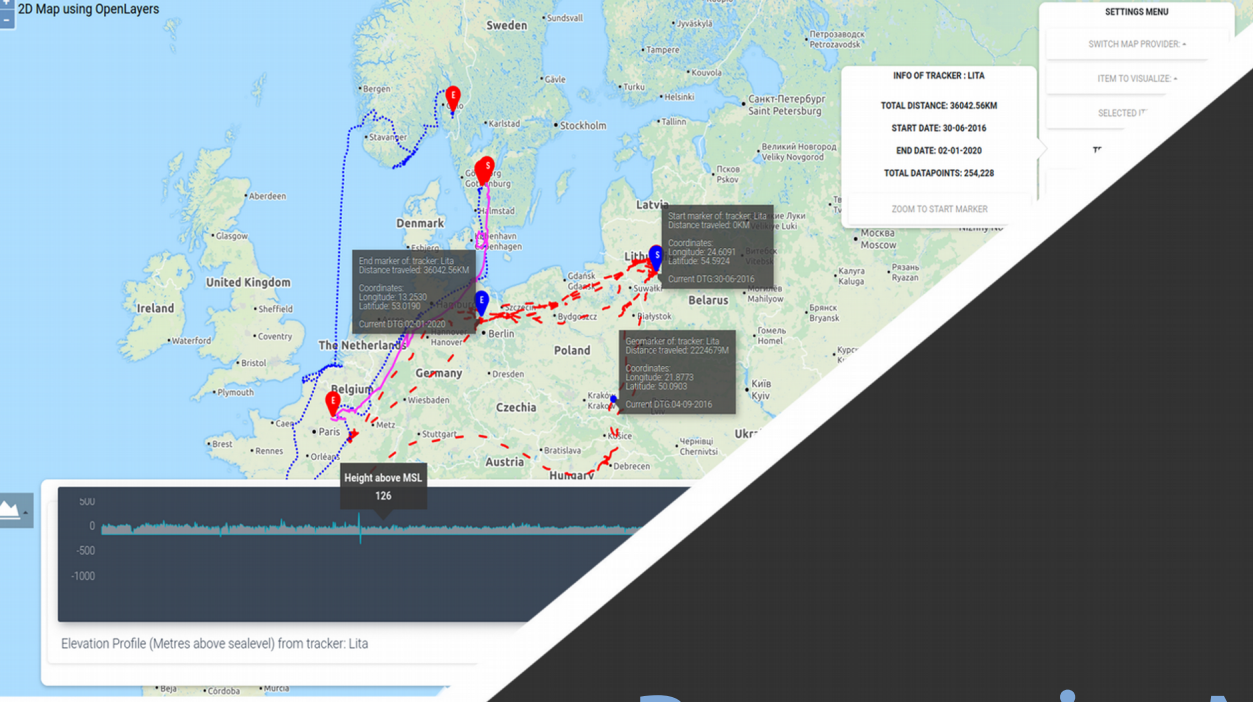
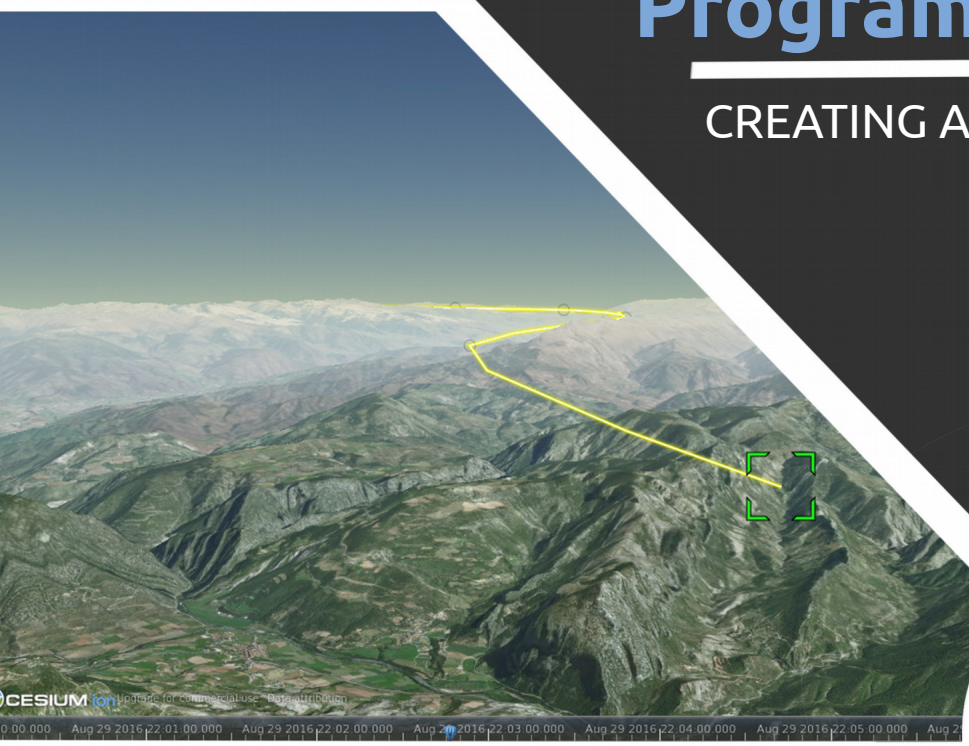
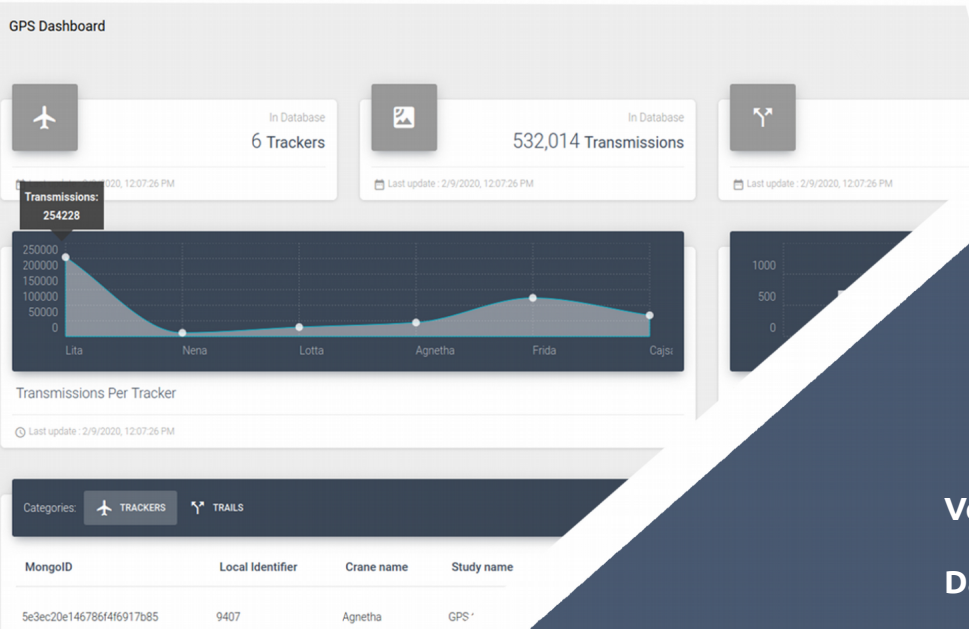


Programming Manual

CREATING A BASIC WEBAPPLICATION USING ANGULARJS





Version : 1.0

Date : 08-04-2020

Author : The GeoStack Project

Purpose of this document

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

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

   The tools and libraries used during this programming manual are installed in the cookbook: Creating the Geostack Course VM.

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

The purpose of this programming manual is to create an application using the Angular JavaScript framework in combination with TypeScript.

This application will serve as base application for our Dataset dashboard, 2D Map Viewer and 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.

For more in depth information related to AngularJS you can visit the following URL:

https://angular.io/docs

Most of the code shown in the programming manual is explained using inline comments in the source code of this application. 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.

A special thanks to the following people and organizations:

- **The people and contributes 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.Introduction to AngularJS

AngularJS is a JavaScript-based open-source front-end web framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications.

Angular aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

- ➔ Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
- ➔ Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

- ➔ The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
- ➔ The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

1.1 Angular Modules

Angular NgModules differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the Router NgModule.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

1.2 Angular Components

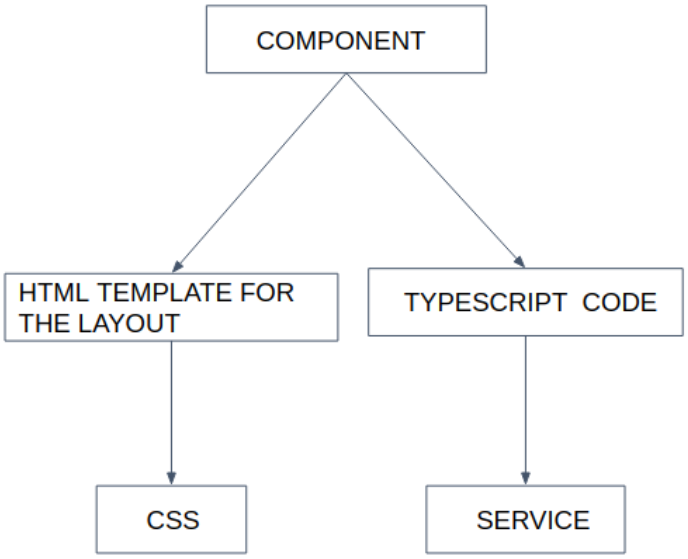
Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

The @Component() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

A component can be seen as a building block to create page. A page can consist of multiple component and component consist of the following 4 parts:

1) A (Optional) service: This is a TypeScript file which contains all the function related to retrieving data by making calls to our Flask-API which then returns the data to the Angular application.
2) A TypeScript component file: This file contains the business logic related to how a component should work.
3) An HTML file which contains the layout of the component.
4) An (Optional) CSS / SCSS file which contains the styling of the component.

The image below shows the structure of a component.



1.3 Angular File structure

In the following chapter a few images are used to describe the file structure, required for our applications, of the Angular application. The full structure can be found at the following link: https://angular.io/guide/file-structure. The file-structure can be divided in the following categories:

1) Workspace configuration files
2) Application project files
   I. Application source files
   II. Application configuration files

For each of the categories an image will be added with the relevant files and their description.

`.editorconfig`	Configuration for code editors. See EditorConfig.
`.gitignore`	Specifies intentionally untracked files that Git should ignore.
`README.md`	Introductory documentation for the root app.
`angular.json`	CLI configuration defaults for all projects in the workspace, including configuration options for build, serve, and test tools that the CLI uses, such as TSLint, Karma, and Protractor. For details, see Angular Workspace Configuration.
`package.json`	Configures npm package dependencies that are available to all projects in the workspace. See npm documentation for the specific format and contents of this file.
`package-lock.json`	Provides version information for all packages installed into `node_modules` by the npm client. See npm documentation for details. If you use the yarn client, this file will be yarn.lock instead.
`src/`	Source files for the root-level application project.
`node_modules/`	Provides npm packages to the entire workspace. Workspace-wide `node_modules` dependencies are visible to all projects.
`tsconfig.json`	Default TypeScript configuration for projects in the workspace.
`tslint.json`	Default TSLint configuration for projects in the workspace.

1.3.1 Workspace configuration files

All projects within a workspace share a CLI configuration context. The top level of the workspace contains workspace-wide configuration files, configuration files for the root-level application, and sub folders for the root-level application source and test files.

1.3.2 Application project files

By default, the CLI command `ng new my-app` creates a workspace folder named "my-app" and generates a new application skeleton in a `src/` folder at the top level of the workspace. A newly generated application contains source files for a root module, with a root component and template.

Files at the top level of `src/` support testing and running your application. The Sub folders contain the application source and application-specific configuration.

`app/`	Contains the component files in which your application logic and data are defined. See details below.
`assets/`	Contains image and other asset files to be copied as-is when you build your application.
`environments/`	Contains build configuration options for particular target environments. By default there is an unnamed standard development environment and a production ("prod") environment. You can define additional target environment configurations.
`favicon.ico`	An icon to use for this application in the bookmark bar.
`index.html`	The main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you typically don't need to add any `<script>` or`<link>` tags here manually.
`main.ts`	The main entry point for your application. Compiles the application with the JIT compiler and bootstraps the application's root module (AppModule) to run in the browser. You can also use the AOT compiler without changing any code by appending the `--aot` flag to the CLI `build` and `serve` commands.
`polyfills.ts`	Provides polyfill scripts for browser support.
`styles.sass`	Lists CSS files that supply styles for a project. The extension reflects the style preprocessor you have configured for the project.

Inside the `src/` folder, the `app/` folder contains your project's logic and data. Angular components, templates, and styles go here.

`app/app.component.ts`	Defines the logic for the app's root component, named `AppComponent`. The view associated with this root component becomes the root of the view hierarchy as you add components and services to your application.
`app/app.component.html`	Defines the HTML template associated with the root `AppComponent`.
`app/app.component.css`	Defines the base CSS stylesheet for the root `AppComponent`.
`app/app.component.spec.ts`	Defines a unit test for the root `AppComponent`.
`app/app.module.ts`	Defines the root module, named `AppModule`, that tells Angular how to assemble the application. Initially declares only the `AppComponent`. As you add more components to the app, they must be declared here.

2. Creating an app and cleaning up

We are going to start off by creating a new angular application, in the angular-apps folder located in the Geostack folder, by using the command: “ng new base-application --minimal”. In this command the “base-application”, is the name the application will get and the “--minimal” makes sure that the new application only creates the files we need.. So let’s run this command.

When running the command you get the following options:

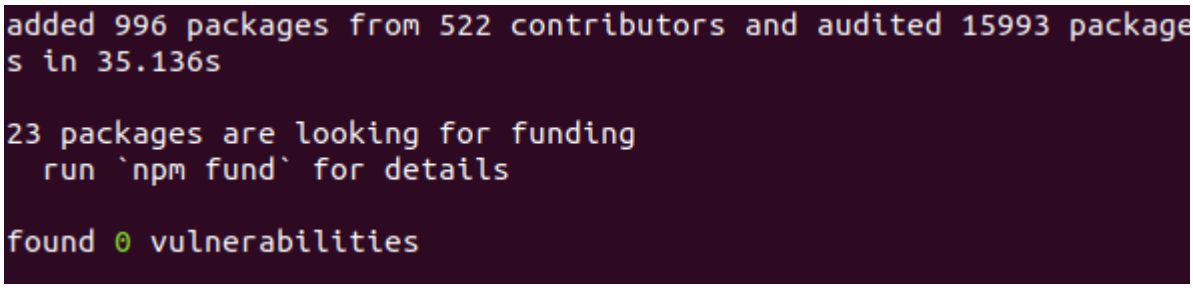
- ➔ **Add Angular routing? : Answer yes**

  Angular routing enables us to switch between views (Pages) while not having to refresh the page.

- ➔ **Which style sheet format would you like to use?: Answer SCSS**

  We are going to use SCSS to style our applications. SCSS, also known as SASS, is a CSS pre-processor with syntax advancements.

If everything is installed correctly, the output of the terminal will look similar to the one shown in illustration.



First we want to cleanup the files that we don't need. The files and the commands to remove them are as follows:

- ➔ The file README.md: rm ~/Geostack/angular-apps/base-application/README.md
- ➔ The file .gitignore: rm ~/Geostack/anSgular-apps/base-application/.gitignore
- ➔ The file .gitkeep:
  rm ~/Geostack/angular-apps/base-application/src/assets/.gitkeep
- ➔ The file browserslist: rm ~/Geostack/angular-apps/base-application/browserslist
- ➔ The image favicon: rm ~/Geostack/angular-apps/base-application/src/favicon.ico

During the programming manuals the focus is mainly on creating the business logic for our GeoStack needs. The component styling is mostly obtained from Creative Tim which is a website which provides Open Source Angular component styling. You can visit his website by clicking on the following URL: https://www.creative-tim.com/. A special thanks to him for creating the base styling of our application. Now lets copy our styling to our application.

The styling is located in the folder: scss which is found in the same folder as this programming manual. To copy the styling to our application we have to run the following command in a terminal:
cp -r {path_to_scss_folder} ~/Geostack/angular-apps/base-application/src/assets/

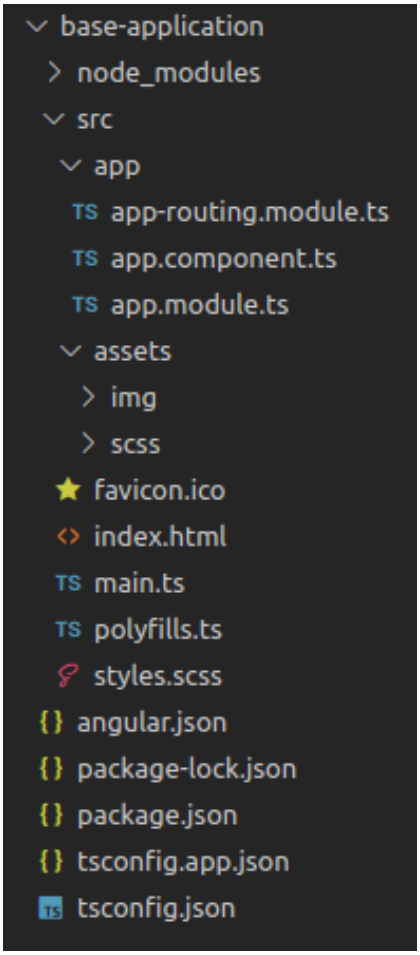
Then we want to copy the image folder to our application . This folder contains all the map markers (start and end markers), our arrow SVG and our GeoStack logo. We copy this folder by running the command:
cp -r {path_to_img_folder} ~/Geostack/angular-apps/base-application/src/assets/

Finally we want to copy the favicon to our application. This is the icon that is displayed in the web browser tab. The command for this is:

cp {path_to_favicon.ico} ~/Geostack/angular-apps/base-application/src/

Now that we have removed all the unnecessary files we should end up with a file structure similar to the one shown in illustration. The following applies to this illustration:

- node_modules: This folder contains all the modules required to build our Angular application.
- src: This folder contains the application source files .
- App: This folder contains the component files in which our application logic is defined.
- app-routing.module.ts: This contains all the routes of our Angular applications. The routes define on what URL a page / component should be available and loaded.
- app.component.ts: This file defines the logic for the app's root component, named AppComponent. The view associated with this root component becomes the root of the view hierarchy as you add components and services to your application.
- app.module.ts: This file defines the root module, named AppModule, that tells Angular how to assemble the application. Initially declares only the AppComponent. As you add more components to the app, they must be declared here.
- assets (Red): This folder contains all the styling, images and later on the Geospatial frameworks of our 2D and 3D Map viewers.
- Img: This folder contains the images used throughout our applications.



- Scss: This folder contains the SCSS style sheets used in our applications.
- index.html: This is the main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you typically don't need to add any <script> or<link> tags here manually.
- angular.json: This file contains the CLI configuration defaults for all projects in the workspace, including configuration options for build, serve, and test tools that the CLI uses.
- package.json: This file contains and configures npm package dependencies that are available to all projects in the workspace.

Now you know a bit about the basic file structure, we are going to add some new folders. These folders are as follows:

- ➔ A services folder by running the command:

  mkdir ~/Geostack/angular-apps/base-application/src/app/services.

  In this folder we are going to add all services of our applications. The service files will contain all the functions related to sending data requests to our Flask-API. During this programming manual we will not add any services yet.

- ➔ A pages folder by running the command:

  mkdir ~/Geostack/angular-apps/base-application/src/app/pages.

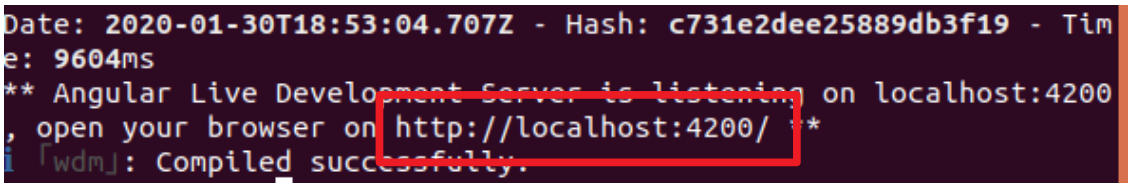
  This folder is going to contain all our pages from our web application. During this programming manual we are only going to create one page.

- ➔ A components folder by running the command:

  mkdir ~/Geostack/angular-apps/base-application/src/app/components.

  In this folder we are going to add all our component which will be reused throughout our application. These components include a navbar and a sidebar. Now that we have cleaned up the application, we want to check if it still works. This is done by running the command: “sudo npm start”. You should run this command while in the base-application folder. The command compiles all our code and then runs the Angular live development server. The Angular love development server will make sure you don’t have to restart the application when editing your application source code. After adding or updating code you can simply save the code after which the changes will automatically be compiled.

The output of the command: “sudo npm start”, should be similar to the one in illustration.



When pressing control+left-mouse click on the link displayed in the terminal (Red), you will be greeted by the default homepage of a new Angular application. This page should be similar to the one shown in the illustration below.

Let’s close the application by stopping the process in the terminal by pressing the keys Ctrl + c on your keyboard.

2.1 Installing the required dependencies

Now we want to install some dependencies which we are going to use in our angular applications. We are going to add these packages to the package.json file located in the root folder of our base-application. The dependencies will be added to the section shown in illustration.

```
"dependencies": {
  "@angular/animations": "~8.2.14",
  "@angular/common": "~8.2.14",
  "@angular/compiler": "~8.2.14",
  "@angular/core": "~8.2.14",
  "@angular/forms": "~8.2.14",
  "@angular/platform-browser": "~8.2.14",
  "@angular/platform-browser-dynamic": "~8.2.14",
  "@angular/router": "~8.2.14",
  "rxjs": "~6.4.0",
  "tslib": "^1.10.0",
  "zone.js": "~0.9.1"
},
```

The packages are as follows:

- ➔ Ng-Bootstrap by adding the entry: "@ng-bootstrap/ng-bootstrap": "^5.2.1",
- ➔ Bootstrap by adding the entry: "bootstrap": "^4.4.1",
- ➔ Chartist by adding the entry: "chartist": "^0.11.4",
- ➔ Chartist-plugin-tooltips by adding the entry: "chartist-plugin-tooltips": "0.0.17",
- ➔ Font awsome by adding the entry: "font-awesome": "^4.7.0",
- ➔ Material design icons by adding the entry: "material-design-icons": "^3.0.1"

After adding the entries, the bottom of dependencies section should look the same as the one shown in illustration.

```
"rxjs": "~6.4.0",
"tslib": "^1.10.0",
"zone.js": "~0.9.1",
"@ng-bootstrap/ng-bootstrap": "^5.2.1",
"bootstrap": "^4.4.1",
"chartist": "^0.11.4",
"chartist-plugin-tooltips": "0.0.17",
"font-awesome": "^4.7.0",
"material-design-icons": "^3.0.1"
```

To install dependencies which were added to the package.json filewe need to run the following command from the root directory of our base-application: `sudo npm install`

Finally we want to install the angular-CDK dependency. This is done by running the following command from the root directory of our base-application:

`sudo npm install @angular/cdk`

NOTE: A breaking update has been added to the Chartist package. To be able to create charts properly we need to add the following lines inside the <head> element from the index.html page located in the folder base-application/src/app/.

```
<script>
  window.global = window;
</script>
```

So the full content of the <head> tag will be the same as the one shown in the illustration below.

```
<head>
  <meta charset="utf-8">
  <title>BaseApplication</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <script>
    window.global = window;
  </script>
</head>
```

2.2 Adding the styling to the application

Now we want add and edit some styling used by our application. This is done in the file called: “angular.json”. So let’s open this file and edit the styles section on line 30 from:

```
"styles": [
  "src/styles.scss"
],
```

to:

```
"styles": [
  "src/assets/scss/styles.scss",
  "node_modules/material-design-icons/iconfont/material-icons.css",
  "node_modules/font-awesome/css/font-awesome.css"
],
```

The following applies to the image above:

- ➔ We changed the location of our default style.scss to the location of the style.scss located in the scss folder which we copied in chapter at the beginning of this chapter. (Red)
- ➔ We added the material-design-icons style sheet. (Blue)
- ➔ we added the font-awesome style sheet. (Green)

That’s it! Now we can use the styling and the icons throughout our application(s).

2.3 Editing the default files

First we need to create a new file called: app.component.html. This HTML file will contain the layout that will apply to all the pages of our application. We create this file by running the command:

```
touch ~/Geostack/angular-apps/base-application/src/app/app.component.html.
```

In this file we are going to add the code shown in the illustration below.

```
<div class="wrapper">
    <div class="main-panel">
        <div class="main-content">
            <router-outlet></router-outlet>
        </div>
    </div>
</div>
```

The following applies to the code shown above:

- ➔ We create an HTML div element and we give it the classname: “wrapper”
- ➔ We create an HTML div element and we give it the classname: “main-panel”
- ➔ We create an HTML div element and we give it the classname: “main-content”
- ➔ We create add the <router-outlet> in the div element created in the previous step. The router-outlet represents all the pages on in our webapplication. Each time we switch to another page, the router outlet makes sure that the previous page is replaced with the newly selected page. Since the <router-outlet> is located in the elements it means that all the pages and components will get the styling that we assigned to the wrapper and main-panel elements.

Now we are going to edit some default files which were created when creating the base-application. We are going to start off with the app.component.ts, located in the folder: base-application/src/app/.

We want to remove everything on the line starting with: “template:”. After removing the code we should make a reference to the HTML page we created above. How this is done is shown in illustration.

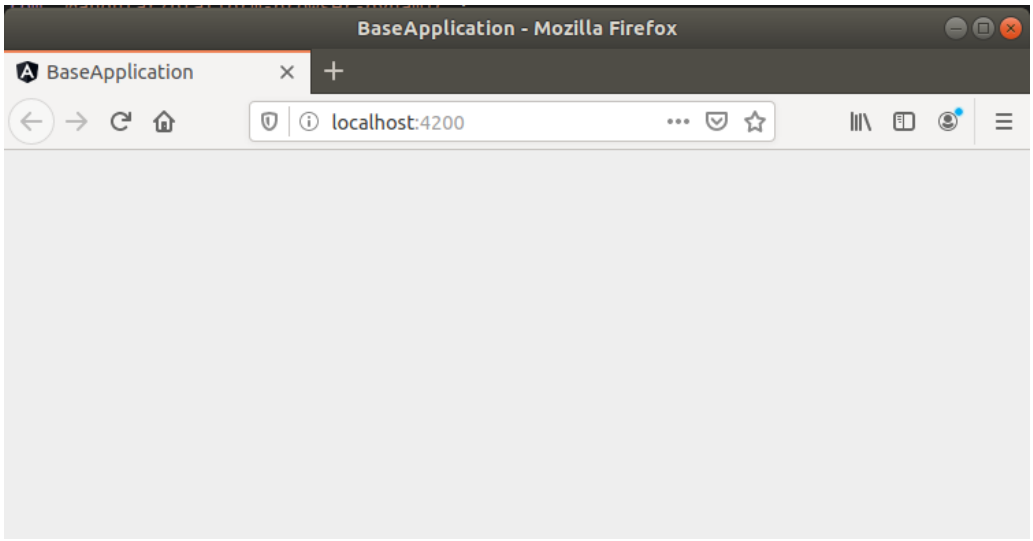
```
templateUrl: 'app.component.html',
```

So the final code should look the same as shown in the illustration below.

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styles: []
})
```

The code we removed was the layout of the default page that was shown when running the application for the first time. The code we added makes sure that the app.component uses the HTML page we created earlier as layout template.

Now when we run the application and navigate to the URL: "http://localhost:4200/", we are greeted with an empty screen as shown in illustration.



So now we want to add our first components. This will be done in the next chapter.

3. Creating the navbar and sidebar components

We are going to start of by creating the sidebar component. First we need a folder called sidebar. In this folder we will create all the files related to the sidebar component. Since the sidebar is going to be a reusable component we put it in our components folder.

Create the folder by running the command:

```
mkdir ~/Geostack/angular-apps/base-application/src/app/components/sidebar
```

Now we want to create a HTML-file which will contain the layout of our sidebar. We do this by running the command:

```
touch ~/Geostack/angular-apps/base-application/src/app/components/sidebar/
sidebar.component.html
```

In this file we are going to start off with adding our logo to the top of our sidebar. the following code:

```
<div class="logo">
    <a class="simple-text">
        <div class="logo-img">
            <img src="/assets/img/favicon.png"/>
        </div>
        Base application
    </a>
</div>
```

The following applies to the code shown above:

- ➔ We create an HTML div element and we give it the classname: “logo”
- ➔ We create an HTML a element and we give it the classname: “simple-text”
- ➔ We create an HTML div element an we give it the classname: “logo-img”
- ➔ In this div element we create an image element and use src=”/assets/img/favicon.png” to point to our logo image.
- ➔ Outside the div element, created in the previous step we add the text: Base application. This is the text that will be displayed in the sidebar.

```
<div class="sidebar-wrapper">
    <ul class="nav">
        <li routerLinkActive="active" *ngFor="let menuItem of menuItems"
         class="{{menuItem.class}} nav-item">
            <a class="nav-link" [routerLink]="[menuItem.path]">
                <i class="material-icons">{{menuItem.icon}}</i>
                <p>{{menuItem.title}}</p>
            </a>
        </li>
    </ul>
</div>
```

The following applies to the code shown above:

- ➔ We create an HTML div element and we give it the classname: “sidebar-wrapper”
- ➔ We create an HTML list (ul) element and we give it the classname: “nav”
- ➔ We create an HTML list item (li) element for each entry in the menuItems list and perform a check whether the menuItem is active or not.
- ➔ In this div element we create an image element and use src=”/assets/img/favicon.png” to point to our logo image.
- ➔ Outside the div element, created in the previous step we add the text: Base application. This is the text that will be displayed in the sidebar.

Then we need to create a component.ts file in which we are going to put all the logic related to the sidebar. We do this by running the command:

`touch ~/Geostack/angular-apps/base-application/src/app/components/sidebar/sidebar.component.ts`

In this file we first need to import some default Anuglar imports. So let’s do this by adding the following code to that file:

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

Now we need to create an interface which defines what attributes a route is going to have. When we add a new route to our sidebar, the route needs to have the attributes defined in the interface.
The following applies to this interface:

- ➔ Path: The path to which will be navigated when the sidebar entry is clicked.
- ➔ title: The title which the entry and the page wil have.
- ➔ icon: The entry icon which will show in the sidebar.
- ➔ class: If any classes need to be passed with this route, they can be passed in this value. A class can be used to set specific styling to a sidebar entry.

The code for this can be found in the illustration below.

```
declare interface RouteInfo {
    path: string;
    title: string;
    icon: string;
    class: string;
}
```

Now we need to add some entries to the sidebar. At this point we do not have any routes yet in our application. These will be added in chapter 4, but we are already going to add the entry to our sidebar. How this is done is shown on the next page.

First we need to create a list which contains all the routes that will be displayed in our sidebar. The list items inherit the interface: "RouteInfo".

Since we are only going to add the route of our base page, we only need to create one route. The following applies to this route:

- ➔ Path: The base page component is located on the path:"/base-page"
- ➔ title: The title of the base page is going to be: "Base Page"
- ➔ icon: The entry icon will be the map icon provided by the package "Material Icons". If you want to add other icons you can navigate to the following URL: https://material.io/resources/icons/?style=baseline
- ➔ class: No extra classes need to be passed in this route.

The code for defining the route for our first page is shown in the illustration below.

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

Next we want to define the component metadata. The following applies to the code below:

- ➔ The selector is going to be: "app-sidebar", so when we want to use this component we use the syntax <app-sidebar/> in an HTML page to which we want to add this component.
- ➔ The templateUrl is going to be: './sidebar.component.html', this HTML page contains the layout of the sidebar component.

```
@Component({
  selector: 'app-sidebar',
  templateUrl: './sidebar.component.html',
})
```

Now we define the SidebarComponent class. This class is going to be relatively simple. The only functionality the SidebarComponent has, is that it has to show the available routes and highlight a route/entry when it's active.

```
export class SidebarComponent implements OnInit {

  /*
  Here we create an empty list called menuItems. This list will be populated
  with all the routes defined at line 37.
  */
  menuItems: any[];

  /*
  Here we create a constructor which is going to be empty since we don't
  need to instantiate anything in this component.
  */
  constructor() { }

  /*
  Here we create the function that's triggered when the component is loaded.
  In this function we define the code which will populate the empty menuItems
  list with all the routes defined in line 37.
  */
  ngOnInit() {
    this.menuItems = ROUTES.filter(menuItem => menuItem);
  }
}
```

Now that we have finished the code for our sidebar, we need to add the component to our app.module.ts. So let's open the file and add the following line below the last import in the file.

```
import { SidebarComponent } from '../app/components/sidebar/sidebar.component';
```

This imports the SidebarComponent and will make sure we can use the component in the app.module.ts file. Now we need to add the SidebarComponent to the declarations section in the app.module.ts. The final declarations section should look the same as the one shown in illustration.

```
declarations: [
  AppComponent,
  SidebarComponent,
],
```

Now we can use the component throughout our application by using the selector which we defined in the sidebar.component.ts, which was <app-sidebar/>.

Since we want to display the sidebar on all the pages of our webapplication, we are going to add the sidebar selector to our app.component.html file. We do this by adding the following code to this file:

```
<div class="sidebar">
    <app-sidebar></app-sidebar>
</div>
```

So the final app.component.html will look like

```
<div class="wrapper">
    <div class="sidebar">
        <app-sidebar></app-sidebar>
    </div>
    <div class="main-panel">
        <div class="main-content">
            <router-outlet></router-outlet>
        </div>
    </div>
</div>
```

When we save the code an go back to our application running in the browser, nothing will change yet. To be able to show the sidebar we have to create a navbar.

So let's create the navbar component. First we need a folder called navbar. In this folder we will create all the files related to the navbar component. Since the navbar is going to be a reusable component we put it in our components folder.

Create the folder by running the command:

```
mkdir ~/Geostack/angular-apps/base-application/src/app/components/navbar
```

Now we want to create a HTML-file which will contain the layout of our navbar. We do this by running the command:

touch ~/Geostack/angular-apps/base-application/src/app/components/navbar/navbar.component.html

We are going to add the following code to that file:

```
<!-- Below we create an HTML element: "nav" because of this element the application
know that everything inside this element had to do with the navigation bar.  -->
<nav class="navbar navbar-expand-lg navbar-transparent navbar-absolute fixed-top">
    <!-- Below we create a div element with the class container-fluid this
    class makes sure everything inside this div element is displayed over the
    full width of the screen.-->
    <div class="container-fluid">
        <div class="navbar-wrapper">
          <!-- Below we add the title of our sidebar, we call the function
          getTitle() defined in our navbar.component.ts file-->
          <a class="navbar-brand">{{getTitle()}}</a>
        </div>
        <!-- Below we create the button which is the navabar Toggle, we add
        3 bars using <span> tags. We add the sidebarToggle() function, defined
        in our navbar.component.ts file a onClick function.-->
        <button mat-raised-button class="navbar-toggler" (click)="sidebarToggle()">
            <span class="sr-only">Toggle navigation</span>
            <span class="navbar-toggler-icon icon-bar"></span>
            <span class="navbar-toggler-icon icon-bar"></span>
            <span class="navbar-toggler-icon icon-bar"></span>
        </button>
    </div>
</nav>
```

Then we need to create a component.ts file in which we are going to put all the code logic related to the navbar. We do this by running the command:

touch ~/Geostack/angular-apps/base-application/src/app/components/navbar/navbar.component.ts

Let's open the newly created file and import the modules required to build the navbar component logic. We need to add the modules shown in the illustration below at the top of the file.

```
/*Below we import the modules required for the navbar to function propperly.*/
import { Component, OnInit, ElementRef } from '@angular/core';
import { ROUTES } from '../sidebar/sidebar.component';
import { Location } from '@angular/common';
import { Router } from '@angular/router';
```

Next we need to create the component metadata. To do this we need to add the following code to our navbar.component.ts file.

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

Next we need to create our Navbar class, global variables and constructor. We do this by adding the following code to the file:

```
/*
Below we create the class of the navbar component. We add a constructor and the
function ngOnInit().
*/
export class NavbarComponent implements OnInit {

    /* Here we create a global variable called: "listTitles". The list assigned
    to this variable will be empty in the beginning but will be filled with the
    titles of all the entries in the ROUTES list defined in our sidebar
    component.*/
    private listTitles: any[];

    /* Here we create a global variable called: "location".
    The active location (page0 will be assigned tto this variable.*/
    location: Location;

    /* Here we create a global variable called: "toggleButton".
    The HTML element representing the toggleButton which is defined in the
    HTML.component file will be assigned to this variable.*/
    private toggleButton: any;

    /* Here we create a global variable called: "sidebarVisible".
    The value of this variable is either true of false. When the sidebar is open
    the value of this variable will be true. When the sidebar is closed the
    value of this variable will be false.*/
    private sidebarVisible: boolean;

    /* Here we create the class constructor.
    We pass the location, an instance of an ElementReference and the Angular
    router as parameters in the constructor.
    The current location (page) is than assigned to the global variable location
    and the value of sidebarVisible is set to false.  */
    constructor(
      location: Location,  private element: ElementRef, private router: Router){

      this.location = location;

      this.sidebarVisible = false;
    }
  }
```

Now we want to add the ngOninit() function below the constructor and inside the NavbarComponent class. We do this by adding the following code:

```
/* Here we create the ngOnInit() function.
We assign all the listTitles of the ROUTES list in the sidebar.component.ts
file to the global variable:"listTitles" using the build in JavaScript
function: ".filter()"*/
ngOnInit(){
  // Here we assign all the titles in the ROUTES list to the listTitles.
  this.listTitles = ROUTES.filter(listTitle => listTitle);
  // Here we assign an nativeElement tot a variable navbar.
  const navbar: HTMLElement = this.element.nativeElement;

  // Here we assign the element with class navbar-toggler to the variable
  // togglebutton. This is our toggleButton in our navbar layout file.
  this.toggleButton = navbar.getElementsByClassName('navbar-toggler')[0];

  // Here we create the logic which is executed when the sidebar is toggled.
  // The function sidebarClose() is called and the layer which makes the
  // screen dark is removed.
  this.router.events.subscribe((event) => {
    this.sidebarClose();
     var $layer: any = document.getElementsByClassName('close-layer')[0];
     if ($layer) {
       $layer.remove();
     }
 });
}
```

Next up is creating the sidebarOpen() function which is triggered when the toggleButton is clicked and the sidebar is closed. We do this by adding the following code below the ngOnInit() function:

```
sidebarOpen() {
    // Here we assign the value of the global toggleButton to a constant
    // called toggleButton.
    const toggleButton = this.toggleButton;

    // Here we assign the HTML body element of the webapplication to a
    // constant called body.
    const body = document.getElementsByTagName('body')[0];

    // Here we add a timeout funtion which is executed after 500 milisecond.
    // This function adds the class: "toggled" to the toggleButton.
    setTimeout(function(){
        toggleButton.classList.add('toggled');
    }, 500);

    // The class nav-open is added to the body. When this class is added
    // the body get's an overlay which makes the content of the body dark.
    body.classList.add('nav-open');

    // We set the global variable: "sidebarVisible" to true.
    this.sidebarVisible = true;
};
```

Now we want to create the sidebarClose() function which is triggered when the toggleButton is clicked and the sidebar is open. We do this by adding the following code below the sidebarOpen() function:

```
/* Here we create a function called: 'sidebarClose()'
This function contains the logic which is executed when the toggleButton is
clicked and the sidebar closes.*/
sidebarClose() {

    // Here we assign the HTML body element of the webapplication to a
    // constant called body.
    const body = document.getElementsByTagName('body')[0];

    // Here we remove the class: "toggled" from the toggleButton element.
    this.toggleButton.classList.remove('toggled');
    body.classList.add('nav-open');

    // We set the global variable: "sidebarVisible" to true.
    this.sidebarVisible = false;

    // The class nav-open is removed from the body. When this class is
    // removed the dark overlay which makes the content of the body is
    // removed.
    body.classList.remove('nav-open');
};
```

Now we want to add the function that toggles the sidebar open or closed. This is a relatively long function so the multiple images will represent this code. The last line of the first image is the first line of the second image. We create this function by adding the following code below the sidebarClose() function:

```
sidebarToggle() {
    // Here we assign the element with class navbar-toggler to the variable
    // togglebutton. This is our toggleButton in our navbar layout file.
    var $toggle = document.getElementsByClassName('navbar-toggler')[0];

    // Here we add a check whether sidebar is toggled or not.
    // When it's not toggled the function: "sidebarOpen()" is triggered.
    // When it's toggled the function: "sidebarClosed()" is triggered.
    if (this.sidebarVisible === false) {
      this.sidebarOpen();
    } else {
      this.sidebarClose();
    }

    // Here we assign the HTML body element of the webapplication to a
    // constant called body.
    const body = document.getElementsByTagName('body')[0];

    // Here we add a timeout function which is executed after 500 miliseconds.
    // This function adds the class: "toggled" to the toggleButton.
    setTimeout(function() {
      $toggle.classList.add('toggled');
    }, 430);
```

```
    }, 430);

    // Here we create a new HTML div element and assign the class:
    // "close-layer" to it.
    var $layer = document.createElement('div');
    $layer.setAttribute('class', 'close-layer');

    // Below we add the code that makes sure that the layer is added to
    // the correct div element.
    if (body.querySelectorAll('.main-panel')) {
      document.getElementsByClassName('main-panel')[0].appendChild($layer);
    }else if (body.classList.contains('off-canvas-sidebar')) {
      document.getElementsByClassName('wrapper-full-page')[0].appendChild($layer);
    }

    // Here we add a timeout function which is executed after 100 miliseconds.
    // This function adds the class:"visible" to the newly created div .
    setTimeout(function() {
      $layer.classList.add('visible');
    }, 100);

    // The function below makes sure that when the sidebar is open and thus
    // a dark overlay is created and we click on the overlay, the sidebar
    // is closed en the overlay is removed.
    $layer.onclick = function() {
      body.classList.remove('nav-open');
      this.mobile_menu_visible = 0;
      $layer.classList.remove('visible');
      setTimeout(function() {
          $layer.remove();
          $toggle.classList.remove('toggled');
      }, 400);
    }.bind(this);

    body.classList.add('nav-open');
};
```

The last function we need to add is a function which obtains the Title of the pages:

```
// Below we create the function: "getTitle()" which obtains the title
// of the active path(page) and passes the title to the layout.html page.
// If no title is available we pass the default title: "Dashboard".
getTitle(){
  var titlee = this.location.prepareExternalUrl(this.location.path());

  if(titlee.charAt(0) === '#'){
      titlee = titlee.slice( 1 );
  }

  for(var item = 0; item < this.listTitles.length; item++){
      if(this.listTitles[item].path === titlee){
          return this.listTitles[item].title;
      }
  }
  return 'Dashboard';
}
```

Now that we have finished creating the layout and logic of our navbar component, we need to add it to the app.module.ts file. So let's open it.

First we need to import the component. This is done by adding the following code:

```
import { NavbarComponent } from '../app/components/navbar/navbar.component';
```

Now we need to add the NavbarComponent to our declarations section in the app.module.ts file. When added the declarations section should look the same as in the illustration below:

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

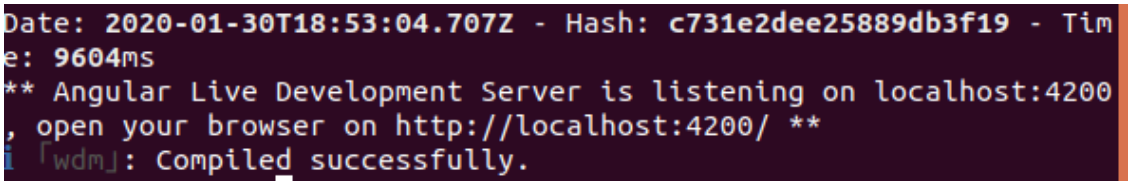
Finally we can add the NavbarComponent to the app.component.html file. The final app.component.html should look the same as in the illustration below. In red encircled is the navbar we needed to add using the selector: "<app-navbar></app-navbar>"

```
<!--First we create a wrapper which will contain all the components in our
application.-->
<div class="wrapper">

    <!--In the wrapper we create a div element with the class: 'sidebar', this
    div element will contain the sidebar component which is called using
    the selector <app-sidebar> </app-sidebar> -->
    <div class="sidebar">
        <app-sidebar></app-sidebar>
    </div>

    <!--Now we create a new div element inside the wrapper. This div element has
    the class: "main-panel". Inside the main-panel we create a new div element
    called: "main-content". these elements make sure the content of our pages in
    the webapplication are displayed properly-->
    <div class="main-panel">
        <div class="main-content">

            <!--Below we the navbar component is added using the selector:
            "<app-navbar></app-navbar>"-->
            <app-navbar></app-navbar>

            <!--Below we the angular router is added using the selector:
            "<router-outlet></router-outlet>". The router is used to switch
            between components.-->
            <router-outlet></router-outlet>
        </div>
    </div>
</div>
```

That’s it! Now you have created a sidebar and navbar component which are used to switch between different pages. If you start the application by running the following command from the root folder of the base application (~/Geostack/angular-apps/base-application):

```
sudo npm start
```

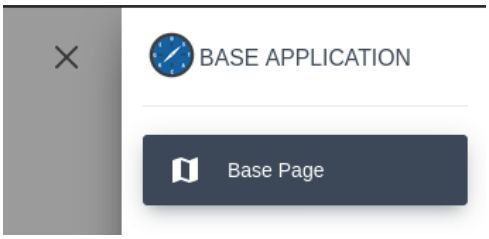
Now wait for a little while for the modules to be compiled and the Angular live development server to be started. If the application is finished loading you the output in the terminal should be similar to the one in the illustration below.



When pressing Ctrl+ left-mouse click on the link displayed in the terminal you will be greeted with our update application. Now click on the burger icon shown in the illustration below

Now open the sidebar in our web application by clicking on the burger icon shown in the illustration below.

You will be greeted with an entry in the sidebar as shown in the illustration below.



This entry is not clickable yet since the page, to which this entry is linked, does not exist yet. In the next chapter we are going to create this page.

4. Creating our first web application page

Our application is still very empty. To fix this we need to add a page. All the pages of our web application(s) are created in the pages folder. So let's create a new folder called "base-page". In this folder we will create all files and code related to the base-page.

To create the folder, run the following command:

`mkdir ~/Geostack/angular-apps/base-application/src/app/pages/base-page`

Then we want to create a file which contains the layout of our base page. This is done by running the command:

`touch ~/Geostack/angular-apps/base-application/src/app/pages/base-page/base-page.component.html`

Since this is just our base-application, we are not going to add any content to this page except for a title to check if the page actually works. Add the following code to the base-page.component.html file:

```
<!-- Below we add the text that will be displayed when we navigate to our
base page we add the styling to make sure the text is displayed in the correct
position-->
<h1 style="padding-top:80px; margin-left: 40px;">Welcome to our base application</h1>
```

This is the title that will be displayed when opening the base-application and thus our base-page.

Then we want to create a file which contains the code logic of our base page. This is done by running the command:

`touch ~/Geostack/angular-apps/base-application/src/app/pages/base-page/base-page.component.ts`

First we are going to add some basic imports in this file. The code for this is shown in the illustration below.

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

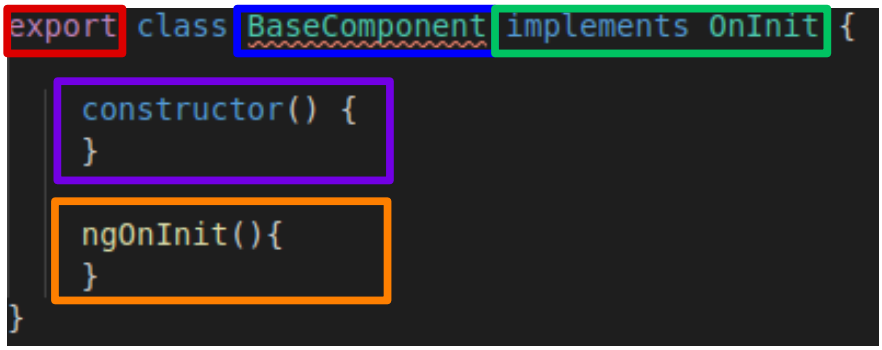
Next we want to create the component metadata. This is done using the following code:

```
@Component({
  selector: 'app-base',
  templateUrl: './base-page.component.html',
})
```

The following applies to the code shown above:

- ➔ We can use the base page throughout our application by using the selector: <app-base/>. (Red)
- ➔ The template HTML page which contains the layout of this component is called: ./ base-page.component.html. (Green)

Now we want tot create the component class which implements the OnInit functionality from Angular. The code for this is shown in the illustration below.



The following applies to the code shown above:

- ➔ We call the class BaseComponent, which is also the name of the component. (Blue)
- ➔ We implement the OnInit functionality from Angular. (Green)
- ➔ We create a constructor which is empty since this is just our base application. (Purple)
- ➔ We create a ngOnInit() function which will trigger when the base-page / BaseComponent is loaded. We also keep this function empty since this is just our base application. (Orange)

Now that we have created our base page, we want to import the component in our app.module.ts file. So lets open this file and import the BaseComponent at the top of this page by using the following code:

```
import { BaseComponent } from '../app/pages/base-page/base-page.component';
```

Next we want to add the imported component to the declarations section in this file, just like we did with the navbar and sidebar. The final declarations section should look the same as shown in the illustration below:

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

Now that we can use our base page throughout our web application, we want to be able to navigate to the page. For this we are going to use Angular routing. The routing is done in the file app-routing.module.ts. So let's open this file and import our BaseComponent below the default imports in this file. The following code is used to import the module:

```
import { BaseComponent } from '../app/pages/base-page/base-page.component';
```

Now we want to add some routes to the route list in the app-routing.module.ts file. The empty route list is shown in the illustration below.

```
const routes: Routes = [];
```

To add the routing for our base-page we add the following line of code to the list:

```
{ path: 'base-page', component: BaseComponent},
```

This means that when we navigate to http://localhost:4200/base-page the BaseComponent will be loaded and thus our base-page.

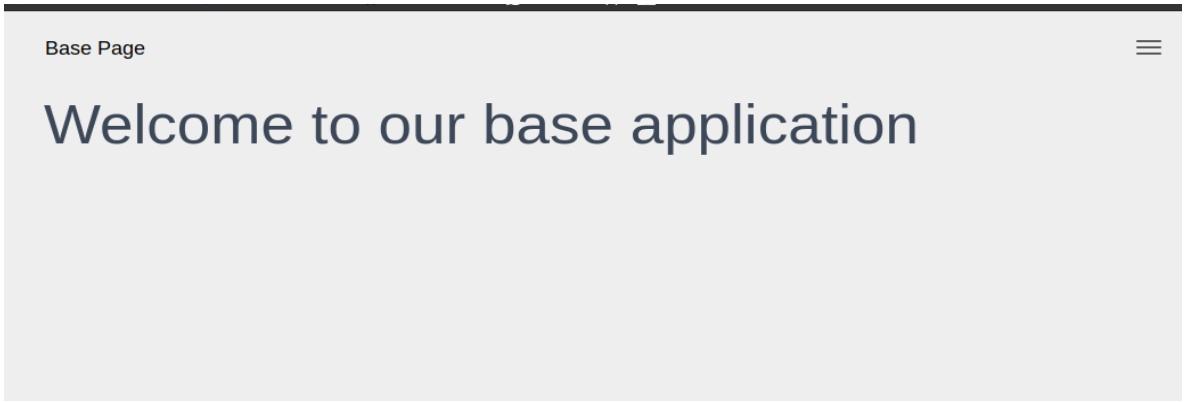
Now we want to make sure that when we navigate to http://localhost:4200/ we are automatically redirected to the base-page.

For this we add the following line to the routes list:

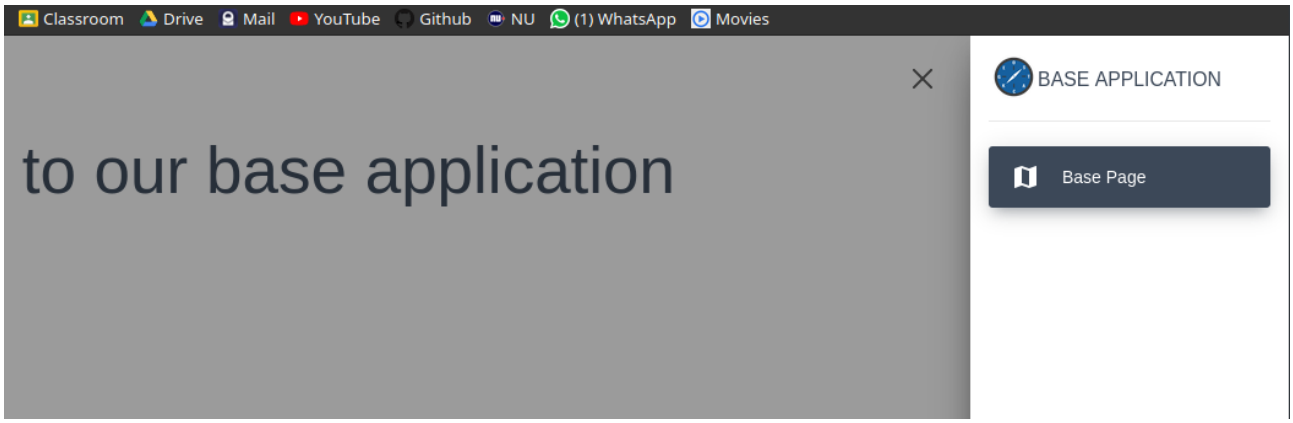
```
{ path: '', redirectTo: 'base-page', pathMatch: 'full',},
```

This means that when we navigate to localhost:4200, we are redirected to value assigned to the redirectTo variable, which is base-page in this case. So when we are redirected to the base-page the BaseComponent is shown.

Now when we save our code and navigate to http://localhost:4200 we should be greeted with the web page shown in the illustration below.



When we click on the burger icon in the top right of the web page, the sidebar should open as shown in the illustration below.



5. Adding a proxy configuration for our data

The last thing we are going to do is adding a proxy configuration file. This file makes sure are links is created from our Flask-API to our web applications.

The proxy configuration file is located in the root of the project folder, is going to have the JSON file format and is called: “proxy.conf.json”. So let’s add this file by running the command:

```
touch ~/Geostack/angular-apps/base-application/proxy.conf.json
```

Using the proxying support in the Angular live development server we can catch certain URLs and send them via a backend server which is our Flask-API.

Our Flask-API is running on http://localhost/api and we want all calls made to the URL: http://localhost:4200/api to go to that server. To be able to do that we need to add the following to the proxy.conf.json file.

```
{
  "/api/*": {
    "target": "http://localhost/api/",
    "secure": false,
    "changeOrigin": true,
    "pathRewrite": {
    "^/api": ""
    }
  }
}
```

If you need to access a backend that is not on localhost, you will need to add the changeOrigin option. Since our API is located on localhost, we don’t have to do this. But when our API is running in a Docker container, this line is required so let’s add it just to be sure.

Then we need to change the line which starts our Angular application when running: “sudo npm start”. This line is located in the file: “package.json”. So let’s open this file and edit the line:

```
"start": "ng serve",
```

to:

```
"start": "ng serve --proxy-config proxy.conf.json",
```

Now when we run the Angular application, the proxy configuration will be used.

To be able to retrieve data via our Flask-API we need to add an extra module to our app.module.ts file. This module is called: “HTTPClientModule” and is required to perform GET, POST and PUT requests. So let’s open the file and import this module by the following line of code at the top of the file:

```
import { HttpClientModule } from '@angular/common/http';
```

Then we need to add this module to the import section in the app.module.ts file. The final section should look the same as the one in the illustration below.

```
imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule,
],
```

The last thing we have to do is import some extra modules in our app.module.ts file. These modules are not required for our base-application to work properly, but they are required for the other applications which we are going to create. The modules are used to add some extra styling to our applications and to create Angular Forms.

To import these modules we add the following code at the top of the app.module.ts file.

```
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
import { FormsModule } from '@angular/forms';
```

We also need to add these modules to the import section in the app.module.ts file. The final import section will look the same as shown in the illustration below.

```
imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule,
  NgbModule,
  FormsModule,
],
```

That's it for the basic Angular web application. Now that we have the base for our other applications you can start reading the next programming manuals. It's highly recommended to start with creating the Dataset Dashboard since this is also a relatively simple application to create.

In the manual: Creating-A-Dataset-Dashboard you will learn the basics of retrieving data from our MongoDB datastore and displaying this data in a web application. This information comes in useful when creating the 2D and 3D Map Viewer applications.