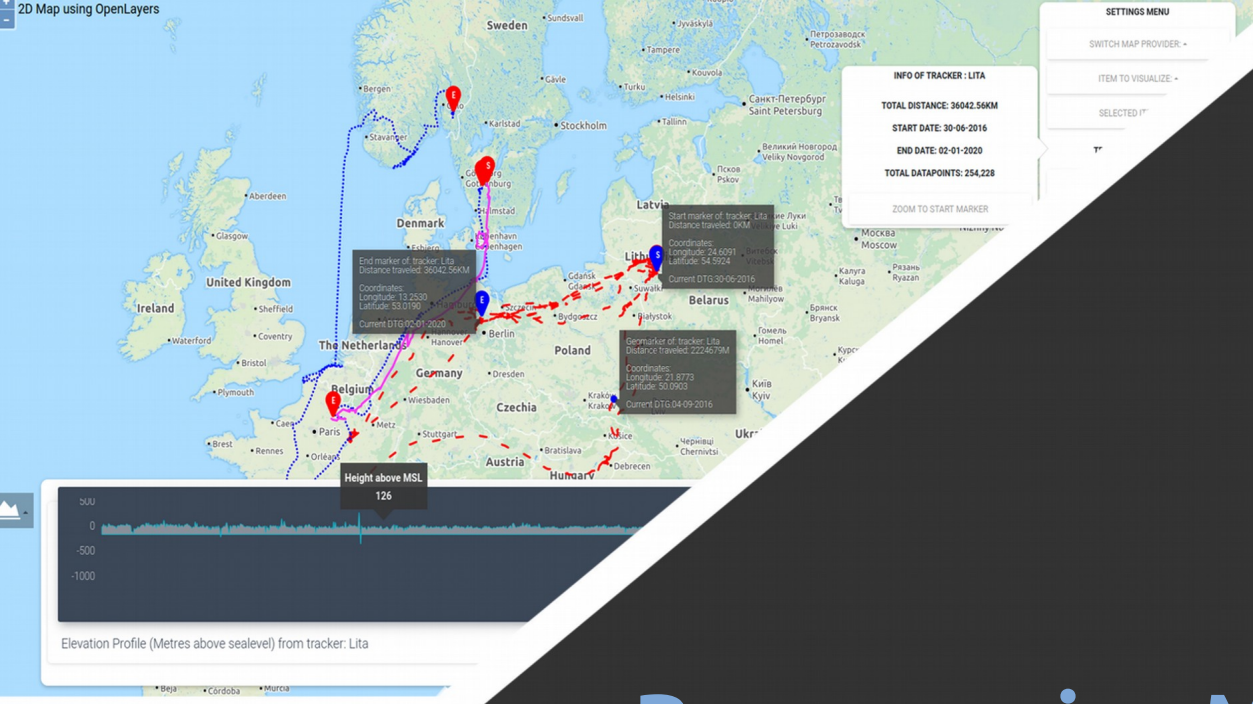
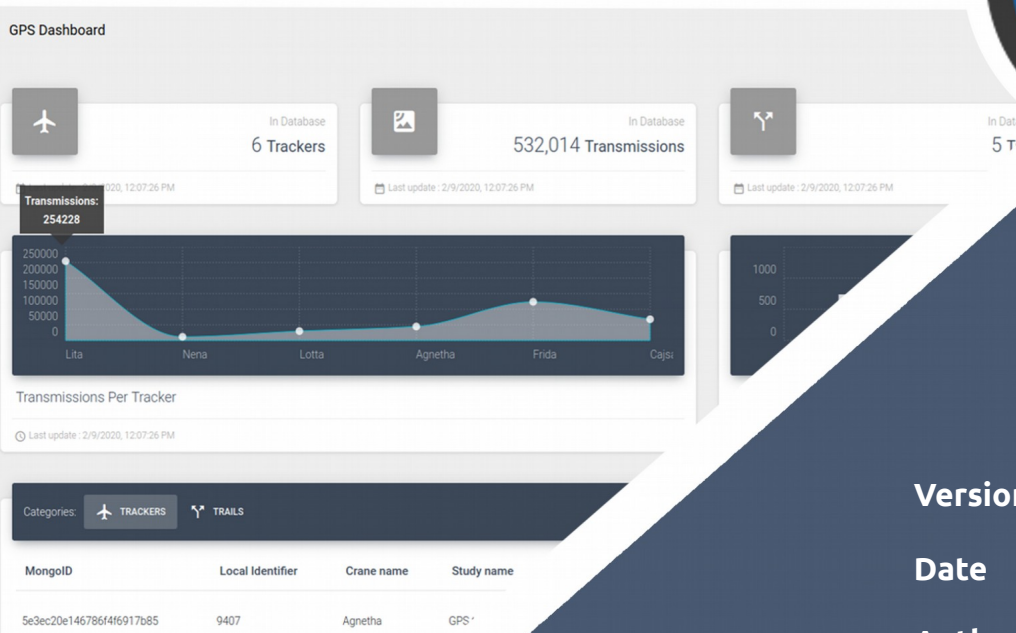


Programming Manual

CREATING AN 2 DIMENSIONAL MAP VIEWER



Version : 1.0

Date : 08-04-2020

Author : The GeoStack Project

Purpose of this document

This programming manual serves as an extension for the following documents:

1) **Cookbook: Creating the GeoStack Course VM:**

   The datastores, tools and libraries used during this programming manual are installed and created in the cookbook: Creating the GeoStack Course VM.

2) **Cookbook: Creating a basic web application:**

   The base application of this Dataset Dashboard has been created during the cookbook: Creating a basic web application.

3) **Cookbook: Data modeling in MongoDB using MongoEngine:**

   The data used during this cookbook, is modeled, indexed and imported in the cookbook: Data modeling in MongoDB using MongoEngine.

4) **Programming manual: Creating the Python-Flask web application:**

   The middleware that will be used during this programming manual is created in the programming manual: Creating the Python Flask web application.

If you have not read these documents yet, please do so before reading this document.

The purpose of this programming manual is to create an 2D map viewer application using the AngularJS JavaScript framework and OpenLayers 6. This application is an extension of our Angular base application.

The Angular apps will perform API calls to our Flask application and our Flask application will then retrieve the requested data via queries, performed on our datastores. The results are then returned to our Angular applications.

This programming manual serves as a guideline for the steps you have to perform to create a 2D Map Viewer using OpenLayers and visualize the data retrieved by the Flask-API.

During this programming manual the code is explained using the inline comments in the source code located in the folder: “POC”. It’s highly recommended to use the source code provided in this folder when creating the web application yourself.

NOTE: Sometimes you will notice that in the code, which you have to create, some functions do not exist yet. Don’t worry about this since they will be added later on during the programming manual!

Table of Contents

1.Introduction

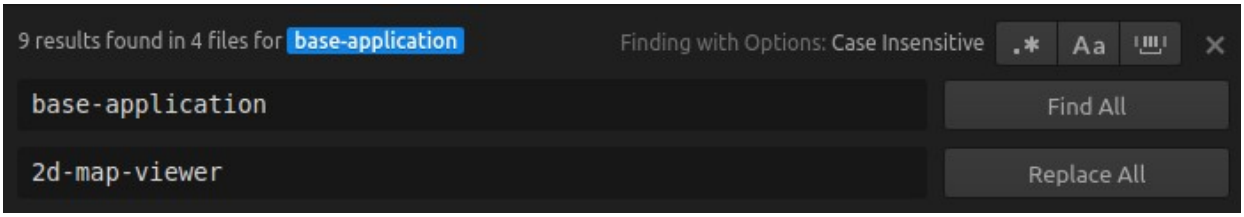
During this chapter we are going to convert the basic web application in such a way that we can start coding the 2D Map Viewer application. We do this because the base application contains the base structure of the 2D Map viewer application which we are going to create during this programming manual. As mention before; if you did not read the programming manual: “Creating a basic web application” you should read it before continuing this programming manual.

1.1 Getting ready

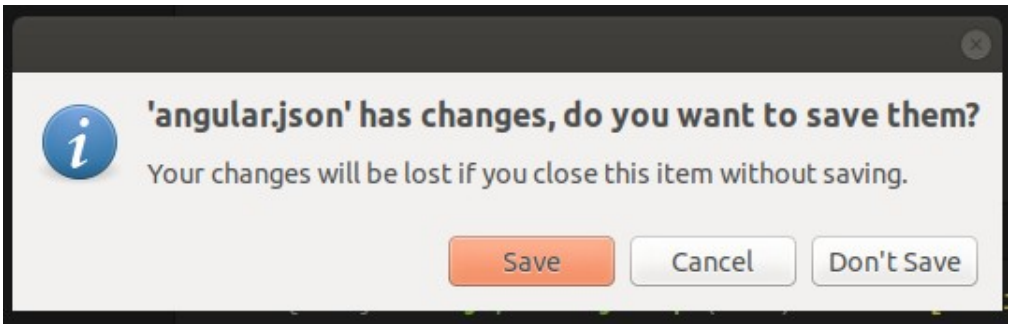
Since this application is an extension of the base-application we can copy this application and start creating the 2D map viewer from there. After we copied the base-application folder, we need to change some names and titles. We start by changing the name of the folder, which was copied, from base-application to 2d-map-viewer, as shown in the image below.

We also need to edit the project name: “base-application” to “2d-map-viewer”. If you are using the code-editor Atom, this is done by performing the following steps:

1) In the editor press the key combination Ctrl + shift + f on your keyboard.
2) In the screen that pops up enter: “base-application” in the find section and “2d-map-viewer” in the replace section, as shown in the illustration below. Then click on find all.



3) Click on replace all and on the save button in the screen that pops up.



4) In the file called: “index.html” which is located in the folder 2d-map-viewer/src, replace the title from BaseApplication to 2D Map Viewer.
5) In the file: sidebar.component.html located in the folder src/app/components/sidebar/, change the text: “Base Application” to “2D Map Viewer.”

1.2 Adding the geospatial framework OpenLayers

Adding a geospatial framework to our Angular application can be done in 2 ways which are as follows:

1) **Installing the NPM Package: "Ol":**

   This is the first technique which you can use to install OpenLayers in your application. During this programming manual we will not be using this technique. If you want to read up on using this technique you should visit the following URL: https://www.npmjs.com/package/ol

2) **Downloading the OpenLayers source code:**

   During this programming manual we are going to use this technique. We do this because, from the version control point of view, this method is the best method since there are no files added to the Node_Modules folder of the application. Using this technique we are going to add the geospatial framework as static files in the assets folder of our application. This enables us to easily switch to a newer or older version of the geospatial framework.

First we want to download the OpenLayers source code from the OpenLayers Github repository which is located on the following URL:

https://github.com/openlayers/openlayers/releases/download/v6.2.1/v6.2.1-dist.zip

When the download is complete you want to extract the folder somewhere, after which you want to create a new folder in the assets folder of our 2d-map-vierwer.

We do this by running the following command:

```
mkdir ~/Geostack/angular-apps/2d-map-viewer/src/assets/geospatial-frameworks
```

Copy the extracted OpenLayers folder to the folder we just created.

Add the following lines inside the <head> element from the index.html file located in the folder 2d-map-viewer/src/app/.

```
<!--Here we add the reference to the OpenLayers style sheet-->
<link rel="stylesheet" href="./assets/geospatial-frameworks/OpenLayers/ol.css"/>

<!--Here we add the reference to the OpenLayers javascript code-->
<script src="./assets/geospatial-frameworks/OpenLayers/ol.js"></script>
```

Now we will be able to use the geospatial framework OpenLayers in our application.

If you want to upgrade to a newer or older version of OpenLayers you can use the same technique as mentioned above but then with the desired version from the OpenLayers Github repository.

2. Creating the services

Now let's start of with creating the services that contain the functions to retrieve data from our datastores. We need 3 services which are a map service in which we are going to add the functions that retrieve the Tilestache entries from our Tileserver and a Crane service which contains all the functions related to performing API calls to our Flask-API to obtain the Crane Tracker data from our MongoDB datastore.

NOTE: In the folder "POC" you can also find the service which is required to perform API calls to our Flask-API to obtain the GPS-Route (Trail) data from our MongoDB datastore. This is not described in this cookbook since it's basically the same as the code for the Crane Trackers.

2.1 Creating the map service

To create the map service we first need to create a new TypeScript file called:"map.service.ts". We do this by running the following command:

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/services/map.service.ts
```

Now let's open this file and import the basic Angular modules at the top of this file by adding the following code:

```
/*
Here we import some basice modules from Angular.
The HttpClient module is required to make requests to our API.
*/
import { Injectable } from '@angular/core'
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs'

@Injectable()
```

Now we want to create the class and it's constructor for the Map service. This is done by adding the following code below the line: "@Injectable()":

```
/*
Here we create a class called MapService. This class will be instantiated
in the MapComponent which we will create later.
*/
export class MapService {

    /*
    Here we the class constructor. We pass the HttpClient and assign it to a
    variable called: "http". If we want to perform HTTP requests we first need
    to call the instance of the HttpClient by using this variable.
    */
    constructor(private http: HttpClient) { }
}
```

Now we want to add a function that is able to retrieve all the entries in our Tilestache Tileserver.

Now we want to add a function that is able to retrieve all the entries in our Tilestache Tileserver. So lets add the function called: "getTilestacheEntries()".

When the function is called in the MapComponent it will perform an HTTP GET request on the Flask-API, which will then activate the function which is bound to the URL: /api/get_tilestache_entries/. The function bound to this URL scrapes all the entries in our Tilestache configuration and returns them as a list. We need these entries to switch between WMS's (Web Map servers)

Adding the function is done by adding the following function in the class: “MapService ”below the constructor:

```
getTilestacheEntries(): Observable<any[]> {
    return this.http.get<any>(`/api/get_tilestache_entries/`)
}
```

That’s all! Now we have created a function that can perform an API call to obtain all the WMS entries in our Tilestache configuration.

2.2 Creating the Crane service

Let’s create the Crane service to which contains all the functions required to retrieve data from our MongoDB datastore. To create the Crane service we first need to create a new TypeScript file called:”crane.service.ts“. We do this by running the following command:

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/services/crane.service.ts
```

Now let’s open this file and import the basic Angular modules at the top of this file by adding the following code:

```
/*
Here we import some basice modules from Angular.
The HttpClient module is required to make requests to our API.
*/
import { Injectable } from '@angular/core'
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs'
```

Now we want to create the class called: ”CraneService” and it’s constructor for this service. This is done by adding the following code below the module imports:

```
@Injectable()
export class CraneService {

    constructor(private http: HttpClient) {}

}
```

This class will be instantiated in the MapComponent which we will create later. This class will contain all the functions which are required to perform API requests to our Flask-API.

The class will contain functions related to requesting Crane data from our MongoDB datastore.

Now we want to add 6 functions to retrieve the Crane Tracker data from our MongoDB datastore, these functions are as follows:

1) getTrackers() to obtain all the trackers in our database.
2) getTracker() to obtain one tracker using the MongoID.
3) getTransmissionsID to obtain all transmissions belonging to a tracker.
4) getTransmissionsAmount to obtain a N amount of transmissions belonging to a tracker.
5) getTransmissionsDTG() to obtain all the transmission in a given timeframe.
6) getTransmissionsCountry() to obtain all the transmissions in a given polygone.

So let's add a function called: "getTrackers()", that is able to retrieve trackers in our MongoDB datastore. This is done by adding the following function in the class: "CraneService " and below the constructor:

```
getTrackers(): Observable < any[] > {
    return this.http.get < any[] > ('/api/trackers/')
};
```

The function called: "getTrackers()", which is used to perform an HTTP GET request to our Flask-API. The function performs a request on the following URL:api/trackers/. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves all trackers from the MongoDB datastore. The function:"getTrackers()" then returns all the trackers to our MapComponent.

Now let's add the function called: "getTracker()", which is used to obtain a specific tracker using it's MongoID, we do this by adding the following function below the function we created above:

```
getTracker(id: string): Observable < any[] > {
    return this.http.get < any > (`/api/trackers/${id}`)
};
```

This function is used to perform an HTTP GET request to our Flask-API. The function performs a request on the following URL: api/trackers/{id}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves a tracker which has the id passed in this function,from the MongoDB datastore. The function:"getTracker()" then returns the tracker to our MapComponent.

Now let's add the function called:"getTransmissionID()" which obtains all the transmissions belonging to a tracker, we do this by adding the following code below the function we created above:

```
getTransmissionsID(id: string): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/transmissions_by_id/${id}`)
};
```

This function is used to perform an HTTP GET request to our Flask-API. The function performs a request on the following URL: api/transmissions_by_id/${id}. This URL is bound to a function in our Flask-API. The function which is bound to this URL executes a query on our MongoDB datastore and retrieves all transmissions belonging to a tracker that has the id passed in this

function. The function:"getTransmissionsID()" then returns the transmissions to our MapComponent.

Now we want to create the function called: “getTransmissionsAmount()” that obtains a N amount of transmissions belonging to a specific tracker. We do this by adding the following code below the function we created above:

```
getTransmissionsAmount(id: string, amount: number): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/transmissions_by_amount/${id}/${amount}`)
};
```

The function performs a request on the following URL: api/transmissions_by_amount/${id}/${amount}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves all transmissions belonging to a tracker that has the id passed in this function. The amount of transmissions it returns is the amount passed in the function call. The function:"getTransmissionsAmount()" then returns the transmissions to our MapComponent.

Now let’s add the function which obtains all the transmissions belonging to a tracker in a given time frame, we do this by adding the following code below the function we created above:

```
getTransmissionsDTG(id: string, dtg_1: string, dtg_2: string): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/transmissions_by_dtg/${id}/${dtg_1}/${dtg_2}`)
};
```

The function performs a request on the following URL: api/transmissions_by_dtg/${id}/${dtg_1}/${dtg_2}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves an N amount of transmissions between the start date (dtg_1) and the end date (dtg_2) belonging to a tracker that has the id passed in this function.

Finally we want to add the function which obtains all the transmissions belonging to a tracker in a given polygon, we do this by adding the following code below the function we created above:

```
getTransmissionsCountry(
    id: string, coords: Number[][]): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/transmissions_in_polygon/${id}/${coords}`)
};
```

The function performs a request on the following URL: api/transmissions_in_polygon/${id}/${coords}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves all transmissions of which the coordinates reside in the list of coordinates passed as parameter in the function, belonging to a tracker that has the id passed in this function.

That’s it! Now we have created the service that contains functions which can perform an API call to obtain data from our MongoDB datastore.

2.3 Creating the Port service

Let's create the Port Service which is going to contain all the functions required to retrieve the World Port Index data from our PostgreSQL datastore. To create the Port Service we first need to create a new TypeScript file called:"port.service.ts". We do this by running the following command:

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/services/port.service.ts
```

Now let's open this file and import the basic Angular modules at the top of this file by adding the following code:

```
/*
Here we import some basice modules from Angular.
The HttpClient module is required to make requests to our API.
*/
import { Injectable } from '@angular/core'
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs'
```

Now we want to create the class and it's constructor for this service we also add the function which is used to perform the API request on the Flask-API. This is done by adding the following code below the module imports:

```
/*
Here we create a class called:"PortService".
The class contains all the functions related to requesting World Port Index data
from our PostgreSQL databe: World_Port_Index_Database via our Flask-API.
*/
@Injectable()
export class PortService {

    // Here we create the class constructor.
    constructor(private http: HttpClient) { }


    /*
    Here we create a function called: "getPorts()", which is used
    to perform an HTTP GET request to our Flask-API.

    The function performs a request on the following URL:
    /api/ports

    This URL is bound to a function in our Flask-API. The function, bound to this
    URL, executes a query on our PostgreSQL database: "World_Port_Index_Database"
    which will the return all the ports in the database.

    The function:"getPorts()" then returns the ports to our
    MapComponent.
    */
    getPorts(): Observable<any[]> {
        return this.http.get<any[]>('/api/ports/')
    };
};
```

4. Creating the 2D Map page

The 2D Map page is going to be the page which contains all the code logic to actually display the data which is obtained from our datastore on a 2D Map from OpenLayers.

To create the map page we first have to add a new folder to our pages folder in our web application. We do this by running the following command:

```
mkdir ~/Geostack/angular-apps/2d-map-viewer/src/app/pages/map-page
```

In that folder we need to create 2 files which are a map.component.ts file and map.component.html file. We do this by running the following commands :

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/pages/map-page/map.component.ts &&
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/pages/map-page/map.component.html
```

Now that we have created the required files we can start creating the basic structure of our map page.

4.1 Creating the Map Component base

We want to start of with importing the Angular modules required to create the map component in the map.component.ts file, so let's open this file and add the following code at the top:

```
/*
Here we import the default angular modules
*/
import { Component, OnInit} from '@angular/core';

/*
Here we import the modules for creating the interactive charts in the map.
We also import a module wich is required to show tooltips in those charts.
*/
import * as Chartist from 'chartist';
import * as tooltip from 'chartist-plugin-tooltips'
```

Next we want to import the map service, crane service and port service we do this by adding the following code to the file:

```
/*
Here we import the services which are used to call functions that perform
API requests to our Flask-API, which will then execute a query on our MongoDB
datastore.
*/
import {MapService} from 'src/app/services/map.service'
import {CraneService} from 'src/app/services/crane.service'
import {PortService} from 'src/app/services/port.service'
```

Now we want to create a global variable which is required to use the OpenLayers code throughout our map component. We do this by adding the following code:

```
/*Here we create a global constant called:"ol".
This constant represents the instance of the geospatial framework OpenLayers.
To use the build in functions of OpenLayers we first need to call this constant*/
declare const ol: any;
```

Now we want to create the component metadata for our map component. We do this by adding the following code:

```
@Component({
  selector: 'app-map',
  templateUrl: './map.component.html',
  providers: [MapService, CraneService, PortService]
})
```

The following applies to this code:

1) selector: If we want to use the map component, we add the code: <app-map/> to the HTML file in which we want to add the component.
2) templateUrl: The HTML file in which we will define the layout of the component.
3) providers: A list of providers (services) in which we have defined the functions required to perform API calls.

Now we want to create the base of our map component. We do this by adding the following code below the metadata:

```
export class MapComponent implements OnInit {
  /*
  Here we create the class constructor of the MapComponent. We pass the map and
  CraneServices in the constructor. We assign the services to a fitting variable,
  this variable can be reused throughout the whole component. We use these
  variables to call the functions in our services which will then perform API
  calls to our Flask-API.
  */
  constructor(private _MapService: MapService,
    private _CraneService: CraneService,
    private _PortService: PortService) {}

  /*
  Here we create the ngOnInit() function. All the logic in this function will
  be executed when the component is loaded.
  */
  ngOnInit() {
  }
}
```

Now we want to add a div element to our map.component.html file. This div element is going to contain our OpenLayers map after we create it. So let's open the html file and add the following code:

```
<!--
Here we create a div element to which the OpenLayers map will be assigned
We give the div a height of 100vh (Full screen height) and a width of 100%
which is the full width of the screen. -->
<div id="map" style="height: 100vh; width: 100%;"></div>
```

Now we want to make the map page available in our 2D map viewer application. We do this by adding the component to our app.module.ts file. So let's open this file and import the map component by adding the following code below the last import:

```
/*Here we import the map component which is our map page and will be added
to the declarations section in this file.*/
import { MapComponent } from '../app/pages/map-page/map.component';
```

Next we want to add the imported component to the declarations section in the app.module.ts file. The final declarations section will look the same as shown in the illustration below:

```
declarations: [
  AppComponent,
  SidebarComponent,
  NavbarComponent,
  BaseComponent,
  MapComponent
],
```

Now we need to add a new route to our app-routing.module.ts file. First we need to import the map component. We do this by adding the following code below the last import in that file:

```
/*Here we import the map component which is required to create a new
Angular route for the map page.*/
import { MapComponent } from '../app/pages/map-page/map.component';
```

Then we need to add the new route to our routes in this file. We do this by changing the current routes to the following:

```
/*
Below we add the route of our base page to the angular routes list. We do this
by adding the following line to the list:
{ path: 'base-page', component: BaseComponent},

Below we also add the route of our base page to the angular routes list.
We do this by adding the following line to the list:
  { path: 'map-page', component: MapComponent}

This means that when we navigate to localhost:4200/map-page, the MapComponent
will be loaded and thus our map-page.

Now we want to make sure that when we navigate to localhost:4200, we are
automatically redirected to the Map page. For this we add the following
line to the routes list:
  { path: '', redirectTo: 'map-page', pathMatch: 'full',},

This means that when we navigate to localhost:4200, we are redirected to value
assigned to the redirectTo variable, which is the map-page in this case.
So when we are redirected to the map-page the MapComponent is shown.*/
const routes: Routes = [
  { path: 'base-page', component: BaseComponent},
  { path: '', redirectTo: 'map-page', pathMatch: 'full',},
  { path: 'map-page', component: MapComponent}
];
```

Now we want to add a new entry to our sidebar. This is done in the file: sidebar.component.ts, located in the folder: src/app/components/sidebar/. So let's open this file and add a new route to our route list. The final code will be the same as shown in the illustration below.

```
/*
Here we create a list which contains all the routes that will be displayed in
our sidebar. The list items inherit the interface: "RouteInfo".

Since we are only going to add the route of our base page, we only need to
create one route. The following applies to this route:
- Path:   The base page component is located on the path:"/base-page"
- title:  The title of the base page is going to be: "Base Page"
- icon:   The entry icon will be the map icon provided by the package:
          "Material Icons". If you want to add other icons you can navigate to
          the following URL:"https://material.io/resources/icons/?style=baseline"
- class:  No extra classes need to be passed in this route.
*/
export const ROUTES: RouteInfo[] = [
    { path: '/map-page', title: '2D Map OpenLayers', icon: 'map', class: '' },
    { path: '/base-page', title: 'Base Page', icon: 'map', class: '' },
];
```

Now when we start our application we will be greeted with the the new page as shown in the illustration below:

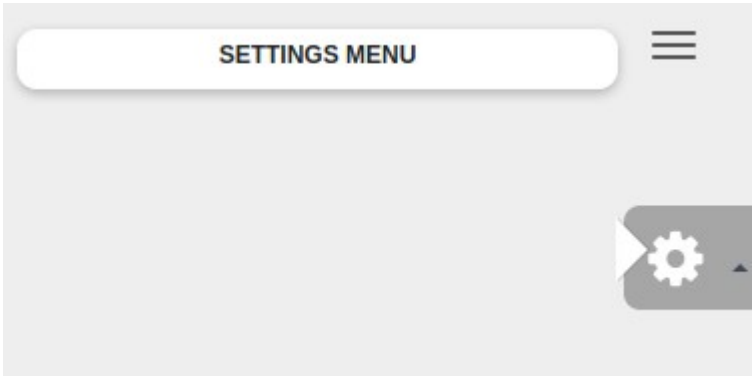
The page does not have any content yet so we are going to add this now.

4.1.1 Creating the base of the settings menu

We want to start of by adding the base of the settings menu to our map.component.html file. We do this by adding the following code, below the div element related to the map, to this file:

```
<!--
In this div element all the logic related to the map settings is added.
The class: "fixed-plugin" make sure the settings menu is shown on the web page.
-->
<div class="fixed-plugin" id="fixed-plugin">
    <!--
    Here we add the dropdown. We set autoClose to false to make sure the menu
    does not close when a setting is selected.
    -->
    <div id="main-dropdown" ngbDropdown [autoClose]="false">
        <!--
        Here we add the togglebutton. We give the button a cog icon.
        -->
        <a ngbDropdownToggle>
            <i class="fa fa-cog fa-2x"></i>
        </a>
        <!--
        Here we add the menu that opens when the dropdown toggle is clicked.
        We add the title settings menu.
        -->
        <div id="main-dropdown-menu" ngbDropdownMenu>
            <li class="header-title"> Settings menu</li>
            <!--
            In this div element we are going to add all the types of Settings
            from our application.
            -->
            <div id="settings">
            </div>
        </div>
    </div>
</div>
```

Now when we reload the application a drop down button will be displayed as shown in the illustration below:



When we you click the button an empty drop down menu will pop up. Let's add some extra functionalities to our application to make sure the drop-down menu will be filled with items. We are going to do this in the next section.

4.2 Creating the Map Component functionalities

In this section we are going to add all the code logic to our application. We want to start of by creating the OpenLayers map instance in which we are going to create all our layers later.

4.2.1 Creating the OpenLayers Map instance

We want to start off by creating the OpenLayers map, we do this in the file: “map.component.ts” First we need to define a global variable called map. We do this by adding the following code below the line where we defined the class name: MapComponent:

```
export class MapComponent implements OnInit {
  /*
  Here we create a global variable called: "map". This is the variable
  to which the OpenLayers map will be assigned after it's created. Because
  of the global variable we can use the map throughout the whole component.
  */
  public map: any;
```

Note: In the illustration above the line where we defined the class name is also included (“export class MapComponent….”). You don’t need to add this line again.

Next we have to create a global JavaScriptMap which is going to be populated with all the entries from our Tileserver after the function is called related to retrieving the Tileserver entries. We do this by adding the following code below the code we added above:

```
public mapProviders: Map < any, any > = new Map();
```

We created a global variable called: "mapProviders". The variable is a JavaScriptMap which will contain key | values. The JavaScriptMap map will be populated with available map providers once the function:"getMapProviders()" is triggered. We will create this function later!

Next we need to create a global variable which is going to contain the base layer of the OpenLayers map. We do this by adding the following code below the code we added above:

```
  private mapLayer: any = new ol.layer.Tile({
    source: new ol.source.XYZ({
      url: "http://localhost/tiles/openstreetmap-local/{z}/{x}/{y}.png"
    }),
    zIndex: 1
  });
```

Above we created a global variable called: "mapLayer". This is the variable to which the base layer of the map will be assigned. We set the default map (The map that will be shown when the component is loaded) to the local OpenStreetMap map. We assign the URL on which the Local OpenSteetMap tiles are available served from the Tilestache Tileserver running behind the NGINX webserver. We set the zIndex to 1 to make sure the mapLayer is the lowest layer on the map and other layers will be displayed on the mapLayer.

Next we want to create a global variable which is going to contain the OpenSeaMap layer. We do this by adding the following code below the global variable: “mapLayer”:

```
private seaLayer: any = new ol.layer.Tile({
  source: new ol.source.XYZ({
    url: "http://localhost/tiles/openseamap-local/{z}/{x}/{y}.png"
  }),
  zIndex: 2
});
```

We created a global variable called: "seaLayer" to which we assign a Tile layer to which we assign the URL on which the Local OpenSeaMap tiles are available served from the Tilestache Tileserver running behind the NGINX webserver. We set the zIndex of this layer to 2 to make sure it is displayed on top of the mapLayer (the base layer containing the OpenStreetMap Tiles)

Next we want to create a function that calls the function: “getTilestacheEntries()” which we defined in our mapservice.ts file. We do this by adding the following code below the ngOnInit() function:

```
getMapProviders(): void {
  this._MapService.getTilestacheEntries().subscribe(
    (providers: []) => (
      providers.forEach(provider => {
        provider != "" ? this.mapProviders.set(provider, provider) : null;
      })
    )
  )
};
```

The function we created above is used to retrieve all the WMS entries in our Tilestache configuration.This function triggers the function getTilestacheEntries() in the MapService. Then it populates the mapProviders JavaScriptMap with all the obtained entries. If an entry is empty (equal to ""), the entry will not be added to the JavaScriptMap.

Now we want to create the function which actually creates an OpenLayers Map instance when it’s triggered. This function will be called:”createOpenLayersMap()”. In the function we create a new OpenLayers View (Map) and assign the base layer, which was created earlier, to the View. The map will be assigned to the HTML div element with the id: "map". This is the div element in the layout of the MapComponent (map.component.html).

The code required to create this function is shown on the next page.

So let's add the createOpenLayersMap() function below the function: "getmapProviders()" by adding the following code:

```
createOpenLayersMap(): void {

  /*Here we trigger the function:"getMapProviders()" which retrieves the
  entries in our Tilestache server and populates the JavaScriptMap:
  "mapProviders" with the retrieved entries.*/
  this.getMapProviders()

  // Here we create the settings that the map is going to have.
  let mapViewSettings = new ol.View({
    maxZoom: 17,
    center: [0, 0],
    zoom: 3
  });

  /*Here we create a new instance of an OpenLayers map.
  We add the settings as value of the view and the base layer which was
  assigned to the global variable:"mapLayer" as first layer.*/
  this.map = new ol.Map({
    target: 'map',
    view: mapViewSettings,
    layers: [this.mapLayer]
  });
};
```

Now we want to add the function we just created to the ngOnInit() function so that it's triggered when the map component is loaded. The function will look the same as shown in the illustration below after you added the function: "createOpenLayersMap()".

```
/*
Here we create the ngOnInit() function. All the logic in this function will
be executed when the component is loaded.
*/
ngOnInit(){
  this.createOpenLayersMap();
}
```

Now when we start the Tileserver and Flask-API and reload the application you will be greeted with a map that shows the local OpenStreetMap map as baselayer in the OpenLayers View as shown in the illustration below:

4.2.2 Switching between map providers (WMS)

Now we want to add the functionality for us to be able to switch between map providers. For this we first need to create a new function in the map.component.ts file. We do this by adding the following code below the function: “createOpenLayersMap()”:

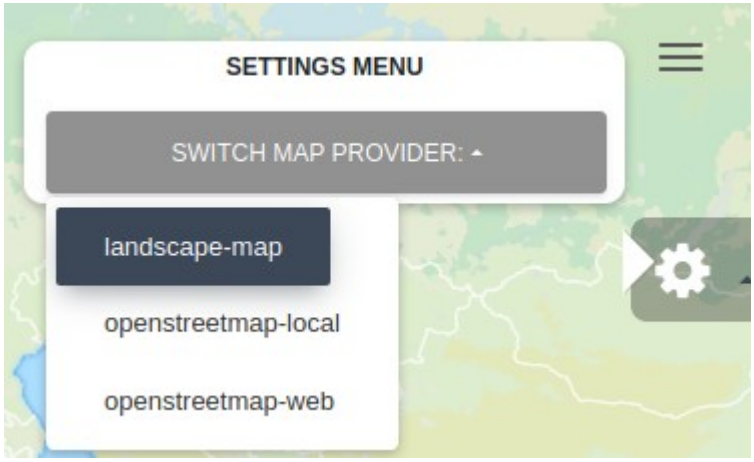
```
setMapProvider(providerKey): void {
  this.mapLayer.getSource().setUrl(
    "http://localhost/tiles/" + providerKey + "/{z}/{x}/{y}.png"
  )
};
```

Above we created the function that changes the mapProvider. When the function is triggered a providerKey is passed. This providerKey is the key of the entry in the JavaScriptMap: "mapProviders". We then call: “getSource” on the mapLayer to obtain the source of the layer. Then we set the URL, on which the map (with the provider key which for example could the landscape-map) is available, using “setURL()”. This function is assigned to the WMSSelection settings menu which we are going to create now!

So let’s add some code to the settings menu in our map.component.html file. Open this file and add the following code in the div element with the id:”settings”:

```
<!--
  In this div element we add the logic for switching between map
  providers.
  -->
  <div id="WMSSelection">
      <!--
      Here we add the dropdown. We set autoClose to false to make sure the menu
      does not close when a setting is clicked.
      -->
      <div ngbDropdown [autoClose]="false">
          <!--
          Here we add the dropdown toggle button and give it some text.
          -->
          <button class="btn btn-white btn-block"
           ngbDropdownToggle>Switch Map Provider:</button>
           <!--
           Here we add the menu that opens when the dropdown button is clicked.
           -->
          <div ngbDropdownMenu>
              <!--
              Here we add a ng-container that contains a FORloop that displays all
              the entries in the JavaScriptMap:"mapProviders" as key values.

              For each of the entries a button will be created which when clicked
              triggers the function: "setMapProvider()" and passes the key of
              the clicked entry as parameter.
              -->
              <ng-container *ngFor="let map of mapProviders | keyvalue">
                  <button ngbDropdownItem (click)='setMapProvider(map.key)'>{{map.key}}
                  </button>
              </ng-container>
          </div>
      </div>
  </div>
```

Now when we reload the page we get a new option in our settings menu which if we click shows all the entries in our Tileserver. If we click on an entry the map will change according to the entry (providerkey) that was selected.



In the next section we are going to add the functionality which is required to select and add items to our application. Each (Crane) Tracker and a GPS-Route (Trail) can be seen as one item.

4.2.3 Adding items from the datastore to our application

Now we want to create the functionality for adding items (Crane Trackers or GPS-Routes/Trails) which are obtained from our MongoDB datastore to the web application.

For this we are first going to create a class called:"Item" which serves as a template for each item that is added. This means that each item is going to contain a specific set of attributes. When loading the items we are going to create a new instance of the item class for each Tracker or Trail in our database.

We are going to add the class below the global constant: "ol" which we defined at the beginning of this document. To create this class we are going to add the following code:

```
export class Item {
  //id: The MongoID of the item
  id: string;
  //name: The name of the item
  name: string;
  //type: The item type (e.g. tracker or trail)
  type: string;
  /*timestampColumn: The name of the timestamp column in the dataset belonging
                     to the item.*/
  timestampColumn: any;
  //totalDataLength: The total amount of transmissions or signals
  totalDataLength: number;
  //totalRouteDistance: The total length of the route
  totalRouteDistance: any;
  //layerGroups: A JavaScriptMap consisting of layerGroups each containing a
  //lineLayer, markerLayer and pointLayer.
  layerGroups: Map < string, any > = new Map();
  //activeLayerGroup: The active Layer group
  activeLayerGroup: any;
  //coordinateList: A list of coordinates
  coordinateList: any = [];
  //altitudeList: A list of altitudes
  altitudeList: any = [];
  //datetimeList: A list of DTG
  datetimeList: any = [];
  //routeDistanceList: A list of distances between datapoints
  routeDistanceList: any = [0];
  //startCoordinate: The first coordinate of the item
  startCoordinate: any;
  //endCoordinate: The last coordinate of the item
  endCoordinate: any;
  /*currentCoordinateIndex: The index in the coordinate list. This value is
                            incremented when the animation is running.*/
  currentCoordinateIndex: number = 0;
  //animation: The instance of the interval which runs the animation
  animation: any;
  //dateRangeTotal: The start/ end date of the total route
  dateRangeTotal: any;
  //dateRangeSelected: The start / end date of the route that is visualized
  dateRangeSelected: any;
};
```

Now that we have created the Item class we need to add a global variable called: "Items" which starts of as an empty list of items.

We create this variable by adding the following code below the global variable: seaLayer which we created earlier.

```
public items: Item[] = [];
```

Above we created a global variable called: "items". The type of the variable is a list of Items. This list starts of empty, but when we call the function(s) that retrieve the Crane Trackers and GPS-Routes from the datastore, the empty list will be populated with these results.

We want to start of by creating a function called:"addItem()". This function will create a new instance from the Item class, which we created earlier, for each item that is obtained from our MongoDB datastore. We add this function by adding the following code below the function "setMapProvider()":

```
addItem(itemId, itemName, itemType, itemRouteLength, itemTimeColumn, itemDTG): void {

  let item = new Item();

  item.id = itemId;

  item.name = itemName;

  item.type = itemType;

  item.totalDataLength = itemRouteLength;

  item.dateRangeTotal = itemDTG;

  item.timestampColumn = itemTimeColumn;

  this.items.push(item);
};
```

Above we created a function called: "addItem()". This function is called on each item retrieved in the getItems() function which we will be creating next.

The function: addItem() then creates a new item and assigns the parameters passed in the function to the newly created item. The parameters are as follows:

- ➔ itemId : The MongoID of the item that is added.
- ➔ itemName : The name of the item that is added.
- ➔ itemType : The itemType, this is necessary because we want to be able to visualize multiple types of datasets such as trackers and trails.
- ➔ itemRouteLength : The total amount of transmissions / signals belonging to the Tracker / Trail that is added.
- ➔ itemTimeColumn : The name of the timestamp column in the dataset. This is required since the column name representing the timestamp differs in each dataset.
- ➔ itemDTG : The TOTAL time frame of the route. This is NOT the time frame of the amount of transmissions / signals which are selected. This value is used to set a begin and end date in the calendar used to select a DTG (Date time group). The value passed as parameter is a line of the start and end date of the route.

When all the values are assigned to the item, the item is added to the global variable: items, which is a list of all items (trackers and trails) located in the MongoDB datastore. The global variable Items populates the list of items that can be selected in the application. We will be adding this list later on. Let's first add the function: "getItems()" which triggers the function:"getTrackers()" which was defined in our CraneService.ts file. We do this by adding the following code below the addItem function:

```
getItems(): void {
  this._CraneService.getTrackers().subscribe(
    (trackers: []) => (
      trackers.forEach(tracker => {
        this.addItem(
          tracker['_id']['$oid'], tracker['name'], 'tracker',
          tracker['transmission_Count'], 'timestamp',
          [tracker['start_date']['$date'], tracker['end_date']['$date']],
        );
      })
    )
  );
};
```

Above we cceated the function called: "getItems()".

This function triggers the function: "getTrackers()" in our CraneService file, which then returns all the trackers in our MongoDB datastore. After the trackers are obtained the function:"addItem()" is called on each tracker.

The syntax used in the function is as follows:

this.{service}.{function}.subscribe({elements} => {elements}.forEach({element} =>{ addItem(element values) }););

The following applies to the syntax above:

- ➔ service = the service which contains the API call functions
- ➔ function = the function from the service you want to trigger. This function will then return the data retrieved from our datastore.
- ➔ elements = this name can be generic. This value stands for the list of data returned by the function. A foreach loop is performed on the list of data because we want to add all rows in the elements to the JavascriptMap it belongs to.
- ➔ element = this name can also be generic. This value stands for 1 row in the data returned by the function. For each element we trigger the function: "addItem()". We pass the required values as parameters in the function: "addItem()"

As mentioned before the function:"getItems" needs to be triggered when the MapComponent loads. So we need to add the function:"getItems()" to our ngOnInit() function so that the items are obtained from our datastore when the map component is loaded.

The final ngOnInit() function will look the same as shown in the image below:

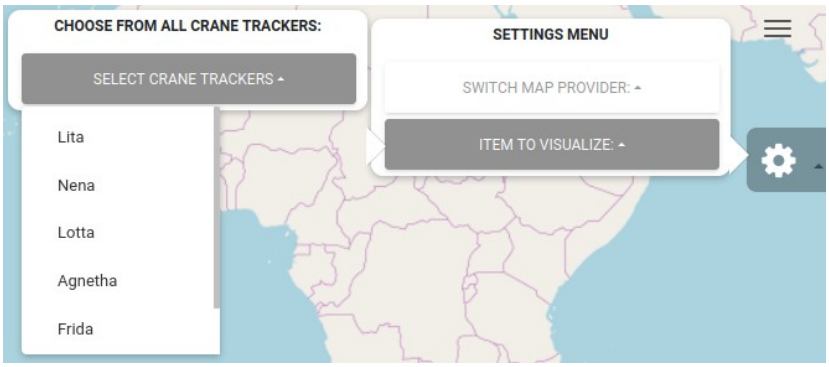
```
ngOnInit() {
  this.createOpenLayersMap();
  this.getItems();
};
```

At this point we cannot select any items yet. To solve this we need to add another drop-down menu to our map.component.html file. We add this html code in the div element with the id:"settings" and below the div tag with the id: "WMSSelection".

```
<div id="itemSelection">
    <!--
    Here we add the dropdown. We set autoClose to false to make sure the menu
    does not close when a setting is clicked.
    -->
    <div ngbDropdown id="main-dropdown" [autoClose]="false">
        <!--
        Here we add the dropdown toggle button and give it some text.
        -->
        <button class="btn btn-white btn-block" ngbDropdownToggle>Item to visualize:</button>
        <!--
        Here we add the menu that opens when the dropdown button is clicked.
        -->
        <div id="main-dropdown-menu" ngbDropdownMenu>
            <!--
            Here we add the dropdown box which contains all the trackers from our database
            -->
            <li class="header-title">Choose from all Crane trackers:</li>
            <div ngbDropdown>
                <!--
                Here we add the dropdown toggle button for out tracker selection dropdown
                -->
                <button class="btn btn-white btn-block" ngbDropdownToggle>Select Crane trackers</button>
                <div ngbDropdownMenu>
                    <!--
                    Here we add a ng-container that contains a FORloop that displays all
                    the objects in our items list which is defined in the component.ts file

                    For each of the entries a button will be created which when clicked
                    triggers the function: "selectItem()" and passes the item which is selected as
                    parameter.
                    -->
                    <ng-container *ngFor="let item of items">
                        <!--
                        Here we check if the item type is equal to tracker. We do this because
                        when we add other types of datasets to our application we dont want them
                        to be in the dropdown box for our trackers.
                        -->
                        <div *ngIf="item.type == 'tracker'">
                            <button ngbDropdownItem (click)='selectItem(item);'>{{item.name}}</button>
                        </div>
                    </ng-container>
                </div>
            </div>
        </div>
    </div>
</div>
```

Now when we reload the web application click on the new drop-down menu in our settings menu we will see all the trackers in our MongoDB database, as shown in the illustration below.

NOTE: Make sure your Tileserver and Flask-API are running otherwise the items cannot be obtained form the datastore!



When we select an item from the drop-down list nothing will happen this is because we assigned a function called:"selectItem()" to each entry in the drop-down menu. To fix this we need to add some extra functionalities to the map.component.ts file.

4.2.4 Selecting items

Before we are going to create the item selection functionality we first need to create a function which is used to convert timestamps to a valid and human readable format. We do this by adding the following code in the map.component.ts file below the getItems() function which we created earlier:

```
timeConverter(timestamp): string {

  // First we create a new Date using the timestamp passed as parameter.
  // We assing the new date to a variable called: "a".
  let a = new Date(timestamp);

  // Here we obtain the year of the timestamp passed as parameter in this
  // function.
  let year = a.getFullYear();
  // Here we obtain the month of the timestamp passed as parameter in this
  // function.
  let month = ('0' + (a.getMonth()+1).toString()).slice(-2);
  // Here we obtain the day of the timestamp passed as parameter in this
  // function.
  let day = ('0' + a.getDate().toString()).slice(-2);

  // Here we add a fix to make sure that when a day or month is equal to
  // 0, it will be set to 1.
  day == '00' ? day = '01' : null;
  month == '00' ? month = '01' : null;

  // Here we create a string by combining the day, month and year.
  let time = day + '-' + month + '-' + year;

  // Here we return the valid datetime as string.
  return time;
};
```

Now to be able to actually select items we first need to add a new global variable called: ‘selectedItems”. We do this by adding the following code below the global variable:”items” which we defined earlier:

```
public selectedItems: Item[] = [];
```

Above we created a global variable called: "selectedItems". The type of the variable is a list of Items. This list starts of empty, but when we select an item from the drop-down menu in the application the function: "selectItem()" will be triggered which will then add the selected item to the selectedItems list. We are going to create the function “selectItem()” later. First we need to create a new global variable called:”activeItem”. We do this by adding the following code below the selectedItems global variable which we created above:

```
public activeItem: Item = new Item();
```

Above we created a global variable called: "activeItem". When an item is selected using the function: "selectItem()" it will become the activeItem. So let now let’s create the function selectItem() which will be triggered when an item is selected from the drop-down list in the application.

We do this by adding the following code below the function:”timeConverter()”:

```
selectItem(item: Item): void {

  // Here we set the global variable activeItem to be the selected item.
  this.activeItem = item;

  // Here we check if the item which is selected has been selected before.
  this.selectedItems.filter(
      data => data.id.includes(item.id)).length == 1 ? null :
    (this.getInitalItemData(item), this.selectedItems.push(item))
};
```

Above we created a function called: "selectItem".This function is triggered when one of the items in the ItemList is clicked using the drop-down menu in the application. The item that is clicked is then passed as parameter in this function.

When the function is triggered the follow steps are performed:

1) The selected item becomes the activeItem.
2) The function: ".filter()" is executed on the global JavascriptMap: "selectedItems".

   This JavascriptMap contains all the items that are have been selected.The filter function is used to check whether the id (From the item that is being selected) is already in the list assigned to the global variable: "selectedItems". If this is the case nothing will happen since the Item was already selected.

   If the itemId, of the item that is being selected, is not in the list of selectedItems the following will happen:

   1) The function: "getInitalItemData()" will be triggered. In the function the item will be passed. This function will retrieve the first 100 transmissions / signals belonging to that item.
   2) The item is added to the selectedItems list.

As you can see we used the function: “getInitialItemData()” which has not been defined yet. So let’s do this now by adding the following code below the function:”selectItem()”:

```
getInitalItemData(item: Item): void {

  switch (item.type) {
    case 'tracker':
      this._CraneService.getTransmissionsID(item.id).subscribe(
        (transmissions) => {
          this.loadItemData(transmissions)
        }
      );
      break;
    default:
      break;
  }
};
```

Above we created a function called: "getInitalItemData()" This function is called in the function: "selectItem()" IF the item that is being selected has not been selected yet (so is not in the selectedItems list). The item from which the data has to be retrieved is then passed as parameter in this function.

This function contains a switch/case. The switch case takes the itemType, which in our case can be a tracker or a trail, as input. Depending on the itemType the corresponding function is in the services is triggered.

The reason we add a switch/case in the function is because when we want other types of datasets we can easily create new cases and add the code, related to triggering a function in the service for performing API calls, to those cases.

The data obtained from the function which is triggered in the service, will then be passed as parameter in the function: "loadItemData()". The function loadItemData() will then assign the returned data to the item which was selected in the function: "selectItem()". We did not create the function: “loadItemData()” yet so let’s do this now.

The function loadItemData() has not been created yet, we are going to do this in the next chapter. But first we want to add another drop-down menu to our settings menu in the map.component.html file.

First we want to create a div element which is only shown in our web application when the amount of selected items is bigger than 0.

We do this by adding the following code below the div element with the id: “itemSelection”.

```
<!--
All the logic in the following div element will only be displayed when
one or more items are selected and thus the size of the selectedItems list
is bigger than 0
-->
<div *ngIf="selectedItems.length > 0">

</div>
```

The rest of the settings we are going to add to our 2D map viewer will be added in this div element. In the div element we want to add a new drop-down box which will contain all the items that have been selected. We do this by adding the following code:

```
<div id="selectedItemsSelection">
    <div ngbDropdown>
      <!--
      Here we create the dropdown toggle button and add the length of the selectedItems list
      as text of the button. We do this by using the syntax {{selectedItems.length}}.
      -->
        <button class="btn btn-white btn-block"
        ngbDropdownToggle>Selected items:{{selectedItems.length}} </button>

        <!--
        Here we create the dropdown menu and set the width of the menu to: "max-content" this is
        done to fix the styling of the remove button next to the selected item entries.
        -->
        <div ngbDropdownMenu style="width:max-content;">
          <!--
          Here we create a ng-container which contains a FORloop that creates a button
          for all of the entries in our selectedItems list.

          We add the function:"selectItem()" to the buttons in which we pass the item that
          is clicked as parameter.

          The text of the button will be the name of the item. we do this by adding the
          syntax: {{item.name}}

          We also create a button next to each item which when clicked triggers the function
          removeItem() and passes the item as parameter.
          -->
            <ng-container *ngFor="let item of selectedItems">
                <div>
                    <button class="pull-left" ngbDropdownItem
                    (click)='selectItem(item);'>{{item.name}}</button>
                    <a class="material-icons pull-left" style="color: red; margin-top: 9px;"
                    (click)='removeItem(item)'>not_interested</a>
                </div>
            </ng-container>
        </div>
    </div>
</div>
```

The function: “removeItem()” has not been created yet. We are going to do this later. First we want to create a function which loads the item data, which is retrieved from the MongoDB datastore, when an item is selected. We are going to do this in the next section.

4.2.5 Loading Item data

To load the item data we first need to create a function called: “loadItemData()”. We are going to create this function by adding the following code below the getInitialItemData() function:

```
loadItemData(data: any[]): void {
  // Here we assign the activeItem to a variable called item
  let item = this.activeItem;

  // Perform a check to see if the data passed as parameter is bigger than
  // 0. If this is the case, no item data will be loaded.
  if (data.length == 0){
    return;
  }

  // Here we create empty lists to which we are going to append the data.
  item.coordinateList = [];
  item.altitudeList = [];
  item.datetimeList = [];

  // Here we create a foreach loop which loops through all the rows in the data
  data.forEach(row => {

    // Here we add the transformed coordinates to the coordinate list.
    item.coordinateList.push(
      ol.proj.fromLonLat(row.geometry.coord.coordinates)
    );

    // Here we add the altitude values to the altitude list.
    item.altitudeList.push(row.geometry.alt);

    // Here we add the DTG values to the datetimeList we pass the value of
    // the item.timestampColumn to obtain the value of this column.
    item.datetimeList.push(
      this.timeConverter(row[item.timestampColumn].$date)
    );
  });

  // Here we add the first entry in the coordinateList as startCoordinate
  item.startCoordinate = item.coordinateList[0];

  // Here we add the last entry in the coordinateList as endCoordinate
  item.endCoordinate = item.coordinateList[data.length - 1];

  // Here we set the last and the first values of the timestamp columns as
  // start en end date of the selected route.
  item.dateRangeSelected = (
    this.timeConverter(data[0][item.timestampColumn]['$date']) + '/' +
    this.timeConverter(data[data.length - 1][item.timestampColumn]['$date'])
  );

  // Here we create a new layerGroup and add the item as parameter.
  this.addLayerGroup(item);
};
```

Above we created the function called: "loadItemData()".This function is called in the following functions which we are going to add later:

- ✔ getInitalItemData()
- ✔ getItemDataByDTG()
- ✔ getItemDataByAmount()
- ✔ getItemDataByCountry()

Each of these functions obtain data from the MongoDB datastore and pass the returned data to the function "loadItemData()" as parameter.

The function then does the following:

1) Assign the activeItem to a variable called: "item". This is done so we only need to use the variable item instead of code: "this.activeItem".
2) Check whether the data passed as parameter is not empty. If the data is empty the function will return because there is no data to be loaded.
3) Set the value of the coordinateList, belonging to the item, to an empty list.
4) Set the value of the altitudeList, belonging to the item, to an empty list.
5) Set the value of the datetimeList, belonging to the item, to an empty list.
6) Execute a forEach loop on the ItemList, the foreach loop does the following for all the rows in the list of data:
    1) Obtain the value of the coordinates and transform them to a format which can be used with OpenLayers. For this we use the the syntax: "ol.proj.fromLonLat()", in which we pass the value of the coordinate column as parameter. After the coordinate has been transformed it is added to the coordinateList belonging to the item.
    2) Obtain the value of the altitude column and append it to the altitudeList belonging to the item.
    3) Obtain the value of the timestamp column and append it to the datetimeList belonging to the item.
7) Assign the first value of the coordinateList (the value at index 0) to the variable: "startCoordinate".
8) Assign the last value of the coordinateList (the value on the index (length of the list of data passed as parameter – 1)) to the variable endCoordinate.
9) Create a list containing the first item in the datetimeList and the last item of the datetimeList, created in step 5.3, and assign it to the variable: "dateRangeSelected".
10) Trigger the function: "addLayerGroup()", and pass the activeItem as parameter. The function: "addLayerGroup()" wil then create the first layerGroup.

The function addLayerGroup() has not been defined yet. We are going to do this in the next section. First we want to create a new drop-down menu in our HTML page which is used to display info related to the selected route.

For this we going to add the following code below the div element with the id: “selectedItemsSelection”:

```
<div id="itemInfoSelection" style="margin-bottom:30px;">
    <!--
    Here we add the dropdown. We set autoClose to false to make sure the menu
    does not close when a setting is selected.
    -->
    <div ngbDropdown id="main-dropdown"  [autoClose]="false">
        <!--
        Here we add a list item which contains the name and type of the activeItem as
        title and an icon as dropdown toggle.
        -->
        <li class="header-title" >{{activeItem.type}}: {{activeItem.name}}
            <button ngbDropdownToggle class="btn btn-white btn-round btn-just-icon info">
                <i class="material-icons">info</i>
            </button>
        </li>
        <!--
        Here we add the menu that opens when the dropdown button is clicked.
        In this menu we add list items that each represent the data belonging to the activeItem.
        -->
        <div id="main-dropdown-menu" ngbDropdownMenu>
            <li class="header-title" style="margin-bottom: 10px;">
              INFO of {{activeItem.type}} : {{activeItem.name}} </li>
            <li class="header-title">Total distance: {{activeItem.totalRouteDistance}}KM</li>
            <!--
            Here we convert the total date range of the route using the timeConverter function.
            We grab the value at index 0 (of the dateRangeTotal) for the start date and the value at
            index 1 for the end date of the total route.
            -->
            <li class="header-title">Start date: {{timeConverter(activeItem.dateRangeTotal[0])}}</li>
            <li class="header-title">End date: {{timeConverter(activeItem.dateRangeTotal[1])}}</li>
            <!--
            Here we add the total amount of datapoints belonging to the selected item.
            We make sure that the number is human readable and uses commas by adding the syntax"
            "|number:'2.'" behind the value the value of the totalDataLength.
            -->
            <li class="header-title" style="margin-bottom: 10px;">
              Total datapoints: {{activeItem.totalDataLength| number: '2.'}} </li>
            <!--
            Here we add a button which has the function: "zoomToLocation()" bound to it.
            When this button is clicked the map will move to the startCoordinate of the selected item.
            -->
            <button class="btn btn-white btn-block" (click)='zoomToLocation()'>
              Zoom to start marker</button>
        </div>
    </div>
</div>
```

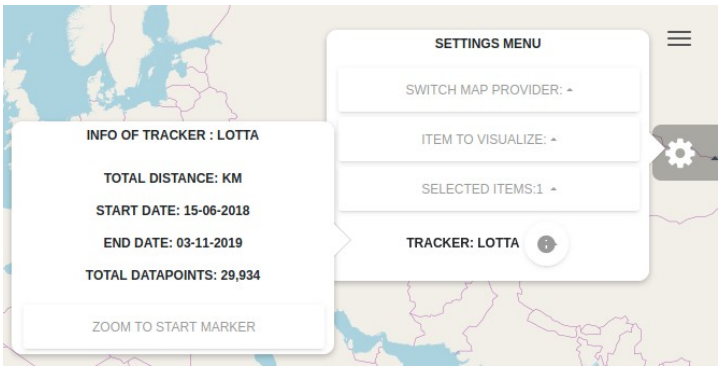
The function: “zoomToLocation()” has not been created yet. So let’s create this function by adding the following code below the function:”loadItemData()” in the map.component.ts file:

```
zoomToLocation(): void {
  /*
  The code below is used for the animation that moves to the start
  coordinates of the activeItem.
  The view of the OpenLayers Map instance is obtained using the syntax:
  ".getView()" on the map instance. Then we animate the view by calling the
  function:".animate()"
  */
  this.map.getView().animate({
    center: this.activeItem.startCoordinate,
    duration: 1500
  });

  /*
  The code below is used for the animation zooms in and out while moving to
  the start coordinates of the activeItem.
  */
  this.map.getView().animate({
    zoom: this.map.getView().getZoom() - 4,
    duration: 1500 / 2
  }, {
    zoom: 12,
    duration: 1500 / 2
  });
};
```

Above we created a function called: "zoomToLocation()". This function is assigned to the button: "Zoom to start", defined in the HTML file of the MapComponent. The function gets the current view and animates it to move to the start coordinates of the item on which the: "zoom to start" button is clicked. There are 2 animations which are executed, these are as follows:

- ➔ Move to the location of the startCoordinate;
- ➔ Zoom out and in again on the startCoordinate.

The value: "1500" defines the amount of time it takes for the animation to complete. You can increase or decrease it if you want. Now when we select a new item we can display information related to that item and zoom to the start location as shown in the illustration below.



At this point no data is shown on the map yet. We are going to add this functionality in the next section.

4.2.6 Creating and setting Layer groups

In this section we are going to add the code which is used to actually visualize the selected data obtained from our MongoDB datastore. We are going to visualize the data by adding extra layers on the base layer which we defined earlier in this document. The layers we are going to be creating are as follows:

1. A pointLayer: This is the layer which will contain the arrows that visualize the direction in which the item is going.
2. A markerLayer: This layer contains the start and end marker of the visualized route.
3. A lineLayer: This is the layer that creates the lines between the datapoints.

Before creating the function which is used to create the layers we need to create a global variable called:"layerStyles". This variable is going to contain the default styling of the layers which we will be creating. To create this global variable we need to add the following code below the global variable called:"activeItem":

```
public layerStyles: any = {
  'lineString': new ol.style.Style({
    stroke: new ol.style.Stroke({
      width: 2,
      color: "#FF0000",
    })
  }),
  'startMarker': new ol.style.Style({
    image: new ol.style.Icon({
      anchor: [0.5, 1],
      src: `assets/img/pins/pin_s_Red.png`
    })
  }),
  'endMarker': new ol.style.Style({
    image: new ol.style.Icon({
      anchor: [0.5, 1],
      src: `assets/img/pins/pin_e_Red.png`
    })
  })
};
```

Above we created a global variable called: "layerStyles". The value of this variable is a dictionary that contains the default styling of the layers.

To create a line we use the OpenLayers style: "Stroke". We give the stroke a width and a color.

To create a marker we use the OpenLayers style: "Icon". We pass the location of our pins (located in the assets/img folder of our application) as data source of the icon. We also anchor the icon to be displayed above the data point.

Now we want to create the function: “addLayerGroup()’, which is triggered in the function: “loadItemData()”, which we created in the previous chapter. First an explanation related to the function we are going to add is given.

The function addLayerGroup() will create the following layers for the item that is passed as parameter when the function is triggered in the function:"loadItemData()":

- ➔ A lineLayer: This is the layer that creates the lines between the datapoints.
- ➔ A pointLayer: This is the layer which will contain the arrows that visualize the direction in which the item is going.
- ➔ A markerLayer: This layer contains the start and end marker of the visualized route.

The following steps are executed when the function is triggered:

1) We assign the value of "this" to a variable called: "_this". We need to do this when we want to use global variables in an nested function. A nested function is a function inside another function.
2) We assign the value of the dateRangeSelected selected to the variable called: "layerGroupSelector". We do this because the keys in the JavascriptMap are the dateRangeSelected values of each layerGroup.

   We are going to use the variable: "layerGroupSelector" to select and edit specific layerGroups later.
3) A check is performed to see whether a layerGroup with that key already exists in the JavascriptMap: "layerGroups".

   If this is the case nothing will happen and the function will stop executing.

   If this is NOT the case the following steps will be executed.
4) We create a new OpenLayers lineString geometry using the syntax: "new ol.geom.LineString()" in which we pass the coordinateList belonging to the item for which we are going to create a layerGroup.

   After the geometry of type LineString is created we assign it to a variable called: "lineGeometry".
5) We create a new lineLayer to which we assign the lineGeometry as geometry. We also use a styling function to assign the styling of the lineString. The styling is defined and assigned to the global variable: "layerStyles".

   The value assigned to this global variable is a dictionary that contains three entries:

   1) lineString, which is the styling of the lineLayer.
   2) startMarker, which is the styling of the startMarker.
   3) endMarker, which is the styling of the endMarker.
6) An empty list of points is created. We will add all the points, which will be created later on, to this list. Then we will pass this list to the pointLayer.
7) An empty list of pointRotations is created. We will add al the calculated rotations of the points to this list. The rotation of the point defines in which way the arrow icon will point. The arrow icons visualize the direction in which the item was moving.

8) Create a FORLoop that loops trough all the coordinates in the coordinateList belonging to the item to which a layerGroup is added.

   In this for loop the following happens for each entry (data row) in the list:

   1) A variable point1 is created to which we assign the value of the coordinate on the index that the FORLoop is on.
   2) A variable point2 is created to which we assign the value of the coordinate on the index + 1 that the FORLoop is on.
   3) The rotation(direction in which the item was moving) is calculated using the build-in JavaScript function: "Math.atan2()". In this function we pass 2 parameters, these parameters are as follows:
      - Parameter 1: The latitude coordinate of point2 - the latitude coordinate of point1.
      - Parameter 2: The longitude coordinate of point2 - the longitude coordinate of point2.

        The result of this calculation is then added to the pointRotations list using the build-in JavaScript function: ".push()".
   4) The distance between point 1 and point 2 is calculated by creating a new OpenLayers geometry of type: "LineString" and passing that lineString as parameter in the build-in OpenLayers function: "ol.sphere.getLength()".

      Then we add the result to the distance that was calculate in the previous pass trough the loop FORLoop.

      Then we add the result of the step above to the list: "routeDistanceList", using the build-in JavaScript function:".push()".

      Using this technique makes sure that when we animate the visualized route we can see the distance that is traveled.
   5) A new feature is created to which we assign the value (coordinates) of point1 as geometry of this feature.

      We also create a new styling which is assigned to the style of the feature. The styling of the feature is an .SVG of an arrow. This SVG is located in the folder: "../../assets/img/" and is called: "arrow.svg".

      We pass the rotation which was calculated in step 7.3 as rotation of the svg. Because we do this the arrow will point in the direction the item was moving.
   6) After the point feature is created it's added to the points list.

9) A new layer is created and assigned to the variable: "pointLayer". We create a new VectorSource and assign it to the value: "source" of the layer.

   In the newly created VectorSource we assign the list of points, created in step 7, to the value: "features".

   We set the layer visibility to false because we only want to show the pointLayer when the user toggles it on.

10) A new Layer is created and assigned to the variable: "markerLayer". We create a new VectorSource and assign it to the value: "source" of the layer. In the VectorSource we create 2 new features. These features are as follows:

    1) A feature with the type: "startMarker". We assign the startCoordinate value of the item as the feature's geometry.

    2) A feature with the type: "endMarker". We assign the endCoordinate value of the item as the feature's geometry.

    Here we also use a styling function to assign the styling of the start and endMarker. The styling is defined and assigned to the global variable: "layerStyles".

    As mentioned before: The value assigned to this global variable is a dictionary that contains three entries:

    - ➔ lineString, which is the styling of the lineLayer.
    - ➔ startMarker, which is the styling of the startMarker.
    - ➔ endMarker, which is the styling of the endMarker.

    Then we set the zIndex of the markerLayer to 100. This makes sure that the markers are displayed on top of the other features.

11) A new layerGroup entry is added to the JavascriptMap: "layerGroups" belonging to the item.

    The key of this new layerGroup entry is the value of the variable: "layerGroupSelector". The value of this new layerGroup entry is a dictionary that contains the following entries:

    - ➔ lineLayer, which has the following values:
        - ➢ layer, which contains the actual lineLayer of this layerGroup.
        - ➢ coordinates, which contains the coordinates of the datapoints in this layerGroup.
        - ➢ altitudes, which contains the altitude values of the datapoints in this layerGroup.
        - ➢ dates, which contains the DTG values of the datapoints in this layerGroup.
        - ➢ distance, which contains the total distance of the layerGroup. The total distance is calculated using the build-in OpenLayers function:"ol.sphere.getLength()", in which we pass the value of the variable: "lineGeometry", which we created in step 4.
    - ➔ pointLayer, which has the following values:
        - ➢ layer, which contains the actual pointLayer of this layerGroup.
        - ➢ pointRotations, which contains the rotations of all the datapoints in this layerGroup.
        - ➢ routeDistance, which contains the distance's from point to point in this layerGroup.
    - ➔ markerLayer, which has the value layer, which contains the actual markerLayer of this layerGroup.

12) The newly created layerGroup is set as activeLayerGroup using the function: "setLayerGroup()", and passing the variable: "layerGroupSelector" as parameter in this function.

13) The lineLayer is added to the map using the build-in OpenLayers function: ".addLayer()", in which the lineLayer is passed as parameter.

14) The pointLayer is added to the map using the build-in OpenLayers function: ".addLayer()", in which the pointLayer is passed as parameter.

15) The markerLayer is added to the map using the build-in OpenLayers function: ".addLayer()", in which the markerLayer is passed as parameter.

Now that you know what steps are going to be executed in the function: “addLayerGroup” we can start coding the function. How to do this is shown in the following illustrations. We add the function: “addLayerGroup()” below the function: “zoomToLocation()”.

NOTE: This function is described using 4 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
addLayerGroup(item: Item): void {

  // Here we assign the value of this to a variable called: '_this'.
  // We need to do this since the function: "addLayerGroup" contains
  // nested functions.
  let _this = this;

  // Here we assign the value of the dateRangeSelected as layerGroup
  // selector.
  let layerGroupSelector = item.dateRangeSelected;

  // Here we check whether the layerGroup has already been selected.
  if (!item.layerGroups.has(layerGroupSelector)) {

    // Here we create a new lineString and pass the coordinateList as
    // parameter.
    let lineGeometry = new ol.geom.LineString(item.coordinateList);

    // Here we create a new Vector, VectorSource and feature.
    // we add the lineGeometry as geometry of the feature.
    let lineLayer = new ol.layer.Vector({
      source: new ol.source.Vector({
        features: [new ol.Feature({
          type: 'lineString',
          geometry: lineGeometry
        })]
      }),
      // Here we use a style function to set the styling of the
      // lineLayer.
      style: function (feature) {
        return _this.layerStyles[feature.get('type')];
      },
      zIndex: 100
    });
```

```
});

// Here we create an empty list of points.
let points = [];

// Here we create an empty list of pointRotations.
let pointRotations = [];

item.routeDistanceList = [0];

// Here we create a FORloop that loops an amount of times that is
// equal to the length of the coordinateList.
for (let i = 0; i < item.coordinateList.length - 1; i++) {

  // Here we create 2 points.
  // The first point gets the value of the coordinates on the index in
  // the coordinateList on which the loop currently is.
  let point1 = item.coordinateList[i];
  // The second point gets the value of the coordinates on the index + 1
  // in the coordinateList on which the loop currently is.
  let point2 = item.coordinateList[i + 1];

  // Here we add the calculated rotations to the pointRotations list.
  pointRotations.push(
    Math.atan2(point2[1] - point1[1], point2[0] - point1[0])
  );


  // Here we add the distance between point1 and point2 to the
  // routeDistanceList. We add the calculated distance to the
  // value of the previous entry of the routeDistance list.
  item.routeDistanceList.push(
    item.routeDistanceList[i] += ol.sphere.getLength(
      new ol.geom.LineString([point1, point2])
    )
  );


  // Here we create the pointStyle.
  let pointStyle = new ol.style.Style({
    image: new ol.style.Icon({
      src: '../../assets/img/arrow.svg',
      anchor: [0.75, 0.5],
      scale: 0.5,
      rotateWithView: true,
      rotation: -pointRotations[i],
      color: '#4271AE',
    }),
  });
```

```
    });

    // Here we create a new feature from which we set the geometry to
    // the geometry of point1.
    let point = new ol.Feature({
      geometry: new ol.geom.Point(point1),
    });

    // Here we add the styling to the point using the syntax: ".setStyle()"
    point.setStyle(pointStyle);

    // Here we add the point to our list of points using the syntax:".push()".
    points.push(point);
  };

  // Here we create the point layer and add the list of points as
  // geometry of this feature. We also set the visibility to false
  // since we only want to show the pointLayer when the user toggles it.
  // We also set the zIndex of this layer to 99 since we want it to
  // be displayed below the other layers.
  let pointLayer = new ol.layer.Vector({
    source: new ol.source.Vector({
      features: points
    }),
    visible: false,
    zIndex: 99,
  });

  // Here we create the markerLayer to which we add 2 features which
  // are the markers.
  let markerLayer = new ol.layer.Vector({
    source: new ol.source.Vector({
      features: [
        // We set the geometry of the startMarker to the startCoordinate of
        // the item which we are going to add.
        new ol.Feature({
          type: 'startMarker',
          geometry: new ol.geom.Point(item.startCoordinate)
        }),
        // We set the geometry of the endMarker to the endCoordinate of
        // the item which we are going to add.
        new ol.Feature({
          type: 'endMarker',
          geometry: new ol.geom.Point(item.endCoordinate)
        })
      ]
    }),
    // Here we use a style function to set the styling of the
    // markers.
    style: function (feature) {
      return _this.layerStyles[feature.get('type')];
    },
```

```
      },
      // We also set the zIndex of this layer to 101 since we want it to
      // be displayed on top of the other layers.
      zIndex: 101,
    });

    // Here we add a new entry to our layerGroups JavaScriptMap.
    item.layerGroups.set(layerGroupSelector, {
      'lineLayer': {
        'layer': lineLayer,
        'coordinates': item.coordinateList,
        'altitudes': item.altitudeList,
        'dates': item.datetimeList,
        'distance': (Math.round(ol.sphere.getLength(lineGeometry) / 1000 * 100) / 100)
      },
      'pointLayer': {
        'layer': pointLayer,
        'pointRotations': pointRotations,
        'routeDistance': item.routeDistanceList
      },
      'markerLayer': {
        'layer': markerLayer,
      }
    });

    // Here we set the layerGroup to be the activeLayerGroup.
    this.setLayerGroup(layerGroupSelector);

    // Here we add the layers to the OpenLayers map.
    this.map.addLayer(lineLayer);
    this.map.addLayer(pointLayer);
    this.map.addLayer(markerLayer);

  } else {
    return;
  };
};
```

That's it! Now you have created the function which is required to actually add features and objects to the map. The function will not work yet since we need to add one more function which is used to set an active LayerGroup. This function is called: "setLayerGroup()" and takes a groupKey, which is the layeGroupSelector value, as input parameter. Before we are going to create this function you should know what the function is going to do and which steps are going to be executed in the function. The function: "setLayerGroup()" can be triggered in the following functions:

1) removeLayerGroup(), when removing the current active layer group we need to set the next selected layer group to be the active layer group. (The function: "removeLayerGroup()" will be added later in this document.
2) addLayerGroup(), when a layer group is added we need to set the added layer group to be the activeLayerGroup

The function: “setLayerGroup()” is used to set the active layer group and assign the values of the active layer group to the correct parameters of the activeItem. These values are as follows:

1) The selected dateRange of the active layer group, which is used to set the start and end date of the DTG selection dropdown boxes.
2) The totalRouteDistance of the active layer group, which is used to calculate the routeDistance traveled by an item.
3) The coordinateList of the active layer group, which is used to visualize the data points and lines on the map. The coordinateList is also used when animating the route.
4) The altitudeList of the active layer group, which is used in the elevationProfile.
5) The datetimeList of the active layer group, which is used to display the start and end date of the route. This list is also used in the information box when animating the route.
6) The startCoordinate of the active layer group, which is used to set the start marker.
7) The endCoordinate of the active layer group, which is used to set the end marker.

This function also toggles all the overlays of the previous active layerGroup off and then creates new overlays related to the new active layerGroup. This will be done by using the function: “toggleOverlay()” and “setStaticOverlay()”. These 2 functions will be added later in this document.

Now that you know what the function: “setLayerGroup()” is used for we can start coding this function. The code required to create this function is shown in the illustrations below.

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
setLayerGroup(groupKey: string): void {

  // Here we assign the current active item to a variable called: "item".
  let item = this.activeItem;

  // Here we clear the animation of the current active layer group. This will
  // only happen if an animation is running. This is done because otherwise
  // the animation of the previous active layer group will keep running when
  // selecting a new layer group.
  this.clearAnimation();

  // Here we assign the layer group selector (which is the dateRange of the
  // selected layerGroup) to the variable:"dateRangeSelected".
  item.dateRangeSelected = groupKey;

  // Here we set the layerGroup which is selected to be the activeLayerGroup
  // using the layerGroupSelector (groupKey).
  item.activeLayerGroup = item.layerGroups.get(groupKey);

  // Here we assign the value of the distance of the lineLayer to the variable
  // totalRouteDistance. This value was added to the lineLayer dict when the
  // layerGroup was created in the function: "addLayerGroup".
  item.totalRouteDistance = item.activeLayerGroup.lineLayer.distance;

  // Here we assign the list of coordinates of the lineLayer to the variable
```

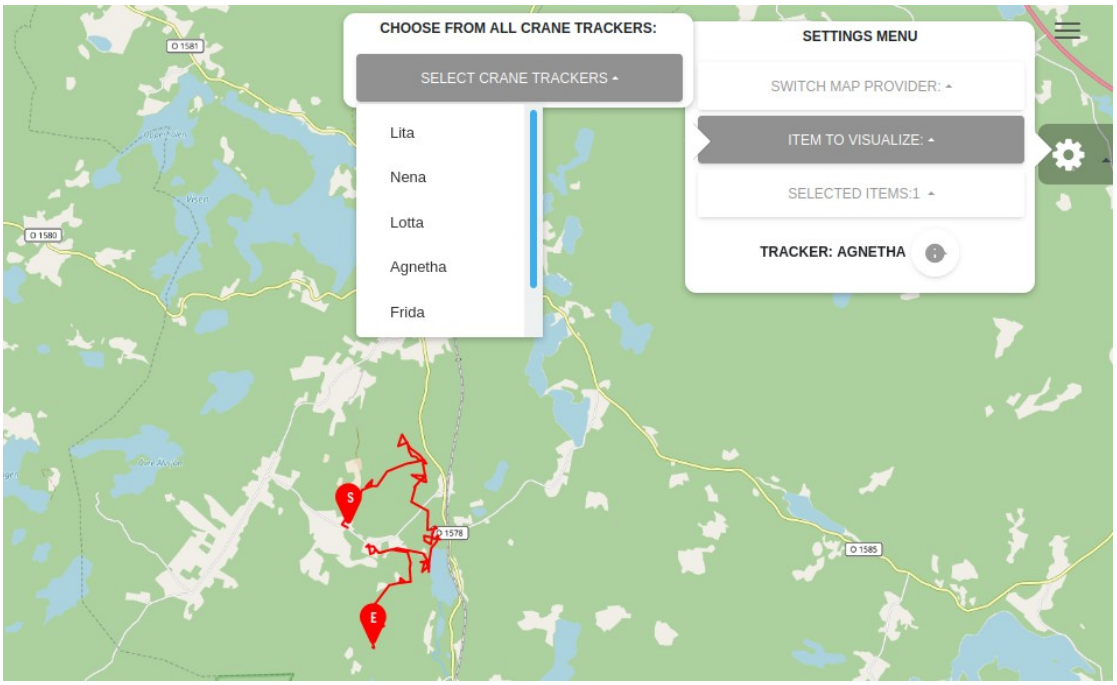
```
  // Here we assign the list of coordinates of the lineLayer to the variable
  // coordinateList. This value was also added to the lineLayer dict when the
  // layerGroup was created in the function: "addLayerGroup".
  item.coordinateList = item.activeLayerGroup.lineLayer.coordinates;

  // Here we assign the list of altitudes of the lineLayer to the variable
  // altitudeList. This value was also added to the lineLayer dict when the
  // layerGroup was created in the function: "addLayerGroup".
  item.altitudeList = item.activeLayerGroup.lineLayer.altitudes;

  // Here we assign the list of DTG's of the lineLayer to the variable
  // datetimeList. This value was also added to the lineLayer dict when the
  // layerGroup was created in the function: "addLayerGroup".
  item.datetimeList = item.activeLayerGroup.lineLayer.dates;

  // Here we assign the first coordinate of the coordinateList (on index 0)
  // of the lineLayer to the variable startCoordinate.
  item.startCoordinate = item.activeLayerGroup.lineLayer.coordinates[0];

  // Here we assign the first coordinate of the coordinateList (on index
  // (length of coordinateList - 1)) of the lineLayer to the variable
  // endCoordinate.
  item.endCoordinate = item.activeLayerGroup.lineLayer.coordinates[
    item.coordinateList.length - 1
  ]

  // Here we call the function toggleOverlay and pass "all" as parameter.
  // This makes sure the old overlays are removed from the map.
  this.toggleOverlay("all");

  // Here we create the new static overlays (start and end marker overlays)
  // using the information (assigned in the lines above) of the current item.
  this.setStaticOverlays(item)
}
```

That's it! If you want to test whether everything is working correctly you should comment out (place "//" in front of the line of code) the following lines (since these functions have not been created yet):

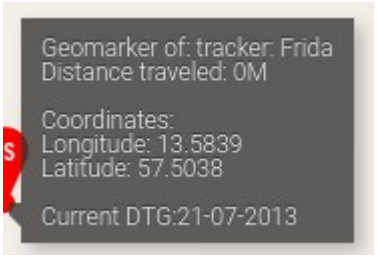
```
// Here we call the function toggleOverlay and pass "all" as parameter.
// This makes sure the old overlays are removed from the map.
//this.toggleOverlay("all");

// Here we create the new static overlays (start and end marker overlays)
// using the information (assigned in the lines above) of the current item.
//this.setStaticOverlays(item)
```

```
// selecting a new layer group.
//this.clearAnimation();
```

Now if you select an item the data related to the selected item will be shown on the map as shown in the illustration below:



Now that we are able to display Points, Lines and Markers we want to add some overlays containing information related to the start, end and geomarker. The end product of this will be the same as shown in the illustration below:



4.2.7 Creating and setting Overlays

To create the overlay information boxes we first need to add some lines of code to the map-component.html file. So let's open this file and add the following lines at the top of the file:

```
<div id="geomarker" style="background-color: red; height: 10px; width: 10px; border-radius: 100px;"></div>
<div id="geomarkerInfo" class="hint--no-animate hint--right hint--always" data-hint=""></div>
<div id="startmarkerInfo" class="hint--no-animate hint--right hint--always" data-hint=""></div>
<div id="endmarkerInfo" class="hint--no-animate hint--left hint--always " data-hint=""></div>
```

As you can see we added 4 elements to the HTML file. Each div element will be filled with data related to the div element in question. The data that will be added consists of the name, coordinates, DTG etc. of the item which is visualized.

The first div element represents the Geomarker which is the circle which will be moving when animating the route. The animation will be added later in this document. The second div element

represents the information box which will also move when the animation is played. This info-box will contain dynamic values which will update as the Geomarker moves along the displayed route.

The third div element represents the StartMaker information box and the forth div element represents the EndMarker information box. The values in these boxes will be static since the will not be updated when animating a route.

Now we need to go back to the map-component.ts file to add a new function which is used to initialize and create the overlays (information boxes) when the application is loaded. Before creating the function: “addOverlays()” you should know what the function will do and what steps are executed in the function.

The purpose of this function is to instantiate the overlays on the map. These overlays are as follows:

1) The GeoMaker, which is the marker that moves from coordinate to coordinate when the animation is playing.
2) Info box of the GeoMaker, which is the information box that moves with the GeoMaker when the animation is running (The currentCoordinateIndex). This information box contains the following DYNAMIC info:
   - The name of the route that is visualized
   - The coordinates on which the GeoMaker is. This value is constantly updated using the coordinateList belonging to the item that is currently visualized
   - The DTG on which the GeoMaker is. This value is also constantly updated using the datetimeList belonging to the item that is currently visualized.
3) The Start Marker info box, which is the information box assigned to the start marker (first coordinate of the coordinateList) of the visualized route. This information box contains the following STATIC info:
   - The name of the route that is visualized
   - The first set of coordinates in the coordinateList belonging to the item that is currently visualized.
   - The first DTG in the datetimeList belonging to the item that is currently visualized.
4) The End Marker info box, which is the information box assigned to the end marker (the last coordinate of the coordinateList) of the visualized route. This information box contains the following STATIC info:
   - The name of the route that is visualized.
   - The last set of coordinates in the coordinateList belonging to the item that is currently visualized.
   - The last DTG in the datetimeList belonging to the item that is currently visualized.

The way to create an overlay is as follows: The syntax used for creating an overlay is “new ol.Overlay()”. We pass the following values as input parameters:

➔ An id which represents the overlay.

- ➔ The positioning in which the overlay should be displayed.
- ➔ The position of the overlay. When instantiating the overlay, we set the position to undefined since we don't have any position yet.
- ➔ The HTML element which represents the overlay. These elements are defined at the top of the map.component.html file. For this we use the syntax: "document.getElementById('{the id of the element in the HTML file}')"

Now that you know what the function: “addOverlays()” will do we can start coding the function. We add the function below the function: “setLayerGroup()” which we created earlier. We do this by adding the following code to the map.component.ts file:

```
addOverlays(): any[] {

  // Here we create the GeoMaker overlay and assign it to a variable called:
  // "marker".
  let marker = new ol.Overlay({
    id: 'geomarker',
    positioning: 'center-center',
    position: undefined,
    element: document.getElementById('geomarker'),
  });

  // Here we create the GeoMaker Info overlay and assign it to a variable
  // called:"markerInfo".
  let markerInfo = new ol.Overlay({
    id: 'geomarkerInfo',
    positioning: 'center-center',
    position: undefined,
    element: document.getElementById('geomarkerInfo'),
  });

  // Here we create the startMarker Info overlay and assign it to a variable
  // called:"startMarkerInfo".
  let startMarkerInfo = new ol.Overlay({
    id: 'startmarkerInfo',
    positioning: 'center-center',
    position: undefined,
    element: document.getElementById('startmarkerInfo'),
  });

  // Here we create the endMarker Info overlay and assign it to a variable
  // called:"endMarkerInfo".
  let endMarkerInfo = new ol.Overlay({
    id: 'endmarkerInfo',
    positioning: 'center-center',
    position: undefined,
    element: document.getElementById('endmarkerInfo'),
  });

  //Here we return a list containing the marker instances.
  return [marker, markerInfo, startMarkerInfo, endMarkerInfo]
}
```

Now we need to make sure the overlays are added to the map when it's created. For this we need to edit a function which we created earlier. The function we need to edit is the function called: "createOpenLayersMap()". So scroll to this function in the map.component.ts file and change the following code:

```
this.map = new ol.Map({
  target: 'map',
  view: mapViewSettings,
  layers: [this.mapLayer]
});
```

```
this.map = new ol.Map({
  target: 'map',
  view: mapViewSettings,
  layers: [this.mapLayer],
  overlays: this.addOverlays()
});
```

The added line (overlays: this.addOverlays()) make sure the overlays are instantiated when the map component is loaded.

At this point the overlays still not show up when selecting data. For this we need to make 2 more functions which are as follows:

1) setDynamicOverlays(), which is used to set the values of the GeoMarker information box and update the position of the GeoMarker itself when animating the route;
2) setStaticOverlays(), which is used to set the values of the Start and EndMarker information boxes;

Before we are going to created the function: setDynamicOverlays() you first need to understand why it's used and what steps are executed.

As mention before; function setDynamicOverlays() is used to used to set the values of the GeoMarker information box and update the position of the GeoMaker itself. The function takes an Item as input parameter on the function call. The item passed in the function is also the activeItem.

The function is triggered in the following functions (which will be created later in this document):

1) setStaticOverlays(),because everytime the static overlays are set the dynamic overlays also needs to be set. This is not the other way around.
2) animateRoute(), because the information displayed in the GeoMaker info box needs to be updated constantly when an animation is playing.
3) clearAnimation(), because when an animation is cleared the GeoMaker info box needs to be reset to the original state.

The function uses the value of the item's currentCoordinateIndex to determine what values have to be extracted from the datalist in question (e.g. the coordinateList). The value of the currentCoordinateIndex is constantly incremented in the animateRoute() function.

The following steps are executed in this function:

1) The div element representing the geomarker is obtained from the HTML page.
2) The div element representing the GeoMaker info box is obtained from the HTML page.

3) The current coordinates from the coordinateList of the activeItem (the coordinate on the index of the value of the currentCoordinateIndex) is obtained and transformed in a coordinate format which is human readable.
4) The longitude coordinate is extracted and transformed to only contain 4 numbers behind the decimal (e.g 1.xxxx).
5) The latitude coordinate is extracted and transformed to only contain 4 numbers behind the decimal (e.g 1.xxxx).
6) The current datetime is extracted form the datetimeList (the datetime on the index of the value of the currentCoordinateIndex).
7) The current distance traveled is extracted from the routeDistanceList (the distance on the index of the value of the currentCoordinateIndex).
8) A check is performed to check if the distance is not equal to undefined. This is done since the first distance value is undefined.

   In case the distance is **EQUAL** to undefined, the distance is set to 0.

   In case the distance is **NOT EQUAL** to undefined current distance traveled is extracted and transformed to not contain any numbers behind the decimal.
9) The position of the GeoMaker is updated using the current coordinates.
10) The position of the GeoMaker information box is updated using the current coordinates.
11) The content of the HTML div element, representing the GeoMaker Info box, is added using the data extracted in the previous steps.

Now that you know what the function is used for, we can start coding the function. Adding the function is done by adding the following code below the function: “addOverlays()” which we created earlier:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
setDynamicOverlays(item: Item): void {

  // Here we obtain the HTML element representing the geomarker.
  let geomarker = this.map.getOverlayById('geomarker')

  // Here we obtain the HTML element representing the GeoMaker info box.
  let geoMarkerInfo = this.map.getOverlayById('geomarkerInfo')

  // Here we transform the current coordinates to a human readable format.
  let transformedCoord = ol.proj.transform(
    item.coordinateList[item.currentCoordinateIndex],
    'EPSG:3857', 'EPSG:4326')

  // Here we extract and transform the longitude coordinates to only contain
```

```
  // Here we extract and transform the longitude coordinates to only contain
  // 4 numbers behind the decimal. We do this by using the syntax: ".toFixed".
  let longitudeCoord = transformedCoord[0].toFixed(4)

  // Here we extract and transform the latitude coordinates to only contain
  // 4 numbers behind the decimal. We do this by using the syntax: ".toFixed".
  let latitudeCoord = transformedCoord[1].toFixed(4)

  // Here we extract the current DateTimeGroup.
  let datetime = item.datetimeList[item.currentCoordinateIndex]

  // Here we extract the current distance traveled.
  let distance = item.routeDistanceList[item.currentCoordinateIndex - 1]

  // Here we perform a check to determine whether the value of the distance
  // is not equal to undefined.
  distance != undefined ? distance = item.routeDistanceList[item.currentCoordinateIndex].toFixed(0)
                        : distance = 0

  // Here we set the position of the geomarker by passing the current coordinates.
  geomarker.setPosition(item.coordinateList[item.currentCoordinateIndex]);

  // Here we set the position of the GeoMaker Info box, using the current coordinates.
  geoMarkerInfo.setPosition(item.coordinateList[item.currentCoordinateIndex]);

  // Here we set the content of the GeoMaker Info box HTML element.
  // We use the syntax: "\u000A" to add a next line to the text.
  geoMarkerInfo.getElement().setAttribute('data-hint',
    'Geomarker of: ' + item.type + ': ' + item.name +
    '\u000A' +'Distance traveled: ' + distance + 'M' +
    '\u000A\u000ACoordinates:\u000ALongitude: ' + longitudeCoord +
    '\u000ALatitude: ' + latitudeCoord +
    '\u000A\u000ACurrent DTG:' + datetime);
}
```

The last function we need to add in order to create the overlays is the a function called: “setStaticOverlays()”. This function is used to used to set the values of the Start and endMarker information boxes. The function takes an Item as input parameter on the function call. The item passed in the function is also the activeItem.

The function is triggered in the following functions:

1) setLayerGroup(),because when a new layerGroup is added it automatically becomes the activeLayerGroup. So we need to update the overlay content and positions according to the new activeLayerGroup.
2) loadItemData(), because when an item is loaded for the first time the item automatically becomes the activeItem so we need to update the overlay content and positions according to the new activeItem.
3) selectItem(), because when an item is select it becomes the activeItem so we need to update the overlay content and positions according to the new activeItem.
4) removeItem(), because when an item is removed the next item in the selectedItem list will become the activeItem so we need to update the overlay content and positions according to the new activeItem. This function will be added later in this document.

The following steps are executed in this function:

1) The function setDynamicOverlays() is triggerd since the dynamic overlays always need to be updated when the static overlays are updated.
2) The div element representing the startMarker info box is obtained from the HTML page.
3) The startCoordinates of the activeItem is obtained and transformed in a coordinate format which is human readable.
4) The content of the HTML div element, representing the startMarker info box, is added using the data extracted in the previous steps.
5) The div element representing the endMarker info box is obtained from the HTML page.
6) The endCoordinates of the activeItem is obtained and transformed in a coordinate format which is human readable.
7) The content of the HTML div element, representing the endMarker info box, is added using the data extracted in the previous steps.

Now that you know what the function is used for, we can start coding the function. Adding the function is done by adding the following code below the function: "setDynamicOverlays()" which we created earlier:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
setStaticOverlays(item: Item): void {

  // Here we trigger the function setDynamicOverlays() in which we pass
  // the item which was passed as input parameter on the function call.
  this.setDynamicOverlays(item);

  // Here we obtain the HTML div element representing the startMarker info box
  // from the HTML page.
  let startMarkerInfo = this.map.getOverlayById('startmarkerInfo');

  // Here we transform the startCoordinate into a human readable format.
  let startCoordTransformed = ol.proj.transform(
    item.startCoordinate, 'EPSG:3857', 'EPSG:4326');

  // Here we set the content of the startMarker Info box HTML element.
  // We use the syntax: "\u000A" to add a next line to the text.
  startMarkerInfo.getElement().setAttribute('data-hint',
    'Start marker of: ' + item.type + ': ' + item.name + '\u000A' +
    'Distance traveled: ' + 0 + 'KM' +
    '\u000A\u000ACoordinates:\u000ALongitude: ' +
    startCoordTransformed[0].toFixed(4) + '\u000ALatitude: ' +
    startCoordTransformed[1].toFixed(4) +
    '\u000A\u000ACurrent DTG:' + item.datetimeList[0]);
```

```
    '\u000A\u000ACurrent DTG:' + item.datetimeList[0]);

  // Here we obtain the HTML div element representing the endMarker info box
  // from the HTML page.
  let endMarkerInfo = this.map.getOverlayById('endmarkerInfo')

  // Here we transform the endCoordinate into a human readable format.
  let endCoordTransformed = ol.proj.transform(
    item.endCoordinate, 'EPSG:3857', 'EPSG:4326')

  // Here we set the content of the endMarker Info box HTML element.
  // We use the syntax: "\u000A" to add a next line to the text.
  endMarkerInfo.getElement().setAttribute('data-hint',
    'End marker of: ' + item.type + ': ' + item.name + '\u000A' +
    'Distance traveled: ' + item.totalRouteDistance + 'KM' +
    '\u000A\u000ACoordinates:\u000ALongitude: ' +
    endCoordTransformed[0].toFixed(4) + '\u000ALatitude: ' +
    endCoordTransformed[1].toFixed(4) +
    '\u000A\u000ACurrent DTG:' + item.datetimeList[item.datetimeList.length - 1]);
}
```

If you commented the lines (in the function: “setLayerGroup()”) described at the bottom of page 42 you need to uncomment the line as shown in the illustration below:

```
// Here we create the new static overlays (start and end marker overlays)
// using the information (assigned in the lines above) of the current item.
//this.setStaticOverlays(item)
```

```
// Here we create the new static overlays (start and end marker overlays)
// using the information (assigned in the lines above) of the current item.
this.setStaticOverlays(item)
```

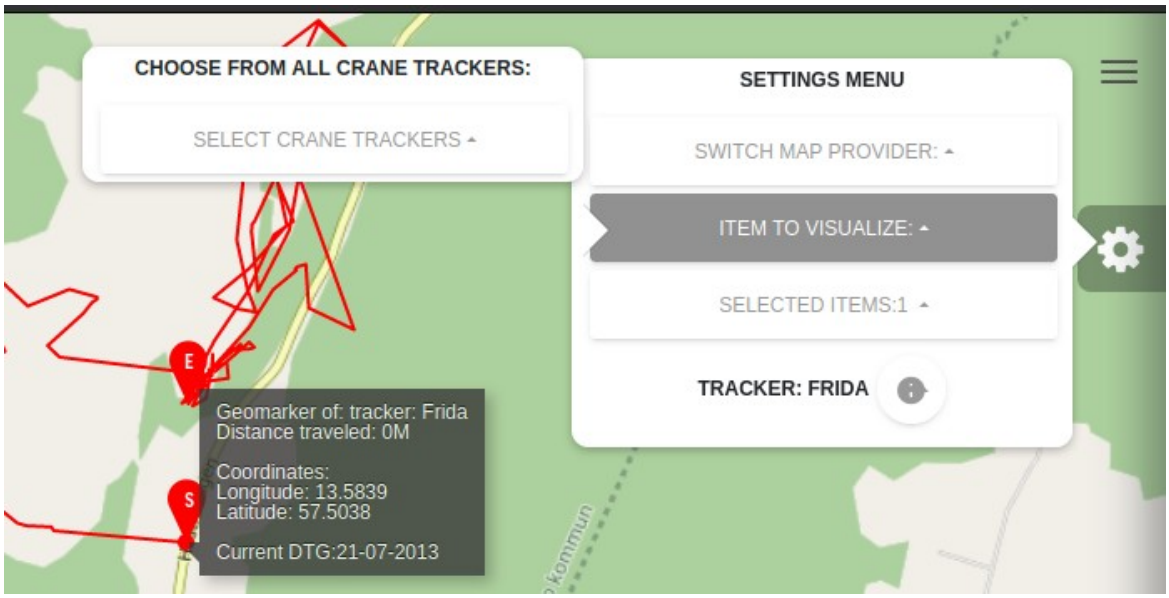
The last thing we need to do is to update some functions which we created earlier. We start of by updating the function: “loadItemData()”. We do this by adding the following at the bottom of the function (so below the line: “this.addLayerGroup(Item))”:

```
// Here we set the static overlays to contain the values of the item.
this.setStaticOverlays(item);
```

Now we need to do the same in the function:”selectItem()”. So we add the following below the line which starts with the syntax: “this.selectItems.filter(…….”:

```
// Here we set the static overlays to contain the values of the item.
this.setStaticOverlays(item);
```

Now when we reload the application and select an item you should be greeted with the information box related to the GeoMarker as shown in the illustration below:



As you can see the Start and EndMarker information boxes are not displayed yet. This will be fixed when adding the layer toggling functionality to the application which we will be doing later in this document. Let's first create the functionalities related to Removing selected items and LayerGroups. This will be done in the next sections.

4.2.8 Removing a selected Item

At this point we can only add items. When we click on the drop-down box related to the selected items we are greeted with a red icon displayed next to each selected item. But when we click on the icon nothing happens. We are going to add this functionality in this section.

To be able to remove items which have been selected we are going to create a function called: "removeItem()". This function is assigned to the delete button (icon) next to each item in the list of selectedItems. Assigning the function to the button was done in the map.component.html file by using the code shown in the illustration below:

```
<button class="pull-left" ngbDropdownItem
(click)='selectItem(item);'>{{item.name}}</button>
<a class="material-icons pull-left" style="color: red; margin-top: 9px;"
(click)='removeItem(item)'>not_interested</a>
```

The item on which the delete button is clicked is then passed as parameter in the function: "removeItem()".

If the function is triggered the following happens:

1) A check is performed to see if the item that is being deleted is the item that is currently active.

   If this is NOT the case (so the item that is being deleted is not the activeItem) nothing happens.

   If this IS the case (so the item that is being deleted IS the active item) the following happens:

   1) If there is an animation running, the animation is cleared. (This will be added later)
   2) The next value in the list of selectedItems (If there is one) becomes the activeItem.

2) A forEach loop is executed on the list containing the layerGroups belonging to the item that is being deleted.

   All layers (LineLayer, PointLayer and MarkerLayer) per layerGroup in the list of layerGroups are then removed from the map using the build-in OpenLayers function:".removeLayer()" in which the layer which needs to be remove is passed.

3) The list of layerGroups belonging to the item that is being removed is cleared.

4) The build-in JavaScript function: ".filter()" is called on the list of selectedItems. The value, on which is filtered, is the ItemId of the item that needs to be removed. If the item is found, the function: ".splice()" is called on the selectedItems list in which we pass the index on which the item which needs to be removed is located in the selectedItems list.

   The function:".splice()" then removes the item from the list.

5) A check if performed to find out whether the item that was removed was the last item in the selectedItems list.

   If this is the case all overlays (information popups) will be toggled off. (This will be added later)

   If this is NOT the case the overlays will be placed in the position of the new activeItem which was set in step 1.

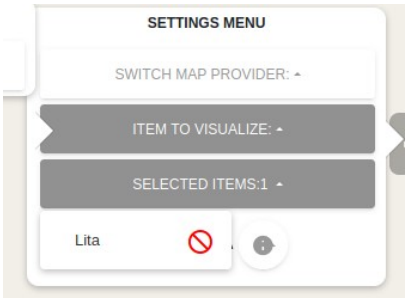
Now that you know what the function is used for, we can start coding the function. Adding the function is done by adding the following code below the function: “setStaticOverlays()” which we created earlier:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

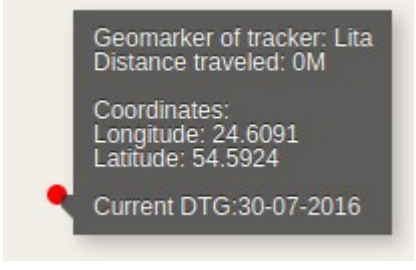
```
removeItem(item: Item): void {

  /*
   If the itemId of the item to remove is the same as the id of the item that
   is currently active. Change the activeItem to the next item in the list.
  */
  this.activeItem.id == item.id ? (
      this.selectItem(this.selectedItems.values().next().value)) :
    null;

  // Here we loop trough all the layers per layer group and remove them from
```

```
  // Here we loop trough all the layers per layer group and remove them from
  // the map.
  item.layerGroups.forEach(layerGroup => {
    for (let [key, value] of Object.entries(layerGroup)) {
      this.map.removeLayer(value['layer'])
    }
  });

  // Here we clear all the layerGroups from the JavaScriptMap: "layerGroups".
  item.layerGroups.clear();

  /*
   Filter the list of selectedItems to find the item which needs to be
   removed. If the id of the item to remove is equal to the id
   of one of the items in the selectItems list, the item is removed using
   the .splice() function on the selectedItems list.
   As parameter we pass the index on which the item to remove was found in
   the selectedItems list.
   */
  this.selectedItems.filter(
    (value, index) => value.id == item.id ? this.selectedItems.splice(index, 1) : null)

  /*
   If the length of the selectedItems list is 0, after removing the item:
   Toggle the overlays of. Else the overlays are set to the new item.
  */
  this.selectedItems.length == 0 ? null :
    this.setStaticOverlays(this.activeItem)
}
```

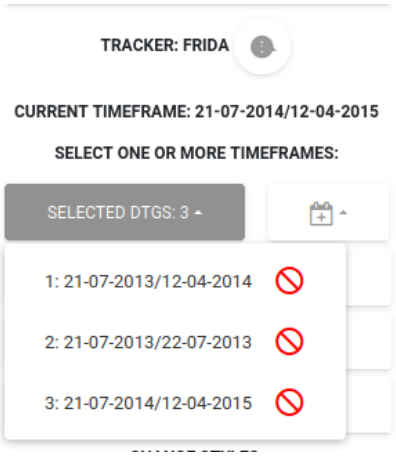
Now when you refresh the page, select and item and remove it using the button shown in the illustration below, the item will be removed from the selectedItems list and the Line, Point and Marker Layers will be removed from the map.



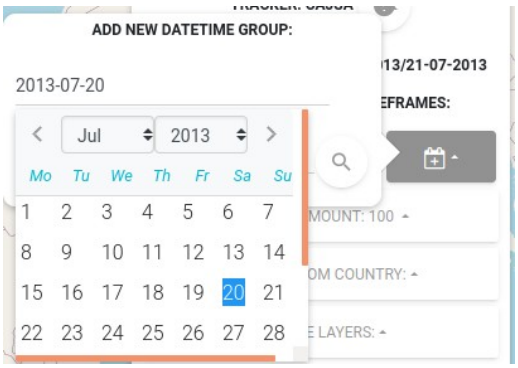
As you can see the GeoMarker and the GeoMarker information box will remain on the map as shown in the illustration below. This is because we did not add the function which toggles of the overlays when no items are selected. This will be done later in this document!



Now that we are able to select and remove items and single layer groups, we want to add the functionality to select data in a certain timeframe. We also want to add the functionality that enables us to select multiple timeframes from one item and switch between the selected timeframes. In the case of the Crane Tracker data this would come in handy when you want to visualize the Crane migration (e.g. in the summer) from multiple years. The end result of this will be as shown in the illustration below:



In this illustration you can see that 3 different time frames have been selected from the Crane: Frida. How to do this is shown in the next section. The timeframes can be selected using a calendar (also known as an NgbCalender) which is shown in the following illustration:



The calendar (also known as an NgbCalender) will be created as separate component called: “Datepicker” which contains a TypeScript file and a HTML file. But more on this later!

4.2.9 Adding DTG (Date time group) selection

To be able to add Date Time / time frame selection, we first need to add a function which is used to obtain data using a given time frame. This function will be called: “getItemDataByDTG()” and takes an Item, Start Date and End Date as input parameters.

This function is triggerd in the function: "getDTGEvent()" (which we will create later in this document) and is used to obtain all the datapoints from a certain item in a given timeframe. This function takes 3 input parameters which are as follows:

1) An item from which the data has to be obtained;
2) A Start date;
3) An End date.

This function triggers the function: "getTransmissionsDTG" in the CraneService which will then perform an API call to the Flask-API to obtain all the data between the given timeframe from the MongoDB datastore.

All the data between the start and end date will be returned and passed in the the function: "loadItemData()".

This function contains a switch/case. The switch case takes the itemType, which in our case can be a tracker, as input. Depending on the itemType, the corresponding function is triggered. The switch/case can be extended in case you want to add the functionality for other Dataset types.

Now that you know what the function is used for, we can start coding the function. This is done by adding the following code below the function: “removeItem()” which we created earlier:

```
getItemDataByDTG(item: Item, dtg_s, dtg_e): void {

  // Here we define the switch/case
  switch (item.type) {
    // In case the item.type is equal to 'tracker', the following code is
    // executed.
    case 'tracker':
      // Here we trigger the function: "getTransmissionsDTG" in the CraneService
      // we pass the itemId, Start Date and End Data as parameters and subscribe
      // the result to a variable called: "transmissions" which is then passed
      // in the function: "loadItemData()"
      this._CraneService.getTransmissionsDTG(item.id, dtg_s, dtg_e).subscribe(
        (transmissions) => {
          this.loadItemData(transmissions)
        }
      )
      break;
    default:
      break;
  }
};
```

Now let’s create the DatePicker component. This will be done in the components folder located at “/src/app/components”. So let’s create a new folder for this component by running the following command:

```
mkdir ~/Geostack/angular-apps/2d-map-viewer/src/app/components/datepicker
```

Now let’s create the Typescript file for the datepicker component. We do this by running the following command:

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/components/datepicker/
datepicker.component.ts
```

The last file we need to create is the HTML file of the datepicker. This is done by running the following command:

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/components/datepicker/
datepicker.component.html
```

Now let's open the datepicker.component.ts file and start coding the logic for the datepicker compontent.

We start of by importing the Angular modules which are required to build the datepicker compontent. We do this by adding the following to the top of the file:

```
import {
    Component,
    OnInit,
    Input,
    EventEmitter,
    OnChanges,
    Output
} from '@angular/core';

import {
    NgbModal,
    ModalDismissReasons,
    NgbDate,
    NgbCalendar
} from '@ng-bootstrap/ng-bootstrap';
```

Now we need to define the component metadata. This is done by adding the following code below the module imports:

```
/*
Here we define the component metadata. The following applies to the code below:
- The selector is going to be: "dtgpicker", so when we want to use this
  component we use the syntax <dtgpicker/> in an HTML page to which we want
  to add this component.
- The templateUrl is going to be: './datepicker.component.html', this HTML page
  contains the layout of the datepicker component.
*/
@Component({
    selector: 'dtgpicker',
    templateUrl: './datepicker.component.html'
})
```

Now we want to create the datepicker component class. This is done by adding the following below the component metadata:

```
// Here we create the datepicker component class
export class DatePickerComponent implements OnInit {

  // Here we create the datepicker constructor. We pass an instance
  // of an NgbModal, used to create a dropdown, and an NgbCalendar, which is
  // used to create the calendar, in the constructor.
  constructor(private modalService: NgbModal, calendar: NgbCalendar) {}

  /*
  Here we create the ngOnInit() function. All the logic in this function will
  be executed when the component is loaded.
  */
  ngOnInit() {
  };
}
```

Since the state of this component is “update-able” we also need another function called: “ngOnChanges()”. All the login in this function will be executed when the content of the component is changed (e.g. The Start and End date of the visualized item). We do this by adding the following code below the function: “ngOnInit()”:

```
/*
Here we create the ngOnChanges() function. All the logic in this function will
be executed when the component is updated / changed.
*/
ngOnChanges(): void {
};
```

Now we want to define the function which is used to open the NgbCalender. This function will be called: “open()’ and is assigned to the input forms which we will define later in the HTML page. We add this function by adding the following code below the ngOnChanges() function:

```
open(content, type, modalDimension) {
    if (modalDimension === 'sm' && type === 'modal_mini') {
        this.modalService.open(content, {
            windowClass: 'modal-mini',
            size: 'sm',
            centered: true
        });
    }
};
```

Now before continuing some things should be clear. Since the datepicker is a loose component we need to send data to the component (the Start and End Date). This data will be coming from the MapComponent when an Item or certain timeframe is selected.

In case an Item is selected we need to send the Start and EndDate of the route from the MapComponent to the DatepickerComponent so that the component knows the Start and EnDate which need to be set in the Calender.

In case an Timeframe is selected we need to send the selected Start and EndDate from the DatepickerComponent to the MapComponentent so that the function: “getItemDataByDTG()” can be triggered with the correct Start and EndDate.

The steps mentioned above is done using something called an EventEmitter. For more information related to EventEmitting in Angular you should read the following tutorial:

https://ultimatecourses.com/blog/component-events-event-emitter-output-angular-2

Since we want to send the date range (StartDate → EndDate) to the DatePickerComponent, we are going to add a global variable called: “dateRange” to the DatePickerComponent. This is done by adding the following code at the top of the DatePickerComponent class (so above the class constructor):

```
// Here we tell the DatePickerComponent to expect an input.
// This input will be the start and enddate of the route that is currently
// visualized.
@Input() dateRange : any;
```

We also want to send data from the DatePickerComponent to the MapComponent. This data contains the start and end date which the user selected using the NgbCalender. We are going to define a global variable called: “dateEvent”. We do this by adding the following line of code below the global variable called: “dateRange”:

```
// Here we tell the DateTimePicker component to output a dateEvent of type
// EventEmitter which needs to be of type NgbDate (The date format created
// by the NgbCalendar).
@Output() dateEvent = new EventEmitter<NgbDate[]>()
```

We also need to define 2 global variables to which we are going to assign the Start and EndDates obtained from the MapComponent. These 2 global variables are as follows:

1. startDate, which is used to set the startDate of the NgbCalender;
2. endDate, which is used to set the endDate of the NgbCalender.

We do this by adding the following code below the global variable:”dateEvent”:

```
// Here we define 2 global variables which are used to set the start and
// end dates of the NgbCalendar.
public startDate: NgbDate;
public endDate: NgbDate;
```

Now we need to create a function which is used to actually set the Start and End Date of the NgbCalender. This function will be called: “setDateRange()” and will take a list (containing a startDate at index 0 and an EndDate at index 1) as input parameter.

This function is triggered in the following functions:

- ➔ ngOnInit(), to make sure the dateRange is set when the component is loaded for the first time.
- ➔ ngOnChanges(), to make sure the dateRange is set when the component data is updated.

The following steps are executed when the function is triggerd:

1) Extract and transform the startDate from the list passed on the function call.
2) Transform the extracted date (from the previous step) into a valid NgbDate format and assign it to the global variable:"startDate".
3) Extract and transform the endDate from the list passed on the function call.
4) Transform the extracted date (from the previous step) into a valid NgbDate format and assign it to the global variable:"endDate".

Now that you know what the function is used for we can start coding the function. How this is done is shown on the next page.

Creating the function: “setDateRange()” is done by adding the following below the function:”open()” which we created earlier:

```
setDateRange(date){

    // Here we extract and transform the startDate.
    let startDate = new Date(date[0]);

    // Here we transform the startDate to a valid format.
    this.startDate = new NgbDate(startDate.getFullYear(),startDate.getMonth()+1,startDate.getDate());

    // Here we extract and transform the endDate.
    let endDate = new Date(date[1]);

    // Here we transform the endDate to a valid format.
    this.endDate = new NgbDate(endDate.getFullYear(),endDate.getMonth()+1,endDate.getDate());
};
```

Now we need to change the NgOnInit() and the NgOnChanges() functions to make sure the setDateRange() function is triggered when the component loads and updates. This is done by changing the 2 functions as shown in the illustration below:

```
/*- */
ngOnInit() {
};

/*- */
ngOnChanges(): void {
};
```

```
/*- */
ngOnInit() {
  this.setDateRange(this.dateRange)
};

/*- */
ngOnChanges(): void {
  this.setDateRange(this.dateRange)
};
```

Now we need to create 2 functions which are used to check whether the dates, selected in the NgbCalender, are not lower than the StartDate of the selected item and not higher than the EndDate of the selected item. These functions are as follows:

1. isRangeStart() which takes a startDate as input parameter;
2. isRangeEnd() which takes an endDate as input parameter;

We do this by adding the following code below the function: “setDateRange()”:

```
// Here we create a function which is used to check whether the input
// startDate is not lower than the startDate of the selected item.
isRangeStart(date: NgbDate) {
    return this.startDate && this.endDate && date.equals(this.startDate);
}

// Here we create a function which is used to check whether the input
// endDate is not higher than the endDate of the selected item.
isRangeEnd(date: NgbDate) {
    return this.startDate && this.endDate && date.equals(this.endDate);
}
```

Now we need to create a function which is used to check whether the selected date is in between the Start and End Date. This function will be called: “isInRange()” and will also take a date as input parameter. We do this by adding the following code below the function: “isRangeEnd()”:

```
// Here we create a function which is used to check whether the selected
// dates is inbetween the start and end dates.
isInRange(date: NgbDate) {
    return date.after(this.startDate) && date.before(this.endDate);
}
```

Now we need to create a function which is used to check what the currently selected date is. We will use this function to mark the currently selected date Blue so that the user knows what date is currently selected. We create this function by adding the following code below the function: “isInRange()”:

```
// Here we create a function which is used to check whether the selected
// dates is inbetween the start and end dates.
isInRange(date: NgbDate) {
    return date.after(this.startDate) && date.before(this.endDate);
}
```

Now we need to create 2 functions which are used to automatically change the start and end date in case one of the 2 becomes higher or lower than the other one. For example: If we have a start date which is 2020-05-20 and an end date which is 2020-05-23 everything is ok. But when we change the start date to 2020-05-25 (so 2 days later than the end date) we want the end date to automatically change to the same date as the start date. By doing this we will make sure no invalid data is send to the Flask-API when searching data in a certain timeframe. We do this by adding the following code below the function: “isInRange()“:

```
// Here we create a function which checks if the selected end date is not
// lower than the selected start date. If this is the case the start date
// will automatically be set to the endDate.
endDateChanged(date) {
    if (this.startDate && this.endDate && (this.startDate.year > this.endDate.year
      || this.startDate.year === this.endDate.year && this.startDate.month > this.endDate.month
      || this.startDate.year === this.endDate.year && this.startDate.month === this.endDate.month
      && this.startDate.day > this.endDate.day)) {
        this.startDate = this.endDate;
    }
};

// Here we create a function which checks if the selected startDate is not
// higher than the selected endDate. If this is the case the endDate
// will automatically be set to the startDate.
startDateChanged(date) {
    if (this.startDate && this.endDate && (this.startDate.year > this.endDate.year
      || this.startDate.year === this.endDate.year && this.startDate.month > this.endDate.month
      || this.startDate.year === this.endDate.year && this.startDate.month === this.endDate.month
      && this.startDate.day > this.endDate.day)) {
        this.endDate = this.startDate;
    }
}
```

The last function we need to add is the function which is used to send the Start and EndDate, selected in the NgbCalender, to the MapComponent after the search button is clicked in the application. This function will be called: “sendDTG” and is assigned to the Search button in the HTML page of the DatePickerComponenet (which we will create later). The function will use the global variable: “dateEvent” to emmit the value assigned to the global variables “startDate” and “endDate”. We create this function by adding the following code below the function: “startDateChanged()”:

```
// Here we create the function which is used to send the selected
// start and endDate to the MapComponent which will then use these values
// to select all the datapoints in a cetain timeframe.
sendDTG() {
    this.dateEvent.emit([this.startDate,this.endDate])
};
```

That’s it! Now you have finished the business logic of the DatePickerComponent. Now we need to add the HTML code to the datepicker.compose.html file. So let’s Open this file.

Let’s first define the div element in which we are going to add all the logic related to the layout of the DatePickerComponent. This is done by adding the following code to the file:

```
<!-- Here we define the HTML code for the DTG Picker -->
<div class="input-daterange datepicker">

</div>
```

Now let’s add the code related to the Start Date selection calendar. This is done by adding the following code inside the div element which we created above:

```
<!-- Here we define the HTML code for the Start Date calendar -->
<div class="form-group">
    <div class="input-group">

       <!-- Here we define the input box for the Start Date. We set the
       placeholder to the first element of the dateRange which is the start
       date of the current datapoints selected.-->
        <input #dtg1 class="form-control datepicker" placeholder={{dateRange[0]}}
        name="dp1" [(ngModel)]="startDate" ngbDatepicker #d1="ngbDatepicker"
        (click)="d1.toggle()" type="text" [dayTemplate]="t"
        (dateSelect)="startDateChanged($event)" autocomplete="off" />

        <!-- Here we add the Angular template which is used to create the
        NgbCalendar filled with dates to select. We set the selectable date
        to the start date of the currently selected datapoints so no dates
        below that date be selected.-->
        <ng-template #t let-date let-focused="focused">
            <span class="custom-day" [class.text-white]="isActive(date)"
            [class.range-end]="isRangeEnd(date)" [class.range-start]="isRangeStart(date)"
            [class.btn-light]="isActive(date)" [class.bg-primary]="isActive(date)"
            [class.range]="isInRange(date)" [class.faded]="isInRange(date)">
           {{ date.day }}
         </span>
        </ng-template>
    </div>
</div>
```

Now let's do the same for the calendar related to selecting the endDate. This is done by adding the following code below the div element related to the startDate calendar:

```
<!-- Here we define the HTML code for the End Date calendar -->
<div class="form-group">
    <div class="input-group">

        <!-- Here we define the input box for the End Date. We set the
         placeholder to the second element of the dateRange which is the end
         date of the current datapoints selected.-->
        <input #dtg2 class="form-control datepicker" placeholder={{dateRange[1]}}
        name="dp2" [(ngModel)]="endDate" ngbDatepicker #d2="ngbDatepicker"
        (click)="d2.toggle()" type="text" [dayTemplate]="t1"
        (dateSelect)="endDateChanged($event)" autocomplete="off" />

        <!-- Here we define the search button which when clicked will trigger
        the function:"sendDTG()"-->
        <button mat-raised-button (click)="sendDTG()"
        class="btn btn-white btn-round btn-just-icon">
            <!--Here we set the icon of the search button to a magnifying glass. -->
            <i class="material-icons">search</i>
        </button>

        <!-- Here we add the Angular template which is used to create the
        NgbCalendar filled with dates to select. We set the selectable date
        to the end date of the currently selected datapoints so no dates above
        that date be selected.-->
        <ng-template #t1 let-date let-focused="focused">
            <span class="custom-day" [class.text-white]="isActive(date)"
             [class.range-end]="isRangeEnd(date)" [class.range-start]="isRangeStart(date)"
             [class.btn-light]="isActive(date)" [class.bg-primary]="isActive(date)"
             [class.range]="isInRange(date)" [class.faded]="isInRange(date)">
           {{ date.day }}
         </span>
        </ng-template>
    </div>
</div>
```

Now that we have the business logic and the layout logic of the DatePickerComponent we need to do one more thing before we can put it to use. We need to add the DatePickerComponent to the app.module.ts file. So let's open this file an add the following module import below the import of the MapComponent:

```
/*Here we import the DatePickerComponent which is our DTG Picker and will be added
to the declarations section in this file.*/
import { DatePickerComponent } from 'src/app/components/datepicker/datepicker.component';
```

The last thing we need to do is adding the DatePickerComponent to the declarations section of the app.module.ts file as shown in the illustration below:

```
declarations: [
  AppComponent,
  SidebarComponent,
  NavbarComponent,
  BaseComponent,
  MapComponent
],
```

```
declarations: [
  AppComponent,
  SidebarComponent,
  NavbarComponent,
  BaseComponent,
  MapComponent,
  DatePickerComponent
],
```

Now that we can use the DatePickerComponent throughout our application, we need to add it to our MapComponent file. So let's open the map.component.ts file.

The first thing we need do is adding a global variable called: "dateRange". This variable will be used to set the default Start and EndDate of the DatePickerComponent to 0. This is done by adding the following line of code below the global variable: "layerStyles":

```
/*
Here we create a global variable called: "dateRange". The value of this
variable is passed to the DateTimePicker Component. The default value is
0 but will change when an item is selected.
*/
public dateRange: any = [0, 0];
```

Now we need to add the function which is used to obtain the dateEvent from the DatePickerComponent. This function will be called: "getDTGEvent()" and takes an itemId and the the emitted event (noted as $event) as input parameters. This function will be assigned to the DatePickerCompontent in the map.component.html file. We are going to add the DatePickerComponent to our HTML file later.

Let's first add the function: "getDTGEvent()" to our map.component.ts file. The following steps are executed in this function:

1) The start date (send by the DatePickerComponent) is extracted and transformed in a format which is understandable for the MongoDB query that will be triggered in our Flask-API.
2) The end date (send by the DatePickerComponent) is extracted and transformed in the same manner as the startDate.
3) The function: "getItemDataByDTG()" is triggerd in which we pass the activeItem, the transformed startDate and the transformed endDate as parameters. The function: "getItemDataByDTG()" will then use the Start and EndDate to obtain the data from the MongoDB datastore.

We create this function by adding the following code below the function:"getItemDataByDTG()":

```
getDTGEvent(id: string, $event): void {

  // Here we obtain and transform the start date.
  let dtg_s = $event[0].year + '-' + $event[0].month + '-' + $event[0].day

  // Here we obtain and transform the end date.
  let dtg_e = $event[1].year + '-' + $event[1].month + '-' + $event[1].day

  // Here we trigger the function: "getItemDataByDTG".
  this.getItemDataByDTG(this.activeItem, dtg_s, dtg_e)
};
```

The last thing we need to do, before we can start adding the HTML code for the DTG selection to our MapComponent layout, is editing a function which we already created.

The function we need to edit is called: “selectItem()”. Go to the function and add the following line of code above the line:”this.setStaticOverlays(item)”.

Note: The line “this.setStaticOverlays(item)” is also included in the illustration below. You do not have to add this line again!

```
// Here we make sure that the dateRange in the NgbCalendar is updated
// with the dateRange of the selectedItem.
this.dateRange = this.activeItem.dateRangeTotal;

// Here we create the new static overlays (start and end marker overlays)
this.setStaticOverlays(item)
```

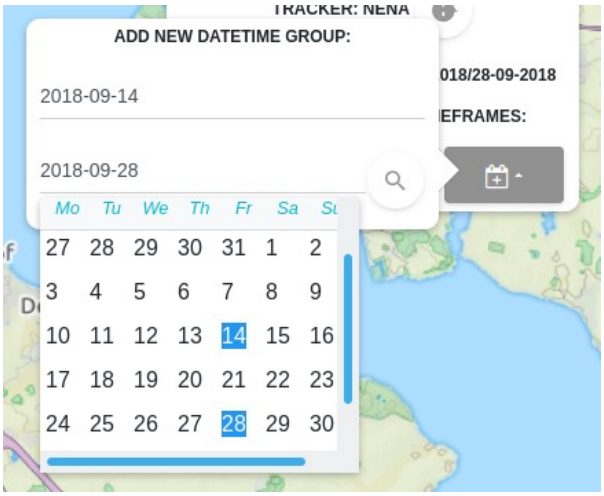
Now that we have all our business logic completed in order to select DateTimeGroups, we need to add some code to the MapComponent HTML page. So let’s open the file map.component.html and add the following code below the div element in which we defined the logic to display item information by clicking on the info icon next to the selectedItem’s name The div element with the id:”itemInfoSelection”):

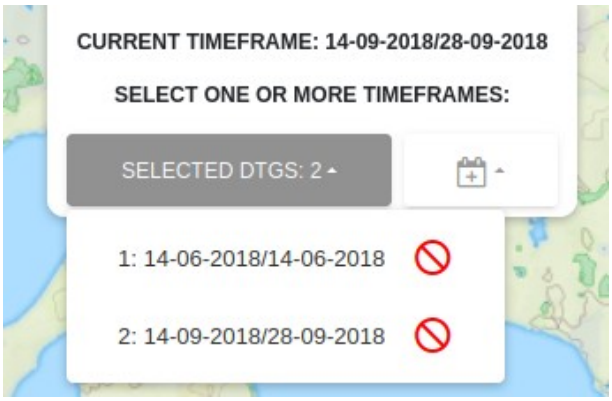
NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
<!--
In this div element we add the logic related to switching between layergroups
and selecting multiple timeframes
-->
<div id="dateSelection" style="height: 100px;">
   <!-- Here we add the text which is used to display the timeframe which is currently active -->
   <li class="header-title">Current timeframe: {{activeItem.dateRangeSelected}}</li>
   <!-- Here we add the text which is used to tell the user that he can add extra timeframes -->
   <li class="header-title">Select one or more timeframes:</li>
   <!-- Here we add an ngbDropdown element -->
   <div ngbDropdown class="pull-left">
      <!-- Here we add the button which is used to open the selected
         layerGroups dropdown-box and switch between layerGroups -->
      <button class="btn btn-white btn-block" ngbDropdownToggle>
      Selected DTGs: {{activeItem.layerGroups.size}}</button>
      <!-- Here we add theh popup that opens when the dropdown button is clicked related to
         switching between selected timeframes. All the code inside this div element is shown
         in the popup-->
      <div ngbDropdownMenu>
         <!-- Here we add a for loop which makes sure that all the selectedDTG's (layerGroups)
            are added in the dropdown box -->
         <ng-container *ngFor="let group of activeItem.layerGroups | keyvalue; let i = index">
            <!-- Here we add a button for each of the selected layerGroups
               if a layerGroup is clicked the function: "setLayerGroup()" is triggerd to make
               the clicked layergroup the activeLayerGroup -->
            <button ngbDropdownItem (click)="setLayerGroup(group.key)">
               {{i+1}}: {{group.key}}
               <!-- Here we add the icon which is used to remove a layerGroup. -->
               <a class="material-icons" style="color: red; margin-left: 15px;"
                  (click)='removeLayerGroup(group.key)'>not_interested</a>
            </button>
         </ng-container>
```

```
        </ng-container>
      </div>
    </div>

    <!-- Here we add another ngbDropdown which is used to select new DTG's -->
    <div ngbDropdown id="main-dropdown"  [autoClose]="false" class="pull-right">
      <!-- Here we add a button which is used to open the date selection menu -->
      <button class="btn btn-white" ngbDropdownToggle>
      <i class="fa fa-calendar-plus-o" aria-hidden="true"></i>
      </button>

      <!-- Here we add the popup which will open when the calendar icon is clicked -->
      <div id="main-dropdown-menu" ngbDropdownMenu >
        <!-- Here we add te text displayed in the popup -->
        <li class="header-title">Add new datetime group:</li>
        <!-- Here we add the DateTimePicker component
        we assign the function: "getDTGEvent()" to th component -->
        <dtgpicker [dateRange]="dateRange" (dateEvent)="getDTGEvent(activeItem.id,$event);">
        </dtgpicker>
      </div>
    </div>
</div>
```

That's it! Now when you reload the page you will be able to select multiple timeframes and switch between the selected timeframes of a selected item as shown in the illustration below:





As you may have noticed you can not remove a selected timeframe yet. How this is done is shown in the next section.

4.2.10 Removing LayerGroups

Now that we are able to select multiple LayerGroups we also want to make sure we can remove the selected LayerGroups. This section will show you how to add this functionality. Do be able to remove LayerGroups we need to add a function called: “removeLayerGroup” and takes in a layerGroupKey as input parameter. The function is triggered when the red button next to a selectedLayerGroup is clicked. The following steps are executed when the function is triggered:

1) This function first clears the running animation, which will only happen if an animation is running (We will be adding this later since we did not add the animation functionality yet).
2) Then the function will remove each layer in the layerGroup from the map using a for loop that loops through all the layers in each layergroup.
3) After the layers of the layerGroup are removed the layerGroup itself is removed from the JavaScriptMap called: "layerGroups".
4) Finally a check is performed to see if the layerGroup that is removed was the last layerGroup in layerGroups JavaScriptMap. If this is the case, the function: removeItem() is called since we want to remove the item which does not have any selectedLayerGroups. If this is NOT the case the next layerGroup in the layerGroups JavaScriptMap will become the active layerGroup.

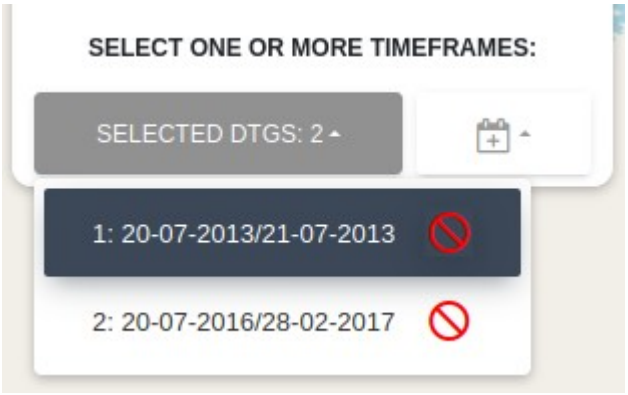
Now that you know what the function: “removeLayerGroup()” is going to do, we should add it to our application. This is done by adding the following code below the function: “getDTGEvent()”:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
removeLayerGroup(layerGroupKey: string): void {

  // Here we assign the activeItem to a variable called: "item"
  let item = this.activeItem;

  // Here we obtain the layerGroup which has to be removed from the
  // layerGroups JavaScriptMap using the layerGroupKey passed as parameter
  // on the function call. We assign the layerGroup to a variable called:
  // "groupToRemove".
  let groupToRemove = item.layerGroups.get(layerGroupKey)

  // Here we loop through all the entries in the layerGroup. We do this to
  // remove all the layers, belonging to the layerGroup to remove, from the
  // OpenLayers Map. We do this by calling the functio:".removeLayer" on the
  // OpenLayers Map instance and passing the value of the layers (pointLayer,
  // lineLayer and markerLayer).
  for (let [key, value] of Object.entries(groupToRemove)) {
    this.map.removeLayer(value['layer'])
  }

  // Here we remove the layerGroup from the JavaScriptMap containing the
  // selected layerGroups. We do this by calling the: ".delete()" function
  // on the JavaScriptMap and passing the layerGroupKey as parameter.
  item.layerGroups.delete(layerGroupKey)
```

```
  item.layerGroups.delete(layerGroupKey)

  // Here a check if performed to see if the size of the layerGroups JavaScriptMap
  // is bigger than 0.
  //
  // If this is the case, it means that there is/are one or more other selected
  // layerGroups, so one of these will be set as activeLayerGroup using the
  // function: "setLayerGroup" and passing the value of the key
  // (the layerGroupSelector) as parameter.
  //
  // If this is NOT the case, it means that there ar NO other layerGroups selected
  // so the activeItem needs to be removed. This is done by calling the function:
  // "removeItem" and passing the activeItem as parameter.
  item.layerGroups.size > 0 ? this.setLayerGroup(item.layerGroups.keys().next().value) :
    this.removeItem(item)

};
```

That's it! Now you are able to remove LayerGroups by clicking on the red icon next to a layergroup entry as shown in the illustration below:



Now that we have completed the basic functionalities of our application we should add some extra options from which you can choose to select datapoints per amount or datapoints in a certain country. This will be done in the following sections.

4.2.11 Adding Amount selection

To be able to select a desired amount of datapoints from a selected item, we need to add a function called: “getItemDataByAmount()” to our MapComponent. This function is triggered when a amount is selected from the dropdown list related to the amount selection (which is defined in the HTML page).

This function contains a switch/case. The switch case takes the itemType, which in our case can be a tracker as input. Depending on the itemType, the corresponding function is triggered.

The function that is triggered triggers a function in the service related to the item which is passed in the function: "getItemDataByAmount()".

The data obtained from the function which was triggered in the service, will then be passed as parameter in the function: "loadItemData()". The function loadItemData() will then assign the returned data to the item which is passed as parameter in this function.

Now that you know what the getItemDataAmount() function does, we should create it. This is done by adding the following code below the function: “removeLayerGroup()”:

```
getItemDataByAmount(item: Item, amount): void {
  switch (item.type) {
    case 'tracker':
      this._CraneService.getTransmissionsAmount(item.id, amount).subscribe(
        (transmissions) => {
          this.loadItemData(transmissions)
        }
      )
      break;
    default:
      break;
  };
};
```

Now we need to add a drop-down box which is used to trigger this function. This is done by adding some code to the MapComponent Layout page. So let’s open the map.component.html file.

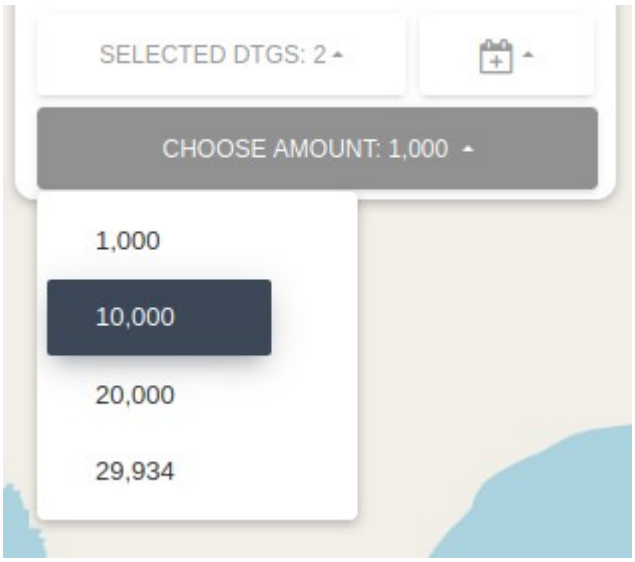
We are going to add the drop-down list by adding the following code below the div element related to the DTG selection:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
<!--
In this div element we add the logic related to selecting a N amount of datarows from the datastore
-->
<div id="amountSelection">
      <div ngbDropdown>
        <!--
        Below we add a dropdown toggle which has the length of the coordinateList of the activeItem
        as text.
        -->
          <button class="btn btn-white btn-block" ngbDropdownToggle>
            Choose amount: {{ activeItem.coordinateList.length | number: '2.'}}
          </button>
```

```
        </button>
            <!--
            Here we add the menu that opens when the dropdown button is clicked.
            In this menu we add the amounts: 1000, 10000, 20000 and the total amount of
            datapoints belonging to the activeItem.

            We use the syntax : {{1000| number: '2.'}} to make sure a comma is added to the number.
            -->
            <div ngbDropdownMenu>
                <!-- Here we add the entry for 1000 datapoints -->
                <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,1000)'>
                  {{1000| number: '2.'}}
                </button>
                <!-- Here we add the entry for 10000 datapoints -->
                <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,10000)'>
                  {{10000| number: '2.'}}
                </button>
                <!-- Here we add the entry for 20000 datapoints -->
                <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,20000)'>
                  {{20000| number: '2.'}}
                </button>
                <!-- Here we add the entry for the total amount of datapoints available-->
                <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,activeItem.totalDataLength)'>
                  {{activeItem.totalDataLength| number: '2.'}}
                </button>
            </div>
    </div>
</div>
```

That's it! Now when you reload the application and select an item you can choose the amount of datapoints you want to visualize as shown in the illustration below:



Now we want to do the same for selecting datapoints in a give Polygon (The polygons of some predefined countries in our case). How this is done is shown in the next section.

4.2.12 Adding Country selection

To be able to select datapoints in a given polygon (predefined polygons of multiple countries in our case), we first need to define a global variable called: “countryList”. This global variable will have the type JavaScriptMap. This means that the variable is a set of key/values. The keys in this case are the country names and the values are the polygon coordinates belonging to that country.

The drop down box related to the country selection will be populated with the keys of the JavaScriptMap.

During this programming manual we will be adding 2 countries and their polygon coordinates. These countries are the Netherlands and Spain. The coordinates of the polygons, representing the countries were obtained using the following website:

https://www.keene.edu/campus/maps/tool/

So let’s add the global variable: “countryList” below the global variable: “dateRange” which we created earlier and above the MapComponent class constructor. The following code should be added:

```
public countryList: Map < string, Number[][] > = new Map([
  ["Spain", [
    [-10.6347656, 44.3081267],
    [-11.0961914, 36.3859128],
    [1.9995117, 36.7740925],
    [3.4716797, 43.7869584],
    [-10.6347656, 44.3081267]
  ]],
  ["Netherlands", [
    [3.1750488, 53.6055441],
    [2.9992676, 50.8336977],
    [6.3391113, 50.7086344],
    [7.3168945, 53.5794615],
    [3.1750488, 53.6055441]
  ]],
]);
```

As you can see we have added 2 entries (Spain and Netherlands) behind each entry we added the Polygon coordinate points belonging to that country. Now we need to add a function called: “getItemDataByCountry()” which takes an Item and the value belonging to the selected key in the JavaScriptMap: “countryList”, as input parameters. This key will be selected using a drop-down box in the application which we are going to add later.

Let’s first create the function:”getItemDataByCountry()”. This function is triggered when a country is selected from the dropdown list related to the countrySelection.

The coordinates of the polygon related to that country (defined in the global JavaScriptMap countryList) are than passed to this function.

This function contains a switch/case. The switch case takes the itemType, which in our case can be a tracker, as input. Depending on the itemType, the corresponding function is triggered.

These functions trigger a function in the service related to the item which is passed in the function: "getItemDataByCountry()".

The data obtained from the function which was triggered in the service, will then be passed as parameter in the function: "loadItemData()". The function loadItemData() will then assign the returned data to the item which is passed as parameter in this function.

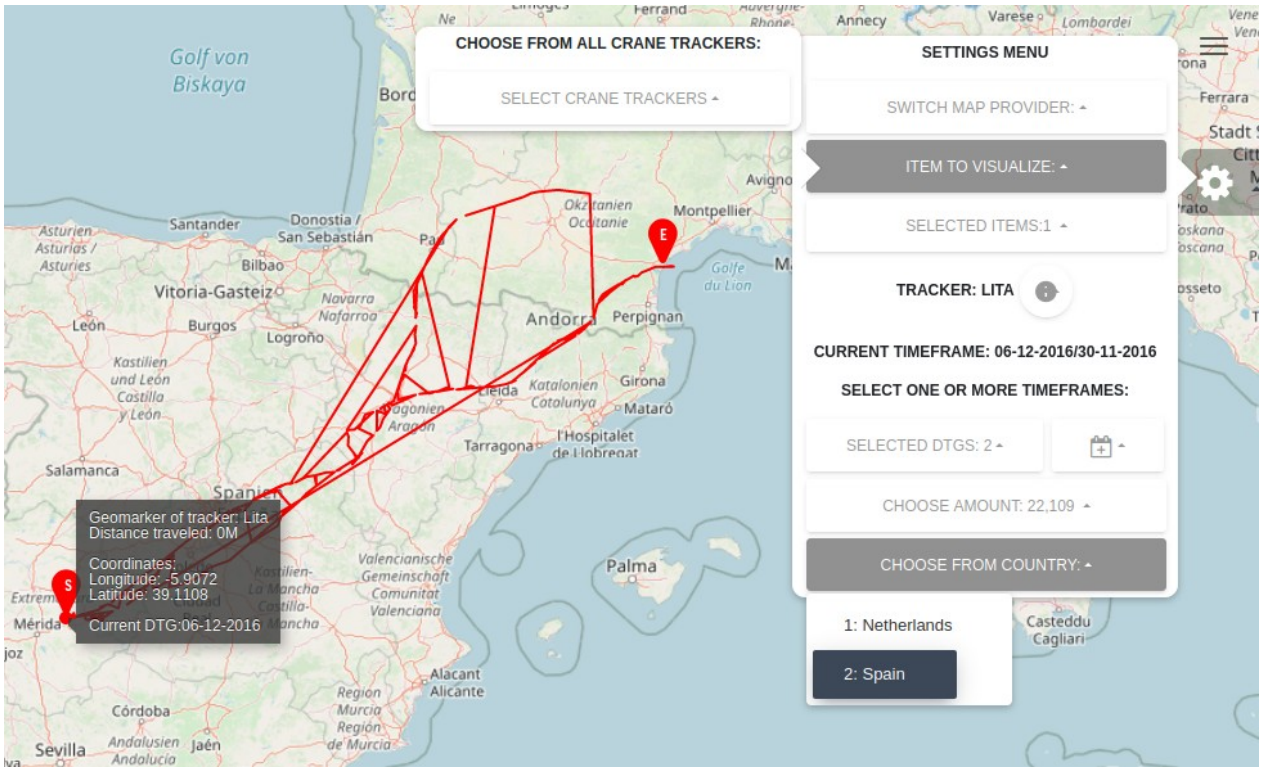
Let's add the function by adding the following code below the function:"getItemDataAmount()" which we defined earlier:

```
getItemDataByCountry(item: Item, coords: Number[][]): void {
  switch (item.type) {
    case 'tracker':
      this._CraneService.getTransmissionsCountry(item.id, coords).subscribe(
        (transmissions) => {
          this.loadItemData(transmissions)
        }
      )
      break;
    default:
      break;
  };
};
```

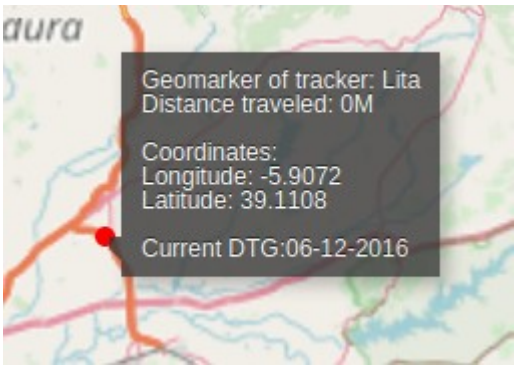
Now we need to add the HTML code related to the country selection to the layout of the MapComponent. So let's open the map.component.html file and add the following below the div element related to the amount selection (The div element with the id:"amountSelection"):

```
<div id="countrySelection">
   <!-- Here we create a ngbDropdown related to selecting countries -->
   <div ngbDropdown>
      <!-- Here we add the dropdown button which is used to open the dropdown list -->
      <button class="btn btn-white btn-block" ngbDropdownToggle>Choose from Country:</button>
      <div ngbDropdownMenu>
         <!-- Here we add a forEach loop which loops trough all the entries in the JavaScriptMap:
            "countryList".-->
         <ng-container *ngFor="let country of countryList | keyvalue; let i = index;">
            <!-- For each entry in the countryList a button is created.
               We assign the key of the entry (the country name) as text to the button.
               We also assign the function: "getItemDataByCountry()" to the button in which we pass
               the activeItem and the set of coordinates (the polygon) belonging to the clicked country. -->
            <button ngbDropdownItem
               (click)='getItemDataByCountry(activeItem,country.value)'>{{i + 1}}: {{country.key}}
            </button>
         </ng-container>
      </div>
   </div>
</div>
```

That’s it! Now when you reload the application and select an item you are able to select all the datapoints in Spain and in the Netherlands as shown in the illustration below:



As you may have noticed, each time you remove a selected item, some objects on the map remain as shown in the illustration below:



This is because when an item is removed the overlays are not toggled off. In the next section you will learn how to create the layer toggling functionality which will also solve this problem.

4.2.13 Adding Layer and Overlay Toggling

To be able to toggle layers such as the MarkerLayer, PointLayer, LineLayer etc. we need to add a function called: “toggleLayer()”. This function will be assigned to some buttons in the HTML page of the map component. The function takes an layerType as input parameter. The layerType could for example be “LineLayer”.

The following steps are executed in this function:

1) The layer which needs to be toggled is obtained from the items activeLayerGroup. Since a layerGroups contains 3 layers (Point, Marker and Line layers) we can obtain the desired layer by passing the layerType.
2) A check is performed to determine whether the layer is visible or not. If the layer is visible it will be come unvisible. If the layer is unvisible it will become visible.

   We do this by using the build in OpenLayers method: ".setVisible()" and passing True or False as input parameter.

Now let’s add the function by adding the following code below the function: “getItemDataByCountry()”:

```
toggleLayer(layerType: string): void {

  // Here we obtain the layer which needs to be toggled from the
  // activeLayerGroup by passing the layerType.
  let layerToToggle = this.activeItem.activeLayerGroup[layerType].layer

  // Here we perform a check to determine whether the layer visibility is
  // set to True or False.
  layerToToggle.getVisible() == true ? layerToToggle.setVisible(false) :
    layerToToggle.setVisible(true)
};
```

Now that we added the business logic which is required to toggle layers on or off, we need to do the same for the overlays. This is done by adding a function called: “toggleOverlay()”. This function will also be assigned to multiple buttons in the HTML layout of the MapComponent. This function is used to toggle overlays such as the GeoMaker, GeoMarker infobox and the Start and endMarker information boxes.

The function takes an overlay type as input parameter which for example could be: “geomarkerInfo”. The overlayType passed as input parameter is used to determine which overlay has to be toggled on or off.

The following steps are executed when the function is triggered:

1) The overlay is obtained from the OpenLayers Map instance using the overlayType which was passed as input parameter. We do this by using the build in OpenLayers method: ".getOverlayById".
2) The activeItem is assigned to a variable called: "item".

3) A switch/case statement determines which code has to be executed depending on the overlayType which was passed as Input parameter.

   The following can happen in the switch/case:

   ➔ **If the overlayType is equal to "geomarkerInfo":**

   A check is performed to see if the position of the geomarkerInfo info box is equal to undefined. If this is the case the position of the geomarkerInfo overlay will be set to the coordinate on which the geomarker itself currently is. If the position of the overlay is not equal to undefined it means the overlay is going to be toggled. This is done by setting the position to undefined.

   ➔ **If the overlayType is equal to "startmarkerInfo":**

   A check is performed to see if the position of the startMarker info box is equal to undefined. If this is the case the position of the startMarkerInfo overlay will be set to the start coordinate of the visualized route. If the position of the overlay is not equal to undefined it means the overlay is going to be toggled. This is done by setting the position to undefined.

   ➔ **If the overlayType is equal to "endmarkerInfo":**

   A check is performed to see if the position of the startMarker info box is equal to undefined. If this is the case the position of the endMarkerInfo overlay will be set to the end coordinate of the visualized route. If the position of the overlay is not equal to undefined it means the overlay is going to be toggled. This is done by setting the position to undefined.

   ➔ **If the overlayType is equal to "all":**

   The positions of all the overlays will be set to undefined. This happens when the last selectedItem is removed by the user.

Now that you know what the function is supposed to do we can start coding the function. This is done by adding the following below the function: “toggleLayer()” which we created earlier:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

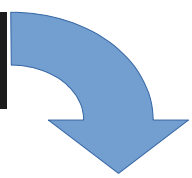
```
toggleOverlay(overlayType: string): void {

  // Here we obtain the overlay which needs to be toggled using the
  // overlayType.
  let overlay = this.map.getOverlayById(overlayType);

  // Here we assign the activeItem to a variable called item.
  let item = this.activeItem;

  // Here we create the switch/case to determine which code should be executed.
  switch (overlayType) {

    // Incase it's geomarkerInfo the following happens:
    case 'geomarkerInfo':
```

```
    case 'geomarkerInfo':
      // Here we perform the check to see if the geoMarkerInfo position is set
      // to undefined. If this is the case the position of the overlay is set
      // to the activeItem's currentCoordinate.
      overlay.getPosition() == undefined ? overlay.setPosition(
        item.coordinateList[item.currentCoordinateIndex]) :
        overlay.setPosition(undefined)
      break;
    // Incase it's startmarkerInfo the following happens:
    case 'startmarkerInfo':
      // Here we perform the check to see if the startMarkerInfo position is set
      // to undefined. If this is the case the position of the overlay is set
      // to the activeItem's startCoordinate.
      overlay.getPosition() == undefined ? overlay.setPosition(item.startCoordinate) :
        overlay.setPosition(undefined)
      break;
    // Incase it's endmarkerInfo the following happens:
    case 'endmarkerInfo':
      // Here we perform the check to see if the endMarkerInfo position is set
      // to undefined. If this is the case the position of the overlay is set
      // to the activeItem's endCoordinate.
      overlay.getPosition() == undefined ? overlay.setPosition(item.endCoordinate) :
        overlay.setPosition(undefined)
      break;
    // Incase it's all the following happens:
    case 'all':
      // Here we set the position of the geomarker to undefined.
      this.map.getOverlayById('geomarker').setPosition(undefined);
      // Here we set the position of the geoMarkerInfo to undefined.
      this.map.getOverlayById('geomarkerInfo').setPosition(undefined);
      // Here we set the position of the startMarkerInfo to undefined.
      this.map.getOverlayById('startmarkerInfo').setPosition(undefined);
      // Here we set the position of the endMarkerInfo to undefined.
      this.map.getOverlayById('endmarkerInfo').setPosition(undefined);
      break;
  };
};
```

The last thing we need to do before adding the HTML code to the MapComponent is updating the functions:"removeItem()" and "setLayerGroup()". We need to make sure that when an Item is removed or when a new layerGroup is set the overlays are also toggled off.

Let's first edit the function: "removeItem()". We need to edit the last line in the function as shown in the illustration below:

```
this.selectedItems.length == 0 ? null :
  this.setStaticOverlays(this.activeItem)
```



```
this.selectedItems.length == 0 ? this.toggleOverlay('all') :
  this.setStaticOverlays(this.activeItem)
```

Now let's edit the function: "setLayerGroup'. This is only necessary if you commented out the code described at the bottom of page 42. If this is the case you need to edit the code according to the illustrations below:

```
// Here we call the function toggleOverlay and pass "all" as parameter.
// This makes sure the old overlays are removed from the map.
//this.toggleOverlay("all");
```

```
// Here we call the function toggleOverlay and pass "all" as parameter.
// This makes sure the old overlays are removed from the map.
this.toggleOverlay("all");
```

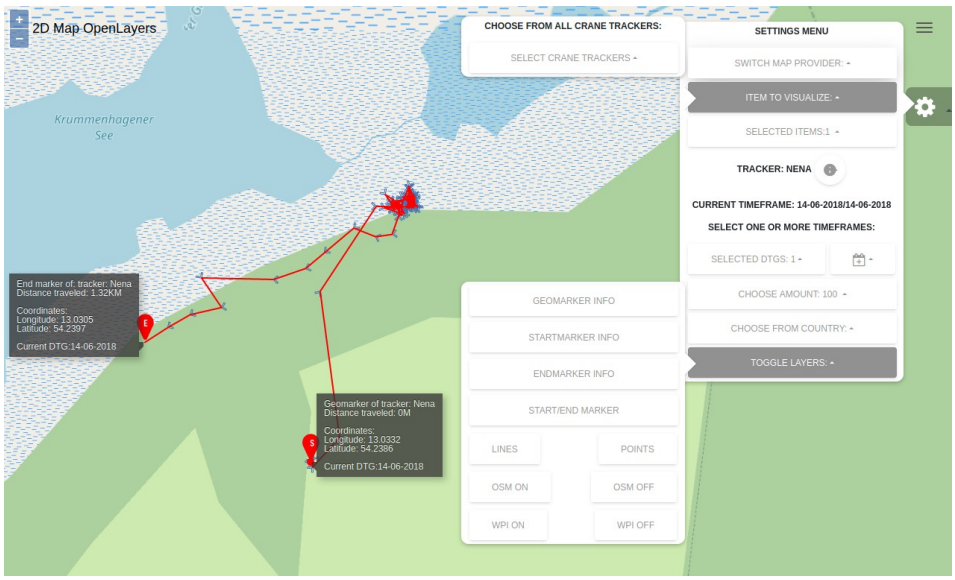
Now let's open the map.component.html file and add the following code below the div element which represents the country selection (the div element with the id:"countrySelection"):

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
<div id="toggleSelection">
   <!--
   Here we add the dropdown which is used to display the buttons used for toggling. We set
   autoClose to false to make sure the menu does not close when a setting is clicked.
   -->
    <div ngbDropdown id="main-dropdown"  [autoClose]="false">
        <!-- Here we create the button which is used to open the layer toggling menu. -->
        <button class="btn btn-white btn-block" ngbDropdownToggle>Toggle Layers:</button>
        <div id="main-dropdown-menu" ngbDropdownMenu>
            <!-- Here we add the button which is used to toggle the GeoMarker info on or off.
            we assign the function: "toggleOverlay()" to the button in which we pass the
            overlayType: "geomarkerInfo". -->
            <button class="btn btn-white btn-block" (click)='toggleOverlay("geomarkerInfo")'>
              Geomarker Info</button>
            <!-- Here we add the button which is used to toggle the Startmarker info on or off.
            we assign the function: "toggleOverlay()" to the button in which we pass the
            overlayType: "startmarkerInfo". -->
            <button class="btn btn-white btn-block" (click)='toggleOverlay("startmarkerInfo")'>
              Startmarker Info</button>
            <!-- Here we add the button which is used to toggle the Startmarker info on or off.
            we assign the function: "toggleOverlay()" to the button in which we pass the
            overlayType: "endmarkerInfo". -->
            <button class="btn btn-white btn-block" (click)='toggleOverlay("endmarkerInfo")'>
              Endmarker Info</button>
            <!-- Here we add the button which is used to toggle the Start and EndMarkers on or off.
            we assign the function: "toggleLayer()" to the button in which we pass the
            layerType: "markerLayer". -->
            <button class="btn btn-white btn-block" (click)='toggleLayer("markerLayer")'>
              Start/End marker</button>
            <!-- Here we add the button which is used to toggle the LineLayer on or off.
            we assign the function: "toggleLayer()" to the button in which we pass the
            layerType: "lineLayer". -->
            <button class="btn btn-white pull-left" (click)='toggleLayer("lineLayer")'>
              Lines </button>
            <!-- Here we add the button which is used to toggle the PointLayer on or off.
```

```
            <!-- Here we add the button which is used to toggle the PointLayer on or off.
            we assign the function: "toggleLayer()" to the button in which we pass the
            layerType: "pointLayer". -->
            <button class="btn btn-white pull-right" (click)='toggleLayer("pointLayer")'>
              Points </button>
            <!-- Here we add the button which is used to toggle the OpenSeaMap layer on.
            we assign the function: "map.addLayer()" to the button in which we pass the
            OpenSeaMap Layer instance. -->
            <button class="btn btn-white pull-left" (click)='map.addLayer(seaLayer)'>
              OSM ON</button>
            <!-- Here we add the button which is used to toggle the OpenSeaMap layer off.
            we assign the function: "map.removeLayer()" to the button in which we pass the
            OpenSeaMap Layer instance. -->
            <button class="btn btn-white pull-right" (click)='map.removeLayer(seaLayer)'>
              OSM OFF</button>
            <!-- Here we add the button which is used to toggle the World Port Index layer on.
            we assign the function: "map.addLayer()" to the button in which we pass the
            World Port Index Layer instance. -->
            <button class="btn btn-white pull-left" (click)='map.addLayer(portLayer)'>
              WPI ON</button>
            <!-- Here we add the button which is used to toggle the World Port Index layer off.
            we assign the function: "map.removeLayer()" to the button in which we pass the
            World Port Index Layer instance. -->
            <button class="btn btn-white pull-right" (click)='map.removeLayer(portLayer)'>
              WPI OFF</button>
        </div>
    </div>
</div>
```

That's it! Now when you refresh the application and select an Item, you are able to toggle the layers and overlays on or of as shown in the illustration below:

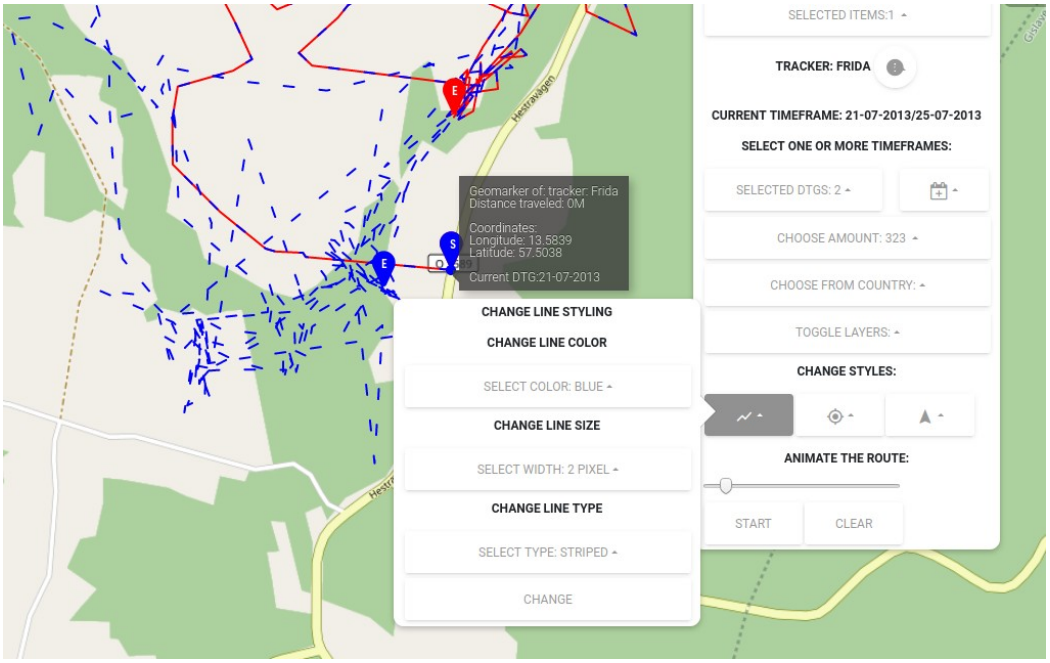


The buttons related to toggling the World Port Index layer on or off do not work yet. We will be adding this later in this document! Let's first add to functionality which enables us to change layer styling. This will be done in the next section.

This comes in handy when multiple layers are displayed on top of each other.

4.2.14 Changing layer styling

Being able to change the styling of certain layers comes in handy when multiple layers are displayed on top of each other. After finishing this section you will be able to edit the styling of each layerGroup separately as shown in the illustration below:



In this illustration you can see 2 selected timeframes belonging to the Crane: Frida. The datapoints of these selected timeframes overlap so the line style, line color and marker color of one of the selected timeframes are changed.

To be able to do this we first need to create some global variables. These global variables are as follows:

- ➔ colorList, which is a JavaScriptMap containing key/values. The keys are the color names and the values are the hash codes which represent the colors. The keys will be displayed in a drop-down box which the user will see in the application. When the user clicks on one of the keys (color names), the value (hex color code) will be passed in a function.
- ➔ WidthList, which is a also JavaScriptMap containing key/values. The keys are the pixel width names and the values are actual pixel values which represent the pixel names. The keys will be displayed in a drop-down box which the user will see in the application. When the user clicks on one of the keys (Pixel widths), the value (Pixel number) will be passed in a function.
- ➔ LineTypeList, which is a JavaScriptMap containing key/values. The keys are the line type names (striped, dotted, fluent) and the values are the numbers which represent the line types. The keys will be displayed in a drop-down box which the user will see in the application. When the user clicks on one of the keys (line type names), the value (numbers belonging to the selected line type) will be passed in a function.

In this manual we will be adding 5 colors, 4 width types and 3 line types. You can extend the available options where you see fit.

So let's start of with the global variable: "colorList". Adding this variable is done by adding the following code below the global variable: "countryList" and above the MapComponent class constructor:

```
public colorList: Map < string, string > = new Map([
  ["Blue", "#0000ff"],
  ["Purple", "#FF33F9"],
  ["Red", "#ff0000"],
  ["LightBlue", "#3377FF"],
  ["Yellow", "#ffbf00"]
]);
```

Now let's do the same for the widthList by adding the following below the global variable: "colorList":

```
public widthList: Map < string, number > = new Map([
  ["1 Pixel", 1],
  ["2 Pixel", 2],
  ["3 Pixel", 3],
  ["4 Pixel", 4]
]);
```

And last but not least let's add the lineTypeList by adding the following below the global variable: "widthList":

```
public lineTypeList: Map < string, number[] > = new Map([
  ["Dotted", [.1, 5]],
  ["Striped", [10, 25]],
  ["Fluent", [0, 0]]
]);
```

Now we need to add another global variable called StyleDict. This variable is a dictionary which will contain 4 entries: color, width, type and src. The values of these entries start of empty at first. When a user selects a certain color or width etc. the selected value will be assigned to the correct entry in the styleDict.

So let's add the global variable: "styleDict" by adding the following code below the global variable: "lineTypeList":

```
public styleDict: any = {
  'color': '',
  'width': '',
  'type': '',
  'src': ''
};
```

Now we need to add a function called: “setLayerStyle()”. This function is used to change the styling (color, width, type) of the MarkerLayers, PointLayers and LineLayers. The function takes in a layerType as input parameter.

The function is assigned to a button which is defined in the HTML page of the MapComponent.

The function contains a switch/case which is used to determine what layerStyle should be changed depending on the layerType.

The function creates a new OpenLayers Style object using the values which are set in the global variable: 'styleDict'. At first this dictionary is empty but when the user changes the styling of a layer, the selected values will be assigned to the correct entry in the styleDict.

The following steps are executed in this function:

1) A switch case is executed which does the following depending on the layerType which was passed as input parameter on the function call.

   ➔ **in case the layerType is equal to lineLayer:**

   A new OpenLayers LineStyle (stroke) is created using the values which are set in the styleDict (selected by the user in the application) as values for the: - width by calling this.styleDict['width'] - color by calling this.styleDict['color'] - lineType by calling this.styleDict['type']

   The linelayer object Style of the currently active layerGroup is then changed using the following syntax:

   activeLayerGroup.lineLayer[layer].getSource() to obtain the layer source.

   .getFeatures() to obtain the features in the lineLayer source.

   .setStyle() to change the styling of all the features. The newly created style is passed as parameter in the setStyle() function call.

   ➔ **in case the layerType is equal to markerLayer:**

   A new OpenLayers IconStyle (image) is created for both the Start and End Markers. We use the color set in the styleDict (selected by the user in the application) to specify the image src (Source). These images represent the pins which are found in the folder: assets/img/pins.

   We pass the selected color in the source link. For example:

   `assets/img/pins/pin_s_${this.styleDict.color}.png`

   would become the following if the user selected the color red in the application:

   `assets/img/pins/pin_s_Red.png`

   After the new styles have been created the overlay HTML element of the GeoMaker (the dot that moves along the visualized route when animating) is obtained. The color of the dot is set to color which was selected by the user.

   Finally the Start and End marker styles are updated using the same method as mentioned above when changing the lineLayer style. We pass the new marker styles as input parameter in the .setStyle() function.

➔ **in case the layerType is equal to pointLayer:**

At this point the functionality for changing the SVG icon representing the arrows has not been implemented. You could do this yourself if you would like.

Now that you know what the function is used for we can start coding it. This is done by adding the following below the function: “toggleOverlay()” which we created earlier:

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don’t need to add this line!

```
setLayerStyle(layerType: string): void {

  // Here we define the switch/case statement which determines what to do next
  // depending on the layerType which was passed on the function call.
  switch (layerType) {

    // The following code is executed when the layerType == lineLayer
    case 'lineLayer':

      // Here we create a new line style (stroke) and assign it to a variable
      // called: "newLineStyle"
      let newLineStyle = new ol.style.Style({
        stroke: new ol.style.Stroke({
          // Here we obtain the value of the widthList entry using the width
          // entry in the styleDict (which was set by the user).
          width: this.widthList.get(this.styleDict['width']),
          // Here we obtain the value of the colorList entry using the color
          // entry in the styleDict (which was set by the user).
          color: this.colorList.get(this.styleDict['color']),
          // Here we obtain the value of the lineTypeList entry using the type
          // entry in the styleDict (which was set by the user).
          lineDash: this.lineTypeList.get(this.styleDict['type'])
        }),
        // We set the zIndex of the layer to 3 to make sure the layer is shown
        // below the marker and point layer.
        zIndex: 3
      })

      // Here we update the activeLayerGroup LineLayer style using the newLineStyle.
      this.activeItem.activeLayerGroup['lineLayer']['layer'].getSource()
      .getFeatures()[0].setStyle(newLineStyle)

      break;

    // The following code is executed when the layerType == markerLayer
    case 'markerLayer':

      // Here we create a new Startmarker style (image) and assign it to a variable
```

```
      // Here we create a new Startmarker style (image) and assign it to a variable
      // called: "newStartMarkerStyle"
      let newStartMarkerStyle = new ol.style.Style({
        image: new ol.style.Icon({
          // Here we set the amount of Pixels on which the icon should be displayed.
          anchor: [0.5, 1],
          // Here we set the source location of the pin using the color value in
          // the styleDict.
          src: `assets/img/pins/pin_s_${this.styleDict.color}.png`
        }),
        // We set the zIndex of the layer to 5 to make sure the layer is shown
        // above the LineLayer.
        zIndex: 5
      })

      // Here we create a new Endmarker style (image) and assign it to a variable
      // called: "newEndMarkerStyle"
      let newEndMarkerStyle = new ol.style.Style({
        image: new ol.style.Icon({
          // Here we set the amount of Pixels on which the icon should be displayed.
          anchor: [0.5, 1],
          // Here we set the source location of the pin using the color value in
          // the styleDict.
          src: `assets/img/pins/pin_e_${this.styleDict.color}.png`
        }),
        // We set the zIndex of the layer to 5 to make sure the layer is shown
        // above the LineLayer.
        zIndex: 5
      })

      // Here we set the div element styling of the GeoMaker to have the color
      // which was set by the user.
      document.getElementById('geomarker').style["background-color"] = this.colorList
      .get(this.styleDict.color)

      // Here we update the endMarker styling using the newEndMarkerStyle.
      this.activeItem.activeLayerGroup['markerLayer']['layer'].getSource()
      .getFeatures()[1].setStyle(newEndMarkerStyle)

      // Here we update the startMarker styling using the newStartMarkerStyle.
      this.activeItem.activeLayerGroup['markerLayer']['layer'].getSource()
      .getFeatures()[0].setStyle(newStartMarkerStyle)

      break;

    // The following code is executed when the layerType == pointLayer.
    case 'pointLayer':
      break;
  }
};
```

That's it! Now we have to edit the HTML layout page to contain the code related to changing layer styles. So let's open the map.component.html file and add the following below the div element representing the layerToggling (the div element with the id:"toggleSelection"):

```
<!--
In this div element we add the logic related to changing layer styles
-->
<div id="styleSelection">
    <!-- Here we add the text which tells the user that he can change layer styling. -->
    <li class="header-title">Change Styles:</li>

</div>
```

Now let's add the the code related to editing the lineLayer styling. This is done by adding the following inside the styleSelection div element and below the Title: "Change Layer Styles":

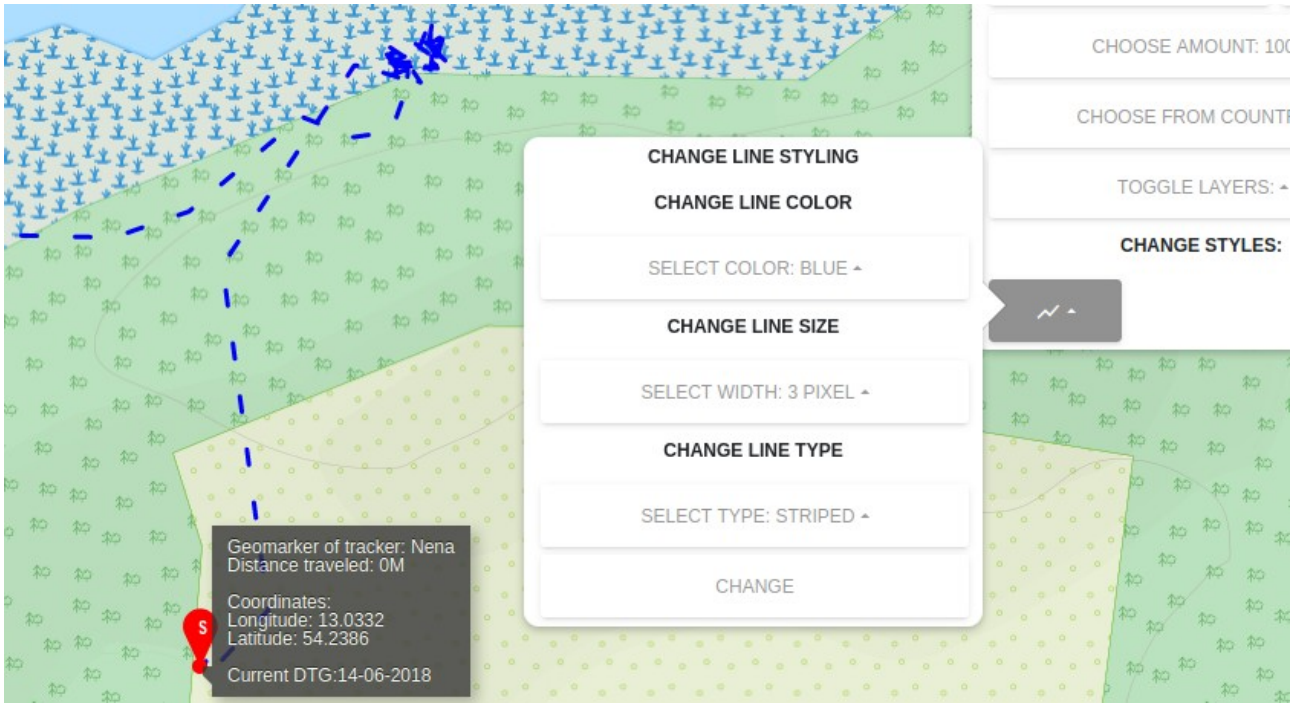
NOTE: This code is described using 3 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
<!-- Here we create the ngbDropdown related to changing the LineLayer styling -->
<!-- We set autoClose to false to make sure the popup does not automatically close. -->
<!-- We set the class to pull-left to make sure the buttons can be placed next to eachother. -->
<div ngbDropdown id="main-dropdown" [autoClose]="false" class="pull-left">
    <!-- Here we add the button which is used to open the lineLayer styling menu -->
    <!-- We set the icon of the button to a chart line icon. -->
    <button class="btn btn-white" ngbDropdownToggle><i id="lineColorButton"
      class="material-icons">show_chart</i></button>
    <!-- Here we define the popup menu which shows up when the toggle button
     is clicked. All the content in this div element is shown in the popup. -->
    <div id="main-dropdown-menu"  ngbDropdownMenu>
        <!-- Here we add the text which tells the user that he is currently
        editing the lineLayer Styling -->
        <li class="header-title">Change Line styling</li>
        <!-- Here we add the text which tells the user that he can change the line
        color using the button below -->
        <li class="header-title">Change Line color</li>
        <!-- Here we add the dropdown menu which is used to select a color -->
        <div ngbDropdown>
            <!-- Here we add the button which is used to open the color selection
            dropdown menu.The text of the button is set to the color which is currently
            active (set in the styleDict) -->
            <button class="btn btn-white btn-block" ngbDropdownToggle>
              Select color: {{styleDict.color}}</button>
            <!-- Here we define the dropdown menu which pops up when the color selection
            button is clicked. -->
            <div ngbDropdownMenu>
                <!-- Here we define a for each loop which loops through the colorList  -->
                <!-- For each entry in the list the following code is executed. -->
                <ng-container *ngFor="let color of colorList | keyvalue">
                    <!-- For each color in the colorList a button is created -->
                    <!-- The text of the button is the key of the entry in the
                    JavaScriptMap (so the color name) When one of the buttons is clicked the
                    value belonging to the key that was clicked is set in the styleDict -->
                    <button ngbDropdownItem (click)='styleDict.color = color.key'>
                      {{color.key}}</button>
                </ng-container>
```

```
        </ng-container>
    </div>
</div>
<!-- Here we add the text which tells the user that he can change the line width
using the button below -->
<li class="header-title">Change Line size</li>
<div ngbDropdown>
    <!-- Here we add the button which is used to open the width selection
    dropdown menu. The text of the button is set to the width which is currently
    active (set in the styleDict) -->
    <button class="btn btn-white btn-block" ngbDropdownToggle>
      Select width:  {{styleDict.width}}</button>
    <!-- Here we define the dropdown menu which pops up when the width selection
     button is clicked. -->
    <div ngbDropdownMenu>
        <!-- Here we define a for each loop which loops through the widthList  -->
        <!-- For each entry in the list the following code is executed. -->
        <ng-container *ngFor="let width of widthList | keyvalue">
            <!-- For each width in the widthList a button is created -->
            <!-- The text of the button is the key of the entry in the
             JavaScriptMap (so the pixel amount) When one of the buttons is clicked
             the value belonging to the key that was clicked is set in the styleDict -->
            <button ngbDropdownItem (click)='styleDict.width = width.key'>
              {{width.key}}</button>
        </ng-container>
    </div>
</div>
<!-- Here we add the text which tells the user that he can change the type
 color using the button below -->
<li class="header-title">Change Line type</li>
<div ngbDropdown>
    <!-- Here we add the button which is used to open the width selection
    dropdown menu. The text of the button is set to the width which is currently
    active (set in the styleDict) -->
    <button class="btn btn-white btn-block" ngbDropdownToggle>
      Select type:  {{styleDict.type}}</button>
    <!-- Here we define the dropdown menu which pops up when the type selection
     button is clicked. -->
    <div ngbDropdownMenu>
        <!-- Here we define a for each loop which loops through the lineTypeList  -->
        <!-- For each entry in the list the following code is executed. -->
        <ng-container *ngFor="let type of lineTypeList| keyvalue">
            <!-- For each line type in the LineTypeList a button is created -->
            <!-- The text of the button is the key of the entry in the JavaScriptMap
             (so the type name) When one of the buttons is clicked the value
             belonging to the key that was clicked is set in the styleDict -->
            <button ngbDropdownItem (click)='styleDict.type = type.key'>
              {{type.key}}</button>
        </ng-container>
    </div>
</div>
```

```
        </div>
        <!-- Here we define the button which is used to update lineLayer style according
        to the values selected by the user.-->
        <!-- We assign the function: "setLayerStyle()" to the button in which we pass
        the layerType:"lineLayer" -->
        <button class="btn btn-white btn-block" (click)='setLayerStyle("lineLayer")'>
           CHANGE </button>
    </div>
</div>
```

Now when refresh the application and select an item we can change the lineLayer styling as shown in the illustration below:



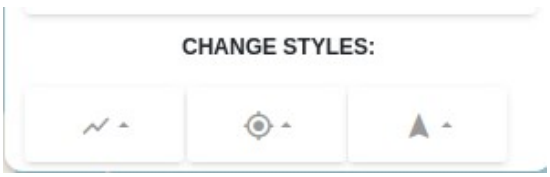
Now we want to do the same for the other layerTypes. This is done by adding the following code below the ngbDropdown related to changing the lineLayer styling:

NOTE: This code is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
<!-- Here we create the ngbDropdown related to changing the markerLayer styling -->
<!-- We set autoClose to false to make sure the popup does not
automatically close.We set the class to pull-left to make sure the buttons can
be placed next to eachother. -->
<div ngbDropdown id="main-dropdown" [autoClose]="false" class="pull-left">
    <!-- Here we add the button which is used to open the markerLayer styling menu -->
    <!-- We set the icon of the button to a GPS logo icon. -->
    <button class="btn btn-white" ngbDropdownToggle><i class="material-icons">
      gps_fixed</i></button>
    <!-- Here we define the popup menu which shows up when the toggle button is clicked. -->
    <!-- All the content in this div element is shown in the popup. -->
    <div id="main-dropdown-menu"  ngbDropdownMenu>
```

```
    <div id="main-dropdown-menu"  ngbDropdownMenu>
        <!-- Here we add the text which tells the user that he is currently
        editing the markerLayer Styling -->
        <li class="header-title">Change marker styling</li>
        <!-- The text of the button is set to the color which is currently
        active (set in the styleDict) -->
        <li class="header-title">Change marker color</li>
        <div ngbDropdown>
            <!-- Here we add the button which is used to open the color
            selection dropdown menu. The text of the button is set to the color
            which is currently active (set in the styleDict) -->
            <button class="btn btn-white btn-block" ngbDropdownToggle>
              Select color {{styleDict.color}}</button>
            <!-- Here we define the dropdown menu which pops up when the width
            selection button is clicked. -->
            <div ngbDropdownMenu>
                <!-- Here we define a for each loop which loops through the colorList  -->
                <!-- For each entry in the list the following code is executed. -->
                <ng-container *ngFor="let color of colorList | keyvalue">
                    <!-- For each line type in the LineTypeList a button is created -->
                    <!-- The text of the button is the key of the entry in the
                    JavaScriptMap (so the type name) When one of the buttons is clicked
                    the value belonging to the key that was clicked is set in the styleDict -->
                    <button ngbDropdownItem (click)='styleDict.color = color.key'>
                      {{color.key}}</button>
                </ng-container>
            </div>
        </div>
        <!-- Here we define the button which is used to update markerLayer style
        according to the values selected by the user.-->
        <!-- We assign the function: "setLayerStyle()" to the button in which we
        pass the layerType:"markerLayer" -->
        <button class="btn btn-white btn-block" (click)='setLayerStyle("markerLayer")'>
           CHANGE </button>
    </div>
</div>

<!-- Here we create the ngbDropdown related to changing the pointLayer styling -->
<!-- We set autoClose to false to make sure the popup does not automatically close. -->
<!-- We set the class to pull-left to make sure the buttons can be placed next to eachother. -->
<div ngbDropdown id="main-dropdown" [autoClose]="false" class="pull-left">
    <button class="btn btn-white" ngbDropdownToggle><i class="material-icons">
      navigation</i></button>
</div>
```

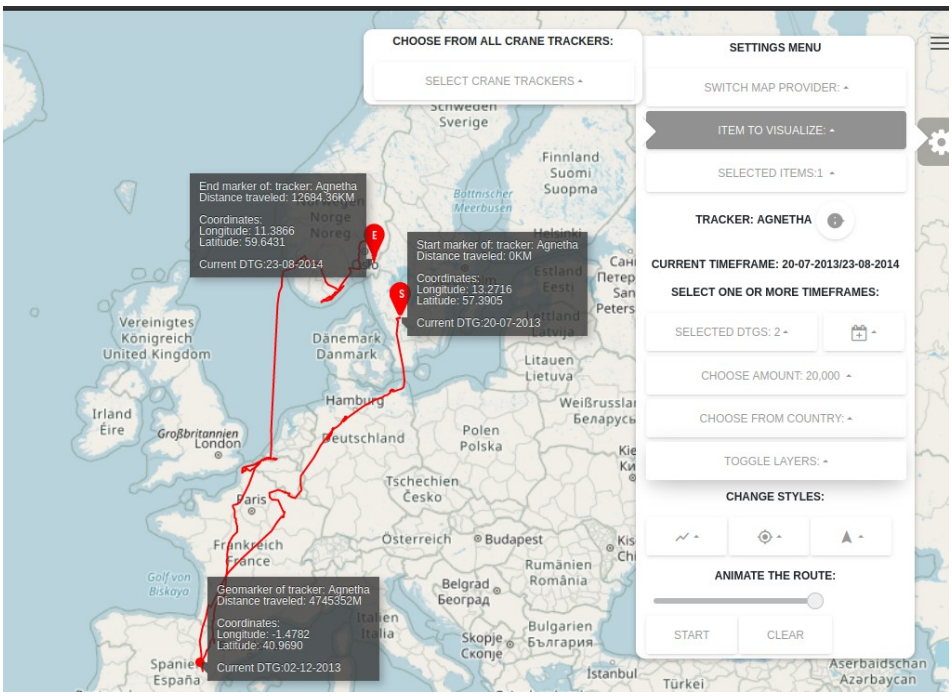
That’s it! Now you are able to edit the styling of the layers displayed on the map as shown in the illustration below:



In the next section you will learn how to animate a visualize route.

4.2.15 Animating routes

Now that we are able to visualize items on a map we also want to be able to animate the items which are visualized. After completing this section you will be able to start, pause and clear animations as shown in the illustration below:



We want to start of by creating a function which is called: “animateRoute()”. This function is used to animate a route which is visualized on the OpenLayers map. This function is bound to a button in the HTML layout page of the MapComponent.

This function contains 2 nested functions which are as follows:

- ➔ startAnimation(), which is only triggered if NO animation is running.
- ➔ pauseAnimation(), which is only triggered if AN animation is running.

When the button is clicked and the function is triggered the following steps are executed:

1) The value of the slider (related to the speed of the animation) is obtained from the HTML page.
2) A check is performed to see if the animation value of the activeItem is not equal to undefined.

   If this is not the case (so the value of the animation is equal to undefined) It means that no animation is running so that an animation has to be started. this is done by creating a JavaScript interval method and assigning the value of the interval to the activeItem's animation (so it's not undefined) anymore.

   The JavaScript interval method is used to trigger a function after a N amount of seconds / milliseconds. The N amount of seconds is the value of the speed slider which is set by the user in the application.

The function that is triggered in the JavaScript interval method is the nested function: "startAnimation()".

This function does the following when it's triggered:

1) Trigger the function: "setDynamicOverlays()" to make sure the GeoMaker and the GeoMaker information box are updated.

2) Increment the activeItem's currentCoordinateIndex by 3 to make sure that the coordinateList and datetimeList values are also updated. As mentioned before; using the JavaScript interval method will make sure that the function:"startAnimation()" is triggered multiple times. So each time the startAnimation() function is triggered, the currentCoordinateIndex will be incremented by +3 to make sure the geomarker moves and the geoMarkerInfo overlay also moves.

3) A check is performed to see if the activeItem's currentCoordinateIndex is lower than the length of the activeItem's coordinateList. If this is the case (so the currentCoordinateIndex is lower) nothing will happen and the function: "startAnimation()" will be triggered again using the JavaScript interval method.

   If this is not the case (so the currentCoordinateIndex is higher) it means that the animation reached the end of the visualized route so the animation has to be paused using the nested function: "pauseAnimation()". The activeItem's currentCoordinateIndex is also set to 0 to make sure the animation can start again if the user clicks on the start button again.

If the activeItem's animation value is no equal to undefined it means an animation is currently running (because an interval is assigned to the activeItem's animation) so the animation has to be paused. This is done by calling the function: "pauseAnimation()".

The function pauseAnimation() does the following when it's triggered:

1) Clear the JavaScript interval which was assigned to the activeItem's animation value.

2) Set the activeItem's animation value to undefined. This makes sure that the animation can be restarted if the user clicks on the start button again.

Now that you know what the function:"startAnimation()' we be used for we can start programming it. This is done by adding the following code below the function: "setLayerStyle()":

NOTE: This function is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
animateRoute(): void {

  // Here we assign the variable this to a variable: "_this".
  // This is required to access global variables in nested functions.
  let _this = this;

  // Here we obtain the value of the speed input slider which is defined in
  // the MapComponent HTML page.
  let speed = ( < HTMLInputElement > document.getElementById('speed')).value;

  // Here we perform a check to see if the activeItem's animation value
```

```
  // Here we perform a check to see if the activeItem's animation value
  // is not equal to undefined. If the value is not equal to undefined, the
  // function pauseAnimation is triggerd. If the value is equal to undefined
  // the JavaScript interval function is created in which we pass the function
  // startAnimation() and the value of the variable: "speed" as the amount
  // of miliseconds after which the function startAnimation() has to be
  // triggered.
  this.activeItem.animation != undefined ? pauseAnimation() :
    this.activeItem.animation = setInterval(function () {
      startAnimation();
    }, parseInt(speed));

  // Here we create the nested function: "startAnimation".
  function startAnimation() {

    // Here we set the dynamic overlays by passing the activeItem as input
    // parameter.
    _this.setDynamicOverlays(_this.activeItem)

    // Here we Increment the activeItem's currentCoordinateIndex by 3.
    _this.activeItem.currentCoordinateIndex += 3;

    // Here we perform a check to see if the currentCoordinateIndex value does
    // not exeed the length of the activeItem's coordinateList.
    _this.activeItem.currentCoordinateIndex < _this.activeItem.coordinateList.length ? null :
      (pauseAnimation(), _this.activeItem.currentCoordinateIndex = 0)
  };

  // Here we create the nested function: "pauseAnimation".
  function pauseAnimation() {

    // Here we clear the JavaScript interval by passing the activeItem's
    // animation value as input parameter.
    clearInterval(_this.activeItem.animation);

    // Here we set the activeItem's animation value to undefined.
    _this.activeItem.animation = undefined
  };
};
```

Now we need to add another function which is used to clear the animation if the user clicks on the clear button (which we will define later) or when the item from which the animation is running is removed. The function will be called:”clearAnimation()” and is used to clear an animation that is currently running. The function is triggered by clicking on the clear button defined in the HTML layout of the MapComponent or in the following functions:

1) setLayerGroup(), this is because when a new layerGroup is set, the running animation needs to be stopped. Otherwise the data (coordinateList etc.) from the new layerGroup will be used in the animation that is running.
2) removeItem(), this is because when an item is removed and an animation of that item is running, error's will occur since the data used by the animation is removed when removing the item.

3) RemoveLayerGroup(), this is because when a layerGroup is removed we also need to stop any running animations. If we don't do this the animation will use the data of the newly selected layerGroup which will cause error's.

When the function is triggered the following steps are executed:

1. The activeItem's currentCoordinateIndex is set to 0.
2. The JavaScript interval is cleared.
3. The activeItem's animation value is set to undefined.
4. The dynamic overlays are set to the new activeItem.

Now that you know what the function:"clearAnimation()' we be used for we can start programming it. This is done by adding the following code below the function: "animateRoute()":

```
clearAnimation(): void {

  // Here we set the activeItem's currentCoordinateIndex to 0.
  this.activeItem.currentCoordinateIndex = 0;

  // Here we clear the JavaScript interval by passing the activeItem's
  // animation value as input parameter.
  clearInterval(this.activeItem.animation);

  // Here we set the activeItem's animation value to undefined.
  this.activeItem.animation = undefined;

  // Here we set the dynamic overlays by passing the activeItem as input
  // parameter.
  this.setDynamicOverlays(this.activeItem)
};
```

Now we need to edit some existing functions to contain the clearAnimation() function. Let's start of by editing the function:"removeItem()". So lets scroll back to the function and edit the first line according to the illustrations below:

```
this.activeItem.id == item.id ? (
    this.selectItem(this.selectedItems.values().next().value)) :
  null;
```

```
this.activeItem.id == item.id ? (this.clearAnimation(),
    this.selectItem(this.selectedItems.values().next().value)) :
  null;
```

Now we need to edit the function: “setLayerGroup()’. This is only necessary if you commented out the lines described at the bottom of page 42. If this is the case you only need to remove the “//” in front of the line: this.clearAnimation() as shown in the illustration below:

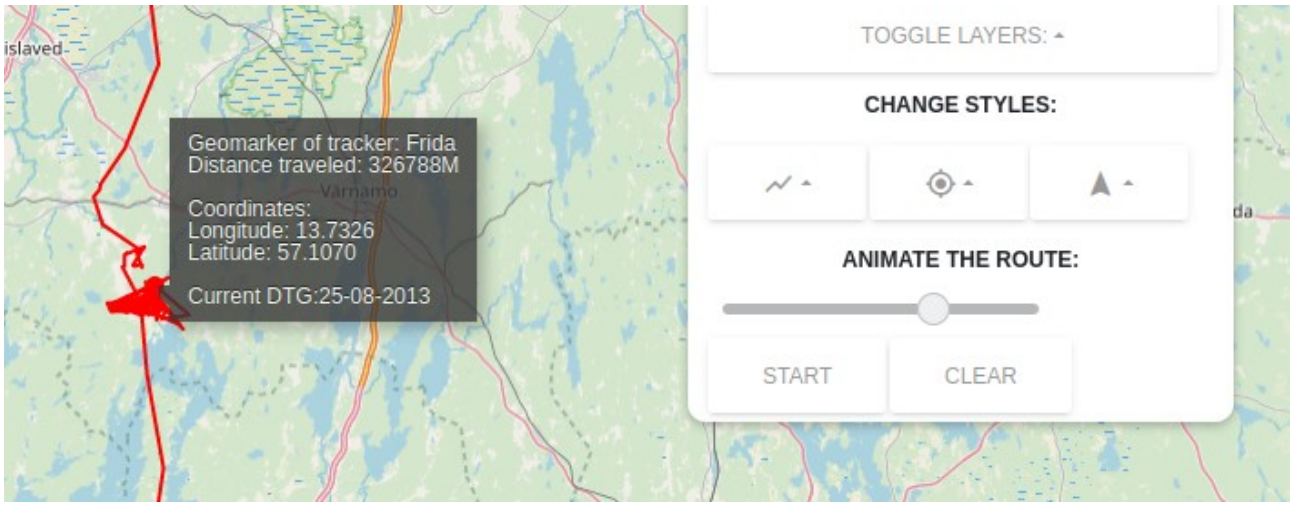
The last function we need to edit is the function: “removeLayerGroup()”. So lets scroll back to this function and add the following line at the top of the function below the line: let item = this.activeItem as shown in the illustration below. **Note: the line: let item = this.activeItem is also included in the illustration, you don’t need to add it again.**

```
// Here we assign the activeItem to a variable called: "item"
let item = this.activeItem;

// Here we clear the animation if any is running.
this.clearAnimation()
```

The last thing we need to do before we can animate routes, is adding the HTML layout code. So let’s open the map.component.html file and add the following below the div element related to changing the layer styles (the div element with the id:”styleSelection”):

```
<div id="animationSelection"  style="margin-top:55px;">
    <!-- Here we add the title which tells the user that he can animate the route. -->
    <li class="header-title"> Animate the route: </li>
    <!-- Here we add the speed slider which is used to set the speed in which
    the route has to be played -->
    <div class="input-group no-border">
        <!-- Here we define the slider. We set the id to speed, the min value to -100,
         the max value to 200, the increment steps to 10 and the start value to 100.
         we also set the direction to RTL which makes sure the steps are inverted. -->
        <input #speed id="speed" type="range" min="-100" max="200"
        step="10" value="100" style="direction: rtl;" />
    </div>
    <!-- Here we define the start button. If the value of the activeItem's animation
    is equal to undefined, this button is shown. We assign the function
    animateRoute() to the button. -->
    <div *ngIf="activeItem.animation == undefined">
        <button class="btn btn-white pull-left" (click)='animateRoute();'>Start</button>
    </div>
    <!-- Here we define the pause button. If the value of the activeItem's animation
    is not equal to undefined, this button is shown. We assign the function
    animateRoute() to the button. -->
    <div *ngIf="activeItem.animation != undefined">
        <button class="btn btn-white pull-left" (click)='animateRoute();'>Pause</button>
    </div>
    <!-- Here we define the button which is used to clear any running animations. We
    assign the function: "clearAnimation" to this button. -->
    <button class="btn btn-white pull-left" (click)='clearAnimation()'>Clear</button>
</div>
```

That’s it! Now when you reload the application and select an item, you can animate the route that is visualized as shown in the illustration below:



Now that we are able to visualize all the 2D data we also want to be able to see the elevation data from our visualized routes. This is done by using an elevation profile as shown in the illustration below:

In the next section you will learn how to create such a profile.

4.2.16 Creating an elevation profile

To create an elevation profile which shows the elevation data of a visualized route, we first need to add 2 global variables which are as follows:

- ➔ elevationProfile, which will be the global variable to which we assign the instance of the Elevation profile graph after it has been created.
- ➔ ElevationProfileOpen, which will be the global variable to which we assign a value which is used to check if the elevation profile is open or not (toggled on or of). We do this since we don't want to load any elevation data when the elevation profile is not opened.

So let's add the global variables by adding the following code below the global variable: "styleDict" and above the MapComponent class constructor:

```
/*
Here we create a global variable called: "elevationProfile".
After creating an instance of the elevation profile, this instance will
be assigned to this variable.*/
public elevationProfile: any;

/*
Here we create a global variable called: "elevationProfileOpen". The value
of this variable will either be TRUE or FALSE depending on whether the
elevation profile is opened or not.*/
public elevationProfileOpen: boolean;
```

Now let's create a function called: "createElevationProfile()". This function is used to create the default chart data, create the chart settings and create a ChartistJS line chart instance using the default chart data and the chart settings.

The function is triggered in the function: "loadItemData()".

The following steps are executed when the function is triggered:

1) Create an instance of the tooltip module.
2) Create the default chart data.
3) Create the chart settings.
4) Create an instance of a ChartistJS Line chart and assign it to the global variable:"elevationProfile".

Now let's create the function by adding the following code below the function: "clearAnimation()" which we created earlier:

NOTE: This code is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
createElevationProfile(): void {

  // Here we instantiate the tooltip module imported at the top of this file.
  let tool = tooltip

  // Here we create the default values of the chart.
  let chartData = {
```

```
  let chartData = {
    // We create an empty list of labels.
    labels: [],
    // We create a list of series with only 1 entry which is the default entry 0
    series: [
      [0]
    ]
  };

  // Here we create the chart settings.
  let chartSettings = {
    // The default max value of the chart is set to 10.
    high: 10,
    // The default low value of the chart is set to 0.
    low: 0,
    // We turn the X-axis gridlines off.
    axisX: {
      showGrid: false
    },
    // We turn the Y-axis gridlines off.
    axisY: {
      showGrid: false
    },
    // We set the area below the line to also be colored.
    showArea: true,
    // We turn the gridlines off.
    showGrid: false,
    // We turn the line off.
    showLine: false,
    // We turn the full chart width on.
    fullWidth: true,
    // We set assign the chartist tooltip plugin as plugin of the chart.
    plugins: [
      Chartist.plugins.tooltip()
    ]
  };

  // Here we create the chartistJS line chart instance and assign it to the
  // global variable: "elevationProfile". We pass the default data and
  // settings as input parameters.
  this.elevationProfile = new Chartist.Line('.ct-chart', chartData, chartSettings);
}
```

Now we need to edit an existing function. The function we need to edit is the function:"loadItemData()" which we created earlier. So go back to this function an add the following code at the bottom of that function:

```
// Here we create an elevationProfile for the Item.
this.createElevationProfile();
```

Now we need to create another function which is used to actually load the elevation data of the selected Item in the ChartJS instance which is created by the function:"createElevationProfile".

This function is triggered by clicking on the Chart Icon in the application (which we will define later in this document) and in the function: "setStaticOverlay()" since every time the setStaticOverlays are set, the elevationData also needs to be loaded.

The following steps are executed when the function is triggered:

1) The elevationData is obtained from the activeItem's activeLayerGroup lineLayer.
2) A check is performed to see if the elevation profile is open.

   If this is the case a JavaScript timeout method is called.

   The JavaScript timeout method is used to wait an N amount of time before the code inside the timeout method is executed. We do this because we need to wait one second after the elevationProfile is opened to load the data. If we don't do this, some errors will occur.

   The code inside the JavaScript timeout method will update the elevationProfile by passing the elevation data as series in the ChartistJS chart.

   The max en min values of the Chart are also updated by calculating the highest and lowest values of in the elevationData.

Now that you know what the function is going to do we can start coding it. This is done by adding the following code below the function: "createElevationProfile()":

NOTE: This code is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
loadElevationData(): void {

  // Here we assign the variable this to a variable: "_this".
  // This is required to access global variables in nested functions.
  let _this = this;

  // Here we obtain the activeLayerGroup's altitudes (elevationData). This
  // data was assigned to the layer in the function:"addLayerGroup()".
  let elevationData = this.activeItem.activeLayerGroup['lineLayer']['altitudes'];

  // Here we perform a check to see if the elevationProfile is opened.
  if (this.elevationProfileOpen) {
    // Here we create the JavaScript timeout method.
    // All the code inside this method is executed after 1 second.
    setTimeout(function () {
    // Here we update the elevationProfile by using the syntax:"profile.update".
    _this.elevationProfile.update({
        // We set the metadata of the data points to: "height Above MSL".
        // This is the text that is shown in the tooltip when hovering over
        // the graphLine.
        series: [{
          meta: 'Height above MSL',
          // We set the values of the series to contain the list of elevationData.
          // We use the slice method to make sure only 1000 datapoints are shown
```

```
          // We use the slice method to make sure only 1000 datapoints are shown
          // in the elevationProfile. This is because the profile will become
          // very laggy if there are more than 100 datapoints selected.
          value: elevationData.slice(0, 1000)
        }]
      }, {
        // Here we set the highest y-axis value to the highest value in the
        // list of elevation data.
        high: Math.max(...elevationData.slice(0, 1000)),
        // Here we set the lowest y-axis value to the lowest value in the
        // list of elevation data.
        low: Math.min(...elevationData.slice(0, 1000))
      }, true)}, 1);
  };
};
```

That's it! Now the last thing we need to do is adding the HTML layout code related to the elevation profile. So let's open the map.component.html file and add the following code outside the div element representing the Settings menu (the div element with the id: "fixed-plugin":

```
<!-- Here we define an angular if statement which determines if the length
of the selectedItems list is bigger than 0. If this is the case the following
HTML code will be executed. -->
<div *ngIf="selectedItems.length> 0">
    <!-- Here we define a new fixed plu
    gin which is shown on the map. -->
    <div class="fixed-plugin-chart" id="fixed-plugin-chart">
        <!-- Here we define a new ngbDropdown. We set autoClose to false so that is does not
        automatically closes when the user clicks somewhere. -->
        <div ngbDropdown [autoClose]="false">
            <!-- Here we create the button which is used to open or close the elevationProfile.
             If the button is clicked the elevationProfileOpen variable is set to True and the
             elevationData is loaded using the function:"loadElevationData()" -->
            <a ngbDropdownToggle
            (click)='elevationProfileOpen = elevationProfileOpen ? false:true; loadElevationData() '>
                <!-- We give the button a chart icon. -->
                <i class="fa fa-area-chart fa-2x" style="color: white; padding: 5px 0px 10px 0px;"> </i>
            </a>
            <!-- Here we define the ngbDropdown menu popup -->
            <div ngbDropdownMenu id="main-dropdown-menu">
               <!-- Here we create the ChartistJS chart.-->
                <div class="row">
                    <div class="col-md-12">
                        <div class="card card-chart">
                            <div class="card-header card-header-primary">
                               <!-- Here we add the element to which the Chart will be assigned
                                after it has been created. -->
                                <div class="ct-chart" id="elevationProfile"></div>
                            </div>
                            <!-- Here we set the text displayed below the ChartistJS chart.
                            We obtain the activeItem's type and name.-->
                            <div class="card-body">
                                <h4 class="card-title"
                                >Elevation Profile (Metres above sealevel)
                                 from {{activeItem.type}}: {{activeItem.name}}</h4>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

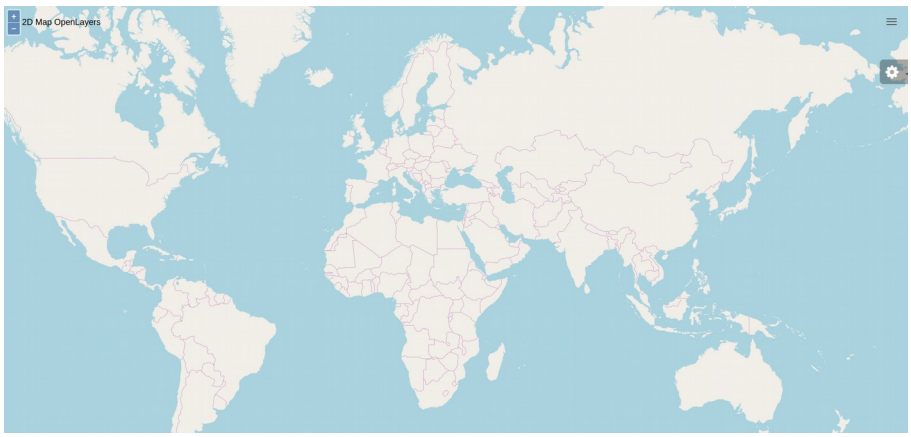
The last thing we need to do is edit an existing function in our map.component.ts file. The function we need to edit is called:"setStaticOverlays()". We need to add the following line at the bottom of this function:

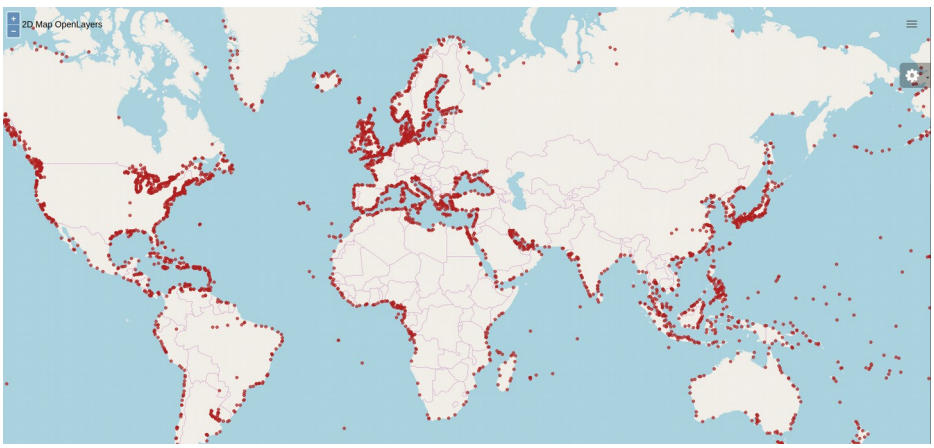
```
// Here we trigger the function which loads the elevationData.
this.loadElevationData();
```

That's it! Now when you refresh the page and select an item you are able to view the elevation data using the elevation profile as shown in the illustration below:

4.2.17 Adding the World Port Index Layer

The last thing we are going to do in this programming manual is creating the World Port Index layer. After finishing this section you will be able to toggle the World Port Layer on or off as shown in the illustration below:





To be able to do this we first need to add a global variable called:"portLayer". We do this by adding the following code below the global variable:"elevationProfileOpen" and above the MapComponent class constructor:

Now we need to add a function called:"createPortLayer". This function is used to create datapoints on the map which each represent a port. The function is triggered in the function:"getItems()" and takes a list of ports as input parameter.

The following steps are executed when the function is triggered:

1) An empty list of points is created.
2) A new OpenLayers style is created.
3) A forEach loop is executed on the list of ports that is passed on the function call.

```
/*
Here we create a global variable called: "portLayer" to which
we are going to assign the portLayer after it has been created.
*/
public portLayer: any;
```

The forEach loop does the following for each of the entries (ports) in the list:

➔ create a new OpenLayers feature, obtain and transform the coordinates of the entry and assign it as geometry of the newly created feature.

➔ Assign the point styling to the newly created feature.

➔ Add the newly created feature to the empty pointList which was created in the first step.

4) A new VectorLayer is created, a new VectorSource is created to which we assign the pointList (containing the features created in the previous step) as features. We assign the newly created VectorSource as source of the newly created VectorLayer.
5) The zIndex of the portLayer is set to 100 to make sure the layer is displayed on top of all the other layers.

Now that you know what the function is used for we can start coding the function. This is done by adding the following code below the function: "loadElevationData()" which we created earlier:

NOTE: This code is described using 2 illustrations. The last line of each illustration is the first line of the next illustration, you don't need to add this line!

```
createPortLayer(ports){
    // Here we create an empty list of points.
    let points = []

    // Here we create the styling of the datapoints
    let pointStyle = new ol.style.Style({
        image: new ol.style.Circle({
            // We set the inside of the Circle to lightRed
            fill: new ol.style.Fill({
                color: 'rgba(178,34,34, 0.7)'
            }),
            // We set the border of the Circle to darkRed with a width of 1
```

```
            // We set the border of the Circle to darkRed with a width of 1
            // pixel
            stroke: new ol.style.Stroke({
                width: 1,
                color: 'rgba(178,34,34, 1)'
            }),
            // We set the radius of the circle (size) to 3 pixels.
            radius: 3,
        }),
    });

    // Here we create a forEach loop which loops through all the entries in
    // the list of ports which is passed on the function call.
    ports.forEach(coord => {
         // Here we create a new Feature. We transform the coordinates
         // in the portList to a format which OpenLayers understands.
         // We assign the transformed coordinates as geometry of the newly
         // created feature.
         let point = new ol.Feature({
             geometry: new ol.geom.Point(ol.proj.fromLonLat(coord)),
         });

        // Set the style of the newly created feature using the pointStyle
        // which we created earlier.
        point.setStyle(pointStyle)

        //Add the newly created feature (point) to the empty pointList.
        points.push(point);
    })

    // Here we create a new VectorLayer and VectorSource to which we assign
    // the pointList (containing all the point features). We assign the
    // newly created VectorLayer to the global variable:"portLayer".
    this.portLayer = new ol.layer.Vector({
        source: new ol.source.Vector({features: points}),
    });

    // Here we set the zIndex of the portLayer to 100.
    this.portLayer.setZIndex(100)
};
```

The last thing we need to do is edit an existing function. The function we need to edit is as follows: “getItem()”. So go back to this function and add the following at the bottom of the function:

```
// Here we call the function getPorts in our PortService file.
// We pass the results in the function: createPortLayer() which will
// then create the portLayer.
this._PortService.getPorts().subscribe(
  (ports: []) => (this.createPortLayer(ports))
);
```

That's it! Now you are able to toggle the World Port Index dataset on and off. This was just a basic implementation of the World Port Index dataset. You could extend the functionality by making it possible to click on a specific port which will open a overlay containing information related to the selected port.

4.3 Adding new types of data to the application

At this point we are able to visualize OpenStreetMap data and OpenSeaMap data from our TileStache Server, Crane (Tracker) Data from our MongoDB datastore and World Port Index data from our PostgreSQL database. In this section we are going to add the functionality to visualize GPS-Route (Trail) Data.

By doing this you will learn how easily you can add new types of datasets to the application. The reason this is easy is because we set up the application in such a way that it's very expandable.

4.3.1 Creating the Trail service file

When adding a new type of dataset (the Trail datasets in this case) we first need to create a new service file in the folder: "2d-map-viewer/src/app/services/" which contains all the functions required to retrieve data from our MongoDB datastore: "Trail_Database". The Trail Service is very similar to the Crane Service file. The file is going to contain the following functions:

- ➔ getTrails() to obtain all the trails in our database.
- ➔ getTrail() to obtain one trail using the MongoID.
- ➔ getSignalsID to obtain all signals belonging to a trail.
- ➔ getSignalsAmount to obtain a N amount of signals belonging to a trail.
- ➔ getSignalsDTG() to obtain all the signals in a given timeframe.
- ➔ getSignalsCountry() to obtain all the signals in a given polygone.

To create the Trail service we first need to create a new TypeScript file called:"trail.service.ts". We do this by running the following command:

```
touch ~/Geostack/angular-apps/2d-map-viewer/src/app/services/trail.service.ts
```

Now let's open this file and import the basic Angular modules at the top of this file by adding the following code:

```
/*
Here we import some basice modules from Angular.
The HttpClient module is required to make requests to our API.
*/
import { Injectable } from '@angular/core'
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs'
```

Now we want to create the class called: "TrailService" and it's constructor for this service. This is done by adding the following code below the module imports:

```
@Injectable()
export class TrailService {

  constructor(private http: HttpClient){};

};
```

This class will be instantiated in the MapComponent. This class will contain all the functions which are required to perform API requests to our Flask-API.

The class will contain functions related to requesting GPS-Route (Trail) data from our MongoDB datastore.

Let's add a function called: "getTrails()", that is able to retrieve all the trails from our MongoDB datastore. This is done by adding the following function in the class: "TrailService " and below the class constructor:

```
getTrails(): Observable < any[] > {
    return this.http.get < any[] > ('/api/trails/')
};
```

The function called: "getTrails()", which is used to perform an HTTP GET request to our Flask-API. The function performs a request on the following URL: /api/trackers/. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves all trackers from the MongoDB datastore. The function:"geTrails()" then returns all the trails to our MapComponent.

Now let's add the function called: "getTrailr()", which is used to obtain a specific trail using it's MongoID, we do this by adding the following function below the function we created above:

```
getTrail(id: string): Observable < any[] > {
    return this.http.get < any > (`/api/trails/${id}`)
};
```

This function is used to perform an HTTP GET request to our Flask-API. The function performs a request on the following URL: api/trails/{id}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves a trail which has the id passed in this function, from the MongoDB datastore. The function:"getTrail()" then returns the trail to our MapComponent.

Now let's add the function called:"getSignalsID()" which obtains all the signals belonging to a trail, we do this by adding the following code below the function we created above:

```
getSignalsID(id: string): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/signals_by_id/${id}`)
};
```

This function is used to perform an HTTP GET request to our Flask-API. The function performs a request on the following URL: /api/signals_by_id/${id}. This URL is bound to a function in our Flask-API. The function which is bound to this URL executes a query on our MongoDB datastore and retrieves all signals belonging to a trail that has the id passed in this function. The function:"getSignalsID()" then returns the signals to our MapComponent.

Now we want to create the function called: “getSignalsAmount()” that obtains a N amount of signals belonging to a specific trail. We do this by adding the following code below the function we created above:

```
getSignalsAmount(id: string, amount: number): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/signals_by_amount/${id}/${amount}`)
};
```

The function performs a request on the following URL: api/signals_by_amount/${id}/${amount}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves all signals belonging to a tracker that has the id passed in this function. The amount of signals it returns is the amount passed in the function call. The function:"getSignalsAmount()" then returns the signals to our MapComponent.

Now let’s add the function which obtains all the signals belonging to a trail in a given time frame, we do this by adding the following code below the function we created above:

```
getSignalsDTG(id: string, dtg_1: string, dtg_2: string): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/signals_by_dtg/${id}/${dtg_1}/${dtg_2}`)
};
```

The function performs a request on the following URL: api/signals_by_dtg/${id}/${dtg_1}/${dtg_2}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves an N amount of signals between the start date (dtg_1) and the end date (dtg_2) belonging to a trail that has the id passed in this function.

Finally we want to add the function which obtains all the signals belonging to a trail in a given polygon, we do this by adding the following code below the function we created above:

```
getSignalsCountry(
    id: string, coords: Number[][]): Observable < any[] > {
    return this.http.get < any[] > (
        `/api/signals_in_polygon/${id}/${coords}`)
};
```

The function performs a request on the following URL: api/signals_in_polygon/${id}/${coords}. This URL is bound to a function in our Flask-API. The function, bound to this URL, executes a query on our MongoDB datastore and retrieves all signals of which the coordinates reside in the list of coordinates passed as parameter in the function, belonging to a trail that has the id passed in this function.

That’s it! Now we have created the service that contains functions which can perform an API call to obtain GPS-Route (Trail) data from our MongoDB datastore.

4.3.2 Importing the Trail service file

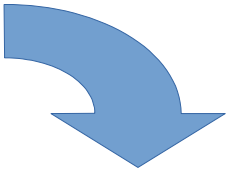
Now we need to import the newly created service file:"trail.service.ts" in our MapComponet file:"map.component.ts".

We start of by importing the file a the top of the map.component.ts file. So let's open this file and add the following below the line which imported the port.service.ts file:

```
import{
  TrailService
} from 'src/app/services/trail.service'
```

Now we need to add the newly imported service to the providers list of the MapComponent. This is done by editing the MapComponent metadata according to the illustrations below:

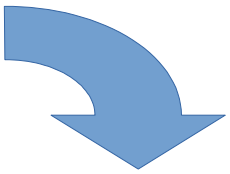
```
@Component({
  selector: 'app-map',
  templateUrl: './map.component.html',
  providers: [MapService, CraneService,PortService]
})
```



```
@Component({
  selector: 'app-map',
  templateUrl: './map.component.html',
  providers: [MapService, CraneService, PortService, TrailService]
})
```

Now we need to update the MapComponent class constructor to make sure we can use the TrailService (and it's functions) in the MapComponent. So Let's edit the MapComponent constructor according to the illustrations below:

```
constructor(private _MapService: MapService,
  private _CraneService: CraneService,
  private _PortService: PortService) {}
```



```
constructor(private _MapService: MapService,
  private _CraneService: CraneService,
  private _PortService: PortService,
  private _TrailService: TrailService) {}
```

That's it! Now we can use the TrailService and it's functions throughout our MapComponent file by using the syntax: "this._TrailService.{a function}".

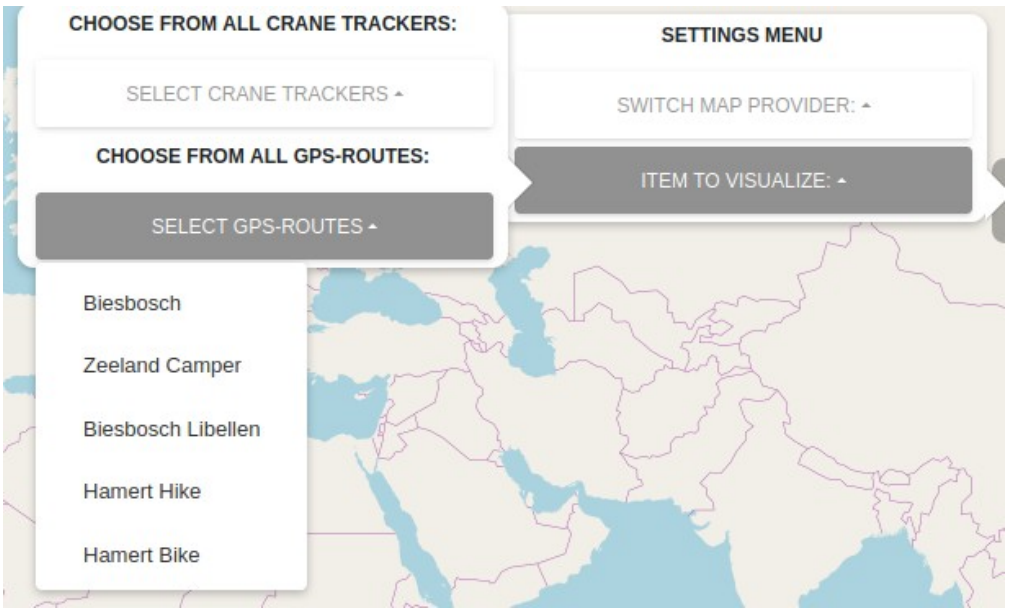
4.3.3 Selecting trails

Now we have to make sure that we are able to obtain GPS-Route (Trail) data in our MapComponent. For this we need to edit some existing functions.

The first function we need to edit is the function: “getItems()” which (as of now) is used to obtain the Crane (Trackers) and the World Port Index data from our datastores when the MapComponent is loaded. To make sure the GPS-Routes (Trails) are also loaded when the component is loaded we need to add the following code in the fucntion:”getItems()” and below the line related to obtaining the World Port Index data:

```
// Here we call the function getTrails in our TrailService file.
// We call the function: addItem() on each of the entries in the list of
// trails which was returned by our Flask-API. We pass the required values
// as input parameters in the addItem function.
this._TrailService.getTrails().subscribe(
  (trails: []) => (
    trails.forEach(trail => {
      this.addItem(
        trail['_id']['$oid'], trail['name'], 'trail',
        trail['t_points'], 'time', [trail['s_date']['$date'],
        trail['e_date']['$date']],
      );
    })
  )
);
```

Now we need to update the map.component.html file to make sure we add a new drop-down list, containing all the obtained Trails, to our application as shown in the illustration below:



So let' open the map.component.html file and go to the div element related to selecting items (the div element with the id:"itemSelection"). Now let's add the following code (encircled in red in the illustration below) below the code related to selecting Crane Trackers.

Note: In the illustration below the code related to selecting Crane Trackers is included and folded. So it may look different for you and you don't need to add this code again!

```
<li class="header-title">Choose from all Crane trackers:</li>
<div ngbDropdown>…
</div>
<!--
Here we add the dropdown box which contains all the trails from our database
-->
<li class="header-title">Choose from all GPS-Routes:</li>
<div ngbDropdown>
    <!--
    Here we add the dropdown toggle button for out tracker selection dropdown
    -->
    <button class="btn btn-white btn-block" ngbDropdownToggle>Select GPS-Routes</button>
    <div ngbDropdownMenu>
        <!--
        Here we add a ng-container that contains a FORloop that displays all
        the objects in our items list which is defined in the component.ts file

        For each of the entries a button will be created which when clicked
        triggers the function: "selectItem()" and passes the item which is selected as
        parameter.
        -->
        <ng-container *ngFor="let item of items">
          <!--
            Here we check if the item type is equal to tracker. We do this because
            when we add other types of datasets to our application we dont want them
            to be in the dropdown box for our trackers.
            -->
            <div *ngIf="item.type == 'trail'">
                <!-- When one of the items is clicked the function:"selectItem()" is triggered
                 in which the clicked item is passed. -->
                <button ngbDropdownItem (click)='selectItem(item);'>{{item.name}}</button>
            </div>
        </ng-container>
    </div>
</div>
```

Now when you reload the application you will be able to select Trails. When a trail is clicked nothing will happen yet since we need to update some more functions. These functions are as follows:

- ➔ getInitialItemData(), to make sure the initial signals from the selected Trail are obtained.
- ➔ getItemDataByDTG (), to make sure we can obtain signals in a timeframe.
- ➔ getItemDataByAmount(), to make sure we can obtain signals by amount.
- ➔ getITemDataByCountry(), to make sure we can obtain signals by country (in a polygon).

Before adding any code related to obtaining the Trail data in our MapComponent.

4.3.3 Updating the map.component.html file

Add to GetItems()

Add to getInitialItemData()

Add to getItemDataByDTG ()

Add to getItemDataByAmount()

Add to getITemDataByCountry()

Add to