in education, tourist guides, and as a tool for history enthusiasts who want to gain a deeper understanding of historical dynamics in a given area.

The project is designed as a flexible and scalable platform, with possibilities for further development, such as adding additional layers of information, integrating advanced recommendation algorithms, and expanding functionality for different target audiences.

2. Technologies

The project leverages modern web development technologies to deliver a responsive and interactive user experience. At its core, the application is built using *Angular*, a powerful JavaScript framework developed by Google. Angular provides a robust structure for creating dynamic single-page applications (SPAs) and ensures efficient rendering and seamless interaction across various devices. Its component-based architecture simplifies development and maintenance while allowing for reusable and modular code.

For mapping functionalities, the project uses *Leaflet*, a lightweight and flexible open-source library for interactive maps. Leaflet provides essential tools for handling geographical data, adding markers, and customizing map layers. To integrate Leaflet into the Angular framework, the project uses the library *@asymmetrik/ngx-leaflet*, which acts as a bridge between Angular's reactive ecosystem and Leaflet's mapping capabilities. This library enables easy embedding of interactive maps within Angular components, providing smooth two-way data binding and event handling.

For data storage and management, the project utilizes *Firebase*, a cloud-based platform by Google. Firebase offers a real-time database solution, ensuring fast and reliable data retrieval and synchronization. The database is used to store historical data records, including events, locations, and metadata, making it accessible in real-time to the application. Additionally, Firebase simplifies authentication and backend management, allowing the project to focus more on frontend development and user experience.

Together, these technologies form a cohesive and scalable stack, enabling the application to handle complex tasks such as map rendering, real-time data fetching, and seamless interaction. The combination of Angular, Leaflet, ngx-leaflet, and Firebase not only ensures efficient performance but also sets the foundation for future expansion of the platform.

3. Project Architecture and Technology Stack

The application is developed using *Angular*, a powerful framework for building dynamic web applications. The project follows a modular structure to ensure scalability, maintainability, and clarity of the codebase. At the highest level, the application is organized into two primary folders: *core* and *features*.

The *core* folder contains reusable elements that are foundational to the entire application. Within it, there are subfolders like *pipes* and *services*. The *pipes* folder holds custom data transformation logic, while the *services* folder includes shared services for managing data, interacting with APIs, and providing utility functions. These core elements ensure consistent behavior across the application and reduce code duplication.

The *features* folder, on the other hand, is dedicated to the specific functional modules of the application. Each module within *features* is self-contained, with its own components, templates, and styles, adhering to Angular's modular approach. This separation allows for easier testing, debugging, and future expansion of the application.

3.1 **Features Folder Structure**

The *features* folder contains several modules that define the core functionalities of the application. Each module is independent and contains specific components, services, and other resources related to a particular feature. These modules are organized to allow easy expansion of the application, as well as simple maintenance and testing. Within the *features* folder, the following modules are included:

- **geo-json** – for handling geo-JSON data.
- **map** – responsible for the map and its interactive features.
- **marker** - a module that handles the markers displayed on the map.

- **marker--info** – a module for managing markers and displaying marker information on the map.
- **search-form** – a module that enables searching and filtering data based on certain parameters.

Each of these modules serves a distinct purpose, and together they form the functional framework of the application. In the following sections, we will describe each of these modules in detail and explain their role in the overall system.

3.1.1 geo-json

The `GeoJsonComponent` is responsible for loading and displaying GeoJSON data on a Leaflet map. It fetches GeoJSON files dynamically based on the specified year and adds them to the map. The component provides functionality to remove the currently displayed GeoJSON data and reload new data based on user input. Additionally, the component applies dynamic styling to the GeoJSON features.

Inputs

- **`year: string`**
  The year for which GeoJSON data will be loaded. The year input will be used to fetch the closest available GeoJSON data from the predefined list of available years.

Properties

- **`data: any[]`**
  A placeholder for the fetched GeoJSON data (though it is not directly used in the component, it can be extended or modified if needed).
- **`availableYears: number[]`**
  An array of years that are available for GeoJSON data fetching. When a user requests data for a specific year, the component finds the closest available year in the array.
- **`geoJsonLayer: Layer | null`**
  A private property that holds a reference to the current GeoJSON

layer on the map. This reference is used to remove or replace the layer when new data is loaded.

Methods

- **`getClosestYear(targetYear: number): number`**
  This method takes a target year and returns the closest available year from the `availableYears` array. It compares the absolute difference between each available year and the target year and returns the closest match. This method ensures that if the exact year is not available, the component will load the data for the nearest year.
- **`loadGeoJsonForYear(year: number): Promise<void>`**
  This asynchronous method is responsible for fetching and loading GeoJSON data for the given year. It uses the `HttpClient` to make a GET request to fetch the GeoJSON file for the closest available year. Upon success, it adds the GeoJSON data to the map with dynamic styling. If the data fetch fails, an error is logged to the console.
- **`removeGeoJson(): void`**
  This method removes the currently loaded GeoJSON layer from the map. It is called before loading new GeoJSON data to ensure that the map only displays one layer at a time.
- **`getStyleForFeature(feature: any): any`**
  This private method returns the style object for each feature in the GeoJSON data. It dynamically determines the style based on the properties of each feature, such as `color`, `weight`, `opacity`, `fillColor`, and `fillOpacity`. This allows for a flexible and customizable appearance of GeoJSON features on the map.

3.1.3 map

Overview

The `MapComponent` is responsible for initializing and managing the Leaflet map in the application. It handles dynamic map loading based on location data, sets up map layers, and displays the location's markers and associated data. The component integrates with various services to fetch city-specific data, process it, and render it interactively on the map.

Inputs

- **None**
  This component does not have any direct input properties but listens to changes in the route to dynamically load and update the map based on location.

Properties

- **`markersData: any[]`**
  A placeholder array that stores marker data fetched from external sources (though the actual use may vary depending on how the data is processed).
- **`map: Map | undefined`**
  Holds the reference to the Leaflet map object. This reference is used to perform actions like adjusting the map view, adding layers, or manipulating markers.
- **`location: string | null`**
  A string that holds the location (city name) fetched from the route parameters. The component uses this location to fetch relevant data and set the map’s center.
- **`centralMarker: any`**
  Stores the data for the central marker on the map. This marker represents the focal point of the map, usually linked to the specified city or location.
- **`radius: any`**
  Represents the radius for the circle drawn around the central marker. It is used for visualization and setting boundaries for fetching data.

- **`origin: any`**
  The geographic coordinates (latitude and longitude) for the center of the map (typically the searched city or location).
- **`options: MapOptions`**
  Contains options to customize the Leaflet map, such as zoom levels, center position, and layers.
- **`options1: any`**
  Holds the options specific to the current location's map setup. This includes map layers, zoom level, and center coordinates based on the city's location.

Methods

- **`ngOnInit(): void`**
  This method is triggered when the component is initialized. It listens to route parameter changes and fetches the city-specific data based on the `location` provided. Upon successful city search, the method initializes the map options with the city's geographical coordinates and fetches relevant data from Firebase using `firebaseDataService`.
- **`onMapReady(event: Map): void`**
  This method is triggered when the Leaflet map is fully loaded and ready. It stores the map instance in the `stateService`, sets the map's view based on the city's coordinates and zoom level, and ensures the map size is correctly calculated.

HTML Structure

The HTML structure for the `MapComponent` includes the following key elements:

```
<div class="map-container" *ngIf="options1"
    leaflet
    [leafletOptions]="options1"
    (leafletMapReady)="onMapReady($event)">

  <app-marker></app-marker>
```

<app-search-form></app-search-form>

- **`<div class="map-container" *ngIf="options1"`**
  The `map-container` div contains the Leaflet map. The `*ngIf="options1"` directive ensures that the map is only displayed when the `options1` object is populated with valid map options.
- **`leaflet`**
  The `leaflet` directive is used to initialize a Leaflet map within the `div`. This directive binds the Leaflet map functionality to the DOM element, making it interactive.
- **`[leafletOptions]="options1"`**
  The `leafletOptions` input binding is used to pass the map options (such as layers, zoom level, and center position) from the component to the Leaflet map.
- **`(leafletMapReady)="onMapReady($event)"`**
  The `leafletMapReady` event is triggered when the map is fully initialized and ready. This event handler calls the `onMapReady()` method in the component to set up the map and trigger further actions (like adding markers or other layers).
- **`<app-marker></app-marker>`**
  The `app-marker` component is used to display markers on the map. It is added within the map container to render dynamic markers based on the data.
- **`<app-search-form></app-search-form>`**
  The `app-search-form` component is a search form that allows the user to search for cities or locations, which will update the map view dynamically. It is also placed within the map container for interaction with the map.

Dependencies

- **Leaflet**
  The component relies on the Leaflet library for rendering

interactive maps. It uses `tileLayer` for map tiles, `circle` for drawing a radius around the location, and various Leaflet functions to manipulate the map.

- **FirebaseDataService**
  This service fetches the necessary data from Firebase, such as historical or geographical data, based on the location coordinates.
- **FetchDataService**
  This service processes the fetched data and triggers further actions to update the map or markers accordingly.
- **CityService**
  The `CityService` is used to search for a city's geographical coordinates (latitude and longitude) based on the location string.
- **StateService**
  The `StateService` is responsible for managing the global state of the map, including storing the map instance and tracking if the map is ready.
- **ActivatedRoute**
  Angular's `ActivatedRoute` service is used to extract the `location` parameter from the route, which determines the city or area to be displayed on the map.

3.1.4 marker

Overview

The `MarkerComponent` is responsible for managing markers on the map. It receives the location data (latitude, longitude) and displays interactive markers on the Leaflet map. The component interacts with other components, such as `GeoJsonComponent` and `MarkerInfoComponent`, to dynamically update the map and provide detailed information about the markers.

Inputs

- **`map: Map | undefined`**
  The Leaflet map instance is passed as an input property. It is used to add markers to the map and manage their interactions.

Properties

- **`layers: Marker[]`**
  An array that stores the markers added to the map. Each marker is represented by a `Marker` object from the Leaflet library.
- **`markerName: string`**
  A string representing the name of the marker, used for displaying the marker's title in the popup.
- **`date: string`**
  The date associated with the event or location represented by the marker. This is displayed in the `MarkerInfoComponent`.
- **`textWrap: string`**
  A string that holds the description or additional information about the marker. This text is displayed in the `MarkerInfoComponent`.
- **`wikiLink: string`**
  A URL to the Wikipedia page for the event or location related to the marker. This link is displayed in the `MarkerInfoComponent`.
- **`lat: string`**
  The latitude coordinate of the marker. This is passed to the `MarkerInfoComponent` for display.
- **`long: string`**
  The longitude coordinate of the marker. This is also passed to the `MarkerInfoComponent` for display.
- **`linkYoutube: string`**
  A URL to a related YouTube video for the marker, if available.
- **`bc_ad: string`**
  A string used for displaying additional advertising or contextual information for the marker.
- **`dataLoaded: boolean`**
  A boolean flag that indicates whether the data for the marker has been loaded and is ready to be displayed.

Methods

- **`ngOnInit(): void`**
  This method is triggered when the component is initialized. It

subscribes to the `processedData$` observable from `FetchDataService`, which provides the data needed to populate the markers. It iterates over the fetched data and calls the `addMarker()` method to add markers to the map and the `setMarkerInfo()` method to populate the `MarkerInfoComponent`. Additionally, it triggers the loading of GeoJSON data via the `GeoJsonComponent` based on the event's year.

- **`addMarker(data: any): void`**
  This method is used to create and add markers to the map. It takes the `data` object, which contains the marker's latitude, longitude, and other relevant information. The marker is created with a custom icon and popup, and is then added to the map.
- **`setMarkerInfo(data: any): void`**
  This method updates the `MarkerInfoComponent` with the details related to the current marker. It sets the various fields (such as date, marker name, description, etc.) in the `MarkerInfoComponent` to provide the user with additional information when interacting with a marker.

HTML Structure

The HTML structure for the `MarkerComponent` includes the following key elements:

```
<div class="marker-container">
  <app-geo-json></app-geo-json>
  <app-marker-info
    [date]="date"
    [marker_name]="markerName"
    [text_wrap]="textWrap"
    [wiki_link]="wikiLink"
    [lat]="lat"
    [long]="long"
    [link_youtube]="linkYoutube"
    [bc_ad]="bc_ad"
    [dataLoaded]="dataLoaded">
  </app-marker-info>
</div>
```

- **`<div class="marker-container">`**
  The container div that wraps the entire marker-related functionality. It ensures that the marker components are organized within a dedicated container.
- **`<app-geo-json></app-geo-json>`**
  This component is included within the `MarkerComponent` to dynamically load the GeoJSON data for the year associated with the marker. It ensures that the map reflects the historical events and data when the marker is clicked or viewed.
- **`<app-marker-info>`**
  The `MarkerInfoComponent` is responsible for displaying detailed information about the marker. It receives several input properties:
  - **`[date]`**: The date associated with the marker.
  - **`[marker_name]`**: The name of the marker.
  - **`[text_wrap]`**: A description or additional information related to the marker.
  - **`[wiki_link]`**: A link to the relevant Wikipedia page for further information.
  - **`[lat]`**: The latitude coordinate of the marker.
  - **`[long]`**: The longitude coordinate of the marker.
  - **`[link_youtube]`**: A YouTube link related to the marker.
  - **`[bc_ad]`**: An additional piece of contextual information, such as advertising or historical context.
  - **`[dataLoaded]`**: A flag to indicate whether the data has been successfully loaded and is ready to display.

Dependencies

- **Leaflet**
  The component relies on the Leaflet library to create and manage markers on the map. The `Marker` and `icon` classes are used to define the markers, and the map is manipulated to add or remove markers.
- **FetchDataService**
  The `FetchDataService` is responsible for fetching and

processing the data needed to create markers on the map. The processed data is emitted to the `MarkerComponent` through an observable.

- **StateService**
  The `StateService` is used to interact with the global map state. It ensures that markers are added to the correct map instance and manages map interactions.
- **GeoJsonComponent**
  The `GeoJsonComponent` is used to load and display GeoJSON data related to the historical events for each marker. This is dynamically triggered when the marker is added to the map.
- **MarkerInfoComponent**
  The `MarkerInfoComponent` is used to display detailed information about the marker when the user interacts with it. The `MarkerComponent` passes relevant data (like name, description, and links) to it.

3.1.5 marker-info

Overview

The `MarkerInfoComponent` is responsible for displaying detailed information about a marker on the map. When the user clicks on a marker, this component presents contextual data such as the marker's name, date, description, related Wikipedia link, and other relevant details. This information is dynamically updated whenever a new marker is selected, providing the user with an interactive experience.

Inputs

- **`date: string`**
  The date associated with the event or location represented by the marker. This date is formatted before being displayed in the component.
- **`marker_name: string`**
  The name of the marker, which is usually the title or the main subject of the event or location represented by the marker. It is displayed prominently in the component.

- **`text_wrap: string`**
  A description or additional text related to the marker. This could be a brief summary or a detailed explanation of the event or place that the marker represents.
- **`wiki_link: string`**
  A URL to the relevant Wikipedia page for further information about the event or location associated with the marker.
- **`lat: string`**
  The latitude coordinate of the marker's location. This information is displayed for reference.
- **`long: string`**
  The longitude coordinate of the marker's location. Similar to latitude, this is displayed for reference.
- **`link_youtube: string`**
  A URL to a related YouTube video for the marker. This could be a video providing more details about the event, person, or place represented by the marker.
- **`bc_ad: string`**
  Additional contextual information, which could be advertising, historical context, or any other relevant data related to the marker.
- **`dataLoaded: boolean`**
  A flag indicating whether the data has been successfully loaded and is ready to be displayed in the component. If this is `false`, the component may display a loading message or remain empty until the data is fully available.

Methods

- **`ngOnInit()`**
  This method is called when the component is initialized. It formats the date input using the `dateFormater` method. The formatted date will be displayed in the component.
- **`dateFormater(date: string): string`**
  This method formats the date into a more readable form, converting numeric month values (e.g., '1' for January, '2' for February) into abbreviated month names (e.g., 'Jan', 'Feb'). It splits

the input date string by `/` and then reassembles it in the format "Month Day Year" (e.g., "Jan 01 2020").

- **`onPlayVideo()`**
  This method is currently not implemented, but it could be used to handle actions related to playing a video (e.g., launching a video modal or opening the YouTube link in a new window).

HTML Structure

The `MarkerInfoComponent` is structured to display the data provided to it through the input properties. Here's an example of how the template could look:

```
<div class="marker-info-container" *ngIf="dataLoaded">

  <h3>{{ marker_name }}</h3>

  <p>{{ text_wrap }}</p>

  <p><strong>Date:</strong> {{ date }}</p>

  <p><strong>Location:</strong> {{ lat }}, {{ long }}</p>

  <a *ngIf="wiki_link" href="{{ wiki_link }}" target="_blank">Learn more on Wikipedia</a>

  <a *ngIf="link_youtube" href="{{ link_youtube }}" target="_blank">Watch related video</a>

  <p *ngIf="bc_ad">{{ bc_ad }}</p>

</div>
```

Key Elements:

- **`<h3>{{ marker_name }}</h3>`**
  Displays the name of the marker as a heading.
- **`<p>{{ text_wrap }}</p>`**
  Displays the description or additional information related to the marker.

- `<p><strong>Date:</strong> {{ date }}</p>`
  Displays the formatted date for the marker.
- `<p><strong>Location:</strong> {{ lat }}, {{ long }}</p>`
  Displays the latitude and longitude of the marker.
- `<a href="{{ wiki_link }}" target="_blank">Learn more on Wikipedia</a>`
  A link to the Wikipedia page for the marker, if available.
- `<a href="{{ link_youtube }}" target="_blank">Watch related video</a>`
  A link to the YouTube video related to the marker, if available.
- `<p *ngIf="bc_ad">{{ bc_ad }}</p>`
  Displays the `bc_ad` if it is provided, for additional context.

3.1.6 search-form

Overview
The `SearchFormComponent` is a simple form that allows users to search for a specific location on the map. The user can type a location (such as a city name) into the input field, and upon submitting the form, the component navigates to a new route that displays data related to that location.

Inputs

- `value: string`
  This property holds the value entered by the user in the input field. It is bound to the input field using two-way data binding (`ngModel`). The value represents the location the user is searching for.

Methods

- `handleSubmit(event: Event): void`
  This method is invoked when the user submits the form. It prevents the default form submission behavior (which would cause a page reload), and then navigates to a new route based on the value entered by the user in the input field. The new route includes

the location value as part of the URL.
The navigation logic is implemented as:

window.location.href = `/${this.value}`;

This method directly updates the browser's URL by setting `window.location.href` to the desired route. For example, if the user enters "London" in the search field, the URL will change to `/London`, and the application will navigate to the relevant page for that location.

HTML Structure

The `SearchFormComponent` consists of a simple form that includes an input field and a submit button. Here's an example of the HTML structure:

```
<form class="search" (ngSubmit)="handleSubmit($event)">

<div class="search-box">

<input class="search-txt" type="text" [(ngModel)]="value"
name="location" placeholder="Type to Search"/>

<button class="search-btn" type="submit">

<i class="fa fa-search"></i>

</button>

</div>

</form>
```

Key Elements:

- **`<input [(ngModel)]="value" />`**
  A two-way binding is used here, meaning the value of the input field will automatically update the `value` property in the component and vice versa.

- **`<button type="submit">Search</button>`**
  This button submits the form and triggers the `handleSubmit()` method.

4.1 Core structures

4.1.1. SafeUrlPipe

Overview
The `SafeUrlPipe` is an Angular pipe that is used to sanitize URLs, making them safe for embedding within the application, particularly in contexts like iframes, videos, or any other resource that requires a trusted URL. This pipe leverages Angular's `DomSanitizer` service to bypass the Angular security checks and allow the application to trust URLs that may otherwise be considered unsafe.

Functionality

The `SafeUrlPipe` takes a string URL as input and transforms it into a `SafeResourceUrl`. This transformation allows the application to embed external resources (such as YouTube videos, maps, etc.) without triggering Angular's built-in security restrictions.

Inputs

- **`url: string`**
  The URL that needs to be sanitized and trusted. This can be any URL that is meant to be embedded as a resource in the application.

PipeTransform Method

- **`transform(url: string): SafeResourceUrl`**
  This method accepts a `string` URL and returns a sanitized and trusted URL (`SafeResourceUrl`) that can be safely used within Angular components without causing security issues.

How It Works

1. The pipe uses Angular's `DomSanitizer` service to sanitize the URL.
2. It calls the `bypassSecurityTrustResourceUrl` method to mark the URL as safe for embedding.
3. The sanitized URL is returned as a `SafeResourceUrl`, which can then be used within the template to embed content such as videos, maps, or other external resources.

Usage

This pipe is typically used when embedding external resources, such as in the case of a YouTube video or an iframe, where the URL must be sanitized to avoid cross-site scripting (XSS) attacks.

Example Usage:

In your component's HTML template, you can use the `safeUrl` pipe to safely bind a URL to an iframe or video tag:

```
<iframe [src]="videoUrl | safeUrl"></iframe>
```

Where `videoUrl` is a string containing the URL to the video or other embedded resource. The `safeUrl` pipe ensures that the URL is sanitized before being used.

Example with YouTube:

```
<iframe width="560" height="315" [src]="videoLink | safeUrl"
frameborder="0" allow="accelerometer; autoplay; clipboard-write;
encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
```

Here, `videoLink` would be a string URL, and the pipe ensures that it's safely embedded.

4.1.2 services folder structure

In the **`core/services`** folder, there are several key services that handle the application's data management and state. These services provide essential functionality for interacting with external data sources, processing and storing information, and managing the application state.

CityData Service

The `CityDataService` is responsible for managing data related to cities. This service may interact with external APIs to search for cities, retrieve geographical information (such as latitude and longitude), and return results based on user input. It acts as a mediator between the application and external city-related data.

FetchData Service

The `FetchDataService` is responsible for handling data fetching and processing. It may interact with external APIs, process the fetched data (e.g., transform it into a usable format), and emit this data to other components or services. This service plays a critical role in making the application dynamic by fetching and preparing data for display on the map or in other parts of the application.

FirebaseData Service

The `FirebaseDataService` handles interactions with Firebase, specifically for retrieving and storing data in a Firebase database. This service allows the application to read and write data from Firebase, ensuring that the application can persist data and synchronize with a cloud database. It may handle operations like querying data, subscribing to changes, and storing new entries.

State Service

The `StateService` manages the global state of the application. It is used to store and retrieve shared information such as the current map instance, the user's settings, or other global parameters. This service ensures that different parts of the application can access and modify the state without directly interacting with each other. For example, it can

manage whether the map is ready, store the user's selected city, or manage the map's zoom level.

4.1.2.1 CityDataService

The `CityService` (referred to as `CityDataService` in the context) is an Angular service that interacts with an external API to search for city information. It leverages the `HttpClient` module to make HTTP requests and returns the data in an `Observable` format, allowing other components to subscribe and react to changes asynchronously.

Functionality

This service is primarily used to search for cities based on a given city name. It sends a request to the Nominatim API (which is part of OpenStreetMap) and returns the results in JSON format. The service is designed to handle city searches in real-time and provides the necessary data to other components in the application.

Methods

- **`searchCity(cityName: string): Observable<any>`**
  This method is used to search for a city based on its name. It constructs the API request URL using the base URL of the Nominatim API and appends the provided city name. The method returns an `Observable` that will emit the search results once the HTTP request is completed.
  Parameters:
  - `cityName: string` — The name of the city to search for.
- **Returns:**
  - An `Observable<any>`, where `any` represents the response data from the Nominatim API (usually a JSON object containing city information such as latitude, longitude, and other metadata).
- **Usage Example:**

```
// In a component or other service
this.cityService.searchCity('New York').subscribe(response => {
  console.log('City search results:', response);
```

});

This will call the `searchCity` method with the city name `New York` and log the response once the data is received.

HTTP Request

The API request is made using the `HttpClient`'s `get` method. The URL is dynamically constructed by appending the city name to the base API URL:

const url = `${this.apiUrl}${encodeURIComponent(cityName)}`;

- The `encodeURIComponent` function ensures that the city name is properly encoded for use in a URL, preventing issues with special characters in the city name (such as spaces or non-alphanumeric characters).

Response Handling

The `Observable` returned by `http.get` can be subscribed to by components or other services. When the HTTP request is complete, the response data is emitted to the subscribers, and they can then process or display the data as needed.

Example of Response Structure:

The response from the API will generally contain an array of objects with the following structure:

[

{

"place_id": "123456",

"lat": "40.7128",

"lon": "-74.0060",

```
    "display_name": "New York, USA",
    "type": "city",
    "importance": 0.8
  }
]
```


The data includes:

- `lat`: The latitude of the city.
- `lon`: The longitude of the city.
- `display_name`: A human-readable name for the city.
- `place_id`: A unique identifier for the city.
- `type`: The type of the place (e.g., city, town, etc.).
- `importance`: A measure of the place's importance.

4.1.2.2 FetchDataService

The `FetchDataService` is an Angular service responsible for processing raw geographical data, calculating distances, and emitting processed data. This service is integral to transforming data, particularly for filtering markers based on proximity to a specified location.

It uses `BehaviorSubject` from RxJS to handle asynchronous data streams, which makes it easy to push updates to components that subscribe to the processed data.

Functionality

- **Distance Calculation:** The service includes methods for calculating the distance between two geographical points (latitude, longitude), allowing it to filter data based on proximity.
- **Data Transformation & Emission:** It processes raw data by filtering markers that fall within a given radius and transforms the

data by adding additional properties (like distance) before emitting it to other parts of the application.

Methods

- **`getDistance(origin: number[], destination: number[]): number`**
  This method calculates the distance between two geographical points, specified as arrays of latitude and longitude. It uses the Haversine formula to calculate the great-circle distance between the origin and destination coordinates, returning the result in meters.
  Parameters:
  - `origin: number[]` — An array containing the latitude and longitude of the origin point.
  - `destination: number[]` — An array containing the latitude and longitude of the destination point.
- **Returns:**
  - A number representing the distance between the two points in meters.
- **Usage Example:**

```
const distance = fetchDataService.getDistance([40.7128, -74.0060], [34.0522, -118.2437]);

console.log(`Distance: ${distance} meters`);
```

`toRadian(degree: number): number`
Converts degrees to radians, which is necessary for the distance calculation.
Parameters:

- `degree: number` — The angle in degrees to be converted to radians.

Returns:

- A number representing the angle in radians.

`processAndEmitData(rawData: any[], origin: number[], radius: number)`

This method processes raw data by filtering the markers that are within a specified radius from the origin point. It also adds additional properties like the calculated distance to each item and marks them as "transformed." It then emits the filtered and transformed data via the `BehaviorSubject` to notify subscribers.

Parameters:

- `rawData: any[]` — An array of raw geographical data (e.g., marker information with coordinates).
- `origin: number[]` — The latitude and longitude of the origin point.
- `radius: number` — The radius in meters, used to filter data based on proximity to the origin point.

Returns:

- This method does not return anything but emits the processed data to subscribers through the `processedDataSubject`.

Usage Example:

typescript
Копирај кôд

```
fetchDataService.processAndEmitData(rawData, [40.7128, -74.0060], 5000);
```

How It Works

1. **Data Filtering:**
   The `processAndEmitData` method iterates over the raw data and calculates the distance between the origin and each marker using the `getDistance` method. If the distance is less than or

equal to the specified radius and the number of records is fewer than 10, the marker is added to the `records` array.

2. **Distance Calculation:**
   The `getDistance` method uses the Haversine formula to compute the distance between two points on the Earth's surface. The result is in meters, which allows for accurate proximity filtering.
3. **Emitting Data:**
   Once the data is processed and filtered, it is emitted via the `processedDataSubject` using `next()`. Any components or services that are subscribed to `processedData$` will receive this updated, filtered data.
4. **Observable Pattern:**
   The service uses RxJS's `BehaviorSubject` to emit processed data. This allows other components or services to reactively consume the filtered and transformed data as soon as it's available.

Example of Filtered Data Structure:

After processing, each item in the `records` array will look something like this:

```
{
  "lat": 40.7128,
  "long_marker": -74.0060,
  "name": "New York",
  "description": "The Big Apple",
  "distance": 2000,
  "transformed": true
}
```

`lat`: Latitude of the marker.

`long_marker`: Longitude of the marker.

`name`: Name of the location.

`description`: Description of the marker.

`distance`: The distance from the origin to this marker in meters.

`transformed`: A flag indicating that the data has been processed.

4.1.2.3 FirebaseDataService

The `FirebaseDataService` is an Angular service that interacts with Firebase Realtime Database. It provides methods to read, write, update, and delete data in the Firebase database. It uses the `AngularFireDatabase` service from the Firebase SDK to facilitate database operations.

Methods

1. **`getData(path: string): Observable<any>`**
   This method retrieves data from a specified path in the Firebase Realtime Database. It returns an `Observable` that emits the data whenever it changes.
   Parameters:
   - `path: string` — The path in the Firebase database from which to retrieve data.
2. **Returns:**
   - An `Observable<any>` that emits the data at the specified path.
3. **Usage Example:**

```
this.firebaseDataService.getData('/2/data').subscribe(data => {
  console.log(data);
});
```

`setData(path: string, data: any): Promise<void>`

This method writes data to a specified path in the Firebase database. It overwrites any existing data at that path.

Parameters:

- `path: string` — The path in the Firebase database where the data should be stored.
- `data: any` — The data to store at the specified path.

Returns:

- A `Promise<void>` that resolves when the operation is complete.

Usage Example:

```
const data = { name: 'John Doe', age: 30 };
this.firebaseDataService.setData('/users/johndoe', data).then(() => {
  console.log('Data saved successfully');
});
```

`updateData(path: string, data: any): Promise<void>`

This method updates data at a specific path in the Firebase database. It only updates the fields provided in the `data` object and leaves the other fields unchanged.

Parameters:

- `path: string` — The path in the Firebase database where the data should be updated.
- `data: any` — The data to update at the specified path.

Returns:

- A `Promise<void>` that resolves when the operation is complete.

Usage Example:

```
const updatedData = { age: 31 };

this.firebaseDataService.updateData('/users/johndoe',
updatedData).then(() => {

  console.log('Data updated successfully');

});
```

`deleteData(path: string): Promise<void>`

This method deletes data from a specific path in the Firebase database.

Parameters:

- `path: string` — The path in the Firebase database from which to delete data.

Returns:

- A `Promise<void>` that resolves when the operation is complete.

Usage Example:

```
this.firebaseDataService.deleteData('/users/johndoe').then(() => {

  console.log('Data deleted successfully');

});
```

How It Works

- The `getData` method listens for changes at a specific path in the Firebase Realtime Database and returns an observable that updates the data whenever it changes. This is useful for applications that need to reactively update the UI whenever the database content is modified.

- The `setData` method writes data to the specified path. If data already exists at that path, it will be overwritten. This method is useful when adding new records or replacing existing data.
- The `updateData` method allows partial updates to be made to an existing record in the database. Only the fields present in the `data` object are updated, leaving the rest of the data intact.
- The `deleteData` method removes data from the specified path. This can be useful for deleting records from the database.

Integration Example

You can use this service in your Angular components to interact with Firebase. For example, you can get data when the component initializes:

```
import { Component, OnInit } from '@angular/core';

import { FirebaseDataService } from
'./core/services/fireBaseData/firebase-data.service';


@Component({

  selector: 'app-my-component',

  templateUrl: './my-component.component.html',

  styleUrls: ['./my-component.component.scss']

})

export class MyComponent implements OnInit {


  data: any;


  constructor(private firebaseDataService: FirebaseDataService) {}
```

```
  ngOnInit() {
    this.firebaseDataService.getData('/path/to/data').subscribe(data => {
      this.data = data;
      console.log('Data received:', data);
    });
  }

}
```

In the above example, the component listens for changes at the `/path/to/data` location in the Firebase database. Whenever data at that location changes, the component will receive the updated data.

4.1.2.1 StateDataService

Overview
The `StateService` is an Angular service used to manage the state related to the map's readiness and its reference. This service allows components to store and retrieve the map object and its readiness status. It ensures that the map is properly initialized before trying to interact with it.

Properties

- **`_mapReady: boolean`**
  A private property that tracks whether the map is ready to be interacted with. It is initially set to `false` and can be updated using the `setMapReady()` method.

- **`_map: any[]`**
  A private property that stores the map object. It is initially set to an empty array and can be updated using the `setMap()` method.

Methods

1. **`setMapReady(mapReady: boolean): void`**
   Sets the readiness status of the map. This method is used to update the `_mapReady` property based on whether the map is ready for interaction.
   Parameters:
   - `mapReady: boolean` — `true` if the map is ready, `false` if it is not.
2. **Usage Example:**
   this.stateService.setMapReady(true); // Sets the map as ready

`setMap(map: any): void`
Sets the map object in the service. This method is used to store a reference to the map once it is initialized.

Parameters:

- `map: any` — The map object to store.

Usage Example:

this.stateService.setMap(this.map); // Store the map object

`getMapReady(): boolean`
Returns the readiness status of the map. It is useful to check whether the map is ready before attempting to interact with it.

Returns:

- A `boolean` that indicates whether the map is ready (`true` or `false`).

Usage Example:

```
const isMapReady = this.stateService.getMapReady();

if (isMapReady) {

  // Safe to interact with the map

}
```

`getMap(): any`

Retrieves the map object stored in the service. This method is used to access the map reference so that components or services can interact with it.

Returns:

- The map object stored in the service.

Usage Example:

```
const map = this.stateService.getMap();

map.addLayer(newMarker);  // Interact with the map
```

How It Works

- The `StateService` manages the state of the map, specifically its readiness and reference. It is used to ensure that the map is initialized properly before any operations (such as adding markers or changing views) are performed on it.
- The `setMapReady()` method updates the `_mapReady` property to indicate whether the map is ready to be used. This allows components to check the map's readiness status and avoid errors related to interacting with an uninitialized map.
- The `setMap()` method stores the map reference, which can then be accessed by other components or services. This is helpful for centralized management of the map object.

5. Links

In this section, I will provide all relevant links that point to where everything is located on the server, including routes and resources used within the application.

Git links -
https://github.com/zirafica98/Markers-Nearby-Angular/tree/main/markers-nearby-angular
Image storage -
https://mappinghistorybucket.s3.us-east-2.amazonaws.com

Our client application is hosted on a Bluehost server, and the link to the login panel is as follows:https://www.bluehost.com/my-account/login.
UserId je : mappingh

Firebase-
https://console.firebase.google.com/project/mapping-history-7242b/database/mapping-history-7242b-default-rtdb/data

LINK APPLICATION:
https://nearby-marker.mappinghistory.com/London

After this URL (`https://nearby-marker.mappinghistory.com/London`), you can enter any location you wish to search for. Instead of "London", simply type the name of the city, town, or region you're looking for, and the system will automatically display the relevant results.

For example:

- `https://nearby-marker.mappinghistory.com/Paris` - This URL will show results for Paris.

This search method allows users to quickly and easily access relevant information about any location they are interested in, by simply entering the location name in the URL.