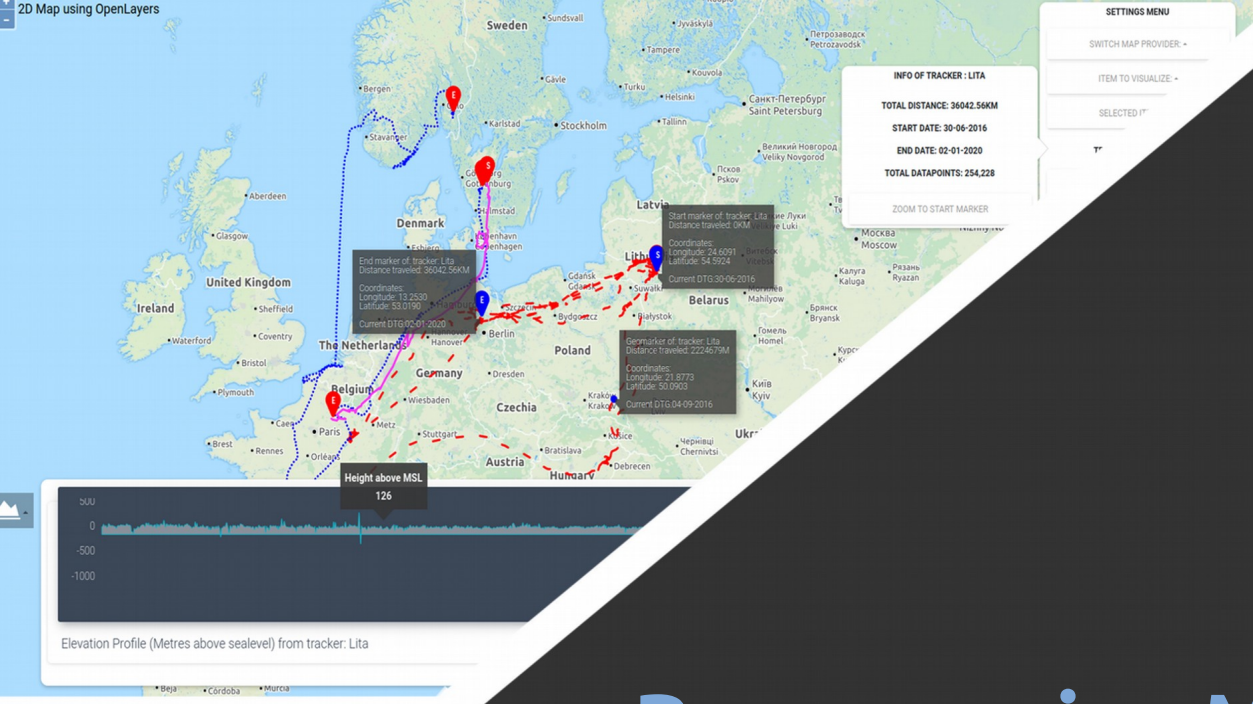
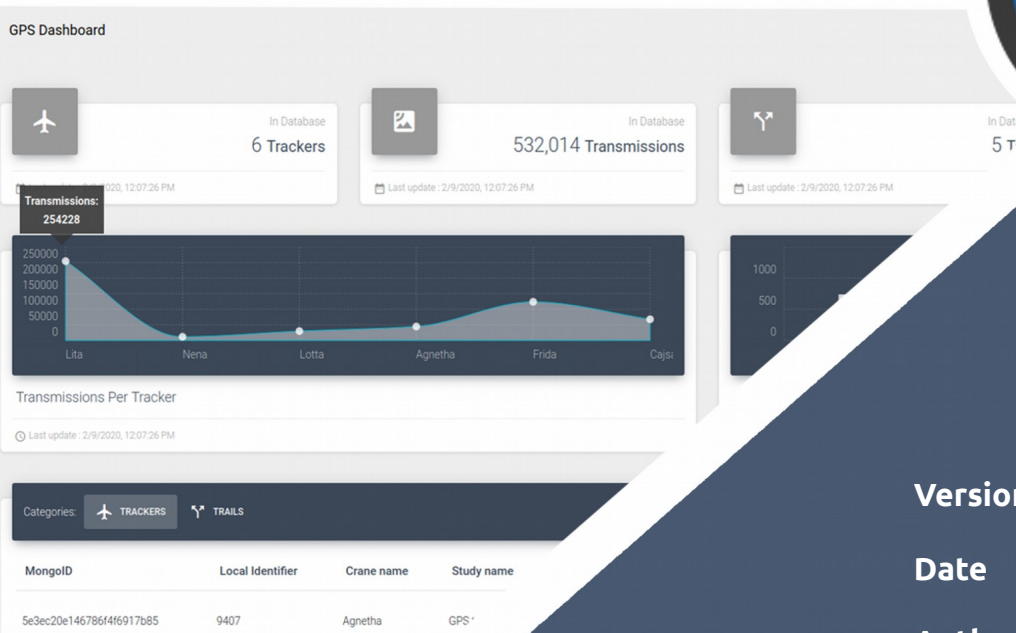


Programming Manual

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

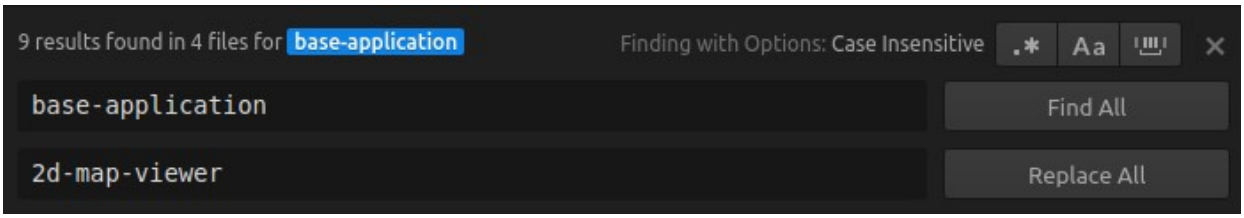
During this chapter we are going to convert the basic web application in such a way that we can start coding the 2D Map Viewer application. We do this because the base application contains the base structure of the 2D Map viewer application which we are going to create during this programming manual. 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

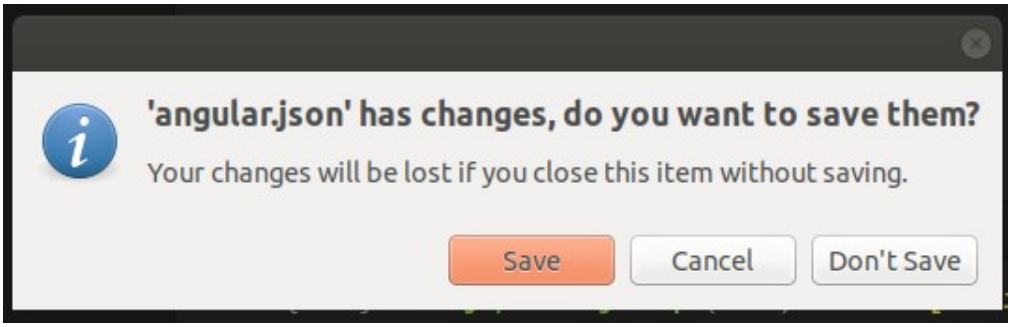
Since this application is an extension of the base-application we can copy this application and start creating the 2D map viewer from there. After we copied the base-application folder, we need to change some names and titles to make the new application the 2D map viewer. We start by changing the name of the folder we just 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 functions related to requesting World Port Index data from
our PostgreSQL database called World_Port_Index_Database.
*/
@Injectable()
export class PortService {
    /*
    Here we create class constructor and pass an instance of an HttpClient.
    The HttpClient is used to perform the following requests to our Flask-API:
    1) GET requests
    2) POST requests
    3) PUT requests
    For more info on the types of requests you can visit the following URL:
    https://www.tutorialspoint.com/http/http_requests.htm
    */
    constructor(private http: HttpClient) { }

    /*
    Here we create a function called: "getPorts()". 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/port/. The function bound to this URL executes a query
    on the World Port Index database which obtains all the ports in the
    database.
    */
    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:

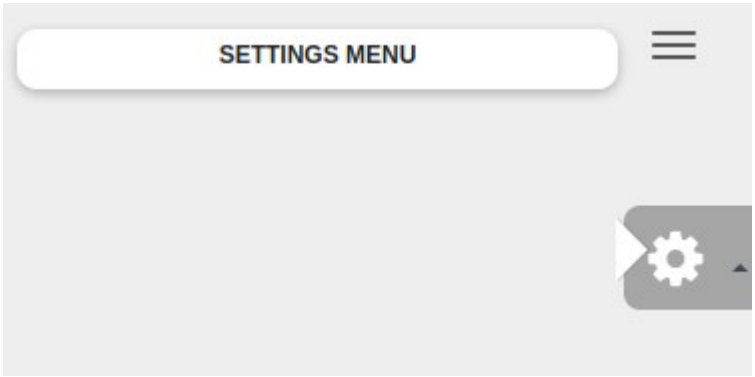
The page does not have any content yet so we are going to add this now.

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 vairable we can use the map throughout the whole component.
  */
  private 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:

```
  private 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 on

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.

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
  });
```

On the previous page 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.

Now we want to create a global variable called portLayer. The layer containing the World Port Index dataset will be assigned to this variable later on. We add this global variable by adding the following code below the global variable: “seaLayer”:

```
/*
Here we create a global variable called: "portLayer"*/
private portLayer: any;
```

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;
      })
    )
  )
};
```

Above we created the function 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"
  )
};
```

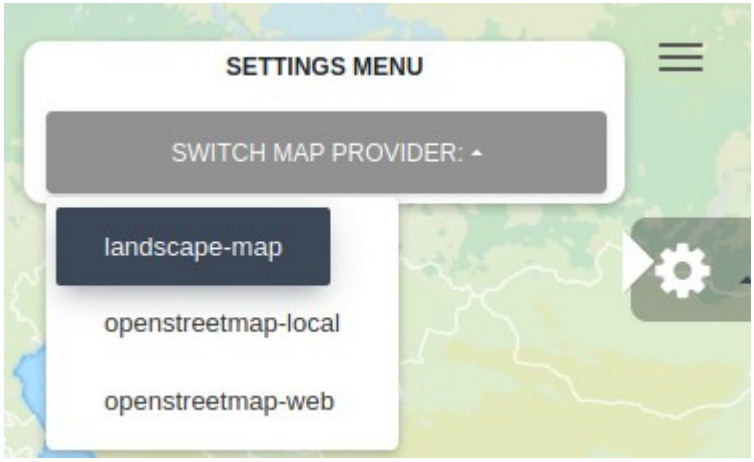
Above we created the function that changes the mapProvider. When the function is triggered a providerKey is passed. This providerKey is the key of the entry in the JavaScriptMap: "mapProviders". We then call: “getSource” on the mapLayer to obtain the source of the layer. Then we set the URL, on which 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: portLayer which we created earlier.

```
private 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 visualized. 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 ccreated 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();
};
```

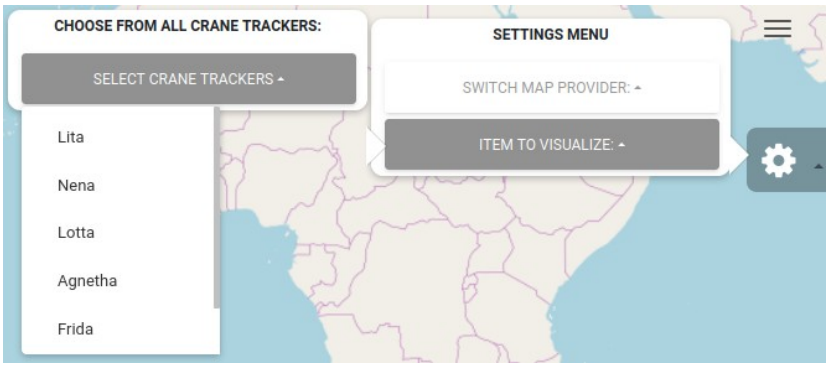
At this point we cannot select any items yet. To solve this we need to add another drop-down menu to our map.component.html file. We add this html code in the div element with the id:"settings" and below the div tag with the id: "WMSSelection".

```
<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:

```
private 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 variabe called:”activeItem”. We do this by adding the following code below the selectedItems global variable:

```
private 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.

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.

How this is done is shown on the next page.

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 at index length of the datalist 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:

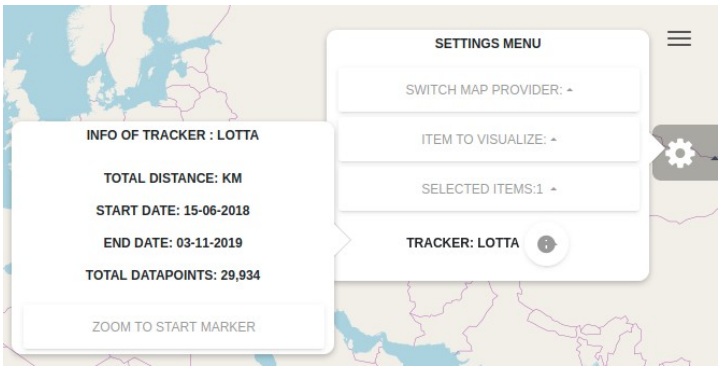
```
zoomToLocation(): void {
  /*
  The code below is used for the animation that moves to the start
  coordinates of the activeItem.
  The view of the OpenLayers Map instance is obtained using the syntax:
  ".getView()" on the map instance. Then we animate the view by calling the
  function:".animate()"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

  /*
  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

Now we are going to add the functionality to actually visualize the selected data obtained from our MongoDB datastore by creating layers.

First we need to create a global variable called:"layerStyles". This variable is going to contain the default styling of the layers we are going to create. To create this global variable we need to add the following code below the global variable called:"activeItem":

```
/*
Here we create 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 as source of the icon. We also anchor the icon
to be displayed above the datapoint.
*/
private 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`
      })
  })
};
```

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 using the inline-comments found in the source code located in the POC folder.

```
/*
Here we create a function called: "addLayerGroup()".

This function will create the following layers for the item that is passed as
parameter when te function is triggered in the function:"loadItemData()":
- lineLayer: This is the layer that creates the lines between the datapoints.
- pointLayer: This is  the layer which will contain the arrows that visualize
              the direction in which the item is going.
- markerLayer: This layer contains the start and end marker of the visualized
               route.

The following happens 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 specific
   layerGroups.

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.

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

```
   - 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 (datarow) in the
   list:

   8.1) A variable point1 is created to which we assign the value of the
        coordinate on the index that the FORLoop is on.

   8.2) A variable point2 is created to which we assign the value of the
        coordinate on the index + 1 that the FORLoop is on.

   8.3) 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()".

   8.4) 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.

   8.5) A new feature is created.
        The value (coordinates) of point1 are assigned as geometry of this
        feature.
```

```
        The value (coordinates) of point1 are assigned 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.

    8.6) 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
   - A feature with the type: "startMarker". We assign the startCoordinate
     value of the item as the feature's geometry.
   - 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.
```

```
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 following values:
      - 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.
*/
```

```
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 following values:
      - 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.
*/
```

On the next pages the images will follow related to actually creating the function.

```
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= [];

      // 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]
```

```
      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.
      point.setStyle(pointStyle)

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

    // Here we create the pointlayer 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.
```

```
// 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.
// We set the geometry of the startMarker to the startCoordinate of
// the item which we are going to add.
// We set the geometry of the endMarker to the endCoordinate of
// the item which we are going to add.
let markerLayer = new ol.layer.Vector({
  source: new ol.source.Vector({
    features: [
      new ol.Feature({
        type: 'startMarker',
        geometry: new ol.geom.Point(item.startCoordinate)
      }),
      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,
});
```

```
        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 we have created the function that creates all our layers and adds them to the map and the item's layerGroups list.

The function: "setLayerGroup()" has not been created yet. So let's create this function.

Create function setLayerGroups

add function addLayerGroup to function LoadItemData

Add HTML for DTG selection

4.2.7 Creating and setting Overlays

Adding overlay html divs to HTML file

Create function addOverlays()

add Adoverlays function in the CreateMap function()

add setDynamicOverLayes function()

add setStatiocOverlays function()

add setStaticOverlays function to selectItem function and to loaditemdata function and to setLayergroup function

4.2.8 Removing a selected Item

Add fcuntion removeItem

4.2.9 Removing a LayerGroup

Add function removeLayerGrooup

4.2.10 Adding DTG selection

Create function for getting DTG item data

Create DTG picker component folder

create HTML DTG Picker

Create TS DTG picker

Import and add datepicker to app.module.ts

add dateRange as global variable

add getDTGEvent function

add html to dtgSelection

add this.dateRange = this.activeItem.dateRangeTotal; to the selectItem function

4.2.11 Adding Amount selection

Now that we have most of our functionalities in place we want to be able to select a desired amount of transmissions or signals.

We do this by adding a function that triggers the function in our service that is related to performing an API call to obtain a certain amount of data.

We need to add the following code below the function: “getDTGEvent()”:

```
/*
Here we create a function called: "getItemDataByAmount()"

This function is called when a amount is selected from the dropdown list related
to the amount selection.

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 triggerd.

These functions trigger 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.
*/
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.

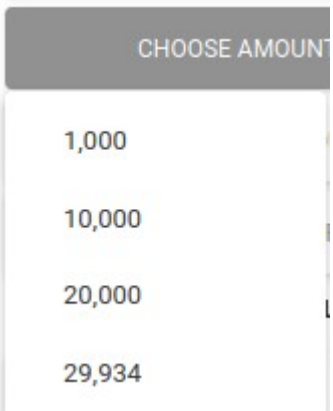
How this is done is shown on the next page.

To create a dropdown box for the amount selection we need to add the following code to the map.component.html file below the div element with the id:”dateSelection”:

```
<!--
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 current coordinateList as text.
        -->
          <button class="btn btn-white btn-block" ngbDropdownToggle>
            Choose amount: {{ activeItem.coordinateList.length | number: '2.'}}
          </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 the total amount of
              transmissions / signals belonging to the activeItem.

              We use the syntax : {{1000| number: '2.'}} to add a comma to the number.
              -->
              <div ngbDropdownMenu>
                  <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,1000)'>
                    {{1000| number: '2.'}}
                  </button>
                  <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,10000)'>
                    {{10000| number: '2.'}}
                  </button>
                  <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,20000)'>
                    {{20000| number: '2.'}}
                  </button>
                  <button ngbDropdownItem (click)='getItemDataByAmount(activeItem,activeItem.totalDataLength)'>
                    {{activeItem.totalDataLength| number: '2.'}}
                  </button>
              </div>
      </div>
  </div>
```

Now when we reload the application we get the option to select an amount of transmissions / signals from the dropdown box as shown in the illustration below.



4.2.12 Adding Country selection

Add global variable country selection

Create function for getting selection by country

Add HTML for amount selection

4.2.13 Adding Layer and Overlay Toggling

Add toggleLayer function

add toggleOverlay function

add HTML

4.2.14 Changing layer styling

Add global colorlist, widthlist , linestylelist

Add global dictionary styleDict

Add function setLayerStyles()

Add HTML

4.2.15 Animating routes

Add animateRoute function

Add clea Animartion fucntion

add HMTL

4.2.16 Creating an elevation profile

Add globals private elevationProfile:any;

 private elevationProfileOpen:boolean;

Add loadElevationData function

Add createLEevationProfile fucntion

add html