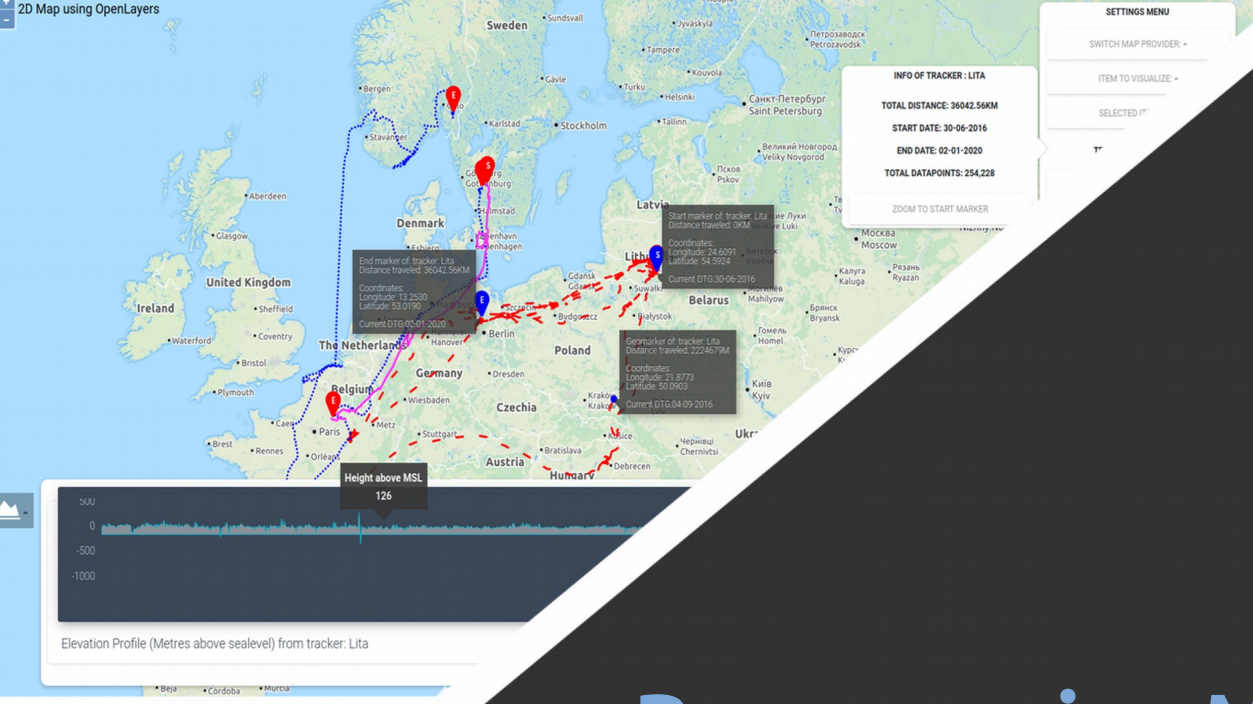
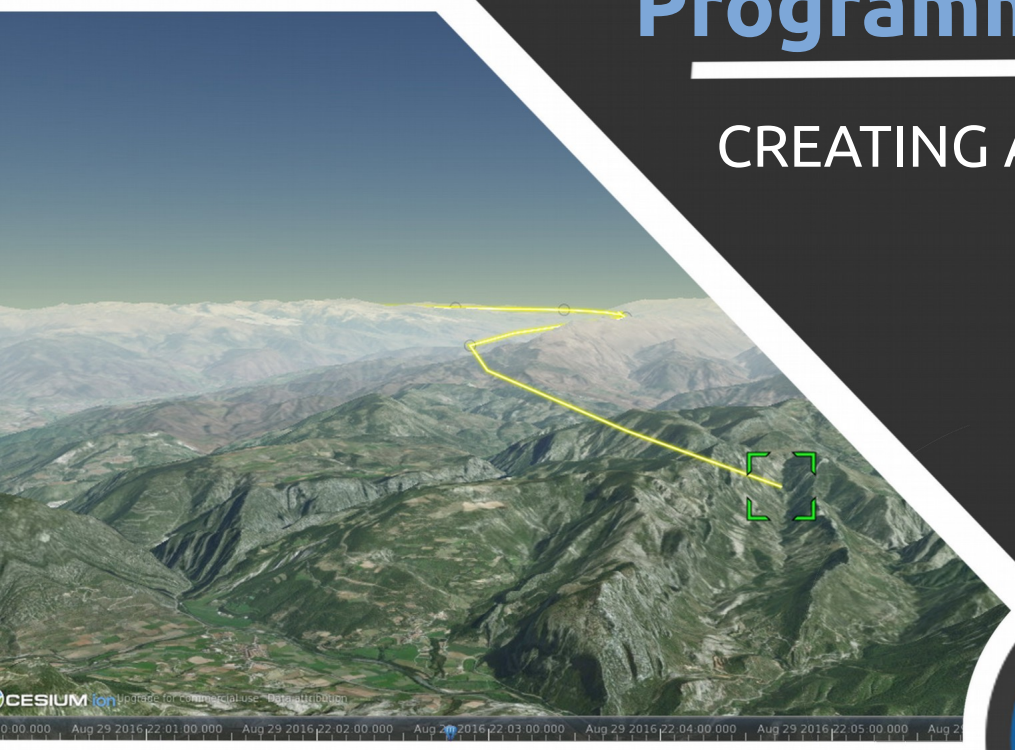
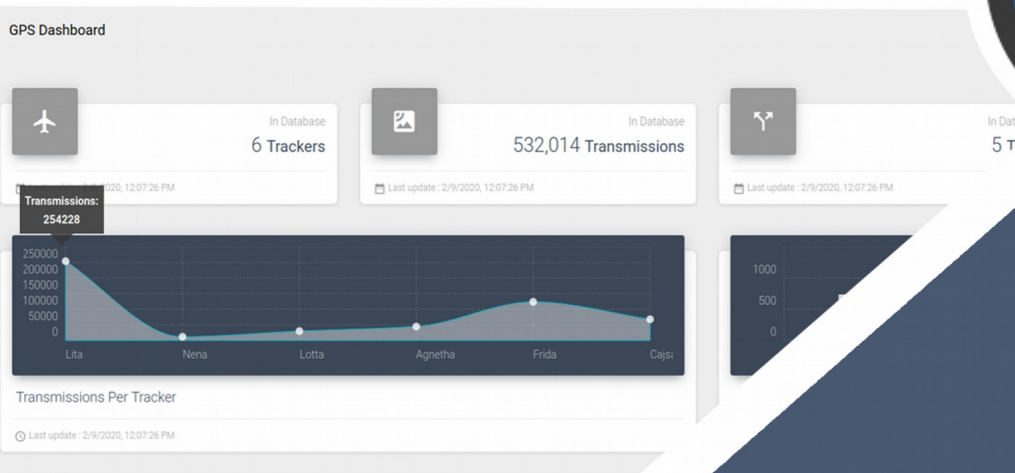
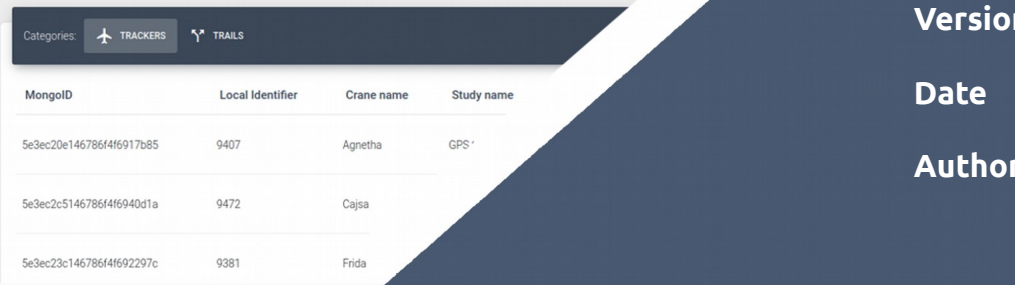


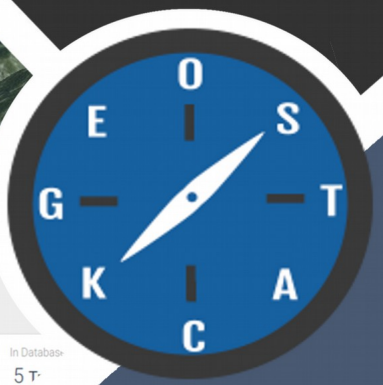
Programming Manual

CREATING AN 3 DIMENSIONAL MAP VIEWER









Version : 1.0

Date : 21-06-2020

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 3D map viewer application using the AngularJS JavaScript framework and the JavaScript framework Cesium. This application is an extension of our Angular base application and the 2D Map Viewer.

The reason this application is an extension of the 2D Map Viewer is because the 3D Map Viewer uses the same functions in order to add and remove items, add and remove layers and selecting item data. We don't want to write the same code as we did in the 2D Map Viewer so we are going to remove the unused code and edit the existing code according to the needs of the 3D Map Viewer.

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 3D Map Viewer using Cesium 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.

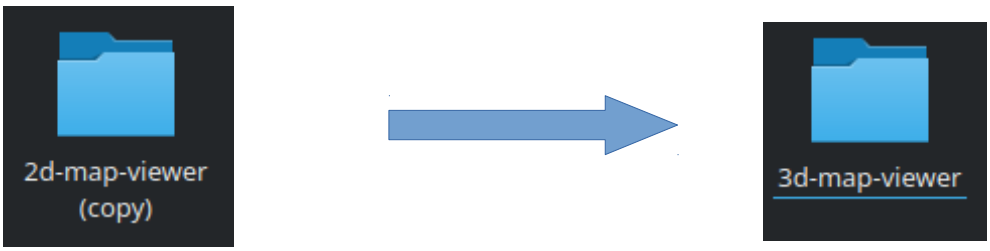
NOTE: Sometimes you will notice that in the code which you have to create some functions do not exist yet. Don't worry about this since they will be added later on during the programming manual!

Table of Contents

1.Introduction

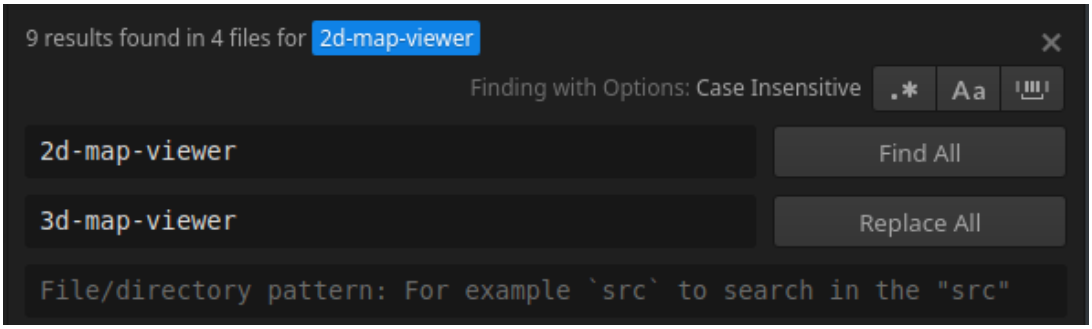
1.1 Getting ready

This application is basically the same as the 2D Map viewer so we can copy this application and start creating the 3D map viewer from there. After we copied the 2d-map-viewer folder, we need to change some names and titles. We start by changing the name of the folder we just copied from 2d-map-viewer to 3d-map-viewer, as shown in the image below.

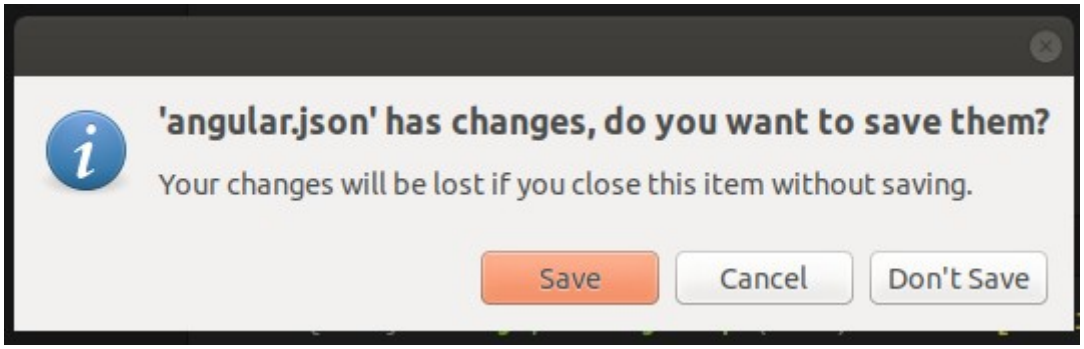


We also need to edit the project name: “2d-map-viewer” to “3d-map-viewer”. If you are using the code-editor Atom, this is done by performing the following steps:

1) In the edit press the keys Ctrl + shift + f in the Atom editor.
2) In the screen that pops up enter: “2d-map-viewer” in the find section and “3d-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: index.html located in the folder 3d-map-viewer/src, replace the title from 2D Map Viewer to 3D Map Viewer.
5) In the file: sidebar.component.html, located in the folder src/app/components/sidebar/, change the text: “3D Map Viewer” to “3D Map Viewer”.

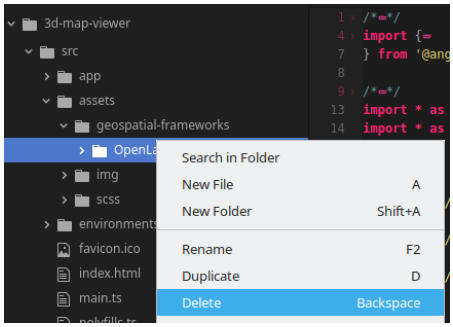
6) In the file: sidebar.component.ts, located in the folder src/app/components/sidebar/, change the title of the route related to the 2D Map Viewer from 2D Map OpenLayers to 3D Map Cesium as shown in the illustration below:

```
export const ROUTES: RouteInfo[] = [
    { path: '/map-page', title: '3D Map Cesium', icon: 'map', class: '' },
    { path: '/base-page', title: 'Base Page', icon: 'map', class: '' },
];
```

1.2 Adding the JavaScript framework Cesium

Since we don't need the geospatial JavaScript framework OpenLayers anymore we can remove the OpenLayers folder which can be found in the folder: "3d-map-viewer/src/assets/geospatial-frameworks/".

Deleting the OpenLayers framework can done by opening the 3d-map-viewer folder in Atom, finding the OpenLayers Folder, right clicking the folder and selecting delete as shown in the illustration below:



Now that we have removed the JavaScript Framework: "OpenLayers" we should add the JavaScript Framework:"Cesium".

Just like with OpenLayers; Adding the geospatial framework to our Angular application can be done in 2 ways which are as follows:

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

   This is the first technique which you can use to install Cesium 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://cesium.com/blog/2018/03/12/cesium-and-angular/

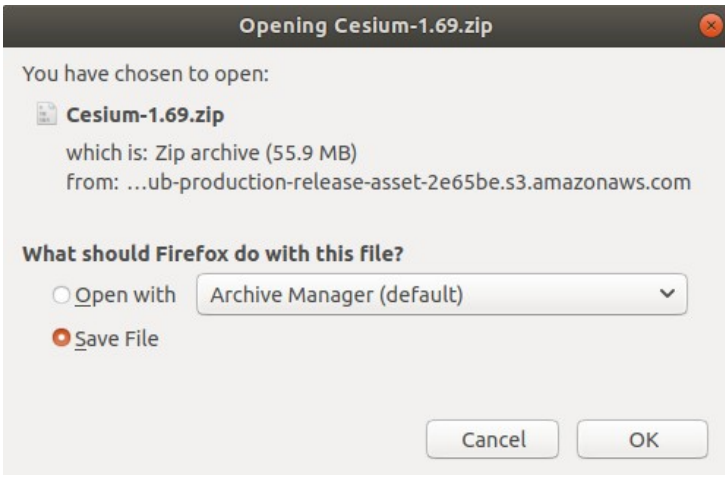
2) **Downloading the Cesium source code:**

   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 Cesium source code from the Cesium website which is located on the following URL: https://cesium.com/downloads/

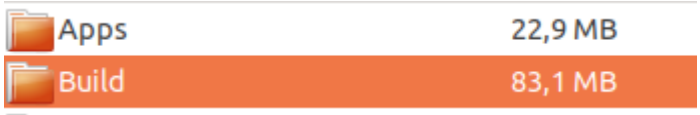
When navigating to the URL mentioned above you should be greeted with a green download button as shown in the illustration below:

Click on the download button and then select Save File and click on Ok as shown in the illustration below:



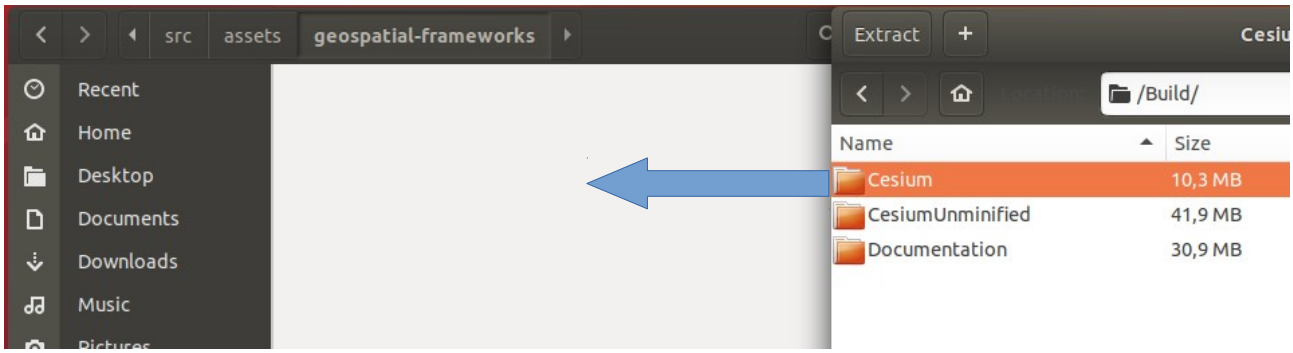
After a few seconds you will end up with a ZIP folder in your downloads folder as shown in the illustration below:

Open the ZIP file. This ZIP contains all the files related to the JavaScript framework Cesium. Since we only need the Cesium source code we should click on the build folder as shown in the illustration below:



Once we are in the build folder we now have to copy the folder called: “Cesium” to the geospatial-frameworks folder in our 3d-map-viewer application (“3d-map-viewer/src/assets/geospatial-frameworks/”)

This can be done by dragging and dropping the Cesium folder as shown in the illustration below:



Now that we have the Cesium source code in our Angular application we need to edit the index.html file of our application so that the application knows where Cesium is located.

So let’s open the index.html file which is located in the folder: “3d-map-viewer/src/”.

We currently have the references to the OpenLayers framework in the <head> tag of the index.html. So let’s change it according to the illustration shown below:

```
<!--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>
```

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

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

That’s it! Now we can use the JavaScript framework Cesium throughout our application. In the next section you will learn how to create a free Cesium ION account. This step is optional but recommended since it will extend the functionalities which you can add to your 3D Map Viewer later

1.3 Creating a free Cesium ION account

This section describes why and how you can create a Cesium ION account. As mentioned above; this step is optional and is not required to create the 3D Map Viewer application. Creating a Cesium ION account will however extend the base functionalities of the 3D Map Viewer application such as using the Cesium World Terrain Layer and the Bing Aerial satellite images.

“Cesium ion is a robust, scalable, and secure platform for 3D geospatial data. Upload your content and Cesium ion will optimize and tile it for the web, serve it up in the cloud, and stream it to any device.

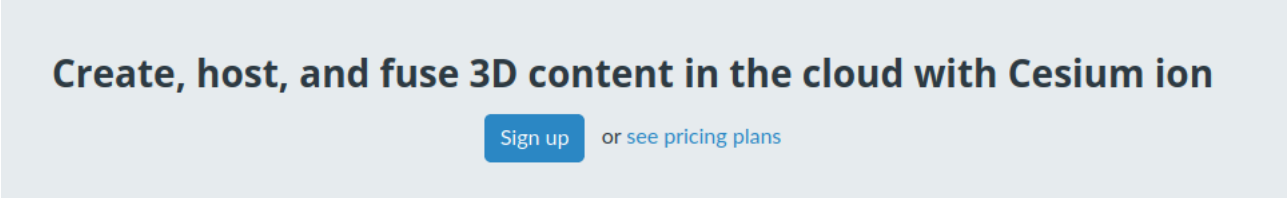
With Cesium ion, you'll have access to our curated 3D content including Cesium World Terrain and Bing Maps imagery. Combine these assets with your own data to provide more context or see it on a 3D map of the world.

The Cesium ION tools are built for 3D geospatial—and tied to precise map coordinates. We first built Cesium to track objects in space, and this incredible precision is baked into our platform.” Source: “www.cesium.com, 2020”

To find more information related to the capabilities of Cesium ION you should visit the following URL:

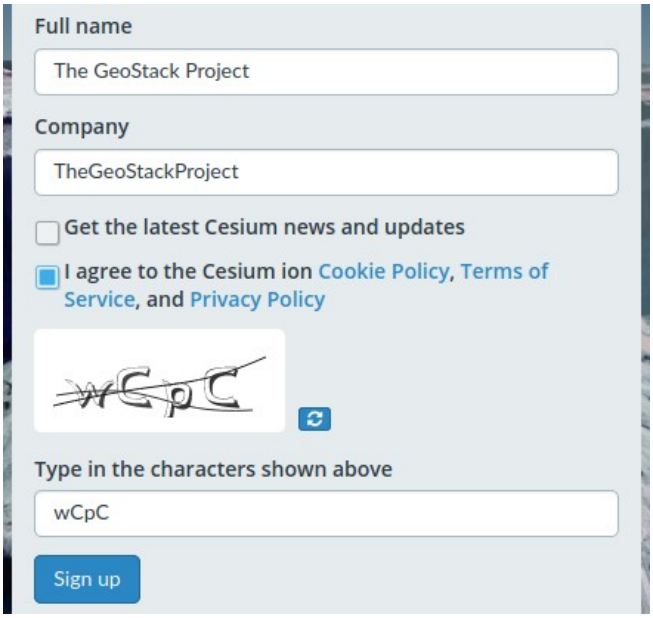
https://cesium.com/cesium-ion/

On this website you will also find the option to register an account as shown in the illustration below:

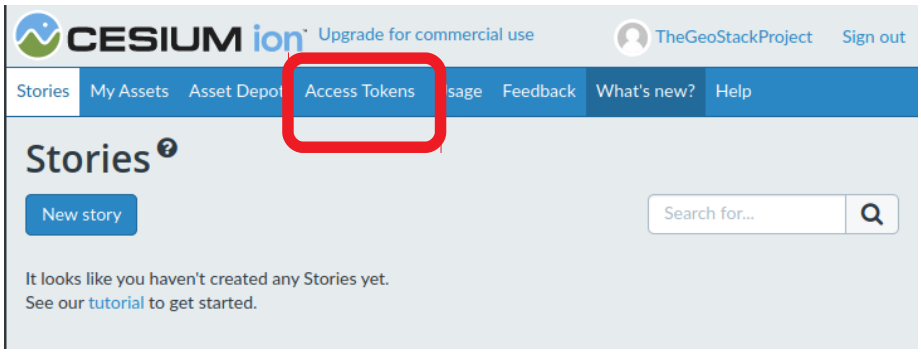


Click on the Sign up button and fill in your information as shown in the illustration below:

NOTE: Make sure to fill in you own information!



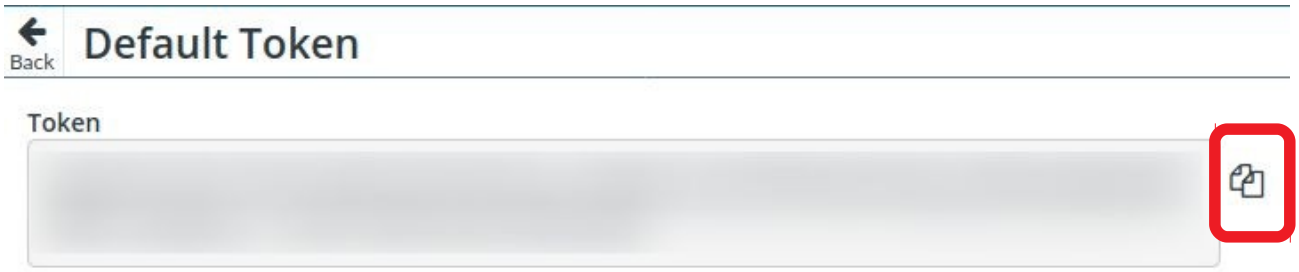
Once you are registered and logged in you should be greeted with the same screen as shown in the illustration below. Click on the button: “Access Tokens” encircled in red.



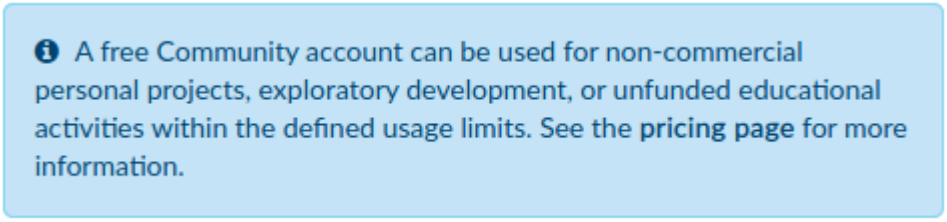
In the next window that shows you should click on the default token entry as shown in the illustration below:

Now copy the token by clicking on the button encircled in red in the illustration below. Save the token somewhere in a text file. We are going to use it later!

NOTE: In the illustration below the token is blurred for privacy reasons!



That’s it! Now you have registered a free Cesium ION account. NOTE: The free Cesium ION account cannot be used for Commercial purposes as shown in the illustration below:



For more information visit this URL: https://cesium.com/pricing/

In the next section we are going to cleanup the map.component.ts file to remove the code from the 2D Map Viewer application which we don’t need for the 3D Map viewer application.

2 Cleaning up the map.component.ts file

Because the 3D Map Viewer is an extension of the 2D Map Viewer we can keep most of the code which we created during the manual: “Creating an 2 Dimensional Map Viewer”.

This section describes what files and code you should remove and keep in order to turn the 2D Map Viewer into the 3D Map Viewer. So let’s open the map.component.ts file located in the folder: “3d-map-viewer/src/app/page/map-page/”.

To make it easier to remove code we can fold the code so that only to first lines of the module imports, functions etc. are shown.

This is done by pressing Ctrl+ Alt + Shift + [on your keyboard. This will result in the following map.component.ts file:

```
/*…*/
import {…
} from '@angular/core';

/*…*/
import * as Chartist from 'chartist';
import * as tooltip from 'chartist-plugin-tooltips'

/*…*/
import {…
} from 'src/app/services/map.service'
import {…
} from 'src/app/services/crane.service'
import {…
} from 'src/app/services/port.service'
import{…
} from 'src/app/services/trail.service'

/*…*/
declare const ol: any;

/*…*/
export class Item {…};

/*…*/
@Component({…})
export class MapComponent implements OnInit {…};
```

2.1 Cleaning the unused imports

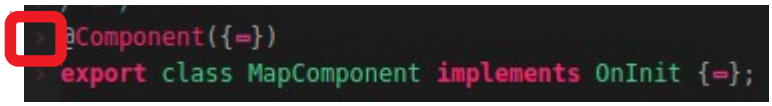
We want to start of by removing the module imports which we don’t need anymore. The following module imports have to be removed:

- ➔ The Chartist module and Tooltip modules, since we don’t need an elevationProfile in our 3D Map Viewer.
- ➔ The PortService import, since we don’t need the World Port Index to be displayed in our 3D Map Viewer application.

So let’s remove the imports by selecting the code and pressing backspace as shown in the illustration below:

```
/*…*/
import * as Chartist from 'chartist';
import * as tooltip from 'chartist-plugin-tooltips'
```

The next thing we need to do is removing the PortService from the providers entry in our MapComponent metdata. So open the Component metadata by clicking on the arrow (encircled in Red) next to the line @Component({}) as shown in the illustration below:



Now remove the PortService from the providers list as shown in the illustration below:

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

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

The last thing we need to do in order to remove the unused imports is removing the PortService from the MapComponent class constructor. So click on the arrow (encircled in red) next to the line : “export class MapComponent implements OnInit” to unfold the MapComponent class coe as shown in the illustration below:

Now find the MapComponent class constructor and edit it according to the illustration below:

```
constructor(private _MapService: MapService,
  private _CraneService: CraneService,
  private _PortService: PortService,
  private _TrailService: TrailService) {}
```

```
constructor(private _MapService: MapService,
  private _CraneService: CraneService,
  private _TrailService: TrailService) {}
```

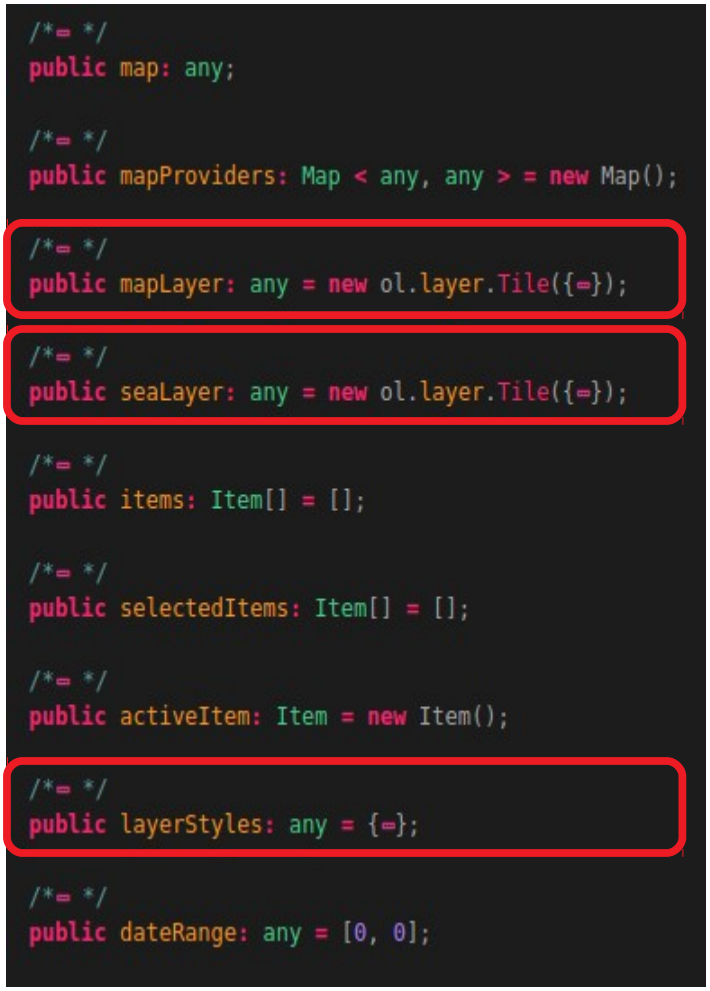
That’s it! Now we have removed the unused imports from our 3D Map Viewer map.component.ts file. In the next section we are going to remove the unused global variables.

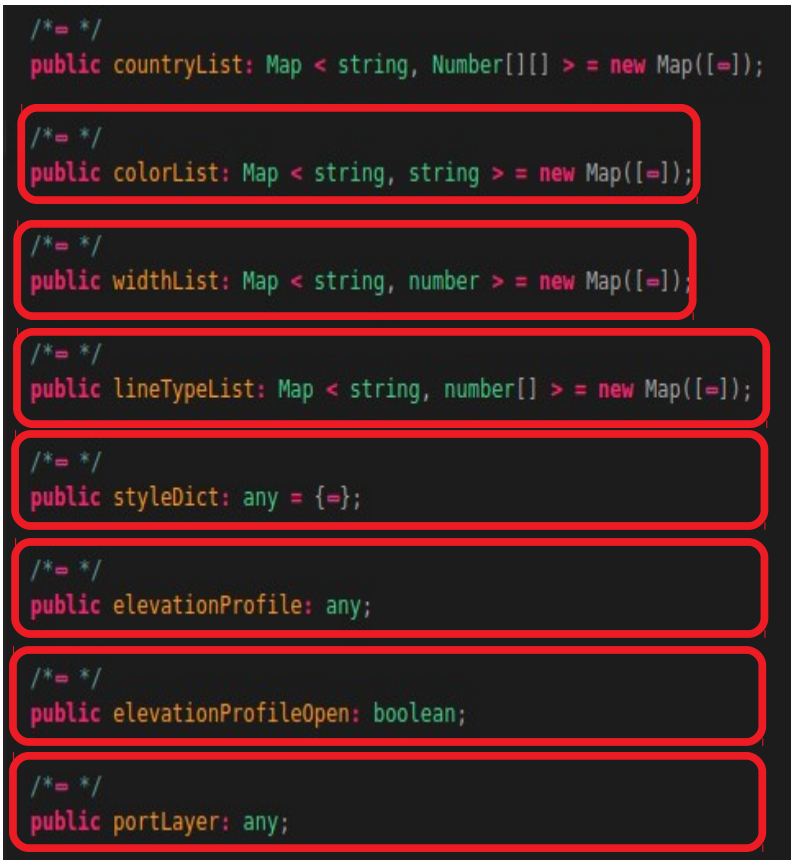
2.2 Cleaning the unused global variables

Now let's remove some global variables which we are not going to need for our 3D Map Viewer application. The global variables which we are going to remove are as follows:

- ➔ MapLayer, since we are going to create a new Map Layer which is usable with Cesium;
- ➔ SeaLayer, since we do not need the OpenSeaMap Layer in the 3D Map Viewer application;
- ➔ LayerStyles, since layer styling works differently in Cesium;
- ➔ colorList, widthList, lineTypeList and StyleDict, again since layer styling works differently in Cesium.;
- ➔ elevationProfile, since we don't need the elevation profile in the 3D Map Viewer application;
- ➔ elevationProfileOpen, again since we don't need the elevation profile in the 3D Map Viewer;
- ➔ portLayer, since we are not going to use the World Port Index dataset in the 3D Map viewer.

Now that you know what global variables we are going to remove we should start removing them. In the illustrations below the global variables encircled in red need to be removed:





2.3 Cleaning unused functions

Now that we have removed the unused global variables we can start removing the unused functions. The functions we are going to remove are as follows:

- ➔ createOpenLayersMap(), since we are going to create a new function which creates a Cesium map;
- ➔ setMapProvider(), since we are going to create a new function which changes map providers;
- ➔ zoomToLocation(), since we are going to create a new function which zooms to the start location of an item;
- ➔ addOverlays(), since we are do not have to create overlays in our 3D Map Viewer application;
- ➔ setDynamicOverlays() and setStaticOverlays(), since we are not going to use overlays;
- ➔ toggleLayer() and toggleOverlay(), since we don't need layer toggling in our 3D Map Viewer;
- ➔ setLayerStyle(), since we are not going to use layerStyling in the 3D Map Viewer;
- ➔ animateRoute() and clearAnimation(), since animating items is done differently in Cesium;
- ➔ createElevationProfile() and loadElevationData(), since we don't need an elevation profile in our 3D Map Viewer;
- ➔ createPortLayer(), since we are not going to use the World Port Index dataset in our 3D Map Viewer.

Now that you know which functions are going to be removed we can start removing them. In the illustrations below the functions encircled in red need to be removed:

```
/*= */
constructor(private _MapService: MapService,=) {}

/*= */
ngOnInit() {=};

/*= */
getMapProviders(): void {=};

/*= */
createOpenLayersMap(): void {=};

/*= */
setMapProvider(providerKey): void {=};

/*= */
addItem(itemId, itemName, itemType, itemRouteLength, itemTimeColumn, itemDTG): void {=};
```

```
/*= */
getItems(): void {=};

/*= */
timeConverter(timestamp): string {=};

/*= */
selectItem(item: Item): void {=};

/*= */
getInitalItemData(item: Item): void {=};

/*= */
loadItemData(data: any[]): void {=};

/*= */
zoomToLocation(): void {=};

/*= */
addLayerGroup(item: Item): void {=};

/*= */
setLayerGroup(groupKey: string): void {=};

/*= */
addOverlays(): any[] {=};

/*= */
setDynamicOverlays(item: Item): void {=};

/*= */
setStaticOverlays(item: Item): void {=};

/*= */
removeItem(item: Item): void {=};

/*= */
getItemDataByDTG(item: Item, dtg_s, dtg_e): void {=};

/*= */
getDTGEvent(id: string, $event): void {=};
```

```
/*= */
removeLayerGroup(layerGroupKey: string): void {=};

/*= */
getItemDataByAmount(item: Item, amount): void {=};

/*= */
getItemDataByCountry(item: Item, coords: Number[][]): void {=};

/*= */
toggleLayer(layerType: string): void {=};

/*= */
toggleOverlay(overlayType: string): void {=};

/*= */
setLayerStyle(layerType: string): void {=};

/*= */
animateRoute(): void {=};

/*= */
clearAnimation(): void {=};

/*= */
createElevationProfile(): void {=};

/*= */
loadElevationData(): void {=};

/*= */
createPortLayer(ports){=};
```

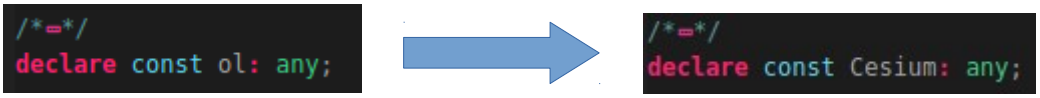
The last thing we need to do before this section is finished is clearing the code inside the functions: “addLayerGroup()” and “setLayerGroup()”. So go to these functions in the map.component.ts file and remove all the code inside these functions so that only the following remains:

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

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

That's it! Now we have removed all the functions which we are not going to need anymore. In the next section we are going to edit some existing functions according to the needs of our application.

2.4 Editing the existing code

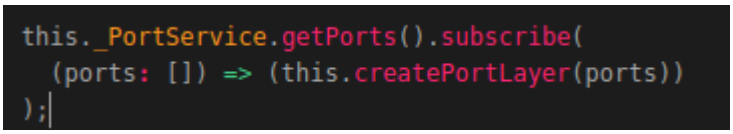
Now that we have removed the unused imports, variables and functions we need to update some code and functions. Let's start off by editing the constant which was declared at the top of the map.component.ts file. At this point we have declared a constant called:"ol" which was used to be able to use build-in OpenLayers functions in our 2D Map Viewer application. We want to use Cesium instead of OpenLayers so let's edit the constant according to the illustrations below:



Now we can use the syntax: "Cesium.{a build in function}" to use the functionalities of the JavaScript framework Cesium.

Next up is editing the function: "ngOnInit()". We need to remove the line that triggers the function: createOpenLayersMap() since we removed that function earlier. So let's edit the function according to the illustrations below:

Next up is editing the function: "getItems()". We have to remove the code which was used to obtain the World Port Index data. We do this because we are not going to create a PortLayer in the 3D Map Viewer application. So let's remove the code shown in the illustration below:



Next up is editing the function: "selectItem()". We want to remove the line of code which was used to set the Static Overlays in our 2D Map Viewer application. We do this because we are not going to use overlays in the 3D Map Viewer. So let's remove the line shown in the illustration below:

The next function we are going to edit is the function: "loadItemData()". We want to edit one line related to converting coordinates in a format which was understandable for OpenLayers.

The line we want to edit is found in the foreach loop which loops through all the datapoints from the list of data that is passed on the function call.

So let's edit the line according to the illustrations below:

```
item.coordinateList.push(
  ol.proj.fromLonLat(row.geometry.coord.coordinates)
);
```

```
item.coordinateList.push(row.geometry.coord.coordinates);
```

We also want to edit the line related to adding the timestamps to the datetimeList in the loadItemData function. We do this since we need the raw timestamps instead of the converted timestamps. So let's edit that line according to the illustration below:

```
item.datetimeList.push(
  this.timeConverter(row[item.timestampColumn].$date)
);
```

```
item.datetimeList.push(row[item.timestampColumn].$date);
```

We also want to remove the 2 lines related to setting the static overlays and creating the elevation profile. This is done by removing the lines of code shown in the illustration below:

```
this.setStaticOverlays(item)

this.createElevationProfile();
```

Next up is the function: "removeItem()". We need to remove the lines of code which we used to clear the running animation, toggle the overlays of and setting the static overlays. We start of by removing the line of code related to clearing the animation. This is done by editing the first line in the function:"removeItem()" according to the illustrations below:

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

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

We also need to remove a few lines of code which were used to remove the layers from the OpenLayers map in our 2D Map Viewer. The lines which you have to remove are shown in the illustration below:

```
item.layerGroups.forEach(layerGroup => {
  for (let [key, value] of Object.entries(layerGroup)) {
    this.map.removeLayer(value['layer'])
  }
});
```

The last line we need to remove the last line of code in the function: “removeItem()”. The line which you have to remove is shown in the illustration below:

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

The last function we need to edit is the function: “removeLayerGroup()”. In this function we also need to remove the line related to clearing any running animations. The line of code which you have to remove is shown in the illustration below:

```
this.clearAnimation()
```

The last thing we need to remove in the function:”removeLayerGroup()” is the code related to removing all the layers from the OpenLayers map in our 2D Map Viewer. The lines that you have to remove are shown in the illustration below:

```
for (let [key, value] of Object.entries(groupToRemove)) {
  this.map.removeLayer(value['layer'])
}
```

That’s it! Now you have removed the unused code and edited the functions which we are going to use in the 3D Map Viewer application. If you run the application you should not encounter any errors.

To run the the application you should navigate to the root directory of the 3D Map Viewer by running the following command:

```
cd ~/Geostack/angular-apps/3d-map-viewer
```

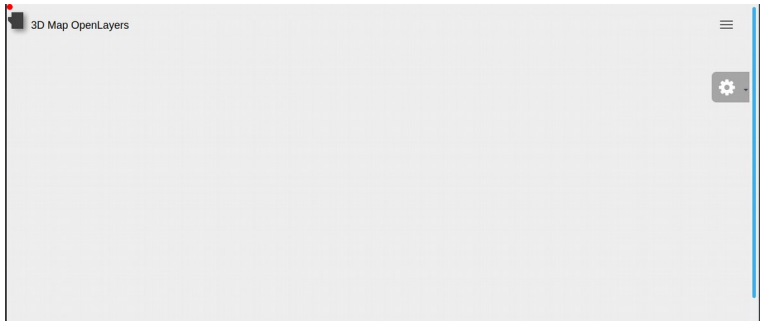
And then start the Angular Live Development server by running the following command:

```
sudo npm start
```

If you run the application you should not encounter any errors as shown in the illustration below:

```
10% building 3/3 modules 0 active[wds]: Project is running at http://localhost:4200/webp
i [wds]: webpack output is served from /
i [wds]: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 147 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 268 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 2.31 MB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.87 MB [initial] [rendered]
Date: 2020-05-27T15:18:13.034Z - Hash: c2f4929c06eaa57308c7 - Time: 11793ms
** Angular Live Development Server is listening on localhost:4200, open your browser on ht
i [wdm]: Compiled successfully.
```

When opening the application in your browser you will be greeted with the page shown in the illustration below:



3 Cleaning up the map.component.html file

Now that we have cleaned the map.component.ts file we also need to remove some code from the map.component.html file. We do this because we will not be using settings such as changing layer styles, animation routes, toggling overlays etc. So let's open the HTML file in Atom.

Now let's fold the code like we did in the map.component.ts file. Folding code is done by pressing the key combination Ctrl + Alt + Shift + [on your keyboard. After you folded the code the map.component.html file will look the same as shown in the illustration below.

NOTE: You will probably do not have to code comments (the green text) shown in the illustration. The code in the folder: "POC" contains these comments.

```
<!-- Here we add the empty div elements related to the overlays. -->
<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>

<!--
Here we create a div element to which the Cesium 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>

<!--
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">
</div>


<!-- Here we definean 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">
</div>
```

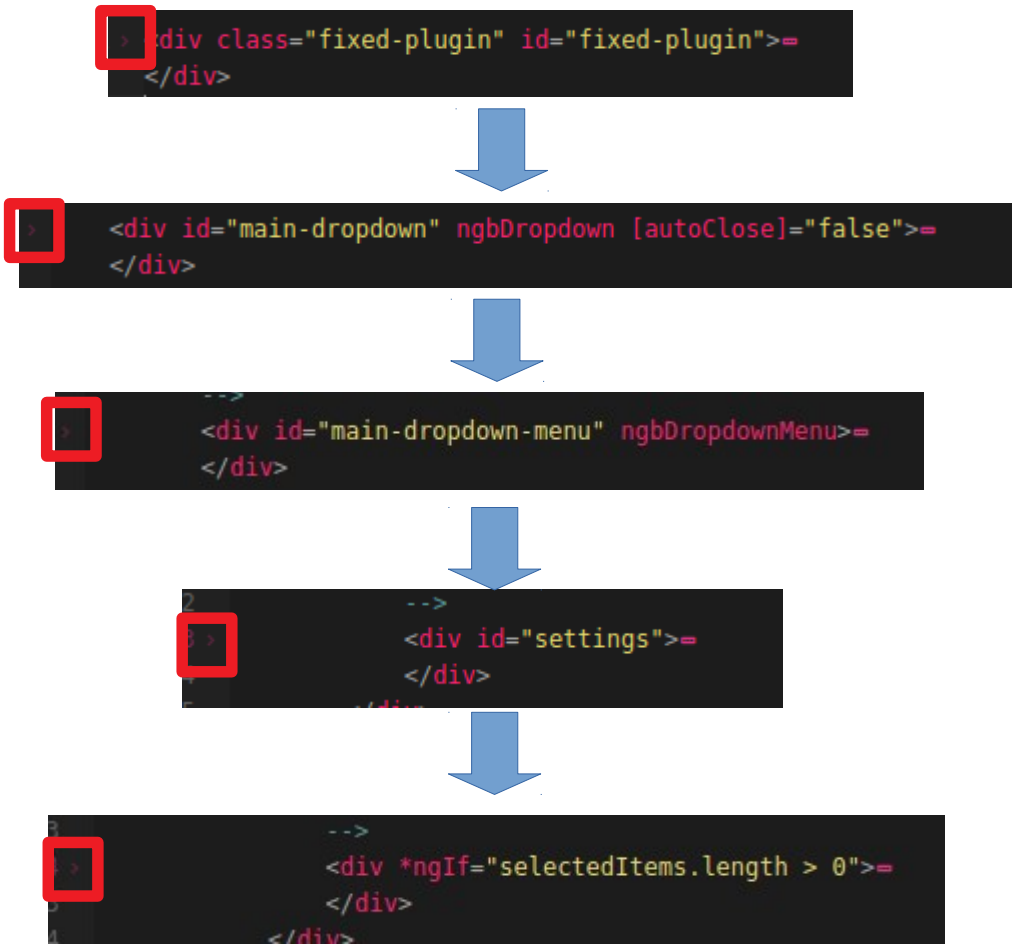
Now let's start of by removing the 4 div elements at the top of the file. These div elements were used for the overlays in the 2D Map Viewer. We remove them because we are not going to use overlays in the 3D Map Viewer. The code that you have to remove is shown in the illustration below:

```
<!-- Here we add the empty div elements related to the overlays. -->
<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>
```

The next code that we want to remove is the last div element (related to the elevation profile) in the HTML page. The code which you have to remove is shown in the illustration below:

```
<!-- Here we definean 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">
</div>
```

The last thing we need to remove in the map.component.html file are the div elements related to the layer toggling, layer styling and the animation. So let's find these div elements by clicking on the red arrows (to unfold the code) next to the line of code shown in the illustrations below:

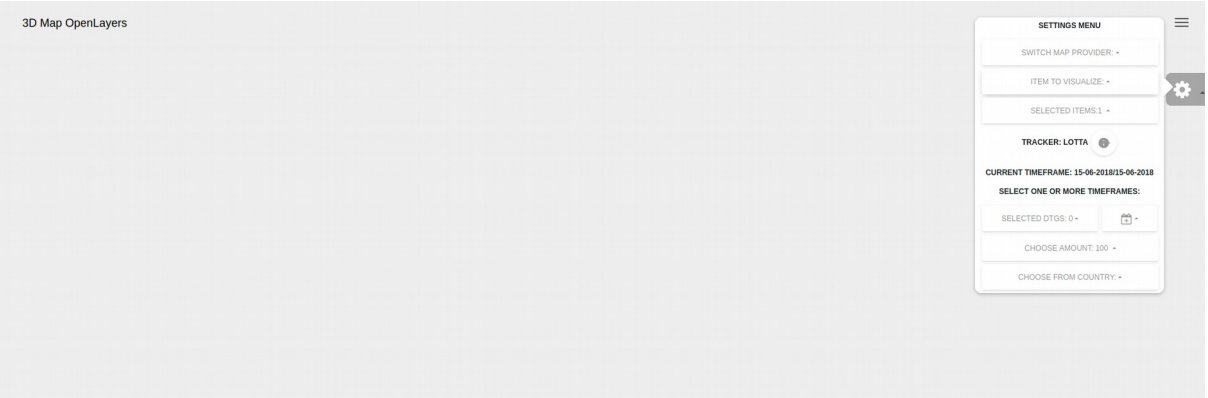


Now find and remove (by highlighting them and pressing delete on you keyboard) the div elements (and the code inside the elements) shown in the illustration below:

NOTE: You will probably do not have to code comments (the green text) shown in the illustration. The code in the folder: "POC" contains these comments.

```
<div id="toggleSelection">
</div>

<!--
In this div element we add the logic related to changing layer styles
-->
<div id="styleSelection">
</div>

<!--
In this div element we add the logic related to animating the route.
-->
<div id="animationSelection"  style="margin-top:55px;">
</div>
```

That’s it! Now when you refresh the application, open the settings menu and select an item you should have see the same as shown in the illustration below:



At this point we are finally done with removing and editing existing code. Now we are ready to start adding the functionalities in order to create the 3D Map Viewer.

4 Creating the Map Component functionalities

At this point we are finally done with removing and editing existing code. Now we are ready to start adding the functionalities in order to create the 3D Map Viewer.

We are going to start of by creating the Cesium Map instance and adding the base map, which was created in section 5.5.3: “Installing the TileStache Tileserver”, and the elevation maps which were created in section 5.5.4.2: “Rendering Digital Terrain Models for Cesium” of the cookbook: “Creating the GeoStack Course VM”.

How this is done is shown in the next section.

4.1 Creating the Cesium Map instance (Cesium Viewer)

Before creating the Cesium Map instance we need to add 2 global variables to our map.component.ts file. These global variables are as follows:

1) mapTileLayer, which is the layer that is going to contain the OpenStreetMap tiles served by our TileStache Tileserver.

   This is the variable to which the baseLayer containing the tiles of the map is assigned. We assign the elevationMaps by creating a new Cesium OpenStreetMapImageryProvider and passing the location of the TileStache Tileserver running behind the NGINX webserver as URL.

   We set the local OpenStreetMap tiles as default WMS. So when the application is loaded, the local OpenStreetMap tiles are loaded as well.

2) MapTerrainLayer, which is the layer that is going to contain the Cesium Terrain Files (Elevation map) served by our Cesium Terrain Server.

   This is the variable to which the baselayer containing the elevationMaps is assigned. We assign the elevationMaps by creating a new CesiumTerrainProvider and passing the location of the Cesium Terrain Server running behind the NGINX webserver as URL.

   We set the local Hamert DSM files as default Terrain files. So when the application is loaded, the local (DTM) terrain files of the Hamert are loaded as well.

Let's start of with creating the mapTileLayer. This is done by adding the following code below the global variable:"map" and above the global variable:"mapProviders":

```
public mapTileLayer:any = new Cesium.OpenStreetMapImageryProvider({
  url : 'http://localhost/tiles/openstreetmap-local/'
});
```

Now let's add the mapTerrainLayer. This is done by adding the following code below the global variable: "mapTileLayer" which we created above:

```
public mapTerrainLayer:any =  new Cesium.CesiumTerrainProvider({
  url : 'http://localhost/terrain/M_52EZ2'
});
```

Now let's add a function called: "createCesiumMap()". This function is used to create the Cesium Map instance which is also known as the Cesium Viewer. The function is triggered in the function: "ngOnInit()" to make sure that the Cesium Map (Viewer) is created when the MapComponent is loaded.

The map instance will be created in the HTML div element with the id:'map'. This div element is defined in the HTML layout of the MapComponent which can be found in the file: "map.component.html".

The following steps are executed when the function is triggered:

1) Trigger the function:"getMapProviders()" which is used to obtain all the available WMS's (WebMapServers) which are defined in our TileStache configuration file.
2) Optional: Assign your Cesium ION Token key to the Cesium instance. (This is the token which you copied in section 1.3 of this document)
3) Create a new Cesium Map instance (Viewer) to which we assing the global variable:"mapTileLayer" as imageryProvider (tile layer) and the global variable:"mapTerrainLayer" as terrainProvider (Elevation map layer)
4) set allowDataSourcesToSuspendAnimation to False to make sure that when an animation is started it will not stop when a Terrain file or Tile is loaded.

Now that you know what the function is used for we can start coding it. This is done by adding the following code below the function: "getMapProviders()":

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!

```
createCesiumMap():void{
  // Here we trigger the function:"getMapProviders()" to obtain all the
  // available WMS's.
  this.getMapProviders()

  // Here we assign the Cesium ION Token to the Cesium instance.
  Cesium.Ion.defaultAccessToken = 'Paste Your Cesium token';

  // Here we create a new Cesium Map instance (Viewer). We pass the id ('map')
```

```
  // Here we create a new Cesium Map instance (Viewer). We pass the id ('map')
  // of the div element in the HTML layout in which the Viewer will be added.
  this.map = new Cesium.Viewer('map',
   {
      // Here we assign the mapTileLayer as imageryProvider.
      imageryProvider: this.mapTileLayer,
      // Here we assign the mapTerrainLayer as terrainProvider.
      terrainProvider: this.mapTerrainLayer,
    });

  // Here we make sure that animations are not stopped when a data object
  // is loaded.
  this.map.allowDataSourcesToSuspendAnimation = false;
};
```

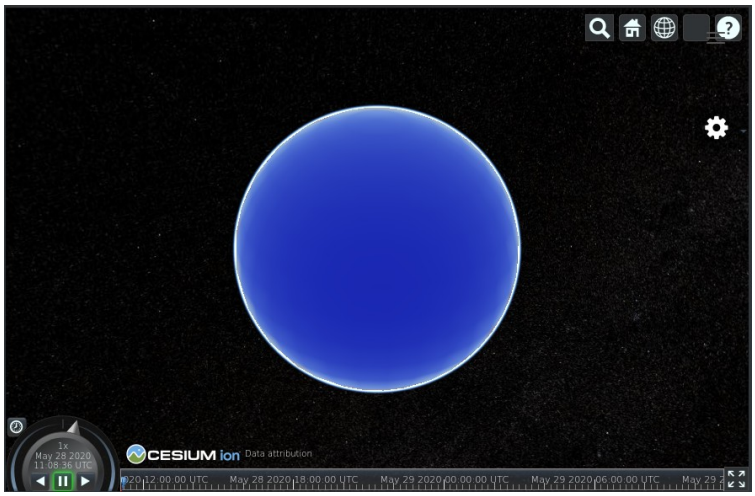
Now we need to add the function: “createCesiumMap()” to our ngOninit function to make sure that the Cesium Map instance (Viewer) is created when the application is loaded. So edit the function according to the illustration below:

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

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

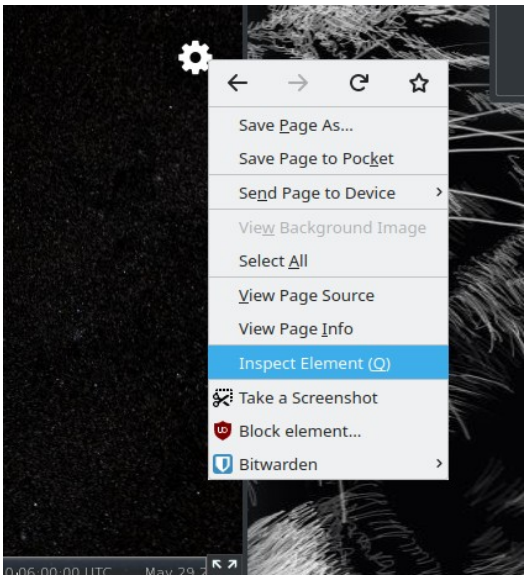
Now start the Flask-API, TileStache Server, MemCached Cache and the Cesium Terrain Server By clicking on the Desktop Icons shown in the illustrations below (from left to right):

Now when we reload the 3D Map Viewer application you should be greeted with the following screen:

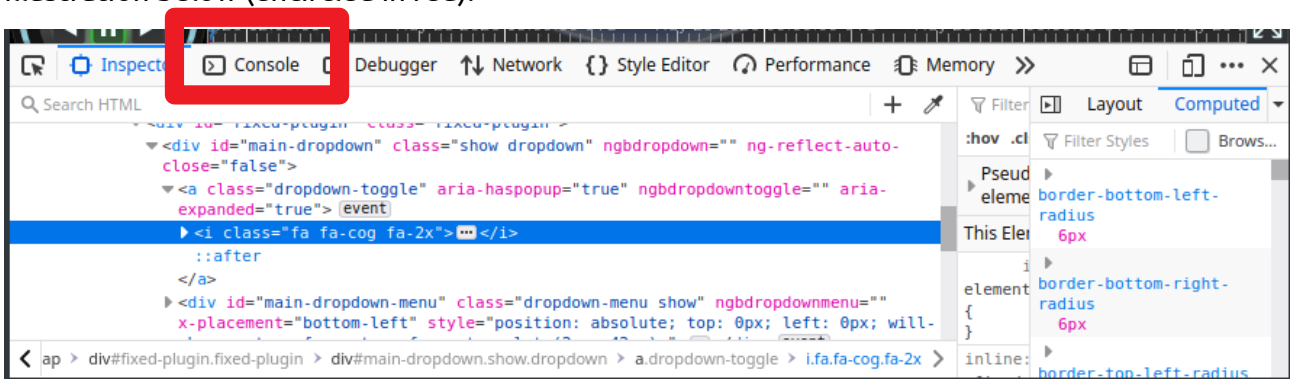


As you may have noticed, you only see a blue sphere which represents the globe. The problem is the map is not showing up yet.

The reason for this can by right clicking on the settings icon and clicking on inspect element as shown in the illustration below:



This will open the Firefox developer options. Now click on the console entry as shown in the illustration below (encircled in red).



This will open the Firefox developer console. As you can see the problem that occurs is that the Cross-Origin requests are blocked as shown in the illustration below. These requests are made to our TileStache Tileserver from which we obtain the OpenStreetMap tiles.

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at http://localhost/tiles/openstreetmap-local/1/1/0.png.
(Reason: CORS header 'Access-Control-Allow-Origin' missing). [Learn More]
```

These error's occur when trying to obtain resources from multiple web servers at the same time. In the example above this error occurred when trying to obtain OpenStreetMap Tiles from the TileStache tileserver (running on "localhost:8081") in the 3D Map Viewer running on localhost:4200.

For more information related to the specifics of CORS you should read the following URL:

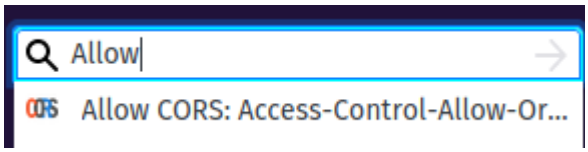
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

4.1.1 Allow Cross-Origin Resource Sharing (CORS)

To solve this problem we need to install a Firefox extension called: “Allow CORS”. This is done by performing the following steps:

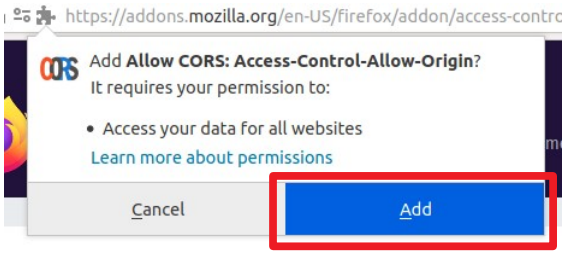
1) Navigate to the Addons website of Firefox by entering the following URL in your browser: https://addons.mozilla.org/en-US/firefox/

2) Type: “Allow CORS” in the search bar in the top right of the page and click on the first option that pops up, as shown in the illustration below:

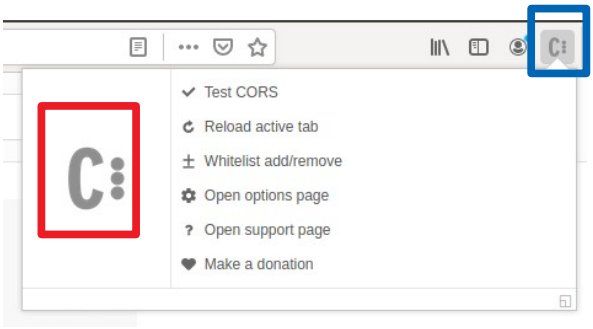


3) On the next screen select: “Add to Firefox” as shown in the illustration below:

4) Then select: “Add” in the popup window as shown in the illustration below:



5) After the extension is add a new icon will pop up in the top right of your Firefox window (Blue), click it and Click on the allow CORS icon as shown in the illustration below (Red) :



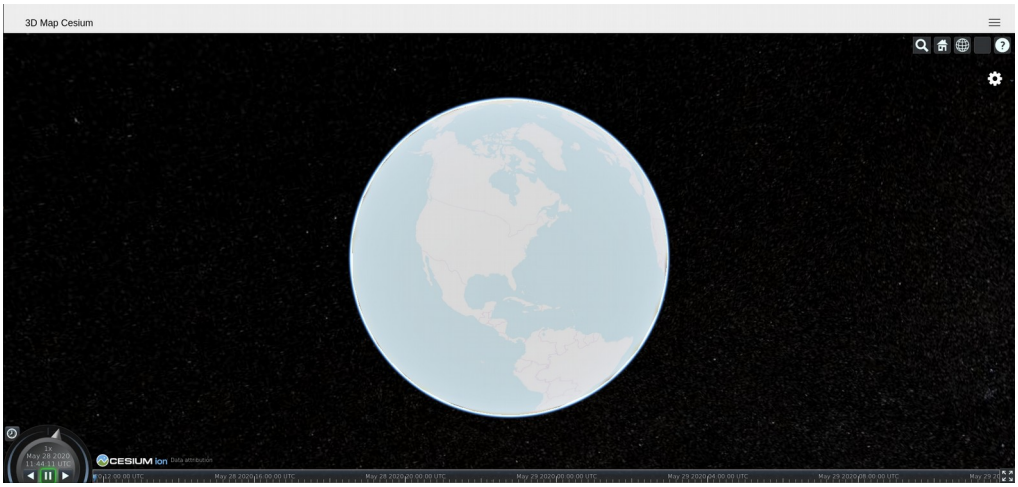
That's it! Now when you reload the application again you should be greeted with the screen shown in the illustration below:

As you may have noticed, the controls at the top right of the screen (encircled in red in the illustration above) can not be clicked. This is because our Navbar overlaps with the buttons. To fix this we need to edit a line of code in our map.component.html file. So let's open the file and edit the div element with the id:"map" according to the illustrations below:

```
<div id="map" style="height: 100vh; width: 100%;"></div>
```

```
<div id="map" style="margin-top:55px; height: 94vh; width: 100%;"></div>
```

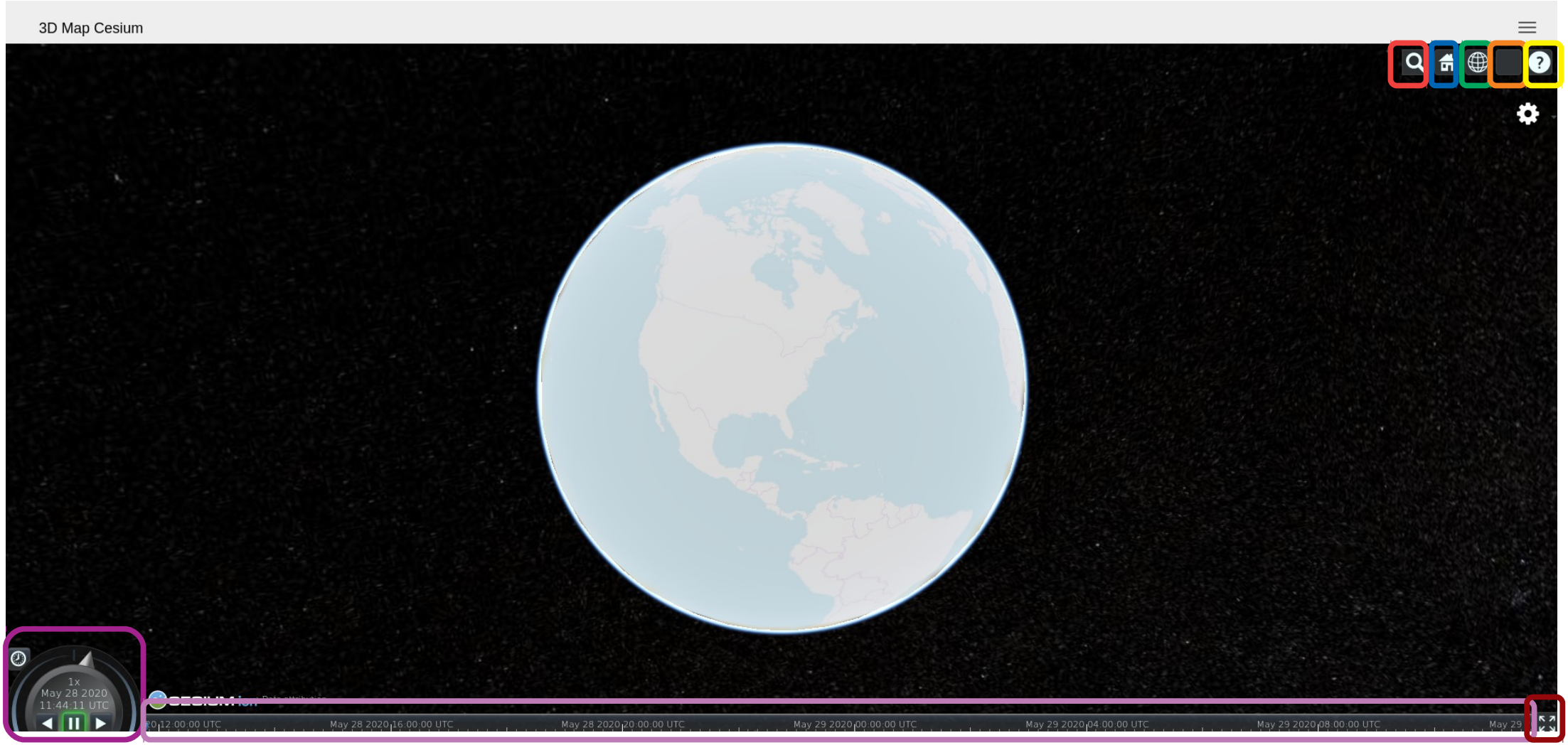
Now if your refresh the page you will be greeted with the following screen in which the buttons are accessible:



In the next section an explanation is given regarding the default buttons in the Cesium Map Viewer.

4.1.2 Explaining the Cesium Viewer

As you may have noticed the Cesium Viewer contains a lot of default options. In this section a description is given regarding the default Cesium Viewer options using the illustration below. The illustration below contains colors which each represent a different functionality which will be explained during this section.



Let's start of with the Cesium Viewer search option. This is the magnifying glass icon in the illustration above encircled in red. This button can be used to search for specific locations around the world. NOTE: This will only work if you have a Cesium ION token which was obtained in section 1.2 of this programming manual.

For example: We have the Elevation data of National park the Hamert in the Netherlands (Translated to: "Landgoed de Hamert" in Dutch). So if we want to zoom in to that location we can click on the magnifying glass and search for the location as shown in the illustration below:

Then click on the option which shows up which will then zoom us in to The Hamert in the Netherlands as shown in the illustration below:

This would also be a good time to check whether our Local Elevation maps are correctly loaded and served by our Cesium Terrain Server. To be able to see height differences in the map we should change the view of the Cesium map. This is done by pressing the Left Ctrl key on your keyboard and pressing the left mouse button while holding the Left Ctrl key down. Then you can drag the map to be able to see differences in elevation.

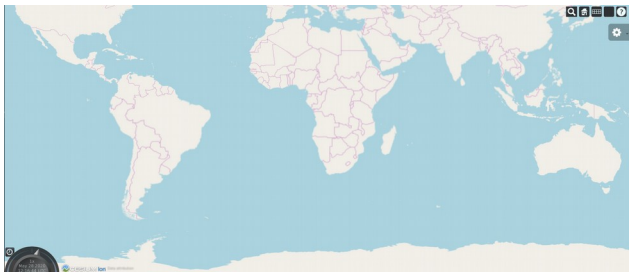
If everything works accordingly your map should look similar to the one shown in the illustration below:

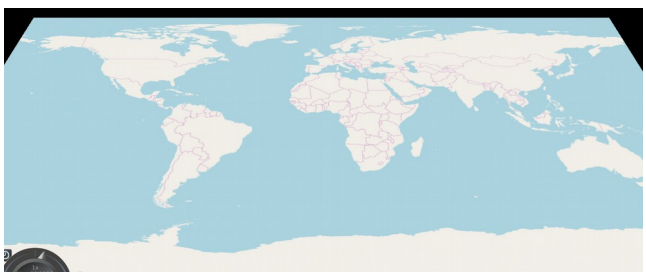
The button next to the magnifying glass (The button with the house icon encircled in blue on page 26) can be used to zoom back to the initial zoom level on which the Viewer was after the map was loaded. So click on the button as shown in the illustration below:

This will result in the map zooming back to the initial zoom level as shown in the illustration below:

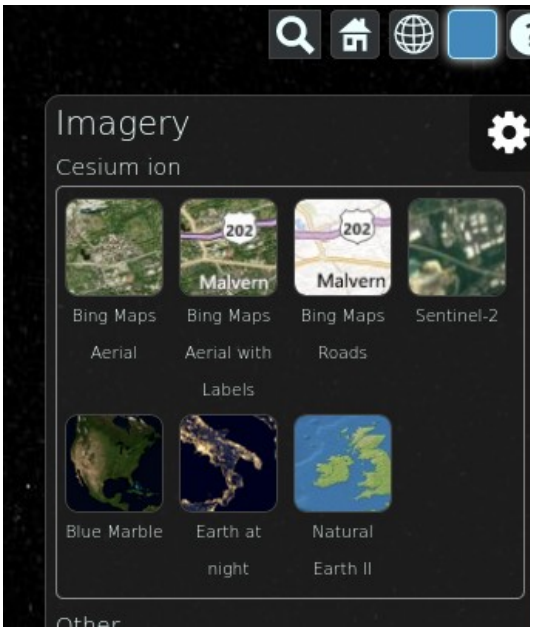
The next button (The globe icon encircled in green on page 26) is used to change the way in which the globe is displayed. So click the button as shown in the illustration below:

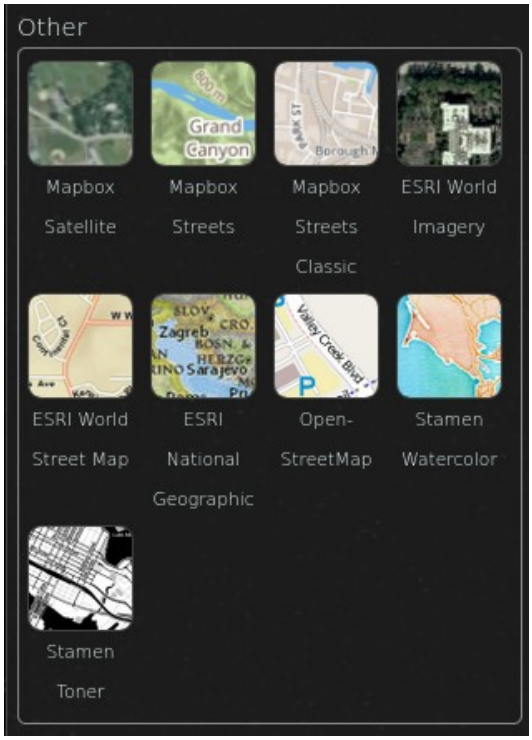
The first option (which is selected by default) makes sure the map is displayed as a globe (top right illustration). The second option makes sure the map is displayed as a 2D Map as shown in the illustration below (middle left illustration) and the third option makes sure the map is displayed as a 3D flat map (bottom right illustration).





The 4th button (without an icon and encircled in orange in the illustration on page 26) is used to change Cesium imagery providers (Tile layers) and Terrain Providers (Elevation Layers). So let's click on the button which will open a drop-down menu containing 3 sections as shown in the illustrations below:



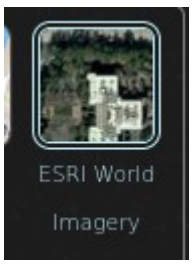


The three sections are as follows:

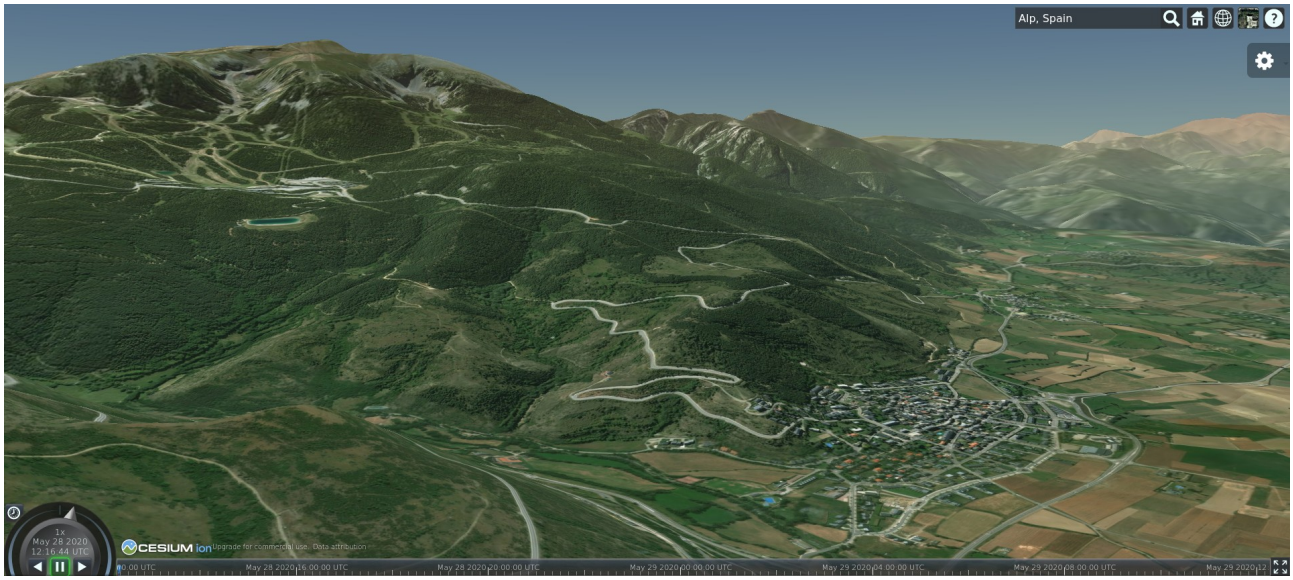
1) Cesium Imagery ION, which are the map providers provided by Cesium ION. Note: To use these map providers you need the Cesium ION Token which we obtained in section 1.2 of this programming manual.
2) Cesium Imagery Other, which are the map providers which can be used without the Cesium ION token.
3) Cesium Terrain ION, which are the Terrain providers provided by Cesium ION. The option: "Cesium World Terrain" can be used to display the elevation maps of the whole globe.

NOTE: To use these options you need an active network connection. This is why we also have our local TileServer (TileStache Tileserver) and Terrain Server (Cesium Terrain Server) which can be used in case you don't have an active network connection.

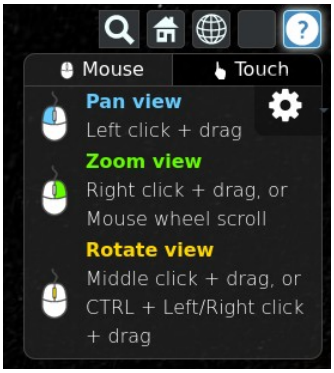
So let's select the ESRI World Imagery and Cesium World Terrain options as shown in the illustrations below:



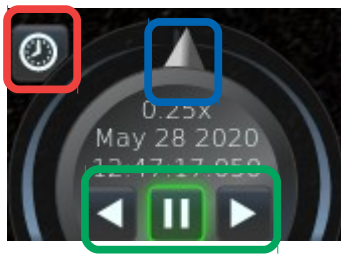
Now let's search for a place (using the magnifying glass icon) called: "Alp" which is a village in Spain close to the Alps. If everything works accordingly you should be able to see mountains and satellite images as shown in the illustration below:



The next button (the button with the question mark icon encircled in yellow in the illustration on page 26) can be used if you forgot how to control the Cesium Map Viewer as shown in the illustration below:



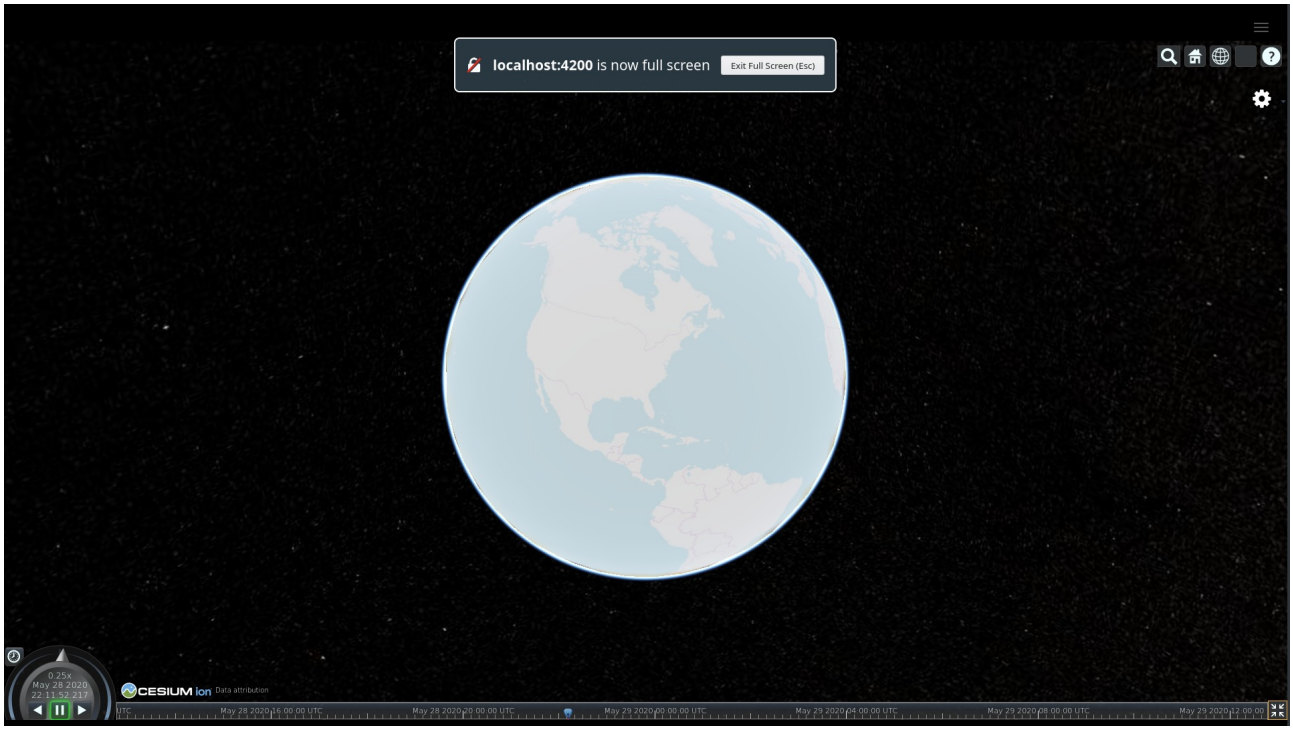
The tool encircled in dark purple in the illustration on page 26 is used to control the animation which we are going to add later. Some more information regarding the tool is as follows:



The clock button (encircled in red in the illustration above) is used to reset the animation. The arrow (encircled in blue) is used to set the speed of the animation. The buttons (encircled in green) are used to rewind, pause / start and fast forward the animation. We are going to use this tool later when adding the animation functionalities to our application.

The timeline bar encircled in light purple in the illustration on page 26. This timeline can also be used to animate the animation by dragging the slider to the left or right as shown in the illustration below:

The last build-in Cesium Viewer functionality is the button encircled in dark red in the illustration on page 26. This button can be used to enable full-screen mode of the Cesium Viewer. When you click the button the result will be the same as shown in the illustration below:



That's it! Now you know what the default functionalities of the Cesium Map Viewer are! Now let's extend these functionalities by adding our own code.

4.2 Switching map providers

At this point we are able to switch between the default WMS's provided by Cesium but we can not switch between our own WMS's which are served by our TileStache server. So let's add this functionality by adding a function called: "setMapProvider()" to our map.component.ts file.

This function is used to switch between the map providers which are served by our TileStache Tileserver.

This function is bound to the buttons (related to switching between map providers) which are defined in the HTML layout of the MapComponent.

The function takes a providerKey as input parameter. The providerKey is the name of the entry that is clicked by the user when selecting a mapProvider from the drop-down list in the application. This providerKey is the key of the entry in the javascript map: "mapProviders".

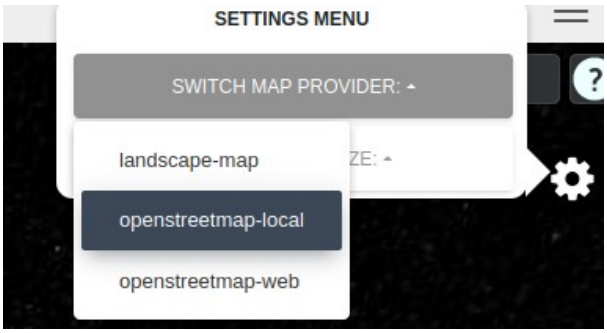
The following steps are executed when the function is triggered:

1) Obtain the current imageryLayers from the Cesium map.
2) Remove all the current imageryLayers from the Cesium map.
3) Create a new OpenStreetMapImageryProvider and set the URL of the imageryProvider using the providerKey that was passed as parameter on the function call. We assign the newly created imageryProvider to the global variable: "mapTileLayer".
4) Add the newly created imageryProvider to the Cesium map.

Now that you know what the function is going to do we should add it to our map.component.ts file. This is done by adding the following code below the function: “getMapProviders()” and above the function: “createCesiumMap()”:

```
setMapProvider(providerKey):void{

  // Here we obtain the current imageryLayers from the Cesium Map.
  var layers = this.map.scene.imageryLayers;

  // Here we remove all the current imageryLayers from the Cesium Map.
  layers.removeAll();

  // Here we create a new OpenStreetMapImageryProvider to which we assign
  // the URL to our TileStache Server running behind the NGINX webserver.
  // We use the providerKey to determine which WMS should be used.
  this.mapTileLayer = new Cesium.OpenStreetMapImageryProvider({
      url : "http://localhost/tiles/"+providerKey+"/"
  });

  // Here we add the new imageryLayer to the Cesium Map.
  layers.addImageryProvider(this.mapTileLayer);
};
```

Now when we refresh the application we are able to switch between our own Web Map Servers which are served by the TileStache Tileserver running behind the NGINX webserver as shown in the illustration below:



We can also switch between the WMS’s provided by Cesium and our own WMS’s if we want to.

4.3 Visualizing Items on the map

At this point we are able to select an item, view the item's information (start date, end date, total datapoints etc.), select per amount, select per DTG (Date Time Groups) and select all the datapoints in a certain country. However when we select an item nothing will be displayed on the map.

This is because we removed all the code in the function: "addLayerGroup()". In this section we are going to add the code required to visualize items on the Cesium Map as shown in the illustration below:

The function: "addLayerGroup()" is going to create 2 types of objects. The first object(s) are the black circles which represent the datapoints on the map as shown in the illustration below:

The second object is the yellow line (which represents the Crane entity) which moves through all the circles when an animation is played as shown in the illustration below:

The entity (the yellow line) will use the positions of the datapoints (the black circles) to determine in which direction it has to move.

The function: “addLayerGroup()’is triggered in the function:”loadItemData()” to make sure a layerGroup is created each time new data is selected.

The following steps are executed when the function:”addLayerGroup” 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 Cesium Map (Viewer) instance to a variable called:”viewer”. We do this so we can simply write viewer instead of this.map.
3) We create an empty list called:”lineLayer” to which we are going to add all the datapoints and entities after the have been created.
4) We assign the value of the dateRangeSelected selected to the variable called: "layerGroupSelector". We do this because the keys (which can be seen as the unique identifier of the layerGroup) in the JavascriptMap are the dateRangeSelected values of each layerGroup.

   We are going to use the variable: "layerGroupSelector" to select, edit and remove specific layerGroups later.
5) A check is performed to see whether a layerGroup with that key (the dateRangeSelected value) already exists in the JavascriptMap: "layerGroups" to make sure we do not add the same datapoints twice.

   If this is the case (so the layerGroupSelector already exists) the selected item’s layerGroup will become the activeLayerGroup.

   If this is NOT the case the following steps will be executed.
6) We obtain and transform the start date of the visualized route into a valid format which is understandable for the Cesium Viewer clock (The animation tool).
7) We obtain and transform the end date of the visualized route into a valid format which is understandable for the Cesium Viewer clock (The animation tool).
8) We calculate the difference (in seconds) between the start and end date of the visualized route.
9) We add the difference in seconds to the start date of the visualized route. We do this because we need the result of this to determine when the animation is finished.
10) We set the settings of the Cesium Viewer Clock (Animation tool) to contain the following settings:
    I. Set the start time of the clock to the start date of the route.
    II. Set the stop time of the clock to the stop date of the route which was calculated by adding the difference (in seconds), between the start and end date, to the start date.
    III. Set the current time of the clock to the start date of the visualized route.
    IV. Set the range of the clock to the start and end date of the visualized route.
    V. Set the clock multiplier (the default speed in which the animation is played) to x200.

VI. Set the range of the timeline at the bottom of the Cesium Viewer to the start date and stop date of the visualized route.

11) Create a nested function called:”generateDataPoints” which is used to create the datapoints (black circles on the map) using the coordinates from the item’s coordinateList and the altitudes from the item’s altitudeList.

The following steps are executed in the nested function:’generateDataPoints()’:

I. Create a new instance of a Cesium SampledPositionProperty which is an object which has an longitude, latitude and altitude.

II. Create a for loop which loops an amount of times equal to the amount of entries in the item’s coordinateList. We are going to create a datapoint for each of the entries in the coordinateList. The following steps are executed in the for loop (so on each of the datapoints):

- Create a new Cesium Julian Date which represents the date on when the datapoint was reached. We assign the created date to a variable called: “timeOfDataPoint”.
- Calculate the difference (in seconds) between the start date and the date the datapoint was reached.
- Create a new Cesium Julia Date by adding the calculated difference (in seconds), between the start date and the date the datapoint was reached, to the start date of the visualized route. We do this because we need to know how long it took for the entity to reach the datapoint. We assign the result to a variable called:”time”.
- Create a new Cartesian3 instance in which we pass the longitude, latitude and altitude values of the datapoint. Cartesian3 is used in Cesium to create the position of a datapoint using the longitude, latitude and altitude values of the datapoint. We assign the result to a variable called:”position”.
- Add a sample to the property object which was created in step 11.I. We pass the calculated time difference as first parameter (time) and the Cartesian3 (position) as second parameter.
- Create a new entity for each datapoint. We set the position of the entity to the Cartesian3 (position) which was created earlier.
- Add the created entity to the list called: “lineLayer” which was created in step 3.

III. Return the created property after the for loop is finished executing. The property will contain a multiple samples which each represent a datapoint.

12) Trigger the nested function: “generateDataPoints()” which was created in the previous step. As mentioned earlier; the function generateDataPoints() will return a property containing samples (which each represent a datapoint). We assign the result (the returned property) of the function:”generateDataPoints()” to a variable called: “position”.

13) Create an new entity (which represents the Yellow line / Crane). We set the following values for the entity:

I. We set the name of the entity to the name of the item which is visualized (e.g. Agnetha)

II. We set the availability of the entity (the time range in which the entity should be displayed on the map) to the startDate and stopDate of the visualized route.

III. We set the position of the entity to the positions generated by the function:"generateDataPoints()" in step 12.

IV. We set the orientation of the entity using a build-in cesium function called: "VelocityOrientationProperty" in which we pass the positions obtained in step 12. This makes sure that the entity moves to the correct position when animating.

V. Create the entity path (Which is the actual yellow line).

14) Set the interpolation options of the entity. We are going to use an interpolation algorithm to make sure the yellow line moves smoothly when animating. The illustrations below show what it will look like without (left illustration) and with (right illustration) an interpolation algorithm:

15) Add the newly created entity to the list: 'lineLayer'.

16) Add the newly created layerGroup to the item's layerGroups.

Now that you know what the function is going to do we should start coding it. So let's open the map.component.ts (if you don't have it open already) and add the following code inside the empty function: "addLayerGroup()":

NOTE: This function is described using 6 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 {

    // Assign this (used to access global variables) to a variable called:
    // _this. We do this because we are going to create nested functions
    // in which we want to access global variables.
    let _this = this;

    // Here we assing the Cesium Map (viewer) instance to a variable called
    // "viewer". We do this so we don't need to write this.map all the time.
    let viewer = this.map

    // Here we create an empty list called: "lineLayer" to which we are
    // going to add all the entities after the have been created.
    let lineLayer = [];

    // Here we set the layerGroupSelector to be the range (Start date ->
```

```
// Here we set the layerGroupSelector to be the range (Start date ->
// End date) of the route. The layerGroupSelector can be seen as the
// unique identifier of the layerGroup.
let layerGroupSelector = item.dateRangeSelected;

// Here we perform a check to see if the layerGroup already exists
// since we don't want to add it twice. If this is not the case (so
// the layerGroup does not exist yet) the following code is executed.
if (!item.layerGroups.has(layerGroupSelector)) {

    // Here we create a new Cesium JulianDate which is the Date format
    // used by the Cesium Clock (Animation tool). We obtain the first
    // entry in the items datetimeList (At index 0) and assign it to a
    // variable called: "startDate".
    let startDate = Cesium.JulianDate.fromDate(
      new Date(item.datetimeList[0]));

    // Here we do the same as above but then for the endDate. We obtain
    // the last item entry in the datetimeList (list length - 1) and
    // assign it to a variable called: 'endDate'.
    let endDate = Cesium.JulianDate.fromDate(
      new Date(item.datetimeList[item.datetimeList.length - 1]));

    // Here we calculate the difference (in seconds) between the start
    // and end date of the visualized route. We do this by passing the end
    // and start dates as parameters.
    let startEndDiff = Cesium.JulianDate.secondsDifference(
      endDate,startDate)

    // Here we create the date of when the route is finished. We do this
    // by adding the difference in seconds (calculated above) to the
    // startDate (The first date in the datetimeList).
    let stopDate = Cesium.JulianDate.addSeconds(
        startDate, startEndDiff,
        new Cesium.JulianDate());

    // Below we set the options of the Cesium clock (animation tool)
    // Here we set the Start DTG of the clock to the startDate.
    viewer.clock.startTime = startDate.clone();
    // Here we set the Stop DTG of the clock to the stopDate.
    viewer.clock.stopTime = stopDate.clone();
    // Here we set the current time of the clock to the startDate.
    viewer.clock.currentTime = startDate.clone();
    // Here we set the option so that the clock loops to the startDate if
    // it reaches the stop Date
    viewer.clock.clockRange = Cesium.ClockRange.LOOP_STOP;
```

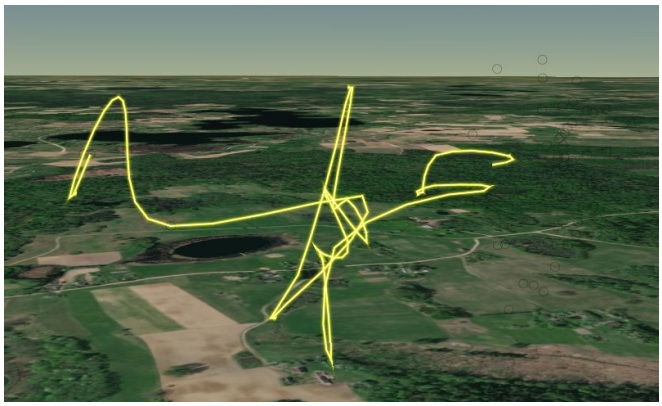
```
        viewer.clock.clockRange = Cesium.ClockRange.LOOP_STOP;
        // Here we set the default clock multiplier to 200. This is the
        // default speed in which the animation is played.
        viewer.clock.multiplier = 200;
        // Here we set the range of the timeline at the bottom of the Cesium
        // viewer to the start and stopDate.
        viewer.timeline.zoomTo(startDate, stopDate);

        // Here we create a nested function called:"generateDataPoints()".
        // This function will create the datapoints (shown as black circles)
        // before the route is animated.
        function generateDataPoints() {

            // Here we create a new instance of a SampledPositionProperty
            // which we assign to a variable called: "property".
            var property = new Cesium.SampledPositionProperty();

            // Here we create a for loop which loops an amount of times equal
            // to the amount of entries in the coordinateList.
            for (var i = 0; i < item.coordinateList.length - 1; i++) {

                // Here we create a new JulianDate representing the DTG on
                // which the datapoint was reached. We pass the value of i (
                // the value on which the loop currently is) to determine what
                // value needs to be obtained from the list.
                let timeOfDataPoint = Cesium.JulianDate.fromDate(
                  new Date(item.datetimeList[i]));

                // Here we calculate the difference in seconds between the
                // start Date an the date when the datapoint was reached.
                let timeDiff = Cesium.JulianDate.secondsDifference(
                  timeOfDataPoint,startDate)

                // Here we add the difference in seconds to the startDate to
                // determine how long it took before the datapoint was reached.
                let time = Cesium.JulianDate.addSeconds(startDate, timeDiff,
                  new Cesium.JulianDate());

                // Here we create a new Cartesian3 in which we pass the
                // longitude (on the first position of the coordinateList
                // entry), latitude (on the second position of the
                // coordinateList entry) and the latitude.
                var position = Cesium.Cartesian3.fromDegrees(
                    item.coordinateList[i][0],
                    item.coordinateList[i][1],
                    item.altitudeList[i] + 10);
```

```
                item.altitudeList[i] + 10);

            // Here we add a new sample to the Cesium
            // SampledPositionProperty in which we pass the calculated time
            // and the position.
            property.addSample(time, position);

            // Here we create the entities for each of the datapoints.
            let entity = viewer.entities.add({
                // we set the position of each entity to the position
                // created earlier.
                position: position,
                // We add a point to the entity (the black circle)
                point: {
                    // We set the size (pixel size) to 10 pixels.
                    pixelSize: 10,
                    // We set the color (the fill color) to transparent.
                    color: Cesium.Color.TRANSPARENT
                }
            });

            // Here we add the created entity to the lineLayer list.
            lineLayer.push(entity)
        }
        // Here we return the created SampledPositionProperty.
        return property;
    };

    // Here we trigger the nested function: "generateDataPoints()" and
    // assign the result to a variable called: "position".
    let position = generateDataPoints();

    // Here we create the entity (yellow line)
    let entity = viewer.entities.add({

        // We set the entity's name to the name of the the item.
        name: item.name,

        // We set the entity availability to the same interval as the
        // startDate and stopDate.
        availability: new Cesium.TimeIntervalCollection(
          [new Cesium.TimeInterval({
            start: startDate,
            stop: stopDate
        })]),
```

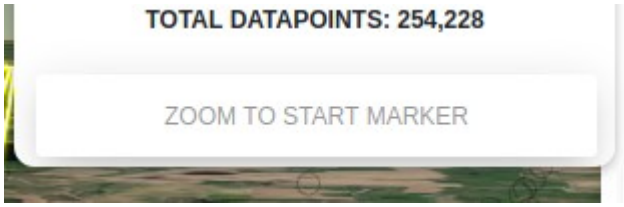
```
        })]),

        // We set the position of the entity to the positions generated
        // by the function generateDataPoints().
        position: position,

        // We Automatically compute the orientation based on position
        // movement.
        orientation: new Cesium.VelocityOrientationProperty(position),

        // Here we add the entity path (which is the yellow line
        // representing) the entity.
        path: {
            // We set the leadTime (the amount of seconds the line should
            // move infront of the entity) to 0 seconds.
            leadTime: 0,
            // We set the trailTime (the amount of seconds) the line
            // should trail behind the entity) 12000 seconds.
            trailTime: 200*60, // 200 minutes, in seconds of simulation time
            // We set the line resolution to 100 pixels.
            resolution: 100,
            // We set the lineStyling to a PolylineGlowMaterialProperty
            // (glowing line).
            material: new Cesium.PolylineGlowMaterialProperty({
                // We set the glowPower to 0.1 (The higher the number the
                // more then line glows).
                glowPower: 0.1,
                // We set the line color to yellow.
                color: Cesium.Color.YELLOW
            }),
            // We set the width of the line to 10 pxiels.
            width: 10
        },
    });

    // Here we add the interpolation options to the entity.
    entity.position.setInterpolationOptions({
        // We set the interpolation degree to 6 (the higher the less sharp
        // the corners are when the entity turns. )
        interpolationDegree: 6,
        // We set the interpolation algorithm to:
        // "LagrangePolynomialApproximation".
        interpolationAlgorithm: Cesium.LagrangePolynomialApproximation
    });

    // Here we add the entity to the list: "lineLayer".
```

```
        // Here we add the entity to the list: "lineLayer".
        lineLayer.push(entity)

        // Here we add the newly created layerGroup to the item's layerGroups.
        item.layerGroups.set(layerGroupSelector, {
          'lineLayer': {
            // We set the lineLayer as layer property.
            'layer':lineLayer,
            // We set the layer's start date as stopDate property.
            'startDate':startDate,
            // We set the layer's stop date as stopDate property.
            'stopDate':stopDate,
            // We set the  coordinateList as datapoints.
            'datapoints': item.coordinateList
          }
        })
    };

    // Here we set the newly created layerGroup to be the active layergroup.
    this.setLayerGroup(layerGroupSelector);
};
```

That's it! Now when you refresh the application and select an item you are able to see the datapoints and animate the datapoints as shown in the illustration below:



As you may have noticed the functionality required to zoom to the start location of a visualized route does not yet work as shown in the illustration below :



In the next section we are going to create the function required to zoom to the start position of a visualized route.

4.4 Zooming to a start location

At this point we are able to visualize routes which are selected. The problem is that we can not yet easily zoom to the start location of a selected route. We are going to animate the zooming functionality by using the Cesium Camera. The Cesium Camera offers a lot of possibilities and can be used to create very cool animations. For more information related to the Cesium Camera you should visit the following URL:

https://cesium.com/docs/cesiumjs-ref-doc/Camera.html

To be able to do this we are going to create a function called: “zoomToLocation()”. This function is bound to the “Zoom to Start” button which we defined in the MapComponent HTML layout page in the programming manual: “Creating a 2 Dimensional Map Viewer application”.

The function gets the Cesium Viewer Scene and animates it to move to the start coordinates of the item on which the: "Zoom to start marker" button is clicked.

We use the build in Cesium function called: ".flyTo()" to zoom to a given location.

In this function we set the destination to which has to be zoomed to the start longitude and latitude coordinates of the active Item.

We also set the duration which defines the amount of time (in seconds) it takes for the animation to complete.

Let’s add the function by adding the following code below the function: “createCesiumMap()” which we defined earlier:

```
zoomToLocation():void{

  // Here we obtain the Cesium Viewer Scene camera an call the function:
  // "flyTo" on the camer.
  this.map.scene.camera.flyTo({
    // Here we set the destination to a Cartesian3 in which we pass the
    // longitude, latitude and a predefined altitude value (1000).
    // The higher the altitude value the more zoomed out the map will be.
    destination: Cesium.Cartesian3.fromDegrees(
      this.activeItem.startCoordinate[0],
      this.activeItem.startCoordinate[1],
      1000),
    // We set the duration of the animation to 3 seconds.
    duration: 3,
  });
};
```

That’s it! Now you are able to zoom in to the start position of a visualized route by clicking on the button: “Zoom To Start Marker”.

4.5 Setting layerGroups

At this point we are not yet able to switch between selected layerGroups. We can select a new layer group by for example selecting a different DTG but we cannot switch back to an previously selected layer group. In this section you will learn how to switch between visualized layerGroups in Cesium.

The function we are going to create 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.
2) addLayerGroup(), when a layer group is added we need to set the added layer group to be the activeLayerGroup.

This function is used to set the active layer group and update the Cesium Viewer Clock according to the select layer group values (Start and stop date).

The following steps are executed when the function is triggered:

1) Assign the activeItem to a variable called item.
2) Assign the Cesium Map Viewer instance to a variable called viewer.
3) Set the item’s daterange to the groupKey which is passed as input parameter on the function call.
4) Set the item’s activeLayerGroup to the newly selected layergroup.
5) Set the item’s coordinateList to the layerGroup's datapoints.
6) Set the item’s startCoordinate to the layerGroup's startCoordinate.
7) Set the Cesium Clock start time to the layerGroup's start date.
8) Set the Cesium Clock end time to the layerGroup's stop date.
9) Set the Cesium Clock currentTime to the layerGroup's start date.
10) Set the Cesium Timeline range to the interval between the item’s start time and stop time.

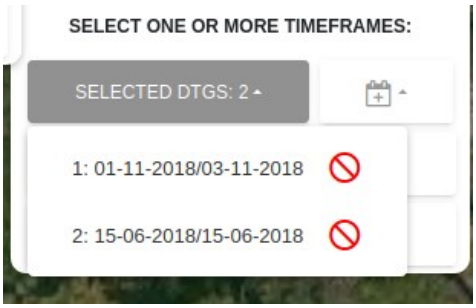
Now that you know what the function: “setLayerGroup()” is used for we can start coding the function which is empty as of now in our map.component.ts file. The code required to create this function is shown in the illustrations below:

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 activeItem to a variable called: "item".
  let item = this.activeItem

  // Here we assign the Cesium Map Viewer instance to a variable called:
  // "viewer".
  let viewer = this.map;
```

```
  let viewer = this.map;

  // Here we assign the groupKey (passed as input parameter) as the item's
  // selected date range (start -> end date).
  item.dateRangeSelected = groupKey;

  // Here we set the items activeLayerGroup to the layerGroup from which
  // the key was passed as parameter.
  item.activeLayerGroup = item.layerGroups.get(groupKey)

  // Here we set the items coordinateList to the datapoints of the newly
  // selected layerGroup.
  item.coordinateList = item.activeLayerGroup.lineLayer.datapoints

  // Here we set the item's startCoordinate to the first datapoint of the
  // newly selected layerGroup.
  item.startCoordinate = item.activeLayerGroup.lineLayer.datapoints[0]

  // Below we set the options of the Cesium clock (animation tool)
  // Here we set the Start DTG of the clock to the startDate of the newly
  // selected layerGroup.
  viewer.clock.startTime = item.activeLayerGroup.lineLayer.startDate.clone();

  // Here we set the Stop DTG of the clock to the stopDate of the newly
  // selected layerGroup.
  viewer.clock.stopTime = item.activeLayerGroup.lineLayer.stopDate.clone();

  // Here we set the current time of the clock to the startDate of the newly
  // selected layerGroup.
  viewer.clock.currentTime = item.activeLayerGroup.lineLayer.startDate.clone();

  // Here we set the range of the timeline at the bottom of the Cesium
  // viewer to the start and stopDate of the newly selected layerGroup.
  viewer.timeline.zoomTo(item.activeLayerGroup.lineLayer.startDate,
    item.activeLayerGroup.lineLayer.stopDate);
};
```

That's it! Now when you refresh the application, select an item and select a new DTG you will be able to switch back to the first layerGroup as shown in the illustration below:

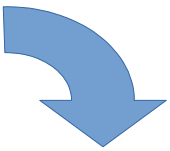


At this point switching between layerGroups only works when you select one of the DTG's in the selected DTG's drop-down box. We cannot switch back and forth between items yet. To be able to do this we need to edit the function: "selectItem()" which was created in the programming manual:"Creating a 2 Dimensional Map Viewer ".

We need to edit the line which checks if an item is already in the list of selected items. As of now nothing happens when an item is selected for the second time. We want to make sure that when an item is selected for the second time the function: "addLayerGroup()" is triggered. This function will not add a new layerGroup since it already exists, however the function will set the the activeLayerGroup to the item's active layerGroup and thus setting the start en stop time of the Cesium clock to the correct values.

So let's go to the function: "selectItem()" in the map.component.ts file and edit the line starting with: "this.selectedItems.Filter(...' according to the illustrations below:

```
this.selectedItems.filter(
  data => data.id.includes(item.id)).length == 1 ? null
  : (this.getInitalItemData(item),this.selectedItems.push(item))
```



```
this.selectedItems.filter(
    data => data.id.includes(item.id)).length == 1 ?
    this.addLayerGroup(this.activeItem) :
    (this.getInitalItemData(item), this.selectedItems.push(item))
```

That's it! Now if we want to switch between selected items, the Cesium Clock and timeline will also be updated according to the selected item's activeLayerGroup.

4.6 Removing entities from the map

The last thing we need to do before finishing the 3D Map Viewer application is making sure that, when an Item or an Item's layerGroup is removed, the entities belonging to the item or layerGroup are also removed from the Cesium Map.

The first function we need to edit is the function related to removing a selected Item. So let's go to that function in the map.component.ts file and add the following line below the statement which checks whether the item that is being removed is the activeItem and above the line which is used to clear the item's layerGroups:

NOTE: The statement related to checking whether the item that is being removed is the activeItem is also included in the illustration, you don't need to add this again!

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

// Here we loop through each of the item's layerGroups.
item.layerGroups.forEach(layerGroup => {
  // For each entry in the in the in the item's layerGroups JavaScriptMap
  // the following code is executed.
  for (let [key, value] of Object.entries(layerGroup)) {
    // We obtain the value of the layerGroup property: "layer" which
    // contains the list of entities belonging to the layerGroup.
    value['layer'].forEach(entity => {
        // Here we remove each entity in the list of entities from the
        // Cesium Map.
        this.map.entities.remove(entity)
    });
  }
});
```

The next function we need to edit is the function:"removeLayerGroup()". So let's scroll back to that function and add the following code below the line starting with: "let groupToRemove = item.layerGroups ….":

```
// For each entry in the groupToRemove's dictionary (which contains the
// layer entities, layer start date, layer end date and the layers datapoints)
// the following code is executed.
for (let [key, value] of Object.entries(groupToRemove)) {
  // We obtain the value of the layerGroup property: "layer" which
  // contains the list of entities belonging to the layerGroup.
  value['layer'].forEach(entity => {
      // Here we remove each entity in the list of entities from the
      // Cesium Map.
      this.map.entities.remove(entity)
  });
};
```

That's it! Now the entities belonging to an item or layerGroup which is being removed are also removed from the map.