1/11/2025

Location Based Services Application

Module: Advanced Web Mapping

CMPU4058

Lecturer: Dr Bujar Raufi

Student Name: Aaron Baggot
TU856/4

C22716399

Table of Contents

1. Introduction

This assignment involved developing a spatial web application using Django, GeoDjango, Leaflet, and PostgreSQL with PostGIS. The aim was to build an interactive Trails API that lets users view and query hiking trails in Ireland through both a REST API and a web map.
The project builds on the "Cities API" lab work but focuses on trail/hiking data and spatial analysis. Within his report includes setup, implementation, errors encountered, debugging steps, and overall progress.

2. Project Setup and Configuration

2.1 Development Environment

- OS: macOS
- Python: 3 (using virtual environment)
- Database: PostgreSQL with PostGIS
- Main Libraries:
- Django 4.2+
- Django REST Framework
- Django REST Framework GIS
- drf-spectacular
- Django Filter
- Leaflet.js
- Bootstrap

2.2 Database Configuration

A spatial database called webmapping_db was created and connected in Django's settings.
PostGIS libraries were configured for macOS to ensure full GIS functionality.

3. Trails API Design

3.1 Models

The Trail model stores trail details and spatial coordinates (start_point) for mapping and analysis.
Each trail includes fields like name, county, region, distance, difficulty, elevation gain, and description.

3.2 API Endpoints

The application includes:

- api/trails/ – List and create trails
- api/trails/<id>/ – Retrieve, update, delete
- api/trails/geojson/ – GeoJSON output for Leaflet
- api/trails/within-radius/ – Radius-based search
- api/trails/bbox/ – Bounding box search
- api/trails/stats/ – Trail statistics
- api/trails/test/ – API test page

3.3 Map Integration

The map interface (Leaflet + OpenStreetMap) supports:

- Radius search by clicking the map
- Dynamic trail display from GeoJSON
- Popup details for each trail

4. Implementation and Debugging

4.1 Migration Errors

- Fixed typos like Charfield CharField
- Corrected field names (trail_name, distance_km)
- Solved import and namespace issues between cities_api and trails_api

4.2 URL and View Issues

- Removed unused imports and duplicate view definitions
- Added the missing api_test_page view

4.3 Map and GeoJSON Fixes

Trails weren’t appearing because coordinates were using the wrong field (location instead of start_point)

Updated serializers and JavaScript to fix it

4.4 Database Cleanup

Removed redundant databases and kept webmapping_db as the main one.

5. Current Status

- CRUD and spatial API endpoints are working
- Map loads successfully and shows proximity searches
- Trails added via Django shell with valid coordinates
- GeoJSON and radius queries verified and functional

- Automate data population using a management script
- Filtering and sorting options in the frontend
- Integrate real Irish trail datasets
- Automated unit testing

Debugging and Feature Enhancement Summary

6. Overview

This phase focused on improving the map's interactivity, proximity search, and marker styling. Users can now click the map to search for trails nearby and view details in both popups and a side panel.

7. Issues Identified and Fixes

Problem	**Description**	**Fix**
Unnamed trails	Missing names in popups	Standardized property keys
Identical icons	No visual difference	Added numbered markers
Missing distance	Distance not displayed	Unified backend/JS variable names
Duplicate functions	Caused syntax errors	Combined into one correct version
Map reload issues	Old layers not cleared	Added proper layer management

8. Debugging Steps

- Added console.log() statements to trace API responses
- Normalized inconsistent property names
- Ensured Leaflet layers reset correctly
- Fixed async syntax and bracket errors

9. Features Added

- Proximity Search: Click the map to find nearby trails
- Numbered Markers: Clear visual order of results
- Dynamic Results Panel: Lists nearest trails with clickable zoom links
- Backend Integration: Added distance calculation with GeoDjango
- Improved Popups: Show trail name, county, difficulty, and coordinates

10. Testing

Confirmed that:

- Trails and towns load correctly
- Filters by population, town type, and trail difficulty all work
- Radius search highlights nearby trails accurately
- Map clears and reloads results properly

11. Town Filters Debugging

Town filters initially failed because two views used the same URL (/api/trails/towns/geojson/).
Removing the duplicate TownGeoJSONView fixed it.
Now the function based towns_geojson view handles filters properly.

Lesson Learned:

- Always check for duplicate routes
- Function-based views are better for custom filtering and debugging

12. SQL Query Testing

- SQL scripts tested:
- Trail and town relationships
- Average distance and elevation
- Spatial joins and radius queries around Galway
- Results saved under /spatial_data/outputs/ for verification.

13. Cleanup and Testing

- Removed unused files and duplicate scripts
- Verified all URLs and namespaces
- Confirmed all tests passed using Django's APITestCase

14. Final Functionality

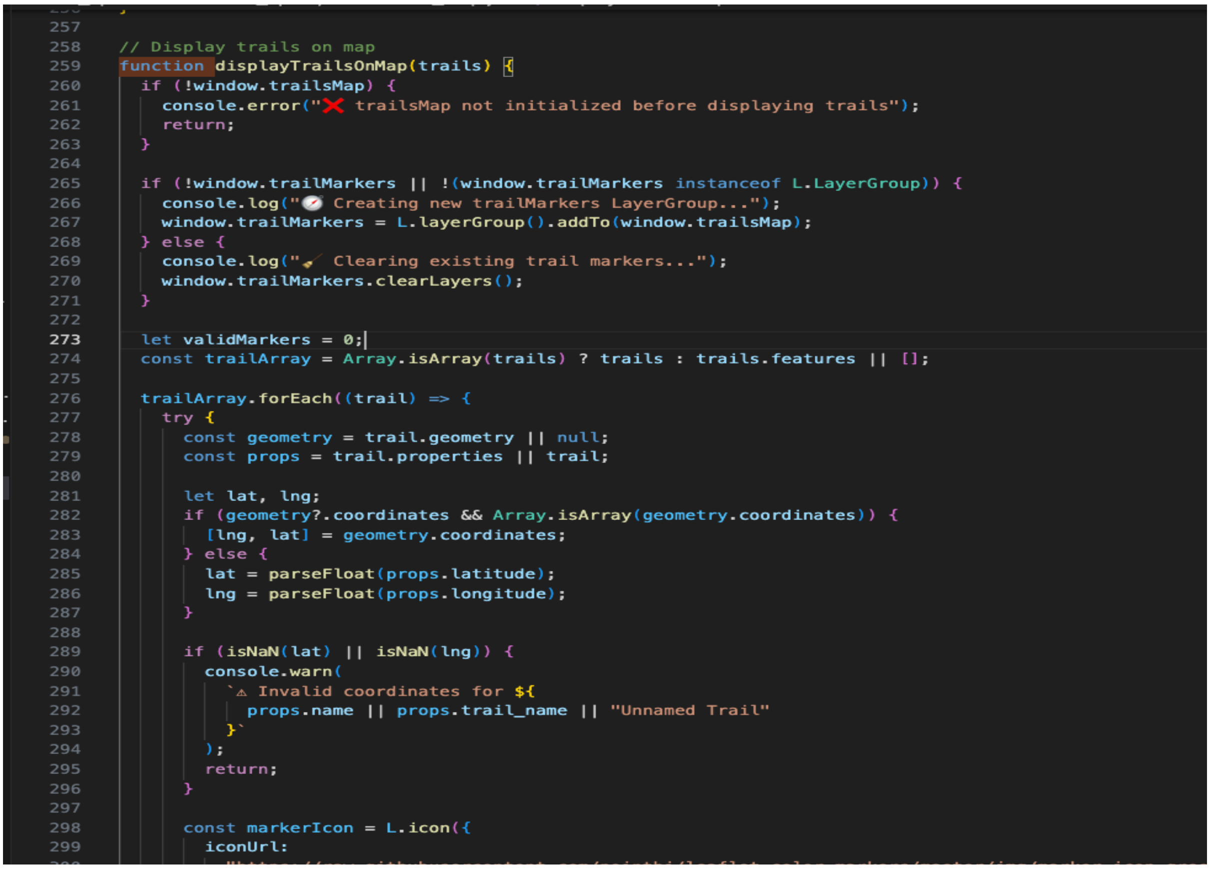
The final system includes:

- Working Leaflet map with search and filtering
- Accurate trail and town GeoJSON endpoints
- Functional proximity and bounding box searches
- Admin panel, dashboard, and API documentation (Swagger + Redoc)

15 Functions Trails API and Dashboard

15.1 Display Trails on Map

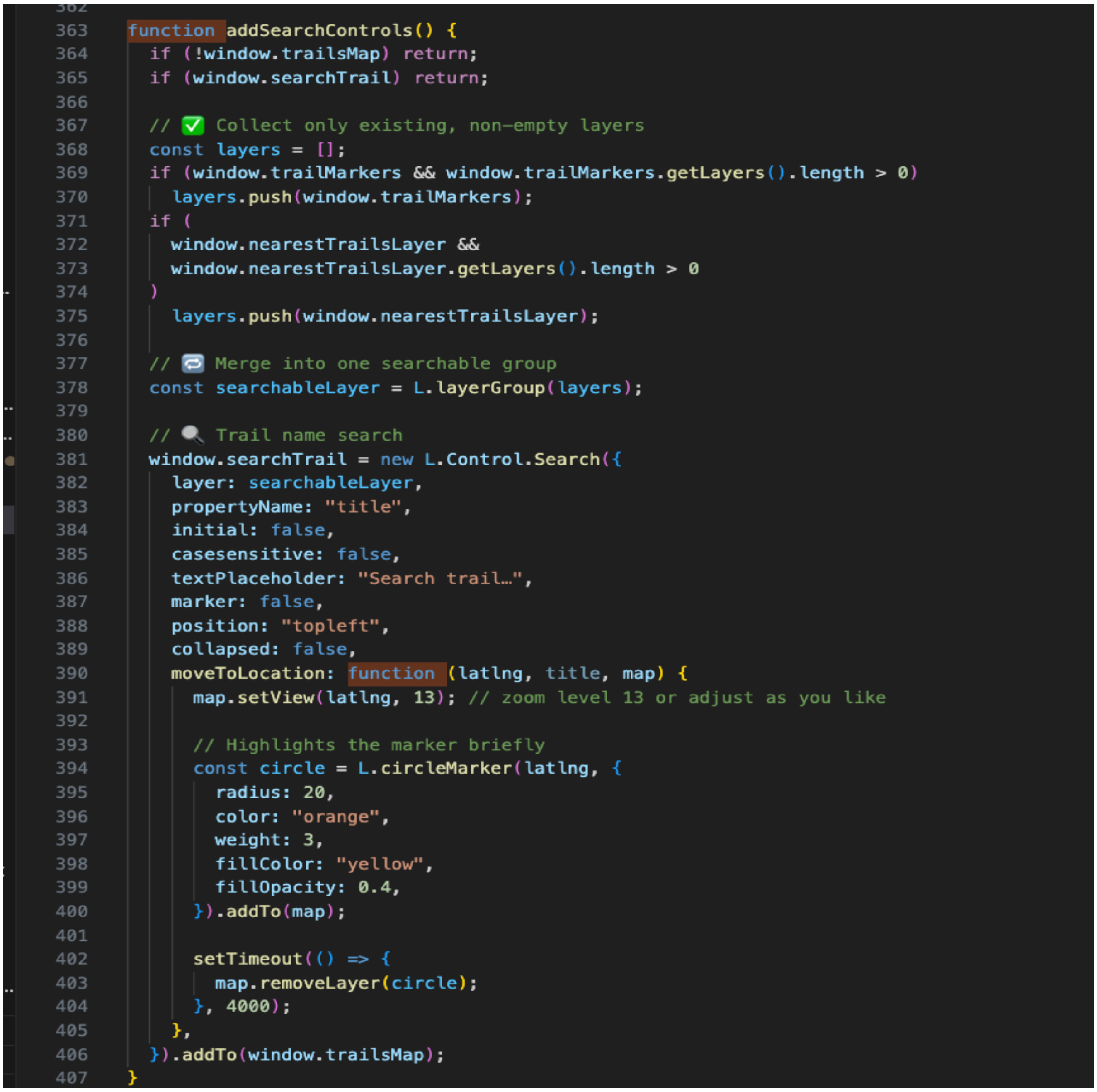
The displayTrailsOnMap function is responsible for displaying all trail data on the interactive Leaflet map. It takes a single parameter, trails, which contains trail information fetched from the Django API in either array or GeoJSON format. The function first checks that the global map object (window.trailsMap) exists before proceeding. It then manages a global layer group called trailMarkers, which holds all markers currently displayed on the map. If this layer group already exists, it is cleared to prevent overlapping or duplicate markers otherwise, a new one is created and added to the map. Each trail in the dataset is looped through, and its coordinates are extracted from either the geometry or property fields. If valid latitude and longitude values are found, a custom green marker icon is created using Leaflet's L.icon() function, and a corresponding marker is added to the map. Each marker includes a popup built dynamically using key trail attributes such as name, county, distance, difficulty, and whether parking or dogs are allowed. These popups allow users to click any marker to view detailed information about that trail. Once all markers are added, the map automatically zooms to show all of them using the fitBounds function. Finally, the function initialises search controls via the Leaflet Search plugin, allowing users to search trails by name once all markers are loaded.



```
// Display trails on map
function displayTrailsOnMap(trails) {
  if (!window.trailsMap) {
    console.error("❌ trailsMap not initialized before displaying trails");
    return;
  }

  if (!window.trailMarkers || !(window.trailMarkers instanceof L.LayerGroup)) {
    console.log("🧭 Creating new trailMarkers LayerGroup...");
    window.trailMarkers = L.layerGroup().addTo(window.trailsMap);
  } else {
    console.log("🧹 Clearing existing trail markers...");
    window.trailMarkers.clearLayers();
  }

  let validMarkers = 0;
  const trailArray = Array.isArray(trails) ? trails : trails.features || [];

  trailArray.forEach((trail) => {
    try {
      const geometry = trail.geometry || null;
      const props = trail.properties || trail;

      let lat, lng;
      if (geometry?.coordinates && Array.isArray(geometry.coordinates)) {
        [lng, lat] = geometry.coordinates;
      } else {
        lat = parseFloat(props.latitude);
        lng = parseFloat(props.longitude);
      }

      if (isNaN(lat) || isNaN(lng)) {
        console.warn(
          `⚠ Invalid coordinates for ${
            props.name || props.trail_name || "Unnamed Trail"
          }`
        );
        return;
      }

      const markerIcon = L.icon({
        iconUrl:
```

15.2 Add Search Controls

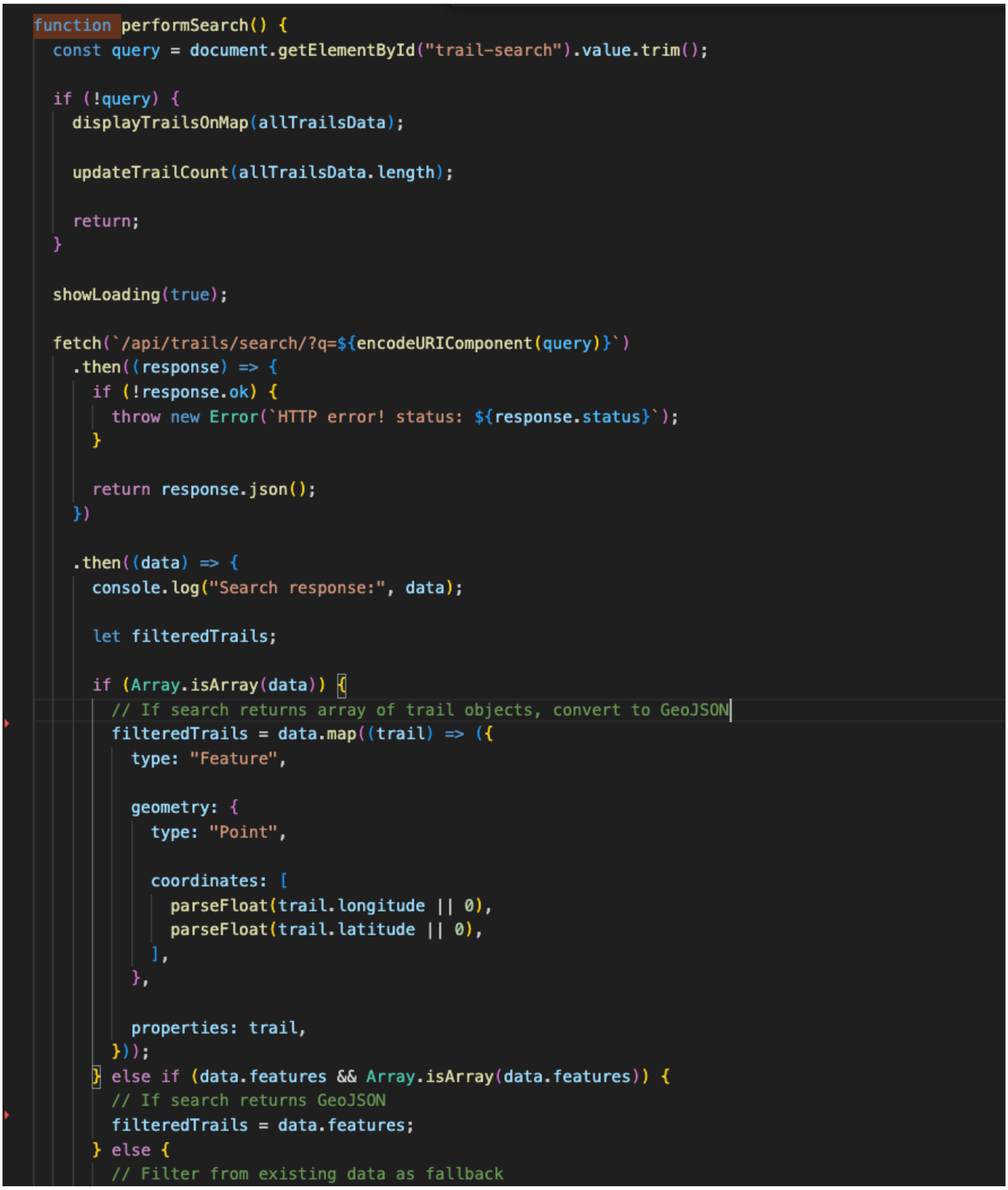
The addSearchControls function enhances the interactivity of the map by adding a custom search feature that allows users to quickly find trails by name. It begins by confirming that the Leaflet map object (window.trailsMap) is available and that a search control hasn’t already been added (window.searchTrail). The function then collects all currently active map layers that contain trail data such as trailMarkers and nearestTrailsLayer (markers for nearby trails) ensuring that only non-empty layers are included. These layers are merged into a single searchable group using L.layerGroup. The main search control is created using the Leaflet Search plugin (L.Control.Search), which indexes all markers by their title property. To provide visual feedback, a temporary orange highlight circle briefly appears around the selected trail marker before going away. The search bar is displayed at the top left of the map, with user-friendly placeholder text and case-insensitive matching.



```
function addSearchControls() {
  if (!window.trailsMap) return;
  if (window.searchTrail) return;

  // ✅ Collect only existing, non-empty layers
  const layers = [];
  if (window.trailMarkers && window.trailMarkers.getLayers().length > 0)
    layers.push(window.trailMarkers);
  if (
    window.nearestTrailsLayer &&
    window.nearestTrailsLayer.getLayers().length > 0
  )
    layers.push(window.nearestTrailsLayer);

  // 🔁 Merge into one searchable group
  const searchableLayer = L.layerGroup(layers);

  // 🔍 Trail name search
  window.searchTrail = new L.Control.Search({
    layer: searchableLayer,
    propertyName: "title",
    initial: false,
    casesensitive: false,
    textPlaceholder: "Search trail…",
    marker: false,
    position: "topleft",
    collapsed: false,
    moveToLocation: function (latlng, title, map) {
      map.setView(latlng, 13); // zoom level 13 or adjust as you like

      // Highlights the marker briefly
      const circle = L.circleMarker(latlng, {
        radius: 20,
        color: "orange",
        weight: 3,
        fillColor: "yellow",
        fillOpacity: 0.4,
      }).addTo(map);

      setTimeout(() => {
        map.removeLayer(circle);
      }, 4000);
    },
  }).addTo(window.trailsMap);
}
```

15.3 Perform Search

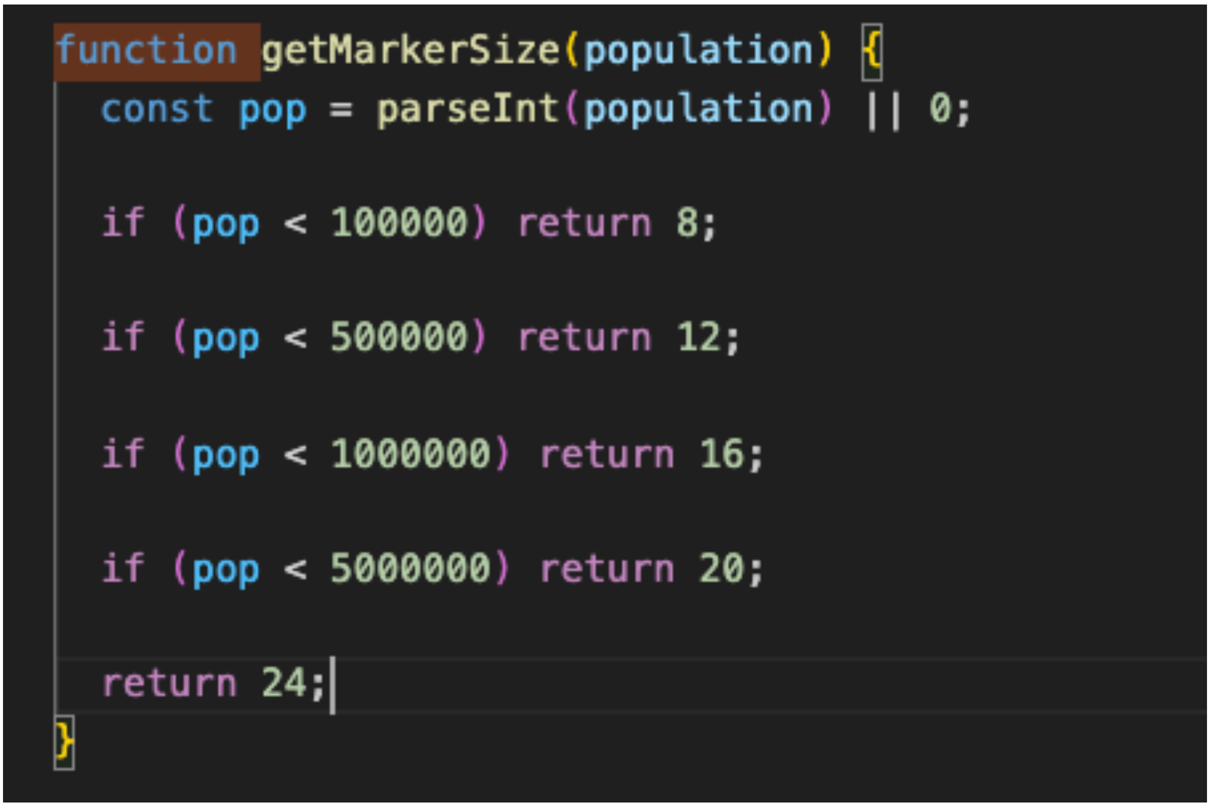
The performSearch function handles user-initiated trail searches and updates the map dynamically based on the results. When a user enters text into the search input field (trail-search) and triggers this function, it retrieves the query string and removes any extra spaces. If the search box is empty, it simply resets the map by calling displayTrailsOnMap(allTrailsData) and updates the trail count to show all available trails. If a query is entered, it activates a loading indicator (showLoading true and sends a request to the Django API endpoint, passing the query as a URL-encoded parameter. Once a response is received, the function checks whether the API returned a standard JSON array or GeoJSON format. If the response is an array of objects, it converts them into proper GeoJSON features with coordinate and property fields. If the response is already in GeoJSON format, it uses the features directly. If the request fails or the API doesn't respond, the function performs a fallback client-side search by filtering through the locally stored allTrailsData, checking if the trail name or country includes the search term. Matching trails are then displayed on the map via displayTrailsOnMap(filteredTrails), and the visible count is updated with updateTrailCount(). If no results are found, a user-friendly alert appears informing them that no trails matched their search.



```
function performSearch() {
  const query = document.getElementById("trail-search").value.trim();

  if (!query) {
    displayTrailsOnMap(allTrailsData);

    updateTrailCount(allTrailsData.length);

    return;
  }

  showLoading(true);

  fetch(`/api/trails/search/?q=${encodeURIComponent(query)}`)
    .then((response) => {
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }

      return response.json();
    })

    .then((data) => {
      console.log("Search response:", data);

      let filteredTrails;

      if (Array.isArray(data)) {
        // If search returns array of trail objects, convert to GeoJSON
        filteredTrails = data.map((trail) => ({
          type: "Feature",

          geometry: {
            type: "Point",

            coordinates: [
              parseFloat(trail.longitude || 0),
              parseFloat(trail.latitude || 0),
            ],
          },

          properties: trail,
        }));
      } else if (data.features && Array.isArray(data.features)) {
        // If search returns GeoJSON
        filteredTrails = data.features;
      } else {
        // Filter from existing data as fallback
```

15.4 Get Marker Size

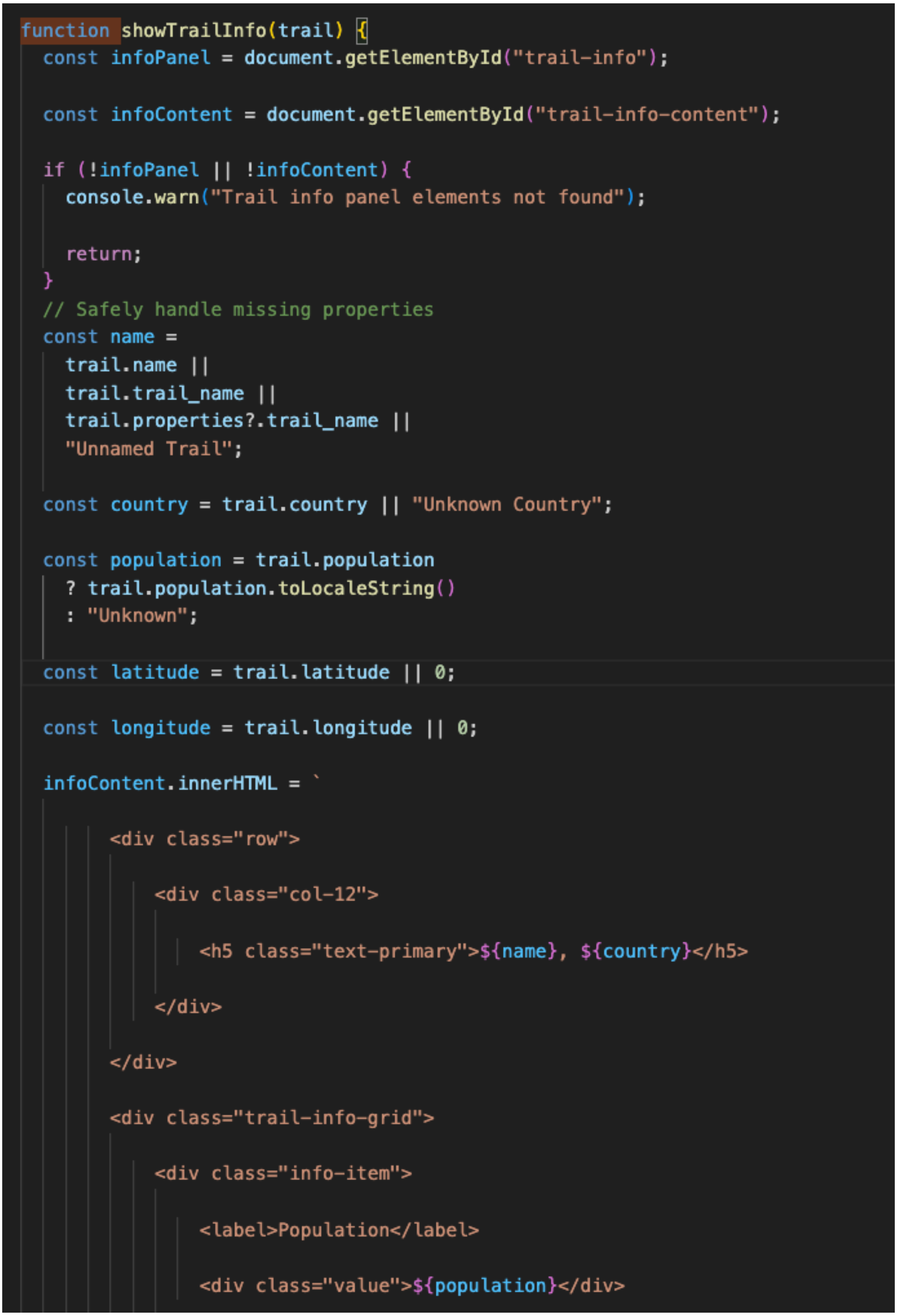
The getMarkerSize function determines the visual size of a map marker based on a town’s population, allowing for a more intuitive and data-driven visualization. It accepts a single parameter, population, which represents the number of residents in a given town. The function first converts this value into an integer using parseInt defaulting to zero if the input is missing or invalid. It then applies a series of conditional checks to categorize the population into ranges, assigning a corresponding marker size in pixels. Smaller towns with fewer than 100,000 residents receive smaller markers, while larger towns and cities scale progressively up to a maximum marker size of 24 for populations exceeding 5 million.



```
function getMarkerSize(population) {
  const pop = parseInt(population) || 0;

  if (pop < 100000) return 8;

  if (pop < 500000) return 12;

  if (pop < 1000000) return 16;

  if (pop < 5000000) return 20;

  return 24;
}
```

15.5 Show Trail Info

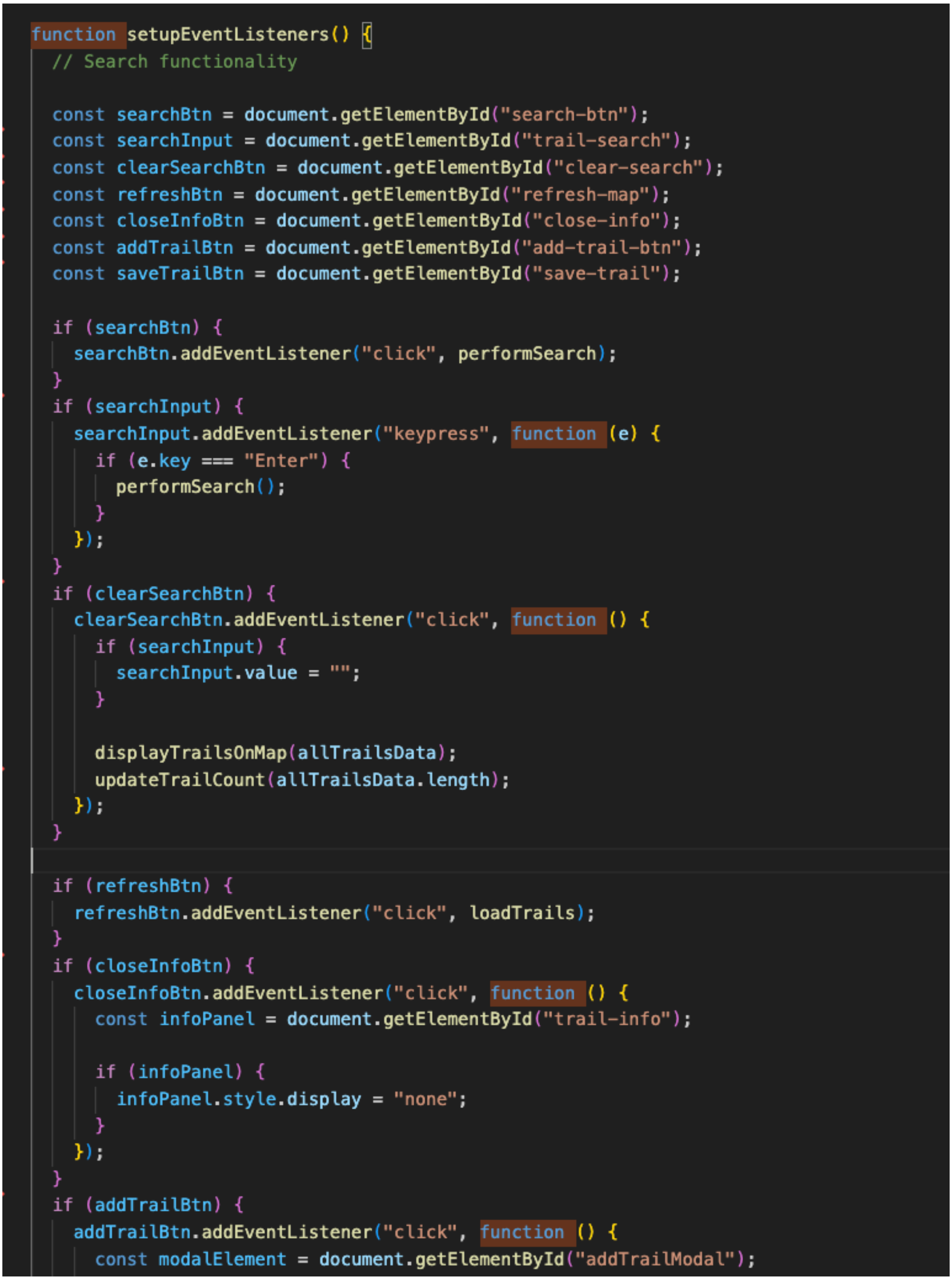
The showTrailInfo function dynamically displays detailed information about a selected trail or town inside an on-screen information panel. When a user clicks on a marker or selects a trail, this function retrieves two key HTML elements: the main info panel (trail-info) and the content area (trail-info-content). If either element is missing, the function logs a warning and stops execution to prevent runtime errors. It then extracts relevant data from the provided trail object including its name, country, population, coordinates, area, and optional attributes like founding year, timezone, or description. Using this data, it constructs an HTML layout that presents the trail details in a structured grid format. This layout includes labels and formatted values, along with interactive buttons to zoom directly to the trail’s location on the map.



```
function showTrailInfo(trail) {
  const infoPanel = document.getElementById("trail-info");

  const infoContent = document.getElementById("trail-info-content");

  if (!infoPanel || !infoContent) {
    console.warn("Trail info panel elements not found");

    return;
  }
  // Safely handle missing properties
  const name =
    trail.name ||
    trail.trail_name ||
    trail.properties?.trail_name ||
    "Unnamed Trail";

  const country = trail.country || "Unknown Country";

  const population = trail.population
    ? trail.population.toLocaleString()
    : "Unknown";

  const latitude = trail.latitude || 0;

  const longitude = trail.longitude || 0;

  infoContent.innerHTML = `

        <div class="row">

          <div class="col-12">

            <h5 class="text-primary">${name}, ${country}</h5>

          </div>

        </div>

        <div class="trail-info-grid">

          <div class="info-item">

            <label>Population</label>

            <div class="value">${population}</div>
```

15.6 Setup event listener

The setupEventListeners function acts as the central controller for managing user interactions within the map interface. It connects various buttons and inputs on the dashboard to their corresponding JavaScript functions, ensuring the interface responds smoothly to user actions. The function begins by identifying key HTML elements such as the search button, search input, clear search, refresh map, close info, add trail, and save trail buttons. It then conditionally attaches event listeners only if those elements exist, preventing runtime errors. The performSearch function is used for triggered, querying and filtering trails. The clear search button resets the search bar, reloading all trails on the map and updating the visible trail count. The close info button hides the trail information panel (trail-info) when the user wants to dismiss details about a selected trail. For adding new data, clicking add trail opens a Bootstrap modal (addTrailModal), allowing users to input details for a new trail, while save trail invokes saveNewTrail, which handles saving the new trail to the database.

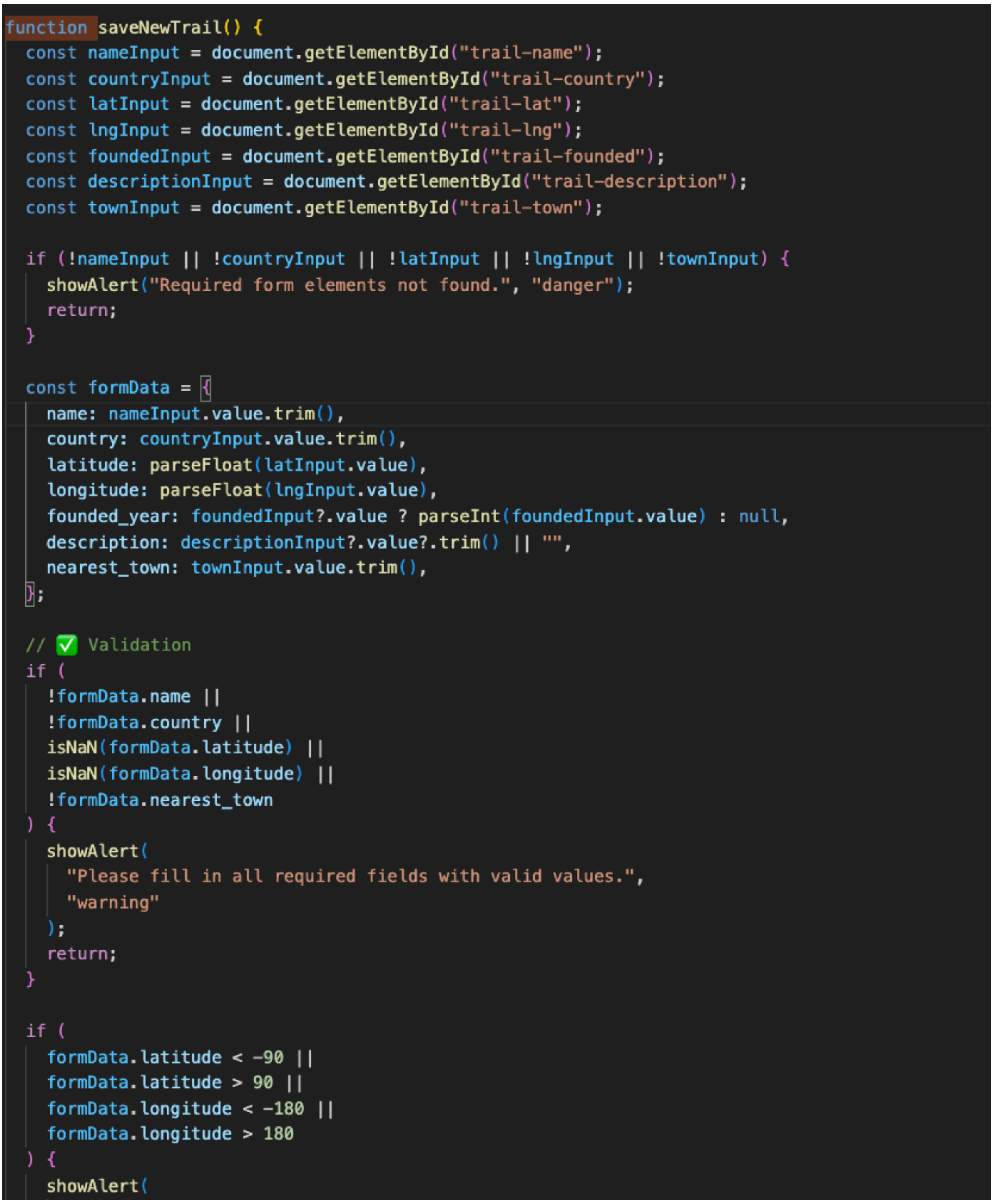


```
function setupEventListeners() {
  // Search functionality

  const searchBtn = document.getElementById("search-btn");
  const searchInput = document.getElementById("trail-search");
  const clearSearchBtn = document.getElementById("clear-search");
  const refreshBtn = document.getElementById("refresh-map");
  const closeInfoBtn = document.getElementById("close-info");
  const addTrailBtn = document.getElementById("add-trail-btn");
  const saveTrailBtn = document.getElementById("save-trail");

  if (searchBtn) {
    searchBtn.addEventListener("click", performSearch);
  }
  if (searchInput) {
    searchInput.addEventListener("keypress", function (e) {
      if (e.key === "Enter") {
        performSearch();
      }
    });
  }
  if (clearSearchBtn) {
    clearSearchBtn.addEventListener("click", function () {
      if (searchInput) {
        searchInput.value = "";
      }

      displayTrailsOnMap(allTrailsData);
      updateTrailCount(allTrailsData.length);
    });
  }

  if (refreshBtn) {
    refreshBtn.addEventListener("click", loadTrails);
  }
  if (closeInfoBtn) {
    closeInfoBtn.addEventListener("click", function () {
      const infoPanel = document.getElementById("trail-info");

      if (infoPanel) {
        infoPanel.style.display = "none";
      }
    });
  }
  if (addTrailBtn) {
    addTrailBtn.addEventListener("click", function () {
      const modalElement = document.getElementById("addTrailModal");
```

15.7 Save New Trail

The saveNewTrail function handles the process of adding a new trail entry to the system by collecting user input, validating it, and sending the data to the Django backend API. It first retrieves references to all relevant form fields such as the trail's name, country, latitude, longitude, founding year, description, and nearest town. If any of these essential form elements are missing, it immediately alerts the user using showAlert and stops execution. Once the inputs are gathered, it builds a formData object containing all the trail attributes, trimming whitespace and converting numeric values where necessary.

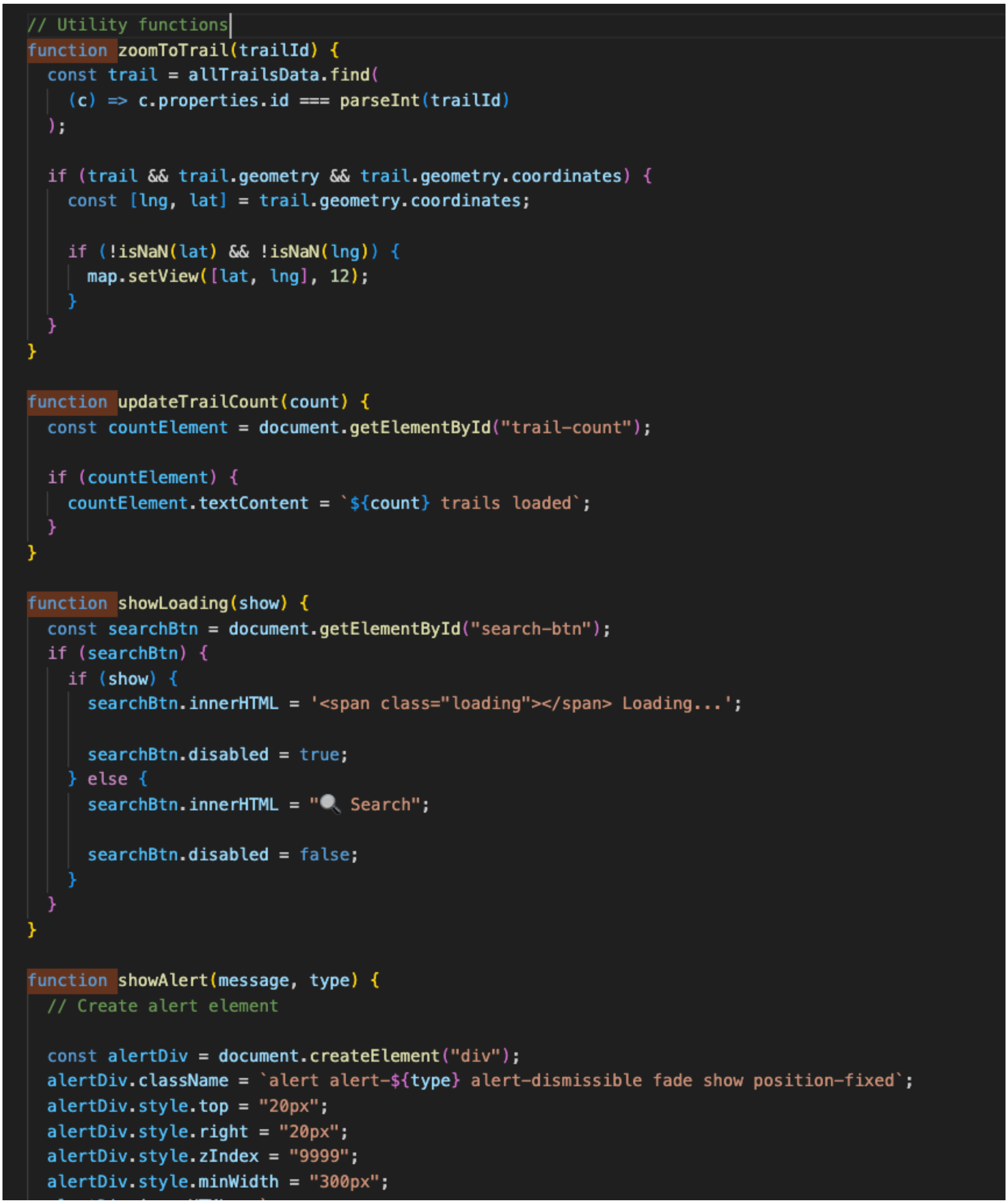
If all checks pass, it makes a POST request to the /api/trails/ endpoint using the Fetch API, sending the data in JSON format and including the CSRF token for security. Overall, saveNewTrail validates user input, securely communicates with the Django REST API, and dynamically updates the map interface to reflect new data without requiring a page reload.



```
function saveNewTrail() {
  const nameInput = document.getElementById("trail-name");
  const countryInput = document.getElementById("trail-country");
  const latInput = document.getElementById("trail-lat");
  const lngInput = document.getElementById("trail-lng");
  const foundedInput = document.getElementById("trail-founded");
  const descriptionInput = document.getElementById("trail-description");
  const townInput = document.getElementById("trail-town");

  if (!nameInput || !countryInput || !latInput || !lngInput || !townInput) {
    showAlert("Required form elements not found.", "danger");
    return;
  }

  const formData = {
    name: nameInput.value.trim(),
    country: countryInput.value.trim(),
    latitude: parseFloat(latInput.value),
    longitude: parseFloat(lngInput.value),
    founded_year: foundedInput?.value ? parseInt(foundedInput.value) : null,
    description: descriptionInput?.value?.trim() || "",
    nearest_town: townInput.value.trim(),
  };

  // ✅ Validation
  if (
    !formData.name ||
    !formData.country ||
    isNaN(formData.latitude) ||
    isNaN(formData.longitude) ||
    !formData.nearest_town
  ) {
    showAlert(
      "Please fill in all required fields with valid values.",
      "warning"
    );
    return;
  }

  if (
    formData.latitude < -90 ||
    formData.latitude > 90 ||
    formData.longitude < -180 ||
    formData.longitude > 180
  ) {
    showAlert(
```

15.8 Utility Functions

This collection of utility functions provides essential interactivity, user feedback, and security for the map-based dashboard. The zoomToTrail allows users to instantly focus on a specific trail by locating it in the allTrailsData array, extracting its coordinates, and zooming to the centre of the Leaflet map. The updateTrailCount updates the screen counter to reflect how many trails are currently loaded or visible, improving user awareness during searches or filtering. The showLoading function enhances usability by temporarily disabling the search button and displaying a loading animation whenever a background operation, then restoring it once complete. Meanwhile, showAlert provides visual, dismissible alerts at the top-right corner of the screen for notifications such as successful saves, validation warnings, or API errors, these alerts automatically fade out after five seconds to keep the interface clean. Finally, getCsrfToken ensures secure communication with Django’s backend by retrieving the CSRF token using multiple fallback methods from cookies, a meta tag, or hidden form inputs.

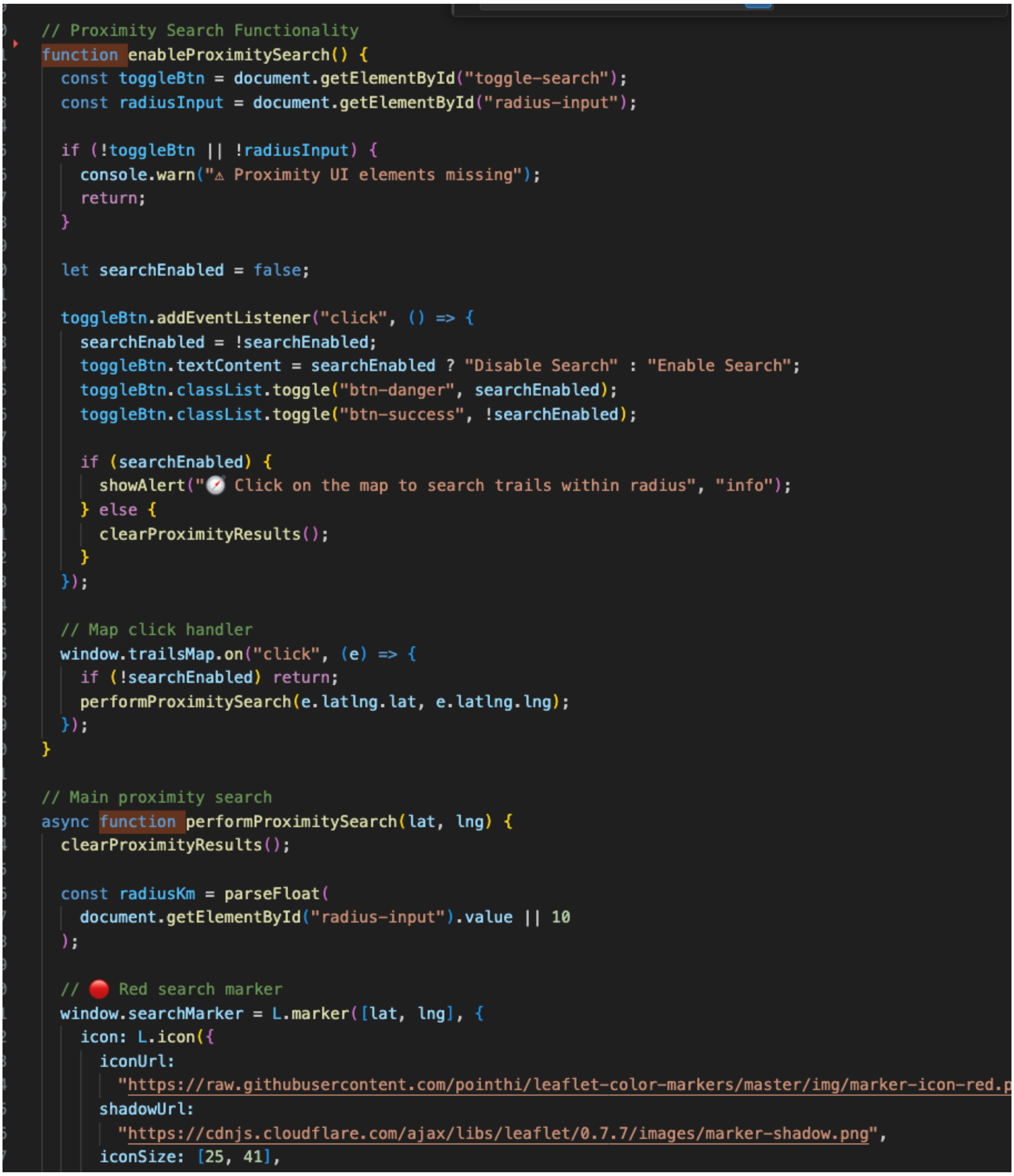


```javascript
// Utility functions
function zoomToTrail(trailId) {
  const trail = allTrailsData.find(
    (c) => c.properties.id === parseInt(trailId)
  );

  if (trail && trail.geometry && trail.geometry.coordinates) {
    const [lng, lat] = trail.geometry.coordinates;

    if (!isNaN(lat) && !isNaN(lng)) {
      map.setView([lat, lng], 12);
    }
  }
}

function updateTrailCount(count) {
  const countElement = document.getElementById("trail-count");

  if (countElement) {
    countElement.textContent = `${count} trails loaded`;
  }
}

function showLoading(show) {
  const searchBtn = document.getElementById("search-btn");
  if (searchBtn) {
    if (show) {
      searchBtn.innerHTML = '<span class="loading"></span> Loading...';

      searchBtn.disabled = true;
    } else {
      searchBtn.innerHTML = "🔍 Search";

      searchBtn.disabled = false;
    }
  }
}

function showAlert(message, type) {
  // Create alert element

  const alertDiv = document.createElement("div");
  alertDiv.className = `alert alert-${type} alert-dismissible fade show position-fixed`;
  alertDiv.style.top = "20px";
  alertDiv.style.right = "20px";
  alertDiv.style.zIndex = "9999";
  alertDiv.style.minWidth = "300px";
```

15.9 Proximity search function

The proximity search functionality enables users to interactively find trails near a chosen location on the map and also identify the nearest town to that point. The process starts with the enableProximitySearch function, which activates a toggle button allowing users to switch proximity mode on or off. When enabled, the button changes colour and label, and a message instructs users to click anywhere on the map to start the search. Upon clicking, the function captures the clicked coordinates and passes them to performProximitySearch(lat, lng), which handles the main logic. This function first clears any previous search results, then places a red marker at the clicked point and draws a blue circular radius (in kilometers, converted to meters for Leaflet) around it to visually represent the search area. It then sends a POST request to the Django API endpoint, passing the latitude, longitude, and radius as JSON data along with the CSRF token for security.

If the API finds nearby trails, they're displayed on the map through displayNearestTrails, and the sidebar results panel is updated using updateResultsPanel. The user is notified with a success alert showing how many trails were found, showLoading(true/false) provides feedback to indicate when the search is active, while showAlert communicates success or errors.

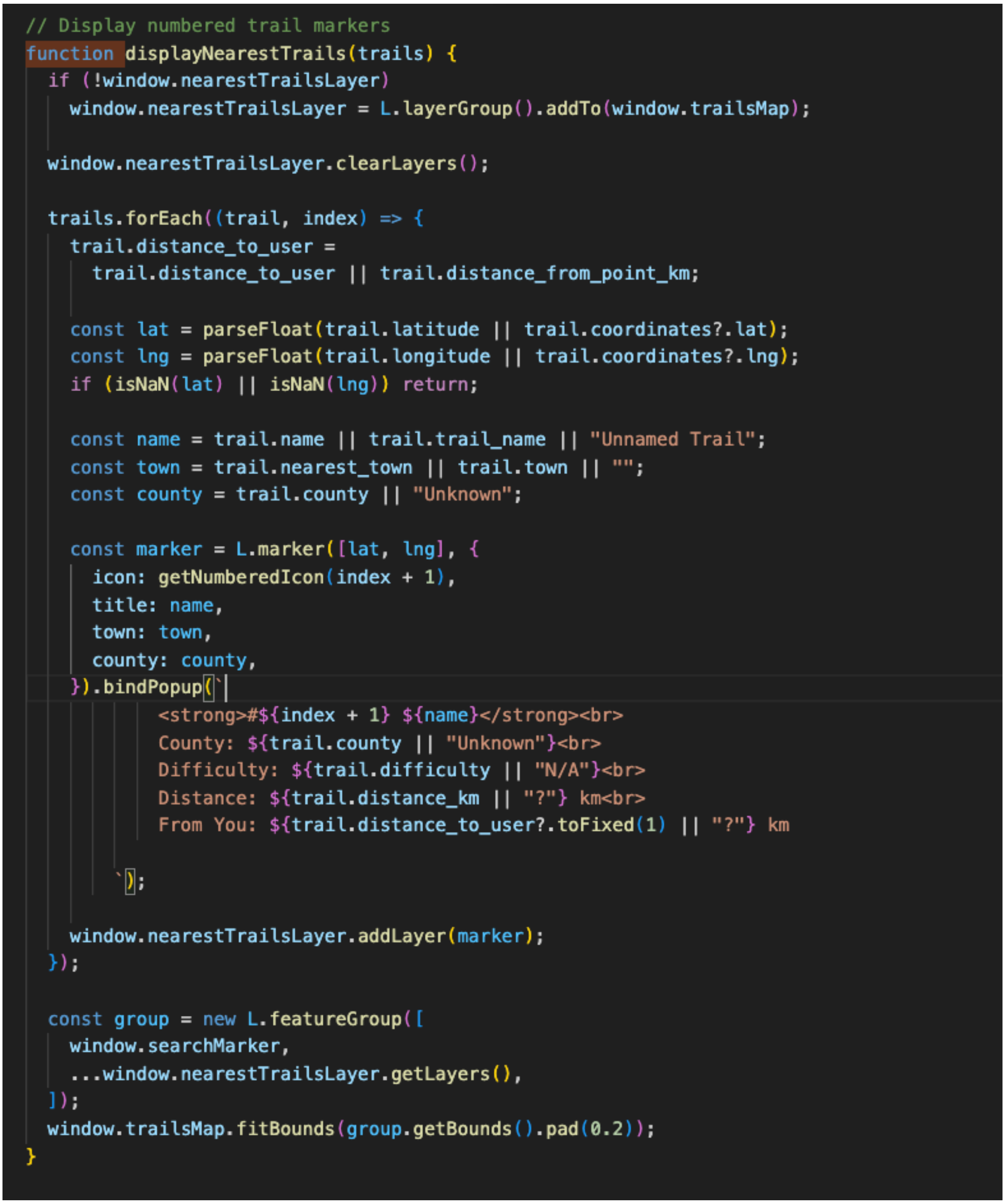


```
// Proximity Search Functionality
function enableProximitySearch() {
  const toggleBtn = document.getElementById("toggle-search");
  const radiusInput = document.getElementById("radius-input");

  if (!toggleBtn || !radiusInput) {
    console.warn("⚠ Proximity UI elements missing");
    return;
  }

  let searchEnabled = false;

  toggleBtn.addEventListener("click", () => {
    searchEnabled = !searchEnabled;
    toggleBtn.textContent = searchEnabled ? "Disable Search" : "Enable Search";
    toggleBtn.classList.toggle("btn-danger", searchEnabled);
    toggleBtn.classList.toggle("btn-success", !searchEnabled);

    if (searchEnabled) {
      showAlert("🧭 Click on the map to search trails within radius", "info");
    } else {
      clearProximityResults();
    }
  });

  // Map click handler
  window.trailsMap.on("click", (e) => {
    if (!searchEnabled) return;
    performProximitySearch(e.latlng.lat, e.latlng.lng);
  });
}

// Main proximity search
async function performProximitySearch(lat, lng) {
  clearProximityResults();

  const radiusKm = parseFloat(
    document.getElementById("radius-input").value || 10
  );

  // 🔴 Red search marker
  window.searchMarker = L.marker([lat, lng], {
    icon: L.icon({
      iconUrl:
        "https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/img/marker-icon-red.p
      shadowUrl:
        "https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shadow.png",
      iconSize: [25, 41],
```

15.1.1 Display nearest trails

The displayNearestTrails function visually presents the list of trails found during a proximity search by placing numbered markers on the Leaflet map, helping users quickly identify each trail's position and ranking relative to their selected point. It first ensures that a global nearestTrailsLayer exists, creating one if needed and clears any existing markers from previous searches. It then iterates through the trails array returned by the API, assigning each trail a distance from the user (either distance_to_user or distance_from_point_km). The function extracts each trail's latitude and longitude, validating them before proceeding. For every valid trail, it creates a custom marker using getNumberedIcon(index + 1), which visually numbers the markers in the order they appear in the search results. Each marker includes a popup displaying key details such as the trail name, county, difficulty level, total trail length, and its distance from the user's clicked location.

After all markers are added, the function groups them together along with the user's red search marker and adjusts the map view to fit all elements neatly within the visible area using fitBounds.



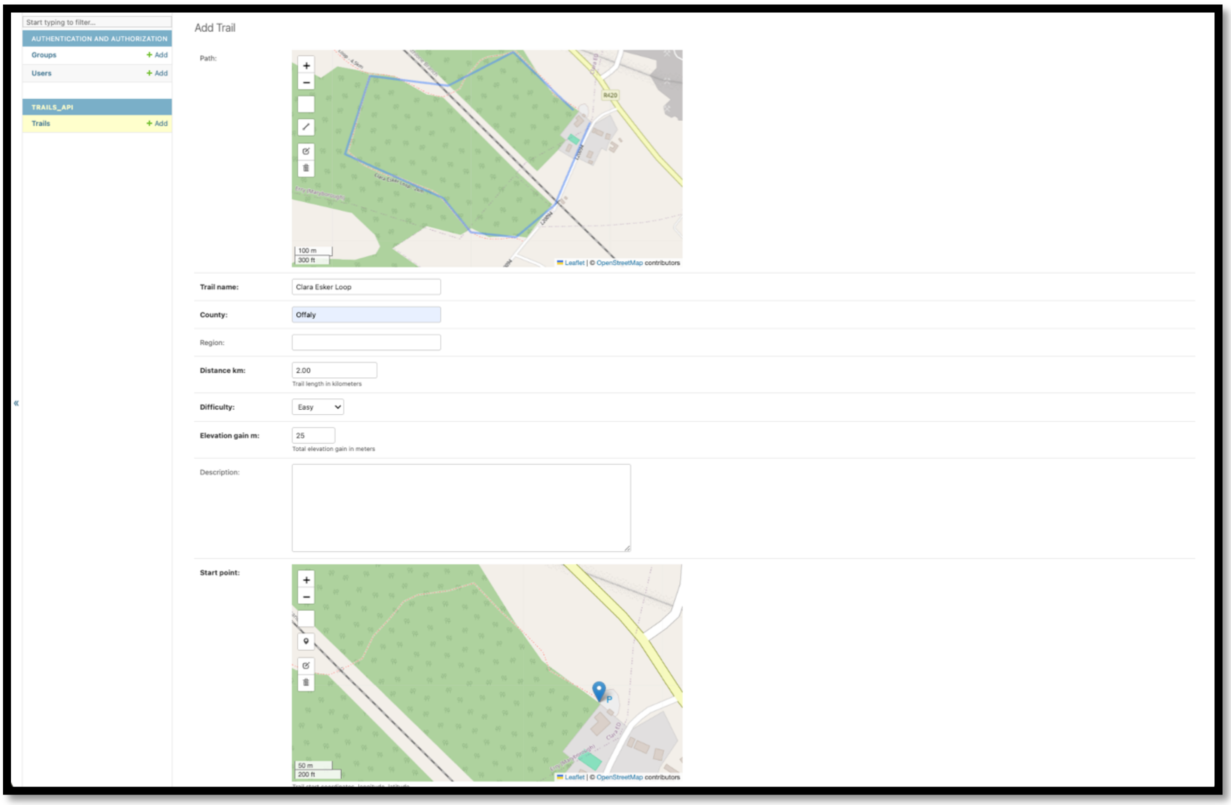
```
// Display numbered trail markers
function displayNearestTrails(trails) {
  if (!window.nearestTrailsLayer)
    window.nearestTrailsLayer = L.layerGroup().addTo(window.trailsMap);

  window.nearestTrailsLayer.clearLayers();

  trails.forEach((trail, index) => {
    trail.distance_to_user =
      trail.distance_to_user || trail.distance_from_point_km;

    const lat = parseFloat(trail.latitude || trail.coordinates?.lat);
    const lng = parseFloat(trail.longitude || trail.coordinates?.lng);
    if (isNaN(lat) || isNaN(lng)) return;

    const name = trail.name || trail.trail_name || "Unnamed Trail";
    const town = trail.nearest_town || trail.town || "";
    const county = trail.county || "Unknown";

    const marker = L.marker([lat, lng], {
      icon: getNumberedIcon(index + 1),
      title: name,
      town: town,
      county: county,
    }).bindPopup(`
            <strong>#${index + 1} ${name}</strong><br>
            County: ${trail.county || "Unknown"}<br>
            Difficulty: ${trail.difficulty || "N/A"}<br>
            Distance: ${trail.distance_km || "?"} km<br>
            From You: ${trail.distance_to_user?.toFixed(1) || "?"} km

        `);

    window.nearestTrailsLayer.addLayer(marker);
  });

  const group = new L.featureGroup([
    window.searchMarker,
    ...window.nearestTrailsLayer.getLayers(),
  ]);
  window.trailsMap.fitBounds(group.getBounds().pad(0.2));
}
```

15.1.2 Clear proximity results

This section combines two important features clearing proximity search results and drawing custom trail paths on the map.

The clearProximityResults function is responsible for resetting the map and interface after a proximity search. It removes the user’s red search marker, clears the nearestTrailsLayer that contains previously displayed numbered trail markers, and hides the proximity results panel from the sidebar. This ensures the map remains uncluttered when the user wants to perform a new search or switch focus to another feature.

The second part introduces an interactive trail path drawing tool using Leaflet’s L.Control.Draw plugin. The configuration disables shapes like circles, rectangles, and polygons, leaving only the polyline tool enabled for drawing trails. These drawn lines are styled in orange with a moderate line weight for visibility. Once initialised, the control is added to the map interface, giving users a toolbar to start creating paths. When a new polyline (trail path) is drawn, the L.Draw.Event.CREATED event triggers. The event captures the line’s latitude and longitude points using layer.getLatLngs, logs them to the console for debugging, and then sends them via a POST request to the Django endpoint /api/trails/add-path/.



Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

TRAILS_API

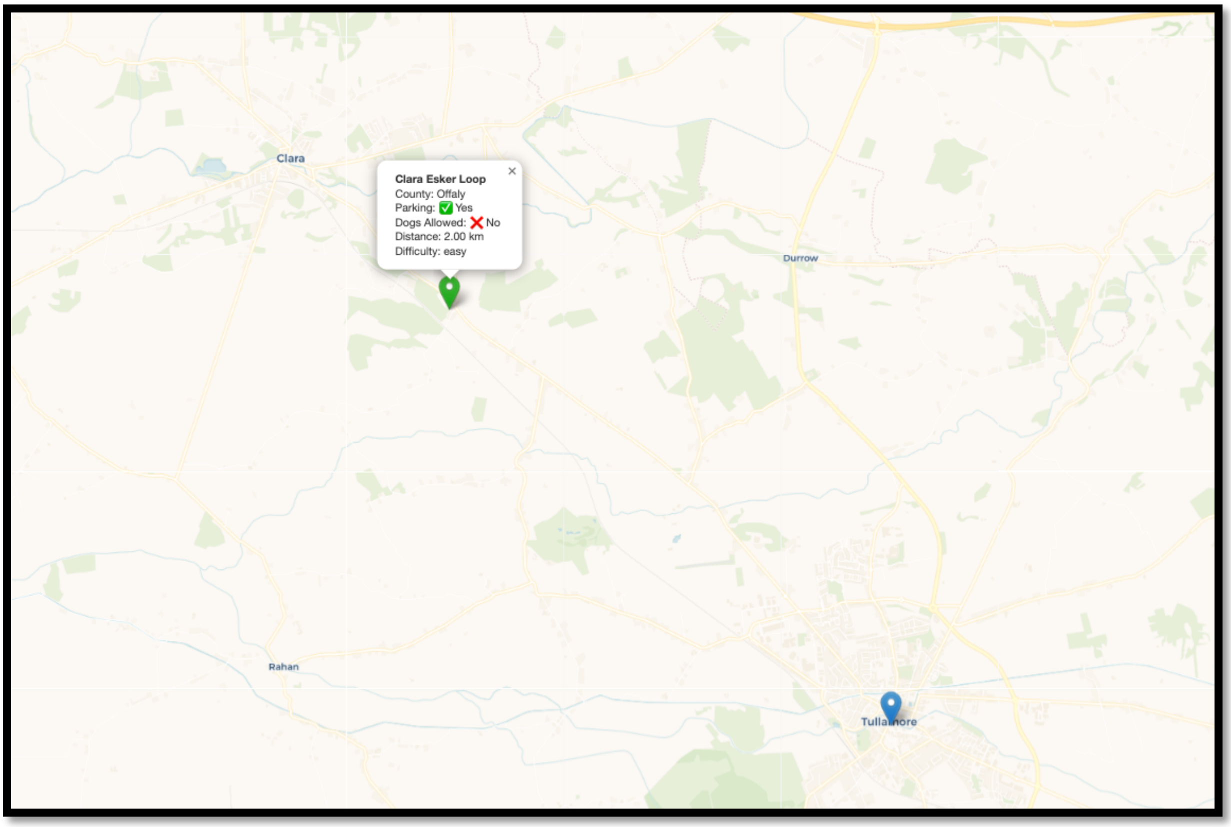
Trails + Add

«

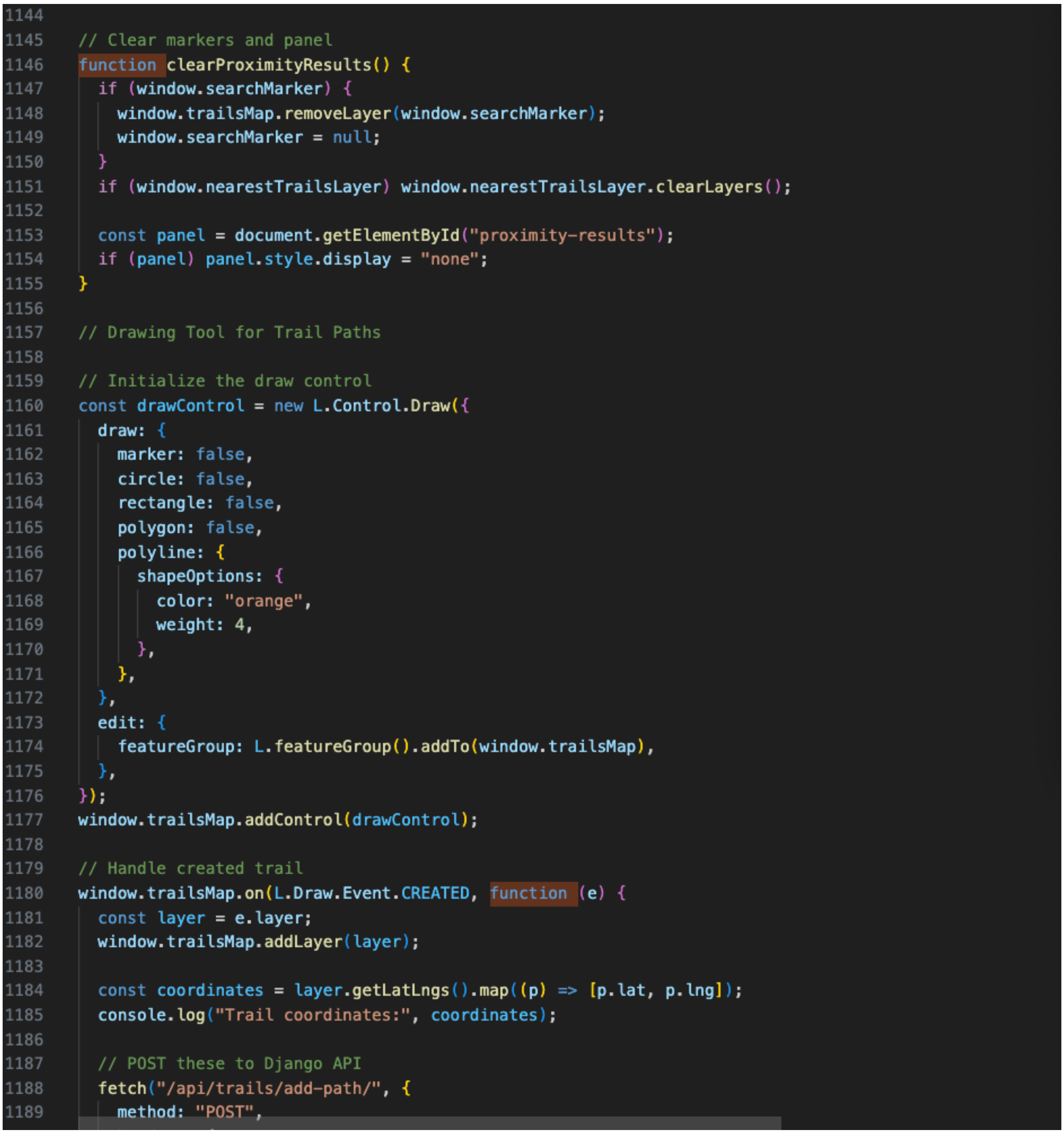
Add Trail

Path:	
Trail name:	Clara Esker Loop
County:	Offaly
Region:	
Distance km:	2.00 Trail length in kilometers
Difficulty:	Easy
Elevation gain m:	25 Total elevation gain in meters
Description:	
Start point:	Trail start coordinates, longitude, latitude

R420
100 m
300 ft
Leaflet | © OpenStreetMap contributors
50 m
200 ft
P
Leaflet | © OpenStreetMap contributors



Clara Esker Loop
County: Offaly
Parking: ✅ Yes
Dogs Allowed: ❌ No
Distance: 2.00 km
Difficulty: easy

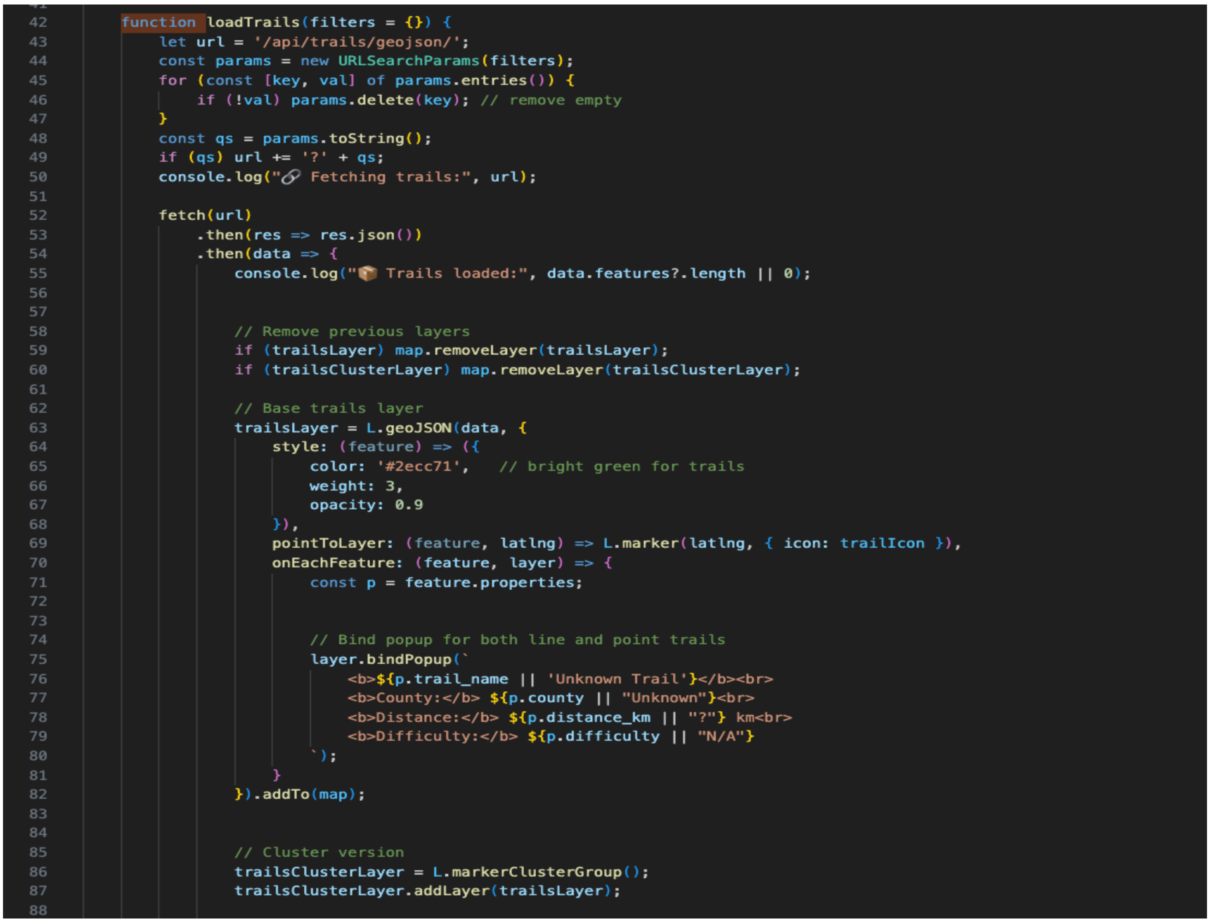


```
// Clear markers and panel
function clearProximityResults() {
  if (window.searchMarker) {
    window.trailsMap.removeLayer(window.searchMarker);
    window.searchMarker = null;
  }
  if (window.nearestTrailsLayer) window.nearestTrailsLayer.clearLayers();

  const panel = document.getElementById("proximity-results");
  if (panel) panel.style.display = "none";
}

// Drawing Tool for Trail Paths

// Initialize the draw control
const drawControl = new L.Control.Draw({
  draw: {
    marker: false,
    circle: false,
    rectangle: false,
    polygon: false,
    polyline: {
      shapeOptions: {
        color: "orange",
        weight: 4,
      },
    },
  },
  edit: {
    featureGroup: L.featureGroup().addTo(window.trailsMap),
  },
});
window.trailsMap.addControl(drawControl);

// Handle created trail
window.trailsMap.on(L.Draw.Event.CREATED, function (e) {
  const layer = e.layer;
  window.trailsMap.addLayer(layer);

  const coordinates = layer.getLatLngs().map((p) => [p.lat, p.lng]);
  console.log("Trail coordinates:", coordinates);

  // POST these to Django API
  fetch("/api/trails/add-path/", {
    method: "POST",
```

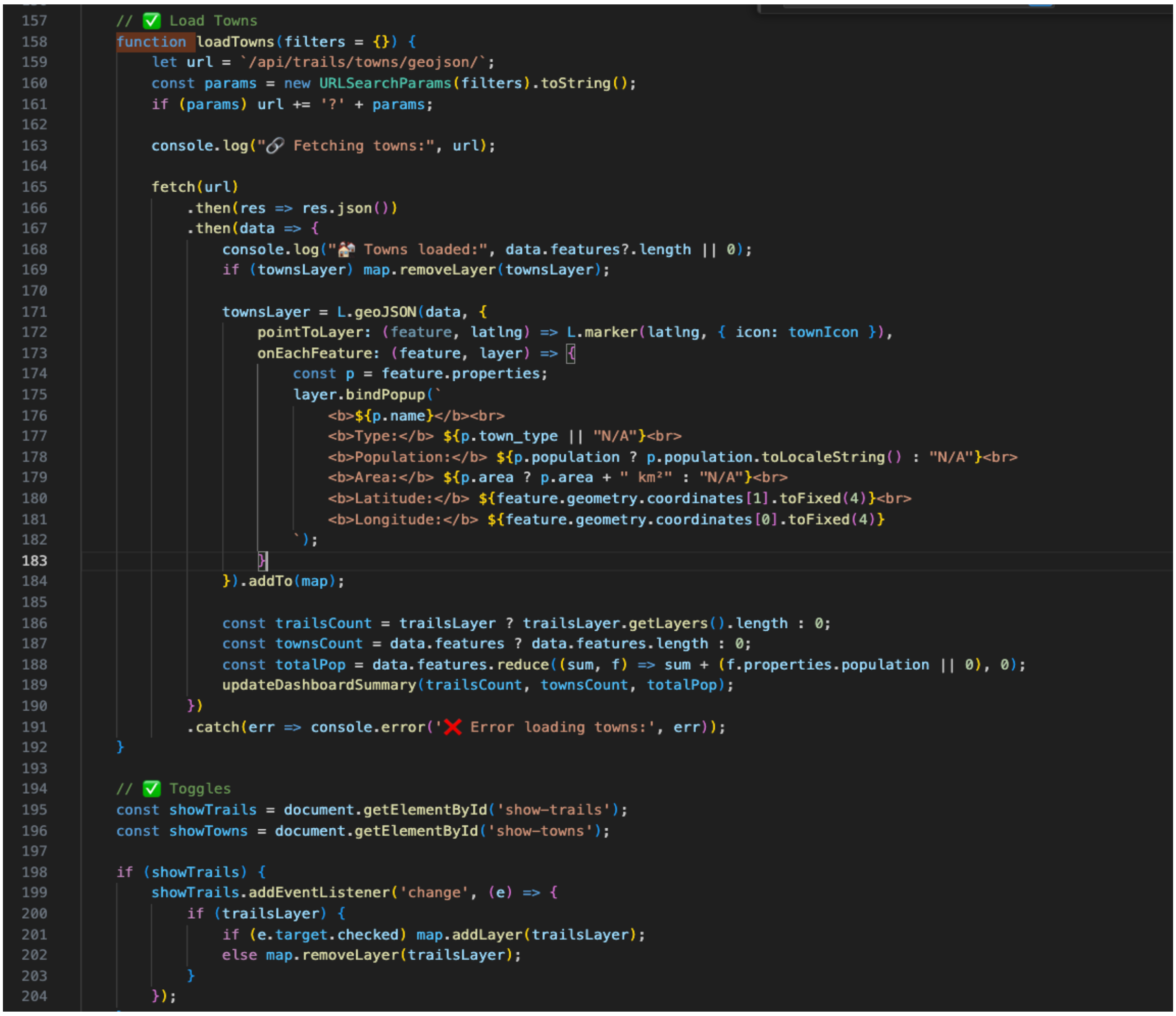
15.1.3 Load trails, towns, filtering and display

This block of code handles loading, filtering, and displaying trail data dynamically on the interactive map. The main function, loadTrails(filters), builds a URL that includes any filters trail length, county, or difficulty as query parameters, then fetches trail data from the /api/trails/geojson/ endpoint. Once the data is received, it removes any existing trail layers from the map and creates new GeoJSON layers with each trail styled in green. Each trail is represented as either a line or point marker with a popup showing its name, county, distance, and difficulty. A cluster version (L.markerClusterGroup) is also created to group nearby markers visually when zoomed out.

The event listener for the “Apply Filters” button collects filter inputs from the dashboard like trail length range, difficulty, county, and trail type validates them, and calls both loadTrails(filters) and loadTowns(filters) to refresh the map view based on those filters.



```
function loadTrails(filters = {}) {
    let url = '/api/trails/geojson/';
    const params = new URLSearchParams(filters);
    for (const [key, val] of params.entries()) {
        if (!val) params.delete(key); // remove empty
    }
    const qs = params.toString();
    if (qs) url += '?' + qs;
    console.log("🔗 Fetching trails:", url);

    fetch(url)
        .then(res => res.json())
        .then(data => {
            console.log("📦 Trails loaded:", data.features?.length || 0);


            // Remove previous layers
            if (trailsLayer) map.removeLayer(trailsLayer);
            if (trailsClusterLayer) map.removeLayer(trailsClusterLayer);

            // Base trails layer
            trailsLayer = L.geoJSON(data, {
                style: (feature) => ({
                    color: '#2ecc71',   // bright green for trails
                    weight: 3,
                    opacity: 0.9
                }),
                pointToLayer: (feature, latlng) => L.marker(latlng, { icon: trailIcon }),
                onEachFeature: (feature, layer) => {
                    const p = feature.properties;


                    // Bind popup for both line and point trails
                    layer.bindPopup(`
                        <b>${p.trail_name || 'Unknown Trail'}</b><br>
                        <b>County:</b> ${p.county || "Unknown"}<br>
                        <b>Distance:</b> ${p.distance_km || "?"} km<br>
                        <b>Difficulty:</b> ${p.difficulty || "N/A"}
                    `);
                }
            }).addTo(map);


            // Cluster version
            trailsClusterLayer = L.markerClusterGroup();
            trailsClusterLayer.addLayer(trailsLayer);

```



```javascript
// ✅ Load Towns
function loadTowns(filters = {}) {
    let url = `/api/trails/towns/geojson/`;
    const params = new URLSearchParams(filters).toString();
    if (params) url += '?' + params;

    console.log("🔗 Fetching towns:", url);

    fetch(url)
        .then(res => res.json())
        .then(data => {
            console.log("🏘 Towns loaded:", data.features?.length || 0);
            if (townsLayer) map.removeLayer(townsLayer);

            townsLayer = L.geoJSON(data, {
                pointToLayer: (feature, latlng) => L.marker(latlng, { icon: townIcon }),
                onEachFeature: (feature, layer) => {
                    const p = feature.properties;
                    layer.bindPopup(`
                        <b>${p.name}</b><br>
                        <b>Type:</b> ${p.town_type || "N/A"}<br>
                        <b>Population:</b> ${p.population ? p.population.toLocaleString() : "N/A"}<br>
                        <b>Area:</b> ${p.area ? p.area + " km²" : "N/A"}<br>
                        <b>Latitude:</b> ${feature.geometry.coordinates[1].toFixed(4)}<br>
                        <b>Longitude:</b> ${feature.geometry.coordinates[0].toFixed(4)}
                    `);
                }
            }).addTo(map);

            const trailsCount = trailsLayer ? trailsLayer.getLayers().length : 0;
            const townsCount = data.features ? data.features.length : 0;
            const totalPop = data.features.reduce((sum, f) => sum + (f.properties.population || 0), 0);
            updateDashboardSummary(trailsCount, townsCount, totalPop);
        })
        .catch(err => console.error('❌ Error loading towns:', err));
}

// ✅ Toggles
const showTrails = document.getElementById('show-trails');
const showTowns = document.getElementById('show-towns');

if (showTrails) {
    showTrails.addEventListener('change', (e) => {
        if (trailsLayer) {
            if (e.target.checked) map.addLayer(trailsLayer);
            else map.removeLayer(trailsLayer);
        }
    });
```

Trails Within Radius

The trails_within_radius function allows users to find all hiking trails located within a given distance of a specific point on the map. It accepts three key parameters latitude, longitude, and radius_km, which are sent via a POST request from the frontend whenever a user clicks on the map and defines a radius. The function converts these coordinates into a GeoDjango Point object and uses spatial filtering with start_point__distance_lte and the Distance function to calculate which trails fall within that circle. The results are then ordered by proximity and returned as a GeoJSON response to the map, where the nearest trails are displayed as numbered markers.

Nearest Town

The nearest_town function determines the closest town to a user's location. It takes latitude and longitude parameters and creates a spatial query that annotates each town in the database with its distance from the provided coordinates. Using Distance('location', user_point) from django.contrib.gis.db.models.functions, the query orders the towns by distance and returns the one with the shortest value.

Trails in Bounding Box

The trails_in_bounding_box function supports rectangular area searches. It receives the bounding coordinates (min_lng, min_lat, max_lng, max_lat) from the frontend and constructs a spatial filter using GeoDjango's Polygon geometry. Any trail whose start_point lies within that polygon is included in the response.

Trails GeoJSON and Towns GeoJSON

Both trails_geojson and towns_geojson functions are responsible for providing spatial data to the frontend map in GeoJSON format. These functions query the Trail and Town models, serialize them using Django's serialize method, and return the data as JSON suitable for Leaflet rendering.

Trail Statistics

The trail_statistics function performs aggregated analysis on the trails dataset. It calculates metrics such as the total number of trails, the average trail length Avg(‘distance_km’), the maximum elevation gain, and counts grouped by difficulty (using values('difficulty').annotate(count=Count('difficulty'))).

Town Filtering and Grouping

Within the towns_geojson view, additional filters can be applied via GET parameters such as town_type, min_population, and max_population. The query dynamically adjusts to these parameters using conditions like Town.objects.filter(town_type__iexact=town_type) and population range filters (population__gte and population__lte). These filters allow users to interactively explore towns by category.

Trail Search and Popup Details

The trail_search function handles keyword-based trail lookups. Users can search by trail name, county, or region. The view uses Django’s Q objects to combine these conditions in a flexible way, for example.

```
Trail.objects.filter(
    Q(trail_name__icontains=query) |
    Q(county__icontains=query) |
    Q(region__icontains=query)
)
```

When results are displayed on the map, each trail marker includes a popup with a description, county, difficulty, and distance. These popups are built dynamically in JavaScript using Leaflet’s bindPopup function, allowing users to click on any marker and view detailed trail information.

Dashboard Analytics

The dashboard.views.analytics function aggregates both trail and town data for visual reporting. It groups trails by difficulty and county, and towns by type using Django's ORM aggregation. This data is passed to Leaflet visualizations, where users can see comparisons such as the number of hard vs. easy trails, average trail distances by county, or the population distribution among town types. The analytics view combines data from multiple endpoints, providing a clear and interactive summary of Ireland's trail and town network.

16. Summary

This project demonstrates a complete, working Django GeoDjango system with map interaction, proximity search, trail and town management, and fully tested endpoints. It involved solving several issues with migrations, imports, and map data, giving a strong understanding of spatial databases and APIs. This work directly supports my Stay & Trek final year project.