**Survey Web App Documentation**

**Introduction**

This document serves as comprehensive documentation for the Survey App codebase. The Survey App is a web application designed for map visualization, placemark management, and layer interactions. This document provides an overview of the primary files and functionalities within the application, aiming to assist future developers in understanding the structure and implementation of the code.

**Scope**

This documentation aims to provide future developers with a clear understanding of the Survey App's codebase, enabling efficient navigation, troubleshooting, and potential enhancements. Each section provides insights into the purpose and functionality of the respective components within the application.

**Overview**

**1. `index.html`:** The main HTML document defining the structure of the Survey App.

**2. `utility.js`:** Utility functions facilitating map visualization and interaction.

**3. `placemarks.js`:** Functionality for managing and rendering placemarks on the map.

**4. `polylines.js`:** Handling the drawing and management of polylines on the map.

**5. `layers.js`:** Management of layers, including saving, deleting, and switching between layers.

**6. `download.js`:** Handling the download functionality for map content.

**7. `upload\_kml.js`:** Uploading KML files, parsing contents, and creating new layers.

**8. `upload\_csv.js`:** Uploading CSV files, processing data, and rendering polylines on the map.

**9. `style.css`:** CSS styles for styling specific elements in the Survey App.

**10. `image\_tile.sh`:** A script for downloading map tiles from Mapbox.

**SPECIFICATIONS**

**1. SOFTWARE REQUIREMENTS**

| **Operating System** | Linux |
| --- | --- |
| **Web Browser** | Google Chrome, Mozilla Firefox |
| **Programming Languages** | JavaScript (ES6+), HTML5, CSS3, Bootstrap (v4.4.1) |
| **Frontend Framework** | HTML5, CSS3 |
| **Backend Framework** | JavaScript (ES6+) |
| **Data Formats** | CSV, KML |
| **Version Control System** | Git |
| **Libraries** | PapaParse (5.3.0), html2canvas (1.4.1) |
| **API Key** | Mapbox |

**2. HARDWARE REQUIREMENTS**

| **PROCESSOR** | Dual-core processor or higher |
| --- | --- |
| **RAM** | 4 GB or Higher |
| **STORAGE** | 20 GB available disk space |
| **GRAPHICS** | Graphics card for enhanced mapping visualization (optional) |

# 

# 

# **`index.html` - Overview and Structure**

The `index.html` file is the main HTML document for the Survey App. It includes various elements such as the navigation bar, buttons for interacting with the map, a container for layers in the sidebar, and the map itself. Additionally, it imports necessary external libraries, Bootstrap styles, and custom JavaScript modules.

## **Key Sections**

**1. Head Section:**

- Meta tags for character set and viewport settings.

- Title of the HTML document.

- Links to Bootstrap CSS stylesheets and a custom stylesheet (`/css/style.css`).

**2. Body Section:**

- Navbar with buttons for adding placemarks, drawing polylines, saving layers, and more.

- Containers for placemarks, polylines, and a canvas within the map container.

- A sidebar for displaying saved layers.

- Importing Bootstrap JavaScript, jQuery, Popper.js, PapaParse, and html2canvas libraries.

- Importing custom JavaScript modules:

- `utility.js`

- `placemarks.js`

- `polylines.js`

- `layers.js`

- `download.js`

- `upload\_kml.js`

- `upload\_csv.js`

## **Buttons and Actions**

**- Navbar Buttons:**

- Add Placemarks (`addPlacemarksBtn`)

- Clear Placemarks (`clearPlacemarksBtn`)

- Add Polyline (`drawPolylineBtn`)

- Clear Polyline (`clearPolylineBtn`)

- Save Layer (`saveLayerBtn`)

- Delete Layer(s) (`deleteLayerBtn`)

- Download as Image or KML (`downloadImageBtn` and `downloadKMLBtn`)

- Upload KML (`uploadKMLBtn`)

- Upload CSV (`uploadCSVBtn`)

## **Map Containers**

**- Map Container:**

- Div with class `container-fluid` containing rows for the sidebar and map.

- Map displayed as an image (`map`), with additional containers for placemarks and polylines.

**- Sidebar (4 columns):**

- Div with class `col-md-4` containing the sidebar (`sidebar`).

- Sidebar content dynamically generated using JavaScript.

**- Map (8 columns):**

- Div with class `col-md-8` containing the map (`map-container`).

- Map image (`map`), placemarks container (`placemark-container`), polylines container (`polylines-container`), and canvas (`canvas`).

## **Script Tags**

**- Script Tags:**

- Importing jQuery, Popper.js, and Bootstrap JavaScript libraries.

- Importing PapaParse and html2canvas libraries.

- Importing custom JavaScript modules with the `type="module"` attribute.

## **Custom JavaScript Modules**

**- Custom JavaScript Modules:**

- `utility.js`: Contains utility functions.

- `placemarks.js`: Handles placemark-related functionality.

- `polylines.js`: Manages polyline drawing and rendering.

- `layers.js`: Manages saved layers, layer buttons, and switching between layers.

- `download.js`: Handles downloading map content as an image or KML.

- `upload\_kml.js`: Handles uploading KML files and plotting features on the map.

- `upload\_csv.js`: Handles uploading CSV files, processing data, and rendering polylines.

## **Conclusion**

The `index.html` file serves as the entry point for the Survey App. It defines the structure of the web page, including navigation elements, map containers, and script imports. The use of Bootstrap and custom JavaScript modules enhances the functionality and appearance of the application.

# **`image\_tile.sh` - Overview**

The `image\_tile.sh` script is designed to download a map tile from Mapbox based on specified parameters such as center longitude, center latitude, and zoom level. The script generates a configuration file (`config.js`) for the Survey App and utilizes cURL to fetch and save the map tile image. Below is a detailed breakdown of the script:

## **Usage Guidelines:**

**1. Make the Script Executable:**

- Make the script executable using the command: `chmod +x image\_tile.sh`.

**2. Execute the Script:**

- Execute the script by providing three command-line arguments:

- `$1`: Center Longitude (float)

- `$2`: Center Latitude (float)

- `$3`: Zoom Level (integer)

- Example Usage:

```bash

./image\_tile.sh 80.203561 12.975272 16

```

## **Script Explanation:**

**1. Configuration File (`config.js`):**

- Generates a `config.js` file for the Survey App with the specified center longitude, center latitude, and zoom level.

**2. Print Information:**

- Outputs the provided center longitude, center latitude, and zoom level to the console.

**3. Mapbox Static Image URL Construction:**

- Constructs the URL for fetching a Mapbox static image based on provided parameters.

**4. Map Tile Download:**

- Uses cURL to download the Mapbox static image and saves it as `mapTile.jpeg` in the `images` directory.

**5. Print Completion Message:**

- Outputs a message indicating that the map tile has been downloaded.

## **Conclusion:**

The `image\_tile.sh` script is a convenient tool for obtaining map tiles from Mapbox based on specified geographical coordinates and zoom levels. It serves the purpose of providing a visual representation of the specified location in the Survey App.

# **`utility.js` - Map Utility Functions**

The `utility.js` file houses essential utility functions utilized in the map visualization application. These functions facilitate the calculation of image coordinates and positions based on geographical coordinates, enabling accurate mapping and interaction within the application.

## **`calculateImageCoordinates` Function**

The `calculateImageCoordinates` function extracts relevant information from the configuration object, particularly the zoom level, center latitude, and center longitude. Using this information, it computes the coordinates of the image corners based on the specified zoom level. The resulting coordinates are crucial for mapping geographical locations to positions on the displayed image.

// Import the configuration object from config.js

import config from './config.js';

// Call the calculateImageCoordinates function with the config object

const coordinates = calculateImageCoordinates(config);

// Log the calculated coordinates

console.log('Image Coordinates:', coordinates);

## **`getImagePosition` Function**

The `getImagePosition` function calculates the X and Y positions within the image based on the clicked geographical coordinates. It takes into account the upper left and lower right coordinates of the image, allowing precise positioning of elements on the map.

// Assuming upper\_left\_lat, upper\_left\_lon, lower\_right\_lat, lower\_right\_lon are known

const clickedCoordinates = { input\_lat: /\* Your latitude \*/, input\_lon: /\* Your longitude \*/ };

// Call the getImagePosition function with the clicked coordinates and image boundaries

const imagePosition = getImagePosition(

clickedCoordinates.input\_lat,

clickedCoordinates.input\_lon,

upper\_left\_lat\_coordinate,

upper\_left\_lon\_coordinate,

lower\_right\_lat\_coordinate,

lower\_right\_lon\_coordinate

);

// Log the calculated image positions

console.log('Image Position:', imagePosition);

## **Conclusion:**

These utility functions play a crucial role in translating geographical data to image positions, forming the backbone of the map interaction capabilities within the application.

# 

# **`placemarks.js` - Placemark Management**

The `placemarks.js` file contains functionality for managing placemarks on the map. Placemarks are interactive markers placed on the map at specified coordinates, providing a visual representation of points of interest.

## **Initialization and Import**

The module begins by importing necessary functions from `utility.js` for coordinate calculations and referencing the configuration object from `config.js`. This establishes the foundation for the placemark-related functionalities.

// Import necessary functions and configuration from utility.js and config.js

import { calculateImageCoordinates, getImagePosition } from './utility.js';

import config from './config.js';

## **Placemark Management Variables**

The module initializes essential variables for placemark management, including a boolean variable to control placemark placement (`isPlacemarkEnabled`) and an array to store placemark objects (`placemarks`). Additionally, a reference to the map image element is obtained.

// Initialize the boolean variable to control placemark placement

let isPlacemarkEnabled = false;

// Initialize the array to store placemark objects

export const placemarks = [];

// Get the reference to the image element with id "map"

export const imageElement = document.getElementById('map');

## **Placemark Event Listeners**

Event listeners are added to the "Add Placemarks" button and the map image element to enable placemark placement and respond to user interactions. The "Clear Placemarks" button is also configured to clear existing placemarks when clicked.

const addPlacemarksBtn = document.getElementById('addPlacemarksBtn');

addPlacemarksBtn.addEventListener('click', () => {

// Toggle the boolean variable for enabling/disabling placemark placement

// Change the button text based on the state

// Add an event listener for the click event on the "Clear Placemarks" button

});

imageElement.addEventListener('click', (event) => {

// Check if placemark placement is enabled

// Get the current image coordinates based on the configuration

// Calculate the latitude and longitude of the clicked point

// Create a new placemark object with calculated coordinates and an image source

// Append the new placemark object to the placemarks array

// Render the updated placemarks on the map

});

## **Placemark Rendering Function**

The `renderPlacemarks` function is responsible for rendering placemarks on the map. It clears existing placemarks and renders each placemark using calculated image positions.

// Function to render placemarks on the map

export function renderPlacemarks(coordinates, placemarks, getImagePosition) {

// Clear existing placemarks on the map

// Render each placemark on the map

}

This function uses the `getImagePosition` utility function to translate geographical coordinates into image positions and appends corresponding images to the map.

## **Conclusion**

The `placemarks.js` module forms a crucial component of the map visualization application, allowing users to interactively add and visualize placemarks on the map.

# **`polylines.js` - Polyline Drawing and Management**

The `polylines.js` file handles the drawing and management of polylines on the map. Polyline drawing is enabled by clicking on the map, and the module provides functionality to render and clear polylines.

## **Initialization and Imports**

The module starts by importing necessary functions and objects from `utility.js` and `placemarks.js`, establishing dependencies for polyline functionality.

// Import necessary functions and objects from utility.js and placemarks.js

import { calculateImageCoordinates, getImagePosition } from './utility.js';

import { imageElement } from './placemarks.js';

import config from './config.js';

## **Polyline Management Variables**

The module initializes variables for managing polyline drawing, including a boolean variable to control polyline placement (`isPolylineEnabled`) and an array to store polyline coordinates (`polylineCoordinates`). Additionally, the `canvas` element and its 2D rendering context (`ctx`) are obtained.

// Initialize the boolean variable to control polyline placement

let isPolylineEnabled = false;

// Exported variable to store polyline coordinates

export let polylineCoordinates = [];

// Add an event listener for the "Add Polyline" button

// Update the event listener for the image click to handle polyline drawing

// Exported function to render a polyline on the canvas

// Function to clear polylines on the map (clear the canvas)

## 

## **Event Listeners for Polyline Drawing**

Event listeners are added to the "Add Polyline" button and the map image element to enable polyline drawing and respond to user interactions. The module also exports a function to render polylines on the canvas.

const drawPolylineBtn = document.getElementById('drawPolylineBtn');

drawPolylineBtn.addEventListener('click', () => {

// Toggle the boolean variable

// Change the button text based on the state

// Console log the state of polyline drawing and coordinates

});

imageElement.addEventListener('click', (event) => {

// Check if polyline drawing is enabled

// Get the latitude and longitude of the clicked point using utility.js functions

// Calculate the clicked coordinates based on the clicked point on the image

// Add the clicked coordinates to the polylineCoordinates array

// Draw the polyline on the canvas using the renderPolyline function

// Console log the updated coordinates

});

// Event listener for the "Clear Polyline" button

clearPolylineBtn.addEventListener('click', () => {

// Clear the polyline array

// Clear the polylines on the map (clear the canvas)

// Console log the action

});

## **Rendering Polylines on Canvas**

The module defines a function, `renderPolyline`, responsible for rendering polylines on the canvas. It clears the canvas and draws the polyline based on the stored coordinates.

export function renderPolyline(coordinates, polylineCoordinates, getImagePosition, ctx) {

// Clear the canvas

// Draw the new polyline on the canvas

// Set the polyline style and stroke

}

## **Conclusion**

The `polylines.js` module extends the functionality of the map visualization application by allowing users to draw and manage polylines. The module offers features for enabling/disabling polyline drawing, rendering polylines on the canvas, and clearing drawn polylines.

# **`layers.js` - Managing Layers and Saved Configurations**

The `layers.js` file manages layers and provides functionality for saving, deleting, and switching between layers in the map visualization application.

## **Initialization and Imports**

The module starts by importing necessary functions, configurations, and objects from external files, including `utility.js`, `placemarks.js`, `polylines.js`, and `config.js`.

// Importing necessary functions and configurations from external files

import { calculateImageCoordinates, getImagePosition } from './utility.js';

import { placemarks, renderPlacemarks } from './placemarks.js';

import { polylineCoordinates, renderPolyline, ctx } from './polylines.js';

import config from './config.js';

## **Managing Saved Layers**

The module declares an array (`savedLayers`) to store saved layers. It also includes an event listener for the "Save Layer" button, which prompts the user for a layer name and creates a new layer object containing placemarks, polyline coordinates, and the layer name.

// Declaring an array to store saved layers

export let savedLayers = [];

// Event listener for the "Save Layer" button

// Event listener for the "Delete Layer(s)" button

// Function to display a checkbox for the saved layer in the sidebar

// Function to switch to a saved layer

## **Save Layer Event Listener**

The `saveLayerBtn` event listener triggers when the "Save Layer" button is clicked. It prompts the user for a layer name and, if provided, creates a new layer object. The new layer is added to the `savedLayers` array, and a button for the saved layer is displayed in the sidebar.

const saveLayerBtn = document.getElementById('saveLayerBtn');

saveLayerBtn.addEventListener('click', () => {

// Prompt the user for a layer name

// Check if the user entered a layer name

// Create a new layer object containing placemarks, polylineCoordinates, and the layer name

// Add the new layer to the savedLayers array

// Display a button for the saved layer in the sidebar

});

## **Delete Layer Event Listener**

The `deleteLayerBtn` event listener triggers when the "Delete Layer(s)" button is clicked. It retrieves all checkboxes for saved layers, iterates through them, and deletes corresponding layers from the `savedLayers` array. The checkboxes and labels in the sidebar are also removed.

const deleteLayerBtn = document.getElementById('deleteLayerBtn');

deleteLayerBtn.addEventListener('click', () => {

// Get all checkboxes for saved layers

// Iterate through checkboxes and delete corresponding layers

});

## **Display Saved Layer Button Function**

The `displaySavedLayerButton` function is responsible for creating a checkbox and label for a saved layer in the sidebar. It takes the layer number and name as parameters and appends the elements to the sidebar.

// Function to display a checkbox for the saved layer in the sidebar

export function displaySavedLayerButton(layerNumber, layerName) {

// Create a container div for the saved layer

// Create a checkbox for the saved layer

// Create a label for the checkbox with the layer name

// Add an event listener to switch to the saved layer when the checkbox is clicked

// Append the checkbox and label to the layer container

// Append the container to the sidebar

}

## **Switch to Saved Layer Function**

The `switchToSavedLayer` function is responsible for switching to a saved layer when its checkbox is clicked. It updates the current placemarks and polyline coordinates arrays based on the selected layers, then renders the placemarks and polylines on the map.

// Function to switch to a saved layer

function switchToSavedLayer(layerNumber) {

// Update the current placemarks and polylineCoordinates arrays based on the selected layer

// Iterate through all checkboxes to combine placemarks and polylineCoordinates from selected layers

// Render the placemarks and polylines on the map

}

## **Conclusion**

The `layers.js` module provides essential functionality for managing layers in the map visualization application. Users can save, delete, and switch between layers, each containing placemarks and polyline configurations. The sidebar displays checkboxes for saved layers, allowing users to toggle their visibility.

# **`download.js` - Downloading Map Content**

The `download.js` file handles the download functionality for the map content, providing options to download the map as an image (PNG) or as KML (Keyhole Markup Language) files. The code imports placemarks and polylineCoordinates from external files (`placemarks.js` and `polylines.js`) and utilizes the `html2canvas` library for capturing the map as an image.

## **Downloading Map as Image (PNG)**

The script adds an event listener to the "Image" dropdown item (`downloadImageBtn`). When clicked, it captures a screenshot of the `map-container` using `html2canvas`, creates a download link for the image, and triggers a click event to start the download.

// DOWNLOAD IMAGE

const downloadImageBtn = document.getElementById('downloadImageBtn');

downloadImageBtn.addEventListener('click', () => {

// Capture the screenshot of the map-container using html2canvas

// Create an anchor element to download the image

// Set the download link href to the image data URL

// Trigger a click event to start the download

});

## **Downloading Map Data as KML**

The script adds an event listener to the "KML" dropdown item (`downloadKMLBtn`). When clicked, it generates separate KML files for placemarks and polylines using the `generatePlacemarksKML` and `generatePolylinesKML` functions, combines them into a single KML file, creates a Blob from the KML content, and initiates the download.

// DOWNLOAD KML

const downloadKMLBtn = document.getElementById('downloadKMLBtn');

downloadKMLBtn.addEventListener('click', () => {

// Generate separate KML files for placemarks and polylines

// Combine placemarks and polylines KML content

// Create a Blob from the combined KML content

// Create a download link for the Blob

// Trigger a click event to start the download

});

## **Generating KML Content for Placemarks**

The `generatePlacemarksKML` function takes placemarks from the `placemarks` array and generates KML content for each placemark, including coordinates, name, and styling information.

// Function to generate KML content for placemarks

function generatePlacemarksKML() {

// Map placemarks to KML content

// Return KML content for placemarks

}

## **Generating KML Content for Polylines**

The `generatePolylinesKML` function takes polyline coordinates from the `polylineCoordinates` array and generates KML content for the polyline, including coordinates, name, and styling information.

// Function to generate KML content for polylines

function generatePolylinesKML() {

// Generate KML content for the polyline

// Return KML content for polylines

}

## **Conclusion**

The `download.js` script enhances the map application by providing users with the ability to download the map content either as an image or as KML files. The script efficiently uses `html2canvas` for capturing the map screenshot and creates downloadable files with relevant content for both placemarks and polylines.

# **`upload\_kml.js` - Handling KML Upload**

The `upload\_kml.js` script facilitates the uploading of KML (Keyhole Markup Language) files, parsing their contents, and extracting placemarks and polylines. The extracted features are then used to create a new layer, which is added to the `savedLayers` array. Additionally, a button for the saved layer is displayed in the sidebar using the `displaySavedLayerButton` function from `layers.js`.

## **Handling KML Upload**

The script defines a function `handleKMLUpload` responsible for handling the file upload. It adds an event listener to the file input element (`uploadKMLBtn`). When a KML file is selected, the function reads its contents using `FileReader`, parses the KML content, and extracts placemarks and polylines.

// Function to handle file upload and plot KML on the map

function handleKMLUpload() {

const uploadKMLBtn = document.getElementById('uploadKMLBtn');

uploadKMLBtn.addEventListener('change', (event) => {

const file = event.target.files[0];

if (file) {

const reader = new FileReader();

reader.onload = function (e) {

const kmlContent = e.target.result;

// Extract placemarks and polylines from KML content

const { placemarks, polylineCoordinates } = extractFeaturesFromKML(kmlContent);

// Prompt the user for a layer name (using the uploaded file name)

const layerName = file.name.replace(/\.[^/.]+$/, ""); // Remove file extension

// Create a new layer object containing placemarks, polylines, and the layer name

const newLayer = {

name: layerName,

placemarks: [...placemarks],

polylineCoordinates: [...polylineCoordinates],

// Add other properties as needed based on your requirements

};

// Add the new layer to the savedLayers array

savedLayers.push(newLayer);

// Display a button for the saved layer in the sidebar

displaySavedLayerButton(savedLayers.length, layerName);

};

// Read the file as text

reader.readAsText(file);

}

});

}

// Call the function to handle KML upload

handleKMLUpload();

## **Extracting Features from KML**

The script defines a function `extractFeaturesFromKML` responsible for parsing KML content and extracting placemarks and polylines. It uses `DOMParser` to parse the KML content, then extracts placemark and polyline elements. The extracted information is stored in arrays (`placemarks` and `polylineCoordinates`).

// Function to extract placemarks and polylines from KML content

function extractFeaturesFromKML(kmlContent) {

const parser = new DOMParser();

const xmlDoc = parser.parseFromString(kmlContent, 'application/xml');

// Extract placemark and polyline elements from the KML document

const placemarkElements = xmlDoc.getElementsByTagName('Placemark');

const polylineElements = xmlDoc.getElementsByTagName('LineString');

// Create arrays to store extracted placemarks and polylines

const placemarks = [];

const polylineCoordinates = [];

// Extract placemarks

for (let i = 0; i < placemarkElements.length; i++) {

const placemarkElement = placemarkElements[i];

const coordinatesElement = placemarkElement.querySelector('coordinates');

if (coordinatesElement) {

// Splitting coordinates and converting to numeric values

const coordinates = coordinatesElement.textContent.trim().split(',');

const latitude = parseFloat(coordinates[1]);

const longitude = parseFloat(coordinates[0]);

// Creating a placemark object with coordinates and image source

const placemark = {

coordinates: [latitude, longitude],

imageSrc: 'icons/pin-fill.svg', // Image source

// Add other properties as needed based on your requirements

};

placemarks.push(placemark);

}

}

// Extract polylines

for (let i = 0; i < polylineElements.length; i++) {

const polylineElement = polylineElements[i];

const coordinatesElement = polylineElement.querySelector('coordinates');

if (coordinatesElement) {

// Splitting coordinates and mapping to an array of [latitude, longitude]

const coordinates = coordinatesElement.textContent.trim().split(' ');

const polyline = coordinates.map((coord) => {

const [longitude, latitude] = coord.split(',').map(parseFloat);

return [latitude, longitude];

});

polylineCoordinates.push(polyline);

}

}

return { placemarks, polylineCoordinates };

}

## **Conclusion**

The `upload\_kml.js` script extends the functionality of the map application by allowing users to upload KML files. The uploaded content is parsed, placemarks and polylines are extracted, and a new layer is created and added to the `savedLayers` array. The script integrates with the existing application architecture for managing layers and updating the sidebar.

# **`upload\_csv.js` - Handling CSV Upload**

The `upload\_csv.js` script handles the uploading of CSV (Comma-Separated Values) files, parsing their contents using the PapaParse library, and processing the data to create polyline coordinates within specified boundaries. The script then renders the polylines on the map, saves the CSV data as a new layer, and updates the sidebar with a button for the saved layer.

## **Handling CSV Upload**

The script defines an event listener for the file input element (`uploadCSVBtn`). When a CSV file is selected, the script uses the PapaParse library to parse its contents. The parsed data is then processed to extract latitude and longitude values. The script converts these values into polyline coordinates and filters them based on specified boundaries. The filtered coordinates are used to render polylines on the map.

// Accessing the file input element from the DOM

const uploadCSVBtn = document.getElementById('uploadCSVBtn');

// Adding an event listener to the file input element for CSV upload

uploadCSVBtn.addEventListener('change', (event) => {

// Get the selected file from the input

const file = event.target.files[0];

if (file) {

// Parsing the contents of the file using PapaParse library

Papa.parse(file, {

header: true, // Indicates that the first row contains headers

dynamicTyping: true, // Automatically converts appropriate values to numbers or booleans

complete: function (results) {

// Extracting longitude and latitude from the CSV data

const points = results.data.map((row) => ({

latitude: row.latitude, // Latitude value from CSV row

longitude: row.longitude, // Longitude value from CSV row

altitude: row.altitude, // Altitude value from CSV row

distance\_from\_boundarycentre: row.distance\_from\_boundarycentre, // Distance value from CSV row

}));

// Calculating image coordinates based on the map configuration

const mapCoordinates = calculateImageCoordinates(config);

// Converting CSV points to polyline coordinates and filtering within boundaries

const filteredPolylineCoordinates = points

.map((point) => {

// Converting latitude and longitude to image positions

const imagePosition = getImagePosition(

point.latitude,

point.longitude,

mapCoordinates[0],

mapCoordinates[1],

mapCoordinates[2],

mapCoordinates[3]

);

// Checking if the point is within the specified boundaries

if (

imagePosition.image\_X\_position >= 0 &&

imagePosition.image\_X\_position <= 893 &&

imagePosition.image\_Y\_position >= 0 &&

imagePosition.image\_Y\_position <= 893

) {

return [point.latitude, point.longitude]; // Including points within boundaries

} else {

return null; // Excluding points outside boundaries

}

})

.filter((point) => point !== null);

// Clearing existing placemarks and polylineCoordinates arrays

placemarks.length = 0;

polylineCoordinates.length = 0;

// Assigning CSV polyline coordinates within boundaries to the polylineCoordinates array

polylineCoordinates.push(...filteredPolylineCoordinates);

// Rendering the polylines on the map

const coordinates = calculateImageCoordinates(config);

renderPolyline(coordinates, polylineCoordinates, getImagePosition, ctx);

// Saving the CSV data as a layer

const layerName = file.name.replace(/\.[^/.]+$/, ''); // Remove file extension

const newLayer = {

name: layerName,

placemarks: [...placemarks],

polylineCoordinates: [...polylineCoordinates], // Copying corrected polylineCoordinates

// Add other properties as needed based on your requirements

};

// Adding the new layer to the savedLayers array

savedLayers.push(newLayer);

// Displaying a button for the saved layer in the sidebar

displaySavedLayerButton(savedLayers.length, layerName);

// Resetting the value of the file input element to allow uploading the same file again

uploadCSVBtn.value = null;

},

});

}

});

## **Conclusion**

The `upload\_csv.js` script enhances the map application by allowing users to upload CSV files. It uses PapaParse to parse the CSV content, processes the data to create polyline coordinates within specified boundaries, renders the polylines on the map, and saves the CSV data as a new layer. The script seamlessly integrates with the existing application architecture for managing layers and updating the sidebar.

# **`style.css` - Overview**

The `style.css` file contains CSS styles for the Survey App, specifically targeting the `#map-container` and `#canvas` elements. Let's break down the styles defined in this file:

## **Styles**

**1. `#map-container`:**

- Purpose: Applies styles to the container for the map.

- Styles:

- `position: relative;`: The map container is positioned relative to its normal position. This is a common style for container elements.

**2. `#canvas`:**

- Purpose: Applies styles to the canvas element used for drawing on the map.

- Styles:

- `position: absolute;`: The canvas is positioned absolutely within its containing element (`#map-container`). This allows precise placement of the canvas.

- `top: 0;`: Positions the canvas at the top of its containing element.

- `left: 5;`: Positions the canvas with a left offset of 5 units. Note: It seems there might be a missing unit (`px` or `%`). It should be `left: 5px;` to specify pixels.

- `z-index: 1;`: Sets the stacking order of the canvas, ensuring it is above the image (`z-index: 1` puts it above elements with the default `z-index: 0`).

- `pointer-events: none;`: Allows click events to pass through the canvas to the underlying image. This is useful when you want the image to handle click events rather than the canvas.

## **Conclusion**

The styles in `style.css` are focused on the positioning and behavior of the canvas within the map container. The corrections ensure the `left` property is specified with the appropriate unit.