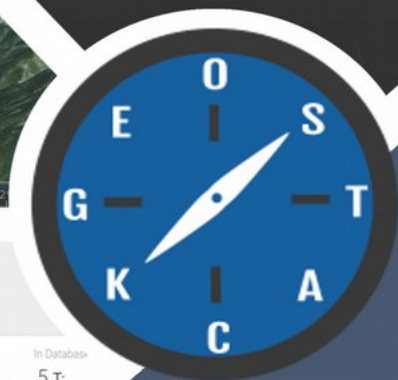
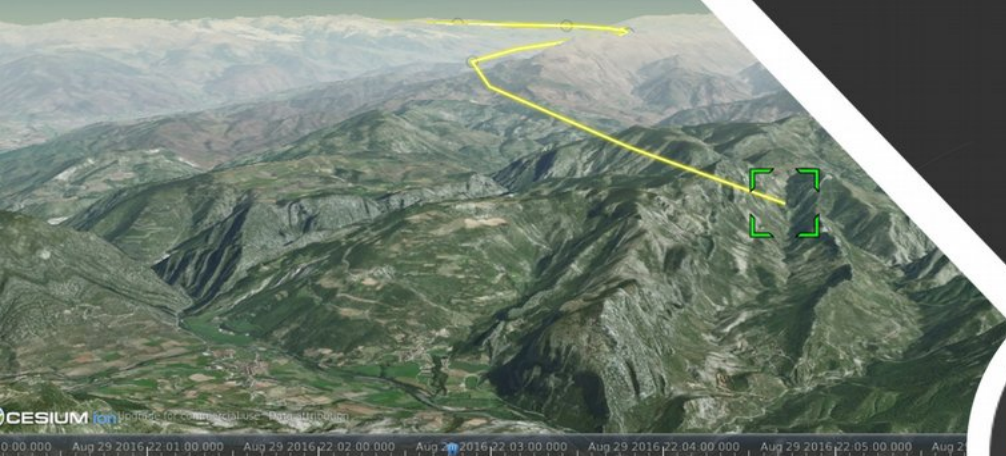


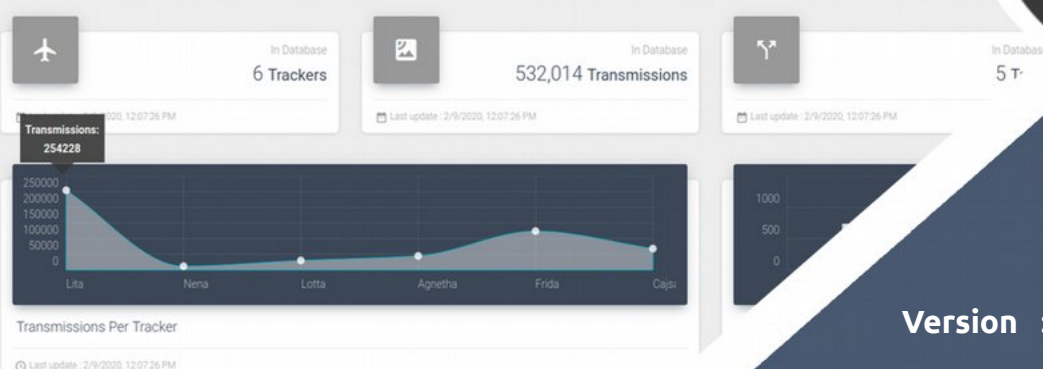


Programming Manual

CREATING A DATASET DASHBOARD



GPS Dashboard



Version : 1.0

Date : 02-10-2020

Author : The GeoStack Project

License : CC BY 4.0

Open Content License

This work is licensed under the Creative Commons Attribution 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

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 a Dataset Dashboard web application with the Angular JavaScript framework as an extension of our Angular Base Application.

The Angular apps will perform API calls to the API of 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.

For more in depth information related to AngularJS see: <https://angular.io/docs>

Some code is explained using the inline comments in the source code. The source code can be found in the folder: "POC" which is located in the same folder as this document.

NOTE: it's highly recommended you use the source code when creating the application yourself and learn from the inline comments!

A special thanks to the following people and organizations:

- The people and contributors from AngularJS for providing an awesome and easy to use JavaScript framework.
- The people from Creative Tim for providing some styled components which are used in the application.
- The people and contributors from all the extra packages used in the applications created during this programming manual.

Table of Contents

1 Getting ready.....	4
2 Creating the service.ts file.....	5
3 Creating the GPS Dashboard page.....	8
3.1 Retrieve the data from our MongoDB datastore.....	11
3.2 Displaying the first row of cards with data.....	14
3.3 Displaying interactive graphs in the dashboard.....	19
3.4 Creating an interactive table.....	25
3.5 Generating pandas data profile.....	32

1 Getting ready

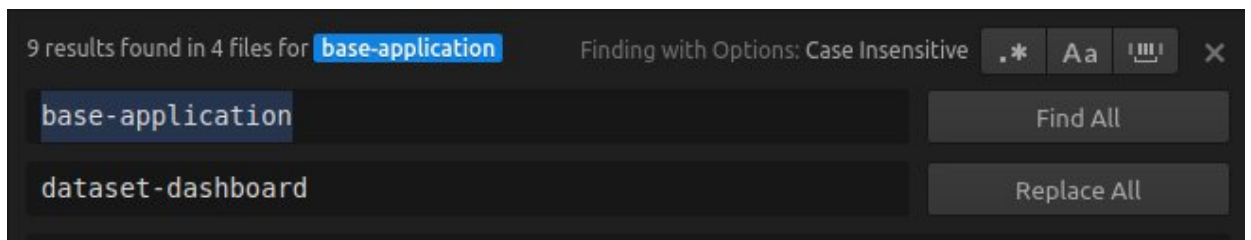
This application is an extension of the base application so we can copy this application and start creating the Dataset Dashboard using this application as base.

After we copied the base-application folder, we need to change some names and titles to make the new application the dataset dashboard. We start by changing the name of the folder we just copied from base-application to dataset-dashboard, as shown in the image below.

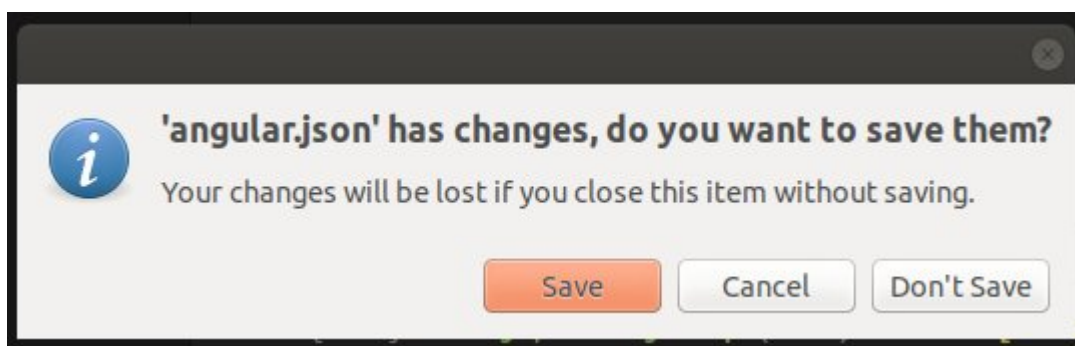


We also need to edit the project name: “base-application” to “dataset-dashboard”. If you are using the code-editor Atom, this is done by performing the following steps:

- 1) In the editor press the keys Ctrl + shift + f
- 2) In the screen that pops up enter: “base-application” in the find section and “dataset-dashboard” in the replace section, as shown in the illustration below.



- 3) Click on replace all and on the save button in the screens that pop up



- 4) In the file: index.html located in the folder dataset-dashboard/src, rename the title from BaseApplication to Dataset Dashboard.
- 5) In the file: sidebar.component.html located in the folder src/app/components/sidebar/, change the text: “Base Application” to “Dataset Dashboard”

2 Creating the service.ts file

In the folder `dataset-dashboard/src/app/services`, create a new service called: "gpsdashboard.service.ts" by running the following command:

```
touch ~/Geostack/dataset-dashboard/src/app/services/ gpsdashboard.service.ts
```

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

```
import { Injectable } from '@angular/core'
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs'
```

Next we need to create an inject able class called `GPSDashboardService`. How this is done is shown in the illustration below.

```
@Injectable()
export class GPSDashboardService {
}
```

In this class we need to create a constructor in which we will pass an instance of the `HttpClient`. This is required to perform HTTP requests to our Flask-API. How this is done is shown in the illustration below.

```
constructor(private http: HttpClient) { }
```

Now we are going to create some functions which will be used to perform API calls to our Flask-API. These functions will be called in our `GPSDashboard`.

The first function we are going to create is a function which retrieves all trackers from our datastore. The function is called: "getTrackers()" and is used to perform an HTTP GET request to our Flask-API.

The function performs a request on the following URL: <http://localhost:4200/api/trackers/>.

This URL will be transformed to: <http://localhost/api/trackers/>, which is our Flask-API, because of the proxy configuration which we added in the previous programming manual.

This URL is bound to a function in our Flask-API. The function bound to this URL executes the query on our MongoDB datastore which retrieves all trackers from the MongoDB datastore.

The code for creating this function is shown in the illustration below.

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


Now we need to create a function called: "getTracker()", which 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 GPSDashboard. The code for creating this function is shown in the illustration below.

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

From here on the functions will be explained using the inline comments found in the source code, provided in the folder: "POC".

```
/*  
Here we create a function called: "getTransmissionCount()", which is used to  
perform an HTTP GET request to our Flask-API.  
  
The function performs a request on the following URL:api/transmissions_count/  
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 the total amount of transmissions from the MongoDB datastore.  
  
The function:"getTransmissionCount()" then returns the amount to our  
GPSDashboard.  
*/  
getTransmissionCount(): Observable<any[]> {  
  return this.http.get<any[]>(`api/transmissions_count/`)  
}
```

Now we need to create a function which returns all our GPS-Routes. The code for this is shown in the illustration below.

```
/*  
Here we create a function called: "getTrails()", which is used to perform an  
HTTP GET request to our Flask-API.  
  
The function performs a request on the following URL:api/trails/.  
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:"getTrails()" then returns all the trails to our GPSDashboard.  
*/  
getTrails(): Observable<any[]> {  
  return this.http.get<any[]>('api/trails/')  
}
```

Now we need to create a function which returns a specific trail using the id passed in the function. The code for this is shown in the illustration below.

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

The function performs a request on the following URL:api/trails/{id}.
This URL is bound to a function in our Flask-API. The function, bound to this
URL, executes a query on our MongoDB datastore and retrieves a trail which
has the id passed in this function,from the MongoDB datastore.

The function:"getTrail()" then returns the trail to our GPSDashboard.
*/
getTrail(id:string): Observable<any[]> {
    return this.http.get<any[]>(`api/trails/${id}`)
}
```

The last function which we are going to create is used to retrieve the total amount of signals from our datastore. The code for this function is shown in the illustration below.

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

The function performs a request on the following URL:api/signals_count/
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 the total amount of signals from the MongoDB datastore.

The function:"getSignalCount()" then returns the amount to our
GPSDashboard.
*/
getSignalCount(): Observable<any[]> {
    return this.http.get<any[]>(`api/signals_count/`)
}
```

3 Creating the GPS Dashboard page

Now let's create the GPS Dashboard page. For this we first need to create a folder in the folder "/dataset-dashboard/src/app/pages" called: "gpsdashboard". We do this by running the following command:

```
mkdir ~/Geostack/angular-apps/dataset-dashboard/src/app/pages/gpsdashboard
```

In this folder create a file called: "gpsdashboard.component.ts" and a file called: "gpsdashboard.component.html"

We start of by importing the required modules in our gpsdashboard.component.ts file.

So let's open this file and add the following code at the top of the file.

```
import { Component, OnInit } from '@angular/core';

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

import { GPSDashboardService } from '../../services/gpsdashboard.service'
```

The imported modules are as follows:

- ➔ The base angular modules
- ➔ The module for creating the interactive charts
- ➔ The module for showing tool-tips in those charts
- ➔ The service we need to perform AP calls to retrieve data.

Now we need to define the metadata of the GPSDashboard component. How this is done is shown in the illustration below:

```
/*
Here we create the component metadata. The following applies to this code:
1) selector: If we want to use the GPSDashboard component, we add the code:
   <app-gpsdashboard/> 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.
*/
@Component({
  selector: 'app-gpsdashboard',
  templateUrl: './gpsdashboard.component.html',
  providers: [GPSDashboardService]
})
```


Now we need to create the basic class structure of the GPSDashboard component. We also create a global variable called currentDate which is used to show when the data displayed in the dashboard was last updated. The code for this is shown in the illustration below.

```
/*
Here we create the component class called:"GPSDashboard".
*/
export class GPSDashboard implements OnInit {

    /*
    Assign the current date to a global variable of type string(currentDate).
    This variable is used to check when the application / Data was last updated.
    */
    private currentDate:string = new Date().toLocaleString()

    /*
    Here we create the class constructor of the MapComponent. We pass the
    GPSDashboardService in the constructor. We assign the service to a variable,
    this variable can be reused throughout the whole component. We use the
    variable to call the functions in our service which will then perform API
    calls to our Flask-API.
    */
    constructor(private _GPSDashboardService:GPSDashboardService) {}

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

Now we need to add the GPSDashboard to our app.module.ts, so let's open this file and add an import to the top of the file. The code for this is shown in the illustration below.

```
import { GPSDashboard } from '../app/pages/gpsdashboard/gpsdashboard.component';
```

Then we need to add the GPSDashboard 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,
    GPSDashboard
],
```

Now we need to add a new route to our app-routing.module.ts file. So let's open this file. First we need to import the GPSDashboard in this file. This is done by adding the following code to the top of that file.

```
import { GPSDashboard } from '../app/pages/gpsdashboard/gpsdashboard.component';
```

Then we need to add a new route to our routelist. The code for this is shown in the illustration below.

```
{ path: 'gpsdashboard', component: GPSDashboard},
```

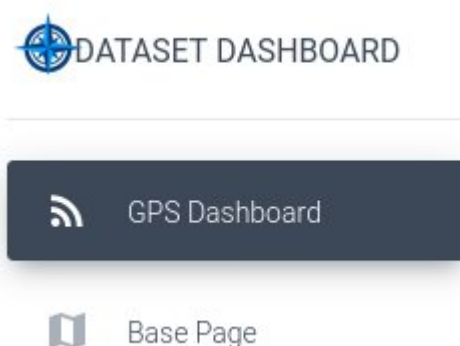
If you want you can change the route which redirects us to the base-page, to redirect us to the gpsdashboard. If you choose to do this the final Route List will look the same as in the illustration below.

```
const routes: Routes = [  
  { path: 'gpsdashboard', component: GPSDashboard},  
  { path: 'base-page', component: BaseComponent},  
  { path: '', redirectTo: 'gpsdashboard', pathMatch: 'full'},  
];
```

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 Routes List. The final code will be the same as shown in the illustration below.

```
export const ROUTES: RouteInfo[] = [  
  { path: '/gpsdashboard', title: 'GPS Dashboard', icon: 'rss_feed', class: ''},  
  { path: '/base-page', title: 'Base Page', icon: 'map', class: '' },  
];
```

Now when we go to our application and open the sidebar we will have 2 entries as shown in the illustration below.



When clicking on the GPS Dashboard entry the page will still be empty. This is because we did not add anything to the layout of the GPS Dashboard. We are going to do this later on but let's first make sure we can obtain data from our MongoDB datastore.

3.1 Retrieve the data from our MongoDB datastore

To retrieve, store and use data, from our MongoDB datastore, in our GPSDashboard we have to start of by defining some global variables. The global variables are going to contain the data retrieved by the functions in our GPSDashboard service. Because the variables are global variables, we can use them throughout the whole GPSDashboard component.

We are going to create a JavaScript map for the Crane trackers and for GPS-Route datasets (Trail). The code for creating the Crane map is shown below.

```
Here we define a global variable called: "craneMap". This variable is a JavascriptMap which stores key/values. The JavascriptMap starts of empty but will be populated with the trackers retrieved from the MongoDB datastore.
*/
private craneMap: Map < string, any[] > = new Map([])
```

The code for creating the Trail map is shown below.

```
/*
Here we define a global variable called: "trailMap". This variable is a JavascriptMap which stores key/values. The JavascriptMap starts of empty but will be populated with the trails retrieved from the MongoDB datastore.
*/
private trailMap: Map < string, any[] > = new Map([])
```

Now we also need to create 2 global variables to which we will assign the total amount of transmissions and signals in our datastore. The code for this is shown below.

```
/*
Here we define a global variable called: "transmissionCount", to which the total amount of transmissions in our datastore will be assigned.
*/
private transmissionCount: any[];

/*
Here we define a global variable called: "signalsCount", to which the total amount of signals in our datastore will be assigned.
*/
private signalCount: any[];
```

Next we need to create the functions which will trigger the API Call functions in our CraneService file which we created in chapter 2.

So we need to define the functions that retrieve all the data from our MongoDB datastore. These functions are as follows:

- ➔ `getTrackers()` which is used to obtain all the (Crane) trackers from our datastore.
- ➔ `getTotalTransmissions()` which is used to obtain the total amount of transmissions.
- ➔ `getTrails()` which is used to obtain all trails (GPS Routes) from our datastore.
- ➔ `getTotalSignals()` which is used to obtain the total amount of signals.

The syntax used in these functions is as follows:

```
this.{service}.{function}.subscribe({elements} =>
  {elements}.forEach({element} =>{
    this.{JavascriptMap}.set(element['name'],element)
  })
)
```

The following applies to the code block 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` function 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. We want to add the element/item to the `JavascriptMap` using the name of the element as the key and the element itself as the value.

In the case of a Crane tracker, an entry in the `JavascriptMap` (`craneMap`) would be: {key = "Agnetha", value = [coordinates, study_name, etc..]}

The code for creating the `getTrackers()` function is shown in the illustration below:

```
This function is used to call the function getTrackers() in our
GPSDashboardService, which will then return all the trackers and add them
to the global JavascriptMap called: "craneMap".
*/
getTrackers(): void {
  this._GPSDashboardService.getTrackers().subscribe(trackers =>{
    trackers.forEach(tracker => {
      this.craneMap.set(tracker['name'], tracker)
    }),
  );
};
```


The code for creating the `getTotalTransmissions()` function is shown in the illustration below:

```
This function is used to call the function getTransmissionCount() in our
GPSDashboardService, which will then return the total amount of
transmission in our MongoDB datastore and the value to the to the global
variable transmissionCount.
*/
getTotalTransmissions():void{
  this._GPSDashboardService.getTransmissionCount().subscribe(
    count => this.transmissionCount = count
  );
};
```

The code for creating the `getTrails()` function is shown in the illustration below:

```
This function is used to call the function getTrails() in our
GPSDashboardService, which will then return all the trails and add them
to the global JavascriptMap called: "trailMap".
*/
getTrails(): void {
  this._GPSDashboardService.getTrails().subscribe(trails => (
    trails.forEach(trail => {
      this.trailMap.set(trail['name'], trail)
    })),
  );
}
```

The code for creating the `getSignals()` function is shown in the illustration below:

```
/*
Here we create the function: "getTotalSignals()".

This function is used to call the function getSignalCount() in our
GPSDashboardService, which will then return the total amount of
signals in our MongoDB datastore and the value to the to the global
variable signalCount.
*/
getTotalSignals():void{
  this._GPSDashboardService.getSignalCount().subscribe(
    signalCount => this.signalCount = signalCount
  );
}
```

Now we need to add these functions to the `ngOnInit()` function to make sure that the functions get triggered when the component and thus the dashboard is loaded. The final `ngOnInit()` function will look the same as shown in the illustration below.

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

Now when you start the Flask-API and you load the web application the data will be obtained from our MongoDB datastore. You can confirm this by opening the Firefox element inspection and the inspect the network traffic. You will see something similar as shown in the illustration below:

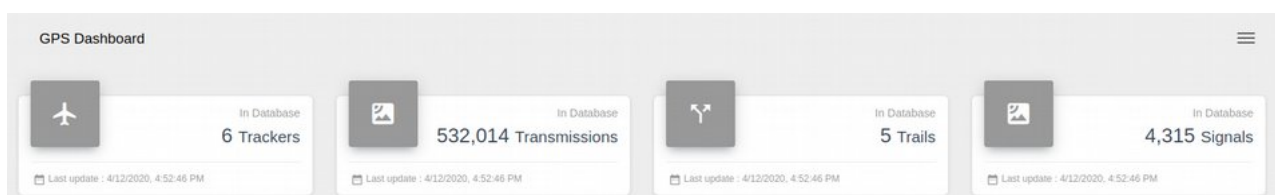
200	GET	localhost:4200	/api/trackers/
200	GET	localhost:4200	/api/transmissions_count/
200	GET	localhost:4200	/api/trails/
200	GET	localhost:4200	/api/signals_count/

We did not add anything to the layout of our dashboard so we will not see anything yet so let's do this in the next chapter.

3.2 Displaying the first row of cards with data

At this point we have a working dashboard that obtains the data from our MongoDB datastore. The problem is that we don't have anything in our HTML layout file yet. So let's add some cards that are going to contain the data that is retrieved by the functions we created in the previous chapter.

The final result will look the same as shown in the illustration below:



First we will add the basic layout of our dashboard page. This is done by adding the following code to our `gpsdashboard.component.html` file:

```
<div class="container-fluid" style='padding-top:80px;'>
  <div class="row">
  </div>
</div>
```

The `div` element with the class: `"container-fluid"`, makes sure that everything in the `div` element has a specific width. The `div` element with the class: `"row"`, makes sure that everything in that `div` is displayed next to each other.

All the cards we are going to create are created inside the `div` element with the class: `"row"`.

We are going to start of with the card that displays the amount of trackers in the database. The code for this is shown in the illustration below.

```
<!-- This section is related to tracker card in the dashboard-->
<div class="col-lg-3 col-md-6 col-sm-6">
  <div class="card card-stats">
    <!-- Below we create the header of the card.
    This is the top section where the icon and the data is displayed-->
    <div class="card-header card-header-primary card-header-icon">
      <!-- Here we create a new div element in which we create an
      icon. -->
      <div class="card-icon">
        <i class="material-icons">flight</i>
      </div>
      <!-- Here we create the sub text of the card.-->
      <p class="card-category">In Database</p>
      <!-- Here we create the main text of the card.
      we obtain the amount of trackers by retrieving the size
      of the JavaScriptMap: "CraneMap" which is defined in the
      gpsdashboard.component.ts file.-->
      <h3 class="card-title">{{craneMap.size}}
        <small>Trackers</small>
      </h3>
    </div>
    <!-- Below we create the footer of the card.-->
    <div class="card-footer">
      <div class="stats">
        <!-- Here we add the date on which the dashboard was last
        updated to the card by obtaining the global variable "currentDate"
        which is defined in the gpsdashboard.component.ts file. -->
        <i class="material-icons">date_range</i> Last update : {{currentDate}}
      </div>
    </div>
  </div>
</div>
```


Now we are going to create the card that displays the amount of transmissions in the database. The code for this is shown in the illustration below, we add the code below the div element representing the card for the amount of trackers in the database.

```
<!-- This section is related to the transmission card in the dashboard-->
<div class="col-lg-3 col-md-6 col-sm-6">
  <div class="card card-stats">
    <!-- Below we create the header of the card.
    This is the top section where the icon and the data is displayed-->
    <div class="card-header card-header-primary card-header-icon">
      <!-- Here we create a new div element in which we create an
      icon. -->
      <div class="card-icon">
        <i class="material-icons">satellite</i>
      </div>
      <!-- Here we create the sub text of the card.-->
      <p class="card-category">In Database</p>
      <!-- Here we create the main text of the card.
      we obtain the amount of transmissions by retrieving the
      global variable: "transmissionCount" which is defined in the
      gpsdashboard.component.ts file.-->
      <h3 class="card-title">{{ transmissionCount | number: '2.' }}
        <small>Transmissions</small>
      </h3>
    </div>
    <!-- Below we create the footer of the card.-->
    <div class="card-footer">
      <div class="stats">
        <!-- Here we add the date on which the dashboard was last
        updated to the card by obtaining the global variable "currentDate"
        which is defined in the gpsdashboard.component.ts file. -->
        <i class="material-icons">date_range</i> Last update : {{currentDate}}
      </div>
    </div>
  </div>
</div>
```


Now we are going to create the card that displays the amount of GPS-Routes (Trails) in the database. The code for this is shown in the illustration below, we add the code below the div element representing the card for the amount of transmissions in the database.

```
<!-- This section is related to the trail card in the dashboard-->
<div class="col-lg-3 col-md-6 col-sm-6">
  <div class="card card-stats">
    <!-- Below we create the header of the card.
    This is the top section where the icon and the data is displayed-->
    <div class="card-header card-header-primary card-header-icon">
      <!-- Here we create a new div element in which we create an
      icon. -->
      <div class="card-icon">
        <i class="material-icons">call_split</i>
      </div>
      <!-- Here we create the main text of the card.
      we obtain the amount of trails by retrieving the size
      of the JavaScriptMap: "trailMap" which is defined in the
      gpsdashboard.component.ts file.-->
      <p class="card-category">In Database</p>
      <h3 class="card-title">{{trailMap.size}}
        <small>Trails</small>
      </h3>
    </div>
    <!-- Below we create the footer of the card.-->
    <div class="card-footer">
      <div class="stats">
        <!-- Here we add the date on which the dashboard was last
        updated to the card by obtaining the global variable "currentDate"
        which is defined in the gpsdashboard.component.ts file. -->
        <i class="material-icons">date_range</i> Last update : {{currentDate}}
      </div>
    </div>
  </div>
</div>
```

The last card we are going to create is the card that displays the total amount of signals in the database. The code for this is shown in the illustration below, we add the code below the div element representing the card for the amount of trails in the database.

```
<!-- This section is related to the signal card in the dashboard-->
<div class="col-lg-3 col-md-6 col-sm-6">
  <div class="card card-stats">
    <!-- Below we create the header of the card.
    This the top section where the icon and the data is displayed-->
    <div class="card-header card-header-primary card-header-icon">
      <!-- Here we create a new div element in which we create an
      icon. -->
      <div class="card-icon">
        <i class="material-icons">satellite</i>
      </div>
      <!-- Here we create the main text of the card.
      we obtain the amount of signals by retrieving the
      global variable: "signalCount" which is defined in the
      gpsdashboard.component.ts file.-->
      <p class="card-category">In Database</p>
      <h3 class="card-title">{{signalCount | number: '2.'}}
        <small>Signals</small>
      </h3>
    </div>
    <!-- Below we create the footer of the card.-->
    <div class="card-footer">
      <div class="stats">
        <!-- Here we add the date on which the dashboard was last
        updated to the card by obtaining the global variable "currentDate"
        which is defined in the gpsdashboard.component.ts file. -->
        <i class="material-icons">date_range</i> Last update : {{currentDate}}
      </div>
    </div>
  </div>
</div>
```

Now when we save the HTML file and reload the application we will be greeted by 4 cards displaying the data from our datastores.

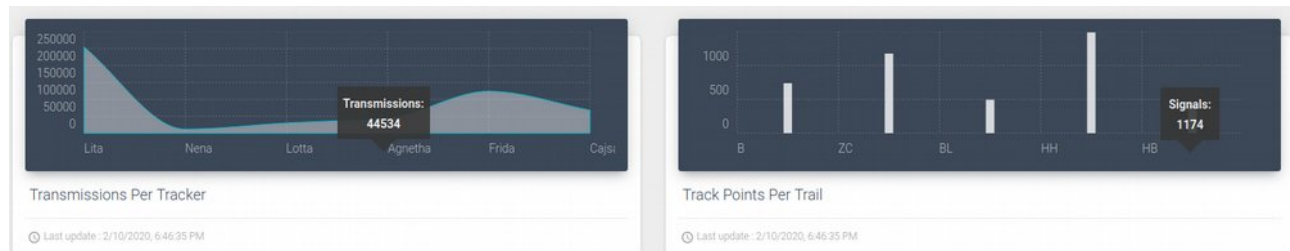
NOTE: the Flask API and the MongoDB datastore have to be running for the application to show the data!

3.3 Displaying interactive graphs in the dashboard

At this point we have a working dashboard that contains the data from our MongoDB datastore and it displays a row of cards containing the amount of items in the database.

Now we want to create some interactive graphs that display the amount of transmissions per tracker and the amount of signals per trail.

The end result will look the same as shown in the illustration below.



First we need to create a generic function which creates charts using the Chartist module we installed. We are going to create this function in the `gpsdashboard.component.ts` file.

We are going to call the function: `createChart()`. The creation of this function is going to be explained using the inline comments in the Source Code, which can be found in the folder POC located in the same folder as this programming manual.

```
/*
Here we create the generic function: "createChart()".

This function is used to create a chart using the Chartist library.
In the function we pass the following parameters:
- type = The type of chart that need to be created. This can either be a
        bar chart or a line chart.
- data = The list of data that is going to be displayed in the chart.
- labels = The name of the column that represents the name of the items.
           in the case of the trackers this column is called: "name"
- points = The name of the column that represents the amount of datapoints.
           in the case of the trials this column is called:
           "transmission_Count"
- HTMLElement = The id of the HTMLElement in which the chart is going to be
                added. This HTMLElement is defined in the html page of this
                component.
- description = The description which is going to be displayed in the tooltip
                in the case of the trackers this is going to be Transmissions.
*/
createChart(type,data,labels,points,HTMLElement,description):void{
```


The code shown in the following illustration is located inside the createChart() function.

```
/*
To be able to display tooltips we need to instantiate an instance of
the tooltip. This is done using the code below.
*/
let tool = tooltip

/*
Below we create an empty list of labels and an empty list of series.
The names of e.g. the trackers are going to be appended to the label list.
The amount of .e.g transmissions per tracker is going to be appended to
the series list.
*/
let _labels= []
let _series= []

/*
Below we create a forEach loop that loops through all the datarows in the
datalist passed as parameter in this function. For each row we append the
name of e.g. the tracker to the labels list and the amount of .e.g.
transmissions per tracker to the series list. We pass the parameters passed
in the function: "createChart()", as names of the columns that contain the
required data.

In the case of the Crane trackers the value in the forEach loop will be as
follows:

trackerlist.forEach(tracker => {
  _labels.push(element['name'])
  _series.push(element['transmission_Count'])
});

Then we will end up with a graph that contains 6 Crane names and their
corresponding amount of transmissions.
*/
data.forEach(element => {
  _labels.push(element[labels])
  _series.push(element[points])
});
```


Now we need to assign the labels and points to a variable called chartData. The code for this is shown in the illustration below.

```
/*
Here we Assign the labels and series to a variable called chartData.
This variable will be used later on when the chart is created.
As you can see we pass the description parameter passed in the function:
"createChart()" as metadata for the series of the chart.
*/
var chartData = {
  labels: _labels,
  series: [{meta: description, value: _series }]
};
```

Now we need to create the properties the chart is going to have. The code for this is shown in the illustration below.

```
/*
Here we the settings / properties of the chart and assign them to a
variable called chartSettings.
This variable will be used later on when the chart is created.

As you can see we pass the description parameter passed in the function:
"createChart()" as metadata for the series of the chart.

For more information on what settings there are, you can visit the website:
https://gionkunz.github.io/chartist-js/api-documentation.html
*/
let chartSettings = {
  /*
  The high and low values represent the max and min value of the y-axis
  in the chart. We use the build-in Javascript function:"Math.max/min"
  to obtain the lowest and highest values in the series (transmissions or
  signals)
  */
  high: Math.max(..._series),
  low: Math.min(..._series),
  /*
  Here we define what the padding of the charts need to be.
  */
  chartPadding: {top: 0, right: 30, bottom: 0, left: 15},
  showArea: true,
  showGrid: false,
  showLine: false,
  fullWidth: true,
  /*
  Here we assign what plugins are going to be loaded in the charts.
  At this point we only need the tooltip plugin. You can visit the
  chartist documentation for more available plugins by clicking on the URL:
  https://gionkunz.github.io/chartist-js/api-documentation.html
  */
  plugins: [Chartist.plugins.tooltip()]
}
```

Now we have to make sure a line or bar chart is created when calling the function: "createChart()". The code for this is shown in the illustration below.

```
/*
Here we create an empty variable called chart. Depending on what the value
of the parameter: "type", a chart of that type will be created
and assigned to the variable: "Chart".
*/
let chart;

/*
Below we create an IF/ELSE statement. If the value of the type, passed as
parameter when this function is called, is equal to "bar", a bar chart
is created. If the value of type is NOT equal to bar, a line chart is
created.

As parameters the HTMLElement in which the chart will be created, the
chartData and the chartSettings are passed.
*/
type == "bar" ? chart = new Chartist.Bar(HTMLElement,chartData,chartSettings)
              : chart = new Chartist.Line(HTMLElement,chartData,chartSettings)
};
```

Finally we have to make sure the Charts are created and filled with data when the dataset dashboard is loaded. Since we are going to create a Line Chart for the trackers in the database, we are going to add some code to the function: "getTrackers()".

The line we are going to add is show in the illustration below.

```
this.createChart('line',trackers,'name','transmission_Count','#craneChart','Transmissions:'))
```

The following applies to the code shown above:

- ➔ The type of chart that will be created is a linechart.
- ➔ The list of data that is going to be passed is the list of trackers obtained from the datastore.
- ➔ The name of the column that represent the names of the trackers is called: "name".
- ➔ The name of the column that represent the amount of transmissions belonging to a tracker is called: "transmission_Count".
- ➔ The HTMLElement in which the chart will be created has the id: "#craneChart".
- ➔ The description that will be shown in the tooltip is: "Transmissions:"

The final code of the `getTrackers()` function will be the same as shown in the illustration below:

```
getTrackers(): void {
  this._GPSDashboardService.getTrackers().subscribe(trackers => {
    trackers.forEach(tracker => {
      this.craneMap.set(tracker['name'], tracker)
    }),
    this.createChart('line', trackers, 'name', 'transmission_Count', '#craneChart', 'Transmissions:')
  });
};
```

Now we need to do the same for the trails and the amount of signals belonging to a trail. To do this we need to add the following line of code to the function: `getTrails()`.

```
this.createChart('bar', trails, 'abr', 't_points', '#trailChart', 'Signals:')
```

The following applies to the code shown above:

- ➔ The type of chart that will be created is a barchart.
- ➔ The list of data that is going to be passed is the list of trails obtained from the datastore.
- ➔ The name of the column that represent the names of the trails is called: "abr".
- ➔ The name of the column that represent the amount of signals belonging to a trail is called: "t_points".
- ➔ The HTMLElement in which the chart will be created has the id: "#trailChart".
- ➔ The description that will be shown in the tooltip is: "Signals:"

The final code of the `getTrails()` function will be the same as shown in the illustration below:

```
getTrails(): void {
  this._GPSDashboardService.getTrails().subscribe(trails => {
    trails.forEach(trail => {
      this.createChart('bar', trails, 'abr', 't_points', '#trailChart', 'Signals:')
    });
  });
};
```

Now when we save the code and go to our web application we will not see any charts yet. For this to change we need to add the HTML code for the charts. So we need to add both charts to the layout of our dashboard which is specified in the file `gpsdashboard.component.html`.

First we want to add a new div element with the classname: "row". In this div element we are going to add the 2 charts. We do this because we want the charts to be displayed next to each other (in a row).

The HTML code for the tracker Line Chart is shown in the illustration below.

```
<!-- This section is related to row of graphs in the dashboard-->
<div class="row">

  <!-- This section is related to tracker chart in the dashboard-->
  <div class="col-md-6">
    <div class="card card-chart">
      <!-- Below we create the header of the card.
      This is the top section where thhe chart is displayed.
      We give the div element the id:"craneChart" so we can obtain this
      element in our gpsdashboard.component.ts file by using the value
      of the id.-->
      <div class="card-header card-header-primary">
        <div class="ct-chart" id="craneChart"></div>
      </div>
      <!-- Below we create the body section of the card.
      We give a suitable title to this section-->
      <div class="card-body">
        <h4 class="card-title">Transmissions Per Tracker</h4>
      </div>
      <div class="card-footer">
        <!-- Here we add the date on which the dashboard was last
        updated to the card by obtaining the global variable "currentDate"
        which is defined in the gpsdashboard.component.ts file. -->
        <div class="stats">
          <i class="material-icons">access_time</i>Last update : {{currentDate}}
        </div>
      </div>
    </div>
  </div>
</div>
```

The HTML code for the trail barchart is shown in the illustration below.

```
<!-- This section is related to the trail chart in the dashboard-->
<div class="col-md-6">
  <div class="card card-chart">
    <!-- Below we create the header of the card.
    This is the top section where the chart is displayed.
    We give the div element the id:"trailChart" so we can obtain this
    element in our gpsdashboard.component.ts file by using the value
    of the id.-->
    <div class="card-header card-header-primary">
      <div class="ct-chart" id="trailChart"></div>
    </div>
    <!-- Below we create the body section of the card.
    We give a suitable title to this section-->
    <div class="card-body">
      <h4 class="card-title">Track Points Per Trail</h4>
    </div>
    <div class="card-footer">
      <!-- Here we add the date on which the dashboard was last
      updated to the card by obtaining the global variable "currentDate"
      which is defined in the gpsdashboard.component.ts file. -->
      <div class="stats">
        <i class="material-icons">access_time</i> Last update : {{currentDate}}
      </div>
    </div>
  </div>
</div>
```

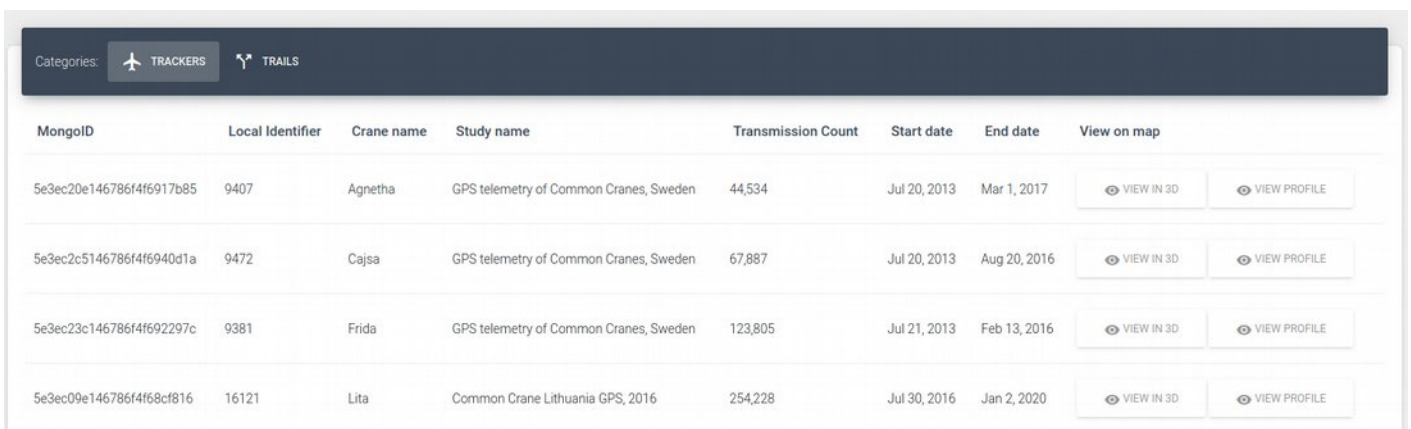

Now when we save the HTML file and reload the application we will be greeted by 2 charts displaying the data from our datastores.

NOTE: The Flask-API and the MongoDB datastore have to be running for the application to show the data.

3.4 Creating an interactive table

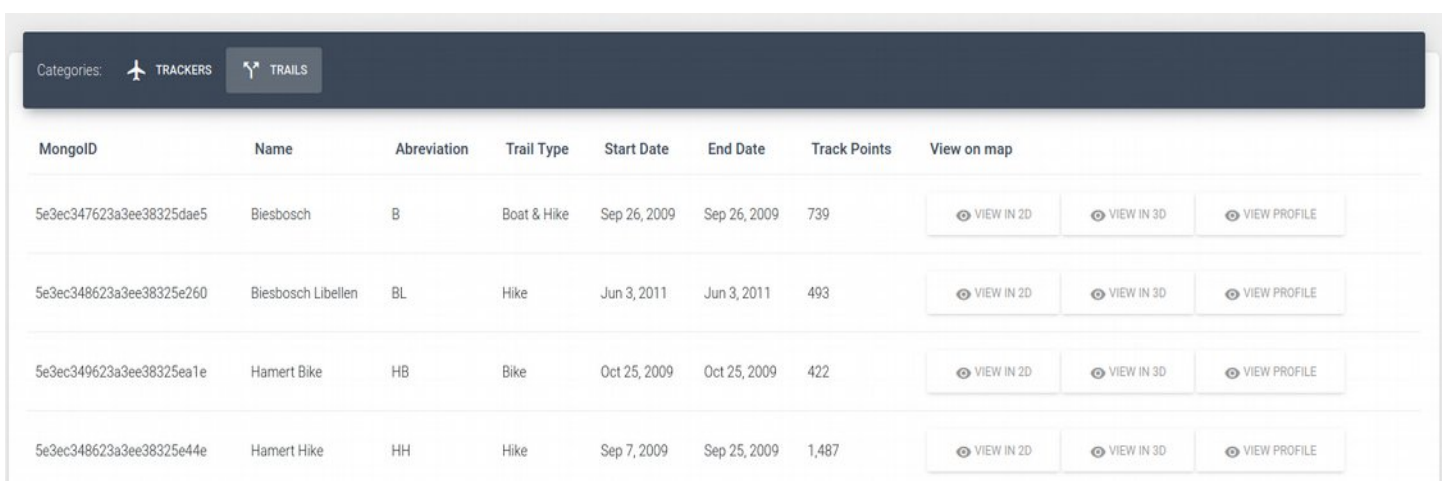
Now we want to create a table which shows detailed information related to the items and trackers. This table is going to be interactive and contains 2 tabs. One tab for the trackers in the datastore and one tab for the trails in the datastore.

The final result when the tracker tab is selected will look the same as shown in the illustration below.



Categories: ✈ TRACKERS 📍 TRAILS							
MongoID	Local Identifier	Crane name	Study name	Transmission Count	Start date	End date	View on map
5e3ec20e146786f4f6917b85	9407	Agnetha	GPS telemetry of Common Cranes, Sweden	44,534	Jul 20, 2013	Mar 1, 2017	VIEW IN 3D VIEW PROFILE
5e3ec2c5146786f4f6940d1a	9472	Cajsa	GPS telemetry of Common Cranes, Sweden	67,887	Jul 20, 2013	Aug 20, 2016	VIEW IN 3D VIEW PROFILE
5e3ec23c146786f4f692297c	9381	Frida	GPS telemetry of Common Cranes, Sweden	123,805	Jul 21, 2013	Feb 13, 2016	VIEW IN 3D VIEW PROFILE
5e3ec09e146786f4f68cf816	16121	Lita	Common Crane Lithuania GPS, 2016	254,228	Jul 30, 2016	Jan 2, 2020	VIEW IN 3D VIEW PROFILE

The final result when the trail tab is selected will look the same as shown in the illustration below.



Categories: ✈ TRACKERS 📍 TRAILS							
MongoID	Name	Abbreviation	Trail Type	Start Date	End Date	Track Points	View on map
5e3ec347623a3ee38325dae5	Biesbosch	B	Boat & Hike	Sep 26, 2009	Sep 26, 2009	739	VIEW IN 2D VIEW IN 3D VIEW PROFILE
5e3ec348623a3ee38325e260	Biesbosch Libellen	BL	Hike	Jun 3, 2011	Jun 3, 2011	493	VIEW IN 2D VIEW IN 3D VIEW PROFILE
5e3ec349623a3ee38325ea1e	Hamert Bike	HB	Bike	Oct 25, 2009	Oct 25, 2009	422	VIEW IN 2D VIEW IN 3D VIEW PROFILE
5e3ec348623a3ee38325e44e	Hamert Hike	HH	Hike	Sep 7, 2009	Sep 25, 2009	1,487	VIEW IN 2D VIEW IN 3D VIEW PROFILE

First we want to create the header of the table (card) so we can switch between tabs. To create the header we need to add the following code to our `gpsdashboard.component.html` file.

```
<!-- This section is related to table in the dashboard -->
<div class="row">
  <div class="col-lg-12 col-md-12">
    <div class="card">
      <!-- This section is related to the table header-->
      <div class="card-header card-header-tabs card-header-primary">
        <div class="nav-tabs-navigation">
          <div class="nav-tabs-wrapper">
            <span class="nav-tabs-title">Categories:</span>
            <ul class="nav nav-tabs">
              <!-- This section is related to selecting the tracker tab-->
              <li class="nav-item">
                <a mat-button class="nav-link active"
                  [ngClass]="{ 'active':activeTab==='tracker'}"
                  (click)="setActiveTab('tracker')" data-toggle="tab">
                  <i class="material-icons">flight</i> Trackers=
                </a>
              </li>
              <!-- This section is related to selecting the trail tab-->
              <li class="nav-item">
                <a mat-button class="nav-link"
                  [ngClass]="{ 'active':activeTab==='trail'}"
                  (click)="setActiveTab('trail')", data-toggle="tab" >
                  <i class="material-icons">call_split</i> Trails
                  <div class="ripple-container"></div>
                </a>
              </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Now when we reload the application we are greeted with the header of the table that contains 2 buttons. The problem is that we cannot click on these buttons yet. For this to be possible we need to add some code to our `gpsdashboard.component.ts` file.

We need to create a global variable called `activeTab`. This global variable will have a default value of 'tracker', which means that the tracker tab is active when the dashboard is loaded. The code for defining this variable is shown in the illustration below.

```
/*
Here we define a global variable called: "activeTab". The default value of the
variable is 'tracker', this means that when the dashboard is loaded the
tracker tab will be active in the interactive table.

The value of this global variable can be changed using the function:
"setActiveTab()". This function is defined later on in this file.
*/
private activeTab:string = 'tracker';
```

Now we need to create a function which sets the value of the global variable: "activeTab" to the value passed in this function. The function we are going to create is called: "setActiveTab()". In this function we pass the name of the tab that needs to be activated. The code required to create this function is shown in the illustration below.

```
/*
Here we create the function: "setActiveTab()".

This function is used to switch between the Categories in our interactive
table. This function will be triggered when one of the buttons in the header
of this table is clicked. Depending on which category/button you click, the
value of this category/button is passed in the function, which then changes
the global variable: "activeTab" to the value passed in the function.
*/
setActiveTab(tab):void{
    this.activeTab = tab;
};
```

Now when we save the file and reload the application we can switch between the categories/tabs in the interactive table.

At this point we only have the header of our table. Now we want to fill the body of the table with the data obtained from our MongoDB datastore.

So let's create the body of the table which will contain all the data related to the selected category/tab. For this we need to add some code to the `gpsdashboard.component.html` file.

First we want to add the column names of the tables by adding the following HTML code:

```
<!-- This section is related to the table body-->
<div class="card-body">
  <div class="tab-content">
    <!-- This section is related to the tracker table -->
    <div class="tab-pane active" id="tracker"
      [ngClass]="{'active':activeTab==='tracker'}">
      <div class="table-responsive table-hover">
        <table class="table table-hover">
          <thead class="text-primary">
            <th>
              MongoID
            </th>
            <th>
              Local Identifier
            </th>
            <th>
              Crane name
            </th>
            <th>
              Study name
            </th>
            <th>
              Transmission Count
            </th>
            <th>
              Start date
            </th>
            <th>
              End date
            </th>
            <th>
              Generate data profile
            </th>
          </thead>
        </table>
      </div>
    </div>
  </div>
</div>
```


Now we want to add the data to these columns. For this we need to add the following code beneath the HTML tag `</thead>`. Note the `</thead>` tag is also included in this illustration, you do not add this tag again.

```
</thead>
<tbody>
  <tr *ngFor="let tracker of craneMap | keyvalue">
    <td>
      {{tracker.value._id.$oid}}
    </td>
    <td>
      {{tracker.value.individual_local_identifier}}
    </td>
    <td>
      {{tracker.value.name}}
    </td>
    <td>
      {{tracker.value.study_name}}
    </td>
    <td>
      {{tracker.value.transmission_Count | number: '2.'}}
    </td>
    <td>
      {{tracker.value.start_date.$date | date}}
    </td>
    <td>
      {{tracker.value.end_date.$date | date}}
    </td>
    <td>
      <button mat-raised-button class="btn btn-white "
        (click)="generateProfile('api/generate-trackers-profile')">
        <i class="material-icons">remove_red_eye</i> View Profile</button>
      </td>
  </tr>
</tbody>
```

Now we have a table filled with data related to the trackers in the database.

Now we want to do the same for the trails.

Again, we first want to add the column names of the table for the trails. We do this by adding the following code to the HTML file.

We add the code below the closing div element representing the tab for the trackers. The code for this is shown in the illustration below.

```
<!-- This section is related to the trail table -->
<div class="tab-pane" id="trails"
  [ngClass]="{ 'active':activeTab==='trail' }">
  <div class="table-responsive table-hover">
    <table class="table table-hover">
      <thead class="text-primary">
        <th>
          MongoID
        </th>
        <th>
          Name
        </th>
        <th>
          Abreviation
        </th>
        <th>
          Trail Type
        </th>
        <th>
          Start Date
        </th>
        <th>
          End Date
        </th>
        <th>
          Track Points
        </th>
        <th>
          View on map
        </th>
      </thead>
    </table>
  </div>
</div>
```

Again, we want to add the trail data to these columns. To be able to do this we add the following code below the `</thead>` tag in our HTML file.

NOTE: just like before, the `</thead>` tag is included in the illustration, so don't add this tag again.

```
</thead>
<tbody>
  <tr *ngFor="let trail of trailMap | keyvalue">
    <td>
      {{trail.value._id.$oid}}
    </td>
    <td>
      {{trail.value.name}}
    </td>
    <td>
      {{trail.value.abr}}
    </td>
    <td>
      {{trail.value.r_type}}
    </td>
    <td>
      {{trail.value.s_date.$date | date}}
    </td>
    <td>
      {{trail.value.e_date.$date | date}}
    </td>
    <td>
      {{trail.value.t_points | number: '2.'}}
    </td>
    <td>
      <button mat-raised-button class="btn btn-white "
        (click)="generateProfile('api/generate-trails-profile')">
        <i class="material-icons">remove_red_eye</i> View Profile</button>
      </td>
    </tr>
  </tbody>
```

3.5 Generating pandas data profile

The last functionality we are going to add is the ability to generate data profile using Pandas Profiling. This function will be triggered when clicking on the button: “generate profile”, in the interactive table.

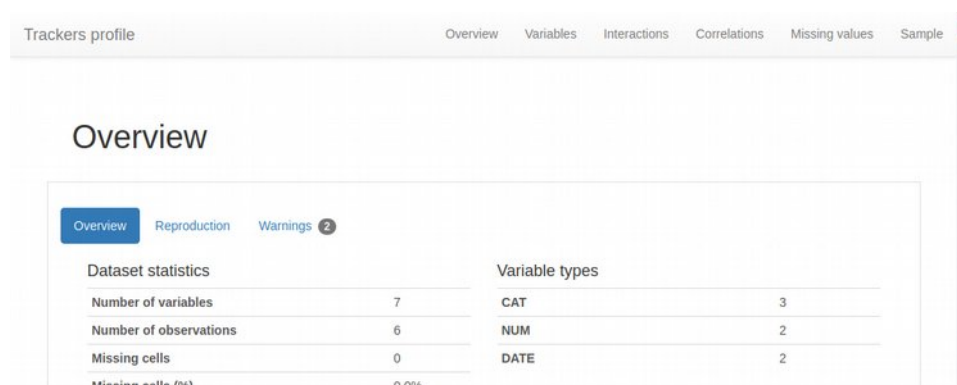
The function then performs an API request to the URL bound to the function in our Flask-API which creates a data profile. The code for creating this function is shown in the illustration below.

```
/*
Here we create the function: "generateProfile()".

This function is used to send perform an API call on the URL passed as
parameter in this function. Since Angular uses a router, we need to open
a new blank window in which the profile will be generated. After the API
is finished generating this profile the profile will be displayed in the
window opened by this function.

This function is assigned and triggered using the buttons in the interactive
table which are defined in the layout page of this component:
"gpsdashboard.component.html"
*/
generateProfile(url):void{
  window.open(url,'_blank');
};
```

That's it! When you click on the button generate profile in the interactive table, the Flask-API will generate a data profile belonging to the tracker from which you clicked the button as shown in the illustration below.



That's it! You now have created your own Dataset Dashboard web application in Angular!

In the next programming manual you will learn how to create a 2D Map Viewer web application to visualize the data displayed in the dataset dashboard on 2D topographical base maps from OpenStreetMap using the geospatial framework: “OpenLayers”.