

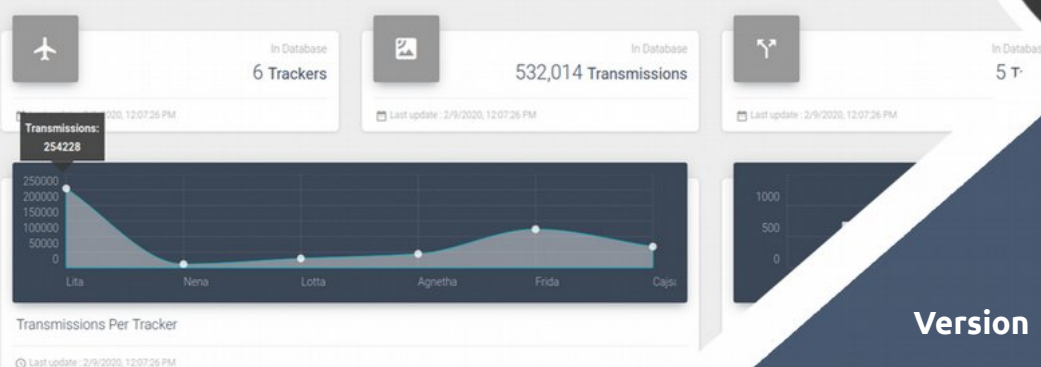


Cookbook

Workshop Assignments



GPS Dashboard



Categories: TRACKERS TRAILS			
MongoID	Local Identifier	Crane name	Study name
5e3ec20e1467864f6917b85	9407	Agnetha	GPS
5e3ec2c51467864f6940d1a	9472	Cajsja	
5e3ec23c1467864f692297c	9381	Frida	

Version : 1.0

Date : 09-10-2020

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 – How it works...

This cookbook 'Workshop Assignments' serves as a 'programming manual' for the second step in the 1-day GeoStack Workshop after working through the first step of creating the Virtual Machine for this workshop as described in the installation cookbook 'Creating-the-Workshop-VM.pdf'.

- The main focus of this cookbook is to provide **Lots of Background Information for Beginners** and **Basic Instructions for the Programming Assignments!**
- If you are NOT a Newbie then you can probably skip a lot of information in this cookbook!

The GeoStack Workshop is a 'Must for Newbies' and the introduction to the full 'Beginner Course in Open Source Geospatial Programming for Data Scientists', simply called the GeoStack Course!

This cookbook 'Workshop Assignments' will guide you in learning the 3 main steps to:

- 1) get online digital topographical maps from OpenStreetMap into the cache folders of the TileStache tile server. And also get datasets of GPS track logs into the MongoDB datastore;
- 2) get the geospatial and data processing middleware running to provide 2 web services to access the digital maps and datasets. And also to provide a web server for the web app;
- 3) program a simplified version of the 2D Map Viewer web application to view dataset plots of GPS track logs in a web browser on the locally cached maps from OpenStreetMap.

To guide you through the programming assignments in the Python programming language there are user friendly Jupyter Notebooks with lots of comments to follow in your web browser.

You will learn how the data gets into the datastores, how to setup the middleware and how the architecture of all the server software components that are required are related to each other.

To keep things simple in the workshop some steps are deliberately skipped, for example how to load maps from OpenStreetMap in PostgreSQL. These explanations are in the GeoStack Course.

To help in learning the basics of programming a web app in the TypeScript programming language with the Angular and OpenLayers frameworks there are source code files with lots of comments.

The source code files need to be opened in the Atom programming editor for code completion of programming statements and configuration specifications to turn these programming assignments into working code by precisely following the instructions in the chapters of this cookbook!

For each completed programming assignment you can run an Assignment Script from the Ubuntu Desktop in the Workshop Virtual Machine that will check if your programming solution is correct and resulted in working code and if so the script will run your code for you too.

If you didn't succeed in getting the example source code files into working code or you just want to peek ahead for what the end result should be, then you can run the accompanying Solutions Script from the Ubuntu Desktop to run the already completed source code file that is provided too.

Also, you can open the completed source code files in your programming editor to study the working source code and the explanations in the inline comments. No cheating though!

Prerequisites for the Workshop Assignments are as follows:

- **A computer with a working internet network connection.**
- **The Workshop Virtual Machine you created earlier for this GeoStack Workshop.**

Now you can start off by starting the Workshop Virtual Machine in which you can work through the programming assignments by following this cookbook. ***So, now Go, Enjoy and... Learn things!***

Table of Contents

0 Workshop - Part 0 – Introduction.....	5
0.1 Introduction – The goals of this workshop!.....	5
0.2 Study Time Estimate.....	5
0.3 Building Skill Levels.....	6
0.4 Workshop Overview.....	7
0.5 Workshop End Product.....	8
0.6 Terminology Explained – How a web application runs!.....	9
0.7 The GeoStack – The Server Software Architecture.....	10
0.8 Development Environment.....	11
0.9 Workshop Folder Structure.....	12
0.10 Preparations Checklist – Workshop Assignments.....	13
1 Workshop - Part 1 – Data Processing: Exploring!.....	14
1.1 What are you going to do?.....	14
1.2 What do you need?.....	14
1.3 The Workshop Datasets – Flight Paths & Car Routes.....	15
1.4 Ringing Crane Birds with GPS trackers.....	17
1.5 About the GPS track logs in CSV files.....	20
1.6 Explore the CSV files.....	21
1.7 Assignment – Prepare Visual Exploration of the Datasets.....	22
1.8 About the GPS track logs in GPX files.....	24
1.8.1 GPX files explained – Some Workshop Basics.....	24
1.8.2 Challenge - Exploring GPX files Online and on the Desktop.....	25
1.9 Assignment – Simple Map Plotting in Python with Cartopy.....	26
1.9.1 Assignment – Plotting the Crane datasets on a Topographical Map.....	26
1.9.2 Assignment – Plotting the Car datasets on a Topographical Map.....	32
1.9.3 Challenge – Build extra skills in Python with Cartopy.....	33
2 Workshop - Part 2 – Data Storage: MongoDB.....	34
2.1 Data Storage Environment for Datasets.....	34
2.2 Terminology explained – Data storage in MongoDB.....	35
2.3 MongoDB use in the GeoStack Workshop.....	36
2.4 Exploring Data Modeling in MongoDB.....	38
2.4.1 Crane Datasets – Modeling the Tracker collection.....	38
2.4.2 Crane Datasets – Modeling the Transmission collection.....	39
2.4.3 Crane Datasets – Modeling the Tracker and Transmission relation.....	40
2.5 Assignments – Getting the Datasets in MongoDB.....	41
2.5.1 Assignment 1 – Programming the MongoDB Data Models.....	41
2.5.2 Assignment 2 – Loading the Crane and Car datasets.....	42
2.5.3 Assignment 3 - Checking the Datasets in MongoDB.....	44
2.5.4 Assignment – Some Black Magic Explained for Newbies!.....	45
2.5.5 Summary of the Assignments.....	52
2.6 A Sneak Peek at MongoDB in the Course.....	52
3 Workshop - Part 3 – Data Usage: Middleware.....	53
3.1 GeoStack Middleware – Web Services Explained.....	53
3.1.1 Development Environment for the GeoStack Middleware.....	53
3.1.2 Sketching the Software of the GeoStack Middleware.....	54
3.1.3 Data usage – The URLs for the Middleware Web Services.....	55
3.2 Flask API – Micro Web Services in Python.....	56
3.2.1 Flask API – How Data Queries work.....	56
3.2.2 Flask API – How the Flask App works.....	57
3.3 Assignment 1 – Programming the Flask API.....	58
3.4 TileStache – Tile Server.....	59
3.4.1 TileStache – How the tile server daemon works.....	59
3.4.2 TileStache – How the tile server configuration works.....	61
3.5 Assignment 2 – Configuring TileStache.....	63
3.5.1 Assignment – Completing the TileStache Configuration File.....	63

3.5.2 Challenge – Read up on Map Styles and Vector Tiles.....	64
3.6 NGINX – Web server.....	65
3.6.1 Terminology Explained – NGINX as a web and proxy server.....	65
3.6.2 Positioning NGINX in the GeoStack.....	66
3.6.3 NGINX configuration for the Flask API explained.....	67
3.7 Assignment 3 – Configuring NGINX.....	69
3.7.1 Assignment – Completing the NGINX Configuration File.....	69
3.7.2 Assignment – Checking the End Result!.....	70
3.7.3 Assignment – Starting the Middleware Services.....	72
4 Workshop - Part 4 – Data Visualization: Web App!.....	73
4.1 Web Application Programming – How does it work?.....	73
4.1.1 Web Server Hosting of the Web Application.....	73
4.1.2 Development Environment for the Web Application.....	74
4.1.3 Illustrating the Inner Workings of the Web Application.....	76
4.1.4 Illustrating the Simplified 2D Map Viewer web application.....	78
4.1.5 Angular Explained – Basics of the 4 Component Files.....	79
4.2 Challenge – How Angular Development Works 4 Newbies!.....	80
4.2.1 Reading instruction to complete the Angular Development!.....	80
4.2.2 Challenge instruction.....	80
4.2.3 Running an Angular Web Application in the Web browser.....	81
4.2.4 Installation of the Angular Development Environment in the VM.....	81
4.2.5 Installation of the two ADE Base Components.....	82
4.2.6 Installation of the Angular CLI package.....	82
4.2.7 Starting a New web application project in Angular.....	82
4.2.8 How the web app installation works for the workshop.....	83
4.2.9 Running an Angular web application.....	83
4.2.10 The Angular run time difference for index.html.....	83
4.2.11 What-goes-Where? --> in an Angular Project Folder!.....	84
4.2.12 Event handling basics in Angular Apps.....	85
4.2.13 Best Practice in Angular Programming.....	85
4.2.14 Terminology --> Single Page App + Bootstrapping + Routing.....	85
4.2.15 What-does-What? --> The inner workings of an Angular App!.....	86
4.2.16 Completing the Circle --> Getting the Crane Data in map.component.ts.....	88
4.2.17 Completing the Circle --> Drawing the Data on the Map in map.component.ts.....	89
4.2.18 Scope Limitation --> Centering the Map at [0,0] explained!.....	91
4.2.19 Scope Limitation --> No Map Refresh when changing Cranes.....	93
4.2.20 How Angular Apps Work Exactly! --> Web links for more!.....	93
4.3 Assignment – Preparing for Web Programming.....	94
4.3.1 Assignment – Understanding the Programming Steps!.....	94
4.3.2 Assignment – Start the Angular Live Development Server.....	95
4.4 Assignment – Complete the TypeScript Service file.....	97
4.4.1 Angular Explained – Component Services.....	97
4.4.2 Assignment – Complete the Component Service File.....	98
4.4.3 Assignment – Check the Result in the Web App's Settings Menu.....	99
4.5 Coding Explained – The Angular and OpenLayers Magic.....	100
4.5.1 Angular Explained – Some Basics on the TypeScript Code File.....	100
4.5.2 OpenLayers Explained – Some Basics on Coding Layers.....	101
4.5.3 The Map Component Explained – map.component.html.....	103
4.5.4 The Map Component Explained – map.component.ts.....	104
4.6 Assignment – Complete the 2D Map Viewer Web App.....	105
4.6.1 Assignment – Check the Angular Development Environment.....	105
4.6.2 Assignment – Open the TypeScript Code file map.component.ts.....	105
4.6.3 Assignment – Plot Geospatial Data on a Topographical Map.....	106
4.6.4 Assignment – Route Animation --> Playback of a Time Series.....	107
4.6.5 Assignment – Checking the Crane's Flight Paths.....	107
4.7 Challenge – Add the Car routes to the Web App.....	108

0 Workshop - Part 0 – Introduction

The introduction about 'Plotting Geospatial Data on Digital Maps' is to explain what the end result of the workshop's programming assignments is going to be and how the assignments will guide you on how to get there.

0.1 Introduction – The goals of this workshop!

To not complicate things in a workshop intended for beginners in the geospatial programming world, the focus is on getting a simplified version up and running of the 2D Map Viewer web application from the full version in the GeoStack Course. The educational goals are to learn all the basic steps in programming and setting up a server side geospatial software stack, the GeoStack!

Learning goals:

1. Basic knowledge of the software architecture and the GeoStack components, including the explanation of some basic terminology on web applications and network connections.
2. Learning the workflow of data processing for geospatial datasets --> Explore & Store!
 1. You will learn about the ETL process: Extract, Transform and Load.
 2. Extract data by reading the raw data from their CSV and GPX files.
 3. Transform and select datafields to the JSON dataformat.
 4. Load the JSON formatted data as JSON documents in the MongoDB datastore.
3. Visualizing processed geographical data from MongoDB on digital topographical maps.
 1. Request map tiles from an online OpenStreetMap web map server;
 2. Store the map tiles in the cache folders of the TileStache tile server;
 3. Create a Python Flask app to get data from the MongoDB datastore;
 4. Run the TileStache and Flask applications as web services for the web application;
 5. Program a simplified 2D Map Viewer web application in TypeScript with Angular;
 6. Use the JavaScript geospatial OpenLayers framework to plot the data on a map.

0.2 Study Time Estimate

Required study time:

1. The 1-day workshop program is a global overview of the complete set of tutorials and guides in the full 10-day GeoStack programming course.
2. Of course the time it takes to complete the workshop or the course depends on the level of pre-existing knowledge the data scientist, programmer or system administrator already has.
3. So the workshop may take 1 – 3 days and the course may take 10 – 30 days depending on your already acquired skill levels. Some enthusiasm will get you there faster!
4. Important advice about taking extra study time:
 - Check the next section 'Building Skill Levels' to see if you need extra study time to build some extra skills first if you are a real beginner in programming.
 - Also read the sections 'Workshop End Product' and 'Terminology explained – How a web application runs!' to help you decide whether or not you need some extra reading time.

--> If so, then take a break from this workshop right here and right now to follow the study advice to enable yourself to successfully continue with this workshop and the full course!

0.3 Building Skill Levels

If you are a true beginner in programming in Python and TypeScript / JavaScript for data science, web applications or geospatial applications it might be a very good idea to invest some extra and well spent study time in building your skill levels before diving into this GeoStack Workshop and the full GeoStack Course after that.

See the explanation of the required and additional skill levels below that would require an extra study time of around 1 - 2 days of reading websites in order to understand the bigger picture and 5 – 10 days for an absolutely necessary basic Python programming course and 5 days to experiment with some TypeScript / JavaScript / Angular tutorials.

Skill levels:

1. Required skill level: ***a basic knowledge of Python programming is definitely a must!***
If you're unfamiliar with Python, then take 5 – 10 days to first follow a basic programming course in Python and learn how to work with JupyterLab and Jupyter Notebooks in your web browser too!
Learn the programming editors IDLE (Python IDE) and especially the Atom editor!
 1. Highly recommended from an educational point of view for data scientists is the great and totally Open Content beginner course 'Python 4 Everybody' by dr. Charles (Chuck) Severance that guides you through his course book 'Exploring Data in Python'!
 2. Weblinks: see <https://www.py4e.com/> for the course website and <https://www.py4e.com/book.php> for the free course e-book.
 3. Look here for Project Jupyter: <https://jupyter.org/> and here for Atom: <https://atom.io/>
 4. For IDLE look here: <https://docs.python.org/3/library/idle.html> (documentation) and here: <https://realpython.com/python-idle/> (tutorial).
2. Helpful additional skills: it helps a lot to have some basic knowledge about and experience with VirtualBox, Ubuntu Linux, Docker, MongoDB, JavaScript / TypeScript / Angular, Atom programming editor and the geospatial web application frameworks OpenLayers / Cesium. Surely, it's a great advantage, but don't worry if it's all pretty new to you! The workshop and course manuals will guide you through the subjects from scratch!
 - Tip 1: if these subjects are indeed new then first take a day or two to read up a little on them by reading their wikispaces on Wikipedia and their project websites.
 - Tip 2: if you're a beginner in programming with the Atom programming editor in TypeScript / JavaScript / Angular, then also read up on that a little too, to get a feel for the web app development environment and possibly invest like 5 days in doing some simple tutorials to experiment a little and learn to use the Atom editor too!
 - Tip 3: don't worry too much about experimenting with VirtualBox, Ubuntu Linux, Docker, MongoDB, OpenLayers and Cesium because you will learn along the way in the workshop and the course. Just read up on these subjects.

Basically, if you're lacking a few or even all of the additional skills, don't worry and just take a little more time to read up on some websites to get the bigger picture and work through a basic Python programming course and tutorials on programming web applications in TypeScript / JavaScript.

Then, at a little slower pace, go through the installation and programming manuals and source code examples of the GeoStack Workshop and GeoStack Course to really understand what's happening and you will definitely get yourself there! That's what this self-study course is for!

0.4 Workshop Overview

See the 'Time plan' and 'Workshop overview' on how to plan your 1-day GeoStack Workshop!

Time plan

1. The 1-day workshop is planned to take 9 hours in total from 9:00 – 18:00 hours.
2. 0.5 hours (30 minutes) to read this chapter: Part 0 – Introduction.
3. 6 hours for the 4 workshop parts of 90 minutes with 30 minutes of reading the part's chapter in this cookbook and 60 minutes for each part's programming assignment.
4. 1 hour of scheduled breaks.
5. 1.5 hours (90 minutes) of extra study time because depending on your skill level you might need a little more time for some parts of the workshop to get things working.

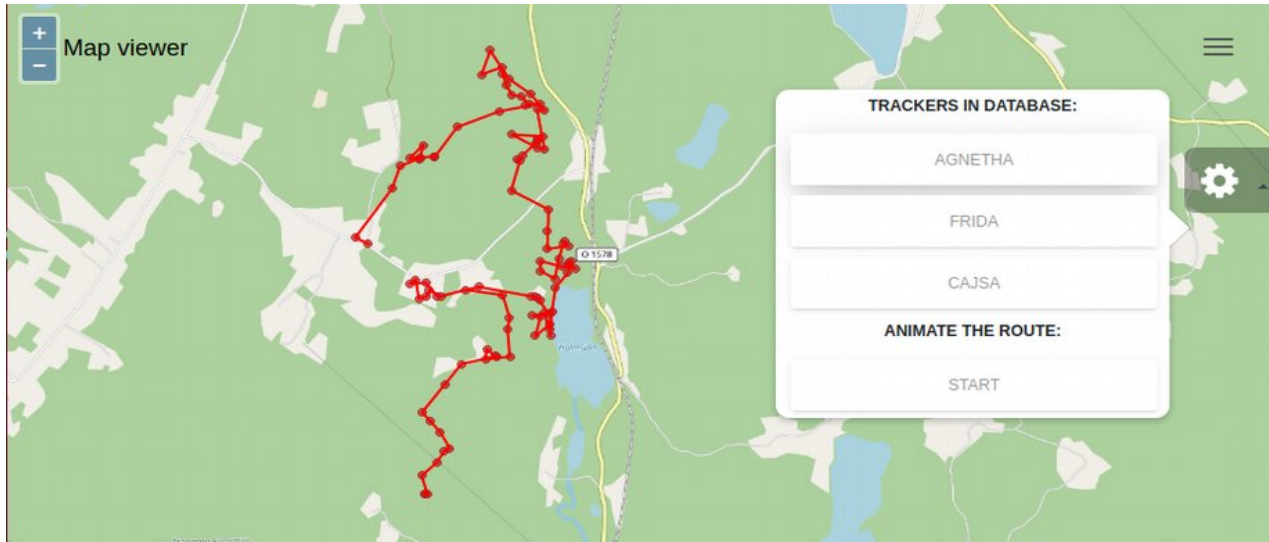
As stated before: depending on your skill level this workshop might take you an extra day!

Workshop overview

1. Part 0: Introduction Workshop (30 min / 09:00 – 09.30 hours)
 1. Read the document introduction page 'Purpose of this document – How it works...'
 2. Read this introduction chapter 'Workshop – Part 0 – Introduction'.
 1. Study advice in the sections 'Study Time Estimate' and 'Building Skill Levels'.
 2. What are you going to create? Presenting the Workshop End Product.
 3. Terminology explained – How a web application runs!
 3. Then start the Workshop Virtual Machine in VirtualBox to get started with Part 1!
2. Part 1: Data Processing and a Simple visualization (90 min / 09.30 – 11.00 hours)
 1. Reading data using Pandas and analyzing the datasets.
 2. Visualizing the datasets using Python Cartopy for a quick visual exploration.
 3. Convert the datasets to the JSON file format to learn about the ETL process.
3. Break (15 min / 11.00 – 11.15 hours)
4. Part 2: Data Storage in the backend (90 min / 11.15 – 12.45 hours)
 1. Creating efficient data models for PostgreSQL and MongoDB to store / search data.
 2. Importing data using the data models to store the datasets in the datastores.
5. Break – Lunch (30 min / 12.45 – 13.15 hours)
6. Part 3: Data usage and Middleware setup (90 min / 13.15 – 14.45 hours)
 1. Tile server setup for a web service to serve map tiles (parts) of OpenStreetMap maps.
 2. Flask-API setup for a web service to serve datasets from the MongoDB datastore.
 3. Connecting to our datastores PostgreSQL for OSM maps and MongoDB for datasets.
 4. Creating queries to get the data of digital maps and GPS track log datasets.
 5. NGINX Web Server setup as a front-end HTTP web server for the web browser.
 6. Connecting the components: setup NGINX as reverse-proxy bridge server to the Gunicorn WSGI web server to run the Python Flask WSGI web application.
7. Break (15 min / 14.45 – 15.00 hours)
8. Part 4: Data Visualization in the Front-end (90 min / 15.00 – 16.30 hours)
 1. Development environment setup to program the 2D Map Viewer web application.
 2. Learn to program the following visualizing features in the simplified 2D Map Viewer:
 1. Showing a digital topographical base map with OpenStreetMap maps.
 2. Showing the data points of a GPS track log.
 3. Connecting the data points with lines to see the route of the track log.
 4. Animating the route of a GPS track log.
9. Extra study time (90 min / 16.30 – 18.00 hours)

0.5 Workshop End Product

The goal of the workshop is to get a simple 2D Map viewer web application up and running which looks like in this screen shot below.



Drawing 1: Screenshot of the Simplified 2D Map Viewer – GPS track log visualized in red on an OpenStreetMap topographical map.

The three (3) basic controls you will learn to program in the basic source code example are:

1. Settings menu (gear icon on the right)
 1. The menu has two (2) options: 1) select a dataset and 2) run an animation
 2. The datasets are GPS track log datasets from GPS trackers on 3 Swedish ringed common cranes (*Grus grus*).
 - For this workshop the crane birds were named after the well know Swedish singers Agnetha Fältskog and Anni-Frid (Frida) Lyngstad from ABBA and singer-songwriter Cajsa Stina Åkerström.
 - Their first names are used as the dataset names for the data of the GPS trackers in the database in the settings menu of the app, as shown in the screenshot above.
 3. Once the dataset is selected you can choose to start an animation of the GPS track log from this settings menu to visualize the crane bird's movements over time.
2. Application menu ('hamburger' menu at the top right).
3. Zoom controls (the blue '+' and '-' buttons at the top left)

Required Software architecture

1. A MongoDB datastore containing the processed and modeled geographical datasets.
2. A TileStache tile server (<http://tilestache.org/>) to store map tiles of digital topographical base maps in a cache folder from TileStache as PNG files to serve to the web browser.
 - For the workshop these map tiles are requested from an online OpenStreetMap Web Map Server (WMS server). In the full course OSM data is locally stored in PostgreSQL.
3. A Python Flask application that offers a middleware micro web service with an API for requesting the GPS track log data from our MongoDB datastore and passing it on after some processing to the front-end client, which is the 2D Map Viewer web application.
 - It is good to realize that this Flask application is where the data magic happens!
 - This is where the Python programmed 'payload code' with the 'business logic' of the web application is run for data processing.
 - The 2D Map Viewer itself is a pure web application in HTML5 + TypeScript, completely dedicated to presenting the Graphical User Interface (GUI) in the web browser.
 - So separate the data processing from the data presentation in different applications!
4. The 2D Map Viewer web application programmed in TypeScript with the Angular framework for it's functionality and the OpenLayers framework for visualizing the processed and modeled geographical data.

0.6 Terminology Explained – How a web application runs!

Later you will learn more about local hosting of web applications, running local web services and choosing network port numbers for your local server 'daemon' processes but some of the terminology is already explained below, so read this section very carefully!

About the Web Application

Remember, a web application is a software application that runs on a web server program that is hosted on a remote server computer.

The web application is accessed over a network connection by using a web browser on the client computer to send requests for data, using the HTTP or HTTPS protocol, to a web server that passes those data requests to the web application to process!

From a software architecture point of view the GeoStack is a Client / Server design pattern.

Therefore a web application as a component in the software architecture is either a middleware program or an end-user program in the GeoStack software architecture.

If a web application is a middleware part, it is run to offer a web service API (Application Programming Interface) on the network to other web applications. A middleware web application is typically used to get, pre-process and present data to an end-user web application.

If a web application is an end-user application, it is run as a front-end program in the GeoStack software architecture. It will use one or more web services to get it's data and then process the acquired data to return that data as the answer to the HTTP request back to the web browser.

About the Web Server

To be able to run a web application on a web server program, like the NGINX web server, on your local computer and use that web application in your web browser requires your computer to be able to act both as a client and as a server computer at the same time.

To do that Client / Server trick, the Linux operating system offers a simulated network connection called 'local host' and web browsers can access these local web services with the URL of `http://localhost` or `http://127.0.0.1`

Also when you start a server program you can choose on which network port number the service will run to offer its network connection for a client program. This process to assign a chosen port number to a network service is called 'port mapping'.

Ports are also often referred to as 'network sockets' and some well known reserved port numbers are port 80 for HTTP and port 443 for HTTPS network connections or port 22 for SSH connections.

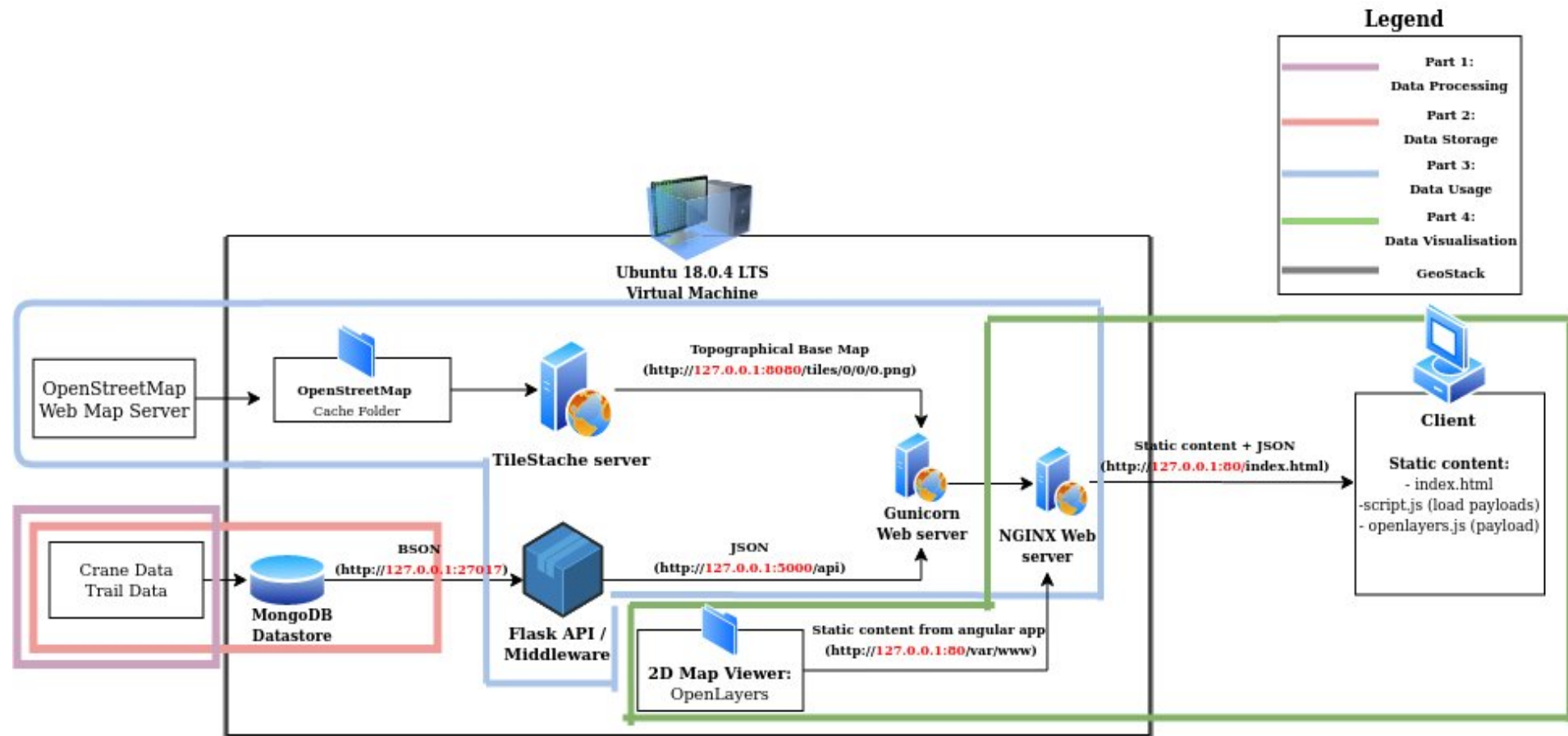
For instance when you start a web server you would not want it to run on a reserved port 80 or 443 because that would block a normal application like the web browser to use the internet.

That is why port mapping is available to start your network service on a port number of your choice (= network socket), let's say to start the Python Flask development web server on port 8080, which would give you an URL of `http://localhost:8080` or `http://127.0.0.1:8080` to access it.

Most server programs already have a default port to run on so most of the time it is fairly simple to run a network service. You will need an understanding of how this works though, when configuring the network ports to connect the NGINX web server as a reverse-proxy 'bridge' server (that talks HTTP to the browser on one side) to the Gunicorn web server (that talks WSGI to the Python Flask web application on the other side). So Gunicorn runs the Flask app behind NGINX as a front-end!

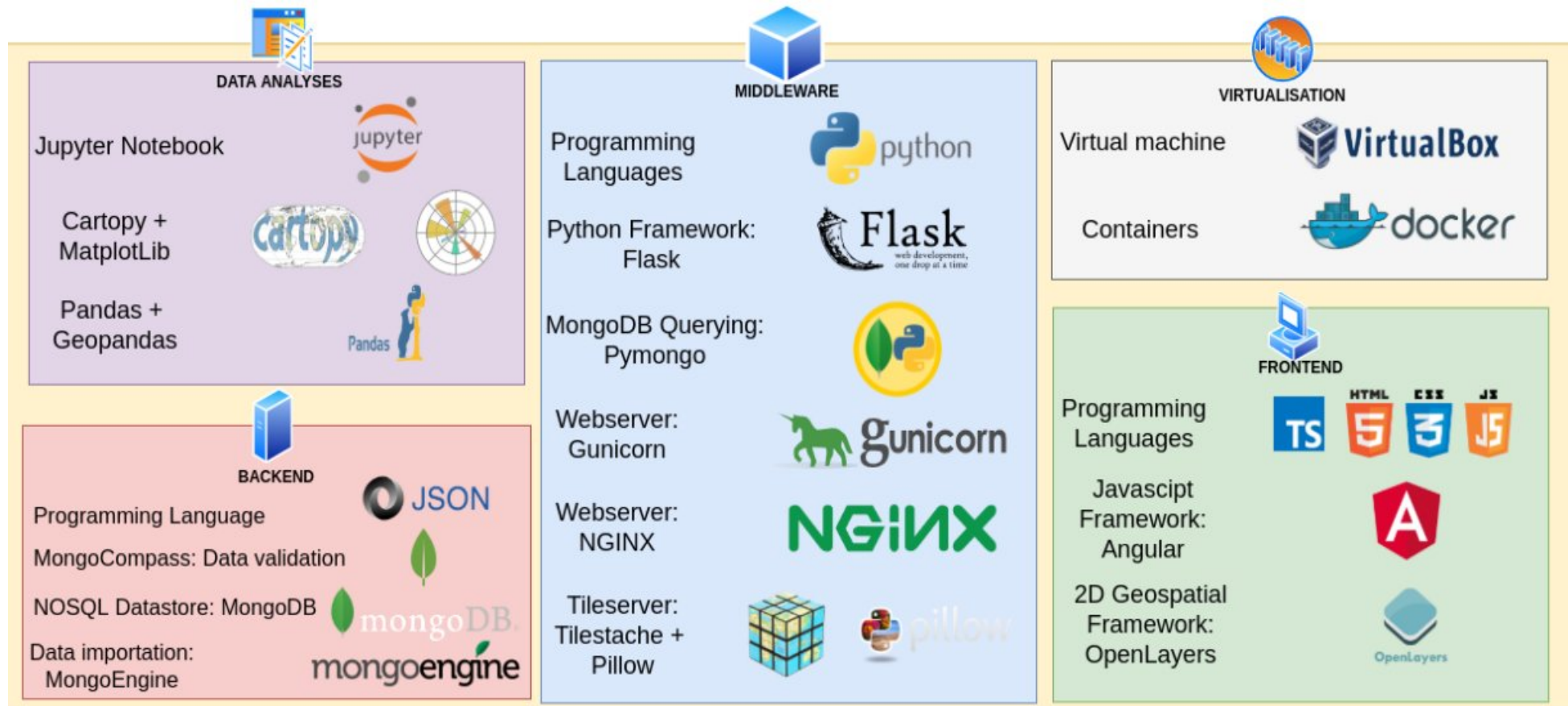
By the way, a server program that runs as a service is called a 'daemon' process in Linux and these background processes have a name often ending with the letter 'd' from daemon, like the Linux system services that run as the process systemd for 'system daemon' and there is a web server program creatively called `httpd` which is short for 'http daemon'.

0.7 The GeoStack – The Server Software Architecture



In Black: the GeoStack with the MongoDB datastore in red as back-end, the middleware web services in blue and front-end web services in Green.
 In Blue: the OpenStreetMap Web Map Service provides the map tiles of digital topographical maps the TileStache tile server serves to the browser.
 In Red: the dataset raw files (CSV + GPX) need to be transformed to JSON and loaded in MongoDB for the Flask app to provide as a web service.
 In Green: the web browser on the client sends HTTP requests to NGINX for the 2D Map Viewer web app and serves the apps middleware services.
 In Blue: NGINX acts as a bridge server to the Gunicorn server to get datasets from the Python Flask app and to get maps from the Python tile server.

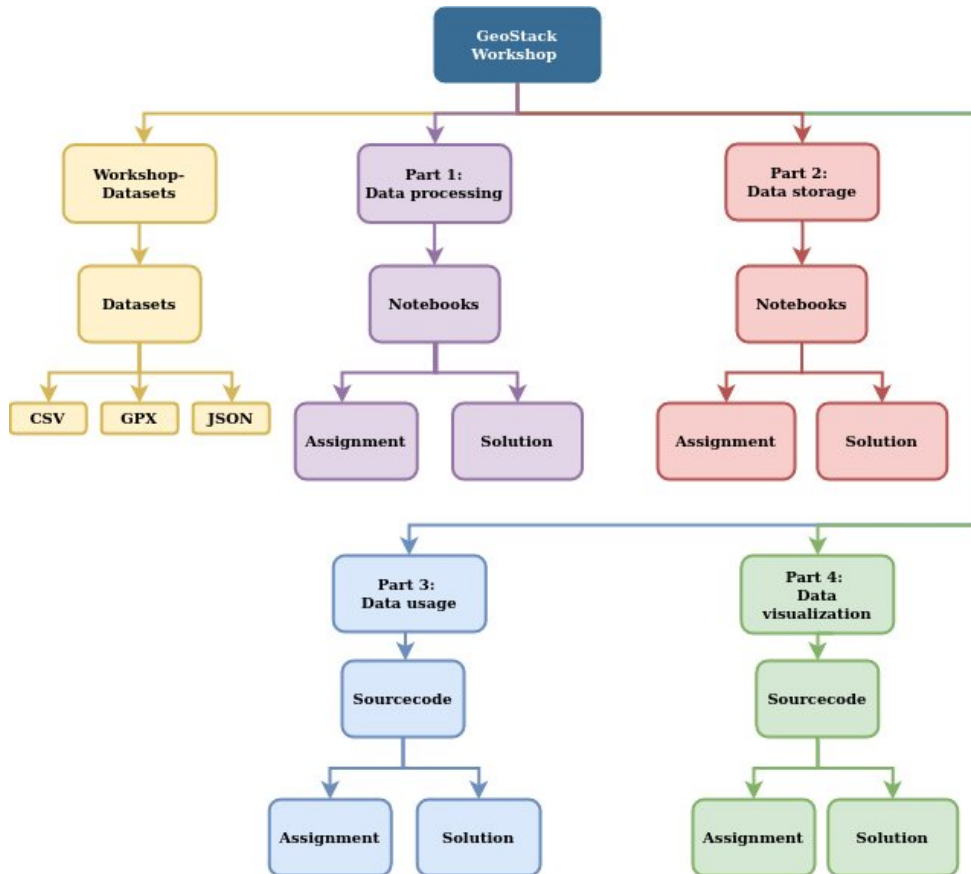
0.8 Development Environment



The components of the Development Environment and the GeoStack you need to read up on! Some for the Workshop and on all for the full Course!

- Tip: read the wikispaces on Wikipedia about these products and also read their project websites!
- Tip: also get familiar with the programming editors IDLE, the standard Python IDE (<https://docs.python.org/3/library/idle.html>) and especially the Atom editor (<https://flight-manual.atom.io/>) because you will be working mainly in Atom to program scripts and edit configuration files!

0.9 Workshop Folder Structure



There are two easy ways to get to the workshop files:

1. Directly access files by clicking the folders for Part 1 – Part 4 of the GeoStack Workshop on the Ubuntu desktop in the Workshop Virtual Machine.
2. Open the Nautilus file manager by clicking the folder icon in the left Favorites menu to access the Home folder and then click the GitHub repository folder GeoStack-Workshop to access the folders for Part 1 – Part 4 of the GeoStack Workshop to find the files.

The datasets for the workshop are located in the folder Workshop-Datasets which is divided into the subfolders CSV, GPX and JSON to store the datasets in these different file formats.

Each part of the workshop either has a Notebooks folder holding the Jupyter Notebook files or a Sourcecode folder holding the script files and they both have 2 sub-folders:

1. An Assignment folder that contains the files for the programming exercises to complete in a Jupyter Notebook file, Python script, TypeScript script or configuration file.
2. A Solution folder that contains all the working solution scripts to the exercises in question and the solution desktop script files (Bash scripts) to run the provided working script file.
 - Check the files in this folder only if 1) re-reading and 2) searching the Internet didn't work to find an answer because peeking definitely defeats the educational goal!

Desktop shortcuts: these help in checking the completed assignments for errors.

1. Use the desktop shortcuts to the Assignment shell scripts to launch the corresponding Bash scripts (.sh files) from the folder GeoStack-Workshop / Scripts to check the completed assignment file after it has been saved and also run it if was correct!
2. Use the desktop shortcuts to the Solution shell scripts to run the provided working scripts in the Solution folder to see what the end result of an assignment should be.

Note: in the Scripts folder there are also 2 Python scripts to install the datasets in a separate MongoDB instance to provide an always working datastore for the solution shortcuts and they are run in installation shell script 5-dataset-import.sh when creating the Workshop VM.

0.10 Preparations Checklist – Workshop Assignments

At this point check if you have completed the following important steps:

1. You have checked and read the project's online resources:
 - You have read **all** about The GeoStack Project on the project's homepage on GitHub:
 - <https://the-geostack-project.github.io/>
 - You have also clicked the button 'View On GitHub' to find the 4 project repositories in case you need them and also read the Readme.md web pages of each repository: GeoStack-Manuals (button 'Download Manuals'), GeoStack-Workshop, GeoStack-Course and GeoStack-Project-Files (button 'Project Files').
 - <https://github.com/The-GeoStack-Project/>
 - You have clicked the web link to the projects YouTube Channel to watch the introduction video (6 minutes) and find the rest of the video tutorials:
 - <https://www.youtube.com/channel/UCiZEImhO8r-LMAWh-KQiH6g>
2. You have downloaded the two (2) GeoStack Manuals on your computer by:
 1. Downloading the ZIP file that holds the GitHub repository GeoStack-Manuals by clicking the blue button 'Download Manuals' at the top right of the project's homepage.
 2. Extracting the ZIP file that will give you the folder GeoStack-Manuals that holds the two course cookbooks 'Creating-the-Workshop-VM.pdf' and 'Creating-the-Course-VM.pdf'.
3. You have followed the cookbook 'Creating-the-Workshop-VM.pdf' for this workshop to:
 1. Install VirtualBox on your computer so it's running on your host operating system.
 2. Create the GeoStack Workshop Virtual Machine with Ubuntu Linux as the Guest Operating System.
 3. Make the folder GeoStack-Manuals on your host computer the Shared Folder in VirtualBox as an exchange folder for your Ubuntu Linux Virtual Machine to:
 1. Copy the 2 course cookbooks into your Ubuntu Home folder if you like.
 2. Exchange files between your host computer and your Virtual Machine!
 4. Clone the GitHub repository GeoStack-Workshop into your Home Folder on Ubuntu Linux as the main folder GeoStack-Workshop that holds all the files for this workshop.
 5. Download the datasets required for this workshop into the subfolder Workshop-Datasets of that GeoStack-Workshop folder.
 6. Find this document 'Cookbook-Workshop-Assignments.pdf' in that main GeoStack-Workshop folder.
4. You have read the introduction page 'Purpose of this document – How it works...' and this introduction chapter 0 'Workshop – Part 0 – Introduction' in this cookbook 'Workshop Assignments'.
5. You have build the extra skill levels, if necessary, which are required to successfully follow and complete this workshop, as per the advice given in section 0.3 'Building Skill Levels'!

If you have completed all of the steps on this checklist, you're ready to go!

1 Workshop - Part 1 – Data Processing: Exploring!

1.1 What are you going to do?

Exploring the Datasets! Some basic data processing steps to read and explore the raw dataset files and some simple visualization of the datasets are the first steps for a data scientist to learn about the data to work with.

A data scientist literally wants to see what the geospatial data is about and also literally find out where on earth this geospatial data is from.

The first two steps in Python programming to achieve the goals mentioned above are to:

1. Read, understand, analyze and transform the datasets to 'unlock' them from their raw file format. The ETL process to Extract data from flat files, Transform it and Load in a database!
2. Visualize the datasets by plotting geospatial data on very simple topographical maps by using Cartopy and also in a few graphs with Matplotlib to explore them just a little.

For Part 1 of the workshop there is a Jupyter Notebook available with the Python source code to run from your web browser. In this chapter you are going to use Python Pandas to read data from the files in order to manipulate and explore the datasets. Storing the data in a database will be done in the next chapter.

1.2 What do you need?

DATA ANALYSIS

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.

Cartopy is a Python package designed for geospatial data processing in order to produce maps and other geospatial data analyses.

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.

In computer programming, pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

If you are new to these important Python data science tools, read more here (it's time well spent!):

- <https://matplotlib.org/> and here: <https://en.wikipedia.org/wiki/Matplotlib>
- <https://scitools.org.uk/cartopy/docs/latest/> and here: https://ravernat.github.io/research_computing_2018/maps-with-cartopy.html
- <https://jupyter.org/> and here: https://en.wikipedia.org/wiki/Project_Jupyter and here:
- <https://en.wikipedia.org/wiki/IPython> (Jupyter uses the IPython shell instead of cpython!)
- <https://pandas.pydata.org/> and here: [https://en.wikipedia.org/wiki/Pandas_\(software\)](https://en.wikipedia.org/wiki/Pandas_(software))

1.3 The Workshop Datasets – Flight Paths & Car Routes

For this workshop you will use 6 datasets: 3 in CSV files and 3 in GPX files.

About the Datasets in the CSV files

While creating the Virtual Machine for the workshop you downloaded 3 datasets with the GPS track logs in the CSV file format containing flight paths of the spring and autumn migration routes across Europe of Swedish common crane birds (*Grus grus*) that were ringed with GPS trackers. The download instructions were found in the section 'Downloading the workshop datasets' of the cookbook 'Creating the Workshop VM' and the CSV files were saved in the folder Workshop-Datasets/CSV.

These crane datasets are published by the animal tracking website: <https://www.movebank.org>

The names assigned to these cranes for the purpose of this workshop are:

1. Agnetha (named after the Swedish pop-singer Agnetha Fältskog from ABBA.)
2. Frida (named after the Swedish pop-singer Anni-Frid (Frida) Lyngstad from ABBA)
3. Cajsa (named after the Swedish singer-songwriter Cajsa Stina Åkerström)

Keep in mind that GPS track logs of crane flight paths are similar to flight path datasets for other flying objects, like for the track logs of the ADS-B GPS tracks of airplanes or GPS tracks of drones.

For more information on CSV files:

- https://en.wikipedia.org/wiki/Comma-separated_values
- https://en.wikipedia.org/wiki/Delimiter-separated_values

About the Datasets in the GPX files

For the workshop you also downloaded 3 datasets with the GPS track logs in the GPX file format of a handheld GPS navigation device the user took along on some (car) trips in The Netherlands. The names assigned to these trails are:

1. Biesbosch (a car trip over a long dike in the north part of National Park De Biesbosch)
2. Zeeland Car 1 (a car trip to the Province of Zeeland for bird watching along the coast)
3. Zeeland Car 2 (a car trip to the same area in Zeeland for some overlap in route segments)

These GPX files were donated by another GitHub user for this project as Open Content datasets.

Keep in mind that the GPS tracks from these trail datasets are similar to those of other land and water vehicles like GPS tracks from cars or AIS GPS tracks from ships.

For more information on GPX files:

- https://en.wikipedia.org/wiki/GPX_Exchange_Format
- <https://www.topografix.com/gpx.asp>

Terminology – What is a GPS track, trail, trace, route or a GPX file?

- A GPS track is also often called a GPS trail or GPS trace. They are synonyms. A 'trace' is for instance the common term used on the OpenStreetMap website.
- A GPS track (log) is a GPX file with a set of sequential GPS coordinates of where a GPS (tracking) device has been since the track log started. A track file is a history file with a time series of data points with an object's GPS locations!
- A GPS route (plan) is a GPX file with a set of sequential GPS coordinates which a GPS (navigation) device still has to follow to reach its navigation end point (destination). Notice, 'track' and 'route' are often used as synonyms but strictly speaking they are not!
- A GPX file is the store and distribution file format for a GPS track (log) or route (planning).
- A GPS tracker device only writes track log GPX files and a GPS navigation device can both write track log files and read route files for navigation guidance.

How it works with the dataset files in the workshop

By following the cookbook 'Creating the Workshop VM' you downloaded the crane dataset CSV files in the folder: /GeoStack-Workshop/Geostack-Workshop-Content/Workshop-Datasets/CSV.

By running installation shell script 5-dataset-import.sh these CSV files were copied for Part 1 'Data processing' of the workshop to the JSON folder in:

- /GeoStack-Workshop/Geostack-Workshop-Content/Part-1-Data-processing/Datasets/
- When you do the assignments later you will learn how to read a crane dataset from the CSV file with Pandas into a Pandas tabular datastructure called a 'dataframe' and from there you will learn how to save the crane dataset into a JSON file in the Part 1 folder /Datasets/JSON.
- You will also learn that the Pandas function `pd.read_csv()` is used to read the CSV files and the Pandas function `pd.to_json()` is used to write the JSON files.

To make sure you can do the assignments of Part 2 without completing Part 1 there is also a /Datasets/JSON folder in the folder for Part 2 'Data storage' that already holds the JSON files:

- /GeoStack-Workshop/Geostack-Workshop-Content/Part-2-Data-storage/Datasets/
- The JSON folder that is here for Part 2 is filled with JSON files because the installation shell script 5-dataset-import.sh runs the Python script dataset-convert.py to do this CSV to JSON conversion automatically .
 - The dataset-convert.py script also does the GPX to JSON conversion for the car trail datasets to make sure these datasets are available too.

To summarize: in the workshop you will write the JSON files for the assignments in Part 1 in the /Datasets/JSON folder of Part 1 and an installation script makes sure the JSON files for the assignments in Part 2 are always available in the /Datasets/JSON folder of Part 2, whether or not you complete the assignments in Part 1 of the workshop.

To complete the 'fault tolerance' the installation shell script 5-dataset-import.sh also runs the Python script crane-datasets-import.py which creates the 'Crane_Database' in MongoDB to make sure the databases exist in case the assignments of Part 2 of the workshop are not completed.

- To do the same with the car trail datasets the Python script trail-datasets-import.py is run to create the database 'Trail_Database' in MongoDB.

IMPORTANT NOTICES: Database names used in MongoDB with and without an underscore ('_')!

- In Part 2 'Data storage' you will learn to pay careful attention to the database names 'Crane_Database_' and 'Trail_Database_' that in the assignments will have an extra underscore ('_') suffix in the database name 'at the end' that are programmed by you!
- In Part 2 'Data storage' you will also learn to pay careful attention to the database names 'Crane_Database' and 'Trail_Database' without that underscore suffix that were provided by these installation scripts crane-datasets-import.py and trail-datasets-import.py to make sure to provide you with working solution scripts and working databases in case you can not complete the assignments or when you want to learn from working examples!
- In the assignments of Part 1 and Part 2 you will learn how these Python scripts work because you will need to complete the source code to convert the data from CSV and GPX files into JSON files and from those JSON files into the databases in MongoDB!

1.4 Ringing Crane Birds with GPS trackers

The first step for a data scientist is to understand the dataset and therefore a little information about the real-world situation is required to give enough context to understand the datasets.

So let's have a look at cranes, why cranes are ringed and how that looks, what a GPS-tracker for crane birds is and what it looks like on a ringed crane and where to find information and datasets.

In Europe the common crane (*Grus grus*; https://en.wikipedia.org/wiki/Common_crane) is an endangered species for which the bird protection programs seem to work very well because after almost being extinct some 30 years ago the West-European population is now estimated to be a little upwards of 400,000 birds although protection is still required.

Cranes are migratory birds (<https://www.kraniche.de/en/crane-migration.html>) that breed up into the far north of Europe and winter in the south and even in Africa and the Middle-East. From Sweden to Spain, so to say. Little was known though about their life to be able to protect them.

Therefore ringing programs were started to build knowledge about their breeding biology and habitats, life cycle, feeding and resting areas along their migration routes and wintering habitats. Now it is known from this bird ringing research that cranes can live for more than 25 years and breeding couples stay together all their life and even have breeding success at those high ages!

The ringing research in Europe for the common crane is coordinated for the ringing programs of 7 countries by the iCora project (<https://www.icora.de/>), the Internet based Crane Observation Ring Archive that maintains a database with the data of around 3,800 ringed cranes.

The iCora project is run by the German NABU Crane Center in Groß Mohrdorf in the province of Mecklenburg-Vorpommern (<https://www.kraniche.de/en/exhibition.html>) and their website (<https://www.kraniche.de/en/>) is the starting point for information about cranes in Europe.

All the ringing groups work together in the European Crane Working Group (<https://www.kraniche.de/en/1064.html>) and there's also a very nice Facebook page about Cranes in Europe (<https://www.facebook.com/CraneInTheEurope/>).



Illustration 1: Crane family with ringed crane BlueBlueWhite-WhiteGreenBlue with juvenile (brown head).

The unique color combination consists of a mostly annually changing country code on the left leg (German in this case) and a combination for the individual bird on the right leg. There are 7 colors in use.

The GPS trackers have a rechargeable battery that is charged by a solar cell and by movements of the crane. Obviously the tracker has a GPS receiver to log the location of the crane and a GPRS transmitter to send text messages with log records.

There are also some other sensors in the trackers, like for the battery status, daylight level and movement speed of the crane etc. All this sensor information is send back in the text messages.

To save battery power and storage memory there is some clever software in the tracker to only log records if the crane moves enough from it's current position and to only send log records if there is enough battery power to make the connection to a cellular tower of the mobile telephone network.

The clever software is important because with technical failures aside, it would be nice if the tracker could keep functioning for the whole 25 year plus lifetime of a crane if it lives that long.

Around 25% of the cranes can reach at least 10 – 15 years of age and normal long lasting batteries only last about 4 – 6 years so the next two decades will tell us how long the technology of the new generation of trackers with rechargeable batteries will last.



Illustration 2: Two ringed cranes with a GPS tracker on their left leg at a crane dormitory in the early light at the break of dawn.

All cranes that are not breeding are sleeping together in large groups at their sleeping places, called crane dormitories. These sleeping places are always in the water to protect the cranes from land predators like foxes, wolves and wolverines.

While white-tailed eagles often try to catch a crane or a chick, they are hardly ever successful as the beaks of the (many) cranes pointing upwards is obviously a very effective defense. Even if the eagles try an attack in pairs or even in groups mostly the cranes come out unharmed.

In the summer it are mostly the young single birds that sleep together in groups of up to a few hundreds of birds but during the autumn and spring migration the numbers can easily run into the 10 or 20 thousands in one sleeping area which is spectacular to see and hear when the large flocks of up to 500 or 1,000 cranes fly in at dusk!

Some crane ringing groups have not only ringed their cranes with unique combinations of color rings but a several cranes were also ringed with a GPS tracker clicked to those color rings. Attaching GPS trackers is done to get a much more detailed picture of the biology of cranes and their migration patterns. This complements the two standard methods of both the annual migration counts in spring and autumn and the reported field observations of ringed cranes by bird watchers.

Notice that GPS tracking of cranes is very important for scientific research because with even less than 4,000 registered cranes tracked in iCora of which not all of them are still alive and an estimated population of 400,000 that also less than 1% of the crane population is ringed.

For a few of these GPS ringed cranes the ringing teams have published their GPS tracks on the animal tracking website Movebank.org (<https://www.movebank.org>).

To see the publicly accessible datasets of these GPS tracks on Movebank:

1. Click the menu option Data / Map at the top right of the home page.
2. Then at the left side type 'crane' in the search field, select GPS from the pull down menu 'All sensor types' and check the checkbox 'Only studies where I can see data'.
3. Click the Search button to get a list of ringing projects under 'Search results' of the public GPS track logs of ringed cranes.
4. Click the green plus ('+') sign to the left of a ringing projects to see the list of individual cranes.
Take note of the number of GPS data points in the dataset (between brackets).
5. Finally click the search icon (magnifying glass) at the right of the crane dataset to get a digital topographical map of Europe with the migration routes and locations plotted.
Note: this may take a few minutes before the blue lines and dots are getting plotted on the map if there are many thousands of data points in the GPS track log dataset.



Illustration 3: Dancing cranes in an old corn field in Germany with a Finnish ringed crane YellowYellowYellow-RedBlueWhite.

Many people come to see the annual autumn bird migration of the cranes as a true Wonder of Nature in the fall! Just before Easter it's spectacular too, to see the returning cranes dance with happy high jumps and loud calls as we then consider them to be our 'Messengers of Spring'!

1.5 About the GPS track logs in CSV files

Now it's time to take a closer look at the datasets in the CSV files in the folder Workshop-Datasets/CSV. The folder was installed by cloning the GitHub repository when creating the Workshop VM.



Drawing 2: The 3 cranes were ringed in the area around the village of Grimsås in Sweden so the first (= oldest) data points in the CSV files will have their GPS coordinates there.

Data fields in the CSV files, like GPS coordinates (long + alt), altitude (height), timestamp.

- **IMPORTANT NOTE:** this example shows the data fields from a Swedish GPS tracker!
- Remember, GPS trackers may differ in fields between brands and versions!
For example the German trackers have more sensor fields, also their sequence of fields differs and they use another measure method for height (altitude!).
- The Python data import scripts in The GeoStack Project work for all crane datasets that have the fields for longitude, latitude, altitude (height) and timestamp.

Column	Type	Desc.
event-id	int	The ID of the transmission (Unique).
visible	bool	Is the tracker still visible or not?
timestamp	datetime	The DTG (Date time group) of transmission.
location-long	float	The Longitude location in coordinates.
location-lat	float	The Latitude location in coordinates.
ground-speed	float	Speed of the Crane measured from ground level.
heading	float	Direction the bird is facing (In degrees)
height-above-ellipsoid	float	Height of the Crane.
individual-taxon-canonical-name	string	The type of bird, in latin.
tag-local-identifier	int	The unique ID of the tracker.
individual-local-identifier	int	The unique ID of the crane.
study-name	string	The name of the study that is done.

Drawing 3: The data fields in the track log records are on one (1) line in the CSV file. See the CSV file as a table or spreadsheet in which each data field of the track log record is a column and each track log record is a row. All the field names are on the 1st line in the CSV file.

1.6 Explore the CSV files

First you go explore the 3 CSV files to get a feel for the data.

1. Start the Workshop VM and go to the folder GeoStack-Workshop/Workshop-Datasets/CSV.
 - You downloaded and named the dataset files here when you created the Workshop VM.
2. Open the CSV file for crane Agnetha by clicking it. Remember you put the personal names in the file names for easy reference when renaming the dataset files after their download.
 - Note: if Atom doesn't open because it is not the default application for CSV files then close the application that opened and set Atom as the default editor for CSV files. You can also use 'Right mouse click' and then choose 'Open with...' and select Atom!
 - Tip: in any programming editor, it is very convenient to have line numbering turned on. If in Atom line numbers are not shown in the left margin of the file edit window, then go to the menu Edit / Preferences / Editor and scroll down to check 'Show Line Numbers'.

After you have opened a CSV file in your Atom programming editor you will find the names of all the data fields on the first line of the file.

Then from top to bottom you will find all the GPS track log records from the oldest record (= 1st log line = 2nd line in the file) to the most recent record (last line of the file).

Now go plot the 1st GPS coordinate pair for all 3 cranes on an online digital topographical map to see where the crane's GPS was turned on for the first time after ringing the bird.

1. In the first line of the file you can see that location-long (longitude) and location-lat (latitude) are the 4th and 5th fields respectively.
2. Now for crane Agnetha copy the GPS location from the 1st log line in the CSV file which will be the GPS lon/lat pair: 13.27158,57.390537
3. Open your web browser (e.g. Firefox) and go to <https://www.openstreetmap.org>
4. Paste the GPS coordinates in the search field on the left side and DO NOT click on the blue 'Go' button just yet!
5. Cut and paste the coordinates in the reverse order, exactly like this: 57.390537,13.27158
The online web map services, like OSM, Google Maps or Bing Maps all use the lat/lon sequence instead of the lon/lat sequence in the CSV file!
6. Now click the blue 'Go' button which will bring you to a map view with this web link:
<https://www.openstreetmap.org/search?query=57.390537%2C13.27158#map=8/57.391/13.272>
7. Finally hover over the blue web link with the GPS coordinates on the left side to see a red marker appear on the map 'on mouse over' of the weblink of where the GPS location is.
 - Note: unfortunately OSM uses 'on mouse over', instead of putting the location marker permanently on the map, which is quite annoying!
8. Now zoom in by clicking on the white plus button sign in the gray menu on the top right until you get to zoom level 14 to see that the red location marker is situated in the forest hamlet of Hunnabo in Sweden.
 - Notice the 'map=8' part in the web link which indicates the zoom level to view the map. Every click on the plus sign will increase that number and the higher the number, the more you zoom in. OSM has 19 zoom levels to zoom in to see more detailed maps.
 - Tip: when you zoom in, check regularly with 'on mouse over' on the location web link to have the red location marker appear to keep the marker in view. If not center the map again by dragging it with the mouse until the marker is centered again.
 - Tip: because this missing default marker preview on OSM is not so user friendly, you can try Google Maps or Bing Maps for a much better orientation and zoom experience!
9. Finally do the same for the first GPS location in the CSV files for the cranes Frida (just north of Grimsås) and Cajsja (between the villages of Moen and Erikstorp).
 - Tip: Scroll down to the last line of the file: number of lines – 1 = number of records!

1.7 Assignment – Prepare Visual Exploration of the Datasets

The goal of this section is to learn how to start the Jupyter Notebook containing the assignments, explain how this assignment works in the next sections and explain how the datasets are handled in the workshop. Notice you are starting with a longer assignment.

The assignment steps to just get this assignment started with Jupyter Lab are as follows:

1. Click on the desktop shortcut called: “jupyter-lab”, as shown in illustration 1.
2. This opens the JupyterLab application in the Launcher screen (tab ‘Launcher’) in which you click the Jupyter Notebook listed on the left under ‘Name’, as shown in illustration 2.



Illustration 1:
Desktop Shortcut

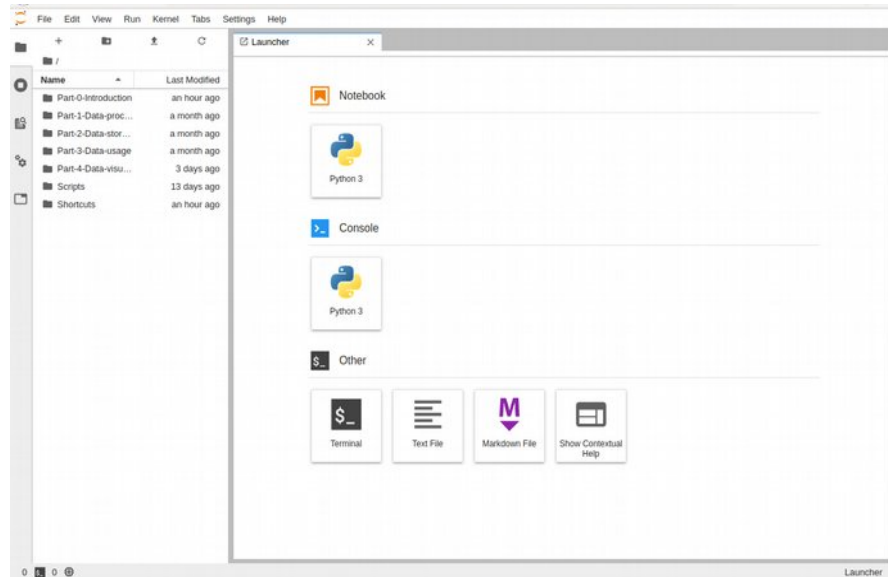


Illustration 2: Launcher screen of the JupyterLab application

In the folder ‘Part-1-Data-processing/Notebooks’ you will find 2 sub-folders:

1. The sub-folder Assignment
 - Here is the notebook file Assignment-Data-processing.ipynb with the assignments.
 - A file with the ‘.ipynb’ file extension is a Python source code notebook for the IPython shell that JupyterLab runs! Read more about IPython here: <https://ipython.org/>
2. The sub-folder Solution
 - Here you will find a completed Jupyter Notebook with the solutions.
 - In case you get stuck you can always take a look at this notebook. No sneak peeks!

Assignment Instruction --> See ‘Assignment - Simple Map Plotting with Python in Cartopy’!

1. Assignment Goals --> 3 Exploration Steps

1. Explore the data by finding the amount of GPS data points in the track logs and the column names that represent the field names in each track log record.
 2. Visualize the track logs with Matplotlib and Cartopy (for simple ‘topo’ maps!).
 3. Convert and store the data in the JSON file format in the /Datasets/JSON folder to get standard data which is required in the next chapter to store as JSON documents in MongoDB.
2. Assignment instruction --> Split assignment into two equal sets of assignments!
 - In the notebook file there are 8 assignments that need code completion:
 - Assignments 1 – 4 are related to processing and visualizing Crane datasets.
 - Assignments 5 – 8 are related to processing and visualizing Car route datasets.
 - Each piece of Python code is called a ‘code cell’ which you need to run in sequence!
 - **To run code (!):** select the code cell first and click ‘Run’ at the top of the screen!
 - Everywhere you see the text: “#TODO” in the assignments, you will have to provide the code completion solution to the assignment after carefully reading the detailed inline comments to help you to learn to program!

Split Assignments

The assignments are split into two equal groups for the datasets of the crane flight paths from CSV files and the car routes from GPX files.

The instructions can be found in the section 'Assignment - Simple Map Plotting with Python in Cartopy' after explaining the GPX basics first in the next section 'About the GPS track logs in GPX files'!

Drawing the Topographical Map in Cartopy --> The source code explained!

Cartopy is a mapping library that extends the matplotlib functionality. It's good to have a closer look at the programming steps in the source code of the assignment notebook on how to draw data (points) as an overlay on a topographical map which itself also consists of one or more layers.

Let's take a look at the cookbook recipe in the source code which consists of 3 main coding steps:

1. Creating the Canvas: Matplotlib needs a figure variable and Cartopy a map variable as they each need a 'blank canvas to draw on'. Later you 'plot' that variable to display the contents.
2. Drawing the Topographical Map: then the Cartopy canvas is used to draw a base map with several layers, like the country borders and rivers that give an orientation.
 - For instance adding an extra overlay with the location and names of the capitols of each country would give even more map orientation but this is not part of the assignment.
3. Plotting the Data: finally with Matplotlib the data points of the crane locations that are in the Pandas dataframe are created and plotted as a scatter plot overlay in 3 steps:
 1. The real-world GPS coordinates of each data point have to be converted to the coordinate system used in the scatter plot to make the mapping.
 - This requires handling map projections and coordinate mapping which is explained in the source code to learn some basics on how to work with Cartopy!
 - Notice the sequence for the GPS coordinates is: North Latitude, East Longitude (Lat, Lon) in the Coordinate Reference System (CRS) that is used in Cartopy.
 2. The real-world GPS coordinates are scaled back to fit the scatter plot window.
 3. Finally to display the scatter plot window on top as the last overlay requires that it has to be aligned exactly with the Cartopy base map window below that.

Scope limitation

The processing of the 3 GPX files with GPS track logs of car trips to load them in MongoDB is not an assignment in (the next chapter of) this workshop to keep things simple.

Visualizing data on a topographical map in Cartopy is the focus in this assignment!

- Still, these 3 datasets are available in the workshop with the objective for you to practice getting them in MongoDB on your own after this assignment with the crane datasets.
 - Therefore this assignment finishes by letting you store the car route datasets as JSON files as well if you want to try on your own in the next chapter.
 - Please do try in that next chapter, because the programming is pretty much the same!
 - What if you don't? Don't worry, it's been taken care of!
 - To make sure the GPX datasets are available at the end of the workshop to use them in the simplified 2D Map Viewer web application, even if you don't want to practice the ETL process (Extract, Transform and Load) here for the GPS track logs of the car trips, this ETL process is already automated by the installation script 5-dataset-import.sh.
 - This ETL script Extracts the GPS log records from the raw GPX files and the records are Transformed to JSON objects, for which a data model is created in a separate instance of a MongoDB datastore after which the JSON objects are Loaded (stored) as JSON documents in MongoDB.
- In this way all datasets are always available in this extra MongoDB instance!

1.8 About the GPS track logs in GPX files

1.8.1 GPX files explained – Some Workshop Basics

As stated in the previous section there are 3 GPS track logs of car trips supplied in the folder Workshop-Datasets/GPX for you to experiment with in the next section to visualize on a simple digital topographical map.

Therefore a little background about those GPX files is given in this section first.

The datasource of these GPX files is a GPS navigation system and if you open the GPX file in the Atom programming editor you can see in the top lines which brand and model the GPS device is.

The following 3 GPS track logs from car trips in The Netherlands are supplied:

1. Car trail in National Park The Biesbosch. Just for a simple visualization test.
2. Two Car trails for bird spotting in the Dutch province Zeeland.

These trails were deliberately chosen because they partially overlap so you can experiment with both in the next section and later in the 2D Map Viewer web application with different ways of visualizing the location points and lines between the location points for a better view.

For more information on GPX files and the GPS track logs, see section 1.3 'The Workshop Datasets' for web links to follow up on and the dataset descriptions.

The basics steps of GPS navigation and tracking are summarized here to get to a GPX file:

- A constellation of GPS satellites sends the GPS signals as short radio messages.
- A GPS receiver stores multiple messages from different satellites for location calculation.
- A complex software algorithm then calculates the precise location of the device.
- The device logs the calculated location of the GPS device as an XML record in a GPX file.

GPX files are designed for storing XML formatted track log records of calculated GPS locations with the following data fields: location (longitude + latitude), height and a date/time group (DTG).

- In the workshop we use the terms GPS log record or GPS signal or GPS signal message depending a little on the context when we mean a GPS track log record.

Column	Type	Desc.
lon	float64	The Longitude coordinates
lat	float64	The Latitude coordinates
alt	float64	The altitude coordinates measured in Height above mean sealevel
time	Datetime	The DTG (date time group) of transmission

Drawing 4: The data fields in the GPS track log records are one XML record in the GPX file.

Note: also remember that in GPX files the sequence of the GPS location coordinates is NOT in the human sequence of North Latitude, East Longitude ('lat,lon') but the field sequence in the GPX file is in the reverse sequence 'lon,lat'!

- On the well known online web map services like OSM, Google Maps and Bing Maps you have to input the GPS coordinates in the normal 'human readable' sequence of the 'lat,lon' format in their search fields and... don't forget that comma in between when you do!

1.8.2 Challenge - Exploring GPX files Online and on the Desktop

Challenge: take an hour or two to explore some web services and read about application options to quickly visualize the track log in a GPX file ***if... the data in the GPX file is not confidential!***

- **Privacy Warning:** think about your privacy if you use your own GPX files with an online service because you upload them to an online third party web service without any control!
- **OSM Warning: DO NOT** use the OpenStreetMap 'upload and edit' options to explore GPX files because these options are only intended to contribute Open Content to the OSM project! So it's no surprise, that your GPX files will be publicly visible on OSM once uploaded!

Exploring the web for quick visual GPX file previews.

- Source: a list of OpenStreetMap based web services is here:
 - https://wiki.openstreetmap.org/wiki/Track_drawing_websites
- Services to explore if your GPX files are not confidential:
 - <https://opentopomap.org>
 - Easy and fast. The default less appealing map style of OTM with the track only in red can be easily changed to the map style of OSM by clicking the white button at the top right of the screen.
 - <http://share.mapbbcode.org/>
 - Super fast GPX viewer demo website for the Leaflet geospatial framework.
 - Click on the Import button on the left to select and import a GPX file.
 - More info here: <http://mapbbcode.org/> (official website) and here: <https://github.com/MapBBCode/mapbbcode> (Over 5 years old code but it still works fine!)
 - <https://umap.openstreetmap.de/de/>
 - This German version for umap is much faster than the French version.
 - Click the green button 'Create a map' at the top right.
 - Then on the left click the white circle icon with the black Up(load) arrow.
 - Select the GPX file and click Import to display the GPS track.

It is also possible to run a local desktop application to visualize the track log in the GPX file which will of course take more time to learn the application than trying an online web service.

- <https://www.gpxsee.org/> - GPXSee is probably the easiest cross-platform GPX viewer app!
 - More on GPXSee on <https://github.com/tumic0/GPXSee-maps> with in the repository also the Style Sheets for different topographical map styles.
- <https://qgis.org/en/site/> - QGIS is a very good cross-platform GIS desktop application you must learn sometime if you are into geospatial stuff but because of the many functions it has a learning curve and therefore it is not the first choice for a quick GPX file preview.
 - More information on QGIS is here: <https://en.wikipedia.org/wiki/QGIS> and here: <https://wiki.openstreetmap.org/wiki/QGIS>
- Other Open Source options: GpsMaster (<https://wiki.openstreetmap.org/wiki/GpsMaster>) and GpsPrune (<https://activityworkshop.net/software/gpsprune/>).

Then there are two other options left for a quick GPX file preview:

1. Copy a fairly simple piece of script to run locally for a nice experiment after this workshop, like this: https://wiki.openstreetmap.org/wiki/Openlayers_Track_example or this: <https://blog.aaronlenoir.com/2019/09/25/draw-gps-track-on-openstreetmap/> (Leaflet).
2. Get the Android mobile app OsmAnd which has GPX file import on a smartphone or tablet.
 - Get it from F-Droid <https://f-droid.org/en/packages/net.osmand.plus/> or Google Play.
 - Tip: the F-Droid version has unlimited free map downloads; the Play version limits to only 7 free map downloads after which you have to pay a fee for additional maps.

1.9 Assignment – Simple Map Plotting in Python with Cartopy

There are 8 assignments in the Jupyter Notebook 'Assignment-Data-processing.ipynb' about visually exploring the 6 GPS track log datasets you have of 3 Swedish cranes (Frida, Agnetha and Cajsja) and there are 3 car trip datasets in The Netherlands to visualize too!

- These datasets are stored in memory in the tabular datastructure of a Pandas Dataframe as mentioned above in the section 'Assignment – Prepare Visual Exploration of the Datasets' and then to store them as JSON files.
- Both the CSV and GPX formatted data are converted to JSON which acts as an intermediate standard data format because the goal is to store each data record as a JSON document in MongoDB in the next chapter. So this is a part of the Transform step in the ETL process.

To visually explore the datasets you are going to use the Cartopy geospatial mapping package from the Python scripts in the Jupyter Notebook. You will learn how to plot the topographical base map in Cartopy with a scatterplot from Matplotlib as a map overlay to plot the data points.

- As Cartopy's home page says (<https://github.com/SciTools/cartopy>): *"Cartopy is a Python package designed to make drawing maps for data analysis and visualization easy."*
- The Cartopy documentation can be found here: <https://scitools.org.uk/cartopy/docs/latest/>

In this section you are going to learn how to get a visual impression of all of these 6 datasets as a quick way of visual exploration of geospatial datasets on very simple topographical base maps.

- The next sub-section is about plotting the crane datasets in 4 assignments and explains more on how Cartopy works.
- The following sub-section is about plotting the car datasets in 4 assignments.
- It's all about getting the 'geospatial picture' and seeking some first 'situational awareness'!

1.9.1 Assignment – Plotting the Crane datasets on a Topographical Map

Since cranes are migrating birds these 3 cranes fly from their birth and breeding grounds in Sweden to their winter quarters in Germany, in the south of France and even all the way to the province of Extremadura in Spain.

- Because of this you may expect a more or less graphical messy plot of multiple flight paths since there are a few hundred thousands of data points in total, spread over several years.
- Although it is not so obvious at first, the simple fact is that you have a Big Data problem on your hands! Therefore you want to have a simple way to explore the datasets first in a visual way by plotting them in different colors on a simple topographical map.

There are two goals for our crane datasets when plotting them in Cartopy:

1. Learn to draw simple digital topographical maps with simple contours of the country borders which are good enough to get a first 'spatial awareness' impression of what is going on.
2. Learn to plot the GPS locations in different colors to visually separate the datasets.

Assignment instruction 1: Assignment 5 in the Notebook has 3 code completion learning steps:

1. Follow the Jupyter Notebook and make a simple visualization of the Crane Flight paths using Cartopy. Remember, this is a geospatial dataset exploration to get a first impression!
2. Look at the source code too, to see how the Pandas dataframe query is formulated and how different colors are used for each dataset as shown on the map projection below.
 - By creating a simple projection, of the datasets, using Cartopy, you will have a quick first glance of the final result later when visualizing the data in the front-end web application of the simplified 2D Map viewer.
 - Remember: lots and lots of data points to plot so calculating will take some CPU time!
3. Store the crane flight path data from the Pandas dataframe in memory in a JSON file for use with MongoDB in the next chapter.

Assignment instruction 2: when you run the Python code for Cartopy in the Jupyter Notebook, notice that when Cartopy is used for the first time in the source code, of course it has to draw the topographical base map first before a scatter plot with data points can be displayed on that map!

This is how running Cartopy for the first time works:

1. Cartopy draws a base map by downloading the datasets for the map layers, called 'feature layers' that are coded in Python in the assignment notebook as shown in the image below.
 - Notice the '10m' resolution parameter that specifies a map scale of 1:10,000,000!

```

the maximum deviation the coastal line can have. The higher the value, the higher the deviation of the correct
location of the lines.
'''
cartopyMapCranes.coastlines(resolution='10m')

'''
Below we add the landsurface to the Cartopy map.
We give the landsurface (face) the color white.
We give the edges of the landsurface (edge) the color black.
'''
cartopyMapCranes.add_feature(cartopy.feature.LAND.with_scale('10m'), edgecolor='black', facecolor = "white")

'''
Below we add the lakes to the cartopy map.
We give the edges of the lake the color black.
'''
cartopyMapCranes.add_feature(cfeature.LAKES.with_scale('10m'), edgecolor = 'black')

'''
Below we add the sea surface to the Cartopy Map
'''
cartopyMapCranes.add_feature(cfeature.OCEAN)

'''
Below we add the rivers to the Cartopy Map
'''
cartopyMapCranes.add_feature(cfeature.RIVERS.with_scale('10m'))

'''
Below we add the borders to the Cartopy Map
'''
cartopyMapCranes.add_feature(cfeature.BORDERS.with_scale('10m'))

'''
Now we want to create a dataplot which shows the flightpaths of the Cranes. We plot this data on the CartopyMap
which we created above, using Mathplotlib. The plot we are going to create is called a scatter plot. This is one
of the many types of Mathplotlib plots.

```

2. Cartopy shows download warnings for the ZIP file of each feature layer to inform you of the required 'first time' downloads as shown in the image below.
 - This will typically take a few seconds per file totaling to something like 10 – 30 seconds, of course depending on your internet speed.

```

/home/geostack/.local/lib/python3.6/site-packages/cartopy/io/__init__.py:241: DownloadWarning: Downloadin
g: https://naciscdn.org/naturalearth/10m/physical/ne_10m_land.zip
warnings.warn('Downloading: {}'.format(url), DownloadWarning)
/home/geostack/.local/lib/python3.6/site-packages/cartopy/io/__init__.py:241: DownloadWarning: Downloadin
g: https://naciscdn.org/naturalearth/50m/physical/ne_50m_ocean.zip
warnings.warn('Downloading: {}'.format(url), DownloadWarning)
/home/geostack/.local/lib/python3.6/site-packages/cartopy/io/__init__.py:241: DownloadWarning: Downloadin
g: https://naciscdn.org/naturalearth/10m/physical/ne_10m_coastline.zip
warnings.warn('Downloading: {}'.format(url), DownloadWarning)
/home/geostack/.local/lib/python3.6/site-packages/cartopy/io/__init__.py:241: DownloadWarning: Downloadin
g: https://naciscdn.org/naturalearth/10m/physical/ne_10m_lakes.zip
warnings.warn('Downloading: {}'.format(url), DownloadWarning)
/home/geostack/.local/lib/python3.6/site-packages/cartopy/io/__init__.py:241: DownloadWarning: Downloadin
g: https://naciscdn.org/naturalearth/10m/physical/ne_10m_rivers_lake_centerlines.zip
warnings.warn('Downloading: {}'.format(url), DownloadWarning)
/home/geostack/.local/lib/python3.6/site-packages/cartopy/io/__init__.py:241: DownloadWarning: Downloadin
g: https://naciscdn.org/naturalearth/10m/cultural/ne_10m_admin_0_boundary_lines_land.zip
warnings.warn('Downloading: {}'.format(url), DownloadWarning)

```

Assignment instruction 3: now make the following 3 steps to learn the Cartopy basics: 1) read the 'cookbook recipe' below about how Cartopy works with Map Scales and Map Projections, 2) then read the comments in the Jupyter Notebook about how the source code for Cartopy use actually works and 3) finally program the working code for the assignments!

About Cartopy Map Scales and the Workshop Assignments!

To create the base map for Cartopy there are 2 main options:

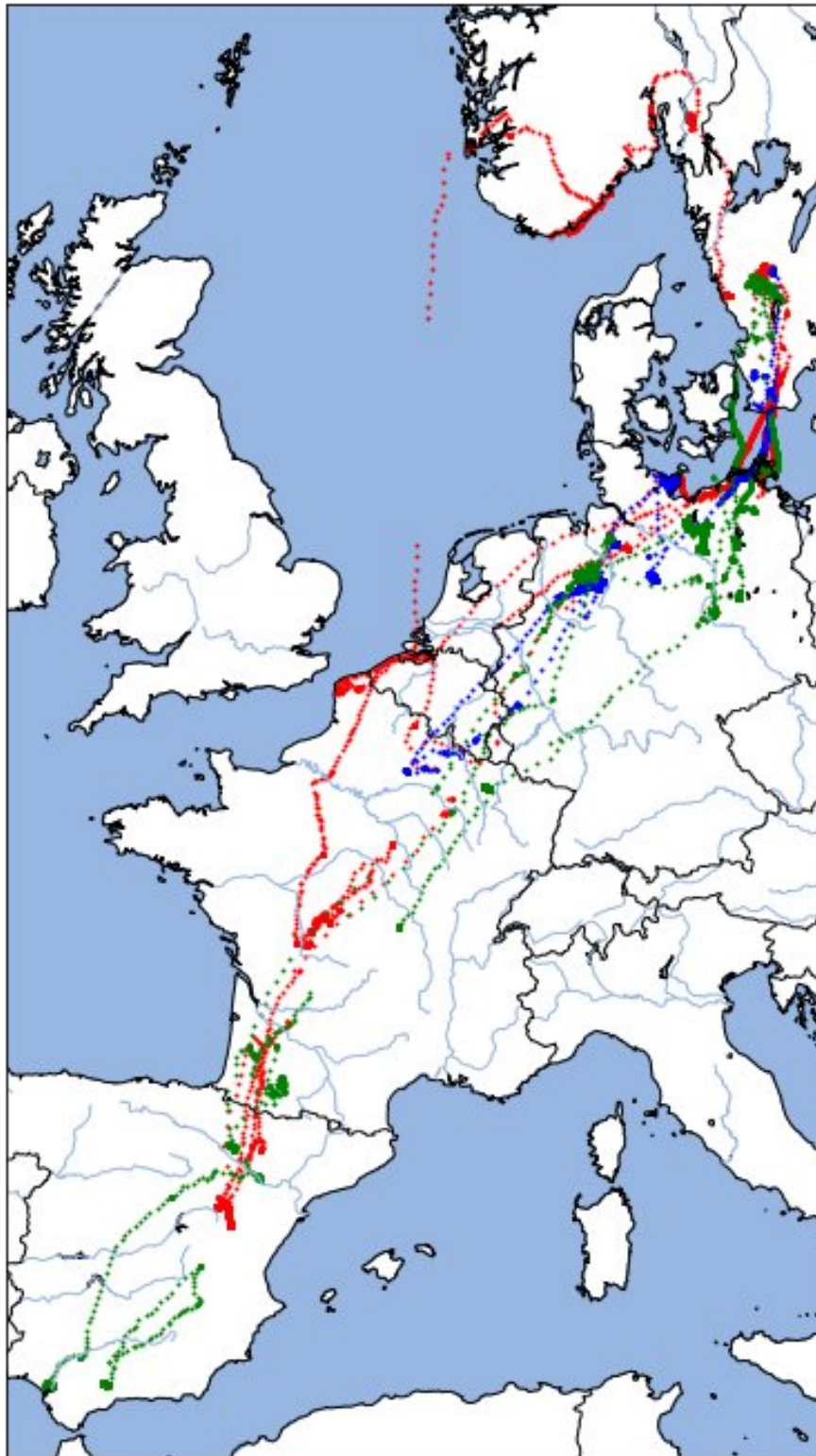
1. Use simple maps with very high map scales, like 1:10,000,000 or even 1:100,000,000 that correspond with low zoom levels on an online map service like OpenStreetMap.org for views at world-map, continent and country border level.
 1. For simple maps Cartopy uses the map datasets from Natural Earth that are automatically used for these 'feature layers' by the `.add_feature()` method in the code that is automatically available after the `cartopy.feature` import statement in Python.
 1. Read more about Natural Earth datasets here: <https://www.naturalearthdata.com/> and here: https://scitools.org.uk/cartopy/docs/latest/matplotlib/feature_interface.html
 2. The topographical datasets for these map feature layers in Cartopy for continent and country borders, rivers, lakes etc. are only available for 3 high map scales from Natural Earth: 1:10,000,000 (10M), 1:50M and 1:110M which makes them 'very low detail' and only usable to plot geospatial datasets that span multiple countries!
 3. The good thing is all the feature layers are aligned exactly at these 3 map scales to make these simple maps look good!
 2. Because the migration flight paths in the crane datasets span multiple countries and for some cranes that even go from Sweden to France or Spain these simple base maps are ideal to get a fast visual view of the datasets in just a few lines of Python code!
 3. Since crane 'Frida' only has data points in Sweden and Germany this will show alignment artifacts in Cartopy, like coast lines and the ocean waterline that don't align properly when the map is enlarged and when displaying the ocean layer it colors the land also blue!
 4. For the car datasets that only cover a distance of about 100 kilometer over 3 provinces in the Netherlands the alignment artifacts of water and land borders make the maps look 'bad to ugly' but for a quick visualization it works but not for publication purposes!
2. Use detailed maps with lower map scales, like 1:100,000 or even 1:10,000 that correspond with high zoom levels on an online map service like OpenStreetMap.org for views at 'street and building' level.
 1. To create detailed maps for countries, provinces, cities and more detailed (street) levels Cartopy has the standard option to use map tiles for topographical maps from Web Map Services (WMS), like OpenStreetMap.org.
 2. The advice is to first finish this workshop before diving into how this works with Cartopy because you will learn about map tiles and WMS in this workshop.

About Cartopy Map Projections and the Workshop Assignments!

Cartopy is strong in providing many topographical map projections for many documentation purposes. In the workshop assignments the 2 most used projections are introduced to learn the differences between the normal perpendicular helicopter view for (paper) topographical maps in the Mercator projection and maps for publications that are often in the Plate carrée projection.

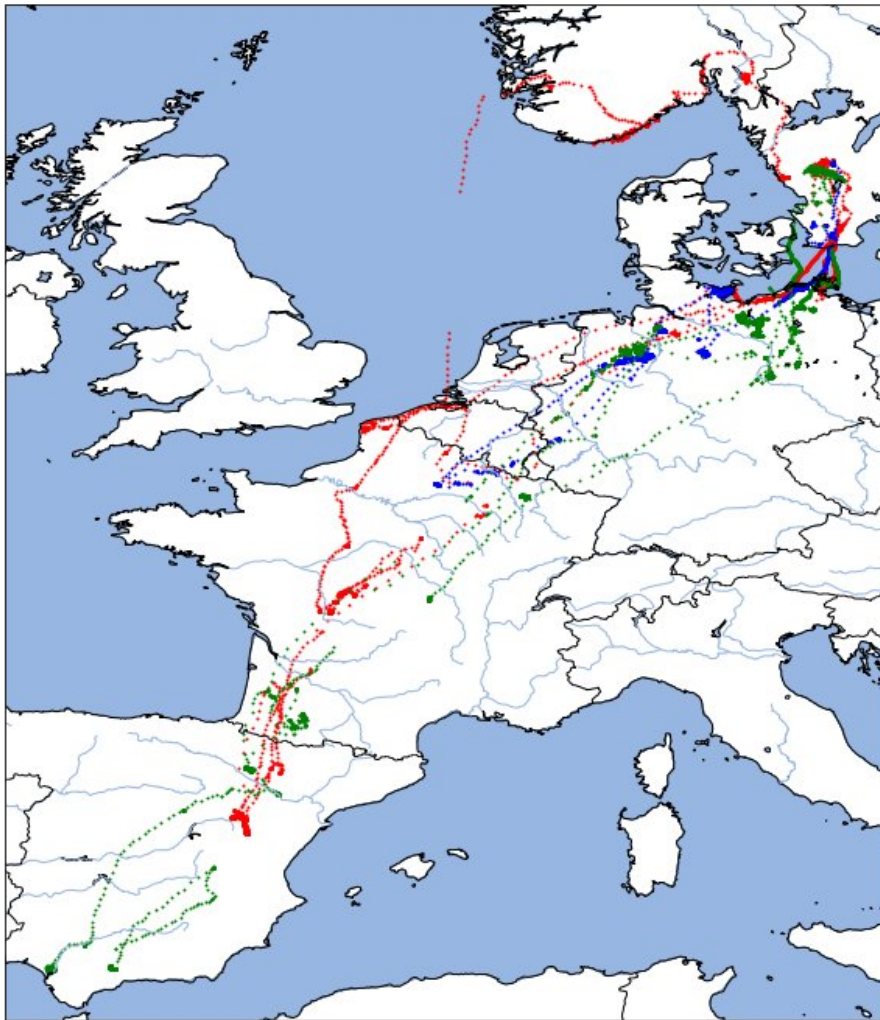
Now have a look at the end result of the scatter plot with the data points of the 3 crane flight paths and the 3 car routes on the Cartopy base map that you should get on the next pages!

- Notice the first image shows the topographical map in the Mercator projection and the second image shows the topographical map in the Plate carrée projection!
- The Mercator projection is the standard map projection for navigation we are familiar with from paper maps. Read more here: https://en.wikipedia.org/wiki/Mercator_projection
- The Plate carrée projection is a map projection with equal distances between longitude meridians and the circles of latitudes, making it very suitable for thematic projections like for book illustrations but it is unsuitable for navigation because of the space distortion.
- Also read more here: https://en.wikipedia.org/wiki/List_of_map_projections and about the Plate carrée projection here: https://en.wikipedia.org/wiki/Equirectangular_projection



Drawing 5: Crane migration flight paths for 3 Swedish cranes in red, green and blue in the Mercator map projection!

Example of a Visual Dataset Exploration: notice the red dots over the North Sea! A very rare flight path overseas but not an anomaly! Normally migratory birds like to see the land at the other side! The crane in this case probably caught a jet stream of wind at high altitude to get to Norway fast without burning unnecessary energy when flying in or even against the lower slower winds. The gap in the dots is probably caused by missing cell towers far out at sea and the dots we do see are probably from GPRS messages caught by receivers on windmill parks, oil platforms etc.



Drawing 6: Crane migration flight paths for 3 Swedish cranes in red, green and blue in the Plate carrée map projection!

Notice the distortion in the map but this would be fine for publication purposes!

For instance if the map was used to illustrate an explanation of where the high altitude 'migration highways' are located for the cranes passing over the mountain range of the Pyrenees between France and Spain which is almost 500 kilometers long and rises up to 3,400 meters!

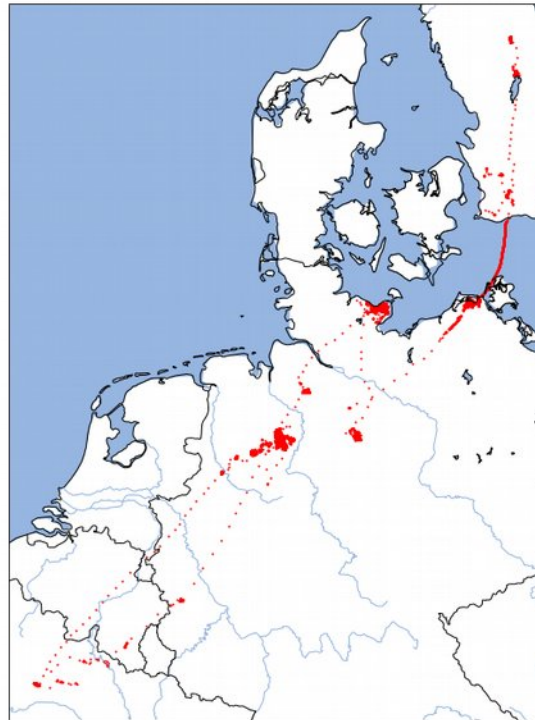
- In the Himalayas the cranes even cross high altitude passages of up to 7 kilometers!

Once across the mountain range many cranes rest often rest right at the south side of the mountains in the lakes of Gallocanta on the high and freezing cold mountain plains at 1,000 meters before going further south to their warm winter quarters in the province of Extremadura.

Even these simple maps give a first indication of the migration paths the birds follow and where the hot spots are for their resting, sleeping and feeding grounds.

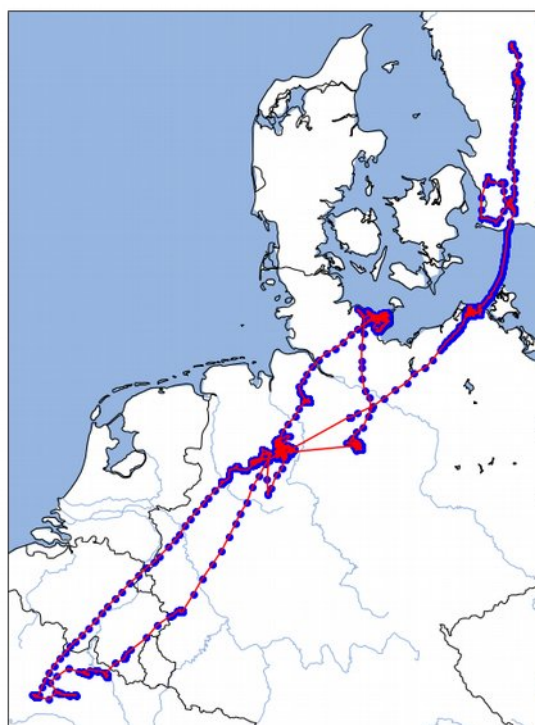
The dataset for crane 'Frida' shows the migration route from Sweden to Germany and France. The example source code in the Jupyter Notebook shows that it is still possible for a smaller geographical area to get a simple map with a fairly good quality as shown in the image below, although land borders and water lines do not align exactly if you would enlarge the images.

- To do this you have to use the '50m' resolution parameter to display the ocean feature layer. When it's '10m' or when omitted the land will color blue too!



The data point overlay in the map image above is made with the scatter plot function which is the preferred way to visualize data points and the basic plot function was used for the image below.

- Notice the difference in color use in the image below for the data points and the connecting red lines between them. Of course this works in the scatter plot function too! In comparison to the small red points in the map view above this highlights the flight paths and the red lines make the hot spots of feeding and resting grounds stand out as red blobs.



1.9.2 Assignment – Plotting the Car datasets on a Topographical Map

As stated before, the 3 supplied GPX files with GPS track logs from car trips are for self-study purposes and were already stored for you to be available for use with Cartopy.

For Assignment 6 follow the Jupyter Notebook too for the next 3 steps:

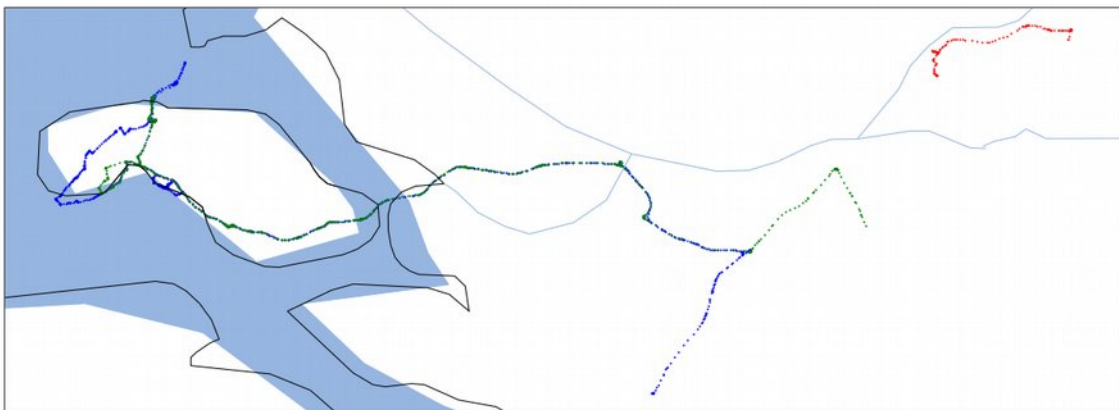
1. Plot the 3 GPX datasets from the Pandas dataframe on a Cartopy map that covers the south-west of The Netherlands.
2. Have a look at the source code for the Pandas dataframe query and the Cartopy code to get the different colors for each of the datasets.
3. Store the car route data from the Pandas dataframe in memory in a JSON file for use with MongoDB in the next chapter.

Notice in red the car trip over a long dike in the Noordwaard which is on the north side of National Park De Biesbosch in the Dutch province of Noord-Brabant.

- Also notice in green and blue the car trips from the province of Noord-Brabant to the province of Zeeland from two bird watching trips to the coast and the Brouwersdam.

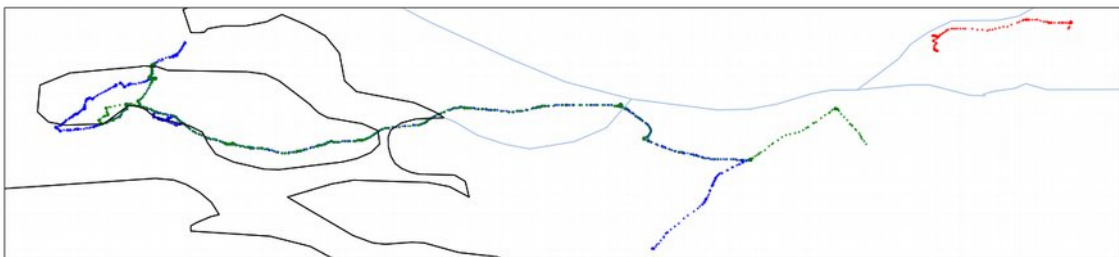
To get a more 'natural' navigation map projection the first code example for the car routes uses a Mercator projection that shows the normal 'helicopter' map view 'from right above' as shown below.

- Notice the very visible non-alignment of land borders and water lines on the '10m' maps!
- Remember, the Natural Earth maps are not meant to be used like this! It's just to illustrate!



Drawing 7: 3 GPS track logs from GPX files visualized from MongoDB in the Mercator projection. Red: Biesbosch, Green and Blue: 2 deliberately overlapping car trips to Zeeland to experiment with different ways of plotting.

The second code example in the Plate carrée projection shows this is not very usable for publication purposes because the map gets distorted too much for situational awareness!



Drawing 8: The same 3 GPS track logs of car routes but now in the Plate carrée projection.

Notice the flattened map projection in Cartopy from the Plate carrée map projection! It looks a little like from a helicopter view when you look (far) ahead instead of looking right down.

- Also notice the lack of 'water color' makes the map more difficult to interpret.

So now you have an idea of what simple topographical maps in Cartopy can look like with the plotted data points that represent the logged GPS locations from multiple GPS track logs!

1.9.3 Challenge – Build extra skills in Python with Cartopy

Tip: for a data scientist it is good to get some solid skills in Python programming with Cartopy because data visualization on both topographical and thematic maps is everywhere!

- In let's say 2 – 4 hours you will be able to do a lot more with Cartopy!
 - If you want to give it a day that's even better!
- Therefore it is a good idea to go on and experiment a little more in Cartopy with these two overlapping car tracks in Zeeland.
- Take a look at the Cartopy documentation to learn more!
 - For instance by finding out how to get a better visualization by plotting the track's location points with different colors and different location marker styles (dots, squares, triangles).
 - Or find out how to zoom in on just a part of the overlapping tracks and then plot lines between the location markers in different line styles for each car trip.
 - It is also nice for orientation on the map to add a few locations with a marker and a text, like the country codes in the center of the country or a few capitols with their names or a few cities, towns and villages with their names.
 - To improve this situational awareness when looking at a map in Cartopy that is just adding a list of data points as a layer in a scatter plot with for each data point the specification of that Point of Interest (POI) the data fields for the GPS coordinates, a marker, the marker text and the text location (e.g. left or right side display from the marker position).
 - You could add a list-of-lists or a list-of-tuples for the list of POIs you want to display with for each POI a list item (list or tuple) with the data fields for their specifications.
 - Then write a simple function that uses a simple for loop to iterate through the list to add the POI in each list item as a data point to the scatter plot by passing the specifications as parameters.
- Remember to also experiment some more with Cartopy by using detailed maps from Web Map Services like OpenStreetMap.org after you have finished this workshop!
 - Take a day or two to experiment and get things up and running.
 - That will give you easy detailed topographical maps directly from Python with Cartopy if you don't need a web application.
 - Otherwise it's using OpenLayers or Leaflet from JavaScript / TypeScript as shown in this workshop.
 - And then there is the way in the middle for a web app by using Folium for Leaflet maps from Python if you want to stay inside the Python ecosystem.
 - Note: unfortunately, there is no Python wrapper package for OpenLayers.
 - Once you now OpenLayers from this workshop as a reference it should be easy to learn to use Leaflet! So, give it a go later!
 - Tip: if you can get geospatial datasets with borders of provinces, city borders etc. then you can draw these as overlays within country borders to create more detailed thematic country maps in Cartopy.

2 Workshop - Part 2 – Data Storage: MongoDB

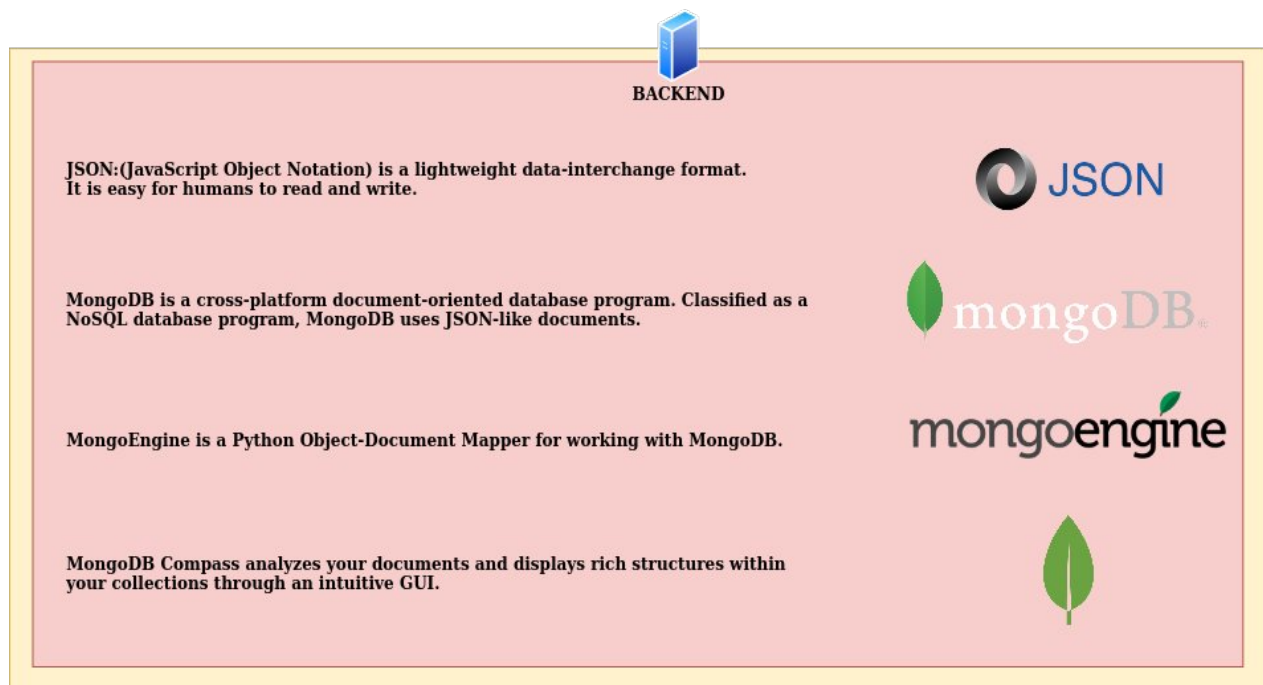
The focus is on data modeling and data storage in MongoDB databases to store datasets from different CSV and GPX file formats in a standardized way in the format of JSON documents.

- To do that you need to convert the CSV files that hold the GPS track logs from cranes and the GPX files that hold the GPS track logs from car trips into JSON data first for further standardized programming because handling multiple data formats is always less easy.

You could just as well use a relational database to store the datasets but the choice for a JSON document store was made to learn how to handle JSON data for 3 reasons:

1. In datascience storing (geospatial) sensor data in JSON documents and spatial objects in the GeoJSON format is maybe even more common than storing these in a spatial SQL database, like in PostgreSQL with its spatial PostGIS extensions or in MariaDB Spatial.
2. The two 'go-to' Open Source JSON document stores would be the MongoDB data store and the Elasticsearch search engine of which MongoDB has the lowest learning curve.
3. Both MongoDB and Elasticsearch have good API's and Python clients so you can stay within the Python ecosystem with Pandas and Flask.
 - For data handling you will use Pandas in this chapter and in the next chapter you will learn to use Flask to access the data in the MongoDB databases.

2.1 Data Storage Environment for Datasets



Simply said, we need MongoDB running as a database server on Ubuntu Linux and then we can use MongoDB Compass as a database management tool to create and manage the databases to store JSON documents and we use mongoengine to write the JSON documents to the database. There is also a Python client called pymongo as the preferred tool to read the data from the MongoDB databases but to keep things simple for the workshop we will stick to using mongoengine for now.

2.2 Terminology explained – Data storage in MongoDB

How MongoDB stores the 'NoSQL' data is pretty much the same as it would be in a relational database for 'SQL' data, so let's make a global comparison.

First of all MongoDB is referred to as a NoSQL datastore because we don't use SQL as a query language to write, change and read data records to and from database tables.

When the MongoDB package is installed on Ubuntu Linux the database server daemon process 'mongod' is started.

- Check this in a terminal with the Linux command 'top' that shows all running processes.
- This is pretty much the same as the Relational DataBase Manager System (RDBMS) server program that would be running to access a relational database with SQL.

A data model in MongoDB is also comparable to a database schema for a relational database and you can also define search indexes on the data fields of the JSON documents.

- The beauty in MongoDB is that you don't have to create a data model and indexes but doing so will speed up the required run time of data queries a lot!
- A set of JSON documents in MongoDB is called a collection and this is comparable to a table in a relational database.
- A JSON document is then comparable to a record which is a table row in a relational database.
- A collection can have up to 64 indexes defined on the data fields in the JSON documents to speed up common queries in searches for data retrieval.

When MongoDB is installed then 3 default databases are created automatically:

1. Admin database: to store passwords for instance.
2. Configuration database: to store configuration information.
3. Local database: to store log records with information about starting, running an stopping MongoDB.

If you start the database management tool MongoDB Compass then you will see those 3 default databases listed too.

Let's summarize the MongoDB terminology!

1. MongoDB is a NoSQL data store for storing JSON documents.
2. It has not a fixed data structure as opposed to a relational database (e.g. PostgreSQL).
3. It works faster and is more flexible than a relational database.
4. A data structure is created using data models.
5. The structure of the data storage is similar to that of a relational database (SQL).

Relational Database – SQL Database		MongoDB – NoSQL Database
A database consists of tables	-->	A database consists of collections
A table consists of records	-->	A collection consists of documents
A record consists of fields (as columns)	-->	A document consists of fields

Table 1: Table of naming conventions for a SQL database compared to a MongoDB NoSQL database.

2.3 MongoDB use in the GeoStack Workshop

The intention of this course is to also learn a few things about Systems & Network Management (SNM) in Linux to understand how the GeoStack is build and run so you can do it yourself later. Therefore this section is about how MongoDB is run in the VM.

The packages for MongoDB, MongoDB Compass and MongoEngine were installed by installation script 4-backend-software.sh.

When the VM is started then the MongoDB database server mongod (= MongoDB daemon) is started automatically.

- If you open a terminal and give the command `top`, then you see the running process for `mongod` listed in the list of running applications.
- The `top` monitoring tool shows a real-time list of all running applications and how much CPU time and memory the use.
- Quit the monitoring application `top` again by entering the key combination `Ctrl + C`.

In the workshop the `mongod` server daemon runs directly on the Ubuntu Linux operating system and two databases are created as JSON document data stores to store the datasets.

- In the Course you will also learn how to run MongoDB in Docker containers and volumes as the sandboxes to isolate them from your Ubuntu host OS in the VM.

The two databases are the `Crane_Database` and the `Trail_Database`:

1. The installation script `crane-datasets-import.py` reads the CSV files with the GPS track logs from the ringed cranes, creates the database `Crane_Database` and writes the log records into the database as JSON documents according to the defined data model.
 - The `Crane_Database` holds two data collections: `tracker` and `transmission`.
 - The `tracker` collection holds the JSON documents (records) with the information of each tracker, like the tracker ID and name of the crane.
 - The `transmission` collection holds the JSON documents with the track log records of each GPS location that was calculated from the GPS satellite radio transmissions and the tracker ID as an unique identifier key so each track log record can be perfectly linked to the individual crane name.
2. The installation script `trail-datasets-import.py` reads the GPX files with the GPS track logs from the car trips, creates the database `Trail_Database` and writes the log records into the database as JSON documents according to the defined datamodel.
 - The `Trail_Database` holds two data collections: `signals` and `trails`.
 - The `signals` collection holds the JSON documents (records) with the information of each car trip. Basically it's just the trail name of the car trip.
 - The `trails` collection holds the JSON documents with the track log records of each GPS location and the signal ID with the trail name as an unique identifier key so each track log record can be perfectly linked to the individual car trip.

What does the MongoDB data model look like?

- The Data Model is in the Python Source Code and NOT in the MongoDB database!
 - MongoDB is a so called schemaless database. This means that the document structures of fields for each collection exist only in the (Python) application source code.
 - This makes it easy to make changes but it makes the class definitions in the (Python) source code the documentation! --> Make inline comments to explain!!!
 - These class definitions for the database schema are called data models in MongoDB!
- As part of the design of the data model to store the data fields in a JSON document collection a track log record is NOT stored as a single record in a JSON document.
 - Instead, the related fields are grouped together in 3 logical subgroups, named: geometry, speed and tracker metadata.
 - These subgroups of data fields are called embedded documents in MongoDB and by defining them it is possible to retrieve them very easy in a query as a separate sub-records.
 - For instance in a crane track log record there is a group of datafields related to the GPS receiver sensor for the location and a group of datafields related to other sensors in the tracker like for the battery level, daylight etc.
 - The idea is to group the related data fields together in separate sub-records which translates to separate embedded JSON documents in MongoDB.
- With the database management tool MongoDB Compass you can take a look at the Crane_Database and see how the data is stored in those 3 embedded document sub-records.
 - Start MongoDB Compass from the left desktop menu Favourites by clicking the 'Green Leaf' icon.

Which databases are created for the programming assignments?

- In the workshop you will create 2 databases yourself with the same database name plus an underscore sign ('_') as a post-fix character to make them unique.
- The database names will be: Crane_Database_ and Trail_Database_
 - Warning: when programming and studying the source code be aware of that extra underscore to distinguish the supplied database and the self-created databases from each other!!!
- In the assignment you will store both the crane and car trip datasets and then use it in Python Cartopy to draw them on a digital topographical map.
To do that you can either pick the self-created databases (with the extra underscore at the end in the database name) or the supplied databases.

If you start MongoDB Compass you will see a total of 5 databases listed:

- 3 default databases from MongoDB itself: Admin, Configuration and Local
- 2 workshop databases created by installation scripts: Crane_Database and Trail_Database

If you look in MongoDB Compass later on in the workshop you will also see the 2 extra databases listed, you are going to create yourself in the assignments: Crane_Database_ and Trail_Database_

In the next section more on data modeling in MongoDB.

2.4 Exploring Data Modeling in MongoDB

Modeling datasets is done by creating the structure of a JSON document.

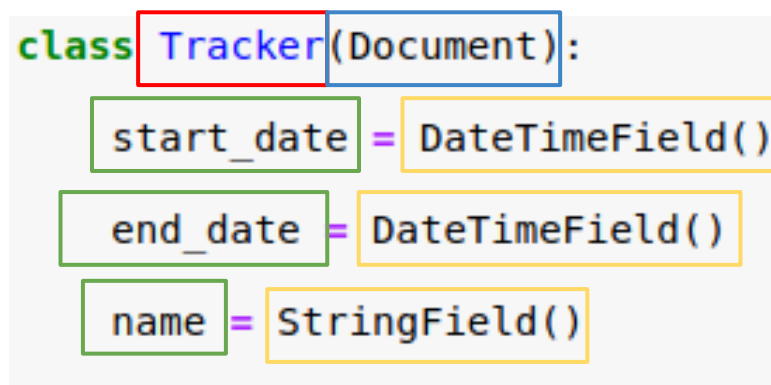
- A JSON document needs a template in which the fields are specified in a class definition.
- A set of JSON documents, stored in a MongoDB database is called a collection.
- Remember, in Python the data model for a MongoDB database is specified with class definitions for the collections because MongoDB is a schemaless database server!

2.4.1 Crane Datasets – Modeling the Tracker collection

The order in which the data fields are stored is determined on the basis of the class definition.

In the case of the crane datasets, we are going to create 2 document structures:

- A tracker document, which contains all the data related to 1 tracker.
- A transmission document, which contains all the data related to a tracker.



```
class Tracker(Document):
    start_date = DateTimeField()
    end_date = DateTimeField()
    name = StringField()
```

Drawing 9: Defining a Tracker JSON document in Python.

The image above is an example of how to define a JSON document structure for the data of a GPS tracker belonging to a crane by specifying a class definition with the self-chosen name Tracker.

Defining a JSON document structure is a simple 3 step process for which you do the following:

1. First indicate what the name of the JSON document collection is going to be (in Red).
 - In this case the document name will be called 'Tracker'.
2. Then indicate what type of JSON document the document is (in Blue).
 - In this case it is a normal document type which is specified as 'Document'.
3. Finally specify the field names (in Green) and their data types (in Yellow) to create the JSON document structure.
 - In this case the fields for the start and end dates are used for the dates of the first and last GPS message in the GPS track log that indicates the time period (= time frame) of the received GPS location transmissions.
 - In this case the name field is used to store the name of the crane for easy reference because it is better to remember than working with the ID numbers of the GPS trackers.

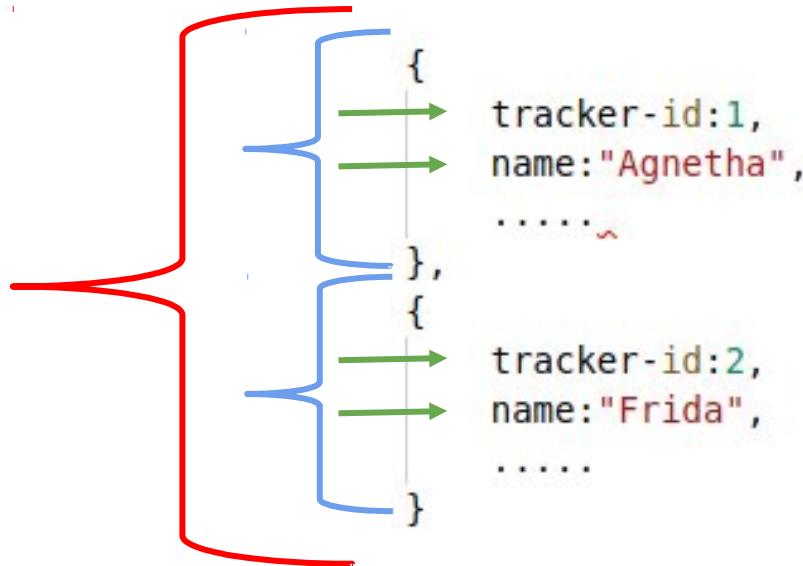
Of course you will need another database collection 'Transmissions' in MongoDB for the GPS transmissions of the GPS trackers which is where you store the track logs for **all** the GPS trackers!

- **IMPORTANT NOTICE:** in the next section about 'Modeling the Transmission collection' the field 'tracker-id' is introduced as the 'shared field' between the two collections and for the transmissions the field 'transmission-id' is also introduced. This is just for educational purposes to learn how relationships work between collections which is then explained further in the section about 'Modeling the Tracker and Transmission relation'!

2.4.2 Crane Datasets – Modeling the Transmission collection

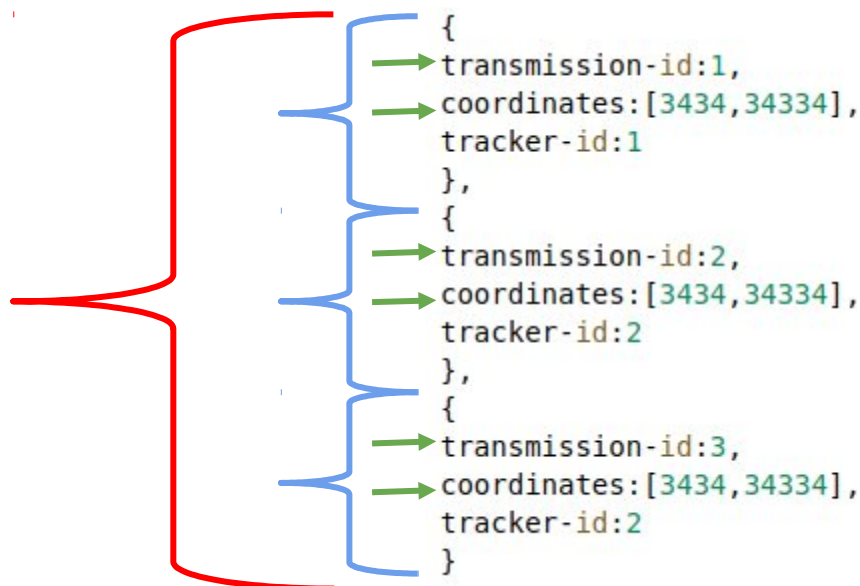
In the case of the Crane data sets, we are going to define 2 types of JSON document structures. This means we will have 2 collections in our database: Tracker and Transmission

1. A Tracker collection containing all tracker documents with the the tracker data.
 - A tracker document contains the description of the GPS trackers, which is more or less the 'STATIC' data, at least the name and start date fields because for live storage you would have to update the field for the end date if new transmissions are received.



Drawing 10: A tracker collection with 2 trackers with tracker-id 1 + 2.

2. A Transmission collection containing **all** transmission documents from **all** GPS trackers, which contains all fields related to the transmission data.
 - A transmission document contains information related to the position of the Crane which is the 'DYNAMIC' data collection because more GPS track log records (transmissions) can be stored if the GPS tracker keeps functioning.
 - Storing all transmissions in one (1) collection is a convenient database design choice!
 - The other design option is a separate Transmission collection for each GPS tracker which is harder to program when new trackers need to be added or removed.



Drawing 11: A transmission collection with 3 transmissions from 2 trackers!

2.4.3 Crane Datasets – Modeling the Tracker and Transmission relation

To be clear:

- Each tracker sends a large amount of transmissions: this is a 1:n (= '1 to many') relation!
- Each transmission in a CSV file of a GPS track log is received from the same GPS tracker!

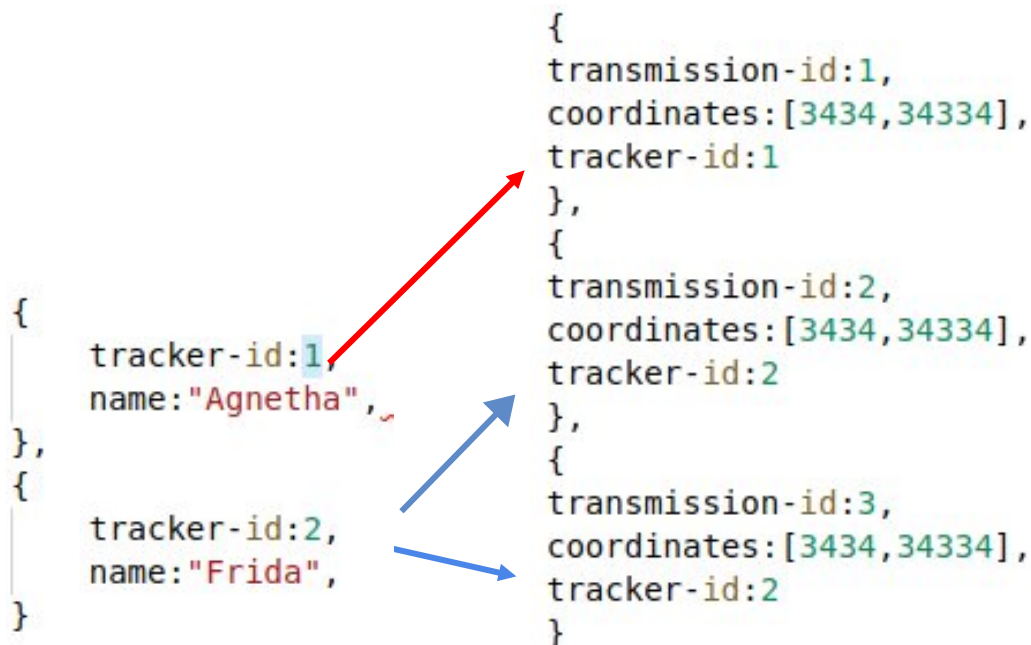
This means there is a cross reference relation between the 2 different types of JSON documents that were defined as the Tracker and Transmission documents.

- Adding relations in MongoDB data modeling is done by using a ReferenceField() statement in the Python class definition for the Transmission document to add that extra relation field that is the 'shared key' or 'shared field' which both collections have in common.

```
class Transmission(Document):  
    .....  
  
    tracker = ReferenceField(Tracker)
```

Drawing 12: Defining a reference field 'tracker' in Python for the tracker ID.

- This reference field is a 'shared field' that can be compared as a 'shared key' with a 'Foreign Key' field in a relational database record to link two data tables together.
- In the crane database design of the workshop you can look at the reference field as a 'shared key' field in a Transmission document that holds the corresponding value of the field tracker-id of a Tracker document.



Drawing 13: Example of two references to the same tracker ID '2' in the Transmission collection that were when the track log records were written as JSON documents to the collection to be able to make a query later to retrieve the crane name 'Frida' that is associated with this document (transmission log record).

IMPORTANT NOTICE: the fields 'tracker-id' and 'transmission-id' work in MongoDB as explained but for programming you just have to know for now these are 'invisible' fields that MongoDB automatically creates when a JSON document for a tracker or transmission is saved in the database. How it works is in the section 'Assignment – Some Black Magic Explained for Newbies!'.

2.5 Assignments – Getting the Datasets in MongoDB

2.5.1 Assignment 1 – Programming the MongoDB Data Models

Goal: program the MongoDB data models by defining Python classes for the JSON document collections.

- You will need these data models in the next assignment to load GPS track logs of ringed cranes and car trips in the self-created databases: `Crane_Database_` and `Trail_Database_`.
- Note: remember, there is the extra post-fix underscore in each self-created database name!

Assignment – Programming the MongoDB Data Models

1. Click again on the desktop shortcut from Part 1 called: “jupyter-lab”.
 - This of course opens the JupyterLab Launcher screen again to select the notebook.
2. Now go to the folder ‘Part-2-Data-storage/Notebooks’, where you will find 2 sub-folders:
 1. Sub-folder Assignment: here you will find the Jupyter Notebook called ‘Assignment-Data-storage’.
 2. Sub-folder Solution: here you will find the solutions.
 - In case you get stuck you can always take a look at this notebook.
3. Click the notebook ‘Assignment-Data-storage’ to open it and start with the assignments.
 - In this notebook the first 3 assignments are related to creating the data models.
 - Note: everywhere you see the text: “#TODO”, you will have to complete the code!
 - Note: data types for MongoDB are in ‘CamelCase()’ notation, so for a integer field name the ‘int’ field data type is written as CamelCase + ‘function’ braces at the end: `IntField()`

Assignment Steps

The notebook will take you through the following 5 steps to handle the data for which most of the code is complete except for the some lacking code in the most educational parts:

1. Reading the datasets
2. Creating data models: remember to use the `CamelCase()` ‘function’ notation for data types!
3. Creating the import functions
4. Connecting to the MongoDB datastores
5. Using the import functions

So, to learn the most important coding parts there are 5 assignments in the notebook with the educational focus to get the data models right by letting you specify the correct data types for the fields and then completing the functions that actually import the data into the MongoDB databases:

1. Assignment 1: Complete the Tracker document with the correct FieldTypes.
2. Assignment 2: Complete the Transmission document with the correct FieldTypes.
3. Assignment 3: Complete the Signal document with the correct FieldTypes.
4. Assignment 4: Complete the Function for creating and importing the Trail documents.
5. Assignment 5: Complete the Function for creating and importing the Signal documents.

Challenge – Experiment with Pandas and the Python module `datetime`

If you are new to Pandas and Python it is a good idea to take 2 – 4 hours to experiment a little bit with handling dataframes and JSON files in Pandas and learning to use the `datetime` module.

- Not only because JSON files, dataframes and date/time handling are used in this notebook but also because they are very often basic building blocks in data science programming.

Check out these weblinks:

- Try this clear and quick tutorial: <https://www.geeksforgeeks.org/python-pandas-dataframe/>
 - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html
 - Tip: also search for ‘pandas json to dataframe’ for some examples and explanations.
- <https://docs.python.org/3/library/datetime.html> (The module’s documentation web page.)
 - Try some nice examples here: https://www.w3schools.com/python/python_datetime.asp

2.5.2 Assignment 2 – Loading the Crane and Car datasets

Goal: load the GPS track logs of the ringed cranes and car trips in the self-created databases: Crane_Database_ and Trail_Database_

- For educational purposes this section will use the Python code for the Car datasets to explain the idea of the assignment instead of the fairly similar source code for the crane datasets.
- Note: remember, there is the extra post-fix underscore in the database name!
- Note: the assignment is at the end of this section but read the explanation on Python programming for MongoDB with Pandas dataframes here first before you dive into the Jupyter Notebook 'Assignment-Data-storage' for Part 2 again to continue!!!

Now we have finished defining our data models in the previous section, we are now going to use them in a Python function to load the data in the MongoDB database in this assignment.

```
def load_route_data(df,name):  
    trail = Trail(name=name)  
    trail.save()  
    for index,row in df.iterrows():  
        signal = Signal(timestamp = row['time'],  
                        coord=[row['lon'],row['lat']],  
                        alt = row['alt'],  
                        trail = trail)  
        signal.save()
```

Drawing 14: Python function to load the GPS track logs of the car trips (trails).

Loading the data is done by using a Python function, see image above.

- Notice in the Python function definition on the first line it says 'df' which is an abbreviation for Pandas dataframe which is a Python data structure in memory (RAM) that holds a table with in our case in each row a GPS track log record as read from the GPX file of a car trip.

So, let's take a look at how this source code works:

1. First we define the Python function in two steps:
 1. We give a name to the function, which is load_route_data(). (Red)
 2. Then we define the 2 arguments for the function for which 2 variables have to be passed as parameters into the function when it is called in the Python script. (Blue)
 - The two arguments are the Pandas dataframe (df) and the route name (name).
2. This load_route_data() function will do the following 2 things for each dataset GPX file:
 1. Process the trail data (track log metadata).
 2. Process the signal data (track log records)

Now let's see how the data processing goes for both the trail and the signal data:

1. Processing the trail data is a simple 2-step process:
 1. Create a new route object in the trail variable (= trail = track) with the data type of class Trail (= instance of class Trail) and pass the (route) name of the car trip to it. (Green)
 2. Then we save the new trail variable as a JSON document of class type Trail in our MongoDB database collection Trail by calling the objects .save() method. (Yellow)
 - Notice this only needs to happen once for each track log file just to register the (route) name of which car trip we process.
2. Processing the signal data is also a 2-step process but this runs in a loop (Orange):
 1. Step 1 in the loop is to get a data record from the next row in the Pandas dataframe and create a message object in the signal variable for each of the position messages (Signals) in the GPX-Track with the data type of class Signal (= instance of class Signal).
 - Notice the mapping of the data fields (columns) from the row of the Pandas dataframe to the corresponding data field names of the class type Signal. (Purple)
 2. Step 2 in the loop is to save the signal variable as a JSON document of class type Signal in our MongoDB database collection Signal by calling the objects .save() method. (Grey)

Finally when we call the function `load_route_data()` in the Python script, we pass the Pandas dataframe 'Biesbosch' and the name "Biesbosch" of the GPS-Route (Trail) as the 2 required parameters as shown in the image below.

```
load_route_data(Biesbosch,"Biesbosch")
```

Drawing 15: Example of the Python function call to load the trail dataset 'Biesbosch' into the MongoDB collections Trail and Signal.

Assignment – Loading the Crane and Car datasets in MongoDB

1. Click again on the desktop shortcut called: "jupyter-lab".
2. This opens the JupyterLab Launcher screen to open the Jupyter Notebook.
3. Go to the folder 'Part-2-Data-storage/Notebooks', where you will find 2 sub-folders:
 1. Sub-folder Assignment: here you will find the notebook 'Assignment-Data-storage'.
 2. Sub-folder Solution: here you will find the solutions.
 - In case you get stuck you can always take a look at this notebook.
4. In this notebook 'Assignment-Data-storage', you will find the remaining assignments to load the crane and car trip datasets into the databases: `Crane_Database_` and `Trail_Database_`.
 - When you finished these assignments go to the assignment in the next section to take a look at the databases and the JSON documents (and embedded documents) with MongoDB Compass!
 - Note: notice the underscore ('_') suffix, added at the end of the database names in the source code in the notebook to differentiate them from the database names without the underscore that were created by an installation script of the workshop.
 - Also see the next section on how you can easily see the database names in the database management tool MongoDB Compass!

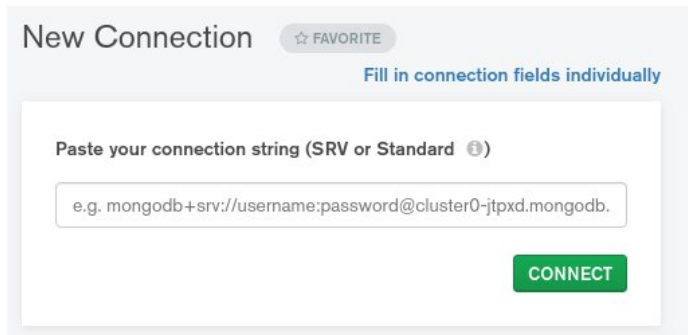
2.5.3 Assignment 3 - Checking the Datasets in MongoDB

To validate the results from the previous part of the workshop, we need to check the datasets:

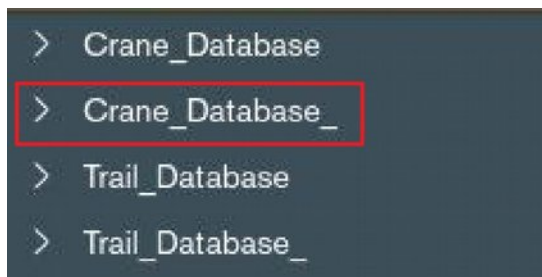
1. Open MongoDB Compass: click on the 'green leaf' icon in the left desktop menu Favorites.



2. Click on the green button 'CONNECT'



3. Click on the database name 'Crane_Database_' in the list of databases in MongoCompass and take care you select the database name with the extra underscore suffix because that is the database you created.



4. Check if the values are the same as shown here in the screenshot:

tracker	3
transmission	236,226

5. Go back to step 3 above and now select the Trail_Database.
6. Check if the values are the same as shown here in the screenshot:

signal	3,684
trail	3

Notice the collection names of the databases are listed in alphabetical order.

2.5.4 Assignment – Some Black Magic Explained for Newbies!

This is an important reading assignment about how the ‘Black Magic’ technical implementations in MongoDB, Flask, Python and Angular influence programming **if you are a Newbie to this!**

- **You need to know this** to understand the programming assignments and source code of Part 3 for the Flask middleware app and Part 4 for the web app, **so read very carefully!**
- **IMPORTANT NOTE: a choice was made to group all ‘Black Magic’ explanations on how to get to the data here in this section! So yes, also for Part 3 and 4! If you find some explanations too complicated at this time, don’t worry! Simply skip those parts for now and come back here later for extra clarification for your assignments!**

Now you have completed the programming assignments to get the datasets in the MongoDB databases of the Cranes and the Cars it’s time to explain some of the inner workings.

- To do that some source code or just lines of source code are given here as a reference to help you in recognizing, finding and understanding this code in the assignments later on!

MongoDB Magic (referencing Part 2: data storage)

MongoDB works with Object Identifiers or Object IDs which is abbreviated to ‘OID’ in text or ‘oid’ or ‘ObjectId’ in source code. A ‘tracker-id’ is an example of a variable name for such an OID.

- An OID is created automatically as an extra data field which is added to a JSON document when a JSON document is saved to a JSON document collection in a MongoDB database.
- The value of an OID data field is a random alpha-numerical code that is generated by MongoDB. Read more: <https://docs.mongodb.com/manual/reference/method/ObjectId/>
- An OID field is ‘invisible’ in the data record itself, which means it is not part of the set of fields that you have to declare in a Python class to specify the data model of a JSON document type.
- As a programmer you can use these OID data fields very conveniently to model relationships because with a reference field you simply refer to a JSON document object to let MongoDB automatically substitute the OID at run time.
- When programming it is a best practice to use clear names for variables that explain what they do and that is why in the source code of the workshop a variable that is needed to hold the value of an OID has an ‘-id’ or ‘._id’ suffix in its name to indicate it holds a value for an identifier or identity code.
 - Examples are: tracker-id, transmission-id, trail-id, signal-id etc. and in the object oriented programming notation it is for instance tracker._id.

Flask Magic (referencing Part 3: middleware)

When a Python Flask application sends an HTTP data request to MongoDB it gets a Binary JSON (BSON) object in return because that is what MongoDB uses as its data format for network communication. Read more about BSON here: <https://en.wikipedia.org/wiki/BSON>

In the Flask app the BSON object is automatically decoded by Flask to a Python object with UTF-8 string values, so all field names and values are converted back from a binary format to text.

- This means as a programmer you will have to address the OID in the workshop as a string!
- In the Flask app script app.py the MongoDB OID use is enabled by the following statement:
 - `from bson.objectid import ObjectId`
- An example is in Part 4 in the file map.component.html where the object.variable notation of ‘tracker-id._\$oid’ is used to get the value of the variable ‘oid’ which is a string variable as indicated by the ‘\$’ prefix.
- In the GeoStack Course some extra BSON decoding code is added in the Flask app to be able to address variables with the short notation of just the variable name, like ‘tracker-id’ as best practice programming requires this instead of the longer object notation. The extra decoding code was left out in the workshop assignments to keep things simple.

Python Magic - Addressing Latitude and Longitude in Pandas dataframes

To load the crane and car data into MongoDB you need to write code that addresses data fields in data records by name if you can to get human readable code, which makes it maintainable.

Let's take the GPS coordinates as an example and see where the field names come from in the Crane CSV files and how they get into a Pandas dataframe to enable data processing in Python.

Remember in the CSV files with the GPS track logs of the cranes the first line holds the field names for the track records. In a tabular notation these would be the column names.

- Latitude and longitude are two of those field names, abbreviated to 'lat' and 'lon' in the first line of the CSV file.

Let's take a look at the source code in the Jupyter Notebook 'Assignment-Data-processing.ipynb' of Part 1 'Data processing'.

- These CSV files are read with the Pandas function `read_csv()`, like for crane 'Agnetha' a Pandas dataframe object with the name 'Agnetha' is created with the statement:
 - `Agnetha = pd.read_csv('../Datasets/CSV/20181003_Dataset_SV_TrackerID_9407_ColorCode_RRW-BuGY_Crane_Agnetha.csv')`
 - Notice: `read_csv()` is called a function here for ease of use but strictly speaking in object oriented programming a function that is defined in a class definition is called a method!
 - This Pandas `read_csv()` method is used here to create a variable 'Agnetha' which is an object instance of the Pandas data structure type of a dataframe.
 - Remember, a dataframe in Pandas is a tabular data structure like in a spreadsheet, with the header that holds the column (= field) names in row 0 (zero; = 1st line of the CSV file) and the other rows hold the data records (= the rest of the lines in the CSV files)!
 - Also remember for Pandas that you can address the header fields and thus the fields in each data row by name, which is called a label in Pandas.
 - This 'field reference by name' is a best practice that is used to keep the code human readable instead of using indices, so 'lat' and 'lon' are used as field names to get the field values (= the actual data values, which are the GPS coordinate values)!
 - Then when you export the crane datasets to JSON files in the folder `Datasets/JSON` the Pandas function (method, actually!) `to_json()` is used to write the JSON file from the dataframe object 'Agnetha' with this statement:
 - `Agnetha.to_json('../Datasets/JSON/Crane-Agnetha.json',orient = 'records')`
 - Writing a data structure to a file is a process that is in general called 'serialization'!
 - When you open the JSON file you can see:
 1. it contains a single 'main' list, noted between square brackets '[' and ']', just as lists are coded in Python;
 2. it is a list of dictionaries, noted in 'curly braces '{' and '}', just as dictionaries are coded in Python, with a dictionary for each data row in the dataframe 'table';
 3. each dictionary holds a set of key-value pairs in which the key is the field name from the dataframe header field labels (that were originally read from the CSV file) that can be used to retrieve the corresponding value.
 - To read the JSON file back in for some data processing in Python the Pandas function `read_json()` is used, like to create a dataframe for crane 'Agnetha' with the statement:
 - `Agnetha = pd.read_json('../Datasets/JSON/Crane-Agnetha.json')`

Python Magic – From Python to MongoDB for GPS coordinates

To handle the GPS coordinates a coordinate variable 'coord' of data type 'PointField()' is declared in the class definitions of classes Transmission and Signal with the statement: coord = PointField()

- Remember, these Python classes define the type and fields of the JSON document.

To get the 'lat' and 'lon' coordinate values you will find code lines with field names to do that, like:

- in the function load_crane_data(): coord = [row['location-long'],row['location-lat']]
- in the function load_route_data(): coord=[row['lon'],row['lat']]

Notice the field names for latitude and longitude in the Crane CSV files are 'location-long' and 'location-lat' and in the Car GPX files they are 'lon' and 'lat', so they must be used exactly like that for field reference by name here too, but...

- if you see things like this it is much better to adjust the field names in either the CSV file or when writing the JSON file to ensure all your Python code has standardized field names. It would have been better for instance if 'location-lon' was renamed to 'lon' etc.

Python Magic – What happens in the load_crane_data() function for MongoDB?

Let's clarify a little bit to learn to read the code on how to get the crane data in MongoDB.

- The same goes for the similar load_route_data() function but let's stick with the cranes because that function is code complete in the notebook 'Assignment-Data-storage.ipynb' but the load_route_data() function is an assignment for code completion in that notebook!
- Note: the full explanation of the source code is in notebook of the GeoStack Course!

Here's the code for the function definition:

```
def load_crane_data(df,name):

    tracker = Tracker(study_name = df.at[0,'study-name'],
                      individual_taxon_canonical_name = df.at[0,'individual-taxon-canonical-name'],
                      individual_local_identifier = df.at[0,'individual-local-identifier'],
                      name = name,).save()

    transmissions = []

    for index,row in df.iterrows():
        transmissions.append(Transmission(event_id = row['event-id'],
                                           timestamp = row['timestamp'],
                                           coord = [row['location-long'],row['location-lat']],
                                           alt = row['height-above-ellipsoid'],
                                           speed = row['ground-speed'],
                                           tracker = tracker))

    Transmission.objects.insert(transmissions,load_bulk=True)

    print("Done importing (Crane)Tracker: " + name)
```

Some code explained --> more detailed explanations are in the GeoStack Course notebook:

1. The crane name, like 'Agnetha', to be passed as parameter 'name' in the function call is used in the assignment statement of the tracker object and in the print() function call.
 - Remember the crane names are from some famous Swedish ladies in pop music!
 - The rest of the field names in the Tracker() and Transmission() class calls in the tracker and transmission object instantiation assignments were defined in the Crane CSV files.
2. Tracker field names: these values are retrieved from the dataframe header ('row zero') by label name with 'df.at[0,'label-name']'. This is the definitely non-scientific explanation but it's just to get the idea!

3. Transmission field names: these 'label' names are the keys in the key-value pairs of the data rows in the dataframe.
4. The 'tracker = Tracker().save()' assignment not only creates the tracker object but calling the save() method on that object in that statement too, also saves it in the database!
 - At that point the tracker object is saved (= written) as a JSON document to the Tracker collection in the crane database in MongoDB.
 - This is where the magic happens because MongoDB automatically adds the extra OID data field with an Unique Identifier (UID) value to that JSON document that is referred to in this workshop as the 'tracker-id' field in the source code of the web app in Part 4!
5. Then an empty list is created to hold all the transmissions (= GPS track log records).
6. Next a for loop is used to iterate over all the rows in the dataframe with the 'df.iterrows()' method to append each transmission to the list.
 - How it works with the 'coord' field for the GPS coordinates is already explained above.
 - The 'tracker = tracker' assignment makes the MongoDB OID of the tracker object is added as the 'shared field' or 'foreign key' to the transmission JSON document. This is where the 'tracker = ReferenceField(Tracker)' declaration in the Transmission() class definition works its magic!
7. Finally the list of transmissions is written in one 'bulk' statement into the Crane Database in MongoDB and here also the invisible magic happens!
 - Each transmission is written as a JSON document in the Transmission collection and also 'automagically' gets its OID which in this workshop is called the 'transmission-id'.
 - The difference between the 'bulk' save after the for loop for all transmissions and the save of each signal in the for loop is explained in more detail in the GeoStack Course but it is to show there are performance benefits when writing large datasets in 'bulk'.

MongoDB Magic in Database Creation

In the Part 2 notebook 'Assignment-Data-storage.ipynb' there is the main part of the script that starts after the comment 'Connecting to the Crane Database' with 2 important statements:

1. disconnect('default'): to stop any currently open database connection, just to make sure.
2. connect('Crane_Database_'): the 'Magic Statement' that either connects to an existing database or creates a new database if it doesn't exist yet and then connects to it!
 - Notice the extra underscore suffix you have to program to distinguish the database name from the database 'Crane_Database' that was provided with the workshop!
 - Note: type the database name carefully because if you make a typo in the database name then a database with the wrong name is created which you will have to remove manually with MongoCompass or from the command line to prevent database pollution.
 - About the connect statement: to be precise, it is actually not the connect statement itself that creates the MongoDB database because the connect statement merely provides the database name as a place holder or alias if you will to identify the database.
 - Look at it in the same way as a specification for a port, a proxy URL or a symbolic link. These specification statements create an alias and do nothing.
 - The actual database with the database name that is provided in the connect statement is created by MongoDB when the application creates a collection with at least one JSON document as a 'database record' in it.

Challenge – Read up on Pandas dataframe labels and Python OOP in 1 hour

If you are new to Pandas and OOP in Python it also might be very nice to have a quick read about that now because it will help you in reading and understanding the source code much easier!

- The documentation on dataframe.at is here:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.at.html>
(but..., it is more complicated than the simplified non-scientific explanation above!)
- Read more about Python classes and Object Oriented Programming (OOP) in the two short tutorials here: <https://www.programiz.com/python-programming/class> and here: <https://www.programiz.com/python-programming/object-oriented-programming> and in the official Python documentation here: <https://docs.python.org/3/tutorial/classes.html>

IMPORTANT NOTE: the rest of this page is intentionally left blank to start everything related to explaining how the web application with Angular works on the next page!

- Notice the Angular part in this section is only explaining how to get to the data!
- How an Angular web app works internally and of which files it is composed is only explained briefly here because in Part 4 this is explained in detail, including the internal routing between the files.
- For now you only need to know the following to understand in general how things work in an Angular web app to get to the data from a MongoDB database:
 - An Angular app has one or more components. The 2D Map Viewer web app only has one (1) component, called the 'map' component.
You can give a component any name you want and we simply chose: 'map'!
 - Each component has at least 3 files:
 1. A component.ts file with the TypeScript code for what the app needs to do.
So the component's TypeScript file name is: map.component.ts
 2. A service.ts file with the TypeScript code to use web services (like how to get to MongoDB databases) which is called (imported) in the component.ts file.
Because the app needs a service to get crane datasets the name 'Crane Service' was chosen, so the TypeScript service file name is: crane.service.ts
 3. A component.html file for the HTML5 stuff the web browser needs for the app.
So the component's HTML file name is: map.component.html

Angular Magic – The MongoDB Connection to the Web App! (referencing Part 4: web app)

The previous explanations all related to MongoDB aspects for the assignments in the Jupyter Notebooks of Part 1 and 2 except for a little MongoDB look ahead for Flask in Part 3.

To complete the list of MongoDB connections in the workshop, let's take a look further ahead to Part 4 of the workshop to see what the Angular Magic is to request and receive data from the databases in MongoDB.

- **Note:** this 'Angular – MongoDB' part of the code explanation is not explained in Part 4 because the choice was made to keep all the MongoDB explanations in this assignment!
 - Tip: read this 'Angular Magic' explanation now and then do come back to it in Part 4!
- How the entire Angular bootstrap process works to start the web application, that is explained in more detail in Part 4 of the workshop, so here's the short version:
 - A web app based on the Angular framework consists of application components and these components always have 2 compulsory files which are a TypeScript file and a HTML file.
 - In the workshop web app these 2 files are `map.component.ts` file and `map.component.html` of which the TypeScript file is loaded first by Angular, so let's walk through that file first.

map.component.ts --> What is MongoDB related?

The import statement for the `CraneService` makes the `map.service.ts` file is loaded to make the network connection to the Flask API for HTTP requests that will then be relayed to MongoDB.

The `@Component` decorator makes Angular aware the `CraneService` has to be linked to the `map.component.html` file which serves the Settings menu in the web app in which you can select the crane of which you want to see the flight path.

Then in the code block `'export class MapComponent implements OnInit {}'` the initialization of some important things happen:

1. The lists for trackers to hold the names of cranes in the Trackers collection of MongoDB is initialized (as an empty list).
2. The same happens for the list of transmissions that will hold the all the data points for the selected crane in the web app.
3. Then the constructor for the `CraneService` actually activates this service to reach the Flask app on the Gunicorn WSGI web server.
4. The `getTrackers()` function gets initialized in the Angular `ngOnInit()` function with the code from the `getTrackers()` function definition below to enable the retrieval of the tracker JSON documents from the Tracker collection in the Crane database in MongoDB.
5. Then the `getTransmissions()` function definition is declared to enable get the transmission JSON documents from the Transmission collection in the Crane database in MongoDB.
6. After that only 4 function declarations remain but they have no relation with MongoDB:
 1. the function `createMap()` to create a topographical base map to plot the data on.
 2. the function `createPointLayer()` to plot the data points of the crane locations from the transmission list on a map in OpenLayers;
 3. the function `createLineLayer()` to plot a vector layer that draws lines between the data points;
 4. the function `animateRoute()` to show an animation of the flight path of the crane.

map.component.html --> What is MongoDB related?

The functions to get the data from the Crane database in MongoDB are specified in the map.component.ts file but these functions need an event trigger for Angular to let them work and this is where the HTML file comes in which displays a Settings menu that can detect clickable events to trigger a function to do something useful!

What does the Settings menu show for functionality?

1. A drop-down toggle menu is displayed with a Gear icon that will open and close the Settings menu on mouse click event detection when 'on mouse over' of the mouse cursor on the icon. How this works is explained in Part 4.
2. When the drop-down menu opens it shows 2 menu options:
 1. The 'Trackers in Database' option which shows a list of buttons with the names of the cranes retrieved from the Trackers collection from the Crane database in MongoDB.
 - When a crane name is clicked this event triggers Angular for the retrieval of all the transmissions in the Transmission collection from the Crane database in MongoDB which is explained below.
 2. The 'Animate Route' option which shows the Start and Stop option buttons for the flight path animation of the selected route which has no relation with MongoDB.

How does the Settings menu work?

1. In the HTML code there is an Angular for loop:
 - `<ng-container *ngFor="let tracker of trackers">`
2. Then there is the `<button>` element with the '(click)' specification to detect a mouse click on the button and then when detected, to trigger the function `getTransmissions()` to retrieve data from MongoDB:
 - `<button class="btn btn-white btn-block"`
 `(click)='getTransmissions(tracker._id.$oid);'>{{tracker.name}}`
 `</button>`
 - The specification of `{{tracker.name}}` assigns the crane name of the current tracker of the trackers list in the for loop to the button text field to display.
 - The parameter `'tracker._id.$oid'` is used to retrieve the Object ID string value for the 'tracker-id' of the current tracker in the for loop in a way that is already explained above in this section.
 - That tracker-id is then passed to the `getTransmissions()` function to retrieve all the transmissions with that tracker-id from the Transmission collection in the Crane database in MongoDB.
 - NOTE: this button element definition will make clickable buttons with the crane name as their button text appear in the settings menu to select a crane dataset!

What happens after the `getTransmissions()` function has the data?

1. After the data retrieval from MongoDB, the internal routing mechanism in the Angular app will take care the transmissions are plotted on a map in OpenLayers. The router mechanism does this by calling the required functions in map.component.ts.
 - How this routing mechanism works is explained a little more in Part 4 of the workshop!
2. Then the web app 'rests' until a next 'mouse click' event triggers another function call in Angular. These click events come from on-screen displayed clickable objects, either from the Settings Menu or from a clickable OpenLayers object, such as a '+' or '-' zoom button.

The end! Some Black Magic Explained! --> So, now you know, it's not 'Black Magic' anymore!

2.5.5 Summary of the Assignments

What do you have accomplished so far? You have learned about 3 important MongoDB subjects!

1) You have identified the position of the MongoDB datastore in the GeoStack.



Drawing 16: In the GeoStack datasets go through an ETL process to convert them from CSV or GPX files to JSON documents that are stored in MongoDB.

2) Then you did a 4 step ETL process where the datasets from raw CSV and GPX files have been:

1. Transformed to standardized JSON files in the folder /Datasets/JSON.
2. Modeled in MongoDB into 2 separate databases
 - Remember a Python class definition was used to define the database models.
3. Read in again from the JSON files.
4. Saved in the corresponding 2 databases in the MongoDB datastore.

3) Finally you created 2 databases in the MongoDB data store, each with 2 data collections:

- A crane database called 'Crane_Database_' which contains the crane datasets in 2 collections, we named the Tracker collection and the Transmission collection.
- A car trip database called 'Trail_Database_' which contains the GPX datasets of car trips (= car trails) coming from a GPS navigation device which contains the car datasets in 2 collections, that are named the Signal collection and the Trail collection.

2.6 A Sneak Peek at MongoDB in the Course

In the full course you can dive deeper into data handling with MongoDB in which you will learn for instance how to add other crane or trail datasets to the MongoDB databases with a supplied Python script or how to remove datasets you don't need anymore from the command line or with the database management tool MongoDB Compass (MongoCompass).

If you would like to do that in and after the workshop too, then learn more here:

1. Read the cookbook 'Creating the GeoStack Course VM' in:
 - section 5.4.2.4 'Managing the MongoDB Databases' on how to remove (drop) an obsolete database from the MongoDB datastore from the command line;
 - section 5.4.2.5 'Automating the dataset import process' on how to easily import new datasets and dataset selections as new databases in MongoDB with a Python script.
2. Read the course cookbook 'Data modeling in MongoDB' on how to work with MongoDB, data modeling, creating search indexes on fields in collections and loading data.
 1. Find this cookbook in the GitHub repository GeoStack-Course in the folder: /Building-the-VM-from-scratch-using-the-manuals/2.Cookbook-Data-Modeling-in-MongoDB
 2. Read chapter 5 'Validating using MongoCompass' on how to use MongoDB Compass and notice you can delete an obsolete database by selecting the database line in the list of databases and then clicking the Garbage Bin icon at the end of that line!
3. Learn more with the MongoDB University tutorials (!): <https://university.mongodb.com/>

3 Workshop - Part 3 – Data Usage: Middleware

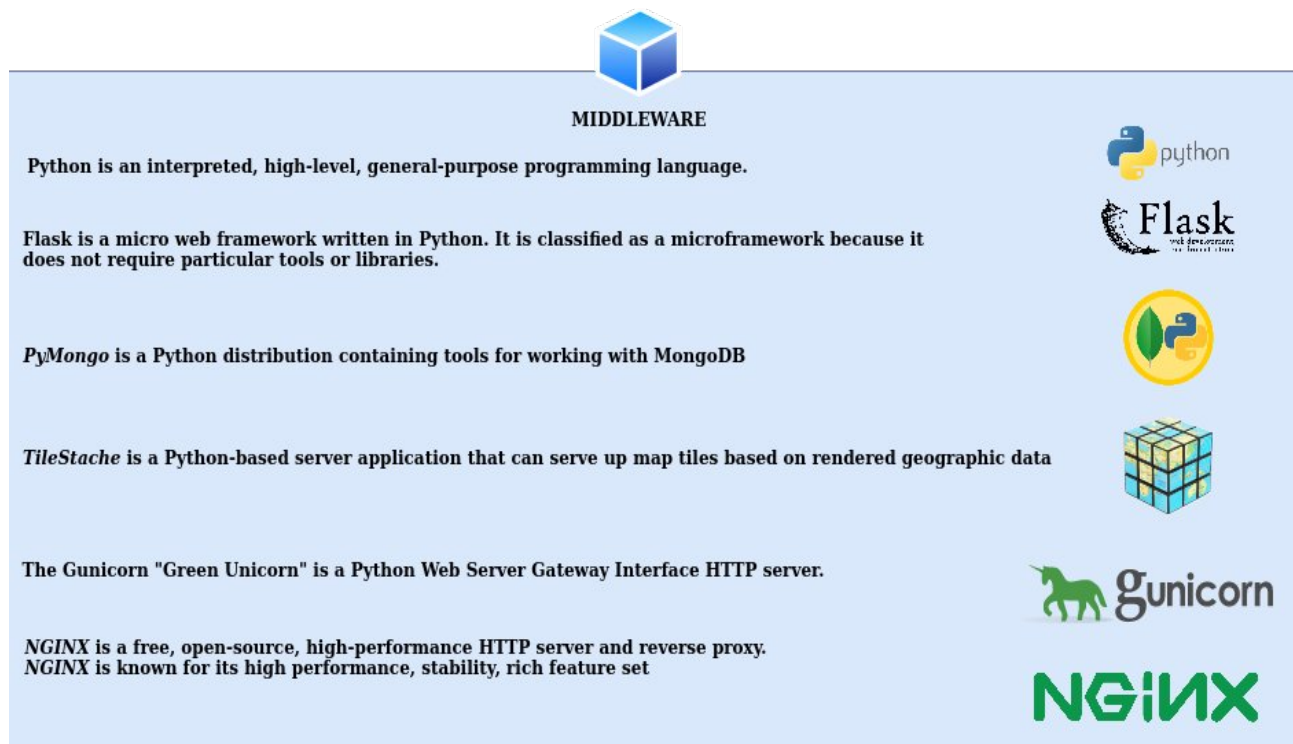
Goal: setting up the 2 middleware web services and the 2 web servers of the GeoStack on the server to be able to get data from the data stores and serve them to the simplified 2D Map Viewer web application that will also be run from the web server.

3.1 GeoStack Middleware – Web Services Explained

3.1.1 Development Environment for the GeoStack Middleware

To get to the data stores of the digital topographical maps and the geospatial datasets in our software architecture of the GeoStack 2 web services and 2 web servers are needed to use them.

Let's take a look at the components we need in the server infrastructure to make it work.



The PyMongo documentation is here: <https://pymongo.readthedocs.io/en/stable/>

- Tip: in the workshop the use of PyMongo is limited but in the course and for real use it is the preferred way of interacting with MongoDB so read a little bit more about it later!

3.1.2 Sketching the Software of the GeoStack Middleware

The 2 middleware web services we need are:

1. A web application programmed in Python with the Flask micro web services framework that offers an Application Programming Interface (API) as a web service that the 2D Map Viewer web application can use to request data of the datasets in the MongoDB datastore from the API of the mongod database server.
 - Note: it is important to remember that this Flask web application will hold the 'payload' source code that implements the 'business logic' to get, process and return the data from the datasets that the 2D Map Viewer web application needs for it's visualization!
 - Also remember that Python web applications use the Open Standard 'whiskey' protocol for the Web Server Gateway Interface (WSGI; pronounced as 'whiskey') which means we need a Python WSGI web server to run the Flask application.
 - Read more here: https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
2. A tile server application programmed in Python that is called TileStache that provides a web services API to serve the map tiles of the digital topographical map from it's cache folders back to the requesting web browser to let the 2D Map Viewer web application render the map tiles to display the assembled map.
 - Note: a tile server is smart software because it has a cache folder in which it stores map tiles that have been previously requested by web applications. Only when the tiles are requested that are not in the cache folder yet, the tile server will request a web map server to generate new map tiles to add to it's caching folder.
 - Also remember a tile server uses a separate cache folder for each zoom level (= map scale) that a web map service can provide. For instance on OpenStreetMap.org there are 19 zoom levels to zoom in on a map to see more detail and that corresponds to 19 cache folders in TileStache.

The 2 web servers we need are:

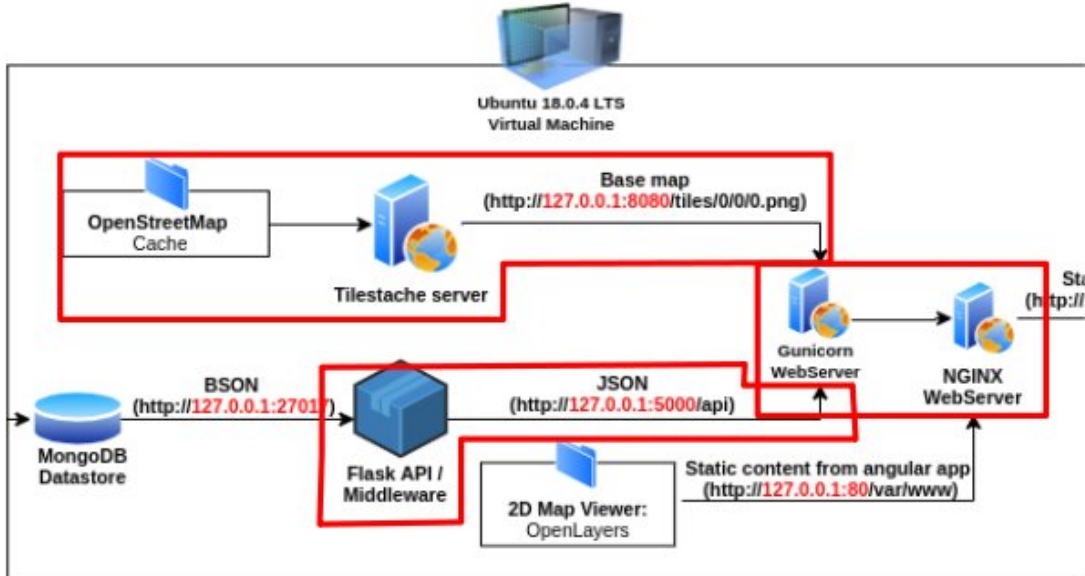
1. Gunicorn as the Python WSGI web server for the Flask web application to provide the Flask-API as a micro web service.
2. NGINX (pronounced as 'Engine X') as the HTTP webserver for 2 tasks:
 1. Run the 2D Map Viewer web application over the Open Standard of the HTTP protocol.
 2. Work as an HTTP to WSGI bridge web server to relay the HTTP requests from the 2D Map Viewer web application to either the TileStache WSGI tile server for map tiles or to the Gunicorn WSGI web server for the Flask app to get geospatial data from MongoDB.

Scope limitation: in the workshop we let you only check if the static index.html start page from the NGINX webserver works as a test to show how things should work. To keep things simple in Part 4 we will let you run the 2D Map Viewer web application there directly from the Angular Development web server that is installed as a development tool to let you see code changes in real-time!

3.1.3 Data usage – The URLs for the Middleware Web Services

In the previous section we sketched the main software components of the middleware in the GeoStack being 2 web services provided by the TileStache tile server and the Flask micro web service application with Gunicorn as their WSGI web server and NGINX as the HTTP web server.

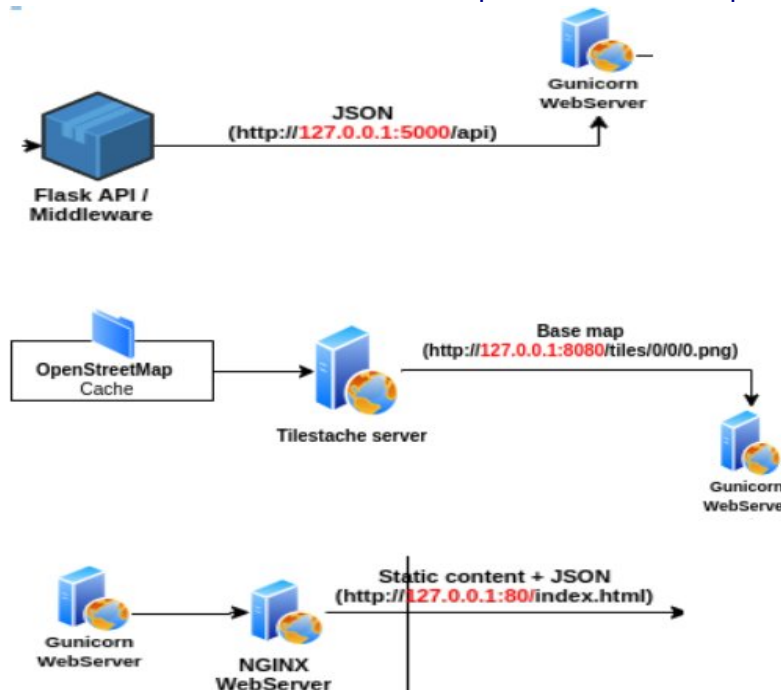
- For the workshop Tilestache gets the requested map tiles from its cache folders (OpenStreetMap Cache folders) and if the tiles are not present there yet, we use an online connection to the OpenStreetMap.org web map server to get the missing map tiles.
- The datasets for the workshop were imported in MongoDB in the previous assignments.



Drawing 17: Positioning of the Middleware in the GeoStack

You will learn to program the URL access for the middleware of the GeoStack in the workshop version for the main 3 connections you need to access as a developer (spoiler alert: you will configure Gunicorn access in the NGINX configuration file, so it's a little 'hidden'):

1. Flask API: accessible via the URL: "<http://localhost:5000/api>"
2. TileStache tile server: accessible via the URL: "<http://localhost:8080/>"
3. NGINX web server: accessible via the URL: "<http://localhost/>" or "<http://localhost:80/>"

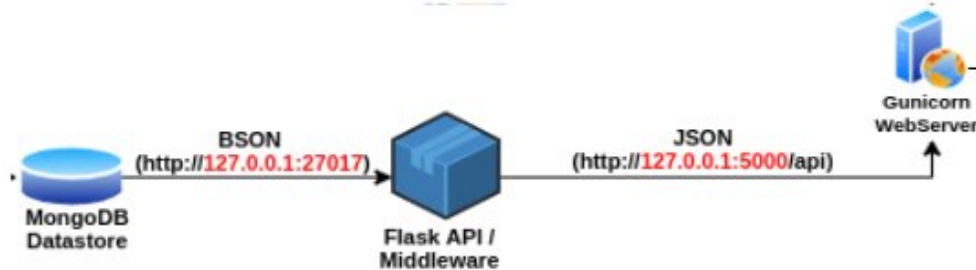


3.2 Flask API – Micro Web Services in Python

3.2.1 Flask API – How Data Queries work

The Flask app is the beating heart of the GeoStack because it contains the ‘payload’ software with the code logic of the Python micro web service web application for the processing of the datasets.

The Flask (WSGI) web application is providing a web services Application Programming Interface (API) that can be used by a front-end web application like the 2D Map Viewer to send an HTTP request to get data from a MongoDB database and process that data before it is send back to the web application that runs in the web browser via the Gunicorn web server for WSGI apps.

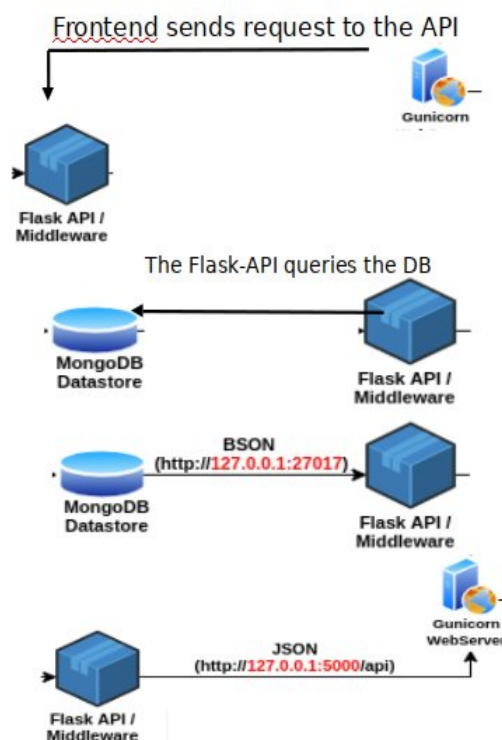


Such an HTTP request consist of an URL (= web link) that contains a query part and a value part for which the query has to search for all matching data records.

An example of such an URL is as follows: http://localhost:5000/api/get_all_trackers/{trackerId}

- ‘get_all_trackers’ is the query part to route to the function and ‘{trackerId}’ is the value part.
- Remember this is just the way of URL formatting for routing an HTTP request to the corresponding function in Flask web applications to handle that request.

In search engines you may see a different but similar query formatting which would look something like: http://search_engine.com/search?query=get_all_trackers&value=trackerId



It's a simple 3-step process:

1. First the Flask app converts the HTTP formatted data query request in an URL to API format the MongoDB datastore requires for data search queries.
2. Then MongoDB sends the requested data back to the Flask API in the BSON file format (Binary JSON).
3. Finally the Flask app converts the BSON data back to JSON, then performs the required data processing and then sends the data back over it's API to the web server.

3.2.2 Flask API – How the Flask App works

In our case the Flask-API is defined in the Python web application 'app.py' and follows the programming standard of the Python-Flask Framework.

- The app.py file contains all the queries for and connections to our MongoDB datastores.

Scope limitation: in this section some basics are explained to help reading the Python source code in the app.py file for the programming assignments.

The full explanation on how to program a web services API in Flask is in manuals of the GeoStack Course.

An example of such a database connection for MongoDB is shown below.

A MongoDB database connection consists of 3 parts:

- A PyMongo instance. (Red)
- The instance of our Flask Application as first parameter. (Blue)
- The connection string as 2nd parameter. (Green)

```
crane_connection = PyMongo(app, uri="mongodb://localhost:27017/Crane_Database")
```

One of the data queries for MongoDB as defined in the app.py file is shown below.

Such a MongoDB query consists of 3 parts:

1. The database on which the query is executed (Red).
2. The collection on which the query is executed (Blue).
3. The field names and values on which the query has to search (Green). In this example the field name (and thus it's value) on which the query has to search is the field name "_id".

```
query_result = crane_connection.db.tracker.find({"_id": ObjectId(id)})
```

3.3 Assignment 1 – Programming the Flask API

In the folder 'Part-3-Data-usage/Sourcecode/Assignment' is a folder called: 1.Flask-API.

- In this folder is the app.py file which contains the first 5 assignments for the Flask API:
 1. Create a database connection for the Trail database in MongoDB with car routes.
 2. Create a database query to get one JSON document of a trail (for the route name).
 3. Create a database query to get a specified amount of signals (= locations) for that trail.
 4. Create a database query to get all the crane Trackers from the crane database.
 5. Create a database query to get all the Transmissions for that GPS tracker.
- Notice for Part 3 the provided databases 'Trail_Database' and 'Crane_Database' are used!
- Note: in Part 4 the web app only focuses on visualizing the cranes! The car routes were deliberately left out for you to experiment in Angular which is why you still find the Trail and Signal programming assignments here so you know how to get the data from MongoDB!

After completing and saving one ore more assignments, you can run the desktop shortcut called: "Flask-API-assignment" as shown below, which will check your answers.



*Drawing 18: Dekstop
Shortcut for the Flask
API assignment.*

If one or more assignment(s) are incorrect it will be shown in the terminal, as shown here:

```
File Edit View Search Terminal Help
assignment 1 is incorrect
assignment 2 is incorrect
assignment 3 is incorrect
assignment 4 is incorrect
assignment 5 is incorrect
geostack@geostack:~$
```

*Drawing 19: Example of terminal output if the checking
script reports errors.*

If all the assignments are correct it will be shown in the terminal as in the image below.

```
File Edit View Search Terminal Help
Good Job, you have completed all the assignments. If you navigate to the URL: 'h
ttp://localhost:5000/api/' you can see the results.
geostack@geostack:~$
```

Note: notice in the terminal output there is a web link you can use in your web browser to look at the result of your successfully completed assignments!

- Tip: if you press the keyboard shortcut 'CTRL + Left mouse click' on the URL in the terminal, the default web browser is started to show the results.

3.4 TileStache – Tile Server

3.4.1 TileStache – How the tile server daemon works

TileStache is a Python server program (a daemon!) to serve map tiles which are small raster image files in the PNG file format with parts of a digital topographical map drawn on each tile.

How map tiles work

How it works is that in a web application you have a view window that has a blank canvas as a background to draw a map on. The four corners of this view port window correspond to the GPS coordinates of the locations as seen from a helicopter view at a height of the chosen zoom level.

- The zoom level is like the height at which the helicopter hovers above the map in the center of the view port window.
- The detail you see when you look down is then like the scale of a map.
For example, for a map on a scale of 1:250,000 you would see less detail because 'the heli flies pretty high' compared to a detailed map of 1:10,000 because 'the heli flies real low'.

The view port window of the web application is divided into a tile grid of rows of small squares which are the map tiles. These map tiles are then requested from the tile server.

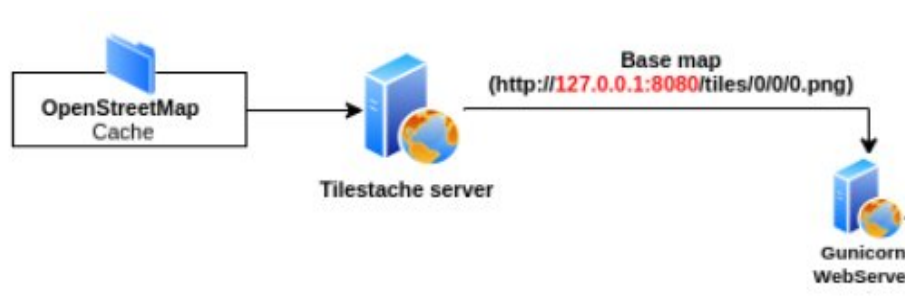
- If the map tiles are already present in the cache folders of the tile server, they are returned to the web application. The web browser will then put them together again in rows in the right sequence to display the topographical map in the view port window.
- If the map tiles are not present in the cache folders, the tile server will request the missing map tiles from a locally hosted or online web map server as set in the configuration file.

Positioning the tile server

The TileStache tile server is a Python web server which serves a cache for our map tiles in the PNG raster image file format.

In the case of the workshop our tiles are obtained from the online OpenStreetMap.org Web Map Service (WMS).

- In the GeoStack Course we are going to create those map tiles ourselves using the RAW OpenStreetMap data that is locally stored in the spatial database PostgreSQL (PostGIS).



Drawing 20: Position of the tile server in the GeoStack.

The tile server receives map tile requests from the front-end web application (= 2D Map Viewer).

These are the so-called {Z}{X}{Y} requests and of course they consist of the following 3 parts:

1. The Z-axis of the map. (Zoom level)
2. The X-axis of the map. (The Latitude east/west coordinates)
3. The Y-axis of the map. (The Longitude north/south coordinates)

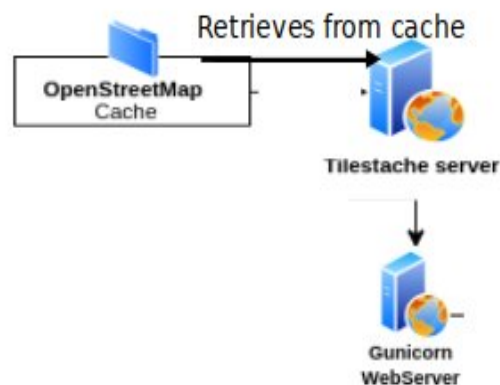
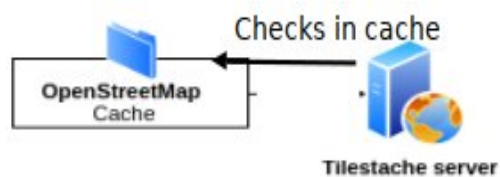
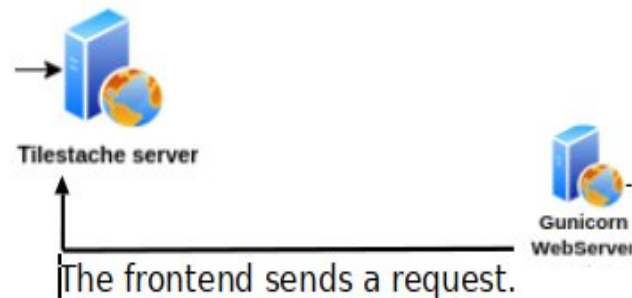
A value of {0}{0}{0} for a {Z}{X}{Y} request will return a small 'World Map' view by default and it is a good way to test if a configured Web Map Service connection works.

- Note: learning how to make such a request is in the next workshop assignment.

The TileStache tile server then checks if the map tile PNG file that has the {Z}{X}{Y} value from the request URL already exists in the cache, see the second illustration below.

If the tile already exists, it is retrieved from the cache and send to the frontend, see the third illustration below.

The whole process of handling a HTTP request for a map tile from a web application is shown in the illustrations below when a map tile is already present in the cache folder:



If the tile does not exists in the cache folder of the tile server, the following will happen:

1. In the workshop the map tile is requested from the online OSM WMS.
2. The map tile file is generated by the WMS.
3. The map tile file is saved in the cache folder of the tile server in the corresponding zoom level cache sub-folder.
4. The map tile is send to the front-end web application (= 2D Map Viewer in the workshop).

3.4.2 TileStache – How the tile server configuration works

The TileStache tile server has 2 main components we need to know a little bit about:

1. A configuration file named: configuration.cfg
2. The cache in which the generated map tiles are stored, which is a folder structure.

Cache specifications

The configuration file contains the specifications related to the configuration of our cache.

```
"cache":{  
  "name":"Disk",  
  "path":"cache",  
  "umask":"0000"  
},
```

The cache configuration consists of 3 specification lines with a key:value pair that define the cache:

1. name: the name of the cache which is 'Disk'.
2. path: the PATH location of the cache folder named 'cache'.
3. umask: the coding of the Linux file access permissions for users to the PNG files stored in the cache which is '0000' in this example.
 - The value of '0000' as umask access code means that there is full access to the files for all users).

Configuration file --> configuration.cfg

The TileStache configuration file configuration.cfg also contains 'entries' which define the Web Map Server location from which we retrieve the Digital Topographical Maps.

- An Open Content WMS example is the OpenStreetMap.org WMS.

To learn how to configure the connection between TileStache and a WMS that uses an API key an example is included for the Thunderforest WMS for which you can sign up for an account to get an API key for a free hobby project at <https://www.thunderforest.com/>

- Such an API key is passed in the query part or query string of the request URL (web link).
- Both commercial and free web map services might use an API key for which you have to sign up to get an account to get a personal or corporate API key to either get a subscription or just to regulate the number of free queries because bandwidth costs them money.
- For this workshop we picked Thunderforest just as an example but you can also use any other WMS to practise, like for instance Google Maps or Mapbox (from the developer of the JavaScript spatial framework Leaflet) and get an API key from them.

About Map Scales and Requesting Map Tiles

It is good to have a little background on how requesting map tiles works because your are going to learn how to program this in the workshop. It's about tiles per zoom level and map scale ratios.

- A map tile is a PNG file that typically has a size of 256x256 pixels and they are generated and stored per zoom level that the web map service offers.
- A zoom level is similar to a map scale ratio like we know from paper topographical maps.
 - OSM has 19 zoom levels. Some web map services may offer up to 24 zoom levels.
 - Definitely read more here (!): https://wiki.openstreetmap.org/wiki/Zoom_levels
- The map tiles are requested per zoom level for a GPS location which is coded like {Z}{X}{Y} in an URL, where Z = Zoom level and X and Y are the latitude and longitude of the GPS location for the center of the map view port window in the web application for which all the map tiles are requested to assemble these tiles as a topographical map in the web browser.

In the image below you can see a map example of the map type 'landscape' that was retrieved from the Thunderforest WMS with a very low zoom level.

You could see it as a map with a scale ratio of '1:World-View' and it is the result of an URL with a {Z}{X}{Y} request in it with the value {0}{0}{0} as you will learn in the next workshop assignment.



Drawing 21: Example of map type 'landscape' retrieved from the WMS of Thunderforest.

An example of an entry in our TileStache configuration file related to the Thunderforest WMS, is shown in the image below.

This entry in the configuration file consists of 5 specification lines with a key:value pair:

1. The name of the TileStache entry: you can decide yourself what you want the name (Red) of the entry to be but make sure it's logical and related to the WMS for easy remembering!
 - Here we chose 'landscapemap' as the entry name as an easy reference to the name of the map type style offered by the WMS.
2. The allowed Origin Type (Blue): this is related to the HTML-headers in the HTTP requests. (* means every type of HTTP request is allowed)
3. The provider sub-entry (Green): this is where you declare the configuration for the WMS provider you are adding and it has 2 specifications for the name and the URL.
4. The provider name (Purple): this is where you declare the entry type which is 'proxy' in this case.
5. The URL (Orange): since the type of this entry is a proxy (which results in name substitution) for a WMS specified with a web link, we need to declare the web link with the URL on which the digital topographical web map service is located and add the query part template for requesting map tiles to it.
 - In the URL we also declare the {Z}{X}{Y} notation as the query template for the requested tiles. (Yellow)

```
"landscapemap": {  
  "allowed origin": "*",  
  "provider": {  
    "name": "proxy",  
    "url": "https://b.tile.thunderforest.com/landscape/{Z}/{X}/{Y}.png"  
  }  
}
```

Drawing 22: Example in the TileStache configuration file for the map type 'landscape'.

3.5 Assignment 2 – Configuring TileStache

3.5.1 Assignment – Completing the TileStache Configuration File

For the configuration assignment of the TileStache tile server the steps are as follows:

1. In the folder: “Part-3-Data-usage/Sourcecode/Assignments”, you will find a folder called ‘Tilestache-Server’.
 - In this folder you will find a file called ‘configuration.cfg’ that contains the 2 assignments related to configuring the TileStache tile server, so open this configuration file in the Atom programming editor to specify the correct configuration.
 1. Specify the location of the TileStache cache folder to store the map tile PNG files.
 2. Specify URL entry for the OpenStreetMap.org Web Map Server.
 - Notice there is an URL specification for the Thunderforest WMS too as an example to learn how to work with a WMS that uses an API key for its users for which you need to create an account first of course which is free by the way.
2. After completing and saving the configuration assignments in the configuration file, you can run the desktop shortcut ‘tilestache-tileserverserver-assignment’ to check your answers.



3. If one or more assignment(s) are incorrect it will be shown in the terminal like this:

```
File Edit View Search Terminal Help
assignment 6 is incorrect
assignment 7 is incorrect
geostack@geostack:~$
```

4. If all the assignments are correct it will be shown in the terminal as in the image below.
 - Tip: when you press the keyboard shortcut ‘Ctrl + Left mouse click’ on the URL in the terminal, the default web browser set in Ubuntu will open with this web link so you can see the result.

```
File Edit View Search Terminal Help
Good Job, you have completed all the assignments. If you navigate to the URL: 'http://localhost:8080/openstreetmap/0/0/0.png' you can see the results.
geostack@geostack:~$
```

Drawing 23: Notice the coding of the {Z}{X}{Y} request with the value {0}{0}{0} in the URL.

3.5.2 Challenge – Read up on Map Styles and Vector Tiles

Notice if you look at the maps from different Web Map Services (WMSs) like OpenStreetMap, Google Maps, Bing Maps, Thunderforest, Mapbox etc., that the way how their maps looks differs significantly between these different providers of digital topographical maps.

- These differences in 'look and feel' are similar to those in printed topographical maps from different vendors of paper topographical maps.
- For the workshop we won't go into this subject of map styling and in the course we will look just a very little bit at it.
- Still, it is good to know you can adjust the look-and-feel if you like. Compare it to CSS files to adjust the way how HTML web pages will look.

That said, it's not easy! So, to get a better feel for how the map styling works this challenge is to take an extra hour to read about how things work in the software ecosystem of OpenStreetMap.

The cookbook recipe to keep in mind while reading is something along the following lines:

- The raw OSM data is vector data in a 'spatial version' XML style that is stored in a spatial database like PostgreSQL + PostGIS extension or MariaDB Spatial etc.
- Then, when a map tile is needed, you need a map renderer like Mapnik that is used by OSM to get the XML data from the spatial database and converts (renders) it into the raster images for the raster image map tiles in the PNG file format.
 - Instead of Raster Tiles a newer development is Vector Tiles for higher quality visualization and less bandwidth. TileStache can serve them.
 - Read more about vector tiles here: https://en.wikipedia.org/wiki/Vector_tiles
 - We don't use vector tiles in the workshop or course but it is good to know they exist!
- To do the rendering for the desired map style, the map renderer (= Mapnik in case of OSM) needs to have a 'style sheet' like specifications of how to draw the map tiles.
 - For Mapnik these drawing style specifications go into the style.xml configuration file.
 - So, how map styling works depends on the map renderer software used for the WMS!
- A digital topographical map has a base map of map base layers like streets and buildings and it can have multiple transparent map overlays to visualize geospatial datasets on top
 - Therefore a programmer has to choose which base layers to show in the base map and in which map style. An example is the color of country roads: white or gray or...?
 - An example of a map overlay is a set of Points of Interest (PoI) like the locations of monumental buildings to be visualized with a monument icon.
 - To get the idea go to: <http://tile.openstreetmap.fr/>
 - In the blue Settings menu at the top right you can select different map styles for the base map (= base layer) and also select a few map overlays for additional geospatial datasets that will then be plotted (= overlaid) on that base map.

Reader's Guide on Digital Topographical Map Styling – Read, Scroll and Click an hour or so!

- Look at Mapnik at <https://wiki.openstreetmap.org/wiki/Mapnik> and <https://mapnik.org/>
- Then read about OSM's 'Slippy Maps' here: https://wiki.openstreetmap.org/wiki/Slippy_Map and here: https://en.wikipedia.org/wiki/Tiled_web_map (= slippy map in OSM terminology).
- For more different OSM styles explore: <https://github.com/mapnik/mapnik/wiki/StyleShare>
 - For example the Dutch Topographic Map style is here: <http://www.openstreetmap.nl/>
 - Programming examples for Dutch public maps: <https://github.com/geo-frontend/nlmaps/>
- Another example of a standard for map layer styles are Styled Layer Descriptor (SLD) files (see: https://en.wikipedia.org/wiki/Styled_Layer_Descriptor) with Symbology Encoding (SE) for the specification like it is used by the open source GeoServer WMS.

3.6 NGINX – Web server

3.6.1 Terminology Explained – NGINX as a web and proxy server

What is NGINX and what does it do in the GeoStack?

The NGINX web server (<http://nginx.org/en/>) is an HTTP web server and a reverse proxy server and in the GeoStack we use NGINX in both roles.

1. As a web server NGINX is used to serve the static HTML, CSS and JavaScript files that compose the 3 front-end (= end-user) web applications (Dataset Dashboard, 2D Map Viewer and 3D Map Viewer) to the web browser (HTTP) that we use in the GeoStack Workshop and Course.
2. To proxy the browser requests as a relay server (see the role as an application-level gateway server) to the upstream Unicorn WSGI web server that will relay the map data requests in return further upstream to the origin server TileStache and Unicorn will relay the datasets requests to the Flask app that accesses MongoDB as the data origin server.

Reader's Guide to NGINX configuration terminology

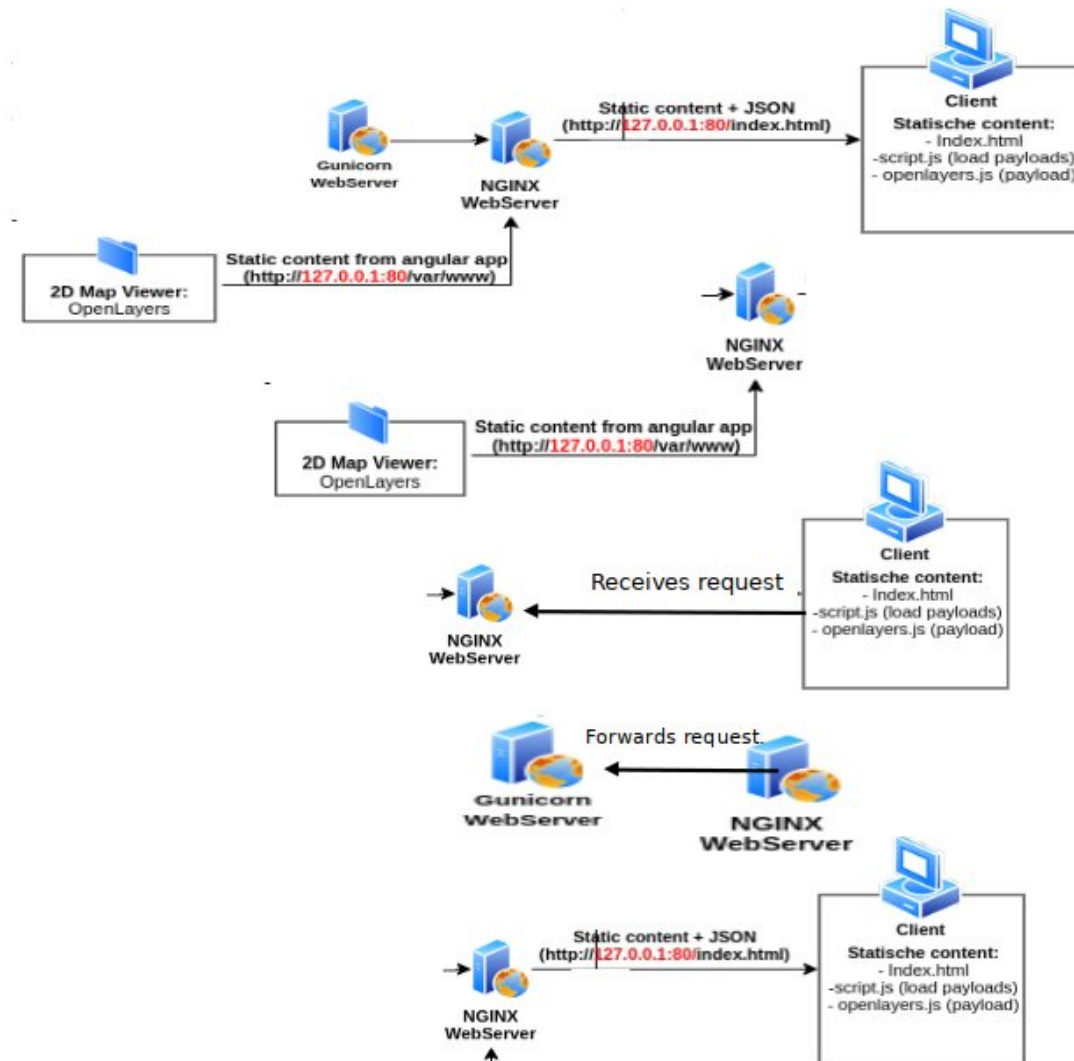
- You need to know what a reverse proxy server is and what an upstream server is for the next assignment to be able to get the NGINX configuration file programmed right!
- Reverse proxies can hide the existence and characteristics of other (origin) servers.
 - This means something like the web application that runs in the browser is not aware of the existence of a tile server and a spatial database server behind that or a WSGI webserver that runs a Flask app with a database server for datasets behind it.
 - The data server that is furthest upstream in the chain to finally handle a data request is the server that actually serves the data back as the result to the request and this server is then aptly named the 'origin server' as it is the source of the data that is provided to the requesting web application. All other servers in between are just 'relay servers' that pass the request further upstream while converting the request to fit the next web service API format like from HTTP to WSGI to the MongoDB query format.
- Now first read this wiki page: https://en.wikipedia.org/wiki/Reverse_proxy
- Then read this wiki page: https://en.wikipedia.org/wiki/Upstream_server
 - Example: in the workshop we use the online OSM WMS so that is the origin server for the TileStache tile server if a map tile is not in it's local cache folder which makes 'origin server' a relative position in a software architecture as what is considered a data source is a point of view perspective.
 - In this example the 'furthest upstream origin server' is the spatial database server behind the OSM WMS because that is where ultimately the map tiles originate that TileStache stores in it's map tile cache folders to serve up as the response to the data request for map tiles.

3.6.2 Positioning NGINX in the GeoStack

The NGINX web server is the place where all the components in the GeoStack come together because it acts as a web server and as the single point of contact for the web browser that loads the web application files from the server and acts as the central communication hub (proxy) to handle all HTTP data requests send by the web browser and the received responses from the upstream servers.

The NGINX web server functionality is shown in the image below and the image reads as follows:

1. NGINX contains the static files for our web application programmed with TypeScript + Angular called from a HTML file (index.html) just as the CSS file (style.css) for some styling.
2. NGINX receives requests send by the Client (the user's web browser).
 - Two examples of such requests are:
 1. The location of a Crane.
 2. All the GPS Routes (Trails) in the MongoDB datastore.
3. NGINX forwards the requests to the corresponding component in the GeoStack.
 - In the case of the workshop, these 2 components are:
 1. The TileStache tile server: for retrieving the topographical base maps.
 2. The Flask app: for retrieving the data from the MongoDB datastores.
4. NGINX receives the data from the corresponding component in the GeoStack and sends the data, together with the static files from our Angular web application, back to the Client.



3.6.3 NGINX configuration for the Flask API explained

The NGINX webserver consists of 2 main components:

1. An HTML file, named 'index.html': this file is the web page that is shown when navigating to the URL: "http://127.0.0.1/" in your web browser. During this workshop we are not going to do anything with this file.
2. A configuration file, named 'nginx.conf'. This file contains all the servers and their locations of the GeoStack components.

The code which defines the Flask API for the upstream micro web service is shown in this image:

```
upstream flask-api { server localhost : 5000; }
```

In this 'upstream' statement the variable 'flask-api' is defined for communication with the upstream Gunicorn web server for Python WSGI web applications that runs on port 5000 on local host which is the URL <http://localhost:5000> which also means the URL notation <http://flask-api> can now be used as a short-hand synonym notation in the NGINX configuration file.

The following 3 specifications are needed to define an upstream server in the NGINX configuration:

1. A server name (Red): in this case the name of the server is going to be 'flask-api'.
2. The server location on which the server is running (Blue): in this case it's the local server (localhost).
3. The port on which the server is running (Green): in this case the Flask API is running (being provided) on port 5000.

The code which defines the location in the NGINX configuration where the Flask-API will be accessible, via the NGINX web server is shown in this image:

```
location /api/ { proxy_pass http://flask-api; }
```

The following 2 specifications are needed to define a server location:

1. The location statement defines the variable '/api/' (Red) which in this case is the Flask API that is available via the NGINX webserver on the URL: <http://localhost/api/> which is in full of course <http://localhost:80/api/> but because port 80 is the default HTTP port the port number can be omitted in the URL.
 - This 'location' specification makes the match with the API specification in the Flask application app.py where choosing 'api' as a prefix in the URL to call the API functions it will be easy to see in the @route decorator definitions for these functions which functions handle an HTTP request on the API and which functions do something else.
 - Say, app.py has a route decorator: @app.route('/api/trails/<id>', methods=['GET']) for an API function trails(), then the URL would start with: <http://localhost/api/trails/>
 - The URL would of course still need some completion for the data request part '<id>'.
 - Read the basics about Flask routing here: <https://pythonbasics.org/flask-tutorial-routes/>
2. The 'proxy_pass' declaration makes the HTTP requests that are received which have '/api/' in the URL (Red) will be proxied to the specified upstream server <http://flask-api> (Blue).
 - Notice this 'location' statement actually defines the proxy role of the NGINX web server as an HTTP – WSGI Bridge server towards the Gunicorn server.

When proxying (redirecting) an URL, the HTTP request is forwarded (relayed) from one URL to another URL by URL substitution of the 'only externally accessible' URL that the web app uses to address the NGINX web server over HTTP with an 'only internally accessible' URL that the NGINX web server uses to relay the HTTP request to the Gunicorn web server that runs the Flask app.

In the case of the Flask API to access the app.py application on the Gunicorn web server it works as follows:

- When you navigate to the external URL: **"http://localhost/api/"**, this URL will be substituted by the internal URL : **"http://localhost:5000"** as the proxy URL.
- This will result in the rerouting of the HTTP request to the Flask API via the NGINX web server with as goal, to leave the 'static' file workload and the security related settings to the NGINX web server with ModSecurity and transfer the 'dynamic' data handling workload to a more suitable server upstream.
- You can see proxying as 'URL forwarding' or 'URL relaying' or 'URL switching' or 'URL redirect'.

For more information related to the workload and security of the web server you should read the GeoStack Course cookbook 'Creating A secure NGINX webserver with ModSecurity'.

Proxy flow

It is good to notice there are 3 important 'redirects' in the client (web app) – server (web server + flask app) that have to be specified in the software architecture for the 'proxy flow' of HTTP data requests and data provisioning for the entire information system to work.

- Notice the Python Flask application app.py specifies the API name of the micro web services API which is in this case '/api/' and it is part of an URL for an API HTTP request.
1. The Flask app also uses the @route decorator declarations to specify the routing (= internal redirect in the Flask app) of an API call by an HTTP request URL to the function in the Flask app that will handle that HTTP request.
The @route decorator defines the function that will handle the HTTP request as the URL end point in the Flask app.
 2. The NGINX server is the 'man-in-the-middle' server between the web app and the Gunicorn web server that runs the Flask app by which it shields the Gunicorn web server from all external access.
 - To handle these incoming and outgoing 'redirects' as a proxy server the NGINX web server needs the two 'upstream' and 'location' declarations in its configuration file.
 3. The web app (2D Map Viewer) has two files where the proxy configuration specified:
 1. In the file proxy.conf.json in the web app's root folder 'map-viewer' the '/api/' mapping is specified for the web services.
 2. In the file map.component.ts the 'CraneService' is specified to make the Angular routing mapping from the function 'getTransmissions()' to the 'CraneService'.
 - Note: which files make up an Angular app, what routing is in Angular and how the flow of events goes that makes the web app work is explained in Part 4 'Data Visualization' of the workshop.

Ports for Network Connections

The server processes provide their network services on standardized port numbers but you can easily reconfigure them in their configuration files if they conflict.

1. NGINX offers its HTTP network service on port 80.
2. Gunicorn offers the API for the Flask app app.py on port 5000 and the TileStache tileserver on port 8080.
 - An example of a port conflict is in the GeoStack Course where the Cesium Terrain Server must run on port 8080 and because of that requirement we simply reconfigure Gunicorn to run TileStache on port 8081.

3.7 Assignment 3 – Configuring NGINX

3.7.1 Assignment – Completing the NGINX Configuration File

The assignment is to complete the nginx.conf configuration file to get the proxy server role of NGINX specified to reach the upstream webservices of TileStache and the Flask app on their API's and of course to get the NGINX web server running.

- Scope limitation: to keep things simple in the workshop we will not host the simplified 2D Map Viewer web application on NGINX but just check if it's up and running by testing if the the NGINX web server returns it's default start page file index.html.
- Tip: first take 5 minutes to just scroll the configuration file from top to bottom and read what the different configuration sections in this file are about.
The configuration file might look complex at first but don't worry because you read about how it works in the sections above and you read the terminology carefully! Yes, you did!

For the configuration assignment of the NGINX web server the steps are as follows:

1. In the folder: "Part-3-Data-usage/Sourcecode/Assignments", you will find a folder called '3.NGINX-Webserver'.
 - In this folder is the configuration file 'nginx.conf' which contains the 2 assignments to:
 1. Specify the 'upstream' server specification for the TileStache tile server.
 2. Specify the proxy configuration for the tile server API to access the 'openstreetmap' overlay to request map tiles from.
 - Notice the similar API specification for the 'landscapemap' overlay for the Thunderforest map tiles, the upstream specification for the Unicorn web server for the Flask API to access the Flask application app.py and of course the specification for NGINX to run itself as a web server to listen on port 80 to receive HTTP request for static files or API requests to redirect (proxy) to upstream servers!
2. After completing and saving on ore more assignments, you can run the desktop shortcut called: "nginx-webserver-assignment" as shown in the image below to check your coding.



3. If one or more assignment(s) are incorrect it will be shown in the terminal, as shown here:

```
File Edit View Search Terminal Help
assignment 8 is incorrect
assignment 9 is incorrect
geostack@geostack:~$
```

4. If all the assignments are correct it will be shown in the terminal as in the image below.
 - Tip: when you press the keyboard shortcut 'CTRL + Left mouse click' on the URL in the terminal, you will see the results in the web browser.

```
Good Job, you have completed all the assignments. If you navigate to the URL: 'http://localhost/tiles/openstreetmap/0/0/0.png' you can see the results of the Tileserver running behind the NGINX Webserver and if you navigate to the URL: 'http://localhost/api/' you will see the result of the Flask-API running behind the NGINX Webserver.

When you encounter the error: '502 Bad Gateway' you should restart the Tileserver by clicking on the shortcut: 'tilestache-tileservers-assignment' or 'tilestache-tileservers-solution'!
```

Drawing 24: This information message is displayed by the nginx.sh script which is run by clicking the desktop shortcut 'nginx-webserver-assignment'.

3.7.2 Assignment – Checking the End Result!

Finally check the results in your web browser by following the 2 web links in the terminal output you see in step 4 of the assignment in the previous section.

Check if the 3 middleware servers TileStache for the map tiles, Unicorn for the Flask API and NGINX for static content files work in the next 3 steps:

1. **Tile Server:** check if TileStache returns map tiles correctly as shown here in the image below and as instructed in the section above by entering the following URL in your web browser: <http://localhost/tiles/openstreetmap/0/0/0.png>
 - In the GeoStack Workshop you will see a small square with the 'World-View' map as shown in the image below because the HTTP request for the map tile is set to a Z/X/Y.png value of 0/0/0.png in the URL.

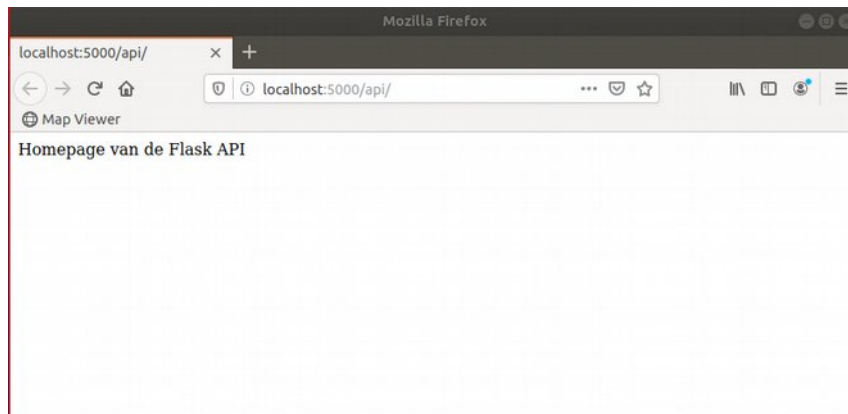


- If you don't provide the Z/X/Y parameters in the URL you will get an HTTP '500 Internal Server Error' in return so that's why you use 0/0/0.png to request the map tile.

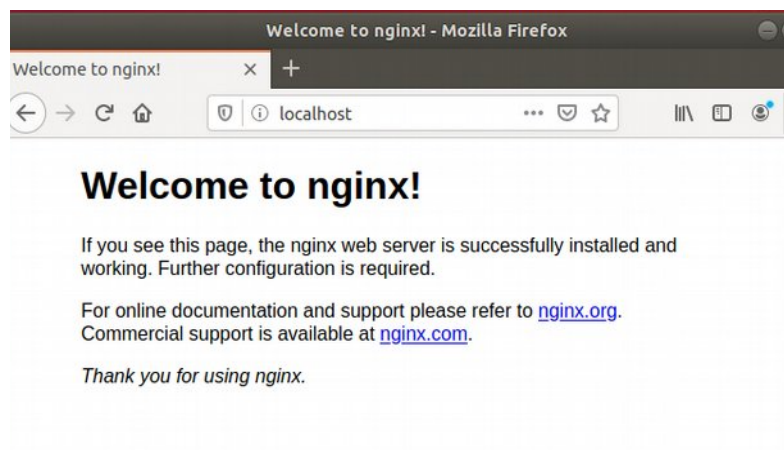
NOTE: if you think it's a nice idea to check if the tile server also works by trying your own Z/X/Y values as the zoom level and map coordinates to center the topographical map, then we agree!

- But..., it's never that easy! The X/Y values are not GPS coordinates! They are 'map tile' coordinates and because a map tile is a file the Y-value has to be a file name, so it's Y.png!
- So you need to learn some more but that is not really a subject for the workshop but for the GeoStack Course!
Still, about centering the map a little 'black magic' is explained in this GeoStack Workshop in section 4.2.18 'Scope Limitation --> Centering the Map at [0,0] explained!'
- If you want to work directly with a zoom level and GPS coordinates than you will do that in the web application and not in the TileStache configuration or in a map tile URL.
 - In an Angular web app with OpenLayers it will work just like on the internet and you will learn that later in the GeoStack Workshop.

2. Check if the Flask API is accessible by sending an URL as an HTTP request for a query.
 - The result should be the default index.html start page as shown in the image below.
 - Note: a workshop installation script installed the index.html file as the API home page.
 - Note: if both TileStache and Flask work this means the Gunicorn web server works too!



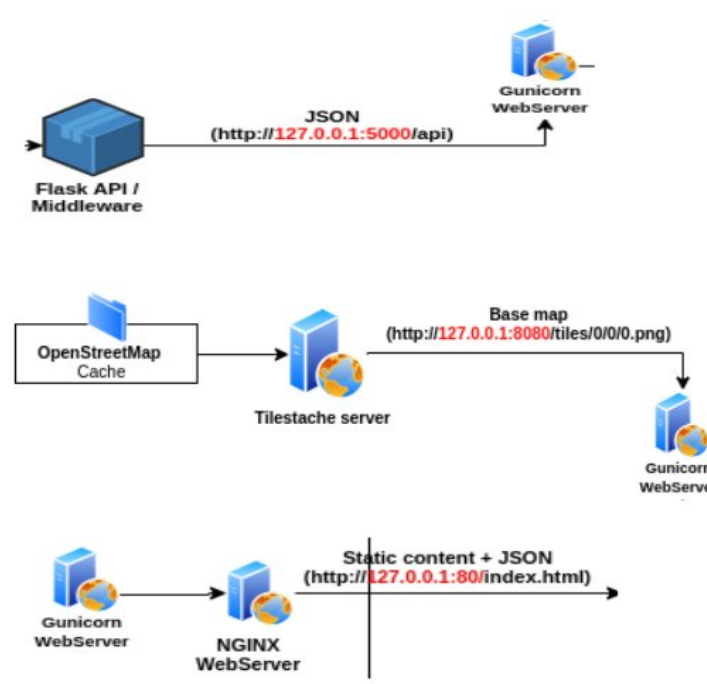
3. Check the NGINX web server to see if it serves static HTML files by requesting it's default start page 'index.html'.
 - Note: the index.html file is served automatically if you enter the URL 'http://localhost' as shown in the image below.
 - Note: the index.html file was installed from the NGINX Ubuntu package.



3.7.3 Assignment – Starting the Middleware Services

The working Middleware server environment that is required for web programming in the next chapter to create the simplified 2D Map Viewer web application and get it up and running now consists of the following 3 components as shown in the image below:

1. A working Flask app running on the Gunicorn web server presenting Flask API as a micro web service for requesting data from our MongoDB datastore using queries; shown as the top component.
2. A working TileStache tile server presenting it's own TileStache API as a web service on the Gunicorn web server for generating digital topographical base maps; shown as the middle component.
3. A working NGINX web server as a proxy server, behind which the Flask API and the TileStache API are provided by the Gunicorn WSGI web server to access both the Python applications app.py with the Flask app and TileStache and NGINX serves the static content files of the web application, all as shown in the bottom component.



These 3 components must be up and running as shown in the previous section 'Assignment – Checking the End Result!' before you start the next chapter 'Part 4 – Data Visualization'.

- If for some reason you did not get the 3 middleware components up and running in the previous programming assignments then study the solution scripts first and try again!

Assignment: the assignment here is to start the following 3 middleware services: Flask API, TileStache tile server and the NGINX web server if you did not get them up and running yourself because they are required to be able to program and run the web application.

- You can start **all 3 services at once** by just running the NGINX solution script by clicking the desktop shortcut: nginx-webserver-solution
- Note: the desktop shortcuts for the Flask and TileStache solution scripts are there to enable checking these individual assignments and to let you experiment with them separately. To check the NGINX assignment, Flask and TileStache must be running too of course and that's why you can use the NGINX solution script to start all 3 middleware services at once!

4 Workshop - Part 4 – Data Visualization: Web App!

In this last workshop part you will learn to program a simplified 2D Map Viewer web application!

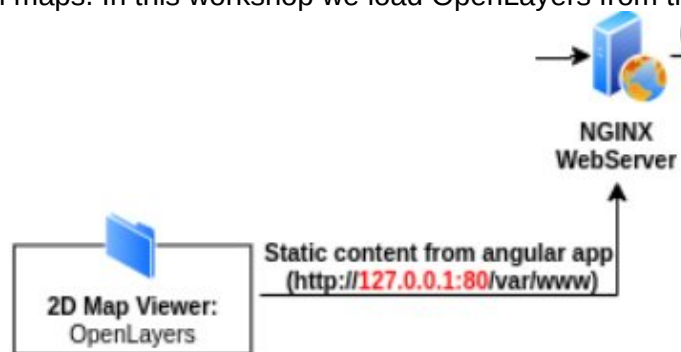
4.1 Web Application Programming – How does it work?

4.1.1 Web Server Hosting of the Web Application

Let's take a look at the web server side of things first to find out where files go and how it works.

The 2D Map Viewer web application consists of a set of at least 4 files:

1. HTML file: the web application part of the application, like the layout of the screen elements ('what-goes-where') and the HTTP communication with the web server.
2. CSS file: the design part of the application for the look and feel ('how-it-looks').
3. TypeScript file: the interactive part of the application to handle the user interface and data.
4. JavaScript file: the active 'OpenLayers' part of the application to handle digital topographical maps. In this workshop we load OpenLayers from the internet for simplicity!



Drawing 25: The web application is hosted as a collection of static content files in folder /var/www on the web server.

The file locations in the folders on the web server of the CSS, TypeScript and JavaScript files are declared in the HTML file and loaded and run by the web browser.

- The user only has to know the URL (web link) to the HTML file to start the web application.
- It is a good practice to separate the different type of files into their own folders but for the workshop we just put them all together in the /var/www folder to keep things simple.
 - What you would normally do is create a folder /var/www/2dmapviewer with subfolders ccs and ts because they belong together and then put the OpenLayers files in a separate folder /var/www/openlayers because it is a generic library to be used by other web applications as well.
- Notice once these files are coded they will not change anymore and therefore they are referred to as 'static content' files.
 - A web server like NGINX or Apache is designed to handle and serve up static content files to a browser real fast to render.
 - If you need dynamic content handling, like getting a digital topographical map for a certain rectangular area or selecting geospatial GPS tracker data from datasets in a data store then you need a web application running on a web server application like TileStache or Gunicorn (to run your Flask app) that can actually do something to get, process and return the requested data.
- To sum it up:
 - NGINX and Apache are for static content only. Requests for dynamic content are forwarded to other web servers upstream that can handle the data request.
 - Dedicated web servers like the TileStache web server for digital topographical maps provide a web service API to provide specific dynamic content.
 - Generic web servers like the Gunicorn web server to run Python scripts provide a web service API you need to program yourself to provide dynamic content as required.In other words: you can program any web application you like to handle your data.

4.1.2 Development Environment for the Web Application

The second aspect of the web application to look at is the (inter)active part because the HTML and CSS files have no source code to run as they are just declarative files that hold the rendering specification for the browser's HTML + CSS rendering engine about what-goes-where and how-it-looks in the browser window.

To handle the interactive part you would typically load a source code file in a scripting language that is programmed to do the typical web app things like:

- handling the graphical user interface;
- sending the HTTP requests for data (maps and geospatial datasets) to the web server;
- handling the returned data from the web server;
- activate the rendering of that received data for display in the web browser window.

Choosing the scripting language is something to think about because there are many scripting languages that can be used to program the (inter)active part for a web app, like the well known PHP programming language.

In our case we chose to stick to the JavaScript and TypeScript combo for good reasons!

- Not only because all web browsers support it with their ECMAScript rendering engine.
- But mainly because the geospatial web frameworks of OpenLayers, Leaflet and Cesium are written in JavaScript as well!
- This choice avoids the use in the development environment of yet another programming language other than Python as the primary choice for coding data science stuff.

Still we introduce the newer TypeScript programming language as the coding language of choice for our web applications because it is a vast improvement of JavaScript! Look at it as JS 2.0!

- With TS you will need a lot less lines of code than with JS!
- It's just faster programming with less errors!

To even improve the TS programming power we are going to combine it with the generic programming framework Angular which makes for even less lines of code!

This means that in this last part of the workshop the focus is on creating an 'Angular web application' centered around the use of as much Angular functionality as possible to make the programmer's life easy for the web part of things.

The TypeScript source code is then just the 'glue code' that 1) holds the web application together and 2) in which to program a few odds and ends to make things work and of course 3) to program the app's 'Payload Code' that holds the business logic of what the web app is supposed to do!

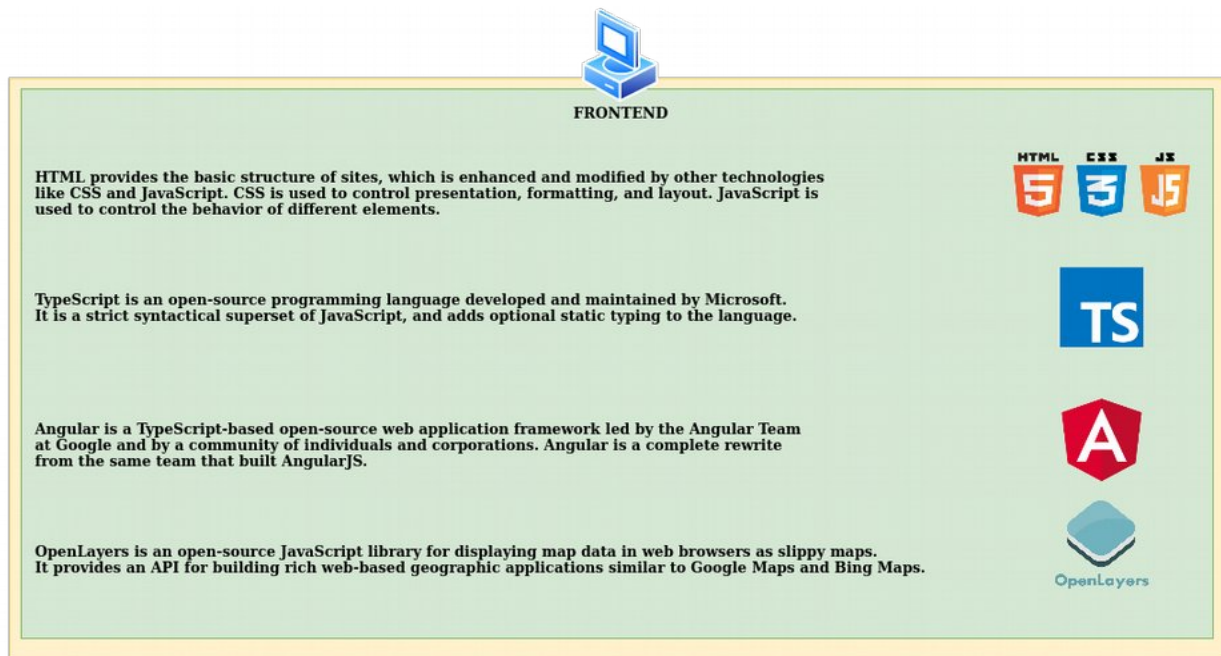
- Remember the other part of Payload Code is in the Flask app to get data for the web app!

The architecture of an Angular based web application is also divided into several separate files which will be explained later in this chapter and which you will learn to code in the programming assignments.

- When Angular is installed you will see when browsing the folder structure with the Nautilus file manager that an Angular web app consists of a very large amount of small files.
- This workshop focuses on explaining just the important parts and to show what-is-where!

For now it is good to know that the development environment for Angular based web applications is centered around the Angular Live Development Server (ALDS) which is a web server that will allow you to see your code changes to the web application live in the web browser every time you save a source code file in your (Atom) programming editor!

- Notice the Angular Live Development Server is used in this workshop to also run your web application but you would not do that normally because you would host the files for the web app in a web application folder on your NGINX web server and run it from there as explained in the previous section!
- In the GeoStack Course you will learn how to run the web app from the NGINX web server.
- The first assignment below will explain how to start the Angular Live Development Server.



Drawing 26: Summary of the Development Environment for Web Applications.

The Angular Live Development Server that is used is not the http-server module of Node.js which you might think because node.js is also installed in the Angular Development Environment but the ALDS is installed from the separate web server package webpack-dev-server that is also installed in the Angular Development Environment.

- The ALDS runs on <http://localhost:4200>
- More information on the Development Environment is here:
 - <https://angular.io/guide/setup-local>
 - There you can see what the GeoStack installation scripts installed: Angular, Node.js to run server side TypeScript / JavaScript applications and npm as the package manager.
- **Security Warning:** run development web servers only on a local host for development purposes and never for production nor connected to the internet for public access!

Challenge – Learn you can use other Development Web Servers too!

- Read up for 30 minutes on other development web servers because for Angular programming you can use other web servers too, like the Python built in development web server http.server or the lightweight Mongoose web server.
- <https://docs.python.org/3/library/http.server.html>
 - This one might be interesting to explore because it keeps you in the Python ecosystem.
- [https://en.wikipedia.org/wiki/Mongoose_\(web_server\)](https://en.wikipedia.org/wiki/Mongoose_(web_server))
 - <https://github.com/cesanta/mongoose>
 - This one might be interesting to explore because you can even run this minimalistic web server on embedded hardware platforms with limited CPU power if you like.

4.1.3 Illustrating the Inner Workings of the Web Application

The goal is to create the simplified 2D Map Viewer as a base web application with the programming language TypeScript and the Angular framework for usage in a web browser to get acquainted with the different files, in which folders they are and the development environment.

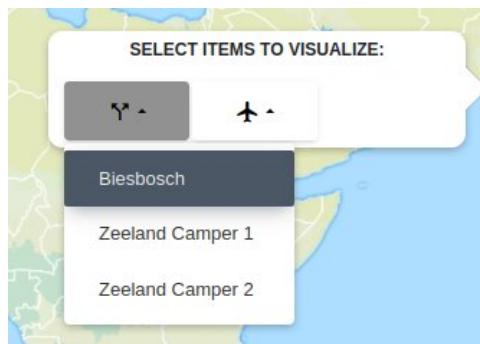
- So let's illustrate some of the web app concepts for the coming programming assignments.

Think of a web application as a program that runs in the window of a web browser for which the window is just a blank canvas to draw different elements on in different places, like:

- a small Settings menu with a Gear icon on the top right side to click open;
- a large background canvas to draw the topographical map on and to plot the geospatial data in transparent layers on top of the map;
- some small controls on the top left side to zoom in and out on the map.

For instance the application contains a few options in the Settings menu to select and visualize the datasets in our MongoDB datastore, like

- a button to select a GPS route from a list of available routes as shown in the image below;
- a button to select the amount of GPS track points to visualize (not shown).



Drawing 27: Example of a GPS track log (trail) selection in the Settings menu of the web application.

If such a menu option is selected, a function will be triggered which sends an HTTP 'GET' request containing an URL to our Flask API as shown in the image below.

localhost/api/trails/5e2351adb91ca7d976cf5efa

Drawing 28: Part of an URL for an HTTP 'GET' request coming from the web application with the MongoDB Object ID at the end which is the 'trail-id' here.

The 'trail' URL in the image above has a query bound to it in the Flask app that was declared with the @route statement in the Flask app.py script file as shown in the image below.

- The full URL (the image above shows only the first part) would have the route name 'Biesbosch' as a part at the end of it as the data request part.
- The Flask app uses these two URL parts to request the data of a route identified by the field '_id' with the value 'Biesbosch' from the Trail_Database in the MongoDB data store.

```
@app.route('/api/trails/<id>', methods=['GET'])
```

Drawing 29: The API route definition in the Flask web application (app.py) to the function 'trails' as the URL end point to convert the API HTTP request for a 'trail' web link with the HTTP 'GET' method into a database query for MongoDB to get trail data.

More information about programming an API in Flask to bound HTTP requests and (re)route or redirect them to database queries and /or information and error messages is in the GeoStack Course cookbook 'Creating a Python Flask web application'.

- Read more about Flask routing here: <https://pythonbasics.org/flask-tutorial-routes/> and here: <https://flask.palletsprojects.com/en/1.1.x/quickstart/#routing> and here: <https://hackersandslackers.com/flask-routes/>

To summarize: a web app sends an HTTP data request and that has to be forwarded in a few steps to a Python function in Flask that will handle that request by retrieving the data from the database in MongoDB.

Let's take a look at the flow of events in network communication starting with the HTTP 'GET' (data) request from the web app.

1. First URL: http://localhost:4200/api/transmissions_by_id/5eff121b78ba6d3fc06df9b3
 1. A first URL like this will be send by the web app to retrieve all transmissions of a crane with the 'tracker-id' shown at the end of the URL which is the MongoDB Object ID for that GPS tracker.
 2. The URL originates from 'localhost:4200' because in the workshop you run the web app from the Angular Live Development Server (ALDS) on port 4200 and not as static files from the NGINX web server which would run on port 80.
2. The 'external' URL from the web app above is send from the ALDS web server to the NGINX web server where it is received on HTTP port 80 as this second URL:
 - http://localhost/api/transmissions_by_id/5eff121b78ba6d3fc06df9b3
3. Then the proxy configuration of the NGINX web server makes this second URL is translated into an 'internal proxy' URL because it has the '/api' part in it! That means it must be send to the Flask API on the Gunicorn web server that runs on port 5000 as the third URL:
 - http://localhost:5000/api/transmissions_by_id/5eff121b78ba6d3fc06df9b3

The same goes for the Trails and Signals of the Car routes and when you would put in an URL directly in your web browser instead of programming a HTTP request in a web app then the direct URL would be something like: <http://localhost/api/trails/5eff12a8ed0bf0f7677735ed>

- This request would get the JSON document for the trail with the 'trail-id' (= MongoDB Object ID) that is shown at the end of the URL.
- In this case you could then get the name of the trail (= name of the car route) from that JSON document.
- To get all signals (= data points) with the GPS locations of the car you would use an URL in the browser like: http://localhost/api/signals_by_id/5eff12a8ed0bf0f7677735eda

4.1.4 Illustrating the Simplified 2D Map Viewer web application

Learning to program works best when starting with a simplified web application to learn the basic structures and then build from there.

That is why this Workshop is about setting up a simplified 2D Map Viewer web application that will be extended in the GeoStack Course to a fully functional web app.

Let's take a look at the end result of the workshop to illustrate what the simplified web app will look like and what it can do.

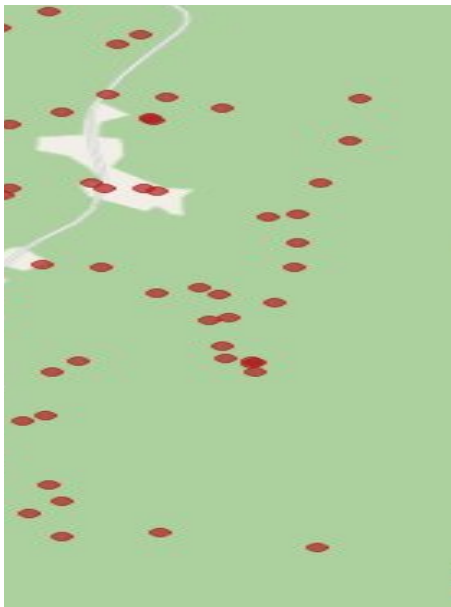
On the blank canvas of the browser window there will be a Settings menu on the right side to select the GPS track. This track log represents a time series of GPS locations where the GPS tracker has been.

The track itself is just a dataset of a travel route made up of data points that need to be plotted on a digital topographical map which results in a canvas to draw the OpenStreetMap base map on.

The next step is to draw a transparent overlay layer that shows the data points on the map.

So it's all about putting data in layers and then display them, which by the way makes 'OpenLayers' a nice name to describe what this geospatial web application framework is used for!

Then in a next map layer the data points are connected by lines for better visualization because just a cloud of points on a map is hard to understand.

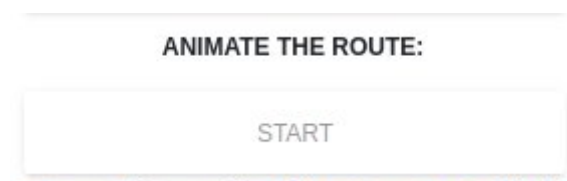


Drawing 30: Plotting geospatial data points on a map.



Drawing 31: Connecting the data points with lines.

Finally as an extra the Settings menu has an animation option with a Start button to draw a playback of the time series of data points by plotting the data points and connecting lines in sequence to get a better understanding of how the route was travelled.



Drawing 32: Option in the Settings menu for route animation.

4.1.5 Angular Explained – Basics of the 4 Component Files

Angular is a JavaScript framework to create web applications which is also usable in TypeScript.

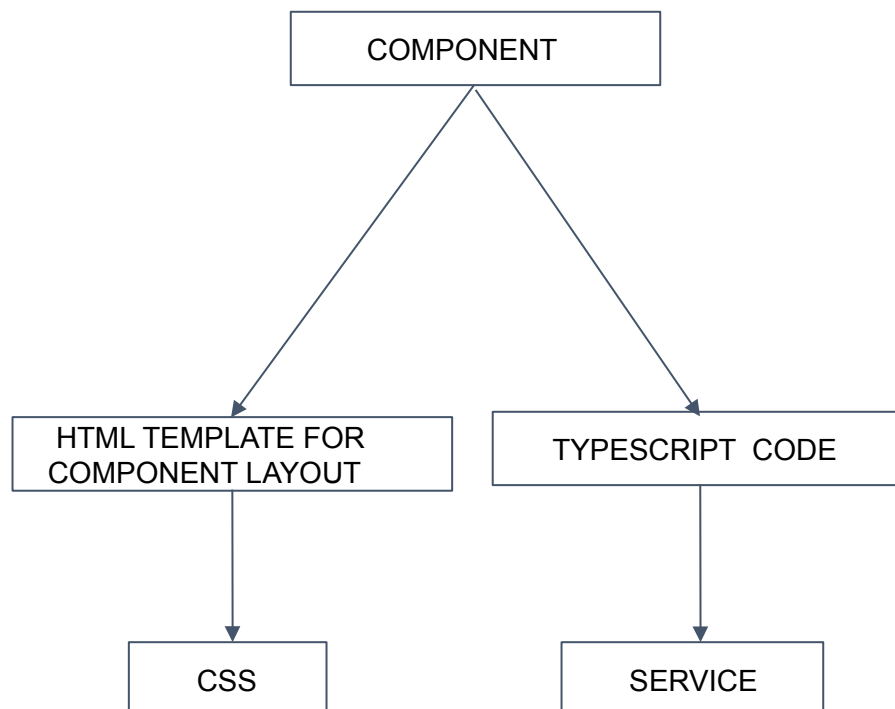
Angular uses stateful components that handle data as opposed to stateless components that do not have data.

- Stateful means data that can change and stateful ('smart') components are able to keep track of data changes.
- Stateless ('dumb') components always render the same object or they just print out what they get passed through property settings.

The stateful components make the data in an Angular web app appear "Live". For instance when data is added to the MongoDB datastore, the web app is automatically updated with the new data!

An Angular Application consists of reusable Components, like a sidebar or a navigation bar and a component consists of 4 parts of 2 optional and 2 compulsory files as shown in the image below:

1. A (optional) TypeScript Service file: this file contains the function which is required to request data from our MongoDB datastore by sending HTTP requests to the Flask API.
2. A TypeScript Code file: this file defines the way the component should function, so it holds the payload code with the component's (business) logic.
 - In this file the function is called that is defined in the TypeScript Service file!
3. An (optional) HTML Style Sheet file: this file contains the styling specifications for the look-and-feel of the HTML elements in the HTML file if the default element style needs changing.
 - This can be a standard CSS file or a SCSS (Sassy Cascading Style Sheet) file.
4. A HTML Template file: this file contains the layout that specifies the 'what-goes-where' structure of the component.
 - In this file we define HTML elements such as a <div> element which is going to contain the OpenLayers specification for the display of a digital topographical map.
 - In this file the CSS or SCSS style sheet file is declared that must be loaded by the web browser that holds the styling specifications of the HTML elements.



Drawing 33: Structure of an Angular Component.

4.2 Challenge – How Angular Development Works 4 Newbies!

4.2.1 Reading instruction to complete the Angular Development!

IMPORTANT NOTICE: this is a very long challenge section ‘explaining the Angular black magic’ for beginners in TypeScript + Angular web application programming!

- This section is deliberately placed in between the previous introduction section ‘Web Application Programming’ and the following 2 sections with programming assignments to prepare for Angular development, called: ‘Assignment – Preparing for Web Programming’ and ‘Assignment – Completing the TypeScript Service File’!
 - Remember, in chapter 2 in the section ‘Assignment – Some Black Magic Explained for Newbies!’ it is explained how an Angular web app gets it’s data from MongoDB with the TypeScript service file!
 - Here you will complete the code for the service file to make it work!
- The result of this choice to explain the ‘black magic’ before the assignments might be that you could find these 2 preparation assignments ‘simple’ or even ‘too simple’ after carefully studying this section.
 - Therefore the advice is to first have a go at the 2 assignment sections to see if you can figure it out on you own! Seeing code first helps in reading this section!
 - Then come back here because you still need to learn about the inner workings!
- After these 2 assignment sections you will have the preparations completed to start with the programming of the 2D Map Viewer web app in which you will use OpenLayers to plot crane flight data on a topographical map from OpenStreetMap.
 - Therefore another explanatory section called ‘Coding Explained – The Angular and OpenLayers Magic’ was deliberately inserted for newbies to explain the ‘black magic’ of how Angular web apps work with OpenLayers!
 - Then follow the section ‘Assignment – Complete the 2D Map Viewer Web App’ to complete the code for the web app to plot crane data on a topographical map!
 - Once you have done that there is a last section, called ‘Challenge – Add the Car routes to the Web App’ to check for yourself if you have mastered the basics of Angular Development with OpenLayers to visualize geospatial data!

4.2.2 Challenge instruction

It might be interesting, especially for Newbies (!), to get a helicopter view first on how the Angular Development Environment is installed, where folders and files are and how an Angular app works.

- The challenge here is to take 1 – 2 hours to read this section with simplified explanations very carefully and 2 hours to follow a few web links to get more detailed explanations!
- Notice again this is a deliberately longer challenge section for more background to learn how to read the source code and understand the workflow sequence in Angular to complete the last two sections with the Assignment and Challenge in which OpenLayers is used!
- For newbies a must (!) and if you have Angular experience it is a fast knowledge check!

4.2.3 Running an Angular Web Application in the Web browser

First let's look at how things work at the web browser side because that is where all the TypeScript files found in the folder structure of the Angular web app will be run.

- Remember the web application is loaded as a set of (static) files from an application folder (structure) on the web server and then the HTML + CSS files are rendered by the browser's Rendering Engine.
- The (inter)active JavaScript code is run by the browser's JavaScript Engine, but... you program in TypeScript, not JavaScript and... that is not a problem! Read on!

This is what the TypeScript web site says itself (<https://www.typescriptlang.org/>):

"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open source.

TypeScript compiles to clean, simple JavaScript code which runs on any browser, in Node.js, or in any JavaScript engine that supports ECMAScript 3 (or newer)."

Basically what it says on the web site is that TypeScript code is converted by Just-in-Time (JIT) compilation to ECMAScript3 which is the official name of the Open Standard specification for JavaScript for which every web browser has a built in JavaScript engine.

- Read more on the JavaScript engine here: https://en.wikipedia.org/wiki/JavaScript_engine

To summarize that: what you program in TypeScript and Angular is converted back to JavaScript by the web browser because every browser has a modern JavaScript Engine built in with Just-in-Time byte code compilation which makes the code run fairly fast too! So, now you know!

4.2.4 Installation of the Angular Development Environment in the VM

IMPORTANT WARNING: DO NOT INSTALL AN ANGULAR DEVELOPMENT ENVIRONMENT AND OTHER ANGULAR COMPONENTS YOURSELF WITH MANUAL COMMANDS IF YOU INSTALLED THE GEOSTACK VIRTUAL MACHINE WITH THE INSTALLATION SCRIPTS!!!

- **Running the installation commands multiple times may break your installation!**
- **If you want to experiment, create a new folder outside the Geostack folder of the GeoStack Course and run the commands in your own folder!**
- **The alternative is of course to make a snapshot or a copy of your VM and learn there!**
- **Of course you can ignore this warning if you try to learn installing from scratch!**
 - **Remember to make snapshots of your VM between installation steps!**

Now let's look at how the Angular Development Environment (ADE) is installed on the computer. For the workshop the installation of the ADE works like this:

1. In the Workshop Virtual Machine you have cloned the GitHub repository for the workshop.
 - The Assignment and Solution folders hold the web app folders, like for Assignment:
/Part-4-Data-visualization/Sourcecode/Assignment/map-viewer/src/app/
 - Notice the folder name of the web app is 'map-viewer' and the source code files are in the 'src' folder.
2. The installation script for the ADE is 7-frontend-software.sh and it is located in the folder:
 - /GeoStack-Workshop/Installation-scripts/
3. When you run this installation shell script the ADE is installed in the Workshop Virtual Machine in a standard way.

4.2.5 Installation of the two ADE Base Components

The installation script installs the following two base components for the ADE:

1. Installation of node.js as the JavaScript runtime environment to run a JS file without a web browser. This makes node.js the server for all programming languages that compile to JavaScript, which includes TypeScript.
 - Read more here: <https://en.wikipedia.org/wiki/Node.js> and here: <https://nodejs.org>
2. Installation of npm (Node Package Manager) as the JavaScript package manager which is used to install JS packages and frameworks from the JS repository called 'npm registry'.
 - It is the default package manager for node.js and it runs on node.js itself!
 - The command line client is simply called 'npm' and this is the tool that is used to install all the packages that are needed for the ADE and for application development.
 - For instance if you need your web application to draw a pie chart you would install the chart.js with npm.
 - More here: [https://en.wikipedia.org/wiki/Npm_\(software\)](https://en.wikipedia.org/wiki/Npm_(software)) and <https://github.com/npm/cli>

4.2.6 Installation of the Angular CLI package

The next step for the installation script is to install the Angular Command Line Interface package by using the npm command as Linux administrator with: `sudo npm install -g @angular/cli`

- It's nice to know other JavaScript programming frameworks, like react and vue, can also be installed with npm.

This will install the Angular command line client called 'ng' and this is the tool to give Angular commands, for instance to create new projects and start web applications.

- Read more on the ng commands here: <https://cli.angular.io/>
- It's nice to know there exists a command mapping option for npm to ng!
- It is possible to define and use npm to give ng command's so 'everything is npm' to make life easy for a programmer.
- An example of such a command mapping is the 'npm start' command that you give in the root folder of the app to start it on the Angular Live Development Server (ALDS), which makes npm run the ng command: `ng serve`
 - Read more about the 'ng serve' command here: <https://angular.io/cli/serve>
 - For instance in the workshop the 'npm start' command is used in the desktop shortcuts for the programming assignments and solutions to launch the web application.

4.2.7 Starting a New web application project in Angular

Now the ADE is installed a programmer would normally do 2 things to start a new project, which for the workshop of course will work a little different, but now let's look at how it normally works:

1. Give the command: `ng new`
 - This will create a new project folder structure for the web application which is populated with a whole lot of folders and small files! Really, very, very many files!
2. Use the npm command to install the JavaScript packages the developer needs for the web application, like the example given above for the chart.js package used to draw a pie chart.
 - Each package that is installed is administered by npm in the dependency section of the package.json file that is in the root folder of the web application.

4.2.8 How the web app installation works for the workshop

For the workshop the installation of the web app in the two assignment and solution folders works a little different because the entire workshop folder structure is already installed by cloning the GitHub repository GeoStack-Workshop, including those two prepared web app folders.

1. The web app folders and files are there for the assignment and solution folders.
2. The package.json file is there. It was manually created to list all the dependency packages.
3. The dependency packages themselves are NOT present in the GitHub repository to save lots of unnecessary disk space use on GitHub of a few hundred megabytes!

Therefore the dependency packages that are listed in the web apps package.json file still need to be installed for which the npm package manager has the convenient parameter 'install'.

- To do this the installation script changes its working directory to the solution folder of the web app and then it installs the dependency packages with the command: `npm install`
- Then the installation script changes its working directory to the assignment folder to install the same dependency packages again.
- Where the files go is explained below in 'What-goes-Where'!

4.2.9 Running an Angular web application

The 'npm start' command runs the 'ng serve' command for you that builds and serves your app, rebuilding on file changes after each source code file save in the (Atom) programming editor.

- In the ADE the command 'ng serve' uses the ALDS, which is the webpack-dev-server package, to run the app directly from local memory.
 - Read more here: <https://webpack.js.org/guides/development/#webpack-dev-server>
- For deployment on a production web server, like NGINX, the command 'ng build' is used to build the folder structure with all the files in it that needs to be copied to an application folder that has to be created on the web server.
 - The 'ng build' command reads the package.json file from the project's root folder to include all the dependency packages in the build folder structure.
 - Read more here: <https://angular.io/guide/deployment>

4.2.10 The Angular run time difference for index.html

There is a little peculiarity with the index.html file between running an app from the ALDS and when running it from a build folder structure because the content of the index.html file slightly differs.

As a web developer you would expect to see a declaration in the index.html file to load the Angular TypeScript files, just like the declaration to load a style sheet file but it works a little different.

When running a web app on the Angular Live Development Server the Angular web app starts by loading the central index.html file found in the /src subfolder of the app's root folder and the required typescript files of the Angular framework itself are automatically loaded in the background. There is no declaration to load Angular files in the index.html file as you normally would expect.

When development is finished and the web app is ready to be installed as a set of static files on a production web server a developer will generate the production version of the app with the Angular build command. This 'ng build' command which will then generate the web app's index.html file including the required declarations to load the Angular files.

- In the index.html file the angular.json file is specified which holds the specification for the TypeScript file main.ts that imports (loads) all the Angular code to run the web application.

4.2.11 What-goes-Where? --> in an Angular Project Folder!

To get an idea what the core files are when learning to program with the Angular framework it is good to take a look at the files that are used to develop a web app and in which folder they are.

- The folder structure of an angular app has many folders and many small files and it's not easy at first to get a helicopter view over what-goes-where.
- When an 'empty' project folder for a new app is created with the command 'ng new' the Angular project folder gets the name of the app and it is often called the app's root folder or the app's Parent Working Directory (PWD).

As an example, let's explore 'What-goes-Where' in the workshop folders:

- For the workshop the chosen app name for the simplified 2D Map Viewer project is just 'map-viewer' so the root folder is /map-viewer.
 - The root folder for the workshop is in /Part-4-Data-visualization/Sourcecode/Assignment or in the other root folder in /Sourcecode/Solution, depending where you want to look.
- In this root folder /map-viewer there are:
 - The file package.json that holds the list of compulsory dependency files to be installed with the command 'npm install' to enable the web app to be run!
 - The installation command creates the folder /node_modules in the root folder where a few hundred megabytes of folders and files are installed, required to run the app!
 - Read more here: <https://docs.npmjs.com/configuring-npm/folders.html> and here: <https://www.sitepoint.com/beginners-guide-node-package-manager/>
 - Also npm creates the file package-lock.json in the root folder for several reasons.
 - Read more: <https://docs.npmjs.com/configuring-npm/package-lock-json.html>
 - The folder /src which holds all the web app's source code files.
- In the folder /src there are:
 - The index.html file that starts the loading of the web app by the browser.
 - In this file is the specification to load OpenLayers from the /src/assets folder to show you can easily install software to run locally on your own if you want version control instead of installing a repository package version with the npm package manager.
 - We created the folder /src/assets/geospatial-frameworks to hold all the products and versions, like multiple OpenLayers versions or also Leaflet as alternative framework to experiment with. For the workshop it is just the /OpenLayers folder that is used.
 - The default favicon.ico file with the Angular logo to display in a browser tab which we replaced by adding our project logo in the file /src/assets/img/favicon.png
 - It is loaded by /src/app/components/sidebar/sidebar.component.html
 - The style sheet file styles.scss for global styling.
 - This file is empty and for the workshop only default styles are used because it looks pretty nice out-of-the-box. All the other style files are in the folder /src/assets/scss.
- In the folder /src/app there are:
 - The two compulsory app files: app.component.html and app.component.ts
 - The default navbar and sidebar are in the folder /app/components
 - The folder /app/pages holds the two compulsory workshop's map viewer core files: map.component.html and map.component.ts.
 - The folder /app/services holds the file crane.service.ts to get crane data in MongoDB.

4.2.12 Event handling basics in Angular Apps

Each component of an Angular web app has its own sub-folder to hold the component files with always the HTML Layout file `app.component.html` and the TypeScript Code file `app.component.ts` as the two compulsory component files and optional the (S)CSS Style Sheet file and the TypeScript Service file.

- If there is no CSS file in the component folder then Angular will use the web app's CSS file to style the elements in the component's HTML file which is in the 'assets' folder.
- Remember the TypeScript Service file is to program the specifications to make a mapping to an external (web) service for the event handling of data requests and receiving results.
 - In our case to communicate with the Flask API to send data requests and receive data.
 - Therefore you will need to learn to program this optional file too in a later assignment!
- For internal event handling a new `app.component.html` file always has two `<div>` elements:
 - One to handle the component title, let's say 'Component Name'
 - One to handle the content of the website which you call new components when a specified event is triggered, like a mouse click on a button or clickable menu choice to display a menu, a map, a date-time picker etc.
- A web application always has a navigation component!

4.2.13 Best Practice in Angular Programming

The programming practice is to put everything that always has to be visible on the screen in the main HTML file `app.component.html`.

- Everything else that only has to appear or has to run when a specified action is triggered, is programmed in separate components to keep the a structured folder structure and source code files maintainable.
- Each component gets its own component folder 'component_name' and their own set of component files for which of course the two files `component_name.component.html` and `component_name.component.ts` are compulsory.

It's like in Python where you have a main program file with import statements to load other modules and then the functions from those modules can be used in the main program.

- Both in Python and in TypeScript/Angular programming you need to learn 'what-goes-where' to keep files readable and thus maintainable!

4.2.14 Terminology --> Single Page App + Bootstrapping + Routing

Angular is a programming framework for so called 'single-page apps' (SPA), meaning the web application is divided into separate components.

- Now first read this (!): https://en.wikipedia.org/wiki/Single-page_application
- This means a switching mechanism is required to swap (rewrite/replace) web pages!
- Remember each app component has a HTML web page file paired to a TypeScript file!

Workflow in an Angular App

1. The process of loading the web application and the Angular framework into the web browser is called 'bootstrapping'. This allows the Rendering Engine to run the HTML + CSS declarations and the JavaScript Engine to run the TypeScript code.
2. The process of switching between the single-page components is called 'routing'. This means there has to be a 'routing engine' in Angular to take care of the workflow when events are triggered to navigate between web pages of the user interface or to run background components to run their 'payload code' to handle data or do 'what ever'.

4.2.15 What-does-What? --> The inner workings of an Angular App!

Here's a *definitely non-scientific* summary of 'What-does-What' in an Angular App to give you a global idea of the workflow in an Angular App. Just to give you a little more background for the programming assignments to get the web application up and running and study all of the code!

- It's quite a chain of events to load, bootstrap and start a web app!
- A simplified look at the inner workings here might be a better idea than a full overview!
- See the web links below for an exact explanation but that is a lot of reading work!

Loading an Angular App and Starting Angular

The index.html file in the folder /map-viewer/src is the starting point and loading this index.html file in the web browser by addressing it's URL is the 'Way of the InterWeb'!

- Pointing your web browser in the workshop to <http://localhost:4200> in the workshop will do the trick to load this index.html file from the ALDS.
- Remember, an index.html file is loaded automatically in the web browser if the URL points to the web application folder on the web server and this file is present there.

In the index.html file there is a <script> declaration '`<script src="angular.js"></script>`' to launch Angular if it was built with the 'ng build' command or launching Angular will happen in a few hidden steps on the ALDS because Angular will do that in the background for you.

The main hidden steps after loading the angular.js file we can't see in our files in the workshop because of the ALDS and because the build command has not yet put all files in place are roughly:

1. In the angular.ts file there is a declaration of the angular.json file which holds the specification for the TypeScript file main.ts that will actually run the Angular framework.
2. That main.ts script imports (loads) all the Angular code to run the bootstrap process for the web application by using the Angular's bootstrap module AppModule.

How the bootstrap continues from the app.module.ts file

The AppModule activates the app by loading the app.module.ts file to start the web app and this file we can see, because it is in the folder /map-viewer/src/app!

There are two import statements (without the .ts file name extension) for the TypeScript files app.component.ts and map.component.ts:

1. `import { AppComponent } from './app.component';`
 2. `import { MapComponent } from './pages/map.component'`
- Then there are the two corresponding run specifications in the 'declarations' section.

These files specify: 1) where the web app is declared and 2) what the main component is to start the web app.

In app.component.ts are 3 important specifications for:

1. The 'HTML element selector' to specify the HTML element 'app-root' that we need to refer to in the body of the index.html file.
 2. The 'Angular URL' specification for the templateUrl to find the corresponding HTML file for app.component.ts to launch the app, which is by default app.component.html.
 3. The title of the web app which is 'map-viewer' in the workshop and also the folder name.
- How the bootstrap continues is explained after the rest of what happens in index.html.

Loading the OpenLayers framework in index.html

To load the OpenLayers framework the index.html file holds 2 declarations:

1. One for the <link> element to specify where the OpenLayers style sheet file is, which defines the 'look-and-feel' to display things like zoom buttons and topographical maps.
2. One for the <script element> to specify the OpenLayers framework itself.
 - Remember, both files are installed locally for the workshop as an example for manual version control in the folder: /map-viewer/src/assets/geospatial-frameworks/OpenLayers

Putting the Placeholder for the Angular App in index.html

The placeholder for the Angular app in the index.html file is the 'Magic' Angular HTML selector element <app-root> which must be placed in the <body> element of the index.html file.

- This is how Angular knows where in the HTML layout to start the web app.
- The simplest HTML <body> declaration in index.html to let Angular work is:

```
<body><app-root></app-root></body>
```
- Remember, the 'app-root' selector for Angular was specified in the app.component.ts file.
- This selector allows Angular to insert the main web app component here.
 - Remember the mechanism for SPA web apps that requires swapping components!
 - At the app-root the HTML code of app.component.html is inserted to run.

Finishing the Bootstrap with app.component.html

In app.component.html there are 2 <div> elements for the side-bar and the main-panel.

1. The <div> element for the side-bar has the <app-sidebar> placeholder element that runs the Angular side-bar component.
2. The <div> element for the main-panel has 2 placeholders elements:
 1. The <app-navbar> placeholder that runs the default Angular navigation bar component.
 2. The <router-outlet> placeholder that runs the Angular routing module for the SPA!
 - This router placeholder finally makes Angular display the main web page of the web app and in the workshop that is /map-viewer/src/app/pages/map.component.html

The side-bar and navbar are always there and located in the folder /map-viewer/src/components.

- In sidebar.component.html the favicon file of the web app is loaded to display the web app icon on the left side a browser tab in which the browser displays a web page.
- The sidebar also gets the content for the default 'hamburger' menu on the navigation bar for quick navigation to other web pages in the app (think SPA!) which opens when clicked as a side-bar menu with a list of available web pages.
- The navbar component then populates the side-bar menu with the titles of the web pages and it would hold only one menu choice for the map viewer web page.
- Only before the programming assignments you can see the navbar with hamburger icon!
 - In the workshop the navbar and side-bar menu is not used because the map component is the only component in the web app, so no need to switch web pages!
 - The navbar with the 'hamburger' icon is not displayed either because the app is set to display map.component.html to use the full browser window so the navbar is hidden.

4.2.16 Completing the Circle --> Getting the Crane Data in map.component.ts

Now the bootstrap is complete the web browser shows a web application with a blank canvas and a default 'bootstrap' navigation bar at the top that only has a 'hamburger' menu icon on the right side as a clickable button that shows the map-viewer as its only menu choice when clicked.

- Notice this is the most basic Angular app you can get!

When the map-viewer menu item is clicked nothing happens because code completion of the Map Component files is required in programming assignments to get working components to display. The map component requires 3 component files to work and 2 have programming assignments:

1. /map-viewer/src/app/services/crane.service.ts
 - This file IS NOT code complete! Complete the 'Payload Code' here!
2. /map-viewer/src/app/pages/map.component.html
 - This file IS code complete! No work here, except to understand exactly how it works!
3. /map-viewer/src/app/pages/map.component.ts
 - This file IS NOT code complete! Complete the 'Payload Code' here!

The event flow in the map.component.ts file needs a little more explanation as to what exactly happens there in the routing to get the crane data and then to draw the map:

1. Remember in map.component.html the Angular app routing to the function `getTransmissions()` in the file map.component.html was triggered by the detection of the event of a mouse click on a menu button to get the transmissions for the selected crane by passing the crane's 'tracker-id' as a parameter to the `getTransmissions()` function.
2. The Angular routing engine will then switch to the map.component.ts file where the `getTransmissions()` function is, so let's take a look of what happens in that file.
3. To hold the list of transmissions that will be retrieved from the 'Crane_Database_' in MongoDB a global list variable is declared with: `private transmissions: any[] = [];`
 - This global list holds the transmissions of the currently selected crane or it is still empty if no crane has been selected yet when the application starts.
 - This global list is used to plot the data points on the map and to run the route animation.
4. The source code of the function to get the transmissions is this:

```
getTransmissions(trackerId): void {  
    this._CraneService.getTransmissionsID(trackerId).subscribe(  
        (transmissions: []) => (  
            this.createPointLayer(transmissions),  
            this.createLineLayer(transmissions),  
            this.transmissions = transmissions)  
        )  
    }  
}
```

- What happens here is that a local list variable 'transmissions' is declared for the `.subscribe()` method to hold the list of transmissions that will be retrieved from the 'Crane_Database_' in MongoDB.
 - Notice it can be any variable name you like but 'transmissions' is what it holds so that's fine but do not confuse it with the global variable with the same name!
- This local list is then filled by the CraneService method `.getTransmissionsID()` which is declared in the component service file map.service.ts.

4.2.17 Completing the Circle --> Drawing the Data on the Map in map.component.ts

Now the GPS track log of the crane flight path is available in the local list 'transmissions' of the `getTransmissions()` function it's time to follow the last flow of events to see how drawing the data on the topographical map works.

In the `getTransmissions()` function some routing 'Black Magic' happens because of the arrow operator `'=>'`, which is officially called a lambda operator.

The lambda operator passes the local list of transmissions for further processing which in this case are the following 3 steps!

1. First the local transmissions list is passed to the `createPointLayer()` function to draw the data points on the map in the last statement: `this.map.addLayer(pointVectorLayer);`
 - How the topographical map itself is created is explained below.
2. Then the local transmissions list is passed to the `createLineLayer()` function to draw the lines on the map between the data points, also with an `.addLayer()` method.
 - Learning how to do this is a programming assignment.
3. Finally the assignment statement: `this.transmissions = transmissions` makes the local transmissions list is assigned to the global variable `transmissions` to make it available for other functions, like the `animateRoute()` function.

Finally the only mystery left is how drawing the topographical base map itself works because it works a little different with the tile server and OpenLayers than it does for getting the tracker and transmission data from the crane database in MongoDB.

The event flow is fairly simple.

1. Notice the topographical base map was already created in the `ngOnInit()` function with the statement: `this.createMap();` that runs the function `createMap()`.
 - The `'this.map'` statement in the `createMap()` function actually creates the map object in memory and also makes it is displayed in the HTML layout of the web page!
2. When you want to draw a topographical map in OpenLayers you have to make 3 steps in the `createMap()` function to make it happen:
 1. Create the base map itself as a `'baseLayer'` object.
 2. Create the `'helicopter viewpoint'` to look at the map as a `'view'` object.
 3. Create the `'map'` object that can be displayed in the HTML layout in the web browser!

Let's take a look at those 3 steps to draw a topographical map.

1. The `'baseLayer'` object in OpenLayers specifies 2 things:
 1. That the base layer for the topographical base map is a tile layer object.
 - This means it is a blank canvas to draw the map tile PNG files on.
 2. What the URL is for the tile server to get the map tiles from.
 1. Notice the TileStache tile server provides the `'/tiles/'` API for its internal routing of HTTP requests to the correct function that will get the requested map tile files from the cache folder or as an intermediate step also route the request to another function to get them from the online OpenStreetMap Web Map Service if they are not present yet and store them in the cache folder.
 2. Also note the URL endpoint to route that request to is `'openstreetmap'` which is the cache folder for the OpenStreetMap map tiles on the TileStache server.

2. The 'view' object specifies 3 things:
 1. The available maximum zoom level 'maxZoom' the app is allowed to use.
 - A value of 18 lets you zoom in to a very detailed 'house and street' level but remember the Web Map Service you get the map tiles from must of course support that level of map detail.
 - Also it can be a disk space management choice to limit the zoom level and not store the map tiles for the more detailed zoom levels even if the WMS can deliver them!
 2. The center viewpoint to display the map.
 - For the workshop it is set to center the view on [0,0] which is the junction where the prime meridian (= zero meridian = Greenwich meridian) crosses the equator.
 - More on that is explained in the 'Scope Limitation' below.
 3. The zoom level for the height of the 'helicopter view' to look at the map.
 - Remember a 'zoom' of 0 is a complete 'World Map' view so 3 is still pretty high up in the sky while a zoom of 18 is a detailed 'house and street' level a WMS can provide.
 - More on that is explained too in the 'Scope Limitation' below.
3. The 'map' object that is created with the 'this.map' assignment specifies 3 things:
 1. The key-value pair "target : 'map' "
 - This specifies the mapping for the Angular routing to HTML file map.component.html to display the map by making 'map' the identifier of the <div> element on the 1st line that displays the map: <div id="map" style="width: 100%; height: 100vh;"></div>
 2. The 'view' key specifies the 'view' object specified from the 'let' declaration.
 3. The 'layers' key specifies the 'baseLayer' object from the 'let' declaration.

4.2.18 Scope Limitation --> Centering the Map at [0,0] explained!

Remember in section 3.7.2 'Assignment – Checking the End Result!' it was explained that the map tiles are requested from the TileStache tile server by using the Z/X/Y.png notation in the URL.

In the Angular web application it works differently with OpenLayers and you can use the normal notation for the zoom level and GPS coordinates in an URL for a HTTP request if you want to display a map at a zoom level and map center of your choice!

Still it is good to explain how things work between the web app and the tile server for a better understanding.

Here is the cookbook recipe on map centering for the tile server:

1. The blank map canvas which is called the 'map extent' in OpenLayers that is shown in the browser window is a rectangular bounding box that must hold a whole number of rows and columns of square map tiles in a Tile Grid.
2. The corners of the bounding box for the tile grid are calculated by TileStache from the zoom level 'Z' and this determines the number of map tiles that has to be requested to fill the grid.
3. The X/Y coordinates to determine the map tile in the center of the tile grid are calculated by TileStache from either real-world GPS coordinates [X,Y] or by re-calculating an on-screen location selection of GPS coordinates on a map tile for which the zoom level is known.
4. The default Z value for the zoom level and X/Y values of the GPS coordinates for the desired map view need to be set on the Angular web app side in the file: map.component.ts

Both in the GeoStack Workshop and in the GeoStack Course the web application centers the topographical base map at [0,0] because the datasets of the cranes contain flight paths that span migration routes over vast areas in Europe, like all the way from Sweden to Spain and back!

- When a dataset is displayed you can use the mouse to drag the map to the part of the 'point cloud' of the flight path you want to see and use the OpenLayers zoom buttons on the top left of the map window to zoom in on the part of the flight path of your choice.
- Remember in Part 1 you already explored roughly in Cartopy where the datasets appear!

It is 'by design' this simple visualization solution was chosen to keep the focus on how displaying a topographical base map in OpenLayers actually works to show the 'world map' in the image below:

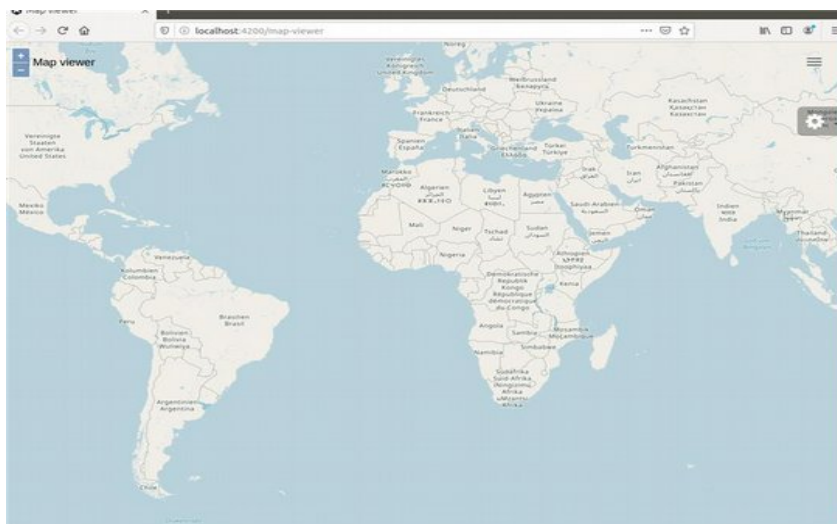


Illustration 4: With a zoom level of 3 this high helicopter view almost displays a 'world map'.

There are at least 3 reasons to select a 'world map' view as the starting point in both the workshop and the course and not make it part of the app design because selecting a dataset or part of a dataset to determine, what is officially called, a 'Bounding Box' for a 'Map View' depends on:

1. The datasets themselves require design choices:
 1. Geospatial datasets can be very large so it could be a choice to just use a selection.
 2. Area coverage: the crane datasets cover much vaster areas than the car datasets.
2. What you want to do with the app determines which map view is the best to start with.
3. The educational point of view: it requires some extra programming and app design to use a bounding box that also comes with extra complexity that would unnecessary complicate things. The choice was to keep the focus on Angular and OpenLayers, nothing else!

To illustrate the design options in the course there is a nice design example of the 'zoom-to-start' function that, when another crane is selected, fairly slowly zooms out from the current map position and then 'flies' to move and zoom in on the first data point that is indicated with a 'Start' marker.

- In this way the user of the web app keeps orientation of where one is currently on the map and where the short 'helicopter flight' of this zoom function takes off and lands on another part of the map. Just refreshing a map view at a new map location might be very confusing!
- The zoom speed gives control for a slow or fast flight to a new map view.
- The zoom level of this function determines the map view at the 'helicopter landing point' which makes how much of the environment is visible for the required map orientation and situational awareness you would want to provide to the user.

Navigating to the dataset yourself is the design choice but it is good to explain a little about what is required to center the map and where the complexity is to make a bounding box for the map view.

1. First decide on using the whole dataset or just a selection of data points in a time frame.
 - For example in the workshop for crane Agnetha the first 1,000 data points of the dataset were used which makes for a point cloud in the south of Sweden instead of a complete migration route across Europe.
 - In the full web app in the course the programming example is available for a date picker to choose a Date-Time Group (DTG) for the start and end of a time frame (= time period) for the selection of a part of the dataset.
2. Once the dataset is selected the bounding box can be determined in at least 3 ways:
 1. Determine a center point and zoom level, like in the workshop [0,0] and zoom level 3.
 2. Use a GPS coordinate for a Point of Interest or center of an area polygon + zoom level!
 1. See this weblink: <https://openstreetmap.be/en/projects/howto/openlayers.html>
 2. This example shows a great summary (!) of the OpenLayers basics and displays the the capitol city of Bruxelles in Belgium with GPS coordinates as a center point.
 3. Calculate the GPS coordinates for the bounding box from the dataset (selection) and... this is where it gets more complicated but... this is usually the way to do things!
 1. This first requires finding the North-West and South-East GPS coordinates for a 'Bounding Box' from the data points in the dataset and then probably also enlarge the bounding box coordinates a little to not display data points on edge of the map!
 2. Then convert these GPS coordinates to the Z/X/Y.png specification for the tile server to determine the bounding box for the map view that fits the dataset which is definitely not a trivial matter to convert GPS coordinates to a map tile selection!
 1. <https://gis.stackexchange.com/questions/109095/convert-xyz-tile-request-to-wms-request> (Read the short version here!)
 2. https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames (The documentation!)
 3. Note: in the GeoStack Course the tile server explanations will help you understand how it works to get the bounding box map view you want.

4.2.19 Scope Limitation --> No Map Refresh when changing Cranes

To keep the workshop web app simple the crane dataset from a previously selected crane is not removed from the map display in OpenLayers when another crane is selected from the menu.

- Of course the data points and connecting lines for the newly selected crane are displayed on the map but the flight path from the previous crane is still there too.
- In the GeoStack Course a smarter approach is used which is called 'item display' in which datapoints, lines, start and end markers etc. are all handled as separate items in their own layers that can be easily removed when a corresponding trigger requires a 'map refresh'.
- The `animateRoute()` function will work as expected for the currently selected crane because the global list variable `transmissions` is refreshed each time when another crane is selected.

4.2.20 How Angular Apps Work Exactly! --> Web links for more!

Because this section is a challenge to educate yourself on the inner workings of an Angular app, here's a short selection of some good web links that explain exactly what happens, including the correct terminology.

- It's a good idea to click the links, just to see what they are about and then study them later because it's quite a lot of reading work. Remember, sooner or later you have to learn this!

Web links – How Angular Apps Work

1. <https://dev.to/casperns/how-angular-trigger-indexhtml-and-start-working-1146> (Read this!)
2. <https://blog.angular-university.io/why-a-single-page-application-what-are-the-benefits-what-is-a-spa/> (A very nice explanation of SPA software design and Angular workings!)
3. <https://www.educba.com/how-angular-works/> (More bootstrap details explained.)
4. <https://medium.com/siam-vit/how-an-angular-app-work-behind-the-scenes-angular-flow-dcc4d1df27bd> (The bootstrap explained from another useful view point.)

Web links – The Inner Working Details

1. <https://docs.angularjs.org/guide/bootstrap> (Loading speed advice.)
2. <https://angular.io/guide/architecture> (Helicopter view of Angular concepts.)
3. <https://docs.angularjs.org/guide/concepts> (Angular concepts in a little more detail.)
4. <https://angular.io/guide/router-tutorial> (Nice example to compare to the workshop code!)
5. <https://angular.io/guide/router> (The Angular router documentation.)
6. <https://docs.angularjs.org/tutorial> (The official Angular tutorial to be understood now!)

Web links – How OpenLayers Works

- <https://openstreetmap.be/en/projects/howto/openlayers.html> (A very nice basics tutorial!)
- Also read the documentation web links at the bottom for each OpenLayers component!

The Next Step in Programming the Web App

The last sections in this chapter give detailed information and instructions to complete the code for the simplified 2D Map Viewer web application to help you learn the Angular programming basics!

4.3 Assignment – Preparing for Web Programming

4.3.1 Assignment – Understanding the Programming Steps!

IMPORTANT NOTE: read this 'cookbook recipe' section carefully (!) to understand the 3 development steps for the programming assignments of which the following 3 desktop shortcuts are available:

1. angular-assignment: start the Angular Live Development Server to run the Web App.
2. angular-service-assignment: program a TypeScript Service file to the Flask API for data.
3. Angular-openlayers-assignment: program the simplified 2D Map Viewer web app.

To perform the programming assignments of this part of the workshop a basic Angular web application is provided that already contains a big part of the code which is required to visualize the geographical data because building an Angular app from scratch might be a little intimidating.

- It's much easier to learn from example code that needs some code completion here and there to see where the important web app parts and geospatial code parts are located.
- The rest is quite a bit of 'glue code' to make it all work but that's for another day to study or if you want to know right away as a Newbie then you can do the extra reading below in the section 'Challenge – Read up on Angular Programming to understand the Source Code' for some enlightenment up front!

Start the work environment: Before we can start working on the assignments we need to start the built-in Angular web development server. This enables us to test the application on the fly (live testing), during the assignments.

- More information related to the Angular Live Development Server can be found in the programming manual: "Creating a basic web application"

During this part of the workshop you are going to complete pre made code to be able to create the following two major functionalities:

1. Data Retrieval: by sending HTTP requests to the Flask API to retrieve the geospatial data from our MongoDB datastore.
2. Data Visualization: by plotting the data retrieved from the MongoDB datastore on a 2D topographical map using the geospatial JavaScript framework OpenLayers.

The 2 major functionalities of data retrieval and visualization mentioned above can be divided in the following 5 smaller steps:

1. Creating the service to obtain the Crane tracker JSON documents via the Flask-API is done to fill the selection menu in the web application.
 1. The goal is to select Transmissions belonging to a select Crane tracker using it's ID. The Transmission selection will be created in sub-step 3.
 2. The function used to obtain the transmission, using a TrackerID, is already defined in the service file that comes with the web application.
 3. This assignment can be found in the file: "crane.service.ts".
2. Creating a base layer that contains the digital topographical base map from OpenStreetMap. This base layer will then be added to the OpenLayers map. This assignment can be found in the file: "map.component.ts".
3. Completing a function which shows the transmissions, obtained from the MongoDB datastore via the Flask-API, as data points on the map layer which was created in the previous sub-step. This assignment can be found in the file: "map.component.ts".
4. Completing a function which connects the data points, created in the previous sub-step, by creating line segments. This assignment can be found in the file: "map.component.ts".
5. Completing a function which enables us to animate the visualized routes using the line segments, created in the previous sub-step. This assignment can be found in the file: "map.component.ts"

IMPORTANT NOTE: the source code in both the crane.service.ts file and the map.component.ts file contain in depth in-comments that explain which steps are performed in the pre-made code. **Please, read these comments carefully!**

4.3.2 Assignment – Start the Angular Live Development Server

There are 3 desktop shortcuts for the assignments.

1. angular-assignment
2. angular-service-assignment
3. angular-openlayers-assignment

Start the work environment: Before we can start working on the assignments we need to start the built-in Angular web development server. This enables us to test the application on the fly (live testing), during the assignments.

- More information related to the Angular Live Development Server can be found in the programming manual: "Creating a basic web application"

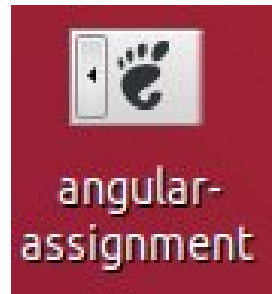
In the folder: "Part-4-Data-visualization/Assignments" you will find a folder called: "map-viewer".

This folder contains the unfinished source code of the (Angular) web application. We are going to finish this source code to accomplish the same result as shown in slide 5 in this presentation.

We start by running the (Angular) web application, which contains the unfinished source code, on the Angular Live Development Server. After each assignment which you correctly finish save, you will see that the application will start to look more like the end result shown in slide 5.

An advantage of using Angular, is that when the web application is running, on the Angular Live Development Server, you only have to save your code and the changes will be automatically added to the web application. This is also the reason why we are first going to start the Angular Live Development Server.

To start the Angular web application, on the Angular Live Development Server, you should click on the desktop shortcut called: “angular-assignment”. (Illustration 1)



Drawing 34: Desktop shortcut to start the Angular Live Development Server

Wait for the web application to start and then click on the web link which is shown in the terminal output, as shown in the screenshot below.

- You can quickly launch your web app by pressing the CTRL key + Left mouse click on the web link <http://localhost:4200> that is listed in the terminal output on the VM!
- This action will then open the still unfinished Angular web application in the web browser.

```
chunk {main} main.js, main.js.map (main) 71.2 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 2.41 MB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.84 MB [initial] [rendered]
Date: 2020-01-23T18:29:18.738Z - Hash: 498db59543334eeabbf2 - Time: 16286ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i [wdm]: Compiled successfully.
```

Drawing 35: Terminal output when starting the Angular Live Development Server.

Notice the ALDS is running on port 4200, so in the web browser surf to: <http://localhost:4200>

IMPORTANT NOTE: this assignment is about starting the Angular Live Development Server because obviously the Angular web application can't run without this web server in the Angular Development Environment.

- Still, it's easy to forget to start it (again), especially after a (re)boot of the Virtual Machine!
- Remember to use the desktop shortcut 'angular-assignment' again to start the web server.
- Remember also to (re)start the Flask, TileStache and NGINX web servers again from part 3 of this workshop with their desktop shortcuts if they are needed for an assignment in part 4 or for some (much encouraged!) experimenting on your own!

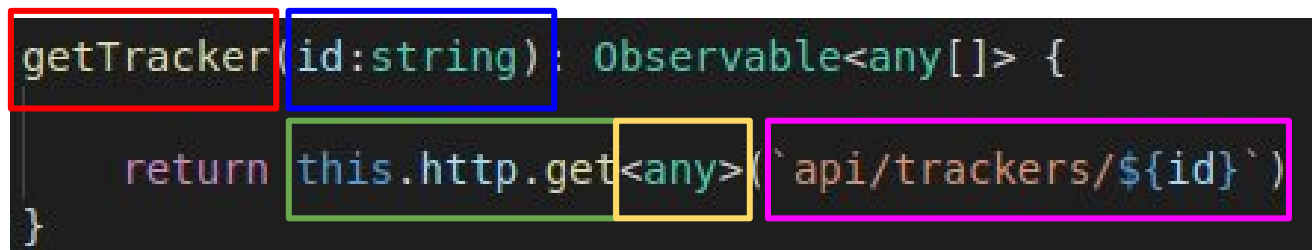
4.4 Assignment – Complete the TypeScript Service file

4.4.1 Angular Explained – Component Services

In the Angular framework the Component Services are programmed in TypeScript files and for the web app a service file is used to define functions that perform HTTP requests to our Flask API.

In the image below you can see an example of such a function, which is the `getTracker()` function that is defined in the service file to retrieve the data of just one (1) GPS Tracker of a ringed crane.

1. In this illustration we first declare the name of the function: `getTracker()`. (Red)
2. Then you declare the values that have to be passed on the function call. (Blue)
 - For the GPS tracker that is just the tracker ID which is passed as a string, not a number.
3. Next you declare the instance of the HTTP client and declare the type of HTTP request that need to be performed which is an HTTP 'GET' request to get data. (Green)
4. Then you declare the data type of the data that is going to be returned by the Flask API which is in this case 'any' to retrieve the tracker name from the database if there there is one (for now read it as a request to get data 'if any'). (Yellow)
 - By calling this function for all tracker IDs the web app collects the names of the cranes from the database to present a list of available datasets in the app's Settings menu.
5. Finally you declare the URL to which the HTTP request needs to be send (Purple)
 - In this case it's the URL: <http://localhost/api/trackers/>
 - Notice the tracker ID from the parameter 'id' in the function declaration is passed here as part of the URL to the Flask API!
 - Remember the matching '@route' declaration in the Flask app.py script is the API specification that is provided as a micro web service API by the Gunicorn WSGI web server for this URL.
 - This lets the browser send the HTTP request via the NGINX web server (as a proxy bridge server for the HTTP to WSGI protocol conversion) to get it to the Gunicorn web server that runs the Flask app to let the Flask app process the HTTP request!
 - Therefore the @route declaration in the Flask app.py script file and the URL specification in the component's service file to access the Flask API must match exactly!
 - Note: in app.py the route for the trackers function was declared with an 'api/' prefix just to be clear it was a route to a function to handle an API call and not for instance a route to a function to do something else, like for instance to display text on a HTML page.



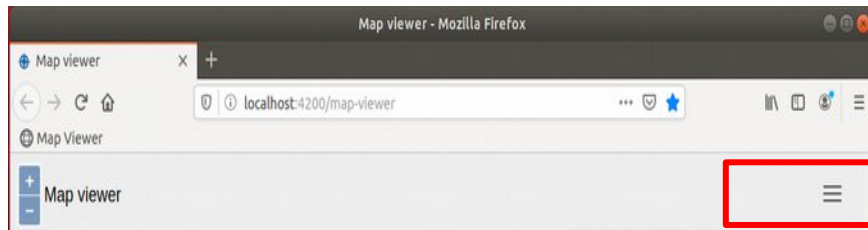
```
getTracker(id:string): Observable<any[]> {  
  return this.http.get<any>('api/trackers/${id}')  
}
```

Drawing 36: Function to request the data of one (1) Tracker using the TrackerID.

Note: the code shown in the illustration above, is explained in more detail in the GeoStack Course in the programming manuals and the source code related to creating the Angular web applications!

4.4.2 Assignment – Complete the Component Service File

Once the (Angular) web application is started and you navigate in your web browser to the URL: <http://localhost:4200> you will be greeted with the same screen as shown in the screenshot below.



As you can see, there is nothing to visualize (yet). Just on the left the '+' and '-' buttons for the zoom controls and the 'Map viewer' app name and on the left a 'hamburger' menu icon. (Red).

- To solve this in this assignment you need to get code up and running that gives an extra 'Gear' icon for a Settings menu a Settings on the right that is filled with the list of names of the cranes that represent the GPS track logs in our MongoDB datastore.
- The menu code is provided to be able to focus on getting an important part of the web service file completed to enable data access get the crane data for name list in the menu and data to display on a topographical map!

Assignment – Code Completion of the Component's TypeScript Service File

- The first assignment for the web app is to create the corresponding functions in the crane.service.ts file to make data requests to the Flask API.
- Click on the desktop shortcut called: "angular-service-assignment" as shown in the illustration below to open this TypeScript file in the (Atom) programming editor from the folder: /Part-4-Data-visualization/Sourcecode/Assignment/map-viewer/src/app/services/



- Now find the first assignment by looking for the header text as shown in the image below.
 - TIP: carefully read the inline comments provided in this file to make sure that you understand everything to complete the assignment.

```
#####  
#                                     START ASSIGNMENT 1                               #  
#####  
#                                     #  
#                                     #  
#    COMPLETE THE CODE WHICH IS USED TO OBTAIN ALL (CRANE) TRACKERS FROM OUR DATABASE    #  
#                                     #  
#####
```

Drawing 37: First Web App Assignment related to creating the 'Crane' service.

When you are finished with the assignment, simply save the file for the changes to be automatically added to the (Angular) web application.

- Notice the web application that is shown in the web browser is updated immediately, since it is running on the Angular Live Development Server.

IMPORTANT NOTICE

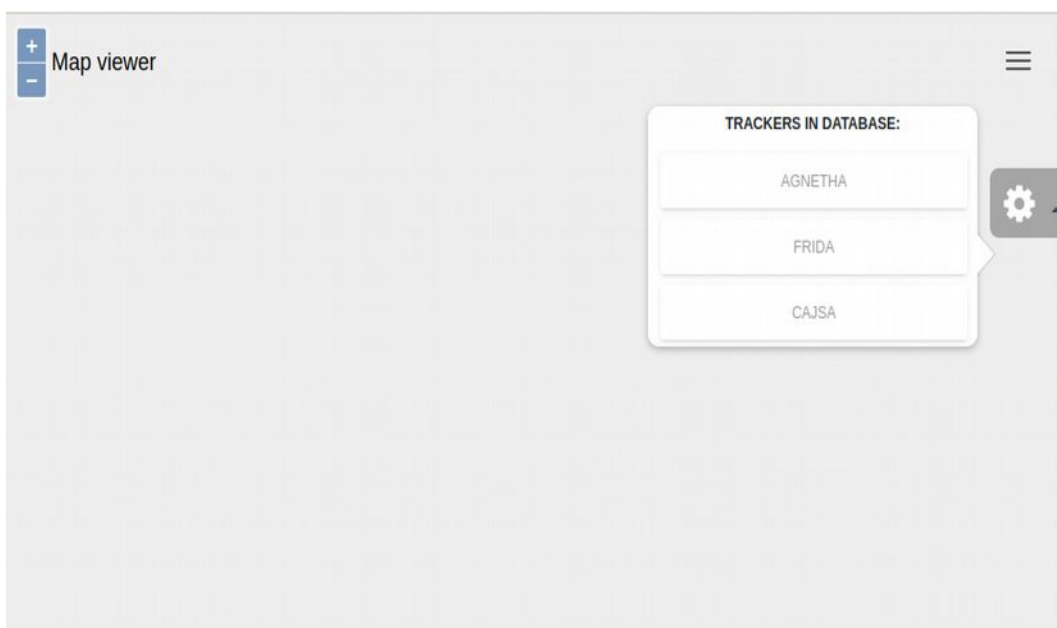
1. Notice in the file crane.service.ts of this assignment that passing a variable as a parameter in TypeScript requires the use of backtick quotes (``) instead of the regular single quotes (!).
 - Backtick quotes are officially called grave accent quotes.
2. If you want to experiment with the source code yourself this tiny detail might be an easy made typo that is hard to find for beginners when debugging error messages, so be aware!
3. In the source code of the crane.service.ts file you can find the format `\${variable name}` as part of a parameter, like for `\${id}` and `\${amount}` that requires the backtick quotes!
 - There is a comment block with this important notice in the source code too!
4. See this example from the source code for the getTracker() function where the URL parameter for the API call needs to be specified in the HTTP 'GET' request specification in backticks because this parameter holds the string variable `\${id}` as a part of it:

```
getTracker(id:string): Observable<any> {  
    return this.http.get<any>(`api/trackers/${id}`)
```
5. For more information see this web link:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

4.4.3 Assignment – Check the Result in the Web App’s Settings Menu

Assignment – Check the Settings Menu

- Finishing the assignment correctly will result data access to the crane datasets in MongoDB.
- This should result in a Settings menu which is filled with a list of names of the cranes for which there are GPS tracker datasets in the MongoDB datastore, so click the 'gear' icon of the Settings menu on the right to check if it matches the image below!
- Remember you can check the corresponding desktop shortcut for the solution if you want!



Drawing 38: The Settings menu shows a list of crane names retrieved from the crane database in MongoDB.

4.5 Coding Explained – The Angular and OpenLayers Magic

IMPORTANT NOTICE: remember, this section for newbies explains the last piece of Angular 'black magic' in order to understand how OpenLayers works in the web application!

4.5.1 Angular Explained – Some Basics on the TypeScript Code File

At this point all the crane names that represent the GPS Crane Trackers are in a list in the Settings menu of our web app but there are still a few steps to go to visualize crane flight paths on a map. Let's explain a few things related to creating the base of a component in the TypeScript Code file.

- The name of the web app is 'map viewer' and the name of the main component is 'map'.
- The code is in the files: map.component.html, map.component.ts and map.service.ts.
- The map.component.ts file is where the magic happens, so let's take a look how it works!

The TypeScript Code file is map.component.ts and it has the following 4 important code parts:

1. A component contains import statements used to load modules and services in the component as shown in the image below.

```
import {CraneService} from '../services/crane.service'
```

2. A component also contains metadata definitions specified as key-value pair declarations as shown in the image below:
 1. Selector (Red): to call the component by name in another file such as in a HTML file.
 2. templateUrl (Blue): to define the location of the HTML file for the component layout.
 3. Providers (Green): a list of services that are going to be used in the component.
 - In this case we are only going to use a single service, the CraneService.
 - Note: an optional key-value pair for a styleUrls can also be added which defines the location of the style sheet file for the styling of the component.

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

3. A component has a class definition which contains a constructor statement that is used to instantiate services. In this case the 'Crane' service as shown in the image below. (Red)

```
export class MapComponent implements OnInit {  
  /** ...  
  constructor(private _CraneService: CraneService) {}
```

4. A component always contains the default function: "ngOnInit()" (Angular On Initialization).
 - This function is automatically run when the component is loaded and in this case the initialization function for the map component of the web app is programmed to first create an OpenLayers map canvas and then get the data of all the crane GPS trackers.

```
ngOnInit() {  
  this.createMap();  
  
  this.getTrackers();  
}
```

4.5.2 OpenLayers Explained – Some Basics on Coding Layers

OpenLayers is a geospatial JavaScript framework to draw 2D digital topographical base maps and then display one or more transparent map layers as data overlays on top of that base map.

- A transparent topographical map overlay is called 'oleaat' in Dutch, 'carte-transparent' in French and 'Deckblattkarte' in German.

During this workshop you are going to use the following 2 types of map layers:

1. a **TileLayer**: to display a digital topographical map consisting of map tiles;
2. a **VectorLayer**: to display data points, lines and icons by using GPS coordinates.

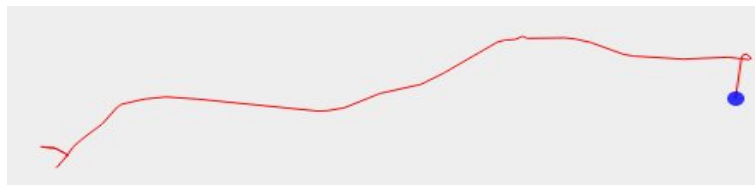
You are going to create 4 map layers in this workshop version of the web application:

1. a **Base layer** for an OpenStreetMap map with map tiles from the TileStache tile server;
2. a **Point layer** to plot the data points by using their GPS coordinate values;
3. a **Line layer** to draw the line segments between the data points;
4. a **Marker layer** to display an object icon when a time series animation is playing. The marker location is constantly updated to the next data point location in the time series.

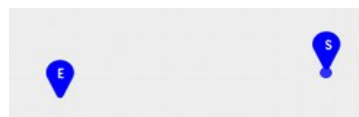
The idea of working with map overlays is shown in the picture series below of a car route:



Drawing 39: Base Layer --> Draw a Topographical Base Map



Drawing 40: Line Layer --> The lines connect the data points of the car route.



Drawing 41: Marker Layer --> Extra layer for Start and End icons.

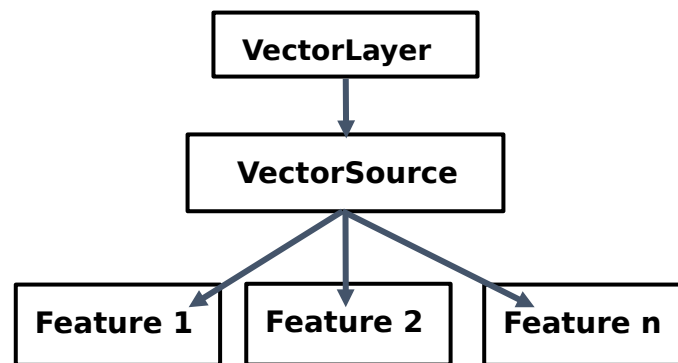


Drawing 42: Marker Layer --> Display a Blue dot at the starting point as the car's object marker to animate that over the route in a time series playback.

Let's zoom in a little more on how a **VectorLayer** is programmed in OpenLayers to start learning how to read the source code to plot your own data layers on a digital topographical map.

A VectorLayer contains a VectorSource which contains a list of features. See the image below.

- The list of features contains all the objects which are going to be displayed on the map.
- The objects could be data points, arrows, markers.



A Feature (geometry) consists of:

1. a Styling: e.g. the line color, line width or line styling;
2. a Geometry type: which could be:
 1. a Point: a set of coordinates;
 2. a LineString: a set of multiple points.
3. a Geometry: which contains the values (x,y values) of the data point coordinates or a list of multiple data point coordinates.
 - Note: read the inline comments in the source code for more information related to a feature geometry!

Let's look at how that translates to the inverse source code structure sequence in the image below.

- This is where the magic happens, although the first time it looks more like Black Magic!
- Indeed, it takes a little getting used to the coding style of OpenLayers to learn how to read and program it but notice the consistent use of key-value pairs to specify the parameters!
- Note: the 3 architecture layers are 'by design' in OpenLayers and that's why you will always see code to create the objects for the features to display and the sources and layers to group the features!

```
let pointFeature = new ol.Feature({
  type: "Point",
  geometry: [0,0],
  style: pointStyle
});
```

```
let pointVectorSource = new ol.source.Vector({
  features: [pointFeature]
});
```

```
let pointVectorLayer = new ol.layer.Vector({
  source: pointVectorSource
});
```

4.5.3 The Map Component Explained – map.component.html

How the Map Component Works --> Useful information for the Programming Assignments!

The HTML and TypeScript files of the map component are in the folder: /map-viewer/src/app/pages

map.component.html --> How the Settings Menu Works!

This file is about getting 3 objects displayed in the layout: a map, a marker and a menu.

To do that there are 3 corresponding <div> elements that provide the display with clickable objects (mostly buttons) that are actionable by calling a function to do (e.g. get data) or display (e.g. a map) something.

1. <div id="map" ...>: this displays the topographical base map with the OpenStreetMap map tiles from the TileStache tile server full screen and it provides the zoom buttons on the left to zoom in on the map.
 - Remember the map object itself is a blank canvas on which the base map with map tiles is drawn as the baseLayer overlay in map.component.ts!
2. <div id="marker" ...>: this displays the 'marker' overlay as defined in map.component.ts for the marker icon to follow the location of the crane in the animation of the flight path. The marker itself is a blue dot created by specifying a square with '100%' rounded edges.
3. <div class="fixed-plugin"> this displays the 'Gear' icon for the Settings menu on the right and when clicked it provides the drop-down menu with the menu buttons for the crane names to select the dataset (transmissions) for that crane and the menu buttons to run the animation with the time series playback of the flight path for the selected crane.

What happens in the map.component.html file for the menu is fairly straight forward to understand but learning that to program yourself might be quite a challenge, so here is some help:

1. A Settings menu is specified as a drop-down menu that can be toggled open and closed with a mouse click.
2. Also the icon is set to the Font Awesome 'Gear' icon with: <i class="fa fa-cog fa-2x"> </i>
 1. <https://fontawesome.com/v4.7.0/icon/cog> (fa-cog = icon and fa-2x = size)
 2. <https://fontawesome.com/v4.7.0/examples/>
3. Then the menu gets a title.
4. The next step is to create an Angular container to get the contents for the menu in a for-loop (ngFor) from the tracker list that was 'exported' by the map.component.ts file of map the component!
 - This will list the names of the cranes as menu items that are clickable buttons.
 - Therefore also notice the specification for the 'click' event in the drop down menu which will get the data for the selected crane by passing the tracker ID by calling the getTransmissions() function.
 - Notice the odd \$-notation in 'tracker._id.\$oid'. This is a full field notation only used in the workshop to pass to MongoDB because it was programmed a little simpler.
 - In the course it will be the short object notation of 'tracker._id' as you would expect.
5. Finally the last menu option in the Settings menu are the two menu items to trigger the Start and Stop of a route navigation by calling the animateRoute() function.
 - For the selected crane in the drop-down menu this will give a time series playback of all the datapoints in the GPS track log to show the flight path of the crane with a marker icon that is displayed to visualize the location of the crane in the playback.

4.5.4 The Map Component Explained – map.component.ts

map.component.ts --> How Plotting Geospatial Data on a Topographical Map Works!

This file is about programming the 'Payload Code' with the 'Business Logic' to get the crane data from the MongoDB datastore and plotting it with OpenLayers on an topographical map from OpenStreetMap.

- To do that the app needs to be able draw the map and to 1) plot just the data points, 2) the data points connected with lines and 3) show an animation for a time series playback of the flight path of a crane.

The map.component.ts file of course also holds the programming assignments for the workshop and in this file the following happens:

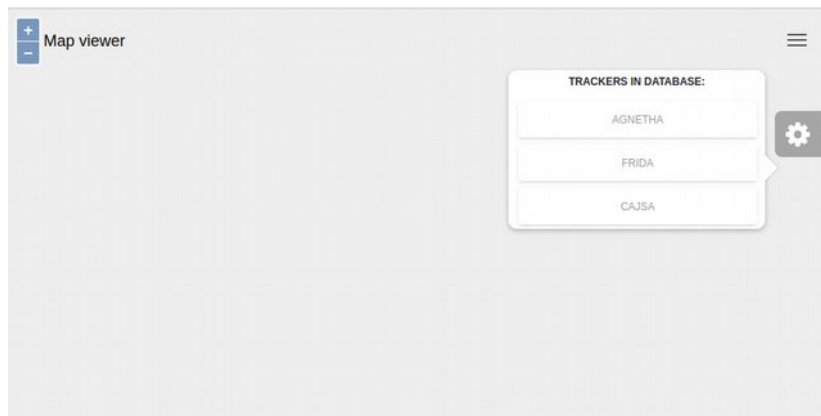
1. First the import statement for the CraneService is required to use the crane.service.ts file to get HTTP requests to the Flask API to get crane data from the MongoDB datastore.
 - To do that, the CraneService is declared as a service provider in the @Component decorator and connected to the HTML file to get actionable click events in the menu.
 - In the class constructor for the MapComponent the CraneService is specified to make the Flask API calls later.
2. Then in the Angular component initialization function ngOnInit() calls 2 functions:
 1. The canvas for OpenLayers to draw maps is created.
 2. The tracker information of the GPS trackers is retrieved from the crane database in MongoDB.
 - In TypeScript the code for these initialization functions is programmed next.
3. Next the list variable 'trackers' that is used in the for-loop in the HTML file is filled in the getTrackers() function and once the tracker IDs are known the GPS transmissions that were sent by the GPS trackers and that are logged in the GPS track log in the crane database can be retrieved too by using the getTransmissions() function.
4. Then the canvas of the OpenLayers map needs to get a topographical base map by defining the BaseLayer, which is a programming assignment.
5. The next step is to specify a PointLayer in OpenLayers to plot the data points of the GPS locations of the cranes on the map, which is also a programming assignment.
6. Then for the LineLayer the same to connect the data points with line segments for a better view of where the crane has been compared to the hard to interpret cloud of data points.
 - The programming assignment here is to add the LineLayer to the OpenLayers map in the same way as the PointLayer was programmed to overlay the base map.
7. Finally the last programming assignment is to complete the startAnimation() function

4.6 Assignment – Complete the 2D Map Viewer Web App

Now it's time to program the 'Payload Code' into the map viewer web application which belongs in the `map.component.ts` file as explained above in the section 'Angular Explained – Some Basics on the TypeScript Code File'.

4.6.1 Assignment – Check the Angular Development Environment

1. Check if the (Angular) web application is still running.
 - If this is not the case you can restart the Angular application by clicking on the desktop shortcut: "angular-assignment".
 - Then wait till the application is restarted and navigate to the URL <http://localhost:4200>
2. Check if the previous assignment has been correctly completed.
 - The Settings menu (Gear icon) must contain the list of menu items with crane names that was retrieved from the MongoDB datastore as shown in the image below.
 - If you click on one of these crane names nothing will happen, Yet!



Drawing 43: The still unfinished 2D Map Viewer Web App.

4.6.2 Assignment – Open the TypeScript Code file `map.component.ts`

In this assignment you are going to complete the code that, when you click on a crane name in the Settings menu, the logged transmissions the corresponding GPS tracker are retrieved from the Crane Database in MongoDB and that they are visualized on a topographical map from OpenStreetMap.

To achieve this code completion in the file `map.component.ts` you will need to complete 4 tasks:

1. create the base layer for the topographical map;
2. create the features to specify the feature geometry;
3. visualize the data points and line segments on the topographical map;
4. create a time series animation of the data points to visualize the traveled route.

To find the assignments to visualize GPS tracker data on a topographical map with OpenLayers, click the desktop shortcut 'angular-openlayers-assignment' as shown in the image below.

- This shortcut will open the `map.component.ts` file in the (Atom) Programming Editor from the folder: `/Part-4-Data-visualization/Sourcecode/Assignment/map-viewer/src/app/pages/`



4.6.3 Assignment – Plot Geospatial Data on a Topographical Map

Now scroll through this TypeScript file `map.component.ts` and look for the 3 programming assignments which are indicated by comment blocks as shown in the image below.

```
#####  
#                               #  
#           START ASSIGNMENT 2           #  
#                               #  
#####  
#                               #  
#           COMPLETE THE CODE FOR THE CREATION OF THE BASELAYER           #  
#                               #  
#####
```

- There are 3 of these comment blocks in the file for the programming assignments for data visualization in OpenLayers to learn to plot geospatial data on a digital topographical map:
 1. Creation of the Base Layer
 2. Creating the Features
 3. Add the Line Layer to the OpenLayers Map
- Remember, these steps to code completion were explained earlier in this cookbook in the section 'Assignment – Understanding the Programming Steps!'.
- Tip: read the assignments and inline comments in the TypeScript source code carefully!
 - Remember, the programming steps are explained in the '//TODO' comment lines.
- Tip: save often and after finishing each assignment check if your new code block works!
 - Remember, when you are finished with an assignment, you only have to save the file to see the live changes in the web browser because the web app is running on the Angular Live Development Server.

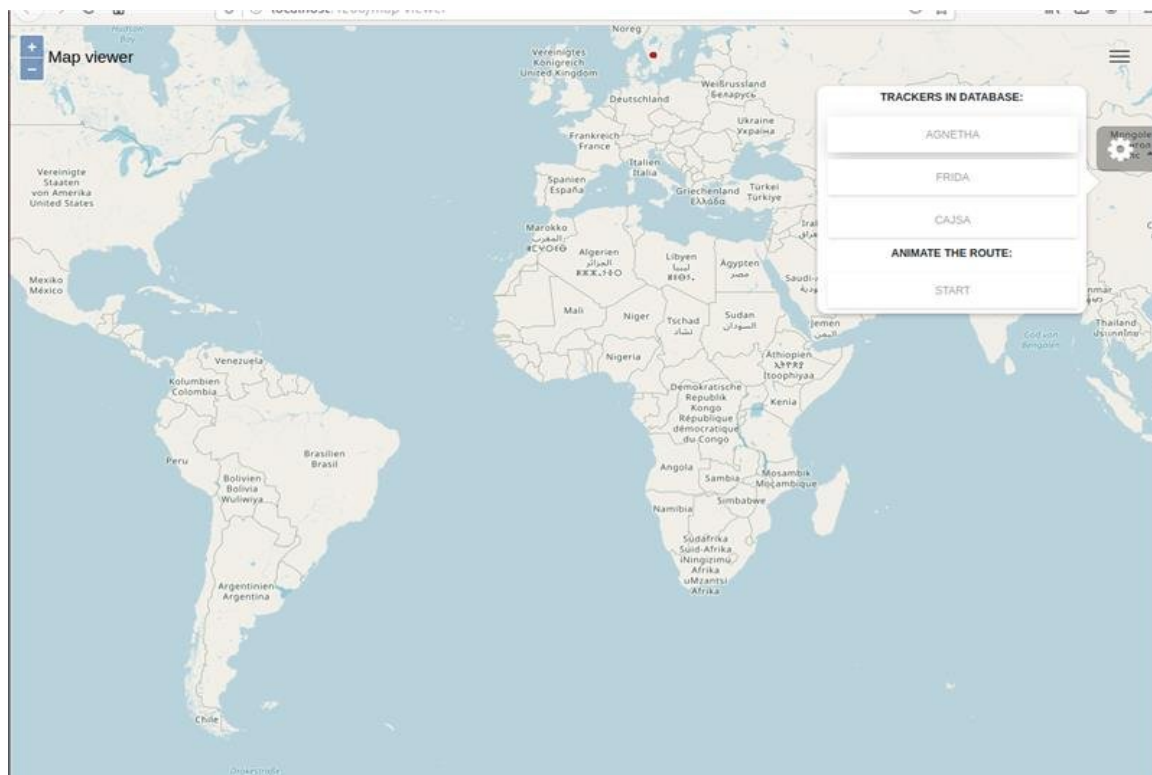


Illustration 5: After completing this assignment you should see the topographical base map centered at [0,0], the junction of the Prime meridian and Equator and when the 'Gear' icon of the Settings menu is clicked the drop-down menu is shown with the crane names of the GPS trackers and the route animation option to play a time series animation of a crane flight path.

4.6.4 Assignment – Route Animation --> Playback of a Time Series

Then there is a final 4th programming assignment at the very end of the map.component.ts file to complete the code for the route animation to get a very nice visual of the flight path of a crane.

- What the animation will do is a time series playback of each data point where a marker icon shows the location of the crane.
- In the workshop the marker icon is just a simple dot but in the GeoStack Course you can program your own marker icon like a square, triangle or even a small crane icon if you like.

4.6.5 Assignment – Checking the Crane's Flight Paths

- Static dataset display: first click a crane in the Settings menu and see where the dataset appears on the map and then drag the map with the mouse and zoom in until you can see the entire flight path with the connecting lines between the data points.
 - Notice when you zoom in there are 'point clouds' of many points in the flight paths and these are the places where the crane is not flying but on the ground during the day at feeding grounds, bathing and resting places or breeding places and at night at the sleeping place of a crane dormitory where they sleep in flocks or at a nesting location. A typical daily, non-migratory, flight path will be at very low altitude of 25 - 250 meters.
 - The lines in between these point clouds are the actual flight paths. A very small point cloud of just a handful of points in a flight path might indicate a circular motion on a thermal upstream to gain altitude without wasting flying energy for a long glide path to a next thermal bell to go up again or even better to find a jet stream at high altitude. When Cranes are migrating the large flocks of hundreds of cranes can easily go up to somewhere between 750 – 1,500 meters to find favorable winds because they really are the 'Sailors in the Skies'!
- Dynamic dataset display: finally click the 'START' button to animate the route which will show a time series playback of the flight path.
- Remember: when you select multiple cranes the previous flight path will still be displayed on the map because there is no map refresh in the simplified web app for the workshop! In case there is substantial overlap, simply restart the web app and select another crane to get a clean map display.
- Also remember: in case you get stuck during these last assignments, you can click on the solution desktop shortcut to see how the simplified 2D Map Viewer web app works and go to the solutions folder to study the completed source code.

4.7 Challenge – Add the Car routes to the Web App

Now you have completed the code to visualize the crane datasets on a topographical map we leave you with a last challenge for this workshop to complete the simplified 2D Map Viewer web app with the visualization for the car routes!

- In this way you can check for yourself if you have build the knowledge and skills to do so!
- It should take about 1 – 2 hours.

Some steps to take are:

1. Add an extra drop-down menu for the car routes in the file `map.component.html`
2. Fill the drop-down menu with a list of buttons with the trail names of the car routes.
3. Program the function call for the button event to get the list of signals (= car route dataset).
4. Add to the `map.component.ts` file everything you need for the Trails and Signals!
 1. Remember you will need a 'CarService' with a 'car.service.ts' file for that too!
 2. Also remember you already learned how to program the database queries for the Trails and Signals in Part 3 of the workshop.
 3. Of course, don't forget to add a car route animation too with the corresponding menu choice in the drop-down menu!
5. Change the Web Map Server from OpenStreetMap to Thunderforest for another map style!
 1. Remember from Part 3 you need to get a free account for the API key to try this.
 2. Then make sure the API key is in the tile server configuration file `configuration.cfg`
 3. Finally make the Angular web app use the tile server's 'landscapemap' service.

To make this a real challenge: see if you can center the map for the car routes with a GPS coordinate and an appropriate zoom level to show just that part of the South-West of the Netherlands that nicely shows the 3 car routes.

1. Remember the explanation in the section 'Scope Limitation --> Centering the Map at [0,0] explained!' and follow the web link there for the GPS coordinate example for Bruxelles!
2. A nice center point for the map is the traffic junction 'Hellegatsplein' built 'the Dutch way' by the way in the middle of some main water ways!
 - The Hellegatsplein is the meeting point,, for the highways A59 from the province of Noord-Brabant and the A29 from the province of Zuid-Holland and the county road N59 from the province of Zeeland.
 - The GPS coordinates are: 51.700163 North Latitude, 4.392988 East Longitude
 - Try a preview on OpenStreetMap with zoom level 11 for the perfect 'helicopter view':
 - <https://www.openstreetmap.org/#map=11/51.700163/4.392988>
 - Note: click the almost invisible light grey 'X' just below the blue 'Go' button to close the information window to see the whole map.

That's all folks!

We hope you enjoyed this workshop and continue to learn more in the GeoStack Course!
