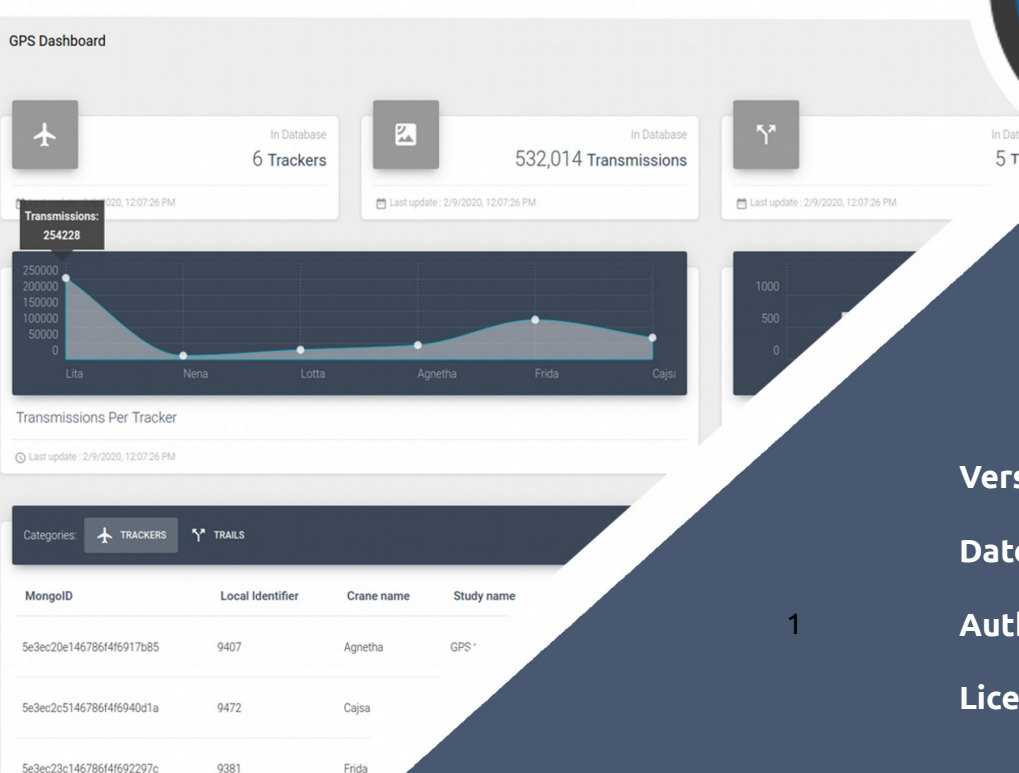
Cookbook

Creating the GeoStack Course VM



Version : 1.0

Date : 08-09-2020

Author : The GeoStack Project

Open Content License

Purpose of this document

This cookbook serves as manual for installing all the software, tools, modules and libraries required to create a fully functional Open Source Geographical software stack which does not need an active network-connection to visualize Geospatial datasets on 2D maps and 3D maps.

During this cookbook references are made to other cookbooks and programming manuals. When such a reference is made the cookbook and / or programming manual in question should be read before continuing to read this cookbook.

First there will be an introduction to the tools and software products used in the GeoStack. During this introduction a global description regarding the functioning of the GeoStack and what task each component fulfills in relation to the complete Geospatial software stack.
These descriptions will be given using a diagram which shows the data flow between the GeoStack components.

After discussing the software, tools and data sets used during the **Beginner Course Open Source Geospatial Programming for Data Scientists**, a description will be given regarding the 2 ways you can go about installing the GeoStack components and creating the web applications.

As mentioned before, this document will give a global description regarding each of the components in the GeoStack. For an in-depth explanation regarding the components you should read the cookbook or programming manual related to the component in question.

If you decide to install the GeoStack software using the automatic installation scripts you will have the complete software stack up and running in no time. If you decide to install the GeoStack software manually, using the programming manuals you will build each component from scratch.

Prerequisites for the Workshop are as follows:

- **A laptop with an active network-connection.**
- **60GB of free space on you hard drive.**

During the GeoStack Workshop you installed VirtualBox on your host system. If you have not conducted this workshop yet, you should start by doing so.
If you have VirtualBox installed you are going to create a new Ubuntu Virtual machine in which you are going to install the software and tools required for the GeoStack.

You can decide whether you want to install the GeoStack using the installation scripts provided in the folder ”Building-the-VM-using-the-installation-scripts” or to install the Complete GeoStack from scratch using the cookbooks, Jupyter Notebooks and programming manuals provided in the folder “Building-the-VM-from-scratch-using-the-manuals”.
Both folders are located in the GitHub repository GeoStack-Course which you will clone later on!

After completing this Cookbook you continue the GeoStack Course by following the manuals to learn how to program the 3 web applications of the Dataset Dashboard, 2D Map Viewer with OpenLayers and the 3D Map Viewer with Cesium.

Now let’s start of with an introduction to the GeoStack from an architectural viewpoint!

Table of Contents

1 Introduction to the GeoStack

1.1 The Software Ecosystem for the GeoStack Course

The purpose of the GeoStack is to provide a free, open source, light-weight solution for visualizing data on digital maps. The GeoStack is a light-weight software stack and can run stand-alone without an active network connection.

The official website of The GeoStack Project (https://the-geostack-project.github.io/) contains a global description of what you will learn during this course and the functionalities that are present in layers of the geospatial software stack, or GeoStack for short!

The GeoStack is divided in 6 software layers as follows:

1) **Virtualization software**
   Each GeoStack component is 'Dockerized' and thus can run separately from the other components in its own Docker container. The components are also installed in one Virtual Machine running Ubuntu Linux in VirtualBox so everything runs locally on just one (1) VM!

2) **General software**
   To create the code and documentation, tools such as Atom and LibreOffice are used. These kind of software products are covered in the general software section.

3) **Data analyses and processing software**
   During the cookbooks you are going to work with multiple types of data formats. These data formats have to be transformed and then stored in the corresponding data store. To be able to do this you need the data-analyses and processing software.

4) **Backend software**
   The backend consists of 2 datastores that each serve their own purpose and store a different type of dataset.

5) **Middleware software**
   The Middleware consists of multiple web servers and an API. The middleware contains the TileStache Tile server, Cesium Terrain Server, Flask API and the NGINX Web server. The middleware is where all the 'magic' happens for the web apps 'business logic'.

6) **Frontend software**
   The frontend consists of 3 web applications, each serving their own purpose. These web applications are as follows:
   - Dataset dashboard: An application which displays all the datasets used in the GeoStack using interactive graphs and tables.
   - 2D Map Viewer: An application which displays all the datasets in 2D on a 2D map using the Geospatial Framework OpenLayers.
   - 3D Map Viewer: An application which displays all the datasets in 3D on a 3D map using the Geospatial Framework Cesium.

You are going to learn how to install the GeoStack software ecosystem in the order of the software layers as described here above.

To learn fast: watch the video tutorial clips on the YouTube Channel of The GeoStack Project!
- Learn more here: https://www.youtube.com/channel/UCiZEImhO8r-LMAWh-KQiH6g

Note: the source code text in many of the code examples can be copied and pasted and script files for all the code examples are provided in the folders in the cloned GitHub repository!

1.2 The Geospatial Ecosystem – A Global Overview!

Of course there is more geospatial software than what is used in the GeoStack self-study course for data scientists and not only is there much more Open Source Software but of course there are many Open Standards and there is a lot of Open Content too!

Next to the 'Open' world there is of course lots of commercial software, proprietary standards and payed-for content available in the geospatial realm.

The overview table below is only intended to give a global 'helicopter view' of what is available in the geospatial domain to enable you to find more information if you are interested!

DISCLAIMER: the table is not intended to be complete or correct! It is provided to give you a broader view on the geospatial world! Go Surf the Internet to get yourself better educated!

Category	Geospatial Software, Standards & Content – Products & Providers
Open Source	❖ OSGeo.org: release of the OSGeo live DVD with a broad selection of high quality Open Source Software geospatial products. ❖ OpenGeospatial.org: an international consortium van 500+ geospatial companies from all over the world for Geospatial Open Standards. ❖ Foss4G.org: an annual conference on Geospatial Innovations and Geospatial Open Standards, Open Source Software and Open Content. ❖ Geonovum: a foundation for geospatial advice and a knowledge center of and for the Dutch Government with a lot 'Open Geospatial' knowledge too. Check if you have something like that in your country!
Commercial	❖ ESRI with its ArcGIS Suite: a defacto standard in the geospatial world. ❖ Hexagon with its Hexagon Spatial Suite: includes the former Luciad Suite ❖ BAE Systems with its Geospatial eXploitation Products suite (GXP). ❖ Planet.com with its geospatial cloudservices and (satellite) imagery: they bought Boundless for its former Open Source Geo Suite.
Mobile	For mobile geospatial applications, working with single file datastores is the norm because a mobile device doesn't run a geospatial database server. The GeoPackage file format is popular and there's use of GeoJSON en JSON files and a single file database of SQLite with the SpatiaLite file format. Global Overview of Development Software for Mobile Geospatial Apps: ◆ National Geospatial-Intelligence Agency: this agency has 3 Open Source Software Development Kits (SDK) to work with GeoPackage files: o NGAGeoINT IOS: a GitHub package for IOS. o NGAGeoINT Android: a GitHub package for Android. o NGAGeoINT Javascript: a GitHub package for JavaScript programming. ◆ Ionic with Cordova: an Open Source development environment for Android and iOS for JavaScript + TypeScript with Angular. ◆ Commercial mobile app solutions can be found for instance at Hexagon with Luciad-RIA at ESRI with ArcGis Web.

Geospatial File Formats	❖ Shape file: an ESRI proprietary, but publicly documented, distribution file format for geospatial data which is a defacto standard. ❖ SQLite + SpatiaLite: SQLite is the name of both the Open Source relational database (software) as the accompanying single file datastore. SpatiaLite is the geospatial version of this database and also the name of the file format. ❖ GeoPackage: this 'fairly new' file format (since 2015) for digital topographical maps is an Open Standard of the Open Geospatial Consortium (OGC). It is both suited for use as a distribution file format (also intended to improve on the classic shape files) and as a single file datastore for light web applications and mobile apps. ❖ JSON: is an Open Standard that can be used both as a distribution file format and a single file datastore. There are three (3) geospatial versions in use, the popular GeoJSON, the less known TopoJSON and OSM JSON (for OpenStreeMap). ❖ OpenStreetMap + OpenSeaMap file formats: there are 3 distribution file formats in use which are all Open Standards: OSM XML, OSM JSON and OSM PBF (Protocolbuffer Binary Format).
Geospatial Datastores	❖ PostgreSQL Spatial: an Open Source relational database with the geospatial extension package PostGIS. ❖ MariaDB: an Open Source relational database with a geospatial extension package. MariaDB is a completely renewed drop-in replacement product for the Open Source database Oracle MySQL. ❖ Oracle Spatial: a commercial relational database with a geospatial extension package. ❖ Microsoft SQL Server Spatial: a commercial relational database with a geospatial extension package. ❖ MongoDB: a commercial schemaless database with storage of spatial data in (Geo)JSON documents. It has a free community edition and its source code is available on GitHub. It has a license limitation that forbids to offer MongoDB as part of a free public web service without disclosing all your own software of that service too (a strong 'copyleft' clause)!
Geospatial Search Engines	❖ Lucene: the Open Source Core Search Engine of the Apache Foundation with spatial search functions. ❖ Solr: the Open Source search engine of the Apache Foundation, that uses Lucene as its core search engine. ❖ ElasticSearch: a commercial search engine from Elastic build around Lucene that is partially Open Source. The source code is on GitHub. - Elastic provides a free Open Source community edition supplemented by a commercial plugin package (X-pack) for extra functionality. - There is an active Open Source project 'Open Distro for Elasticsearch' aimed at corporate use with a goal to provide free Open Source alternatives for X-pack modules, currently mainly for monitoring and access management. ❖ Idol (which includes the former Autonomy search engine from HP): this is a commercial search engine from Microfocus with proprietary geospatial search functionality. ❖ Sharepoint Search (included the former Fast search engine): this is an application search engine from Microsoft that is built into the commercial Sharepoint web document management system with proprietary geospatial search functionality. It may or may not merge into the developing cloud version of Sharepoint Online (SPO). ❖ Internet search engines: they all have 'Maps' services and spatial search services in both their start pages and in their search result web pages.

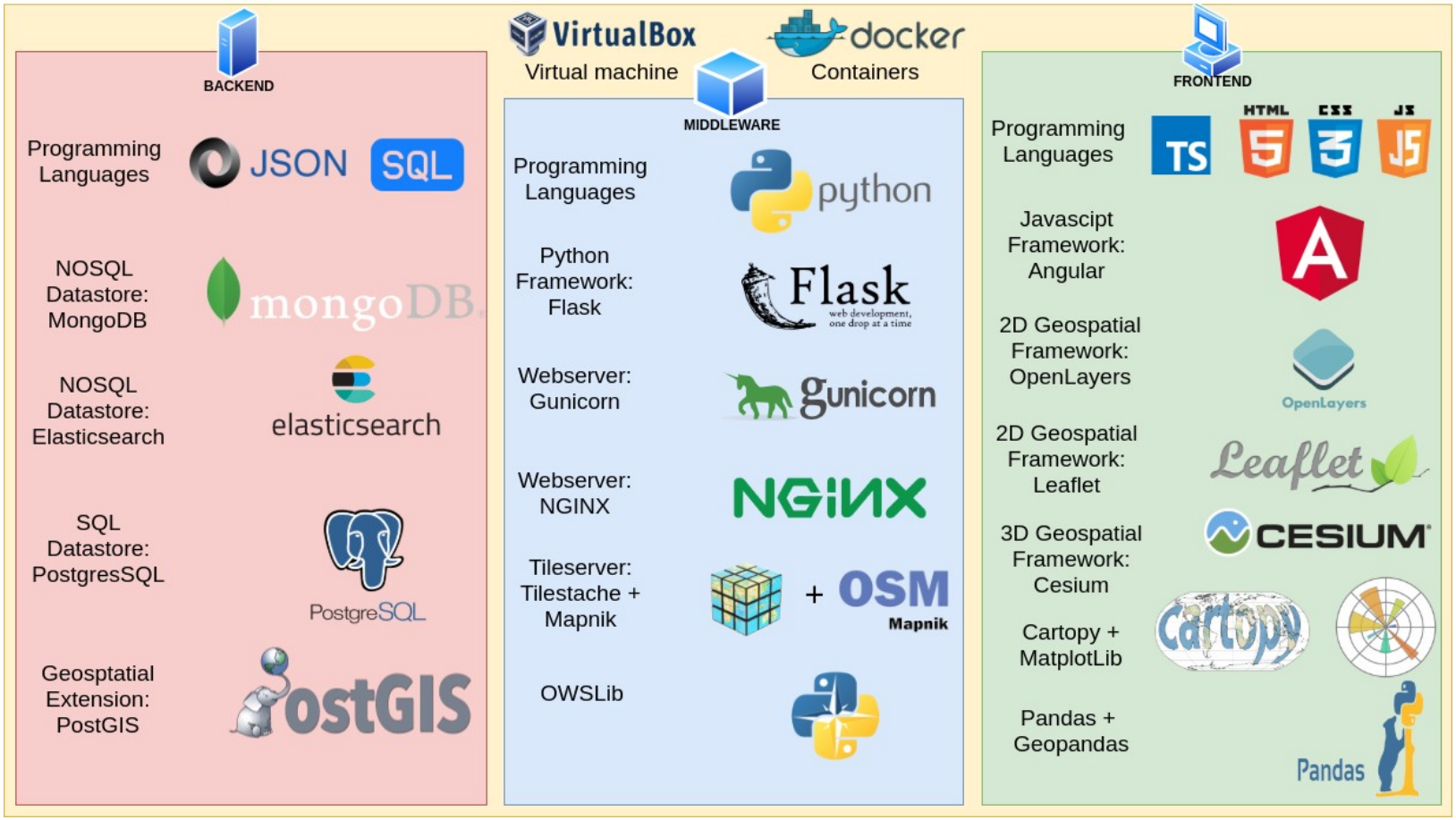
Geospatial Open Content (Just a few examples!)	❖ OpenStreetMap + OpenSeaMap: its free content is used a lot on the Internet, by corporations and in Datascience. Also used to facilitate creating new topographical maps fast for humanitary disaster areas. ❖ Publieke Diensten Op de Kaart (podk.nl): Translated this means 'Public Services on the Map' which is the website for all public geospatial datasets of the Dutch government. Check if you have a website like that in your country too! ❖ Actueel Hoogtebestand Nederland (AHN): dataset with Digital Elevation Map (DEM) files of the Netherlands from the Dutch government infrastructure organsation Rijkswaterstaat. Check if you have high resolution DEM files for your country too! DEM files are made with laser altimetry using LiDAR technology and are used to create 3D topographical maps. These files are very large!!! ❖ DEM files European Union: datasets of all countries are published by the EU project Digital Elevation Model over Europe (DEM-EU) of the European Environment Agency / Copernicus Land Monitoring Service. ❖ DEM files of the World: the NASA used the Space Shuttle create a 3D terrain profile of Planet Earth! - See the results with 30 meter resolution of the Shuttle Radar Topography Mission here: https://www2.jpl.nasa.gov/srtm/ ❖ Geonames.org: a website with a large dataset of geographical names with GPS coordinates for geolocation services. The dataset contains more than 20 million locations for places, lakes, rivers, mountains etc.! ❖ World Port Index: dataset with all sea ports of the world. The GPS coordinates are unfortunately not very accurate. ❖ Language datasets: remember, you might need translations and synonyms! For instance of location names for geolocation services.
Geospatial Commercial Content (Just a few examples!)	❖ ESRI and BAE Systems: eg. Maritime Charts ❖ Hexagon and Planet.com: eg. Satellite Imagery ❖ Garmin and TomTom: eg. topographical maps, route planners ❖ AIS data: subscriptions to get ship locations is broadly available. - For Terrestrial-AIS eg. from MaritimeTraffic, VesselFinder or AISHub (AISHub is free but you have to deliver AIS data you received yourself in return). - Satellite AIS data for instance from ExactEarth or OrbCom. - AIS-data is also provided for instance by EMSA, the European Maritime Safety Agency. ❖ ADSB data: subscriptions for airplane locations is also broadly available. ❖ Geospatial related content: weather data, oceanographic data, traffic data etc. is all area located data. ❖ Object related content: for instance a subscription to a photo service for pictures of ships and planes to show in an application next to plotting a location of an observed object on a topographical land or nautical map.

Table 1: Helicopter view over the Ecosystem of Geospatial Software, Standards & Content.

1.3 The GeoStack work environment

During the **Beginner Course Open Source Geospatial Programming** you will be working with lots of different tools and software products. The most important tools and software products are shown in the illustration below.

- A small introduction to each of the tools and software products is given on the next pages.



1.3.1 The Virtualization software

VirtualBox is used to run a Virtual Machine with Ubuntu Linux completely isolated from the host operating system on your computer to isolate the software that is used in the GeoStack Course. With Docker containers and Volumes you will learn to isolate software components and data from the Ubuntu Linux OS for easy maintenance and distribution to run them on other computers. It helps data scientists cooperate with others!

VIRTUALIZATION

VirtualBox is a free, open source solution for running other operating systems virtually on your PC.
With VirtualBox, you can install any version of an operating system, such as Linux, Solaris, and other versions of Windows (as long as you have the original installation files, of course) and run them within your current version of Windows.
The first thing you notice about VirtualBox is that it's extremely easy to setup and use.
VirtualBox holds your hand through the whole process so you never feel out of your depth.

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.
Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.
And importantly, Docker is open source. This means that anyone can contribute to Docker and extend it to meet their own needs if they need additional features that aren't available out of the box.

1.3.2 The Backend software

BACKEND

JSON:(JavaScript Object Notation) is a lightweight data-interchange format.It is easy for humans to read and write.

SQL: stands for Structured Query Language. SQL is used to communicate with the PostgreSQL database.

MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents

MongoEngine is a Python Object-Document Mapper for working with MongoDB.

MongoDB Compass analyzes your documents and displays rich structures within your collections through an intuitive GUI.

PostgreSQL, also known as Postgres, is a free and open-source relational database management system (RDBMS)

PostGIS provides spatial objects for the PostgreSQL database, allowing storage and query of information about location and mapping.

PgAdmin is a web-based administration tool for PostgreSQL.

1.3.3 The Middleware software

MIDDLEWARE

Python is an interpreted, high-level, general-purpose programming language.

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries.

***PyMongo* is a Python distribution containing tools for working with MongoDB**
Psycopg is the most popular PostgreSQL database adapter for the Python programming language.
Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety

***TileStache* is a Python-based server application that can serve up map tiles based on rendered geographic data**

***Mapnik* is an open source toolkit for rendering maps. Among other things, it is used to render the four main Slippy Map layers on the OpenStreetMap website.**

The Gunicorn "Green Unicorn" is a Python Web Server Gateway Interface HTTP server.

***NGINX* is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server.**
***NGINX* is known for its high performance, stability, rich feature set**

1.3.4 The Frontend software

FRONTEND

HTML provides the basic structure of sites, which is enhanced and modified by other technologies like CSS and JavaScript.
CSS is used to control presentation, formatting, and layout.
JavaScript is used to control the behavior of different elements.

TypeScript is an open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript, and adds optional static typing to the language.

Angular is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS.

OpenLayers is an open-source JavaScript library for displaying map data in web browsers as slippy maps. It provides an API for building rich web-based geographic applications similar to Google Maps and Bing Maps.

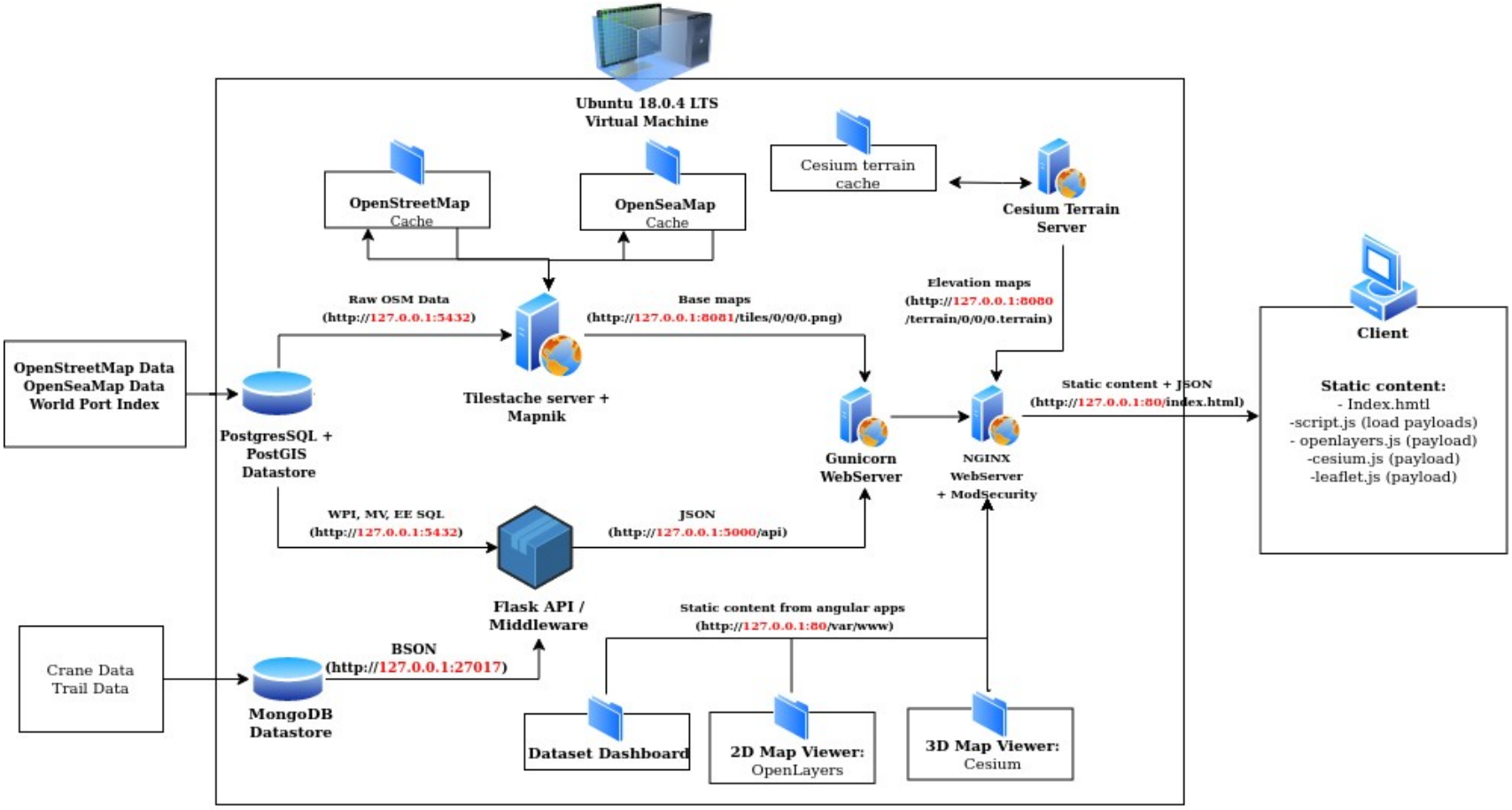
CesiumJS is an open source JavaScript library for creating world-class 3D globes and maps with the best possible performance, precision, visual quality, and ease of use.

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.

1.4 The GeoStack Software Architecture!

This **very important diagram** shows the positioning of each component in the GeoStack! Please, carefully do read the explanation about this diagram and the components shown in the following subsections! Also look at the architecture video clips on the GeoStack's YouTube Channel!



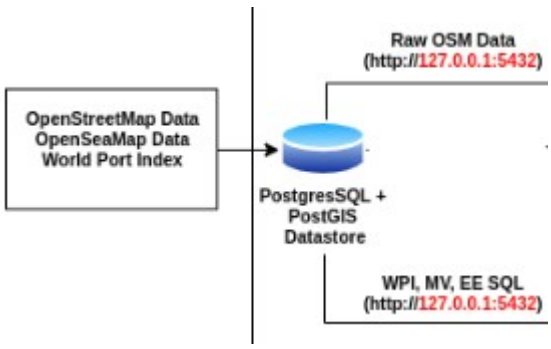
1.4.1 The PostgreSQL datastore

The SQL relational database called: “PostgreSQL” is going to contain the following data:

- **OpenStreetMap data**
  OpenStreetMap is an open source map provider. The RAW data shown on the map has to be stored in an SQL database. The dataset contains data related to streets, mountains etc. This data is stored as RAW data in the PostgreSQL datastore. This data combined with a style sheet is going to be used to generate OSM (OpenStreet and OpenSeaMap) Tiles (PNG images that make up a base map). This process is done by the tile server.

- **OpenSeaMap data**
  OpenSeaMap is also an open source map provider. The data shown on the map also has to be stored in an SQL database. The dataset contains data related to waterways, anchorages buoys etc. This dataset is especially useful for maritime related visualizations.

- **World Port Index data**
  The World Port Index dataset contains data related to sea ports around the world.
  The dataset contains data about the location, type of port, how big a port is and more.

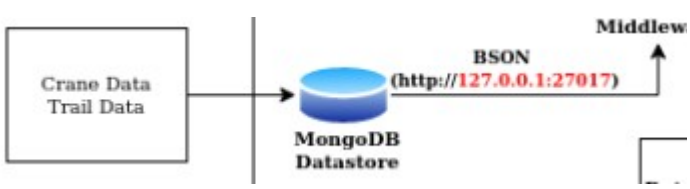


1.4.2 The MongoDB datastore

The NoSQL JSON document store called: “MongoDB” is going to contain the following data:

- **Crane datasets (Crane tracker datasets)**
  These datasets contain data related to multiple Cranes. They are a good representation of objects that contain coordinates and altitude data.

- **GPS-Route datasets (Trail datasets)**
  These datasets contain GPS data related to routes traveled throughout the Netherlands. The routes are a good representation of how land or water vehicles can be visualized.



1.4.3 The TileStache Tile server

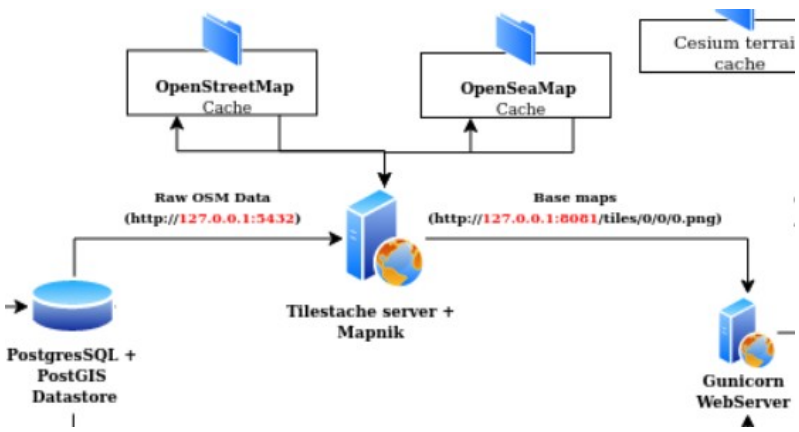
A tile server is the server that generates rendered images (tiles) from a database. There are multiple services online that provide a tile server. The problem with using online services is that the tile server will not work when you don't have a n active network-connection. This is also the reason why you are going to create your own tile server for generating base maps.

There are multiple ways in which you can create a tile server. One of these is using an Apache web server in combination with Mod_Tile and Mapnik. You are not going to use Apache since you want a lightweight tile server which serves your needs for generating, caching and serving base maps using the RAW OpenStreetMap data from our PostgreSQL datastore.

If you want to create a tile server using Apache and Mod_Tile, you should follow the guide on the following website: https://switch2osm.org/serving-tiles/building-a-tile-server-from-packages/.

You are going to create a tile server using the lightweight web server: "NGINX" in combination with TileStache (for caching the generated tiles) and Mapnik (for generating the tiles using the RAW OpenStreetMap data in our PostgreSQL datastore).

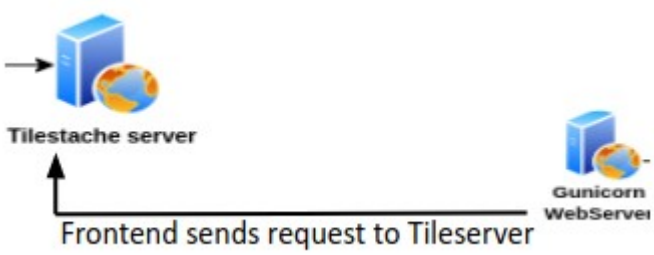
In the diagram below you find the structure and dataflow of the tile server in our GeoStack.



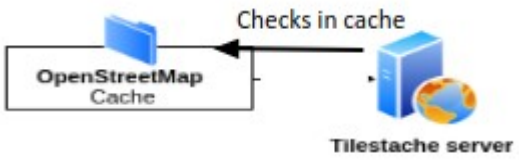
The way the TileStache tile server works is as follows:

1) The frontend sends a so called "{Z}{X}{Y}" request to our tile server. This request consists of the following values:
   - The name of the entry in the TileStache configuration file concerning the map that is requested. Such an entry contains information related to where map data is stored.
   - An Z value, which represents the Zoom Level of the map that is being requested.
   - An X value, which represents for the X-axis of the map that is being requested.
   - An Y value, which represents for the Y-axis of the map that is being requested.

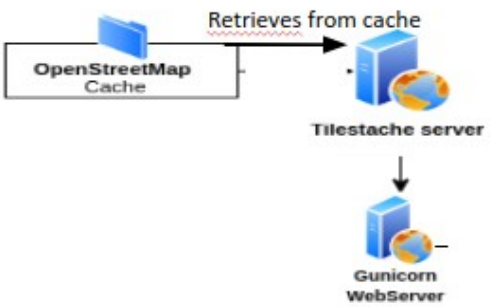
   These values are then used to only return the tiles than need to be displayed in the web application instead of the complete map.



2) The tile server then checks in the cache, belonging to the map that is being requested, to see if the requested tiles have already been generated before.



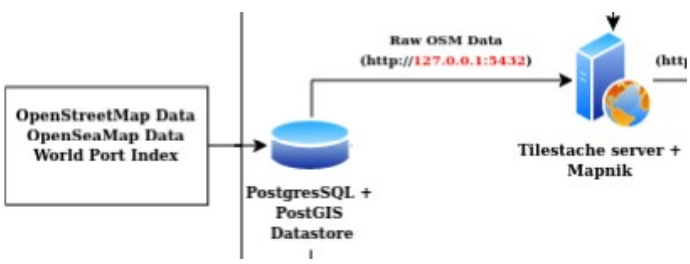
3) If the tiles have been generated before the tile server will extract these tiles from the cache and send them back to the front end (Our web applications).



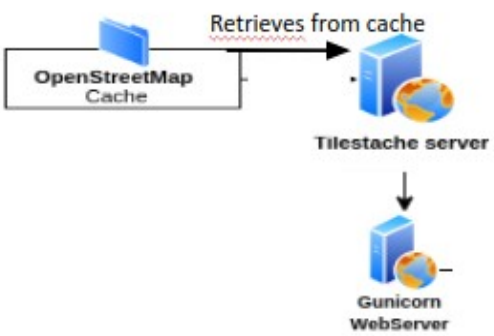
4) If the tiles have **NOT** been generated before the tile server will request the RAW OSM data from the PostgreSQL datastore after which Mapnik will generate the requested Tiles using the OpenStreetMap-Carto style sheet.

   The way Mapnik works is as follows:

   - The raw OSM data consists of nodes which contain a set of coordinates and a tag (e.g. a water tag).
   - The tag: “water”, is then linked to the styling of water in the style sheet. The style sheet will then for example give all the nodes containing the tag:”water” the color blue.



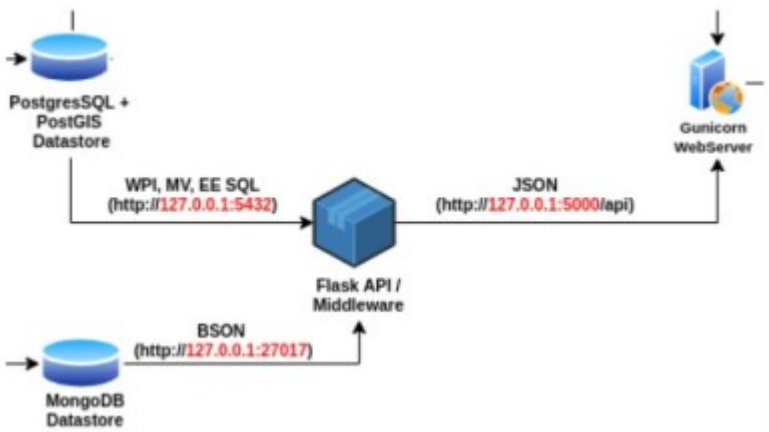
After the requested tiles have been generated they will be stored in the cache belonging to the requested map after which they are send to the frontend. When the tiles are requested a second time they don’t have to be generated anymore. This makes the loading of a map a lot faster.



1.4.4 The Flask API

Flask is a light-weight Python WSGI web application framework. It's also known as an micro framework since it's so small and doesn't need a lot of configuration to get it up and running. It is especially design to get started quick and easy, with the ability to scale up to a complex application.

The Flask API can be seen as the beating heart of the GeoStack. This is where all the "magic" happens. The Flask API contains the Source code related to the processing of the datasets from both the PostgreSQL datastore and the MongoDB datastore.



The Flask API contains functions that contain queries. A function is bound to a "route" which can be seen as an URL. An example of such a function can be found in the illustration below.

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 6) Create the function which retrieves a tracker by using it MongoID
@app.route('/api/trackers/<id>', methods=['GET'])
def get_one_tracker(id):

    # Assign the results of the query to a variable called: "query_results"
    query_result = crane_connection.db.tracker.find({"_id": ObjectId(id)})

    # Return the data obtained by the query in a valid JSON format
    return json.dumps(query_result, default=json_util.default)
```

The name of the function in the illustration above is: "get_one_tracker()"

- The function takes an ID as input parameter.
- The function is bound to the URL: /api/trackers/<id>.

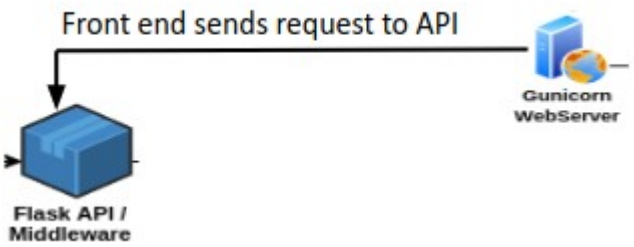
If a request is send to this URL the function "get_one_tracker()" is triggered and a query is executed on the MongoDB datastore.

- This query then returns the tracker data belonging to the tracker with the ID passed as the input parameter in the function.
- The Flask API (function) then sends that requested and obtained tracker data back to the frontend web application.

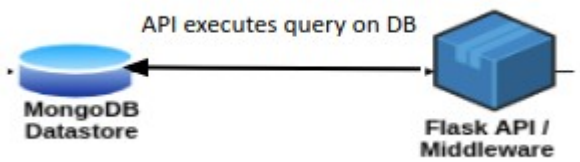
See the illustrations on the next page to see how things work.

The way the Flask API works is as follows:

1) The Flask API receives a request from the frontend. Such a request consists of an URL and values which are used to query on specific items in the datastores.



2) The Flask API then triggers the function, bound to the URL from the request after which it performs a query on the MongoDB or PostgreSQL datastore.



3) The datastore then returns the data to the Flask API which then performs the necessary transformations.

The API then sends the transformed data back to the frontend.

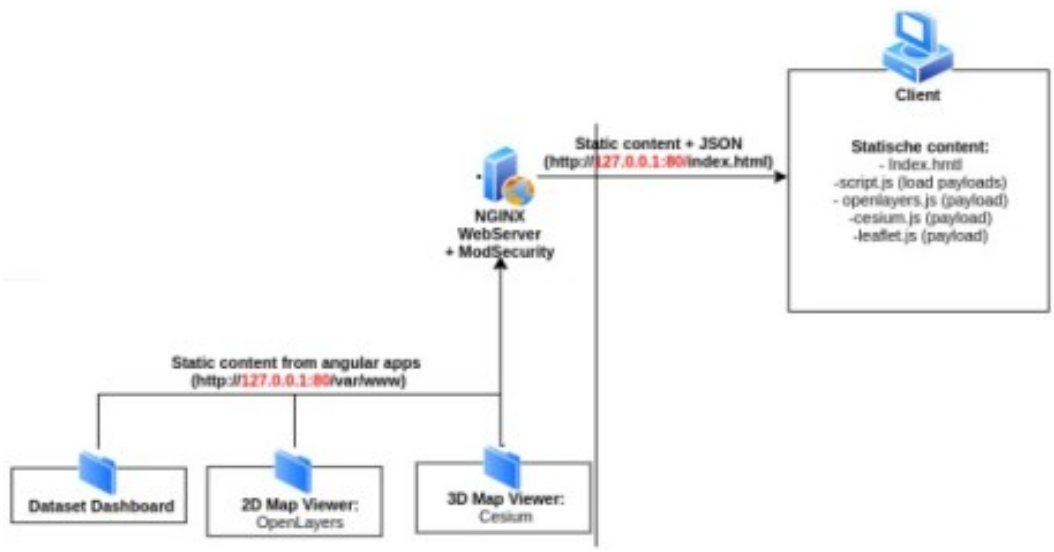
1.4.5 The NGINX Web server

NGINX, pronounced as "Engine X", is a fast and lightweight web server, that can be used to serve static files, but is often used as a reverse proxy. It has some very nice features like load balancing and rate limiting.

NGINX is commonly used if you need a lightweight web server with:

1) **A reverse proxy:**
   A reverse proxy is useful if you have multiple web services listening on various ports and you need to reroute requests internally. This allows you to run multiple domain names on port 80 which is the port the web server is running on.
2) **Load balancing:**
   Load balancing is a technique used to distribute workload across multiple machines. Load balancing could be division of processes, hard drives and other resources.
3) **Rate limiting:**
   Rate limiting is kind of the same as load balancing, but in the case of rate limiting the work load is divided among users of the web server.

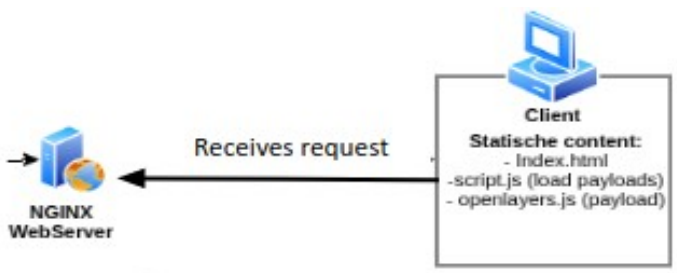
The three web applications of the GeoStack are hosted as static files on the NGINX web server.



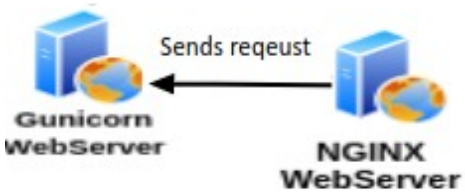
The NGINX web server works as follows:

1) The web server receives request from the client (User). A request could be the following:
   - The location of a Crane.
   - All GPS-Routes in the MongoDB datastore.
   - All Ports in the PostgreSQL datastore.

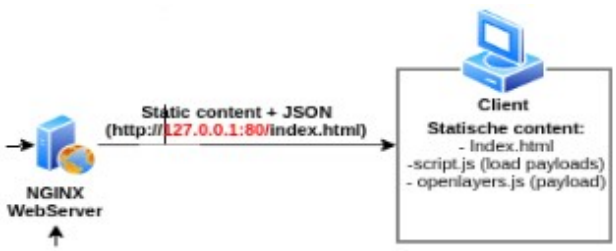
A map from our TileStache tile server.



2) The web server then relays the request to the corresponding component in the GeoStack. A corresponding component could be:
   - The TileStache tile server for generating a base map.
   - The MongoDB datastore for the location of a Crane.
   - The PostgreSQL datastore for all the Ports in the database.



3) The web server than receives the requested data from the corresponding component and sends it back to the client in combination with the static files from the web application which the user is currently using.

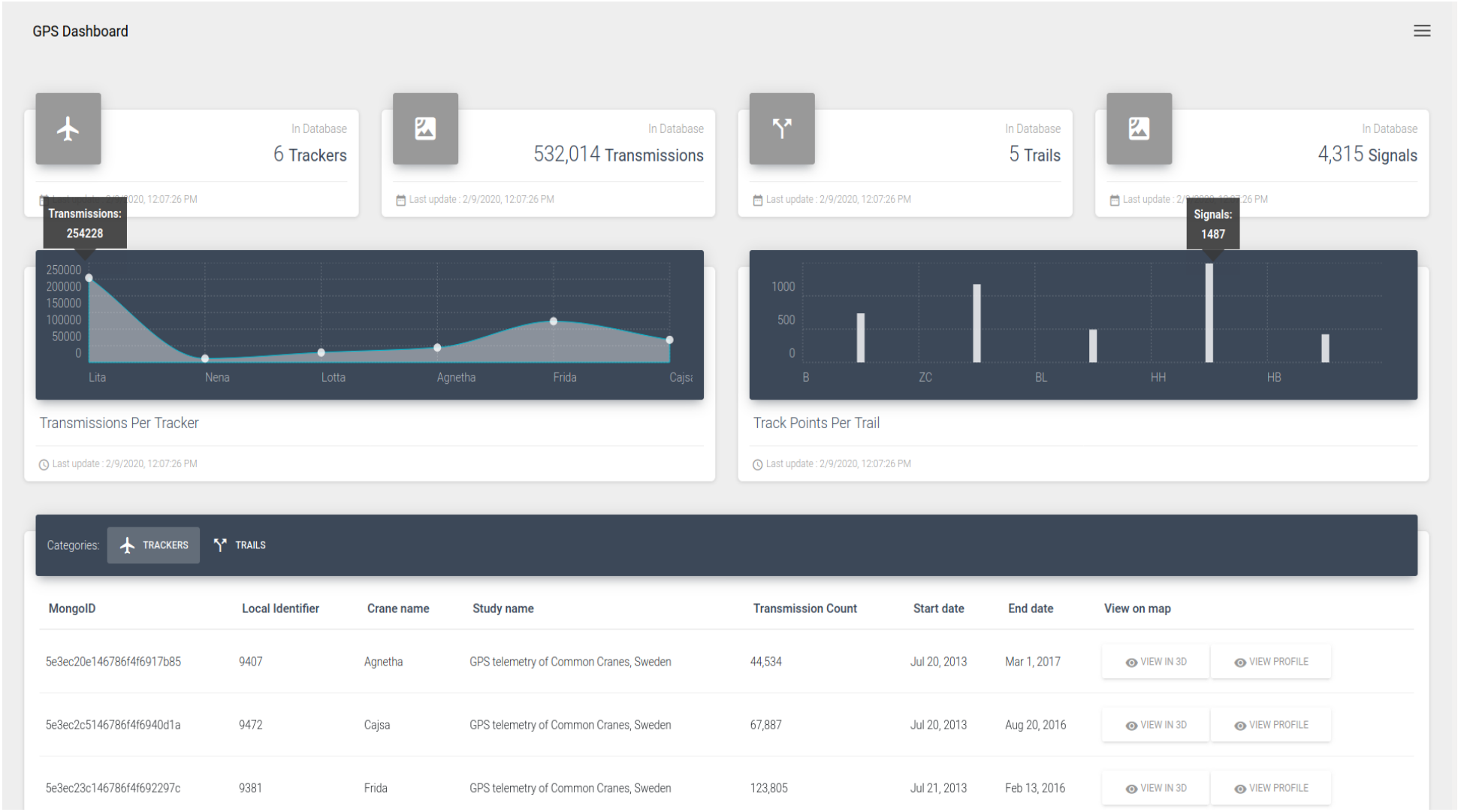


1.4.6 The Dataset Dashboard

The Dataset Dashboard is the first in a set of three applications created during the Beginner Course in Open Source Geospatial Programming for Data Scientists.
It shows all the datasets in our MongoDB datastore and contains interactive graphs and tables.
In the table below you can find an overview of the functionalities of the dataset dashboard.

#	Functionality	Description
1)	Overview of the Crane datasets	The dataset dashboard shows information related to the total amount of Crane trackers and transmissions in the MongoDB datastore.
2)	Overview of the GPS-Route datasets	The dataset dashboard shows information related to the total amount of GPS routes and signals in the MongoDB datastore.
3)	Overview of improvised AIS data	The dataset dashboard shows information related to the total amount of Ships and transmissions in the MongoDB datastore.
4)	Overview of World Port Index dataset	The dataset dashboard shows information related to the amount of ports in the database.
5)	Interactive graphs related to the data in the datastore.	The dataset dashboard contains interactive graphs related to the data in the MongoDB datastore.
6)	Interactive tables with more information related to the datasets	Each of the datasets have their own table which shows in detail information related to the dataset in questions.
7)	Generate dataset profiles using Pandas-Profiling	The dataset dashboard offers possibility to create a dataset profile for each of the datasets in the dataset dashboard. These profiles show information related to the dataset.

Below you find an image of the Dataset Dashboard created using the JavaScript framework Angular in combination with TypeScript.
Notice the list of datasets in the datastore below and above are the graphs with some statistics of the crane trackers and the trails datasets.



1.4.7 The 2D Map Viewer

The 2D Map Viewer is the second in a set of three applications created during the Beginner Course in Open Source Geospatial Programming for Data Scientists.
This application is used to visualize the datasets stored in the MongoDB and PostgreSQL datastore in 2D on a 2 Dimensional map using the JavaScript framework OpenLayers in combination with base maps of OpenStreetMap and OpenSeaMap.

All the data shown in the 2D Map Viewer is "live" which means that when you add data to your datastores the data will automatically be updated.

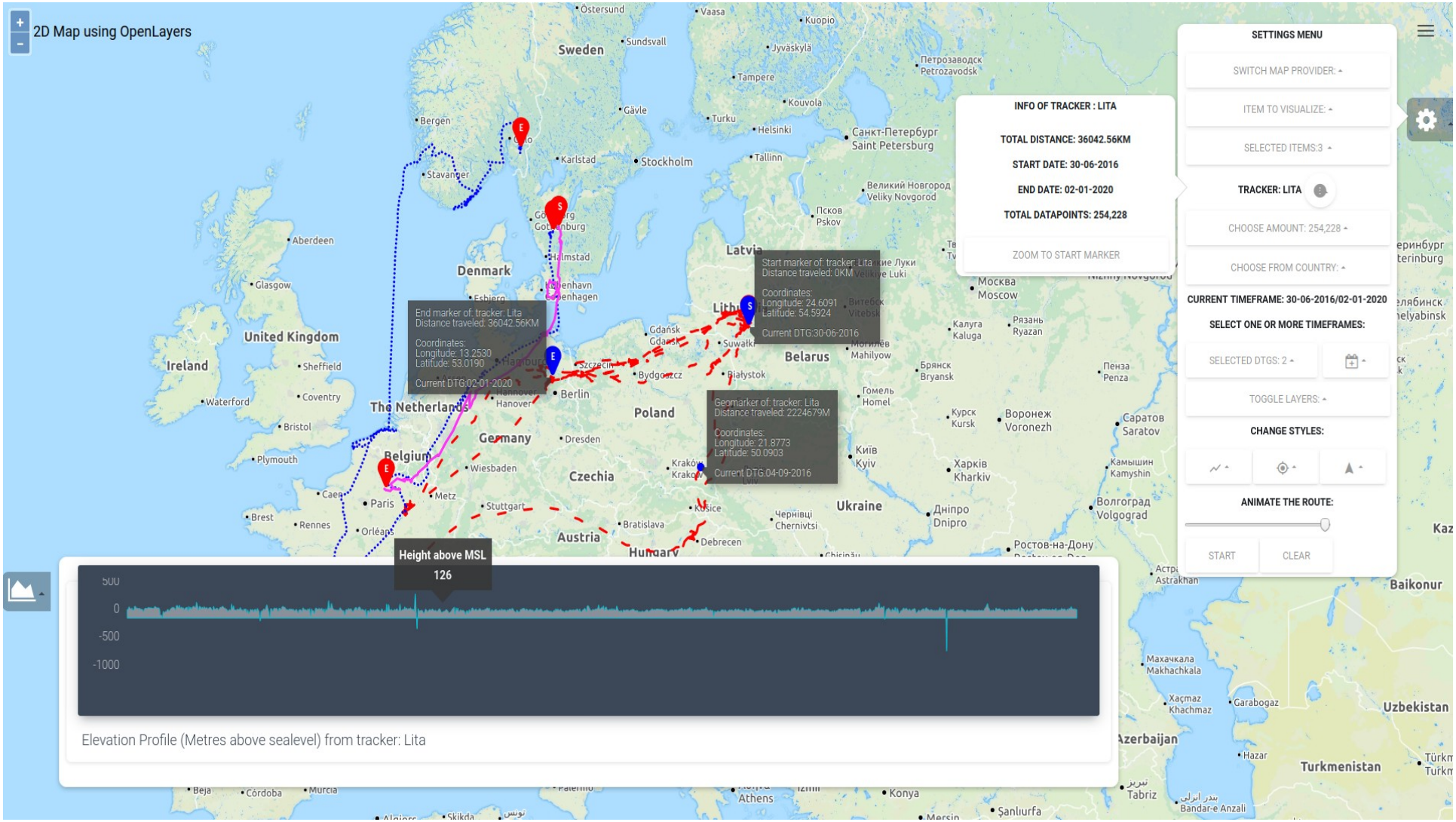
The functionality of this application can be found in the table below.

#	Functionality	Description
1)	Switch between WMS's (Web Map Server)	The application offers the possibility to switch between the Web Map Servers defined in the TileStache tile server.
2)	Select items	The application offers the possibility to select Crane data, GPS Route data and improvised AIS data.
3)	Select multiple items	The application offers the possibility to select multiple items and put them next to each other.
4)	Request detailed information related to an item.	The application offers the possibility to request detailed information related to an item.
5)	Select a given amount of datapoints	The application offers the possibility to choose the amount of datapoints you want to visualize.
6)	Select datapoints in a certain area	The application offers the possibility to select datapoints in a given polygon
7)	Select one or more timeframes	The application offers the possibility to select data in a given timeframe (DTG) there is also an option to choose multiple timeframes from one item.
8)	Change styling of features	The application offers the possibility to change the styling of features such as lines, arrows and markers.
9)	Animate, pause and continue routes.	The application offers the possibility to animate the selected routes. You can also pause and continue the route. The animation speed can also be changed.
10)	Live information.	While animating a route the information is updated. The information includes the current DTG, coordinates and distance travelled.
11)	Request information related to the start and end points of the route.	The application offers the possibility to request information related to the start and end points of the selected items. This information contains the coordinates, date and total distance of the visualized route.
12)	Elevation profile	The application offers the possibility to generate an elevation profile of the visualized routes. This profile contains an interactive graph in which you can move your cursor on the line to view the specific height at a certain point in the route.
13)	Toggle layers on and off.	The application offers the possibility to toggle layers such as an OpenSeaMap layer, World Port Index layer, lines, points and markers.

An illustration of the 2D Map Viewer application can be found on the next page.

A Video of the 2D Map Viewer is also available in the folder: "GeoStack-Demo-Videos". This folder can be found in the root folder of The GeoStack Project.

In the illustration below you can see multiple visualized GPS tracker routes of migrating cranes using the 2D Map Viewer web application created with the geospatial framework OpenLayers.



1.4.8 The 3D Map Viewer

The GeoStack also contains an 3D Map Viewer using the geospatial framework Cesium.

This is the third in a set of three applications created during the Beginner Course in Open Source Geospatial Programming for Data Scientists.

This application is used to visualize the datasets stored in the MongoDB datastore in 3D on a 3 Dimensional map using the JavaScript framework Cesium in combination with a base map of OpenStreetMap.

All the data shown in the 3D Map Viewer is “live” which means that when you add data to your datastores the data will automatically be updated.

This application is used to visualize data in 3D and displaying height maps. The functionalities of this application are similar to the ones of the 2D application.

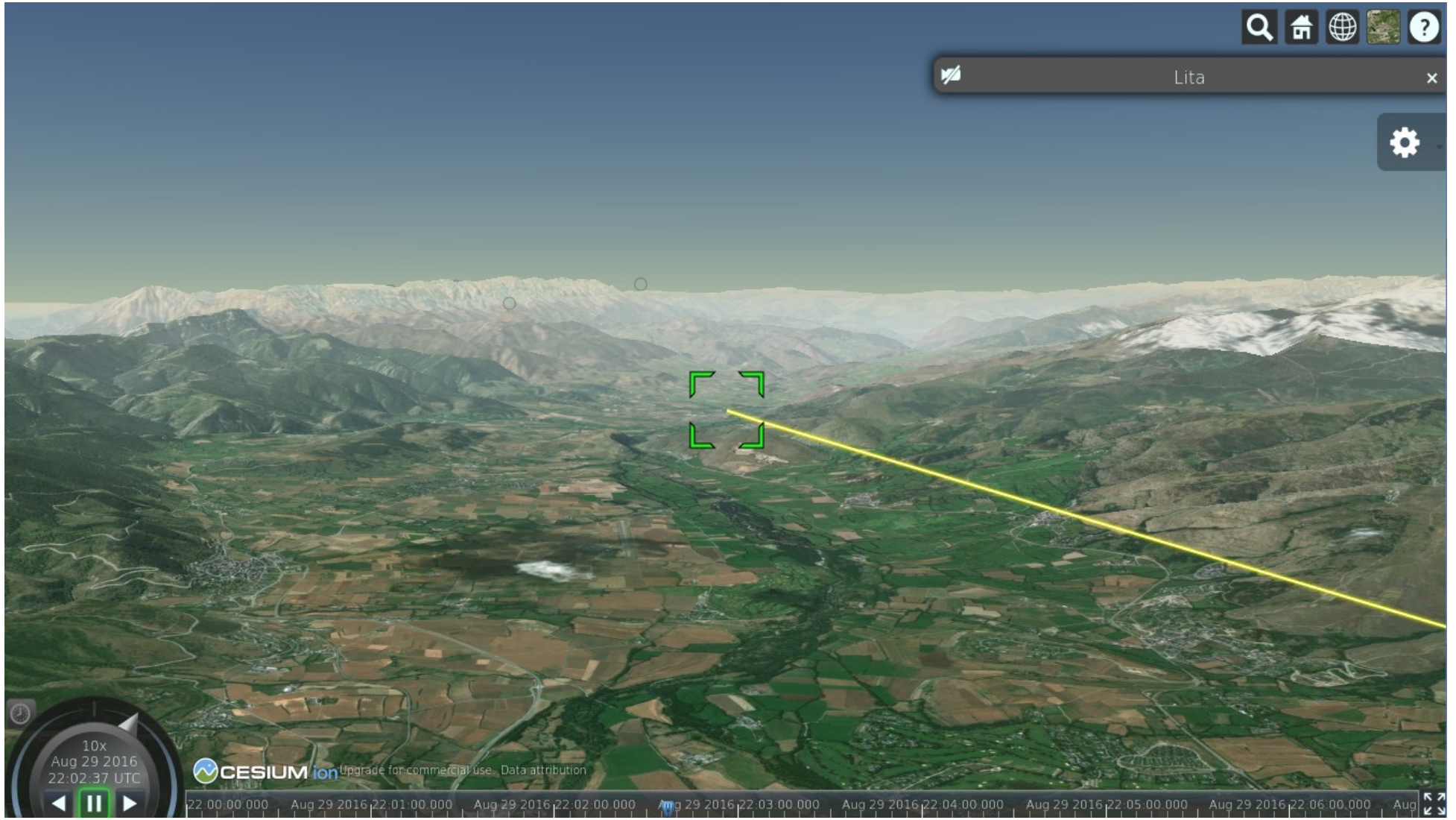
#	Functionaliteit	Omschrijving
1)	Switch between WMS’s (Web Map Server)	The application offers the possibility to switch between the Web Map Servers defined in the TileStache tile server.
2)	Select items	The application offers the possibility to select Crane data, GPS Route data and improvised AIS data.
3)	Select multiple items	The application offers the possibility to select multiple items and put them next to each other.
4)	Request detailed information related to an item.	The application offers the possibility to request detailed information related to an item.
5)	Select a given amount of datapoints	The application offers the possibility to choose the amount of datapoints you want to visualize.
6)	Select datapoints in a certain area	The application offers the possibility to select datapoints in a given polygon
7)	Select one or more timeframes	The application offers the possibility to select data in a given timeframe (DTG) there is also an option to choose multiple timeframes from one item.
8)	Displaying elevation maps.	The application offers the possibility to display elevation maps served by the Cesium Terrain Server.
9)	Animating a route.	The application offers the positbilty to visualize the visualized routes in 3D. The animation speed can be increased or decreased. The route can also be animate using a dateslider.

An illustration of the 3D Map Viewer application can be found on the next page.

A Video of the 3D Map Viewer is also available in the folder: “GeoStack-Demo-Videos”.

- This folder can be found in the root folder of The GeoStack Project.

In the illustration below you can see a visualized GPS tracker route from a birds-eye perspective of a crane flying over the Alps using the 3D Map viewer created with the geospatial framework Cesium.



2 Installing VirtualBox

To be able to create the GeoStack Course VM you will need to have VirtualBox installed on your system. How this is done is described in chapter 1 of the cookbook: “Creating the GeoStack Workshop VM”.

If you started with the workshop you should have VirtualBox installed on your system. If you skipped the workshop you should start with the workshop before continuing this cookbook!

It’s highly recommend you to start with the workshop first since the GeoStack workshop provides fast situational awareness relating to the complete **Beginner Course in Open Source Geospatial Programming for Data Scientists!**

The 1-day Workshop focuses on building a Client / Server infrastructure in a VM to get familiar with the server side software architecture of the entire software stack, called the GeoStack!

3 Installing the GeoStack

Before you can start with the installation process of the GeoStack software, tools and applications you first have to create a new Virtual Machine for the GeoStack Course and install the Ubuntu Linux Operating System on that VM. This will be done in the next section.

After you have created a new Ubuntu Virtual machine, you can decide which technique you want to use to install the GeoStack software and applications.

As mentioned in the introduction of this document, there are 2 techniques to install the complete GeoStack. These techniques are as follows:

- ➔ **1) Creating a new Virtual Machine and using the installation scripts to install all the software and applications by following Chapter 4:**

  If you choose to install the GeoStack using this technique you will have a fully functional geographical software stack in no time.

  In the GeoStack-Course Github repository you will find a folder called: “Building-the-VM-using-installation-scripts”. This folder contains everything you need to install the GeoStack software using the automated scripts.

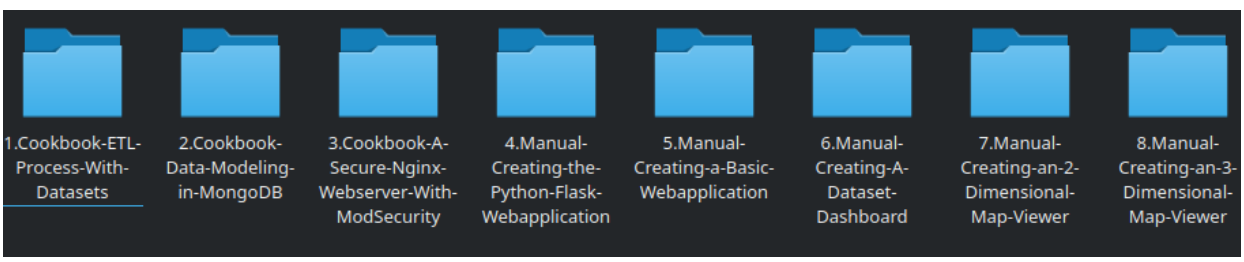
  If you choose to install the Geographical software stack using this technique you have to follow this chapter (3) and of course chapter 4.

- ➔ **2) Build the GeoStack from scratch using this document, the Jupyter Notebooks, the cookbooks, the programming manuals and the presentations by following Chapter 5:**

  If you choose to install the GeoStack using this technique you will learn a lot about the tools, software and data types discussed at the beginning of this cookbook.

  In the GeoStack-Course Github repository you will find a folder called: “Building-the-VM-from-scratch-using-the-manuals”. This folder contains all the cookbooks and manuals required to do the following and a lot more.

  The contents of this folder is shown in the illustration below.



  If you choose to install the Geographical software stack using this technique you can skip chapter 4 because you have to follow chapter 5!
  IMPORTANT NOTE: if you are a newbie do chapter 4 anyway to learn fast how things work (check out the installation scripts!) and to see what the end result should be!

In section 3.1 is described how you should create a new Virtual Machine and in section 3.2 references are shown to the GeoStack Workshop cookbook which describes the steps you have to perform to get the Virtual Machine ready.

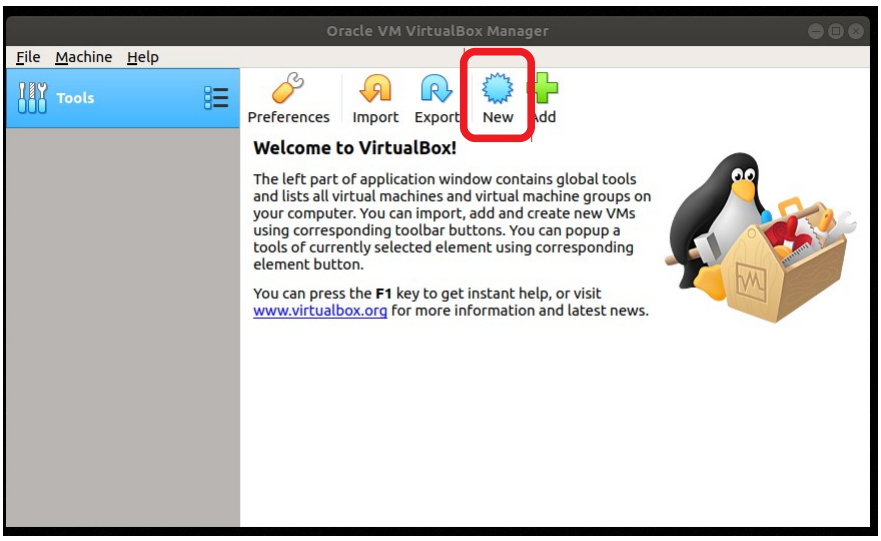
As mentioned above if you choose to install the GeoStack using the installation scripts you should read chapter 4 and If you choose to install the GeoStack manually, you should skip chapter 4 and read from there on.

3.1 Creating the Virtual Machine in VirtualBox

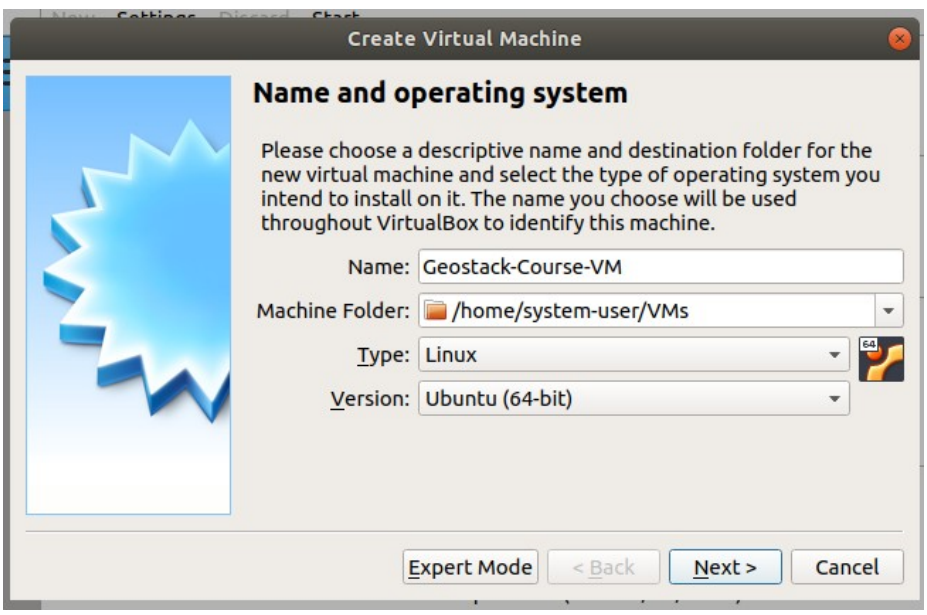
At this point you have VirtualBox on your system. Now you need to install the GeoStack Course Virtual Machine. To be able to install the GeoStack Course VM you first need to create a new Virtual Machine. After creating the Virtual Machine the next step is to install an Ubuntu ISO file.

Creating a Virtual Machine is done by performing the following steps:

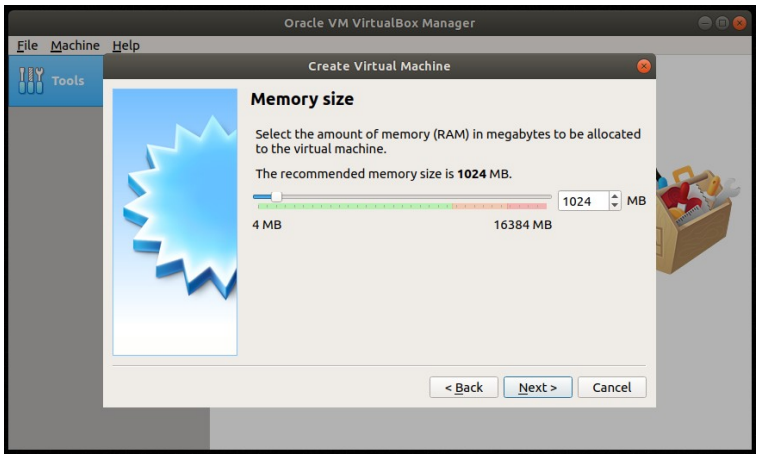
1) Click on the button: “new” in the VirtualBox start screen.



2) Give the Virtual Machine a suitable name in this case you give it the name: “Geostack-Course-VM”. In fact, you can choose any name, so GeoStack-Course-VM works fine too!

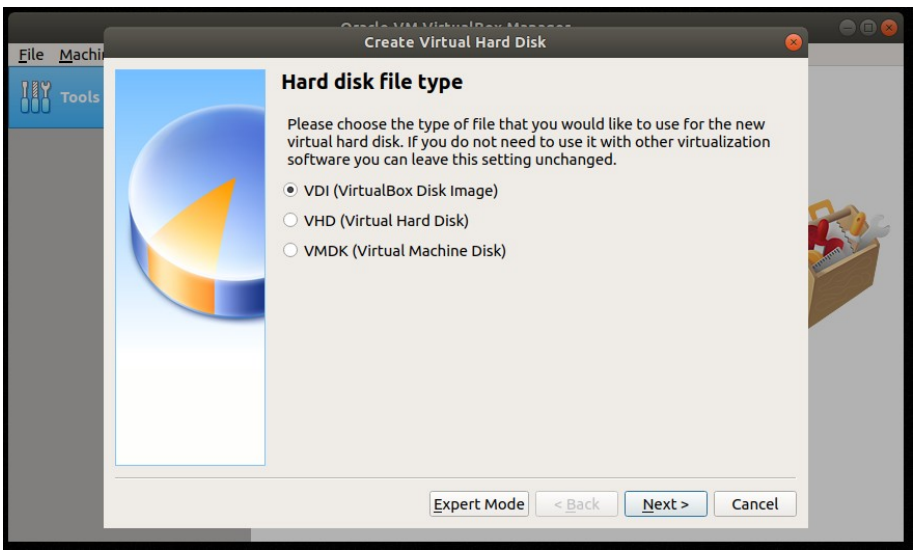


3) Assign the amount of RAM you want the machine to use. the minimal amount of RAM is 1 GB (1024MB) for a VM. If your PC has enough RAM the advice is to assign 3 GB (3072 MB) or 4 GB (4096 MB) RAM because Ubuntu will use up to 2 GB (2048 MB) already for itself.

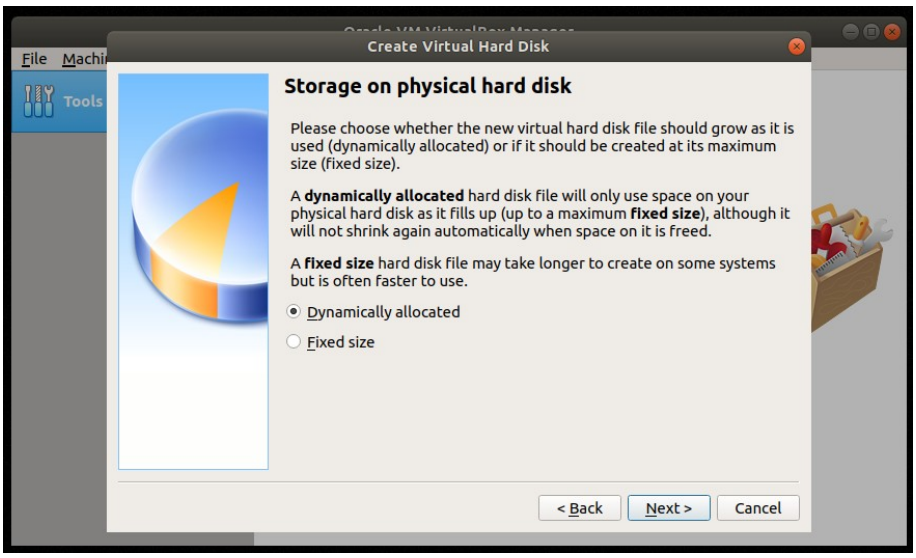


4) Select create a virtual hard disk.

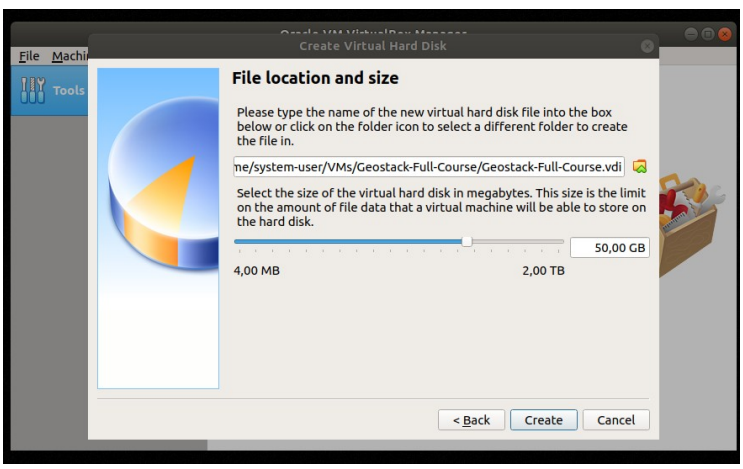
5) Set hard disk file type to VDI (Virtual Disk Image).



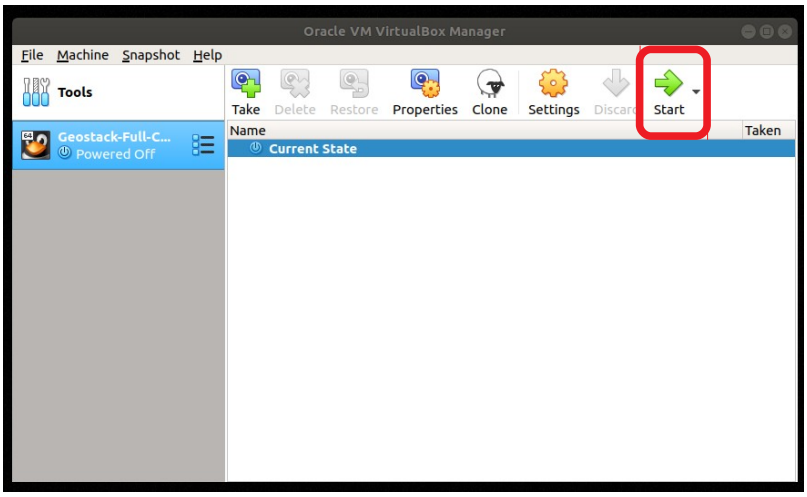
6) Select allocate disk space dynamically (Dynamically allocated). Tip: choose fixed disk size if performance matters to you.



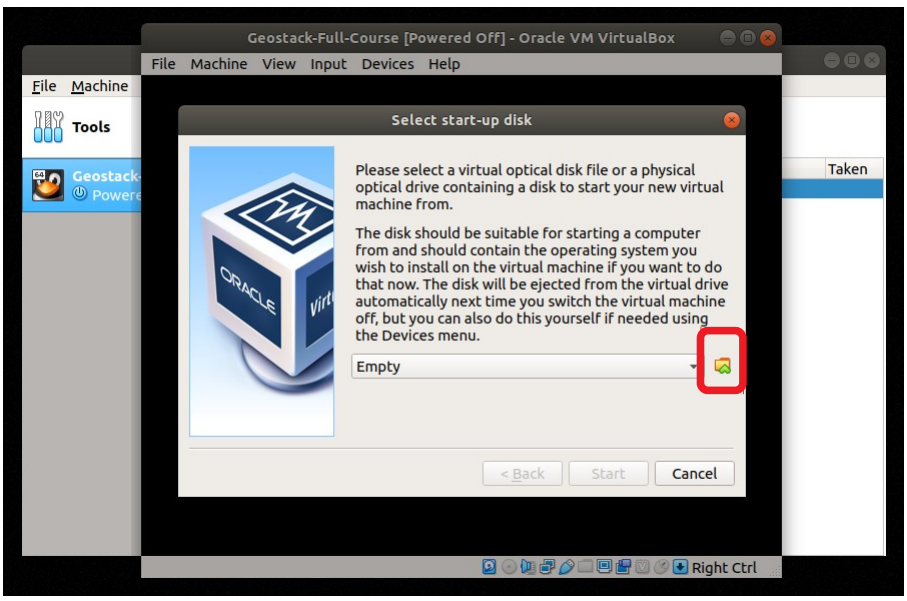
7) Select the location where you want to store the Virtual Disk Image file (.vdi file). In the perspective of managing your Virtual Machines, it's useful to create a centralized folder on your host system in which you create your VM's. The folder name could for example be called: " VirtualBox_Vms". This folder can also be used to store files such as ISO's and VBox extension packs. Then set the size of the hard disk to a minimum of 50 GB. The reason you are choosing to set the disk size to 50GB is because increasing the disk size of an VDI file afterwards can be a very tricky process.



8) Click on the green “Start” arrow while the Virtual Machine is highlighted (This is done by clicking on the VM) to start the VM.



9) Select a startup disk which is the Ubuntu ISO file that you have downloaded in the previous cookbook to create the Workshop VM.
Click on the yellow folder icon on the right to browse and select the ISO file.



3.2 Installing Ubuntu Linux in the Virtual Machine

As mentioned before, if you skipped the workshop it's highly recommend you to start with the workshop first! In the cookbook to create the Workshop VM you can find information related to setting up and finalizing a Virtual Machine installation.

The following chapters should be read:

1) **Section 2.2: Installing the Ubuntu ISO**

   To install an Ubuntu ISO in the new Virtual Machine, which was created in the previous section, you should read chapter 2.2 of the cookbook: 'Creating the GeoStack Workshop VM'.

2) **Section 2.3: Connecting the Virtual Machine to the network**

   To connect your newly create Ubuntu system to a network you should read section 2.3 from the cookbook: "Creating the GeoStack Workshop VM".

3) **Section 2.4: Creating a shared folder**

   To create a shared folder between your host system and the GeoStack Course Virtual Machine, you should read section 2.4 from the cookbook: "Creating the GeoStack Workshop VM". After reading section 2.4 you should come back to this cookbook to finalize the GeoStack Course VM Installation.

Now that you have a new Virtual Machine up and running you can choose whether you want to install the GeoStack automatically using the installation scripts or to create it manually from scratch using the cookbooks, programming manuals and Jupyter Notebooks.

3.3 Cloning the Github Repository of the GeoStack Course

Whether you choose to install the GeoStack Automatically or Manually, in both cases you first need to clone the GeoStack-Course Github repository to your newly created Virtual Machine.

Now clone the Github repository, which contains all the GeoStack Course files, by performing the following steps:

1) Open a terminal by pressing the Ctrl + Alt + T on your keyboard.
2) Install GIT by using the following command: `sudo apt install git`
3) Clone The GeoStack-Course Github repository by running the following command:

   `git clone https://github.com/The-GeoStack-Project/GeoStack-Course.git`

   The download process will take around 4 minutes depending on your network speed. After the download process is completed the terminal output should be similar to the one shown in the illustration below:

```
Receiving objects: 100% (2812/2812), 288.07 MiB | 7.30 MiB/s, done.
Resolving deltas: 100% (417/417), done.
Checking out files: 100% (3639/3639), done.
```

   If everything works accordingly you should end up with a folder called: "GeoStack-Course" in your home directory of your Virtual Machine.

3.4 Downloading the Course Datasets

As previously mentioned you will be using multiple types of datasets and data formats during the GeoStack Course. These types and formats are as follows:

1) **GPS Route (Trail) Datasets:**

   The Trail datasets are provided in the folder “GeoStack-Course/Course-Datasets/GPX”. These datasets are obtained from a GARMIN GPS-Device.

2) **Crane (Tracker) Datasets:**

   The Crane (Tracker) datasets are provided by an organization called MoveBank. The Movebank Data Repository contains published data sets of animal movement data in the Movebank format. This is distinct from the main Movebank tracking database, in which users control access and are responsible for their data quality, and where most data are stored. To be published in the Movebank Data Repository, a data set in Movebank undergoes an official review process and, when accepted, is granted a unique identifier (DOI) and license and is made publicly available.

3) **The World Port Index Dataset:**

   The World Port Index holds locations, physical characteristics, facilities, and services offered by major sea ports around the world. This dataset is created by the Maritime Security Office of the National Geospatial-Intelligence Agency (NGA) to document the locations and features of major ports around the world.

4) **OpenStreetMap Datasets:**

   OpenStreetMap (OSM) is aimed at creating and providing free geographic data such as street maps to anyone who wants them. The project was started because most maps you think of as free actually have legal or technical restrictions on their use, holding back people from using them in creative, productive or unexpected ways.

5) **DEM (Digital Elevation Model) files:**

   DEMs are files that contain either points (vector) or pixels (raster), with each point or pixel having an elevation value. They come in a variety of file formats, from . csv to . dem to . txt, and you can derive lots of other information - like contours or 3D surface models - from them.
   You are going to download the DEM files from a Dutch data provider called “PDOK” which is “Publieke Dienstverlening op de Kaart” in Dutch and Public Services On the Map in English. PDOK.nl is the website to publish geospatial datasets of the Dutch government as data services and files.

The Crane (Tracker) Datasets, World Port Index Dataset, OpenStreetMap Datasets and DEM Files have to be downloaded separately since The GeoStack Project does not own the right’s of these datasets.
However The GeoStack Project is the owner of the GPS Route datasets so they are already provided in the GeoStack-Course folder which you downloaded in the previous chapter.

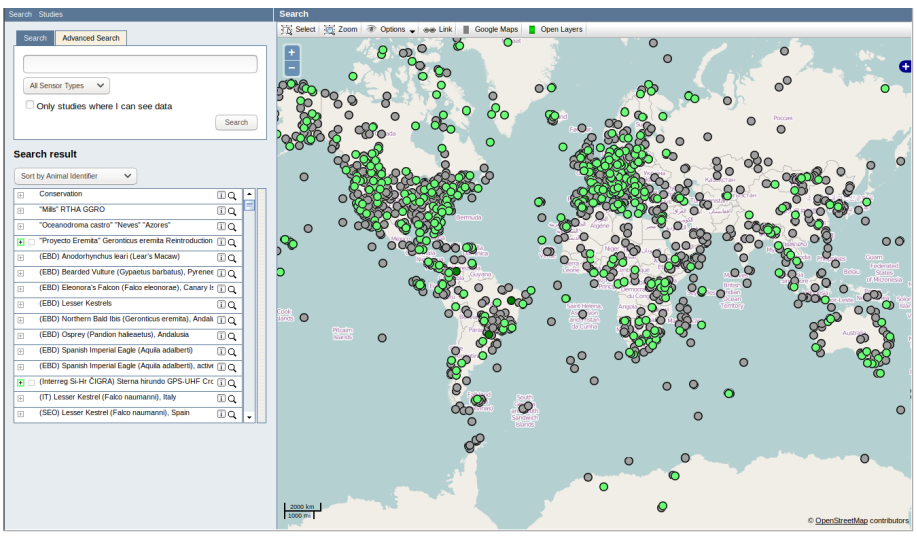
The GeoStack-Course folder contains a sub-folder called: “Course-Datasets” which contains a sub-folder called: “GPX” in which you can find the GPS Route datasets in the file format: “GPX”.

The CSV, JSON, SHP, OSM and DEM folders are empty at first since your are going to download these datasets yourself in this chapter.

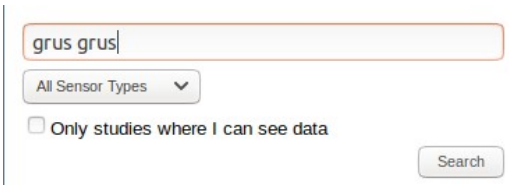
3.4.1 The Crane (Tracker) Datasets

The way you are going to download the Crane Tracker datasets is by navigating to the Movebank website and downloading the complete datasets. This procedure is described in the steps below:

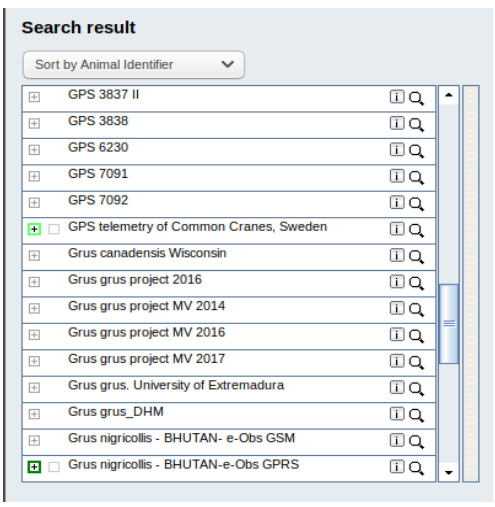
1) Navigate to the Movebank downloading page: https://www.movebank.org/panel_embedded_movebank_webapp



2) In the search field type: “grus grus” and click on Search as shown in the illustration below:

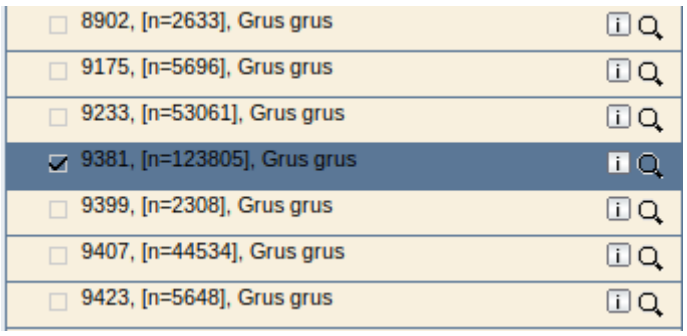


3) Scroll down till you find the entry : GPS telemetry of Common Cranes, Sweden (n=19) as shown in the illustration below:

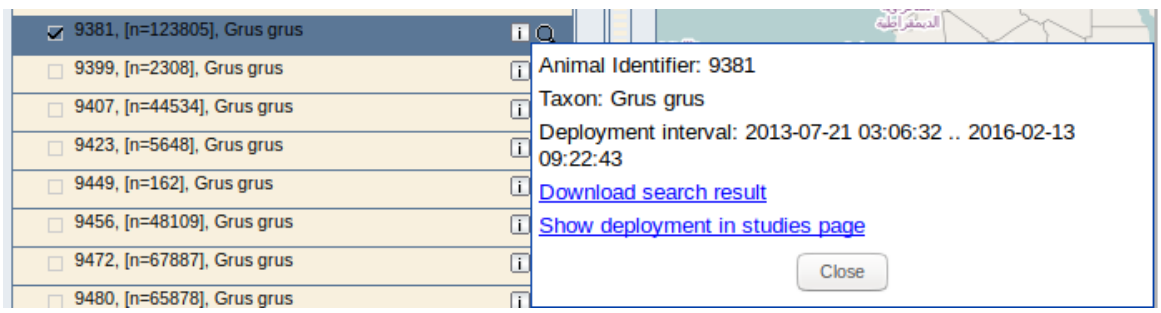


4) Click on the plus sign to expand the search results.

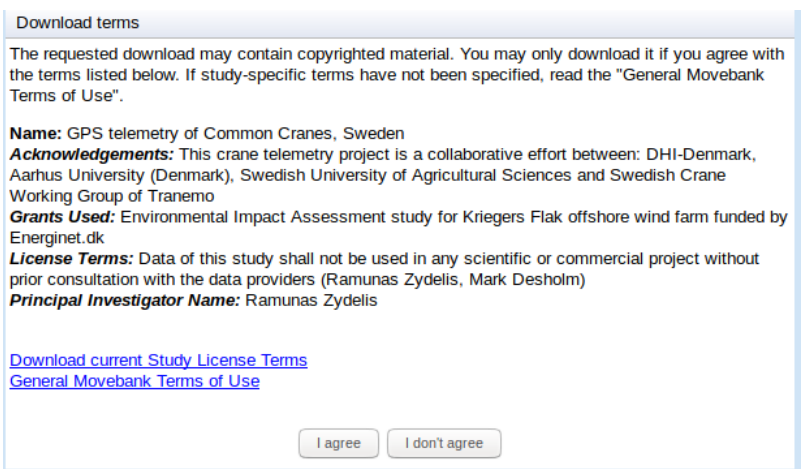
5) Select the box with the ID-number : 9381



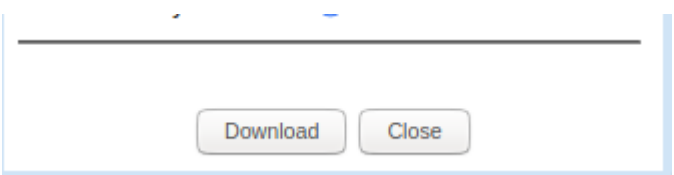
6) Click on the information icon and then click on Download Search result:



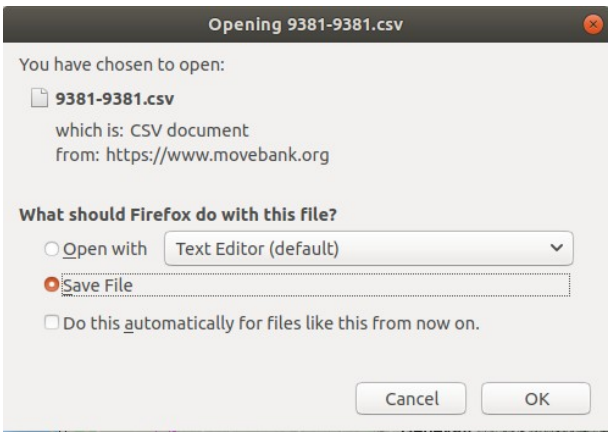
7) Click on I agree in the screen that pops up:



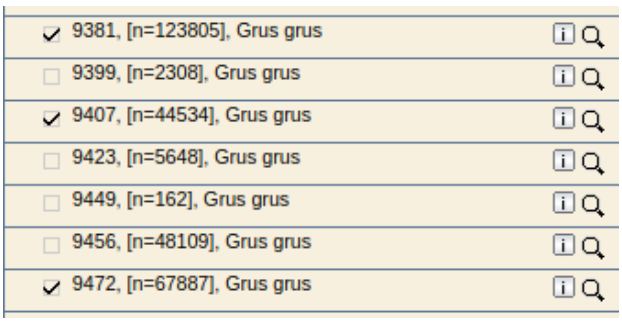
8) Click on Download in the screen that pops up:



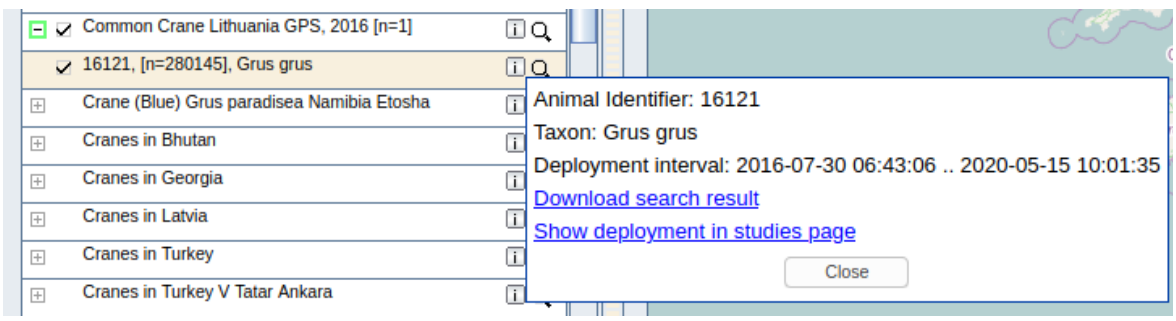
9) A Screen will pop up asking you to save the file. Select Save File and click on OK as shown in the illustration below:



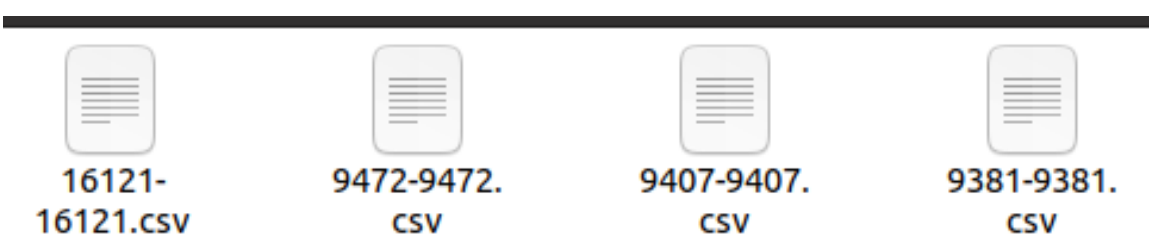
10) Now that you know how to download a Crane Tracker dataset you should repeat step 5, 6, 7, 8 and 9 on the Crane Tracker datasets with the id's 9407 and 9472 as shown in the illustration below:



11) Next you have to do the same for a Lithuanian Crane dataset. This is done by scrolling untill you find the search result: "Common Crane 2016 Lithuania 2016" as shown in the illustration below. Download the dataset with Tracker ID: "16121" by clicking on the information button next to the entry and selecting: "Download search result":



12) After you have downloaded the 4 datasets you will end up with 4 CSV files in your system's downloads folder as shown in the illustration below:



13) The last thing you need to do is renaming the files to give the datasets a more human readable descriptive name. This works in the same way as in the workshop with the commands below.

- Notice the color code for rings of the Lithuanian crane 'Lita' is unknown and therefore is not part of the file name.
- For the Swedish cranes the color codes can be found by clicking the blue information icon 'i' after the selected crane in the list, then in the pop-up window click the weblink 'Show deployment in studies page' and then in the left menu click the link 'Deployments' under the menu item 'Animal – Tracker_ID_number' to display the 'Animal Details' overview on the right side. The field 'Comments' holds the color code.

To rename and copy all the datasets to the correct place you should open a terminal (with Ctrl + Alt + T) and run the following commands (remember, the backslash is to split long commands over multiple lines in the terminal!):

For the Swedish Crane with the Tracker ID: "9381":

```
cp ~/Downloads/9381-9381.csv \
~/GeoStack-Course/Course-Datasets/CSV/20181003_Dataset_SV_GPS_TrackerID_9381_\
ColorCode_RRW-BuGBk_Crane_Frida.csv
```

For the Swedish Crane with the Tracker ID: "9407":

```
cp ~/Downloads/9407-9407.csv \
~/GeoStack-Course/Course-Datasets/CSV/20181003_Dataset_SV_TrackerID_9407_\
ColorCode_RRW-BuGY_Crane_Agnetha.csv
```

For the Swedish Crane with the Tracker ID: "9472":

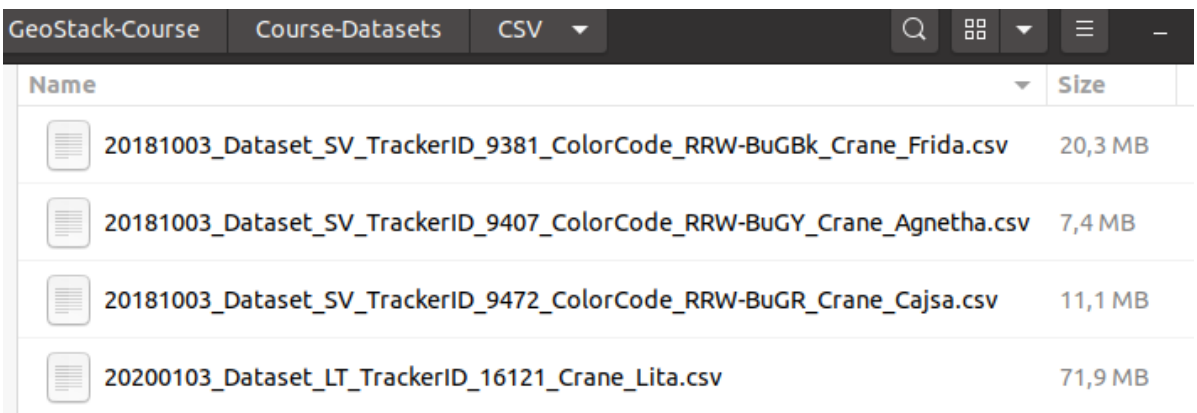
```
cp ~/Downloads/9472-9472.csv \
~/GeoStack-Course/Course-Datasets/CSV/20181003_Dataset_SV_TrackerID_9472_\
ColorCode_RRW-BuGR_Crane_Cajsa.csv
```

For the Lithuanian Crane with the Tracker ID: "16121":

```
cp ~/Downloads/16121-16121.csv \
~/GeoStack-Course/Course-Datasets/CSV/20200103_Dataset_LT_TrackerID_16121_\
Crane_Lita.csv
```

If everything is done accordingly you should end up with the following datasets in the folder "/GeoStack-Course/Course-Datasets/CSV/".

NOTE: Before continuing please check whether the names of the datasets match the ones in the illustration below.



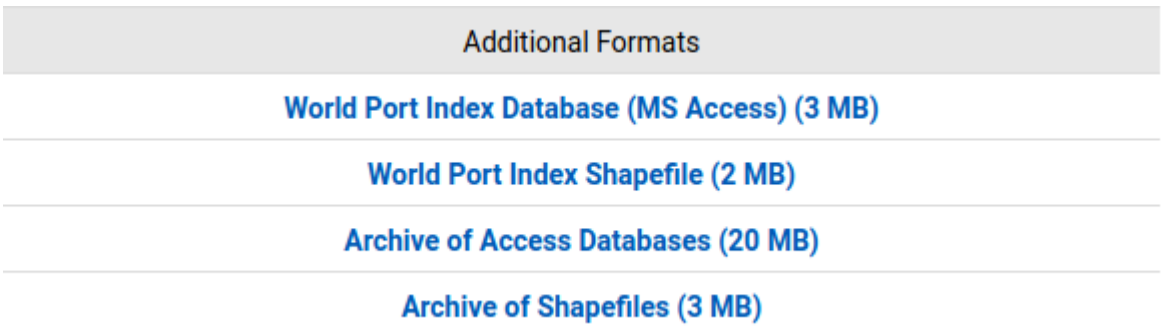
Note: the file size and number of datapoints in the file for crane Lita can still grow because the GPS tracker is still working and therefore new datapoints are still added to the dataset!

3.4.2 The World Port Index Dataset

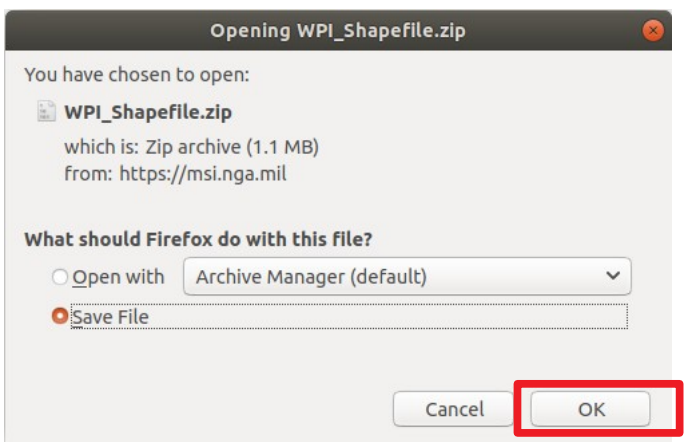
Downloading the World Port Index dataset is done by performing the following steps:

1) Navigate to the website from where you can download the dataset by clicking on the following URL: https://msi.nga.mil/Publications/WPI

2) Scroll down till you see the table as shown in the illustration below and then click on World Port Index Shapefile entry as shown in the illustration below.



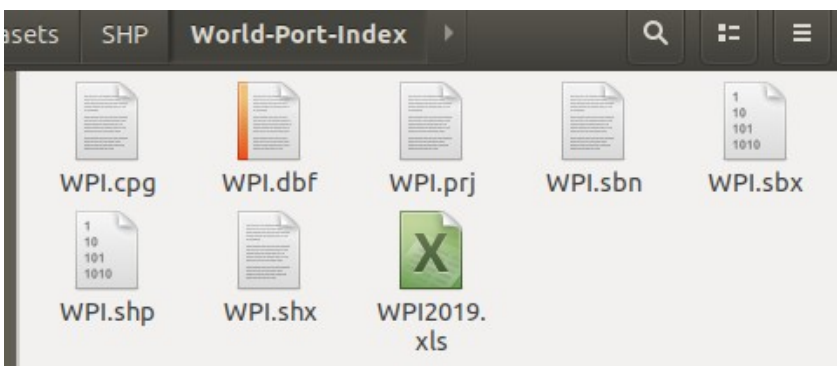
3) A window will pop up asking you to open or save the file. Select "Save file" and click on the OK button as shown in the illustration below.



4) Open a terminal (Ctrl + Alt + T on your keyboard) and unzip the downloaded file to the Course-Datasets folder by running the following command in the terminal:

```
unzip ~/Downloads/WPI_Shapefile.zip -d \
~/GeoStack-Course/Course-Datasets/SHP/World-Port-Index
```

5) That's it! Now you have the World Port Index dataset in the Course-Datasets Shapefiles folder located at "~/GeoStack-Course/Course-Datasets/SHP/World-Port-Index" as shown in the illustration below.



3.4.3 The OpenStreetMap data

Downloading the OpenStreetMap data of the south-eastern Dutch province “Limburg” is done by performing the following steps:

1) Navigate to the website from where you can download the dataset by clicking on the following URL: http://download.geofabrik.de/

2) Select Europe in the table shown on the web page as shown in the illustration below:

Sub Region	Quick Links			
	.osm.pbf		.shp.zip	.osm.bz2
Africa	[.osm.pbf]	(3.7 GB)	✖	[.osm.bz2]
Antarctica	[.osm.pbf]	(29.0 MB)	[.shp.zip]	[.osm.bz2]
Asia	[.osm.pbf]	(8.0 GB)	✖	[.osm.bz2]
Australia and Oceania	[.osm.pbf]	(775 MB)	✖	[.osm.bz2]
Central America	[.osm.pbf]	(378 MB)	✖	[.osm.bz2]
Europe	[.osm.pbf]	(21.4 GB)	✖	[.osm.bz2]
North America	[.osm.pbf]	(9.2 GB)	✖	[.osm.bz2]
South America	[.osm.pbf]	(1.9 GB)	✖	[.osm.bz2]

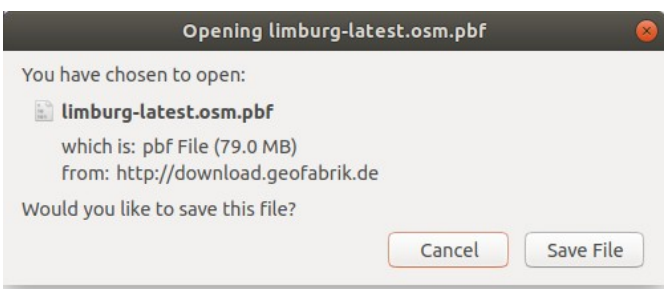
3) Scroll down till you find The Netherlands and click on it as shown in the illustration below:

Montenegro	[.osm.pbf]	(22.9 MB)	[.shp.zip]	[.osm.bz2]
Netherlands	[.osm.pbf]	(1.0 GB)	[.shp.zip]	[.osm.bz2]
Norway	[.osm.pbf]	(803 MB)	[.shp.zip]	[.osm.bz2]

4) Find Limburg and click on the .osm.pbf button as shown in the illustration below:

Groningen	[.osm.pbf]	(44.3 MB)	[.shp.zip]	[.osm.bz2]
Limburg	[.osm.pbf]	(79 MB)	[.shp.zip]	[.osm.bz2]
Noord-Brabant	[.osm.pbf]	(164 MB)	[.shp.zip]	[.osm.bz2]

5) A window will pop up asking you to save the file. Click on Save File as shown in the illustration below (Note: sometimes this pop-up will not be shown and the file is downloaded instantly) :



6) Open a terminal (Ctrl + Alt + T on your keyboard) and copy the downloaded file to the Course-Datasets folder by running the following command in the terminal:

```
cp ~/Downloads/limburg-latest.osm.pbf ~/GeoStack-Course/Course-Datasets/OSM
```

3.4.4 Cesium Elevation Map data (DEM Files)

Downloading the Elevation Map data of a section of the Dutch province: “Limburg” is done by performing the following steps:

1) Navigate to the website from where you can download the dataset by clicking on the following URL: https://downloads.pdok.nl/ahn3-downloadpage/

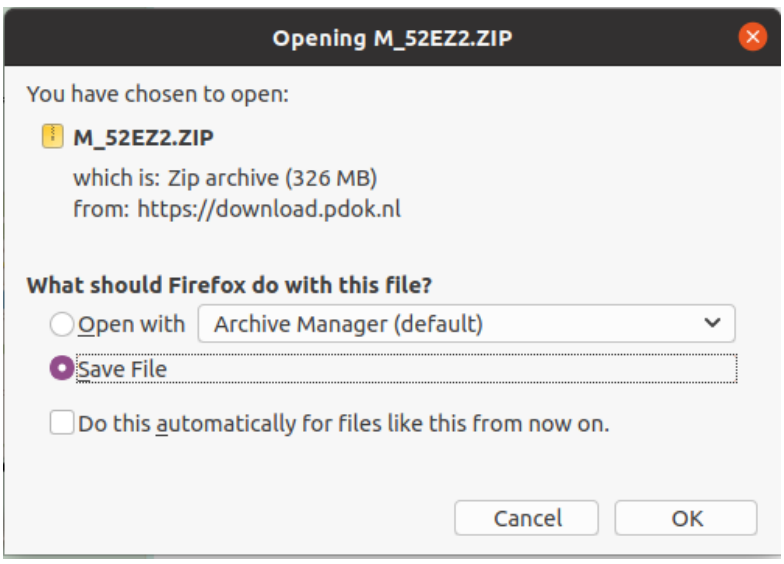
2) Zoom in on the location enclosed in red as shown in the illustration below:

3) Click on the box noted with 52ez2 as shown in the illustration below :

4) In the table below the map you will now get the option to download the data. Click on the download button encircled in red in the illustration below:

KAARTBLAD: 52EZ2		
INHOUD	FORMAAT	LINK
0,5 meter raster dsm	GeoTIFF (gezipt)	Download
0,5 meter raster dtm	GeoTIFF (gezipt)	Download
5 meter raster dsm	GeoTIFF (gezipt)	Download
5 meter raster dtm	GeoTIFF (gezipt)	Download
Puntenwolk	LAZ	Download

5) A pop up dialog will open asking you to open or save the file. Select Save File as shown in the illustration below (Notice the file of this small area is already about 350MB in size! The download process takes about 1 to 2 minutes depending on your network speed):



6) The last thing you need to do is to copy the downloaded dataset to the correct place in our system. This is done by opening a terminal (Ctrl + Alt + T on your keyboard) and running the following command:

```
cp ~/Downloads/M_52EZ2.ZIP ~/GeoStack-Course/Course-Datasets/DEM/
```

That's it! Now you have the DEM dataset in the correct place in your system located at ~/GeoStack-Course/Course-Datasets/DEM/ as shown in the illustration below.

Don't worry about unzipping the file, this will be done later!

Now that you have all the datasets in place you can start creating the Open-Source Geographical Software stack. As mentioned before; you can choose whether you want to install the GeoStack using the installation scripts (Chapter 4) or to install it using the cookbooks, programming manuals and notebooks (Chapter 5).

4 Installing the GeoStack AUTOMATICALLY

So you chose to install the GeoStack automatically? That's a good choice! If everything goes accordingly you will have a fully working Geographical software stack in less than 3 hours.

4.1 Folder location of the GeoStack installation scripts

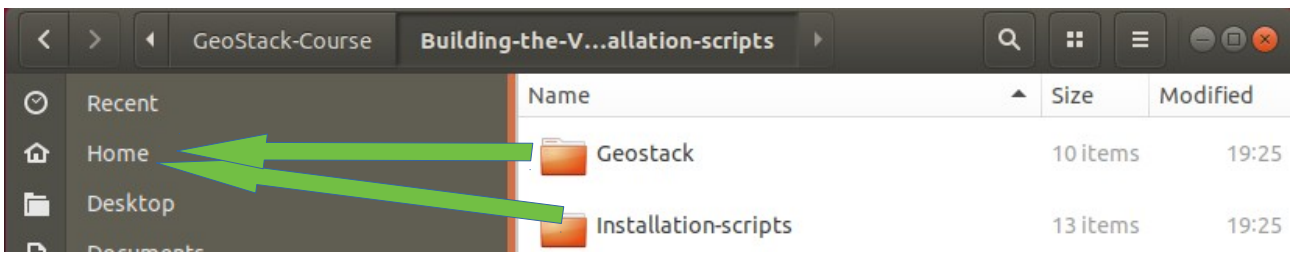
In the GeoStack-Course folder, which was downloaded in the previous section, you will find a folder called:"Building-the-VM-using-the-installation-scripts". This folder contains the following sub-folders:

- ➔ **a folder called: "Geostack":**
  In this folder you will find all the source code of the GeoStack components and web applications.

--> IMPORTANT NOTICE: the folder name is 'Geostack' and NOT the project name 'GeoStack'!

- ➔ **a folder called: "Installation-scripts":**
  In this folder you will find all the installation scripts to install the GeoStack components and web applications.

Let's copy these two folders to our home directory by entering the GeoStack-Course directory and then copying the 2 folders to our home directory as shown in the illustration below.



4.2 How to run the Installation Scripts!

Now that you have the required files in your Virtual Machine you can start the installation process but first we need to clear some things up.

IMPORTANT NOTE: installing the software automatically, using the installation scripts, does not mean you don't have to pay attention to what is happening!

Once in a while you are asked for some input such as changing default passwords, accepting updates and adding package sources to our system's sources-list.
The following should be done in those cases:

1. **Password prompt enter the password: geostack**

```
[sudo] password for geostack:
```

2. **Accepting updates enter: y**

```
After this operation, 97,2 MB disk space will be freed.
Do you want to continue? [Y/n]
```

3. **Adding packages to source list: press ENTER**

```
Report non-packaging Atom bugs here: https://github.com/atom/atom/issues
 More info: https://launchpad.net/~webupd8team/+archive/ubuntu/atom
Press [ENTER] to continue or Ctrl-c to cancel adding it.
```

4.3 Running the GeoStack Installation Scripts

The folder: “Installation-scripts” contains 12 scripts. The backend, middleware, frontend and virtualization software products each have their own installation script.

Now you can start with the execution of the installation scripts:

1) Close all windows and open a new terminal (by pressing the keys Ctrl + Alt + T on your keyboard).
   - Note: you can Copy and Paste the commands mentioned below to increase your workflow speed, but make sure that you understand what is happening.

2) Run the first script by entering the command: `bash ~/Installation-scripts/1-pre-reboot.sh`

   This script will do the following:

   ➔ Update the system.
   ➔ Install open-vm-tools.
   ➔ Install Bleachbit.
   ➔ Install net-tools.
   ➔ Install LibreOffice Writer.
   ➔ Install Python3 and Python3-pip.
   ➔ Install Curl.
   ➔ Install GIT.
   ➔ Install Docker.
   ➔ Install Docker-compose.
   ➔ Install NodeJS.
   ➔ Install Atom.
   ➔ Add new sidebar shortcuts for the newly installed software and tools.

   After the completion of this script the system will reboot for the updates to take effect. This script takes around 10 minutes to complete depending on the speed of you network-connection. After the script is complete 2GB of additional disk space is used.

3) After the reboot open a new terminal (with Ctrl + Alt + T) and enter the command: `bash ~/Installation-scripts/2-post-reboot.sh`

   This script will do the following:

   ➔ Copy the desktop shortcuts and to the correct place.
   ➔ Create file links on the desktop so you can easily access the component folders.
   ➔ Clean the unused files.

   This script will make sure all the folders and shortcuts are placed in the correct place and takes about 10 seconds to complete. After the script is complete no additional disk space is used.

4) Now when that script is done and no errors were encountered you can move on to the script for installing the data-analyses software required for performing the data analyses. This is done by running the following command:
`bash ~/Installation-scripts/3-data-analyses-software.sh`

   This script will do the following:

   ➔ Install Jupyter Lab.
   ➔ Install Pandas and GeoPandas.
   ➔ Install Pandas Profiling.
   ➔ Install Cartopy and MathPlotLib.
   ➔ Install GPXPy.

   This scripts takes about 10 minutes to complete depending on the speed of your network-connection. After the script is complete 2GB of additional disk space is used.

5) Now when that script is done and no errors were encountered you can move on to the next script for installing the backend software required to store all the data. This is done by running the command: `bash ~/Installation-scripts/4-backend-software.sh`

   This script will do the following:

   ➔ Install PostgreSQL.
   ➔ Install PostGIS.
   ➔ Install PGAdmin4.
   ➔ Install PsycoPG2.
   ➔ Install MongoDB.
   ➔ Install MongoCompass.
   ➔ Install MongoEngine.
   ➔ Add the new sidebar shortcuts.

   This scripts takes about 10 minutes to complete depending on the speed of your network-connection. After the script is complete 1GB of additional disk space is used.

6) Now when that script is done and no errors were encountered you can move on to the next script for importing all the datasets in the correct datastores. This is done by running the command: `bash ~/Installation-scripts/5-dataset-import.sh`

   This script will do the following:

   ➔ Import, model and index the Crane(Tracker) datasets in MongoDB.
   ➔ Run the dataset-convert.py script to transform and copy the datasets.
   ➔ Import, model and index the GPS-Route (Trail) datasets in MongoDB.
   ➔ Create a World Port Index database and import the WPI (World Port Index) dataset in the PostgreSQL datastore.

   The above is done by calling the import scripts located in the folder: “~/Geostack/import-utilities”.

These scripts will transform and import the Crane (Trail) datasets, GPS Route (Trail) datasets and the World Port Index dataset. The script takes about 15 minutes to complete and 3GB of additional disk space is used.

NOTE: During the import process the VM can become slow. Don't worry about this! The reason for this is because the MongoDB import scripts use a feature called: "bulk import". This feature adds all the data at once. This makes the import process a lot faster but can take up a lot of RAM which results in a slower VM (temporarily).

7) Now when that script is done and no errors were encountered you can move on to the next script for installing the Flask API software. This is done by running the command: `bash ~/Installation-scripts/6-middleware-software-Flask.sh`

   This script will do the following:

   ➔ Install Python-Flask.
   ➔ Install Flask-Pymongo.
   ➔ Install Gunicorn3.
   ➔ Install BeautifulSoup 4.

   This scripts takes about 3 minutes to complete depending on the speed of your network-connection. After the script is complete 300MB of additional disk space is used.

8) Now when that script is done and no errors were encountered you can move on to the next script for installing the tile server software. This is done by running the command: `bash ~/Installation-scripts/7-middleware-software-Tileserver.sh`

   This script will do the following:

   ➔ Install Gunicorn.
   ➔ Install TileStache and Pillow.
   ➔ Install Mapnik.
   ➔ Install OpenStreetMap-Carto.
   ➔ Install OSM2PGSQL.
   ➔ Install OpenSeaMap Renderer.
   ➔ Create the OSM PostgreSQL database: "gis" and add the PostGIS extensions.
   ➔ Download and import the OpenStreetMap-Carto base shapefiles in the OSM PostgreSQL database: 'gis'.
   ➔ Import the OSM data of the dutch province: "Limburg".

   This scripts takes about 15 minutes to complete depending on the speed of your network-connection. After the script is complete 1.2GB of additional disk space is used!

9) Now when that script is done and no errors were encountered you can move on to the next script for installing the Cesium Terrain Server. This is done by running:
   bash ~/Installation-scripts/8-middleware-software-Cesiumserver.sh

   This script will do the following:

   ➔ Download and install Cesium Terrain Server.
   ➔ Download and install GDALWarp.
   ➔ Download and install ctb-quantized-mesh which is used to generate Cesium terrain files.
   ➔ Generate the Hamert DTM terrain files.
   ➔ Generate the Hamert DSM terrain files.

   This scripts takes about 20 minutes to complete depending on the speed of your network-connection. The reason the script takes a longer than the other scripts Is because generating terrain files is a lengthy process. After the script is done executing 1.2GB of additional disk space is used.

10) Now when that script is done and no errors were encountered you can move on to the next script for installing the NGINX web server. This is done by running:
    bash ~/Installation-scripts/9-middleware-software-NGINX.sh

    This script will do the following:

    ➔ Remove older NGINX version(s) and install the latest NGINX version.
    ➔ Install ModSecurity and the NGINX ModSecurity connector modules.
    ➔ Install the ModSecurity Core Rule Set (CRS).

    This scripts takes about 20 minutes to complete depending on the speed of your network-connection. The reason this script takes a bit longer than the other scripts is because the process of compiling the NGINX ModSecurity modules takes a while. After the script is done executing 100MB of additional disk space is used.

11) Now when that script is done and no errors were encountered you can move on to the next script for installing the frontend software required for the web applications. This is done running the command:
    bash ~/Installation-scripts/10-frontend-software.sh

    This script will do the following:

    ➔ Install the Angular CLI (Command Line Interface) Tool.
    ➔ Install the Dataset Dashboard Node Modules.
    ➔ Install the 2D Map Viewer Node Modules.
    ➔ Install the 3D Map Viewer Node Modules.

NOTE: During the execution of this script you are asked to share anonymous usage data with the Angular Team at Google under Google's Privacy Policy. Select No and press enter in all cases. During the script you will also see some NPM warnings. Don't worry about these warnings. These security / update warnings are normal when installing Angular Node modules. If you want to find more information related to updating Angular modules, you should read section 5.6.1 from this cookbook.

This scripts takes about 3 minutes to complete depending on the speed of your network-connection. After the script is complete 500MB of additional disk space is used.

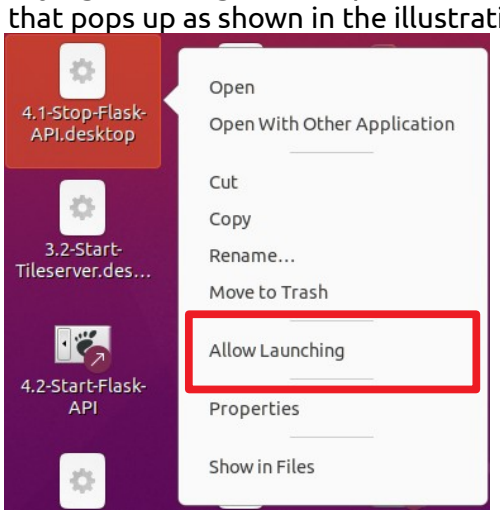
12) Now when that script is done and no errors were encountered you can move on to the script that builds the Docker containers. This script is optional so if you don't feel like using the Docker containers you don't have to run this script.
To run the script enter the command: `bash ~/Installation-scripts/11-docker-containers.sh`
This script will do the following:

- ➔ Install Xterm Terminal
- ➔ Build all the Docker containers.
- ➔ Stop the local instances of the datastores and NGINX.
- ➔ Start the Docker containers
- ➔ Import the corresponding datasets in a MongoDB data volume.
- ➔ Import the corresponding datasets in a PostgreSQL data volume.

NOTE: This script will take a while to complete since it builds all the GeoStack components in Docker containers and imports the required data!

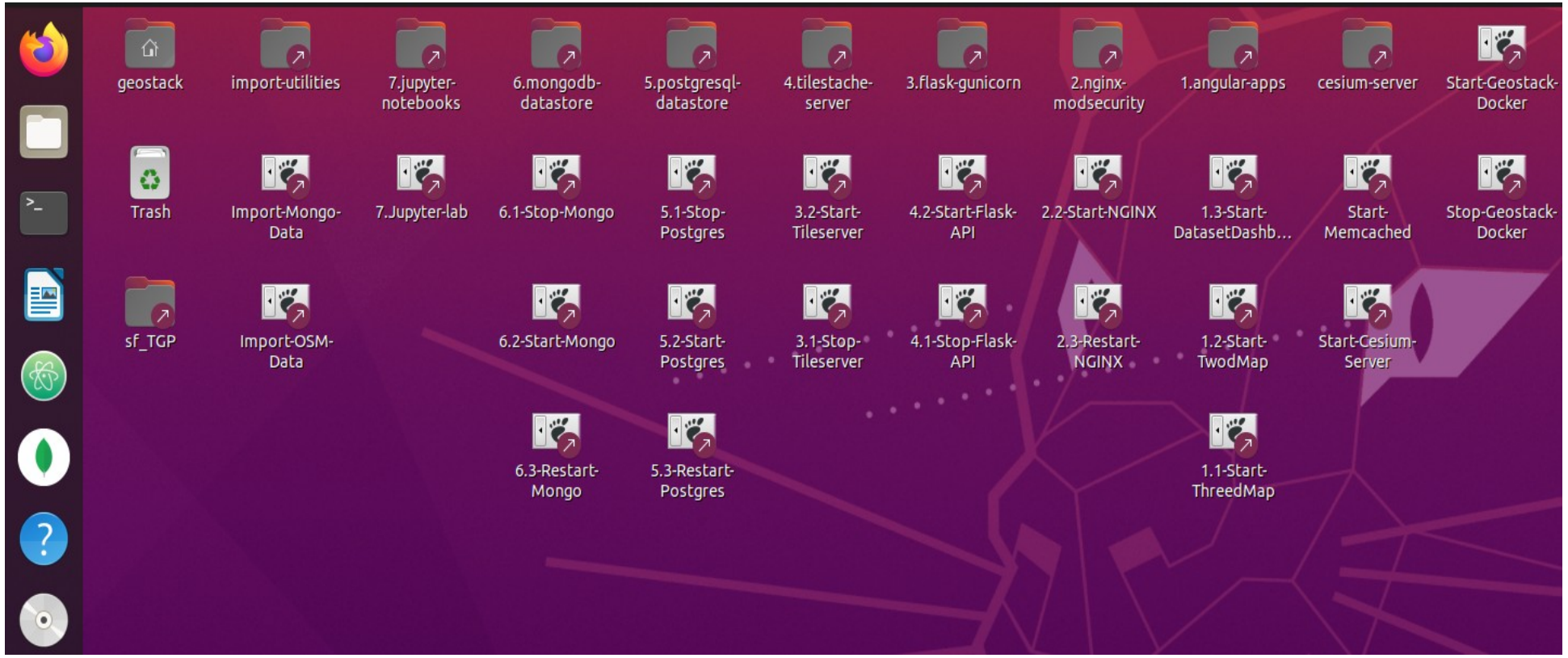
13) Now when that script is done and no errors were encountered you can move on to the last script which removes all the unused packages and clears the temporary files. This is done by running the application: "Bleachbit".
`bash ~/Installation-scripts/12-post-installation-cleanup.sh`
This script takes about 3 minutes to complete and clears about 1.7GB of disk space.

14) If you are on **Ubuntu 19.10 or 20.04** you need to allow the desktops shortcuts to be executable. This is done by right-clicking a desktop shortcut and the selecting Allow Launching in the menu that pops up as shown in the illustration below:



When the last script is done executing you will end up with a lot of shortcuts on you desktop. These shortcuts have to be organized. The illustration below shows the recommended way to order the shortcuts. You can change it where you see fit.

NOTE: This screenshot was created using Ubuntu 20.04.



4.4 How to continue the GeoStack Course?

Congratulations! You have completed the automatic installation of the GeoStack with the installation scripts so your VM is now up and running! Now go to chapter 6 ‘How to continue the GeoStack Course?’ to read what the best way is to continue!

5 Installing the GeoStack MANUALLY

So you chose to install the GeoStack manually? That's a good choice! It's highly recommend for you to clear some serious time in your schedule because this will be a lengthy but fun process which will result in a lot of knowledge on subjects reaching from data analyses up to and including data visualizations using web applications.

In the folder: "~/GeoStack-Course/Building-the-VM-using-the-installation-scripts/Geostack" you can find all the files and folders which you are going to create in the following chapters. These files can be used in case you get stuck during the creation of one or more of the components.

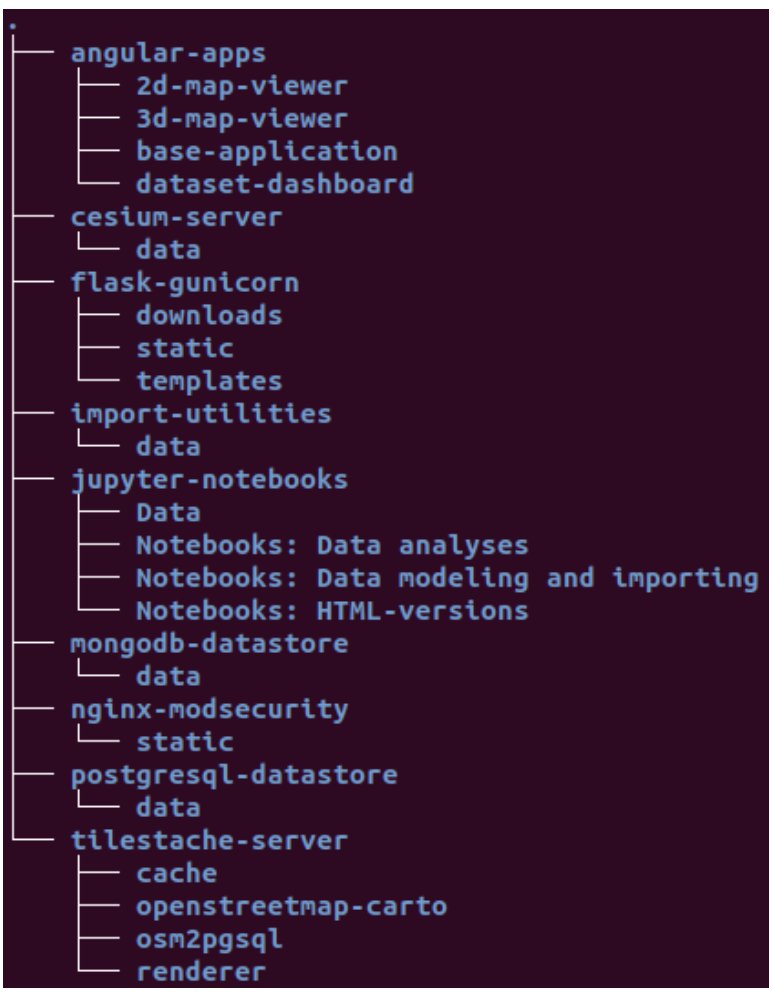
Have Fun and Best of Luck!

5.1 Creating the GeoStack folder

You are going to start of by creating the folder in which you are going to create and add all the GeoStack related files and folder. Create this folder by opening a terminal (Ctrl + Alt + T on your keyboard) and running the command: `mkdir ~/Geostack`

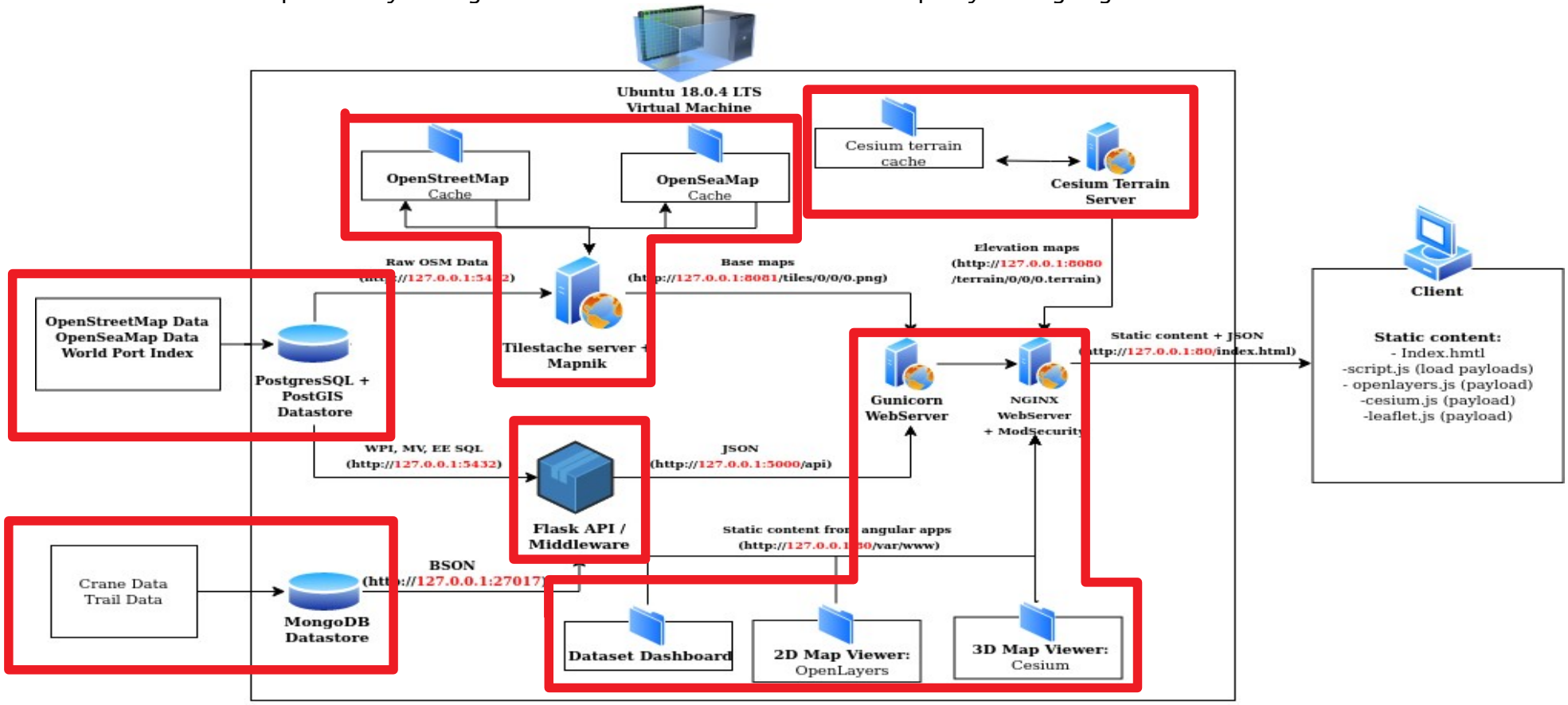
--> IMPORTANT NOTICE: the folder name is 'Geostack' and NOT the project name 'GeoStack'!

This Geostack folder is going to serve as the root folder for all the GeoStack components and utility scrips. Each component will be created in its corresponding folder.
The final Geostack root folder structure will look the same as shown in illustration below:



5.2 Installing the Docker virtualization software

The illustration below shows all the GeoStack components encircled in red. Each GeoStack component is Dockerized and thus can run separately from the other components. Because of this compartimentalization you can for example run the PostgreSQL database on a system at home and the TileStache tile server on a system located at work. The components are also installed in one Virtual Machine so everything can run locally on one machine. This is all possible by making use of Virtualization software. In this chapter you are going to install the tools for the virtualization.



So let's install the Docker virtualization software which is done in the following steps:

1) Update the local database by running the following commands: `sudo apt-get update`
2) Install the required dependencies for Docker.

   `sudo apt-get install apt-transport-https ca-certificates curl software-properties-common`
3) Install Docker by using the command: `sudo apt install docker.io`
4) Validate whether Docker has been correctly installed. The output should be the docker version. Check this by running the command: `docker --version`

Install Docker-compose by performing the following steps:

1) Enter the directory: "/usr/local/bin" by running the following command: `cd /usr/local/bin`
2) Download the required binary's and executable for Docker-compose by running the following command:

   ```
   sudo curl -L "https://github.com/docker/compose/releases/download/1.25.0-rc2/docker-\
   compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
   ```
3) Change the file permissions of the executable by running the following command:

   `sudo chmod +x /usr/local/bin/docker-compose`
4) Validate whether Docker-compose has been correctly installed by running the following command: `docker-compose --version`
5) Add docker to the current user and reboot the system for the changes to take effect by running the following command: `sudo usermod -a -G docker $USER && reboot`

Creating the Docker-compose configuration file

In the GeoStack folder, located in the Home folder of the VM, you are first going to create an empty file called 'docker-compose.yml'.
You do this by opening a terminal (with Ctrl + Alt + T) and running the command:
`touch ~/Geostack/docker-compose.yml`

For what is Docker Compose used?

In this configuration file you are going to add all the GeoStack components as services after creating them.

This configuration file is then used to run all the docker containers / components for the GeoStack at once which will take something like 10 – 15 minutes to get them all up and running.

Later in the GeoStack Course you will run this configuration file in Docker compose by running the command: "docker-compose up" from the directory (= folder) "Geostack" which is the root directory of the GeoStack.

But more on this later! Now continue by first installing some general software and tools which are used to increase your workflow during this **Beginner Course Open Source Geospatial Programming for Data Scientists.**

5.3 Installing the general software

1) Install Bleachbit, which is a tool to keep your Linux system clean, by running the following command: sudo apt install bleachbit

2) Install Curl (A package for downloading files using the CMI): sudo apt install curl

3) Install Net Tools (A package to analyse network statistics) : sudo apt install net-tools

4) Install LibreOffice Writer : sudo apt install libreoffice-writer

5) Install Python3-pip by running the following command:

sudo apt install python3-pip

IMPORTANT NOTE: as of Ubuntu Linux 20.04 there is no version of Python 2 installed anymore because it is deprecated. As of the year 2020 you should only use Python 3 and Python 3 tools!

1. If you still come across Python 2 tools we strongly advice against using them because it requires installing Python 2 next to Python 3 with a very serious risk of breaking your Operating System because it relies heavily on Python 3 for system management like for package management. Do not even try this in your working Virtual Machine!
2. If you must, then learn how to install and run Python 2 and the Python 2 tools you need only when they are securely detached from your OS in a virtual environment sandbox like with virtualenv! Always work in a copy of your Virtual Machine to keep the original safe!

6) Install GDAL for Geographical tools: sudo apt install gdal-bin

7) Install Atom, which is the code editor used during the cookbooks and programming manuals, by performing the following steps:

1) Add the Atom signing key to the system by running the following command:

   wget -qO - https://packagecloud.io/AtomEditor/atom/gpgkey | sudo apt-key add -

2) Add the Atom repository to the system's repositories list, using the following command:

   sudo sh -c 'echo "deb [arch=amd64] https://packagecloud.io/AtomEditor/atom/any/ any \
   main" > /etc/apt/sources.list.d/atom.list'

3) Update the local database and install Atom by running the following command:

   sudo apt update && sudo apt-get install atom

8) Install NodeJS & NPM, which are used for the tile server and the Angular applications, by performing the following steps:

1) Temporarily download the NodeJS installation script using the following command:

   curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -

2) Install the required packages for NodeJS by using the following command:

   sudo apt install build-essential

3) Install NodeJS by using the following command: sudo apt install nodejs
4) Validate whether NodeJS has been correctly installed the output should be a version of NodeJS. Check this by running the following command: node -v
5) Validate whether NPM has been correctly installed the output of this command should display the version of NPM. Check this by running the following command: npm -v

5.4 Installing data-analysis software

Now let's install the tools required for the analysis of the datasets used during the Beginners Course Open Source Geospatial Programming for Data Scientists.

Installing Jupyter Lab is done by performing the following steps:

1) Install the Jupyter package by running the following command:
   sudo -H pip3 install jupyterlab
2) Create a configuration file for a desktop shortcut to start Jupyter Lab by running the following command: touch ~/Desktop/Jupyter-lab.desktop

   NOTE: in this course you will create a lot of desktop shortcuts to conveniently start and stop the GeoStack components to speed up your workflow.
   The process of adding shortcuts in Ubuntu 19.10 or newer versions requires an extra configuration step for the 'Allow Launching' permission, so you should read section 7.1.1: "Creating desktop shortcuts" to learn how to get a working desktop shortcut!
3) Add the following code to the configuration file that was created on the desktop:

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "fuser -k 8888/tcp; jupyter lab"
Icon=gnome-panel-launcher
Name[en_US]=jupyter-lab
```

4) Make sure the shortcut is trusted by running the following command:

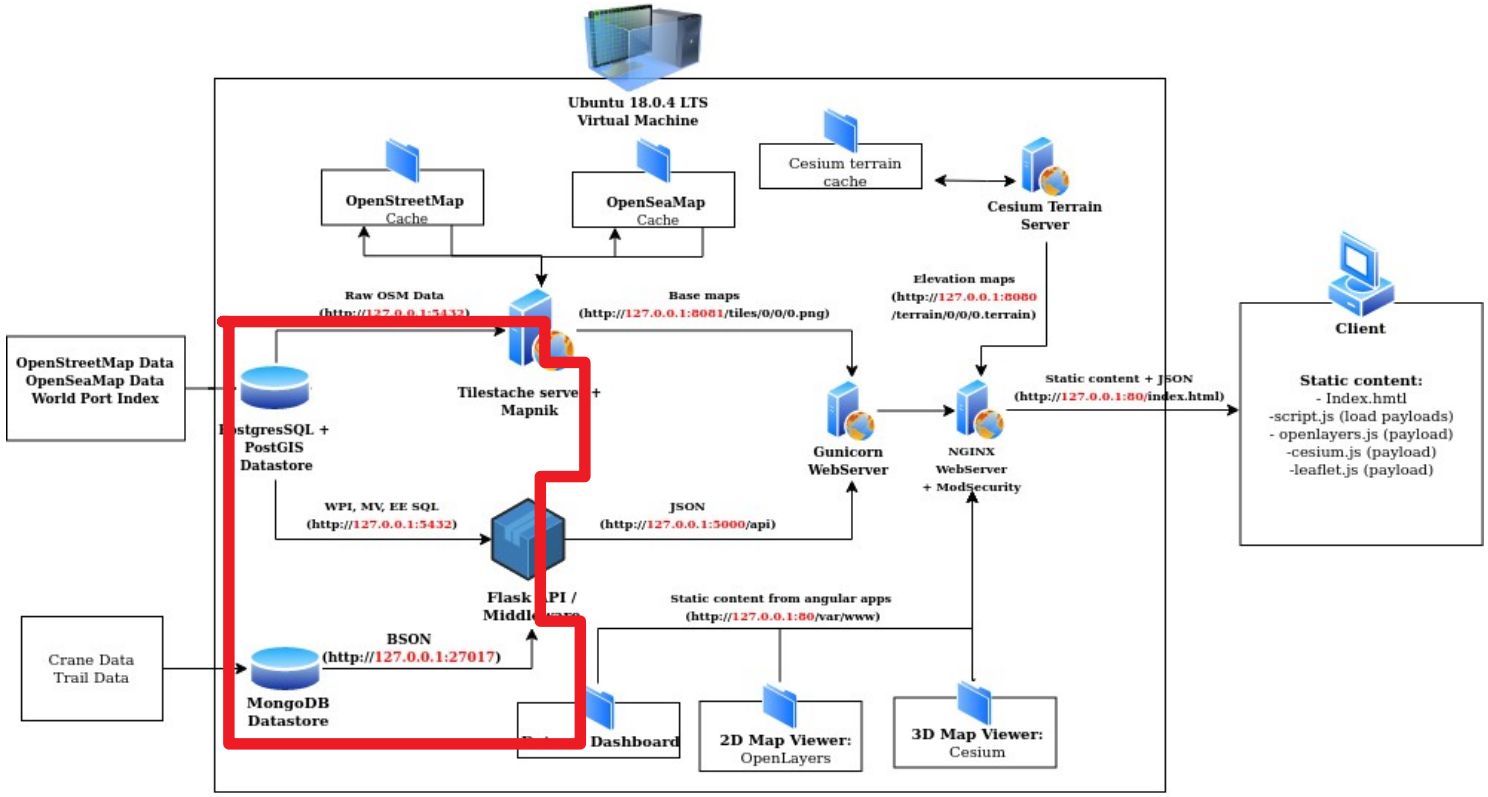
   for i in ~/Desktop/*.desktop; do gio set "$i" "metadata::trusted" yes ;done
5) Make sure the shortcut is launch-able by running the following command:
   sudo chmod +x ~/Desktop/Jupyter-lab.desktop
6) Now go to section 7.1.1. to learn how to set the 'Allow Launching' permission!

Now JupyterLab is installed and the desktop shortcut is working, continue installing the rest of the data analysis software:

1. **Install Pandas and GeoPandas :** pip3 install pandas geopandas
2. **Install Ubuntu libraries required for Cartopy:**
   sudo apt install libproj-dev proj-data proj-bin libgeos-dev
3. **Install Python packages required for Cartopy:** pip3 install cython
4. **Install MatPlotLib and Scipy:** pip3 install cartopy matplotlib scipy
5. **Install Cartopy:**
   pip3 install git+https://github.com/SciTools/cartopy.git --no-binary cartopy
6. **Install Python packages required for the GPX file format:**
   pip3 install gpxpy geopy numpy
7. **Install Pandas-Profiling:** pip3 install pandas-profiling

5.5 Installing the backend software

During the cookbooks you are going to work with multiple types of data formats. These data formats have to be transformed and then stored in the corresponding data store. During this section you are going to install all the tools and software required to perform this process.
First some information is discussed on how to perform certain actions which can come in useful later on in the course. Afterwards you are going to put this software and information to use in the cookbooks:"ETL-Process-with-datasets" and "Data-modeling-in-MongoDB-using-MongoEngine".
Note: things you will learn in the beginning of the section (Dockerizing, managing and importing in the datastores) will become more clear after reading the cookbooks mentioned above. This information is added in this section because it will be useful later on in this course.



5.5.1 Installing PostgreSQL

Install PostgreSQL by performing the following steps:

1) Add the official PostgreSQL repository key to the system.

   `wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -`

2) Add the PostgreSQL repository to the system's repository list.

   `sudo sh -c 'echo "deb [arch=amd64] http://apt.postgresql.org/pub/repos/apt/ \`
   `lsb_release -cs`-pgdg main" >> /etc/apt/sources.list.d/pgdg.list'`

3) Update the package database, using the following command: `sudo apt update`

4) **Depending on the Ubuntu version install PostgreSQL:**

   Install PostgreSQL 11 if you are using **Ubuntu 18.04 or lower** by the following command:

   `sudo apt install postgresql-11`

   Install PostgreSQL 12 if you are using **Ubuntu 19.10 or higher** by the following command:

   `sudo apt install postgresql-12`

5) Validate whether PostgreSQL has been correctly installed by running the following command: `sudo service postgresql status`

6) Login to the PostgreSQL user and the PostgreSQL CLI (Command line interface), using the following command: `sudo -u postgres psql`

7) Change default PostgreSQL password by running the following: `\password postgres`
   Then enter the password: `geostack`

8) Exit the PostgreSQL CLI by using the following command: `\q`

9) Create a desktop shortcut, which is used to start the PostgreSQL service, by running the following command: `touch ~/Desktop/Start-Postgres.desktop`

10) Add the following code to the file that is created on the desktop and save the file.

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "service postgresql start"
Icon=gnome-panel-launcher
Name[en_US]=Start-Postgres
```

11) Create a desktop shortcut, which is used to stop the PostgreSQL service, by using the following command: `touch ~/Desktop/Stop-Postgres.desktop`

12) Add the following code to the file that is created on the desktop and save the file.

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "service postgresql stop"
Icon=gnome-panel-launcher
Name[en_US]=Stop-Postgres
```

13) Make sure the shortcuts are trusted by running the following command:

    for i in ~/Desktop/*.desktop; do gio set "$i" "metadata::trusted" yes ;done

14) Make sure the shortcuts are launch-able by running the following command:

    sudo chmod +x ~/Desktop/Stop-Postgres.desktop && sudo chmod +x ~/Desktop/Start-\
    Postgres.desktop

Now depending on the Ubuntu version install PostGIS:

installing PostGIS on Ubuntu 18.04 or lower : sudo apt install postgis postgresql-11-postgis-2.5

installing PostGIS on Ubuntu 19.1 or higher : sudo apt install postgis postgresql-12-postgis-2.5

install PGMyAdmin: sudo apt install pgadmin4

install packages required for PsycoPG2:

sudo apt-get install libpcap-dev libpq-dev python-dev python3-dev

Install PsycoPG2: pip3 install psycopg2

5.5.1.1 Dockerizing PostgreSQL and PostGIS

Dockerizing the PostgreSQL datastore can be done by performing the steps below. As mentioned in the beginning of the section; Some steps may seem unclear at this point but don't worry about this. They will become more clear during the Course!

1) In the GeoStack folder, create a folder called: "postgresql-datastore" by running the following command: mkdir ~/Geostack/postgresql-datastore
2) Create a folder called: "data" which is going to contain all our data from the PostgreSQL Docker container. Do this by running the following command:

    mkdir ~/Geostack/postgresql-datastore/data

3) Next you are going to create a script which will run when the container is started for the first time. You are going to call this script: "initdb.-postgis.sh". Do this by running the following command: touch ~/Geostack/postgresql-datastore/initdb-postgis.sh
4) Open the file, add the following code to the file and save the file afterwards:

```
#!/bin/bash

# Create the user geostack and set the password to geostack
createuser geostack
psql -c "ALTER USER geostack WITH PASSWORD 'geostack';"

# Create the gis database with user: "gis" and add the required
# extensions to the database
createdb -E UTF8 -O geostack gis
psql -c "CREATE EXTENSION IF NOT EXISTS postgis;" gis
psql -c "CREATE EXTENSION IF NOT EXISTS postgis_topology;" gis
psql -c "CREATE EXTENSION hstore;" gis
```

The script above is going to run when the Docker container is being build for the first time. The script makes sure a database called: "gis" is created in which the required PostGIS extensions are loaded.

5) Create a file called: “Dockerfile” in the folder postgresql-datastore. This file is going to contain all the logic required to build the PostgreSQL Docker container. Do this by running the following command: touch ~/Geostack/postgresql-datastore/Dockerfile

6) Now open the newly created Docker file which is located in the folder: “~/Geostack/postgresql-datastore/” and add the following to this file and save it afterwards:

```
# The line below creates a layer from the PostgreSQL V. 11 Docker image.
FROM postgres:11

# Set  build environment to the versions required for our PostgreSQL installation.
ENV POSTGIS_MAJOR 2.5
ENV POSTGISV 2.5

# Download the required packages, software and modules using the environment
# variables which we set above. After the modules are downloaded, they are
# unziped and compiled.
RUN apt-get update \
 && apt-get install -y --no-install-recommends \
 postgresql-$PG_MAJOR-postgis-$POSTGISV \
 postgresql-$PG_MAJOR-postgis-$POSTGISV-scripts \
 postgresql-$PG_MAJOR-pgrouting \
 postgresql-$PG_MAJOR-pgrouting-scripts \
 postgresql-server-dev-$PG_MAJOR \
 unzip \
 make \
 && apt-get purge -y --auto-remove postgresql-server-dev-$PG_MAJOR make unzip

# Create a directory in which we will copy the script that creates the database.
RUN mkdir -p /docker-entrypoint-initdb.d

# Copy the script which will run when the container is started for the first time
COPY ./initdb-postgis.sh /docker-entrypoint-initdb.d/postgis.sh

# Set the permissions of the postgis.sh init script.
RUN chmod +x /docker-entrypoint-initdb.d/postgis.sh
```

7) Add the following to the docker-compose.yml, which is located in the folder: “~/Geostack”, and save it afterwards.

```
#Define the docker compose version
version: '3.7'

#Defining the GeoStack services  (components) is done below
services:
  # Here we define the name of the PostgreSQL datastore Docker service
 postgresql-datastore:
   container_name: postgresql-datastore
   # Set the directory in which the dockerfile is located
   build: ./postgresql-datastore
   # Add the data volume of the docker container
   volumes:
    - ./postgresql-datastore/data:/var/lib/postgresql/data
   # Set the port on which the docker container is available to port 5432
   # Since we set it to port 5432:5432, the docker container will also  be accessible on
   # our host system via localhost:5432
   ports:
    - '5432:5432'
   # Here we add the environment variable which allows connections without a pass.
   environment:
     POSTGRES_HOST_AUTH_METHOD: "trust"
```

5.5.1.2 Importing data in the PostgreSQL Docker datastores

The process of importing data in a Dockerized PostgreSQL database is similar to importing data in a Local PostgreSQL database instance. Before starting the import process you have to make sure that the Local PostgreSQL instance is not running and the Docker PostgreSQL instance (container) is running with a port that is exposed to our local system (Localhost).

Making sure the docker container is exposed to the Localhost is done by setting the ports of the postgresql-datastore service in the docker-compose.yml file to: **'5432:5432'.** As you can see in the code above you set the container to also be available on our localhost.

This means that the PostgreSQL database is available locally and in the docker container on port: 5432. If you set it to just 5432 the docker container will not be exposed to the LOCALHOST. Doing this comes in handy if you want to run a Local and a Dockerized database simultaneously.

5.5.1.3 Exporting the PostgreSQL Docker data volume

Exporting a PostgreSQL Docker data volume can be done by performing the following steps:

1) Change the permissions of the PostgreSQL Data volume to the permissions of our current user by running the following command:

   `sudo chown -R $USER ~/Geostack/postgresql-datastore/data`

2) ZIP the data folder so that the size is decreased which is useful when distributing the PostgreSQL Docker data volume to other systems. This is explained later!

   NOTE: When rebuilding a Docker container with a large data volume attached to it the rebuilding process will take longer depending on the size of the data volume!

5.5.1.4 Managing the PostgreSQL Databases

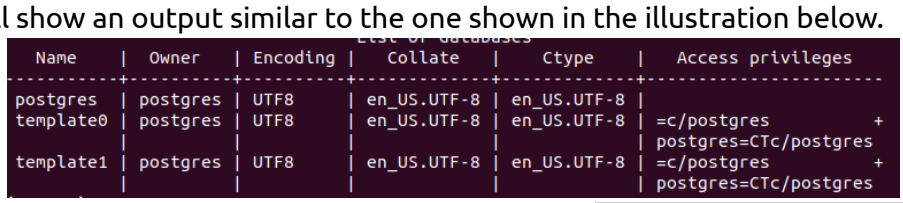
To manage the PostgreSQL databases you can perform the following steps:

1) Open a terminal by pressing the key combination Ctrl + Alt + T on you keyboard.

2) Login to the PostgreSQL user and the PostgreSQL CLI by running the following command and entering the password: "geostack": `sudo -u postgres psql`

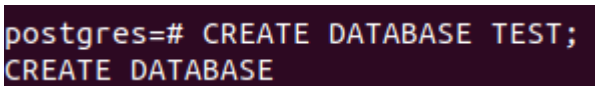
   This will log you in to the PostgreSQL user and open the PostgreSQL CLI. You can type the command \h to see all available commands for the PostgreSQL CLI.

3) List the current databases by entering the command: `\l`
   This will show an output similar to the one shown in the illustration below.



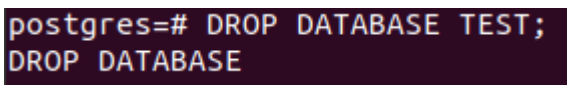
```
                                  List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
postgres   | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
template0  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
template1  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
```

4) You can **create** databases by running the command: `CREATE DATABASE {DB_NAME};`



```
postgres=# CREATE DATABASE TEST;
CREATE DATABASE
```

5) You can **delete** databases by running the command: `DROP DATABASE {DB_NAME};`



```
postgres=# DROP DATABASE TEST;
DROP DATABASE
```

6) Select a database by running the command: \c {name of the database}. This will connect you to the selected PostgreSQL database as shown in the illustration below. NOTE: You will not have a database called: “gis” yet. You will be creating this database later during the course!

```
postgres-# \c gis
psql (12.2 (Ubuntu 12.2-2.pgdg18.04+1), server 11.7 (Ubuntu 11.7-2.pgdg18.04+1))
You are now connected to database "gis" as user "postgres".
gis-#
```

7) You can list all the tables in the database by running the command: \dt
The output should be similar to the one shown in the illustration below.

```
gis-# \dt
              List of relations
  Schema  |        Name         | Type  |  Owner
----------+---------------------+-------+----------
 public   | planet_osm_line     | table | postgres
 public   | planet_osm_nodes    | table | postgres
 public   | planet_osm_point    | table | postgres
 public   | planet_osm_polygon  | table | postgres
 public   | planet_osm_rels     | table | postgres
 public   | planet_osm_roads    | table | postgres
 public   | planet_osm_ways     | table | postgres
 public   | spatial_ref_sys     | table | postgres
 topology | layer               | table | postgres
 topology | topology            | table | postgres
(10 rows)
```

This illustration shows all the tables in our gis database which is used to store our raw OpenStreetMap data. You are going to create this database later when creating the TileStache tile server, so don’t worry about not having the database yet!

8) To show the contents of a table you can run the command: SELECT * FROM {table name};

9) To remove an entry from a table you can run the command:
DELETE FROM {table name} WHERE {table name}.{column name} = ‘{value}’;

10) You can create tables by running the command: CREATE TABLE TEST (test Int);

```
gis=# CREATE TABLE TEST ( test Int );
CREATE TABLE
```

11) You can delete tables by running the command: DROP TABLE TEST;

```
gis=# DROP TABLE TEST;
DROP TABLE
```

12) To exit the PostgreSQL CLI you can type the command : exit

For more information related to the PostgreSQL CLI commands, you should read the documentation which can be found on the following URL: https://www.postgresql.org/docs/

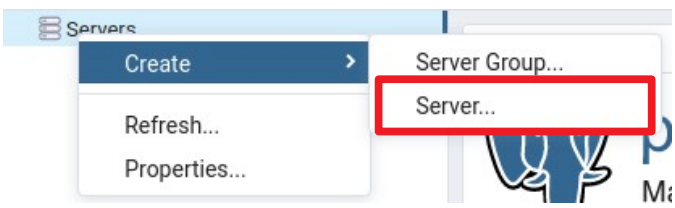
You can also use the GUI version called: “PGAdmin 4”. You already installed this application which can be started by clicking on the shortcut shown in the illustration below.

5.5.2 Setup PGAdmin4 to manage PostgreSQL databases

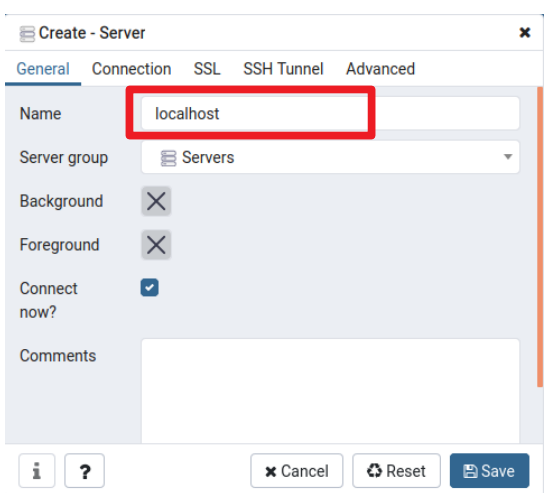
To easily manage PostgreSQL database you can also use the GUI version called: “PGAdmin4”. You already installed this software product which can be started by clicking on the shortcut shown in the illustration below which can be found by searching for “PGAdmin” in the systems applications.

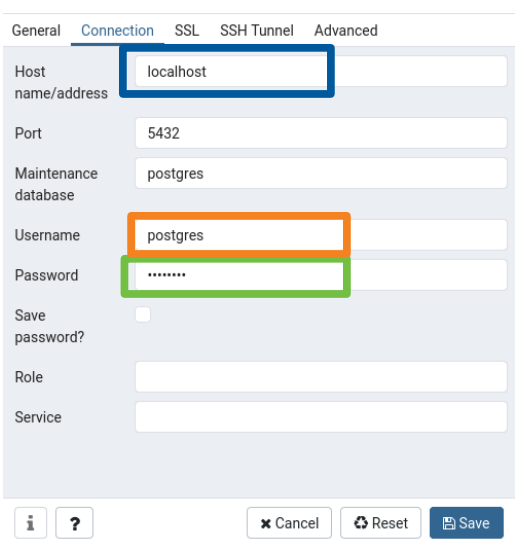
Loading PGAdmin4 for the first time can take a while but after a few seconds the screen shown in the illustration below will show up as an Tab in your Firefox browser:

Right click on the servers entry in the left menu then select Create → Server as shown in the illustration below:

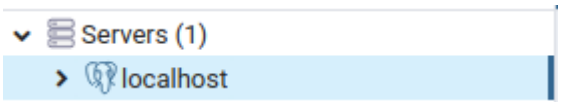


The set the Name (Red) and Host name / Address (Blue) to localhost, the username (Orange) to postgres and the password (Green) to geostack and select Save as shown in the illustrations below:

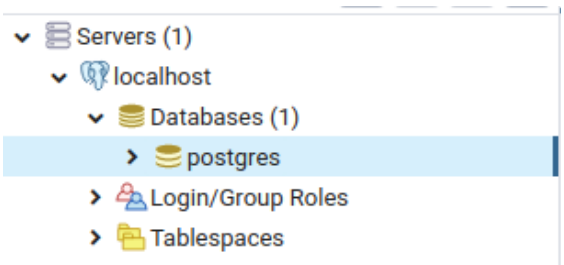




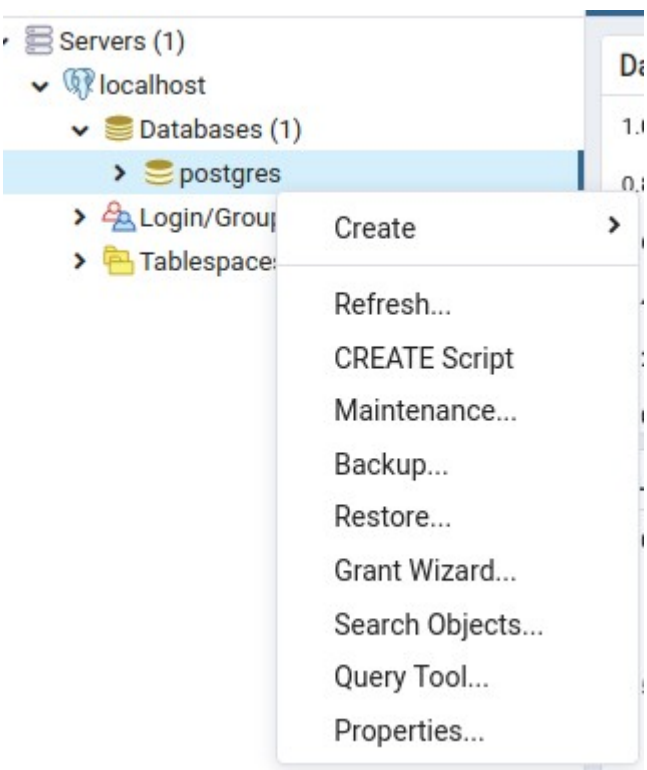
As you can see you now have a new server called: “localhost” as shown in the illustration below:



When you click on localhost → databases you will get a list of all the database currently running on our local PostgreSQL instance as shown in the illustration below.



You can manage database by right-clicking on the desired database an choosing one of the options shown in the menu that pops up as shown in the illustration below:



For more in depth instructions related to PGAdmin4 you should visit the following URL: https://www.pgadmin.org/

5.5.3 Installing MongoDB

Install the libraries which are required for MongoDB: sudo apt install libgconf-2-4

Install MongoDB by performing the following steps:

1) Install the MongoDB package by running the following command:

   sudo apt install mongodb

2) Validate if MongoDB is correctly installed by running the following command: mongo

3) Create a desktop shortcut to start the MongDB service, using the following command:

   touch ~/Desktop/Start-Mongo.desktop

4) Add the following code to the file that is created on the desktop and save the file:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "service mongodb start"
Icon=gnome-panel-launcher
Name[en_US]=Start-Mongo
```

5) Create a desktop shortcut to stop the MongoDB service by running the following command: touch ~/Desktop/Stop-Mongo.desktop

6) Add the code below to the file that is created on the desktop and save it afterwards:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "service mongodb stop"
Icon=gnome-panel-launcher
Name[en_US]=Stop-Mongo
```

7) Make sure the shortcut is trusted by running the following command:

   for i in ~/Desktop/*.desktop; do gio set "$i" "metadata::trusted" yes ;done

8) Make sure the shortcuts are launch-able by running the following command:

   sudo chmod +x ~/Desktop/Stop-Mongo.desktop && sudo chmod +x ~/Desktop/Start-\
   Mongo.desktop

Installing MongoCompass is done by performing the following steps:

1) Download the MongoCompass package by running the following command:

   wget https://downloads.mongodb.com/compass/mongodb-compass_1.20.4_amd64.deb

2) Install the MongoCompass package by running the following command:

   sudo dpkg -i mongodb-compass_1.20.4_amd64.deb

3) Remove the MongoCompass package by running the following command:

   sudo rm mongodb-compass_1.20.4_amd64.deb

Install MongoEngine: pip3 install mongoengine

5.5.3.1 Dockerizing MongoDB

To Dockerize the MongoDB datastore you have to perform the following steps:

1) Create a folder called: "mongodb-datastore" in the Geostack folder by running the following command: mkdir ~/Geostack/mongodb-datastore

2) Create a folder called: "data". This folder is going to contain all our data from the MongoDB Docker container.

   mkdir ~/Geostack/mongodb-datastore/data

3) Add the following to the docker-compose.yml file which is located in the Geostack folder:

```
# Here we define the name of the MongoDB datastore Docker service
mongodb-datastore:
 # Here we define the name which the MongoDB container is going to have
 container_name: mongodb-datastore
 # Here we define the image that is used for this container
 image: mongo:latest
 # Add the data volume of the docker container
 volumes:
  - ./mongodb-datastore/data:/data/db
 # Set the port on which the docker container is available to port 27017
 # Since we set it to port 27017:27017, the docker container will also  be
 # accessible on our host system via localhost:27017
 ports:
  - '27017:27017'
```

   NOTE: in the case of 'Dockerizing' the MongoDB datastore, you do not have to create a Dockerfile because you don't have to add any extra logic when creating the MongoDB Docker container.

4) Build the new service you just added in the docker-compose.yml file.

   cd ~/Geostack && docker-compose build

   The command above will build the PostgreSQL and MongoDB images which you added in this section and in the previous section.

5.5.3.2 Importing data in the MongoDB Docker datastore

The process of importing data in a Dockerized MongoDB database is similar to importing data in a Local MongoDB database instance. Before starting the import process you have to make sure that the Local MongoDB instance is not running and the Docker MongoDB instance (container) is running with a port that is exposed to our local system (Localhost).

Making sure the Docker container is exposed to the Localhost is done by setting the ports of the mongodb-datastore service in the docker-compose.yml file to: **'27017:27017'.** As you can see in the code above you set the container to also be available on our localhost.

This means that the MongoDB database is available locally and in the docker container on port: 27017. If you set it to just 27017 the docker container will not be exposed to the Localhost. Doing this comes in handy if you want to run a Local and a Dockerized database simultaneously.

5.5.3.3 Exporting the MongoDB Docker data volume

Exporting a MongoDB Docker data volume can be done by performing the following steps:

1) Change the permissions of the MongoDB Data volume to the permissions of our current user by running the following command:

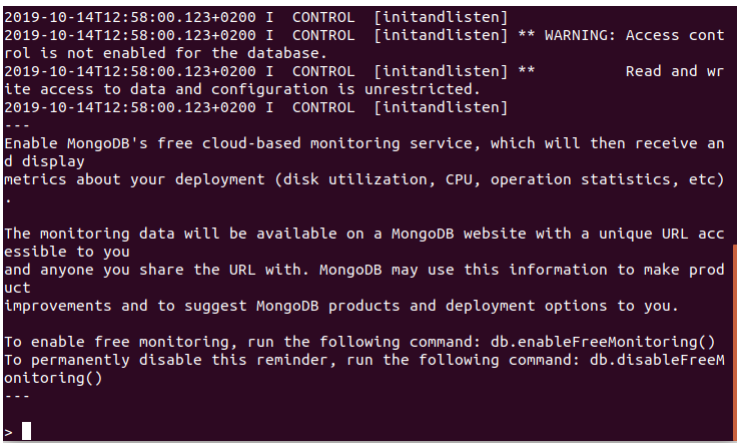
   `sudo chown -R $USER ~/Geostack/mongodb-datastore/data`

2) Zip the data folder so that the size is decreased which is useful when distributing the MongoDB Docker data volume to other systems.

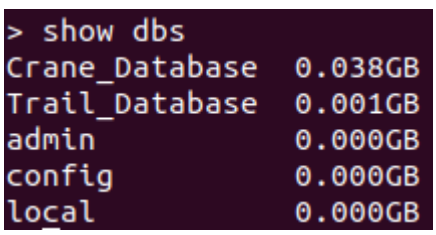
5.5.3.4 Managing the MongoDB Databases

If you want to remove a database from you MongoDB instance, take the following steps:

1) Open a terminal (with Ctrl + Alt + T) and enter the command: `mongo`
   This will open a MongoDB CLI as shown in the illustration below.

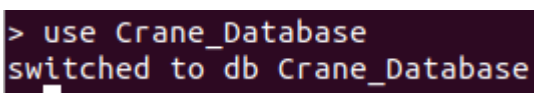


2) To get a list of all the databases on our system, enter the command: `show dbs`

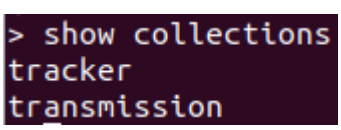


Note: You will not have the Crane and Trail Databases yet. You will create them later!

3) Look for the name of the database and select it with: `use {name of the database}`



4) You can display the collections in a database by running the command: show collections



5) To remove the database enter the command: `db.dropDatabase()`

Cleaning MongoDB Completely!
If you want to completely clean your local instance of MongoDB you should run the following command to remove everything from your local instance:

```
mongo --quiet --eval 'db.getMongo().getDBNames().forEach(function(i)\
{db.getSiblingDB(i).dropDatabase()})'
```

WARNING: this will remove all the content and data from MongoDB which is the following:

- ➔ **All databases**
- ➔ **All indexes**
- ➔ **All documents**

5.6 Installing the Datasets

5.6.1 Getting datasets with the Cookbook ETL Process with datasets

Now that you have all the tools, libraries and software products required for your data-analyses you can continue by performing the data-analyses on the datasets.

The cookbook: “ETL-Process with datasets” gives information related to performing the ETL-Process using one Crane (Tracker) dataset as example. The data analyses steps which you are going to perform are the first 2 steps of the so called: “ETL-Process”.

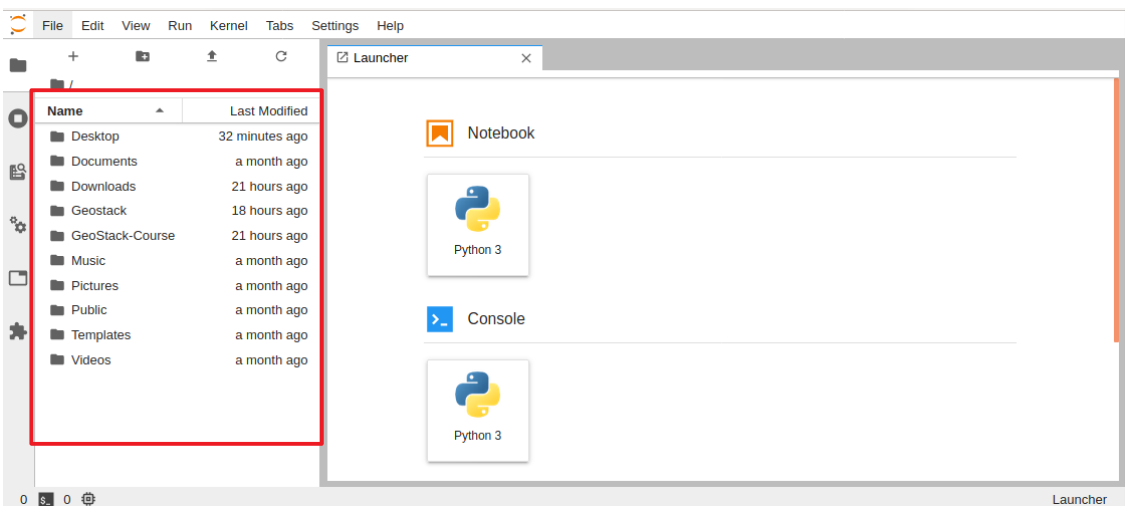
You are going to use Jupyter Lab application in which you are going to create a Jupyter Notebook. In this Jupyter Notebook you are going to create the code which is required to analyze the datasets.

To start Jupyter Lab you can click on the desktop shortcut (that was created earlier) as shown in the illustration below:

This will open a terminal telling you the Jupyter Lab server has been started on port 8888 as shown in the illustration below:

```
    To access the notebook, open this file in a browser:
        file:///home/geostack/.local/share/jupyter/runtime/nbserver-568-open.htm
l
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=564ac4276d147db4daf49b5b72ab0b6bf9763607ad4
450b7
     or http://127.0.0.1:8888/?token=564ac4276d147db4daf49b5b72ab0b6bf9763607ad4
450b7
[I 12:55:14.000 LabApp] Build is up to date
```

A Firefox window with the Jupyter Lab instance will also be opened as shown in the illustration below:

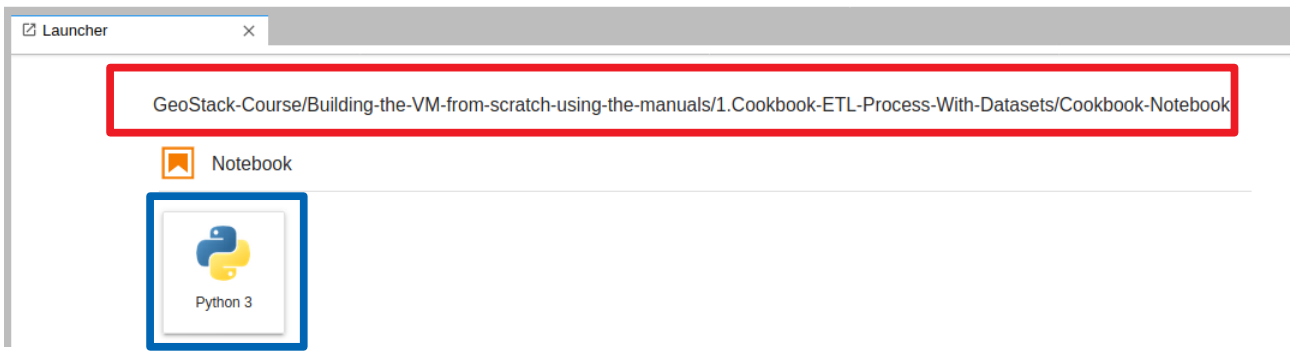


Before you can start reading the cookbook: “ETL-Process with datasets” you should create a new Jupyter Notebook in which you are going to create the code.

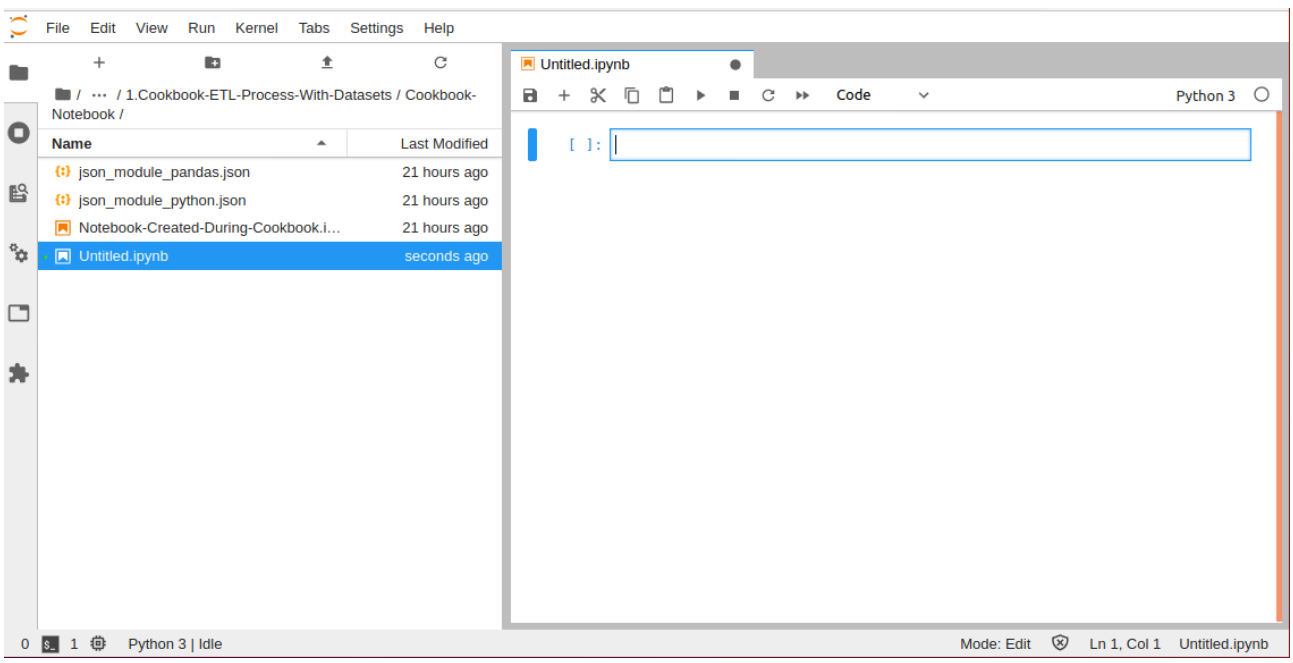
You should create this new Jupyter Notebook in the same folder as the notebook called: “Notebook-Created-During-Cookbook” which can be found in the folder: “~/GeoStack-Course/Building-the-VM-from-scratch-using-the-manuals/1.Cookbook-ETL-Process-With-Datasets/Cookbook-Notebook/”.

So navigate to this folder using the sidebar which is encircled in red in the illustration above.

Now when you are navigated to the folder: “~/GeoStack-Course/Building-the-VM-from-scratch-using-the-manuals/1.Cookbook-ETL-Process-With-Datasets/Cookbook-Notebook/” (boxed in red in the illustration below) you should click on the Python 3 button below the Notebook entry (encircled in blue in the illustration below):



This will open a new notebook as shown in the illustration below:



In this notebook you are going to create all the code which is described in the cookbook:“ETL-Process with Datasets”. Save the notebook with a filename of your choice and save it regularly!

So keep this window open and start reading the cookbook: “ETL-Process with Datasets” which can be found in the folder: ‘~/GeoStack-Course/Building-the-VM-from-scratch-using-the-manuals/1.Cookbook-ETL-Process-With-Datasets’.

In this cookbook you will learn about the following subjects and many more:

- ➔ How to extract data from a data source;
- ➔ How to filter data for the application goals;
- ➔ How to transform data to the file formats JSON and GeoJSON.

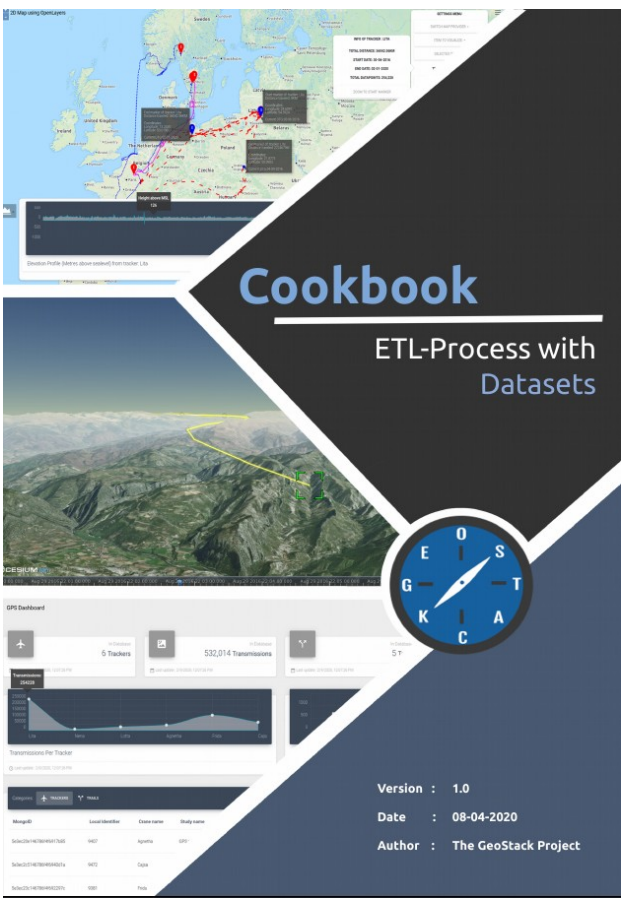
As mentioned before; this cookbook uses ONE Crane dataset as an example. The notebook used during this cookbook can be found in the folder: “Cookbook-Notebook” which is located in the same folder as the cookbook: “ETL-Process with datasets”.

In the folder of the cookbook you will also find a folder called: “Remaining-Analyses-Notebooks”. This folder contains the Jupyter notebooks for the data-analyses of the following datasets:

- ➔ Crane (Tracker) datasets;
- ➔ GPS Route (Trails) datasets;
- ➔ World Port Index dataset.

As you will see in the notebooks the process of performing a data-analyses on the datasets, which are not described in the cookbook: “ETL-Process with datasets” is similar to the one described in the cookbook. You should run all the cells in the remaining notebooks since you need to have these datasets transformed to reach the end-goals in our applications.

But now first read the cookbook: “ETL-Process with Datasets” as shown in the illustration below:



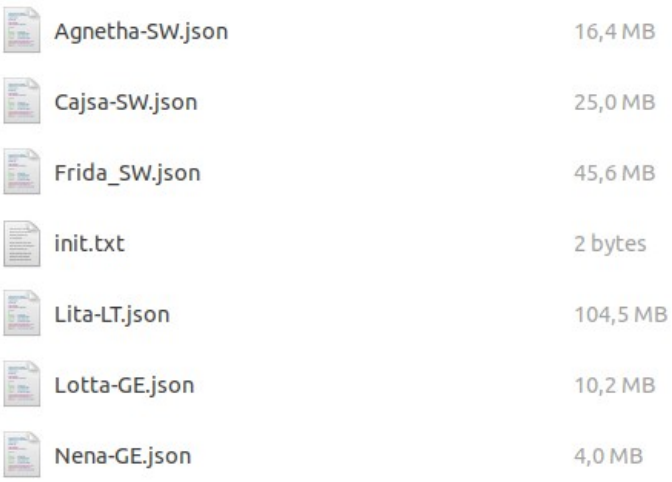
After you have finished reading this cookbook and completing the assignments in the new Jupyter notebook you created for it, you can continue reading this cookbook from here on!

5.6.2 Checking the dataset files for the GeoStack Course

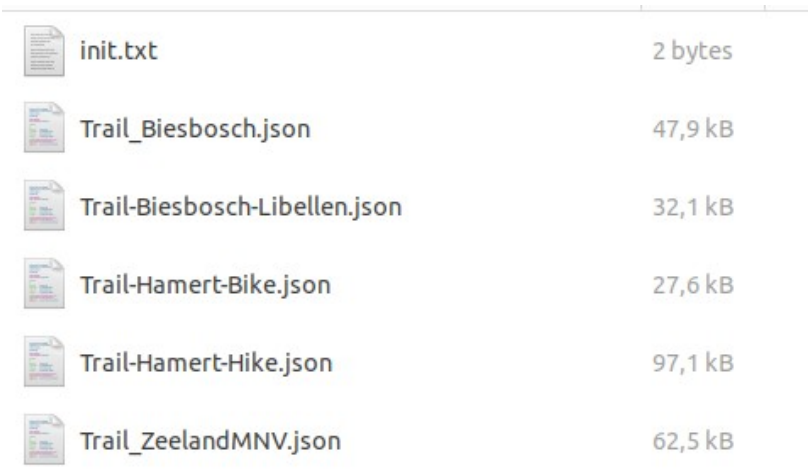
At this point you should have read the cookbook: “ETL-Process with datasets” and the notebooks in the folder: “Remaining-Analyses-Notebooks”. If you did everything accordingly you should end up with the following files in the following folders:

➔ The folder: “~/GeoStack-Course/Course-Datasets/JSON/Crane_GeoJSON” should contain the following files:

➔ The folder: “~/GeoStack-Course/Course-Datasets/JSON/Crane_JSON” should contain the following files:



➔ The folder: “~/GeoStack-Course/Course-Datasets/JSON/Trail_JSON” should contain the following files:



Now that you have analyzed and transformed the datasets, you continue by reading the cookbook: “Data-Modeling-in-MongoDB-using-MongoEngine” in which you will learn how to create a data model for the Crane datasets.

5.6.3 Modeling Datasets with the Cookbook Data Modeling in MongoDB

The cookbook can be found in the folder: '~/GeoStack-Course/Building-the-VM-from-scratch-using-the-manuals/2.Data-Modeling-in-MongoDB '.
In that folder you will find the cookbook: "Data-Modeling-in-MongoDB-using-MongoEngine".

In this cookbook you will learn the following and much more:

- ➔ How to create data models using MongoEngine.
- ➔ How to import the data in a MongoDB datastore using the model.
- ➔ How to create indexes on databases.

This cookbook uses the Crane dataset that was used in the cookbook: "ETL-Process-with-datasets", as an example.

The notebook containing the final code created during this cookbook can be found in the folder: "Cookbook-Notebook" which is located in the same folder as the cookbook: "Data-Modeling-in-MongoDB-using-MongoEngine"

In the folder of the cookbook you will also find a folder called: "Remaining-Data-Modeling-Notebooks".
This folder contains the Jupyter notebooks related to the data importing of the following datasets:

- ➔ The remaining Crane (Tracker) datasets;
- ➔ GPS Route (Trails) datasets;
- ➔ World Port Index dataset.

As you will see in the notebooks; the process of modeling the datasets, which are not described in the cookbook: "Data-Modeling-in-MongoDB-using-MongoEngine" is similar to the one described in the cookbook.

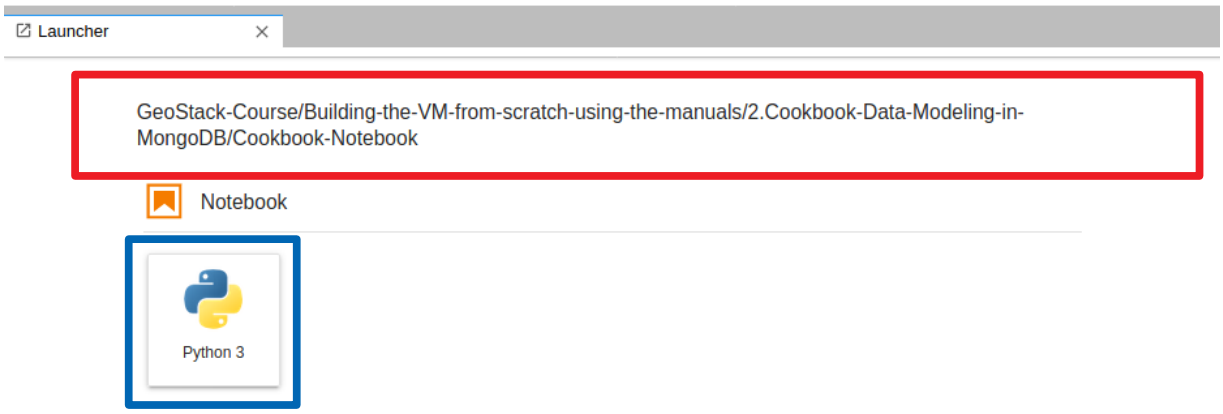
IMPORTANT NOTE: you should run all the cells in the remaining notebooks (!) since you need to have all these datasets modeled, imported and indexed in our MongoDB and PostgreSQL datastores to reach the end-goals to create working web applications.

First you create a new empty notebook in which you are going to add all the code which is described in the cookbook: "Data-Modeling-in-MongoDB-using-MongoEngine".

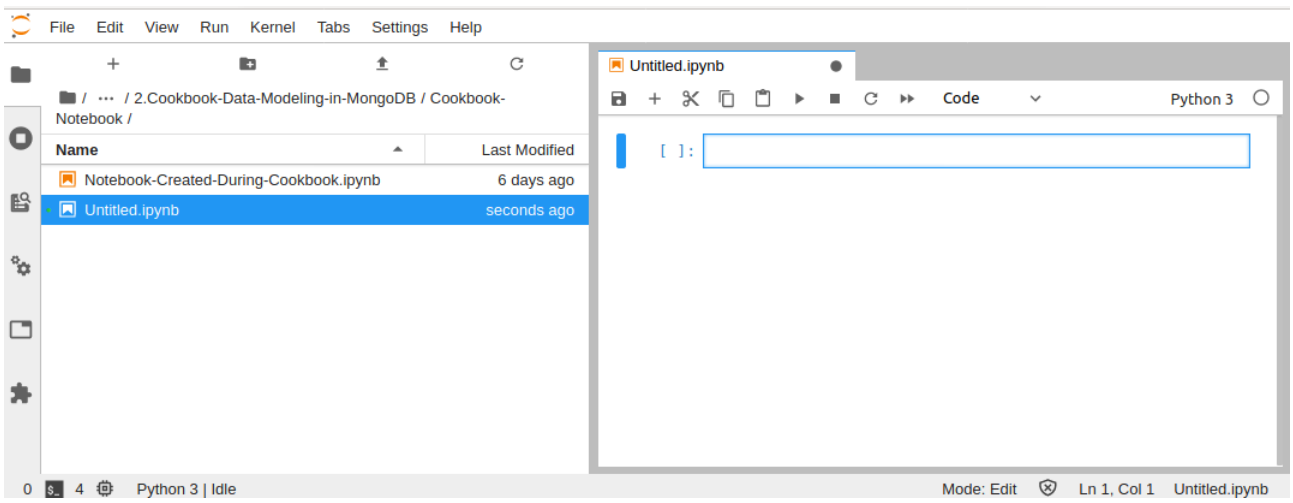
You do this by clicking the Jupyter-Lab shortcut on the desktop as shown below:

Once Jupyter lab is started you should navigate to the folder: "~/GeoStack-Course/Building-the-VM-from-scratch-using-the-manuals/2.Cookbook-Data-Modeling-in-MongoDB/Cookbook-Notebook/" (encircled in red in the illustration below)

You should now click on the Python3 button below the Notebook entry (encircled in blue in the illustration below):



Which will open a new notebook as shown in the illustration below:



Now you can start reading and completing the assignments in the cookbook: “Data-Modeling-in-MongoDB-using-MongoEngine” as shown in the illustration below.

You should add all the code, described in the cookbook, in the notebook you created above.

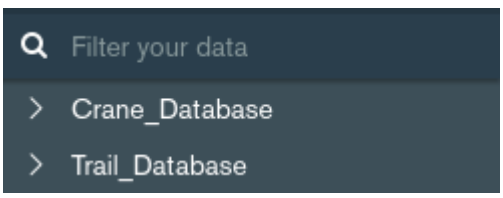
After you finished reading this cookbook and completing the assignments in the Jupyter notebooks that come with it, you should continue to read this cookbook from here on!

5.6.4 Checking the MongoDB databases

Now let's confirm if everything up till now has been done correctly. You start of by confirming if the Crane (Tracker) datasets and the GPS Route (Trail) Datasets have been correctly imported. This is done by performing the following steps:

1) Start MongoDB Compass by clicking on the sidebar shortcut icon which is shown in the illustration below:

2) The following database should be available in the left sidebar of the of the MongDB Compass start screen:



3) The following should be shown in the main screen of the MongoDB Compass start screen:

Database Name ▲	Storage Size	Collections	Indexes
Crane_Database	22.6MB	2	6
Trail_Database	213.0KB	2	6
admin	16.4KB	0	1
config	32.8KB	0	2
local	36.9KB	1	1

4) When clicking on the

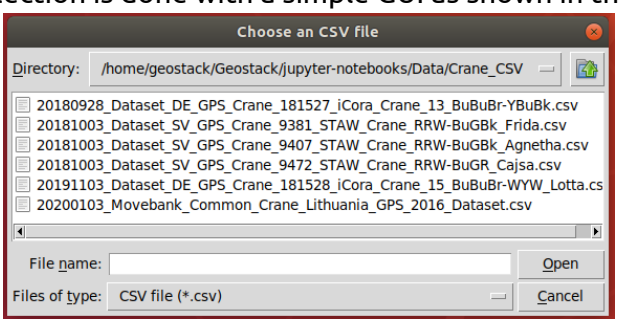
5.6.5 Automating the dataset import process for MongoDB

Now you know how to manually model, index and import datasets in a MongoDB datastore, it's time to learn **in this long section (!)** how to make a convenient Python script with a simple GUI which automates this process.

- Important note: for educational purposes it is good for Python beginners to just retype the code from the images in this section to learn how to code from scratch!
- Lot's of lines with inline comments will help you understand how the code works!
- Of course the full working script is provided too, in case you make a typo or coding error to compare your code to the working code.
- If you are more experienced in Python programming then save yourself some typing time, but read this section anyway because you still need to learn how the script works!
- The working script is mongo-data-import.py and this file can be found in the folder:
  - Building-the-VM-using-the-installation-script/Geostack/import-utilities

This script will do the following if you want to add new datasets to the MongoDB datastore:

➔ Give you the option to select and import a dataset file in either CSV, JSON or GPX format. The folder and file selection is done with a simple GUI as shown in the illustration below.



➔ Create a data model for the datasets and filter the datasets.

➔ Import the data in a database, which you can chose yourself, according to the data model.

➔ Create indexes on the newly imported dataset.

In some datasets the names of the longitude, latitude, altitude and timestamp columns differ.

- This script will ask you to enter the names of those columns to make sure that every type of dataset can be imported using the script.
- The script also employs multiple checks before the actual import process starts. This is useful to prevent the loading of incorrect data in the database.

First you need to install Python's standard GUI package Tkinter which is used for the simple GUI to select folders and files by using the following command: `sudo apt-get install python3-tk`

Now you are going to create a new folder in the Geostack folder. This folder is called: "import-utilities" and is going to contain all our scripts related to automating the import processes. This is done opening a terminal (with Ctrl + Alt + T) and entering the command:
`mkdir ~/Geostack/import-utilities`

Now you need to create a new Python file called: "mongo-data-import.py". You do this by running the following command: `touch ~/Geostack/import-utilities/mongo-data-import.py`

- Notice you create this file in the Geostack folder you created in your Home directory ('~'), while the provided working script is in the folder of the GeoStack Course repository you cloned from GitHub!

Now open this empty Python script file in the Atom code editor to start coding from scratch!

You start by adding the import statements for the modules which are required in the script. This is done by adding the following code at the top of the file:

```
# MongoEngine is used to create the datamodel and load the data according
# to the datamodel.
from mongoengine import *
# datetime is used to convert the Invalid datetimes to a valid format.
from datetime import datetime
# Pandas is used to create dataframe's and check the intergrity of the datasets.
import pandas as pd
# The tkinter module is used to create a GUI.
import tkinter
# The filedialog module is used to show a file selection GUI.
from tkinter import filedialog
# The gpxpy module is used to import GPX datasets
import gpxpy
```

Next you need to let the script know it needs to create a Tkinter GUI in the background. You do this by adding the following code below the module imports.

```
# Here we create a new instance of a Tkinter GUI.
GUI = tkinter.Tk()
# This line makes sure the GUI is instantiated in the background.
GUI.withdraw()
```

Next you are going to create 3 functions to import 3 different types of datasets for the CSV, JSON and GPX file formats. For the CSV import function add the following lines of code:

```
# Here we create a function called: "csv_import"
def csv_import():
    # Here we add the code logic required to open the selection GUI.
    # The following parameters are passed in this code:
    # - parent = the instance of the Tkinter GUI
    # - mode = specifies if the file shoulde be loaded or selected.
    # - filetypes = specifies the extensions which are allowed to be selected
    #               since the user chose CSV, we are only allowing CSV.
    # - title = the text displayed at the top of the GUI.
    input_file = filedialog.askopenfile(
    parent=GUI,mode='r',filetypes=[('CSV file','*.csv')],title='Choose an CSV file')

    # Here we check if the selected file is not equal to None.
    # If this is the case the following code is executed.
    if input_file != None:

        # Here we read the JSON file using pandas. We assign the dataframe
        # to a variable called: "df".
        df = pd.read_csv(input_file)

        # Here we call the function:"transform_data" in which we pass the
        # dataframe as parameter.
        transform_data(df)
```

Now you need to create the function which is used to import JSON datasets. It's an extra function in case you have datasets in JSON and you want to experiment with direct data import from these JSON files. You do this by adding the following code below the csv_import function:

```
def json_import():

    # Here we add the code logic required to open the selection GUI.
    # Here we pass JSON as the filetypes parameter.
    input_file = filedialog.askopenfile(
    parent=GUI,mode='r',filetypes=[('JSON file','*.json')],title='Choose an JSON file')

    # Here we check if the selected file is not equal to None.
    # If this is the case the following code is executed.
    if input_file != None:

        # Here we read the JSON file using pandas. We assign the dataframe
        # to a variable called: "df".
        df = pd.read_json(input_file)

        # Here we call the function:"transform_data" in which we pass the
        # dataframe as parameter.
        transform_data(df)
```

Before you are going to create the function which is used to import GPX datasets, you first need to create a function which creates a Pandas dataframe to enable working with raw GPX data (for instance: a GPX file that is exported from a GPS Navigation Device).

This is done by adding the following code below the json_import() function:

```
# Here we create a function called: "create_dataframe".
# The function takes raw GPX data as input.
def create_dataframe(data):

    # Here we create a new dataframe with the input columns
    # as columns.
    df = pd.DataFrame(columns=['lon', 'lat', 'alt', 'time'])

    # Here we loop through all the points in the list of data
    # which was passed as parameter in this function.
    for point in data:
        # We append the lon,lat,alt and time values to the
        # correct columns in the dataframe which was created earlier.
        df = df.append({'lon': point.longitude,
                        'lat' : point.latitude,
                        'alt' : point.elevation,
                        # Here we convert the datetime to a timestamp.
                        # This is required since we want to remove the timezone
                        # info from the timestamp.
                        'time' : datetime.timestamp(point.time)}, ignore_index=True)

    # Here we return the DataFrame
    return df
```

Now you can create the function gpx_import() which is used to import GPX data.
This is done by adding the following code below the create_dataframe() function.

```
# Here we create a function called:"gpx_import".
def gpx_import():

    # Here we add the code logic required to open the selection GUI.
    # Here we pass GPX as the filetypes parameter.
    input_file = filedialog.askopenfile(
    parent=GUI,mode='r',filetypes=[('GPX file','*.gpx')],title='Choose an GPX file')

    # Here we use GPXPY to parse the RAW GPX data.
    # We pass the input_file as parameter.
    parsed_file = gpxpy.parse(input_file)

    # Here we obtain the datapoints of the GPX dataset and assing it to a
    # variable called: "data".
    data = parsed_file.tracks[0].segments[0].points

    # Here we pass the data to the function:"create_dataframe".
    df = create_dataframe(data)

    # Here we pass the newly created dataframe to the transform_data function.
    transform_data(df)
```

As you can see in the code above, the function “transform_data()” is used. This function has not been created yet because you first need to create 3 extra functions.
You start with the function check_dataframe() to check if the dataframe has data in it or not by adding the following code below the gpx_import() function.

```
def check_dataframe(df):

    # Here we check if the amount of rows in the dataframe is bigger than 0.
    # We do this by using the syntax: ".shape[0]" on the dataframe.
    # If the dataframe size is bigger than 0 the following code is executed.
    if df.shape[0]>0:

        # Print the amount of rows in the dataframe.
        print("---->>Selected "+ str(df.shape[0])+ " rows.<<----")

        # Print the column names in the dataset.
        print("The dataset has the following columns:")
        print(df.dtypes)

        # Return True.
        return True
    # If the dataframe is not bigger than 0, it means the dataframe / dataset
    # is Invalid. If this is the case, the following code is executed.
    else:
        # Print that the dataframe is incorrect.
        print("Dataframe is incorrect, please try again\n")
        # Return False.
        return False
```

Next, you are going to create a function called:"column_selection()". This function is used to ask the user what the column names are of the longitude, latitude, altitude and timestamp columns.

This is required because not all datasets have the same columns names. This function makes sure that the import script can be used for every type of dataset.

```
# Here we create a function called: "column_selection".
def column_selection():

    # Here we create an empty list which is going to be populated with the user
    # inputs.
    selected_columns = []

    # Here we ask the user what the names of the ongitude, latitude, altitude
    # and timestamp columns are. We append the user inputs to the selected_columns
    # list.
    selected_columns.append(input('What is the name of the latitude column?\n'))
    selected_columns.append(input('What is the name of the longitude column?\n'))
    selected_columns.append(input('What is the name of the altitude column?\n'))
    selected_columns.append(input('What is the name of the timestamp column?\n'))

    # Finally populated list of selected_columns is returned.
    return selected_columns
```

Now you are going to create a function check_columns(). This function is used to check whether the columns, selected by the user in the previous function column_selection() are valid or not.

```
# Here we create a function called: "check_columns".
# This function takes a dataframe and a list of selected columns as input.
def check_columns(df,selected_columns):

    # We add a try/execpt to catch any errors if the following code fails after
    # which the function will return false.
    # The following code is always executed.
    try:
        # Here we loop through each column in the selected_columns list.
        for column in selected_columns:
            # Here we check if the column is found in the dataset.
            # If this is the case the code below is executed.
            # If this is not the case (so the column is not found), the function
            # will fail and trigger the except.
            if df[str(column)].shape[0] != 0:
                # Print that the column in found in the dataset.
                print("Found the " + str(column)+ ' column!')
        # Return True if all the columns are found.
        return True
    # The following code is executed if one or more columns are not found.
    except:
        # Return false if a column is not found.
        return False
```

Now the 3 extra functions are defined you can add the function transform_data() which is used to perform checks on the dataframe and the selected columns.
If everything is correct this function will trigger the data loading function which you will create later. So add the following code below the function check_columns().

NOTE: The following illustration is divided in 2 parts split over two pages.
The last line of the first illustration is the same as the first line of the second illustration.
You don't need to add this line twice.

Also notice in the code below the functions load_crane_data() to load the Crane datasets and load_trail_data() to load Trail datasets have not been created yet.
You will do this later after you have coded the main function import_tool() that follows this function transform_data().

```
def transform_data(df):
    # Here we check if the input dataframe is valid. If this is the case
    # the function:"check_dataframe" returns true, after which the following
    # code is executed.
    if check_dataframe(df):

        # Here we trigger the function: "column_selection" and assign the
        # selected_columns list to a variable called columns.
        columns = column_selection()

        # Here we check whether the selected_columns are in the dataframe by
        # triggering the function:"check_columns()" in which we pass the
        # dataframe and the selected_columns list. If the columns are found
        # the function:"check_columns" returns true, after witch the following
        # code is executed.
```

```
    # code is executed.
    if check_columns(df,columns):

        # Here we ask the user to choose between 2 options. We assign the
        # result of the user input to a variable called:"type"
        type = input(
        'What type of dataset do you want to import? \n[1]Crane\n[2]Route\n'
        crane={'1','crane','CRANE'}
        trail={'2','trail','TRAIL'}

        # Ask the user in which database the database has to be imported.
        dbname = input('To which database do you want to add the data?\n')

        # If the user chose to import a Crane dataset the following happens:
        if type in crane:
            print("--->> Importing a Crane dataset <<---")
            # Call the Crane data loading function and pass the dataframe
            # columns and database name as parameters.
            load_crane_data(df,columns,dbname)
        # If the user chose to import a Crane dataset the following happens:
        elif type in trail:
            print("--->> Importing a GPS Route dataset <<---")
            # Call the Crane data loading function and pass the dataframe
            # columns and database name as parameters.
            load_trail_data(df,columns,dbname)

        # If the user input is not in the words assigned to the variable:
        # "crane" or the variable: "trail" the following code is executed.
        else:
            print("Not a valid input, please try again\n")

    # If one or more selected_columns are not found the following code is
    # executed
    else:
        # Print the column(s) that was not found.
        print('One or more columns could not be found, please try again!')
        # Rerun this function to restart the column selection.
        transform_data(df)
# If the dataframe is Invalid the following code will be executed.
else:
    #Print that the dataframe is Invalid.
    print("Invalid dataframe, please check if the dataset is valid!")
    # Rerun the function.
    transform_data(df)
```

As mentioned above you will now first create the main function import_tool() which is triggered when running this Python script mongo-data-import.py.
This function let's you choose between an CSV, JSON or GPX dataset file.

To add this main function you add the following code below the transform_data() function.

NOTE: please read the inline comments that explain the source code of this function!

```
def import_tool():
    # Print the start text in the terminal.
    print('---->>WELCOME TO THE DATASET IMPORT TOOL<<----')

    # Here we ask the user to choose between 3 options. We assign the result
    # of the user input to a variable called:"format"
    format = input(
    'What file format does the dataset have? \n[1]CSV\n[2]JSON\n[3]GPX\n')

    # Here we create the options which the user can select to choose between
    # importing a CSV, JSON or GPX dataset.
    csv = {'1','CSV','csv'}
    json = {'2','JSON','json'}
    gpx = {'3','GPX','gpx'}

    # Here we define the code which is executed depending on the user input.
    # If the user input, which is assigned to a variable called: "format" is
    # in the list of words assigned to variable:"csv" the following code will
    # be executed.
    if format in csv:
        print("--->> Selected CSV file format <<---")
        csv_import()

    # Here we define the code which is executed depending on the user input.
    # If the user input, which is assigned to a variable called: "format" is
    # in the list of words assigned to variable:"json" the following code will
    # be executed.
    elif format in json:
        print("--->> Selected JSON file format <<---")
        json_import()

    # Here we define the code which is executed depending on the user input.
    # If the user input, which is assigned to a variable called: "format" is
    # in the list of words assigned to variable:"gpx" the following code will
    # be executed.
    elif format in gpx:
        print("--->> Selected GPX file format <<---")
        gpx_import()

    # If the user input is not in the words assigned to the variable:"csv" or
    # the variable: "json" the following code is executed.
    else:
        print("Not a valid input, please try again\n")
```

Now the workflow of the import script is complete you only need to add the 'payload code' to the script to write the datasets from the Pandas dataframe to the MongoDB database by adding the functions load_crane_data() and load_trail_data().

For each function you need to create the MongoDB database model for the JSON documents which have to be stored in the collections in the database in the Python script by adding class definitions because MongoDB is a schemaless datastore so the class definitions specifiy your data storage schema.

So, to create the import function load_crane_data() for the Crane datasets you first need to create a new Python file CraneModel.py which is going to contain the Crane data model and later you will use an import statement to import this Python file as a module in the main script!

You create this Python module file by opening a terminal (with Ctrl + Alt + T) and running the following command: touch ~/Geostack/import-utilities/CraneModel.py

Open this file and add the following module import statement at the top of the CraneModel.py file to enable the use of MongoDB through the module MongoEngine:

```
# MongoEngine is used to create the datamodel and load the data according
# to the datamodel.
from mongoengine import *
```

Now let's add the Crane data model to this Python script. You have already created the data model in the cookbook: "Data-modeling-in-MongoDB". You can copy this data model to the file. The illustrations on the next pages contain the code that should be copied to the Python script.

First, copy the data model for the JSON document TransmissionMetadata:

```
class TransmissionMetadata(EmbeddedDocument):

    #Is the tracker still visible or not?
    visible = BooleanField()

    # Type of sensor used in tracker.
    sensor_type = StringField()

    # Voltage level of the tracker.
    tag_voltage = FloatField()
```

Next, copy the data model for the JSON document Geometry:

```
class Geometry(EmbeddedDocument):

    # The coordinates of transmission
    # PointField automatically adds an 2dspehere index
    coord = PointField()

    # altitude of tansmission
    alt = FloatField()
```

Next, copy the data model of the JSON document Speed:

```
class Speed(EmbeddedDocument):

    # Speed of the Crane
    ground_speed = FloatField()

    # Heading of the Crane in degrees
    heading = IntField()
```

Then, copy the data model for the JSON document Transmission:

```
class Transmission(Document):

    # Identifier of the transmission
    event_id = IntField()

    # Timestamp of when transmission was send
    timestamp = DateTimeField()

    # Embedded geometry of transmission
    geometry = EmbeddedDocumentField(Geometry)

    # Embedded speed related data of transmission
    speed = EmbeddedDocumentField(Speed)

    # Embedded metadata of transmission
    metadata = EmbeddedDocumentField(TransmissionMetadata)

    # Reference to the tracker the transmission belongs to
    tracker = ReferenceField(Tracker)
```

NOTE: notice the specification of the embedded JSON documents geometry, speed and metadata, specified as the data type EmbeddedDocumentField for the defined classes Geometry, Speed and TransmissionMetadata.

Also notice the specification of the tracker field as a ReferenceField for the class Tracker as defined in the class definition below.

Finally, copy the data model of the JSON document Tracker:

```
class Tracker(Document):

    # Name of the study
    study_name = StringField()

    # Name of the bird, in latin.
    individual_taxon_canonical_name = StringField()

    # Id of the crane
    individual_local_identifier = IntField()

    #Start date of the study
    start_date = DateTimeField()

    #End date of the study
    end_date = DateTimeField()

    #Name of the crane
    name = StringField()

    #Amount of the transmissions related to the tracker
    transmission_Count= IntField()
```

Now that you have completed the data model for the crane database you need to go back to the mongo-data-import.py script file and import the CraneModel.py file as a Python module in this script.

You do this by adding the following import statement line below the import_tool() function:

```
# Here we import the CraneModel Python file.
import CraneModel
```

Now you can create the function load_crane_data() which is used to import the Crane datasets. This function takes a dataframe, a list of columns and the database name as its input.

This function is basically the same as the one you created in the cookbook:"Data-modeling-in-MongoDB-using-MongoEngine" but there are a few minor differences which are crucial to automating the import process.

So let's create this function by adding the code below the line of 'import CraneModel' as the first function definition in the script.

NOTE: The following illustration is divided in 2 parts split over two pages.
The last line of the first illustration is the same as the first line of the second illustration.
You don't need to add this line twice.

```
def load_crane_data(df,columns,dbname):

    # Ask the user what the name of the Crane has to be.
    name = input('What is the name of the item you want to import?\n')

    # Here we connect to the database that is passed as parameter (dbname)
    # when the function: "load_crane_data" is triggered.
    connect(str(dbname))

    # Create metadata for the tracker.
    # We use the value at the 3rd index of the list of columns which was passed
    # as parameter in this function. This index contains the value of the column
    # representing the timestamp (which was defined by the user input in the
    # function:"selected_columns()").
    start_Date = df.at[0,columns[3]]
    end_Date = df.at[df.shape[0]-1,columns[3]]
    transmission_Count = df.shape[0]

    # Create a new tracker, this is only done once. We first call the CraneModel
    # import and than the Tracker document in which we pass the required values
    # (from the dataframe which was passed as parameter in this function)
    # as parameters in the Tracker document.
    tracker = CraneModel.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'],
              start_date = start_Date,
              end_date = end_Date,
              name = name,
              transmission_Count = transmission_Count).save()

    # Create an empty list of transmissions to which will append the new
    # transmissions after they have been created. This list will be passed to
    # the mongodb bulk insert feature.
    transmissions = []

    # Print when list appending process starts.
    print('Start appending transmissions to list from: ' + str(name) )

    # For each row in the dataframe the following code is executed.
```

```
# For each row in the dataframe the following code is executed.
for index,row in df.iterrows():
    # Here we create geometry document.
    # We use the value at the 1st index of the list of columns which was passed
    # as parameter in this function. This index contains the value of the column
    # representing the longitude (which was defined by the user input in the
    # function:"selected_columns()"). We do the same for the latitude column
    # name at index 0 of the Selected columns list and the altitude column
    # name at index 2 of the selected columns list.
    geometry = CraneModel.Geometry(coord = [row[columns[1]],row[columns[0]]],
                        alt = row[columns[2]])

    # Here we create the metadata document in which we pass te required values.
    metadata = CraneModel.TransmissionMetadata(visible = row['visible'],
                       sensor_type = row['sensor-type'],
                       tag_voltage = row['tag-voltage'])

    # Here we create speed document in which we pass te required values.
    speed = CraneModel.Speed(ground_speed = row['ground-speed'])

    # Here we create a transmission document and append it to the
    # transmissions list. We use the value at the 3rd index of the column
    # list, passed in this function, as column name for the timestamp
    # column in the dataframe.
    transmissions.append(CraneModel.Transmission(event_id = row['event-id'],
                                    timestamp = row[columns[3]],
                                    geometry = geometry,
                                    speed = speed,
                                    metadata = metadata,
                                    tracker = tracker))

# Print when list appending is done.
print('Bulk inserting: '+str(transmission_Count)+' transmissions from: '+str(name))
# Bulk insert, the populated transmissions list, in the database.
CraneModel.Transmission.objects.insert(transmissions,load_bulk=True)
# Print if the insert process is succesfull.
print("Done inserting "+ str(len(df.index)) + " transmissions")
# Print that the indexing process has started.
print("Creating indexes on database ")
# Create and index on the tracker field in the Transmissions documents.
CraneModel.Transmission.create_index(("tracker"))
# Create and index on the timestamp field in the Transmissions documents.
CraneModel.Transmission.create_index(("timestamp"))
# Create and index on the coordinates field in the Transmissions documents.
CraneModel.Transmission.create_index(("geometry.coord"))
print("Done importing the dataset!")
# Here we make sure the tool is restarted after the dataset is imported
import_tool()
```

That’s it! The code required to automatically import Crane datasets is finished.

Now you want to do the same for the GPS Route (Trail) datasets by creating the data model and the load_trail_data() function. How this is done is shown on the next pages.

Now you want to create the import function load_trail_data() for the GPS Route (Trail) datasets. Before you can do this you need to create a new Python file TrailModel.py which is going to contain the Trail data model.

You create this file by running the following command:
touch ~/Geostack/import-utilities/TrailModel.py

Open this file and add the following module import at the top of the file:

```
# MongoEngine is used to create the datamodel and load the data according
# to the datamodel.
from mongoengine import *
```

Now add the Trail data model to this Python script. You have already created the data model in the Jupyter Notebook related to the GPS Routes (Trail). You can copy this data model to the script. The illustrations below contain the code that should be copied to the Python script.

Start by copying the class definition with the data model for the JSON document Trail:

```
class Trail(Document):

    # Name of the Trail
    name = StringField()

    # Abreviation of the Name
    abr = StringField()

    # Start date
    s_date= DateTimeField()

    # End date
    e_date = DateTimeField()

    # Trail type (Biking,Hiking,Driving,Sailing )
    r_type = StringField()

    # Amount of trackpoints in the dataset
    t_points = IntField()
```

Next, copy the data model for the JSON document Signal:

```
class Signal(Document):

    # Timestamp of signal
    time = DateTimeField()

    # Geometry of signal
    geometry = EmbeddedDocumentField(Geometry)

    # Reference to the route of signal
    trail = ReferenceField(Trail)
```

Finally, copy the data model for the JSON document Geometry:

```
class Geometry(EmbeddedDocument):

    # coordinates of signal coord=[1,2]
    coord = PointField()

    # altitude of signal
    alt = FloatField()
```

Now that you have to model you need to go back to the mongo-data-import.py script file and import the TrailModel in this script.

This is done by adding the line for the following import statement below the load_crane_data() function:

```
# Here we import the TrailModel Python file.
import TrailModel
```

Now create the function load_trail_data() which is used to import the Trail datasets.
This function takes a dataframe, a list of columns and the database name as input and is basically the same as the one which you created in Jupyter notebook related to the GPS Routes.

Add the following code below the line where you imported the TrailModel.

NOTE: The following illustration is divided in 3 parts. The last line of the first illustration is the same as the first line of the second illustration. You don't need to add this line twice.

```
def load_trail_data(df,columns,dbname):

    # Here we ask the user what the name of the Route should be.
    name = input('What is the name of the item you want to import?\n')

    # Here we ask the user what the abreviation of the Route should be.
    abreviation = input('What do you want the abreviation to be?\n')

    # Here we ask the user what the type of the Route should be.
    type = input('What is type of route is it (e.g. Hike, Bike, Car)?\n')

    # Here we connect to the database which was passed as input in this
    # function.
    connect(str(dbname))

    # Below we create metadata for the Route.

    # Here we get the value of the time column of the first row in the dataframe.
    # We use the value at the 3rd index of the list of columns which was passed
    # as parameter in this function. This index contains the value of the column
    # representing the timestamp (which was defined by the user input in the
    # function:"selected_columns()"). We apply /1000 to remove the UTC (Timezone)
    # info. This is required to create a valid timestamp.
    s_date = datetime.fromtimestamp((df.at[0,columns[3]]/1000))
```

```
    s_date = datetime.fromtimestamp((df.at[0,columns[3]]/1000))

    # Here we get the value of the time column of the last row in the dataframe.
    # This value is located at the index of the lenght of the dataframe.
    # We pass the value on the 3rd index of the list of columns which was passed
    # as parameter in this function. We apply /1000 to remove the UTC (Timezone)
    # info. This is required to create a valid timestamp.
    e_date = datetime.fromtimestamp((df.at[len(df.index)-1,columns[3]]/1000))

    # Get the total lenght of the dataframe we do this because it's the same
    # as the amount of signals in the dataset.
    t_points = df.shape[0]


    # Create the trial document by calling the Trail document in the TrailModel
    # in which we pass the required values.
    trail = TrailModel.Trail(name = name,
                 s_date = s_date,
                 e_date = e_date,
                 abr = abreviation,
                 r_type = type,
                 t_points = t_points).save()

    # Create an empty list of signals to which we will append all the signal
    # documents after they have been created. We will pass the list to the
    # mongodb bulk insert feature.
    signals = []

    # Here we itterate through all the rows in the dataframe.
    # For each row in the dataframe the following code is executed.
    for index,row in df.iterrows():

        # Convert the datetime to a valid format by removing the timezone info.
        # We use the value at the 3rd index of the list of columns which was
        # passed as parameter in this function.
        time = datetime.fromtimestamp(row[columns[3]]/1000)

        # Here we create the geometry document.
        # We use the value at the 1st index of the list of columns which was passed
        # as parameter in this function. This index contains the value of the colum
        # representing the longitude (which was defined by the user input in the
        # function:"selected_columns()"). We do the same for the latitude column
        # name at index 0 of the Selected columns list and the altitude column
        # name at index 2 of the selected columns list.
        geometry = TrailModel.Geometry(coord = [row[columns[1]],row[columns[0]]],
                           alt = row[columns[2]])
```

```
        geometry = TrailModel.Geometry(coord = [row[columns[1]],row[columns[0]]],
                                       alt = row[columns[2]])

        # Here we create a signal document in which we pass the required values.
        signal = TrailModel.Signal(time = time,
                                   geometry = geometry,
                                   trail = trail)

        # Here we append the created document to the signals list.
        signals.append(signal)

    # Bulk insert, the populated signals list, in the database.
    TrailModel.Signal.objects.insert(signals,load_bulk=True)

    # Print if the insert process is succesfull.
    print("Inserted " + str(len(df.index))+" trackpoints from dataset: " + str(name))

    # Print when starting the indexing process.
    print("Creating indexes on database ")

    # Create and index on the trail field in the Signal documents.
    TrailModel.Signal.create_index(("trail"))

    # Create and index on the time field in the Signal documents.
    TrailModel.Signal.create_index(("time"))

    # Create and index on the coordinates field in the Signal documents.
    TrailModel.Signal.create_index(("geometry.coord"))

    # Here we make sure the tool is restarted after the dataset is imported
    import_tool()
```

The last thing you need to do is to add one line of code for the function call of the import_tool() function at the bottom of the script to ensure the function runs when the Python script is run:

```
import_tool()
```

Finally create a desktop shortcut for convenience to run the Python script when it's clicked. You do this by performing the following steps:

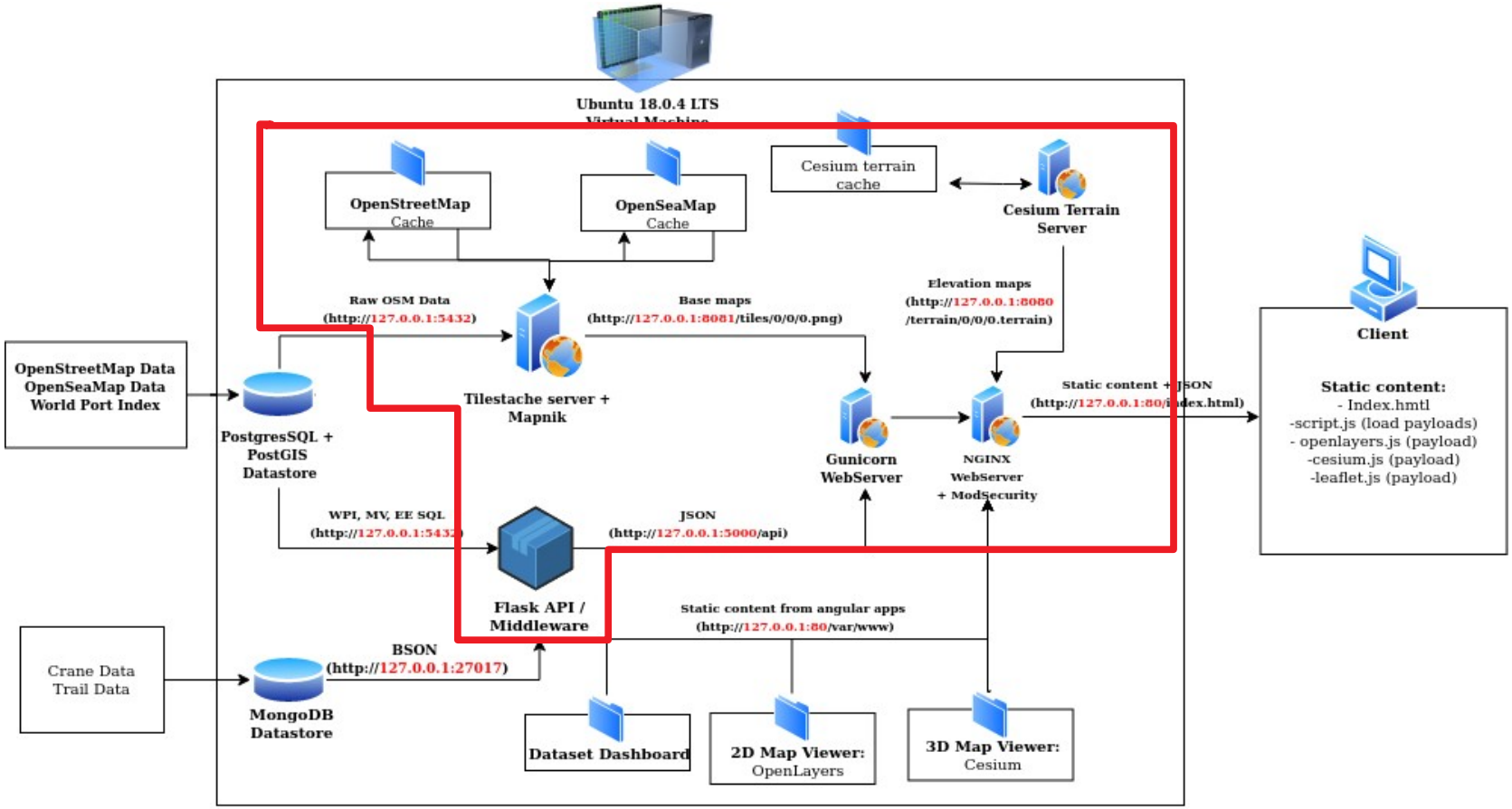
1) Create a desktop shortcut file to start the Python script, using the following command:
   touch ~/Desktop/Import-Mongo-Data.desktop
2) Add the following code to the file that is created on the desktop and save the file:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c 'python3 ~/Geostack/import-utilities/mongo-data-import.py'
Icon=gnome-panel-launcher
Name[en_US]=Import-Mongo-Data
```

3) Make sure the shortcut has execute permission ('+x') by running the following command:
   for i in ~/Desktop/*.desktop; do gio set "$i" "metadata::trusted" yes ;done && sudo \
   chmod +x ~/Desktop/Import-Mongo-Data.desktop

5.7 Installing the middleware software

The Middleware consists of multiple web servers and an API. It contains the TileStache tile server, the Cesium Terrain Server, the Flask API and the NGINX web server. The middleware is where all the “magic” happens for the web apps. During this section you are going to install everything you need to create the middleware of the GeoStack.



5.7.1 Installing the NGINX Web server and Python Flask

5.7.1.1 Dockerizing NGINX and Flask

Before you start installing the software required for the middleware of our GeoStack, you should first carefully work through the cookbook "A secure NGINX web server with ModSecurity" to learn more about the basics of Docker, how to create separate Docker containers for NGINX and Flask, how to run these containers and how to let containers communicate with each other.

IMPORTANT NOTE: this is an important cookbook for your basic skills in Docker and Flask!
In this NGINX cookbook you will first learn how to create the base Python Flask application and then setup the NGINX web server in separate Docker containers.

In this cookbook you will also learn about the following and much more:

- ➔ How to work with Docker containers. (**TIP**: there's a nice Cheat Sheet in section 7.2.2!)
- ➔ Creating a basic Flask application and Dockerize it. (See the IMPORTANT NOTE below!)
- ➔ Creating the secure NGINX web server with ModSecurity and Dockerize it.
- ➔ Running both the Docker containers separately for NGINX and Flask.
- ➔ Linking and running the Docker containers together using Docker-Compose to learn the basics about running a multi-container application in Docker.

After you have finished reading the cookbook: "A secure NGINX web server with ModSecurity" you should come back to this cookbook 'Creating the GeoStack Course VM' to continue here!

IMPORTANT NOTE: more on Dockerizing Flask follows later in chapter 10 'Dockerizing the Flask API' of the programming manual 'Creating a Python – Flask Web Application' about coding the API for the Flask web app to provide a micro web service and also run it in a Docker container.

5.7.1.2 Installing NGINX and Flask in the GeoStack

Now that you have finished the cookbook "A secure NGINX web server with ModSecurity" you end up with a folder structure which contains the files for the base of the Flask API and the NGINX web server as shown in the image on the right.

You need to make some transformations to ensure these files serve the needs of the GeoStack. First copy the nginx-modesecurity folder to the Geostack root folder in which all the other components are.

Press Ctrl + Alt + T to open a terminal and then enter the command:

`cp -r {path to nginx-modsecurity folder} ~/Geostack/`

In your case it would be:

`cp -r ~/20200612_Secure_Webserver_NGINX/nginx-modescurity ~/Geostack/`

Now rename of the file: "nginx.conf" to "nginx-docker.conf" because you want to keep this GeoStack Docker container configuration. Do this by running the following command:

`mv ~/Geostack/nginx-modsecurity/nginx.conf ~/Geostack/nginx-modsecurity/nginx-docker.conf`

Now change the content of the nginx-docker.conf file to the following:

```
#  Load the modsecurity connector module.
load_module modules/ngx_http_modsecurity_module.so;

# Set the user to root
user  root;

# Set the amount of processes
# A worker process is a single-threaded process.
# If Nginx is doing CPU-intensive work such as SSL or
# gzipping and you have 2 or more CPUs/cores
# then you may set worker_processes to be equal to the
# number of CPUs or cores.
worker_processes  1;

events {
   # The worker_connections and worker_processes from the main section
   # allows you to calculate max clients you can handle:
   # max clients = worker_processes * worker_connection
   worker_connections  1024;
}

http {
   # Toggle Modsecurity on
   modsecurity on;

   # Set the location of the modsecurity rules configuration file.
   modsecurity_rules_file /etc/modsecurity.d/include.conf;

   server {
      # Here we define the port on which the NGINX webserver has to listen for incoming traffic.
      Listen      80;
      # Here we set the location of the landing page which is the default NGINX  index.html.
      location / {
         root   /usr/share/nginx/html;
         index  index.html index.htm;
      }
   }
}
```

Save the file and then add the NGINX web server to the GeoStack docker-compose.yml file which is located in the GeoStack root folder.
Open this file and add the following service below the mongodb-datastore service:

```
# Below we define the service for the Middleware components in Geostack
 nginx-webserver:
  # Below we define the name of the NGINX docker container.
  container_name: nginx-webserver
  # Set the directory in which the docker file is located.
  build: ./nginx-modsecurity
  # The line below makes sure the NGINX container restarts when stopping
  # accidentally.
  restart: always
   # Set the port on which the docker container is available to port 80
   # Since we set it to port 80:80  the docker container will also be accessible
   # on our host system via localhost:80 or just localhost
  ports:
    - "80:80"
  # Below we define the services on which the NGINX webserver depends.
  # These services have not been defined yet at this point.
  depends_on:
    - flask-api
    - tilestache-server
    - cesium-terrain-server
```

As you can see you have defined services on which the NGINX web server depends (using the depends_on entry). These services have not been created yet. You are going to create these later on in the course. Because of this the Docker-compose up and docker-compose build commands will not yet work. First install NGINX with ModSecurity locally in the next section before you are creating the Flask API, TileStache tile server, the Cesium Terrain Server and their Docker services.

NOTE: when running the local version of the NGINX web server, the Dockerized version of the NGINX web server will not work directly. This is because both the Local and the Docker version of NGINX will be running on port 80.
To make it easy to stop the local version of NGINX before you start the Dockerized version you will create a desktop shortcut in the next chapter.

Now it's time to install the Local instance of the NGINX web server, ModSecurity and the ModSecurity Core Rule set. How this is done is shown in the next section.

5.7.1.3 Installing the NGINX Web server with ModSecurity Locally

Installing NGINX locally is done by performing the following steps:

1) Download and add the official NGINX signing key.

   `wget -qO - wget http://nginx.org/keys/nginx_signing.key | sudo apt-key add -`

2) Add the official NGINX repository to the system's repository list.

   `echo 'deb [arch=amd64] http://nginx.org/packages/mainline/ubuntu/ bionic nginx' |\ sudo tee --append /etc/apt/sources.list.d/nginx.list`

   `echo 'deb-src http://nginx.org/packages/mainline/ubuntu/ bionic nginx'| sudo tee --\ append /etc/apt/sources.list.d/nginx.list`

3) Update the packages database by running the following command: `sudo apt update`

4) Remove any old NGINX installation by running the following command:

   `sudo apt remove nginx nginx-common nginx-full nginx-core`

5) Install the latest NGINX package by running the following command:`sudo apt install nginx`

6) Validate if NGINX is correctly installed by running the following command: `sudo nginx -t`

   The output of this command should return that everything is OK.

7) Create a desktop shortcut which is going to be used to start the NGINX service. Do this by running the following command: `touch ~/Desktop/Start-NGINX.desktop`

8) Add the following code to the file that is created on the desktop and save the file.

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "service nginx start"
Icon=gnome-panel-launcher
Name[en_US]=Start-NGINX
```

9) Create a desktop shortcut which is going to be used to stop the NGINX service.

   `touch ~/Desktop/Stop-NGINX.desktop`

10) Add the following code to the file that is created on the desktop and save it afterwards

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "service nginx stop"
Icon=gnome-panel-launcher
Name[en_US]=Stop-NGINX
```

11) Make sure the shortcuts are trusted by running the following command:

    `for i in ~/Desktop/*.desktop; do  gio set "$i" "metadata::trusted" yes ;done`

12) Make sure the shortcuts are launch-able by running the following command:

    `sudo chmod +x ~/Desktop/Stop-NGINX.desktop && sudo chmod +x ~/Desktop/Start-\ NGINX.desktop`

Installing ModSecurity for NGINX is done by performing the following steps:

1) Install the required packages for ModSecurity using the following command:

   sudo apt-get install -y apt-utils autoconf automake build-essential git \
   libcurl4-openssl-dev libgeoip-dev liblmdb-dev libpcre++-dev libtool libxml2-dev \
   libyajl-dev pkgconf wget zlib1g-dev

2) Login as superuser using the following command: sudo -i

3) Enter directory: "/opt/", clone the ModSecurity Github repository and enter the cloned directory called: "Modsecurity" by using the following command:

   cd /opt/ && sudo git clone https://github.com/SpiderLabs/ModSecurity && cd \
   ModSecurity

4) Checkout the master branch of the repository that was cloned in the step above by using the following command: sudo git checkout v3/master

5) Initialize a new git sub-module using the following command: sudo git submodule init

6) Update the sub-module initialized in the previous step by using the following command:

   sudo git submodule update

7) Run the build script by using the following command: sh build.sh

8) Run the configuration script by using the following command: ./configure

9) Compile the binary's by running the following command: make

10) Install the binary's, compiled in the previous step by using the following command:

    make install

11) Enter the directory: "/opt/" and clone the Modsecurity-NGINX connector repository

    cd /opt/ && git clone --depth 1 https://github.com/SpiderLabs/ModSecurity-nginx.git

12) Find your current NGINX version, this is required for the next steps so write it down somewhere. You can find the NGINX version by running the following command: nginx -v

13) Download the binary's of your NGINX version, where {version} should be the numbers from the output of the command executed above.

    wget http://nginx.org/download/nginx-{You NGINX Version}.tar.gz

14) Extract the binary's which were downloaded in the previous step, then enter the Directory of the folder that is extracted, again where {version} is your NGINX version.

    tar zxvf nginx-{version}.tar.gz && cd nginx-{You NGINX Version}

15) Run the configuration script and add the Modsecurity-NGINX connector modules

    ./configure --with-compat --add-dynamic-module=../ModSecurity-nginx

16) Compile the modules by running the following command: make modules

17) Copy the compiled modules to the modules directory of the NGINX installation

    sudo cp objs/ngx_http_modsecurity_module.so /etc/nginx/modules/

18) Logout of the superuser account by using the following command: exit

Installing ModSecurity CRS (Core Rule set) is done by performing the following steps:

1) Create a new directory in our NGINX directory using the command:

   sudo mkdir /etc/nginx/modsec && cd /etc/nginx/modsec

2) Copy the NGINX ModSecurity unicode mapping (used to process files types) to the ModSecurity folder by running the following command:

   sudo cp /opt/ModSecurity/unicode.mapping /etc/nginx/modsec/unicode.mapping

3) Clone the ModSecurity CRS Github repository by running the following command:

   sudo git clone **https://github.com/SpiderLabs/owasp-modsecurity-crs.git**

4) Rename the example core rule set configuration file, located in the repository, to : "crs-setup.conf" by running the following command:

   sudo mv /etc/nginx/modsec/owasp-modsecurity-crs/crs-setup.conf.example \
   /etc/nginx/modsec/owasp-modsecurity-crs/crs-setup.conf

5) Copy the recommend ModSecurity configuration, located in the cloned ModSecurity repository, to our NGINX ModSecurity folder and rename it to: "modsecurity.conf" by running the following command:

   sudo cp /opt/ModSecurity/modsecurity.conf-recommended \
   /etc/nginx/modsec/modsecurity.conf

6) Open the main ModSecurity configuration file by running the following command:

   sudo nano /etc/nginx/modsec/main.conf

7) Open the main.conf file and add the following lines to that file.

```
# Make sure the ModSecurity Configuration is used.
Include /etc/nginx/modsec/modsecurity.conf
# Make sure the (default) Core Rule Set Configuration in used.
Include /etc/nginx/modsec/owasp-modsecurity-crs/crs-setup.conf
# Pass the list of rules we want to use.  If you want to remove or add
# rules you can do this in the folder  /etc/nginx/modsec/owasp-modsecurity-crs/rules/
Include /etc/nginx/modsec/owasp-modsecurity-crs/rules/*.conf
```

8) Save the file by pressing the Ctrl + S keys on your keyboard.

9) Close the file by pressing the Ctrl + X keys on your keyboard

At this point you have all the software in place to run our LOCAL instance of the NGINX web server with ModSecurity.

The only thing that's missing is an NGINX configuration file for our Local NGINX instance and how to create that configuration for your Local instance is shown on the next page.

Creating the LOCAL NGINX Configuration is done by performing the following steps:

1) Create a new NGINX configuration file which is going to contain the configuration of our local NGINX web server instance by running the following command:

   `nano ~/Geostack/nginx-modsecurity/nginx-local.conf`

2) Add the following to this file:

```
#  Load the modsecurity connector module.
load_module modules/ngx_http_modsecurity_module.so;

# Set the user to root
user  root;

# Set the amount of processes
# A worker process is a single-threaded process.
# If Nginx is doing CPU-intensive work such as SSL or
# gzipping and you have 2 or more CPUs/cores
# then you may set worker_processes to be equal to the
# number of CPUs or cores.
worker_processes  1;

events {
   # The worker_connections and worker_processes from the main section
   # allows you to calculate max clients you can handle:
   # max clients = worker_processes * worker_connection
   worker_connections  1024;
}

http {
   # Toggle Modsecurity on
   modsecurity on;

   # Set the location of the modsecurity rules configuration file.
   modsecurity_rules_file /etc/nginx/modsec/main.conf;

   server {
      # Here we define the port on which the NGINX webserver has to listen for
      # incoming traffic.
      listen      80;
      # Here we set the location of the landing page which is the default index.html.
      location / {
         root   /usr/share/nginx/html;
         index  index.html index.htm;
      }
   }
}
```

3) Save the configuration files by pressing the Ctrl + S keys on your keyboard.

4) Copy the NGINX configuration to the correct folder in the system.
   `sudo cp ~/Geostack/nginx-modsecurity/nginx-local.conf /etc/nginx/nginx.conf`

5) Restart the NGINX service, for the changes to take effect, by running the following command: `sudo service nginx restart`

To validate whether everything is correctly installed, you can perform the following steps:

1) Perform a dot dot slash attack in your browser, to check if the attack will be logged, by navigating to the following URL in your browser: **http://localhost/?abc=../../**
2) Check the contents of the ModSecurity audit log to see if the attack was logged. Do this by running the following command: cat /var/log/modsec_audit.log
3) Check if a malicious request is logged, it should look like something similar to the output shown in the illustration below:

```
---5J6kekfi---F--
HTTP/1.1 304
Server: nginx/1.17.7
Date: Sat, 11 Jan 2020 15:33:36 GMT
Last-Modified: Tue, 24 Dec 2019 13:07:53 GMT
Connection: keep-alive
ETag: "5e020da9-264"

---5J6kekfi---H--
ModSecurity: Warning. Matched "Operator `Rx' with parameter `(?i)(?:\x5c|(?:%(?:c(?:0%(?:[2aq]f|5c|9
v)|1%(?:[19p]c|8s|af))|2(?:5(?:c(?:0%25af|1%259c)|2f|5c)|%46|f)|(?:(?:f(?:8%8)?0%8|e)0%80%a|bg%q)f|%
3(?:2(?:%(?:%6|4)6|F)|5%%63)|u(?:221[56]|002f|EFC8|F025)|1u|5 (400 characters omitted)' against vari
able `REQUEST_URI_RAW' (Value: `/?abc=../../.' ) [file "/etc/nginx/modsec/owasp-modsecurity-crs/rule
s/REQUEST-930-APPLICATION-ATTACK-LFI.conf"] [line "29"] [id "930100"] [rev ""] [msg "Path Traversal
Attack (/../)"] [data "Matched Data: /../ found within REQUEST_URI_RAW: /?abc=../../."] [severity "2
"] [ver "OWASP_CRS/3.2.0"] [maturity "0"] [accuracy "0"] [tag "application-multi"] [tag "language-mu
lti"] [tag "platform-multi"] [tag "attack-lfi"] [tag "paranoia-level/1"] [tag "OWASP_CRS"] [tag "OWA
SP_CRS/WEB_ATTACK/DIR_TRAVERSAL"] [hostname "127.0.0.1"] [uri "/"] [unique_id "157875681684.116147"]
 [ref "o8,4v4,13"]
```

4) Now you need to make sure ModSecurity does not only log the attacks, but also blocks malicious requests by returning a “49503 Forbidden error”. Do this by opening the ModSecurity configuration file with the following command:

   sudo nano /etc/nginx/modsec/modsecurity.conf

5) Change the line “SecRuleEngine DetectiononOnly” to SecRuleEngine On
6) Restart the NGINX service for the changes to take effect by running the following command: service NGINX restart
7) Perform the dot dot slash attack again by entering following URL in your browser: **http://localhost/?abc=../../**

   It should display the screen shown in the illustration below:

5.7.1.4 Installing Python Flask Locally

Now that you have your Local NGINX web server up and running you want to transform the base Python Flask application in such a way that it's going to serve the needs of the GeoStack.
To do that you first have to install the Flask software locally.

Install Gunicorn3 by running the following command: sudo apt install gunicorn

Install Flask-PyMongo by running the following command : pip3 install flask-pymongo

Install Flask by running the following command: pip3 install Flask

Install BeautifulSoup4 by running the following command: pip3 install bs4

5.7.1.5 Creating and Dockerizing the Flask API for a Micro Web Service

In the cookbook:"A secure NGINX web server with ModSecurity" you have created a basic Flask application and learned how to Dockerize it.

Now you want to extend this base application to serve the needs of the GeoStack by creating a Flask API to provide a micro web service which does the following:

- Connect to our MongoDB datastores and PostgreSQL datastore;
- Perform queries on the datastores;
- Create data profiles using Pandas-Profiling;
- Retrieve the WMS (Web map server) entries from our tile server to be able to switch between map providers in our web applications later on after creating the tile server.

To extend the base Python-Flask application you should now work through the programming manual: "Creating-the-Python-Flask-Webapplication" to learn 1) how to code the API, 2) install the Flask web app Locally and 3) also to learn more about Dockerizing this 'Flask API' web app.

After you have completed the Flask API return to this cookbook and continue reading from here.

NOTE: the functionality of retrieving the tile server entries in the Flask web app will not work since you have not created a tile server yet. How to do this is explained in the next section!

5.7.2 Installing the TileStache Tile server

Now that you have a working Flask API the next step is to create the TileStache tile server. Start by creating a new folder in the GeoStack root folder in which you create the TileStache tile server code and configuration files.
Create the folder by running the following command: mkdir ~/Geostack/tilestache-server

Installing TileStache & Pillow & Gunicorn Python packages by running the following command: pip3 install tilestache pillow gunicorn

Installing Mapnik which is used for rendering tiles using the RAW OSM data in the PostgreSQL database: sudo apt install libmapnik-dev mapnik-utils python3-mapnik

Installing the fonts which are used to generate the names on the OpenStreetMap map.

sudo apt-get install fonts-noto-cjk fonts-noto-hinted fonts-noto-unhinted ttf-unifont

Now you want to create the PostgreSQL Database which is going to contain all your RAW OpenStreetMap data. Do this by performing the steps described below:

1) Login as the PostgreSQL user by using the command: sudo -u postgres -i
2) Create a new PostgreSQL user by using the command: createuser geostack
3) Create a new database and start the PostgreSQL CLI by using the command:
   createdb -E UTF8 -O geostack gis && psql
4) Log in to the database:"gis" by using the command: \c gis
5) Add the PostGIS extensions to the newly created gis database by using the commands:
   CREATE EXTENSION postgis;
   CREATE EXTENSION hstore;
6) Set the permissions of the database to our newly created user by using the commands:
   ALTER TABLE geometry_columns OWNER TO geostack;
   ALTER TABLE spatial_ref_sys OWNER TO geostack;
7) Change the password of the newly created user to 'geostack' by using the command:
   ALTER USER geostack WITH PASSWORD 'geostack';
8) Close the database and logout of the PostgreSQL user using the command: \q
9) Logout of the PostgreSQL user using the command: exit

Installing Openstreetmap-Carto, which provides our OpenStreetMap tile styling, is done by performing the following steps:

1) Enter the TileStache directory by running the following command:
   cd ~/Geostack/tilestache-server
2) Clone the openstreetmap-carto Github repository by running the following command:
   git clone git://github.com/gravitystorm/openstreetmap-carto.git
3) Enter the openstreetmap-carto directory by running the following command:
   cd openstreetmap-carto
4) Install OpenStreetMap-carto by running the following command:
   sudo npm install -g carto

5) Run the get-external-data.py script which will download the base OpenStreetMap map and import it in the PostGIS datastore. The download process can take a while since the script will download 4 shapefiles. These files are as follows:

   ➔ simplified_water_polygons.shp, which has a size of **22.7MB:** This shapefile contains all the waterway polygons of the base map.

   ➔ ne_110m_admin_0_boundary_lines_land.shp, which has a size of **56.3MB** : This shapefile contains the land borders of the base map.

   ➔ water-polygons-split-3857.shp, which has a size of **609MB:** This shapefile contains all the ocean water polygons of the base map.

   ➔ Antarctica-icesheet-polygons-3857.shp, which has a size of **50.8MB:** This shapefile contains all the icesheet polygons of the base map.

   ➔ Antarctica-icesheet-outlines-3857.shp, which has a size of **51.4BM**: This shapefile contains all the icesheet borders of the base map.

   These shapefiles combined will serve as a base map as shown in the illustration below:

   As you can see in the illustration above, this map is empty and does not contain any countries, cities, roads etc. This data will be added later when you are going to import RAW OpenStreetMap data in our PostgreSQL database called: “gis”.

   To execute the script, which downloads the shapefiles required for the base map, open a terminal (Ctrl + Alt + T) and run the following command (Make sure you run this command from the directory “/Geostack/tilestache-server/openstreetmap-carto/”):

```
scripts/get-external-data.py
```

   After the script is done executing the following should be displayed in the terminal:

```
geostack@geostack-VirtualBox:~/Geostack/tilestache-server/openstreetmap-carto$ scripts/get-external-data.py
INFO:root:Checking table simplified_water_polygons
INFO:root:Checking table water_polygons
INFO:root:Checking table icesheet_polygons
INFO:root:Checking table icesheet_outlines
INFO:root:Checking table ne_110m_admin_0_boundary_lines_land
```

Installing OSM2PGSQL, which is used to import the RAW OpenstreetMap data in the PostgreSQL database, by performing the following steps:

1) Enter the tilestache-server directory by running the following command:

   ```
   cd ~/Geostack/tilestache-server
   ```

2) Clone the osm2pgsql repository and enter the osm2pgsql directory

   ```
   git clone git://github.com/openstreetmap/osm2pgsql.git && cd osm2pgsql
   ```

3) Install the required packages for osm2pgsql

   ```
   sudo apt install make cmake g++ libboost-dev libboost-system-dev \
   libboost-filesystem-dev libexpat1-dev zlib1g-dev libbz2-dev libpq-dev \
   libgeos-dev libgeos++-dev libproj-dev lua5.2 liblua5.2-dev
   ```

4) Create a directory called: “build” and enter that directory by running the following command: `mkdir build && cd build`

5) Create and run the code compiler by running the following command: `cmake .. && make`

6) Install the code that was compiled in the previous step by running the following command: `sudo make install`

Install OpenSeaMap Renderer, which is used to generating OpenSeaMap Tiles, by performing the following steps:

1) Add the OpenJDK PPA repository to the repository list by running the following command: `sudo add-apt-repository ppa:openjdk-r/ppa`

2) Update the local package database by running the following command: `sudo apt update`

3) Install openjdk-8-jdk package using the following command:`sudo apt install openjdk-8-jdk`

4) Install the remaining packages by running the following command:

   ```
   sudo apt install libbatik-java ant
   ```

5) Enter the directory: “tilestache-server”, Clone the SeaRenderer repository and enter the searenderer directory by running the following command:

   ```
   cd ~/Geostack/tilestache-server && git clone \
   https://github.com/OpenSeaMap/renderer.git && cd renderer/searender
   ```

6) Compile the code by running the following command: `make -f Makefile.linux`

7) Go to the directory: “jsearch” and build the .jar file by running the following command:

   ```
   cd ../jsearch && ant
   ```

8) Move the newly build .jar file to the work directory by running the following command:

   ```
   sudo mv jsearch.jar ../work/
   ```

9) Enter directory: “jtile” and download the newest version of batik.

cd ../jtile && wget https://www.apache.org/dist/xmlgraphics/batik/binaries/batik-1.7.zip

10) Unzip the newest batik version by running the following command: unzip batik-1.7.zip

11) Open the build.xml file and change the following line:

```
<property name="batik.dir" value="/usr/local/batik-1.7/"/>
```

to the line shown below:

```
<property name="batik.dir" value="./batik-1.7/">
```

12) Build the .jar file by using the following command: ant

13) Move the newly build .jar file to the work directory and Enter the work directory by running the following command: sudo mv jtile.jar ../work/ && cd ../work

14) Create a directory called: “tmp” and a directory called: “tiles” by running the following command: mkdir tmp && mkdir tiles

15) Cleanup by removing unnecessary files and folders using the following command.
Note: this very big command is meant to be written as one (1) line in the terminal!

```
rm ~/Geostack/tilestache-server/renderer/README.md && \
rm -r ~/Geostack/tilestache-server/renderer/jharbour && \
rm -r ~/Geostack/tilestache-server/renderer/jsearch && \
rm -r ~/Geostack/tilestache-server/renderer/jtile && \
rm ~/Geostack/tilestache-server/renderer/work/sync && \
rm ~/Geostack/tilestache-server/renderer/work/upload && \
rm ~/Geostack/tilestache-server/renderer/work/tiler && \
rm ~/Geostack/tilestache-server/renderer/work/getworld
```

16) Create a file called: “world.osm” by running the following command: touch world.osm

17) Open the render script,remove all the code and add the following code:

```
#!/bin/bash
echo "Started rendering of OSM file please be patient..."

# This greps the differences between the OpenSeaMap data which already has
# been rendered and the new OpenSeaMap data. This makes sure no data is
# rendered twice. If the world.osm file is empty the scripts will render al the tiles
diff world.osm next.osm | grep id= | grep -v "<tag" > diffs

# This starts the jsearch java application. This application cuts the next.osm
# file in smaller osm files that each represent a tile.
java -jar jsearch.jar ./

# This renames the next.osm file to world.osm so that when you render a new
# osm file no data will be rendered twice.
mv next.osm world.osm

# if the content of the diffs text file is not empty (this means that new data
# will be rendered), the script will then print:"Found files to render". If the text file is
# empty there is no new data to be added, the script will then print:
#"Found no tiles to render".
if [ -s diffs ]; then
   echo "Found files to render"
else
   echo "Found no files to render"
fi
```

18) Save the render script by pressing Ctrl + S on your keyboard.

19) Open the tilegen script, remove all the code and the following code:

```
#!/bin/bash
while true; do
 # Check whether there is an OSM file in the tmp folder.
 # If this is not the case the generating process is done
 if [ $(ls tmp | grep -c "\.osm") -eq 0 ]; then
  # When done generating the tiles, notify the user and copy the generated files to
  # the openseamap folder in the tilestache cache folder.
  echo "Finished generating Tiles"
  echo "Starting copying of tiles to Tilestache Cache folder located at:
            ~/Geostack/tilestache-server/cache/openseamap-local"
  cp -a tiles/. ../../cache/openseamap-local
  exit
 else
  # For each file in the temporarie folder execute the code below.
  for file in $(ls tmp | grep "\.osm"); do
   echo "Generating SVG's from OSM file: " $file
   # The y , x and z (zoom level) of the tile is assigned to the variable it
   # belongs to.
   tx=$(echo $file | cut -f 1 -d'-')
   ty=$(echo $file | cut -f 2 -d'-')
   z=$(echo $file | cut -f 3 -d'-')
   z=$(echo $z | cut -f 1 -d'.')

   # If the zoom level is equal to 12 execute the code below. This is because
   # openseamap data is not shown below zoom level 12.
   # NOTE: The higher the zoomlevel the more zoomed in you are in the map.
   #       So if the zoomlevel is equal to 18 you are zoomed in far.
   if [ $z = 12 ]; then
    # for every number between 12 and 18 (zoomlevels) execute the code below.
    for k in {12..18}; do
     # The line below assigns the correct OpenSeamap symbol to the correct
     # svg icon
      ../searender/searender ../searender/symbols/symbols.defs $k \
      >tmp/$tx-$ty-$k.svg <tmp/$file
    done;

   # If the zoom level is equal to 12 execute the code below. This is because
   # openseamap data is not shown below zoom level 12.
   else
     # The line below assigns the correct OpenSeamap symbol to the correct
     # svg icon.
     ../searender/searender ../searender/symbols/symbols.defs $z \
      >tmp/$tx-$ty-$z.svg <tmp/$file
   fi
   # The code below is always executed no matter what the zoom level is.
   # The line below makes sure the file is removed from the tmp folder after
   # the tiles have been generated.
   rm tmp/$file

   # the Jtile application is started to generated the png's using the svg's
   # generated by the jsearch application.
   java -jar jtile.jar tmp/ tiles/ $z $tx $ty
  done
 fi
done
```

Before you are going to put the software to use you first need to understand some of the basics of the datasets (OpenStreetMap and OpenSeaMap) which you are going to process using the TileStache tile server.

5.7.2.1 OpenStreetMap explained

Before you are going to use the tools, which you set up in the previous section, it's important to understand the background of the datasets which are going to be processed using these tools.

The raw OpenStreetMap data, which was downloaded in section 3.4.3, was downloaded in the file format .**osm.pbf** ("OpenStreetMap Protocol buffer Binary Format"). The **.osm** file format is specific to OpenStreetMap. You won't come across it elsewhere.

The Protocol buffer Binary Format is intended as an alternative to the XML format which is shown in the illustration below:

```
<?xml version="1.0" encoding="UTF-8"?>
<email>
 <to>The Geostack Project</to>
 <from>Student</from>
 <heading>Reminder</heading>
 <body>Learning a lot aren't you?</body>
</email>
```

An **.osm** file consists of thousands of so called "nodes". In the illustration below you can see a sample representation of an .osm file which contains 2 nodes:

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version='1.0' upload='true' generator='JOSM'>
  <node id='1' action='modify' visible='true' lat='1.43535' lon='0.564646'>
    <tag k='name' v='Lidl'/>
    <tag k='shop' v='Grocery Store'/>
  </node>
  <node id='2' action='modify' visible='true' lat='139.4' lon='0.919'>
    <tag k='name' v='BeijenKorf'/>
    <tag k='shop' v='Clothing store'/>
  </node>
</osm>
```

Each node contains one or more tags which define specifications (e.g. names or types) of the node. In the illustration above you can see a a node which is located on the coordinates (lat='1.43535' lon='0.564646'). This node represents a grocery store called "Lidl". The file also contains a second node (located on the coordinates lat='139.4' lon='0.919') which represents a clothing store called "Beijenkorf".

This RAW OpenStreetMap data will be imported in the PostgreSQL database called: "gis". The TileStache tile server will then generate the map using the RAW data from the datastore.

To visualize any of these nodes on a map you need a style sheet. This style sheet contains the links between the tag types (e.g. a certain color, shape and size).

Let's say that each node with the tag:"shop" should be shaped as an circle, be colored red and have a radius of 2 pixels. After the generation process is finished you will then end up with an empty map with 2 red dots with a size of 2 pixels.

The first dot represents the grocery store: "Lidle" located on the coordinates: lat='1.43535' lon='0.564646' and the second dot represents the clothing store: "BeijenKorf" located on the coordinates: lat='139.4' lon='0.919'.

Below the OpenStreetMap tile generation process is described using a pseudo representation.

Say you download the map extract shown in the illustration below in the file format **osm.pbf**. The map contains 3 main objects which are 2 diners and one shop.

You know that the **.osm** file will contain at least 3 nodes as shown in the illustration below:

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version='1.0' upload='true' generator='JOSM'>
  <node id='1' action='modify' visible='true' lat='1.43535' lon='0.564646'>
    <tag k='name' v='La Cazuela'/>
    <tag k='diner' v='Italian'/>
  </node>
  <node id='2' action='modify' visible='true' lat='139.4' lon='0.919'>
    <tag k='name' v='Dunya Lokanta'/>
    <tag k='diner' v='Turkish'/>
  </node>
  <node id='2' action='modify' visible='true' lat='139.4' lon='0.919'>
    <tag k='name' v='CH Aarnoudse'/>
    <tag k='shop' v='Sleutelmaker'/>
  </node>
</osm>
```

1
2
3

After downloading an .osm.pbf file you want to import the Raw OSM data in your PostgreSQL datastore “gis”. This process is done by using a tool called OSM2PGSQL which is a command-line based program that converts OpenStreetMap data to postGIS-enabled PostgreSQL databases.

Later on you will learn how to use this tool to import newly downloaded data in the PostgreSQL database, but for now it’s useful to understand when this tool comes in play during the process of converting RAW OpenStreetMap data to tiles.

After the import process is done the data will end up looking something similar to the illustration shown on the next page.

NOTE: The data in the illustration below does not represent the data in the illustration above (containing the data in XML format). The data in the illustration below is obtained from the RAW OpenStreetMap data which you are going to import in the next section.

Data Output | Explain | Messages | Notifications

	osm_id bigint	tags hstore
95	2802225848	"addr:city"=>"Maastricht", "addr:street"=>"Titanenhof", "addr:postcode"=>"6215EA"
96	2802222853	"addr:city"=>"Maastricht", "addr:street"=>"Titanenhof", "addr:postcode"=>"6215EA"
97	2802236610	"addr:city"=>"Maastricht", "addr:street"=>"Koninksemstraat", "addr:postcode"=>"6215KA"

The data in the illustration above was obtained by running the following query in the PostgreSQL database: “gis”. This was done using the PostgreSQL database management tool called: “PGAdmin4” .

gis/postgres@localhost

Query Editor | Query History

```
1  SELECT osm_id,tags, way
2      FROM public.planet_osm_point;
```

Now let’s say you zoom in on the location (in your web application which you will built later) as shown in the image below. The TileStache tile server will generate the image using the RAW OpenStreetMap data in our PostgreSQL database: ”gis”.

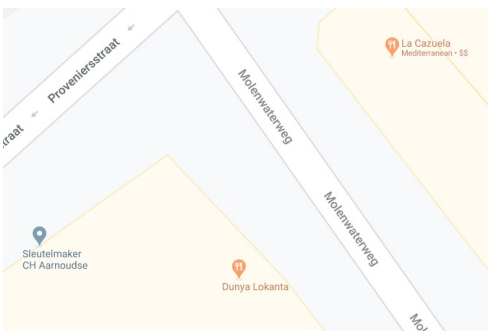
First the TileStache tile server will request a certain amount of data from the PostgreSQL datastore. The data which is requested depends on the part of the map the user currently views.

Then the OpenStreetMap tile generation process will start depending on the data which is being requested by the TileStache tile server.

This generation process is done by linking the nodes and their tags to their corresponding styling which can be found in the OpenStreetMap-Carto. The linking process is done by using a tool called Mapnik.

Combining Mapnik and PostgreSQL is good since PostgreSQL is one of the most efficient and flexible databases Mapnik can use for querying large amounts of data.
For more information on using osm2pgsql to render OpenStreetMap data with Mapnik you should visit the following URL: https://wiki.openstreetmap.org/wiki/Mapnik

Now let’s say that a style defined by OpenStreetMap-Carto says nodes with the tag: “diner” need to be colored orange and the nodes with the tag: “shop” need to be colored light blue. When the generation process is finished you will end up with the map as shown in the illustration below:



5.7.2.2 Downloading and importing OpenStreetMap data in PostgreSQL

Before importing RAW OpenStreetMap data, you have a base map but it's still very empty.
In the illustration on the right you see what the map looks like without any OpenStreetMap data.

This "base" map is generated using the shapefiles downloaded when installing OpenStreetMap-Carto.

First you are going to create a folder which will contain all our downloaded OpenStreetMap data. Do this by running the command:
`mkdir ~/Geostack/tilestache-server/osm-data`

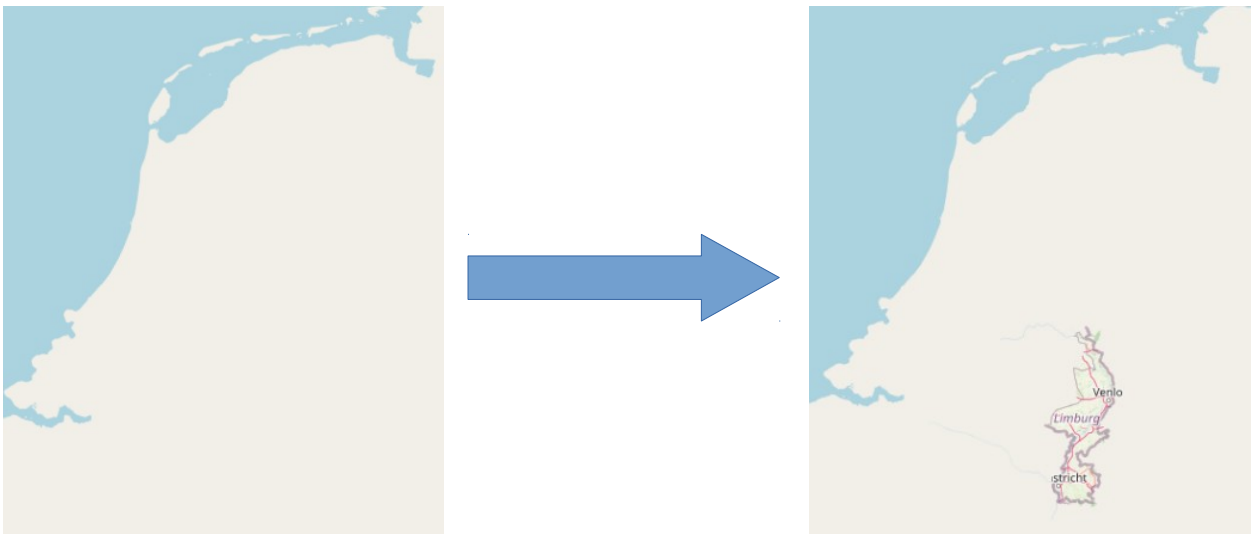
Now you need to download the RAW OpenStreetMap data.
This data will be imported in the "gis" database using the import tool: "osm2pgsql".

There are 3 locations from which you can download the RAW OpenStreetMap data. These locations are as follows:

- ➔ If you want to download the RAW OpenStreetMap data of the whole planet, you should use this website: https://planet.openstreetmap.org/
- ➔ If you want to download pre-selected regions such as a continent, a country or a specific region in a country, you should use this website: http://download.geofabrik.de/
- ➔ If you want to download small sections of the map (e.g. for test purposes) you should use this website: https://extract.bbbike.org/

NOTE: When downloading and importing RAW OpenStreetMap data you have to make sure you download an OSM.PBF (OpenStreetMap Protocolbuffer) file! This is the only file format that will work with the osm2pgsql OpenStreetMap data import tool.

The RAW OpenStreetMap data will make sure that countries, cities, roads etc. will be displayed on the base map which is created using the shapefiles that were downloaded at the beginning of section 5.5.3. In this course you are going to download, import and generate the OpenStreetMap data of the south-eastern Dutch province "Limburg".
The illustration below shows what it will look like before and after the RAW OpenStreetMap data is imported (in the gis database) and is generated by the TileStache tile server.


Venlo
Limburg
stricht

To generate the OpenStreetMap tiles (and achieve the goal shown in the illustration on the previous page) you need to create a style.xml file which contains the rules for the styling of the nodes / tags in the RAW OpenStreetMap data obtained from the PostgreSQL database: “gis” by using queries (which are also defined in the project.mml file using the database connection information). Generating the style.xml file is done by performing the following steps:

1) Open and edit the project.mml file located in the folder: ”~/Geostack/tilestache-server/openstreetmap-carto/” so that it contains the following database login information:

```
osm2pgsql: &osm2pgsql
  # This line defines the type of Geospatial extension added to the database
  type: "postgis"
  # This line defines the name of the database containing the RAW OSM data.
  dbname: "gis"
  # This line defines the server on which the OSM database is running.
  host: "localhost"
  # This line defines the name of the owner of the OSM database.
  user: "geostack"
  # This line defines the password of the OSM database.
  password: "geostack"
  # This line defines any key fields in the database. (Not necessary in our case)
  key_field: ""
  # This line defines the name of the field that contains the geometry of the datarow.
  geometry_field: "way"
  # This line defines the  extend of the map.
  extent: "-20037508,-20037508,20037508,20037508"
```

   The project.mml file contains the connection to our PostgreSQL database and the queries that are executed on the database to retrieve the RAW OpenStreetMap data. The project.mml file will make sure that the style.xml file is updated with the correct database connection information.

2) Create and generate the style.xml file using the database settings defined in our project.mml file by running the following commands:

   ```
   touch ~/Geostack/tilestache-server/openstreetmap-carto/style.xml
   ```

   ```
   carto ~/Geostack/tilestache-server/openstreetmap-carto/project.mml > \
   ~/Geostack/tilestache-server/openstreetmap-carto/style.xml
   ```

   NOTE: If you change database settings (location, name, password etc.) of the PostgreSQL database you need to redo step 1 and 2 to edit the project.mml file according to the new settings and then generate a new style.xml file with the new database settings.

3) Importing the downloaded OSM data in the gis database using osm2pgsql:

   The following command will insert the OpenStreetMap data you downloaded earlier into the database. This step is very disk I/O intensive; importing the full planet might take many hours, days or weeks depending on the hardware. For smaller extracts the import time is much faster accordingly, and you may need to experiment with different -C values to fit within your machine’s available memory.

   ```
   osm2pgsql -d gis -H localhost -P 5432 -U geostack -W --create --slim -G --hstore --tag-transform-script ~/Geostack/tilestache-server/openstreetmap-carto/openstreetmap-carto.lua -C 2500 --number-processes 4 -S ~/Geostack/tilestache-server/openstreetmap-carto/openstreetmap-carto.style {The path to the downloaded osm.pbf file}
   ```

   NOTE: In section 5.5.3.8 is described how you can create a Python script which automates the OpenStreetMap data import process.

Some explanation related to the command above is as follows:

- ➔ The -d flag is used to specify in which database you want to import the OpenStreetMap data (The database called: “gis” in our case).
- ➔ The -H flag is used to specify on which server the database is running (localhost in our case).
- ➔ The -P flag is used to specify on which port the database is running (5432 in this case).
- ➔ The -U flag is used to specify which user is the owner of the database (geostack).
- ➔ The -W flag is used to specify that a password is required to import the OSM data.
- ➔ The --create flag is used to specify that the data is loaded into an empty database rather than trying to append to an existing one.

  NOTE: If do not have any OpenStreetMap data in the database yet, you should add this flag to the command. If you have already imported data in the gis database you should add the flag: “--append” instead of the: “--create” flag which make sure the existing data is not overwritten.

- ➔ The –slim flag is used to specify the table layout which need to be created. The “slim” tables works for rendering. If you plan on updating the database, you should add this flag.
- ➔ The -G flag determines how multi-polygons are processed.
- ➔ The –hstore flag allows tags for which there are no explicit database columns to be used for rendering.
- ➔ The –tag-transform-script flag defines the lua script used for tag processing. This an easy is a way to process OSM tags before the style itself processes them, making the style logic potentially much simpler.
- ➔ The -C flag specifies the amount of memory that is allocated to the osm2pgsql process. If you have less memory you could try a smaller number, and if the import process is killed because it runs out of memory you’ll need to try a smaller number or a smaller OSM extract.
- ➔ The --number-processes flag specifies the amount of CPU cores which are used in the import process. If you have more cores available you set this amount higher.

  NOTE: An osm.pbf file with a size of 33.6MB, takes around 100 seconds to import. The OSM data of the Netherlands is around 1GB when downloaded and the file will need around 50 minutes to be processed and then it will take up 25GB of disk space!

4) Then enter the password: “geostack” when prompted.

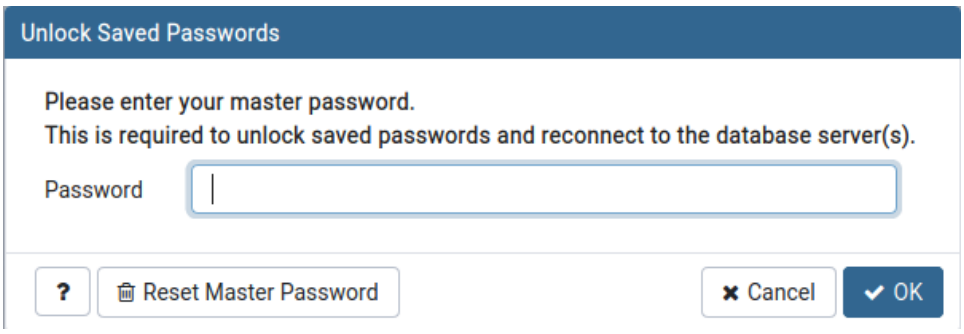
NOTE: After each import you should clear the folder containing the local OpenStreetMap cache. If you do not do this the newly generated files will not be shown because the TileStache server thinks they already exist because of the base OpenStreetMap tiles which were created using the OpenStreetMap-Carto shapefiles. Removing the existing cache can be done by opening a terminal (with Ctrl + Alt + T) and running the following command:

```
rm -r ~/Geostack/tilestache-server/cache/openstreetmap-local/*
```

Now to confirm you imported the data correctly you can perform the following steps:

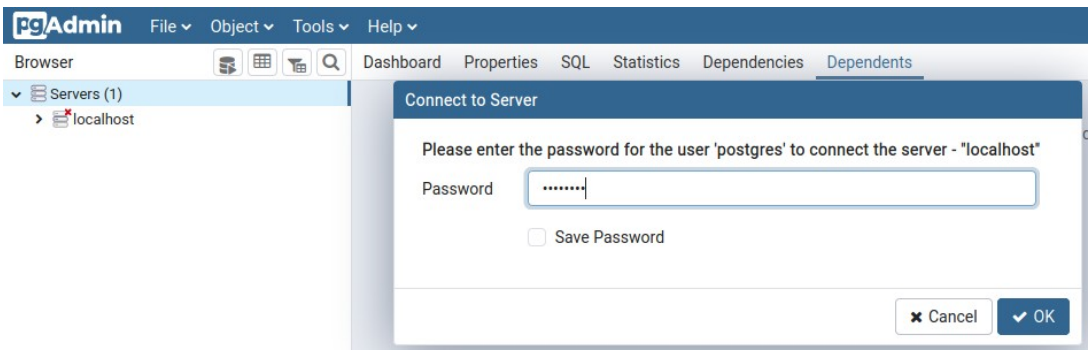
1) Open the PostgreSQL database management tool by clicking on the shortcut shown in the illustration below:

2) When the prompt, shown in the illustration below, pops-up you should enter the password: “geostack”:

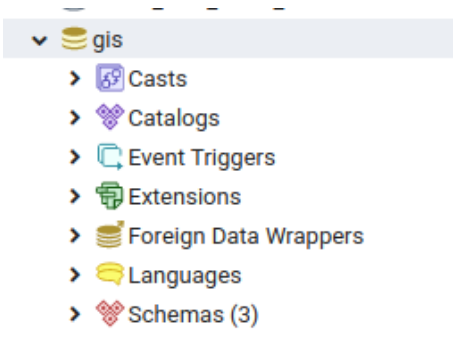


3) Now click on the “Servers” entry in the left sidebar of the screen that pops-up and enter the password: “geostack”as shown in the illustration below:

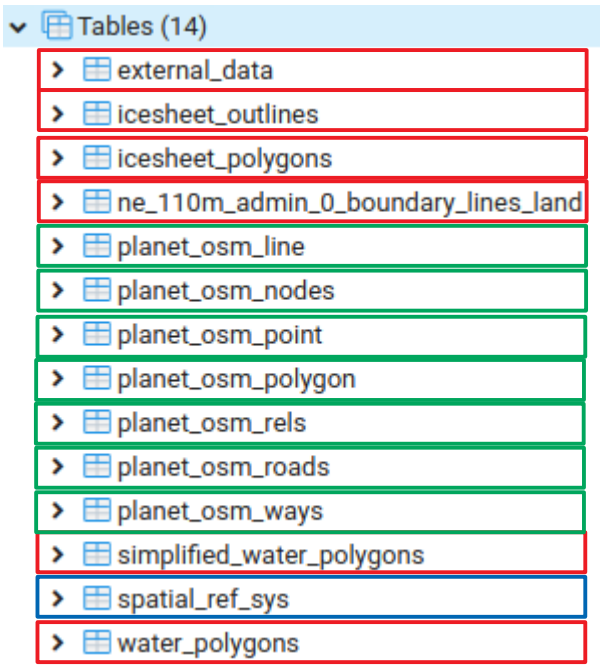
NOTE: If you do not have the localhost server, it means that you have not created it yet. This was done in section 5.4.1.5. So you should go back to that section and perform the steps described in that section.



4) Once connected to the localhost server you should click on the databases entry and then on the name of the ‘gis’ database as shown in the illustration below:



5) Now click on Schemas → public → Tables and you will be greeted with a list of all the tables in the gis database as shown in the illustration below.



Some explanation about the list of database tables is as follows:

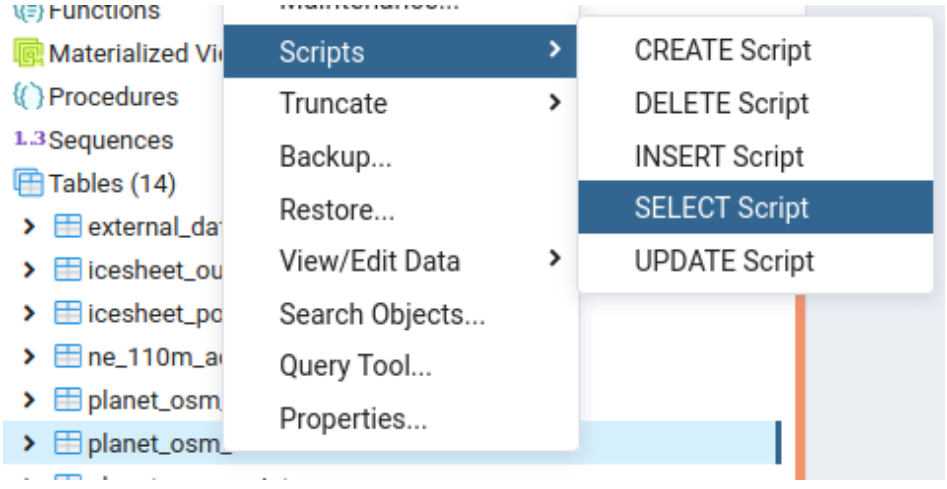
The table 'spatial_ref_sys' framed in Blue contains the spatial reference data which is required by the PostGIS and PostGIS topology extensions, installed in the "gis" database.

The 6 tables framed in Red contain the data of the "base map" (the empty map without any OpenStreetMap data which can be found at the beginning of section 5.5.3.2) which was obtained in step 5 of section 5.5.3 when obtaining the external shapefiles from OpenStreetMap-Carto.

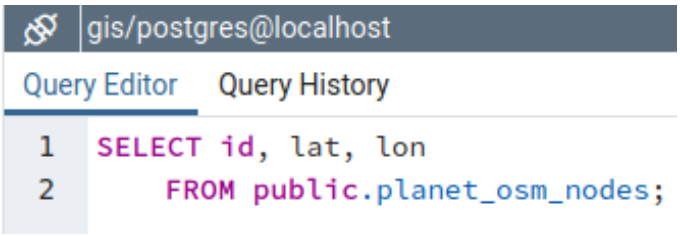
The 7 tables framed in Green contain the OpenStreetMap data which you imported at the beginning of this section.

6) The table called: "planet_osm_nodes" can be seen as the main table in the database. This table contains three columns.

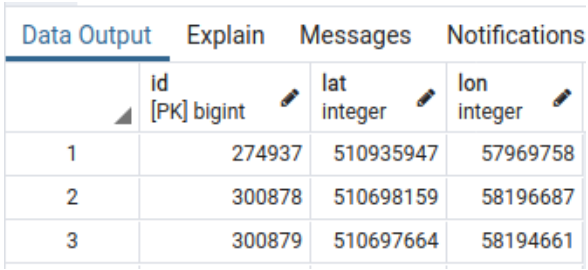
Let's see these columns by right clicking on the table name then go to scripts and the select: 'SELECT Script' as shown in the illustration below:



7) The query which will be created is shown in the illustration below:



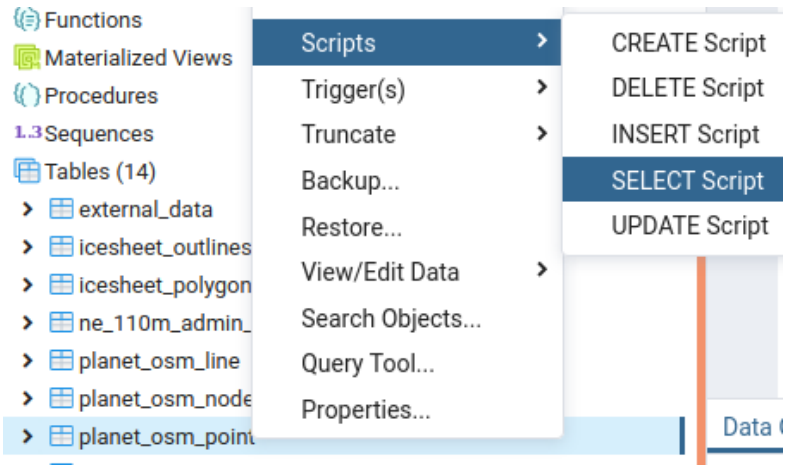
And the result of the query is as follows (only 4 rows are shown in the illustration below):



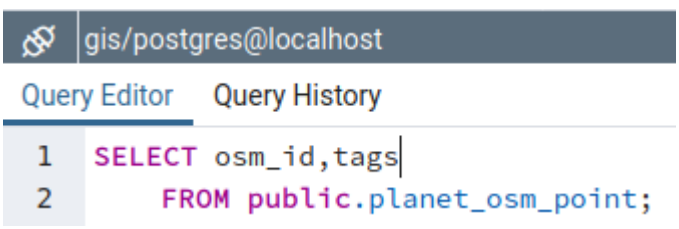
	id [PK] bigint	lat integer	lon integer
1	274937	510935947	57969758
2	300878	510698159	58196687
3	300879	510697664	58194661

The planet_osm_nodes table can be seen as the main table which specifies the location on which the nodes have to appear on the map. The remaining information related to the nodes (so the tag's) can be found in the remaining tables in the "gis" database.

8) For example let's perform a select query on the table: "planet_osm_point" by right-clicking on the table name then go to scripts and the select: 'SELECT Script' as shown in the illustration below:



9) Then remove all the select statements except for osm_id and tags so that the query looks the same as shown in the illustration below:



10) The result of the query will look similar to the one shown in the illustration below:

Data Output | Explain | Messages | Notifications

	osm_id bigint	tags hstore
95	2802225848	"addr:city"=>"Maastricht", "addr:street"=>"Titanenhof", "addr:pos...
96	2802222853	"addr:city"=>"Maastricht", "addr:street"=>"Titanenhof", "addr:pos...
97	2802236610	"addr:city"=>"Maastricht", "addr:street"=>"Koninksemstraat", "ad...

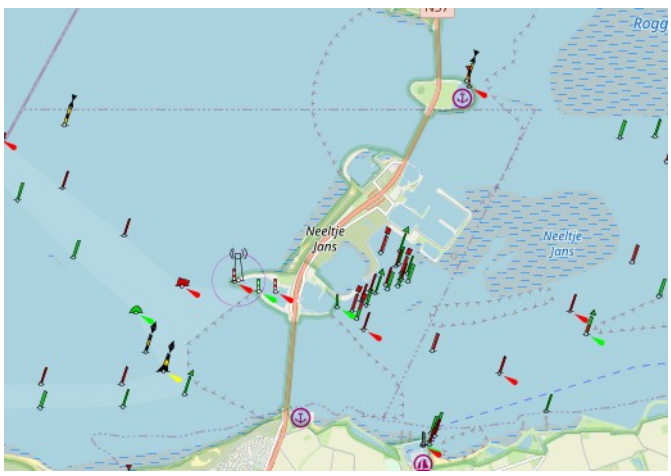
The osm_id in the table: “planet_osm_point” is the foreign key to the id in the table: osm_planet_nodes“. So the data of these two tables combined specify what type of object (obtained from the table: “osm_planet_points” should be displayed where on the map (obtained from the table: “osm_planet_nodes”) during the map generation process.

5.7.2.3 OpenSeaMap explained

Now that you know how to download, process and generate OpenStreetMap data, it's time to learn how to do this with OpenSeaMap data. First of all; OpenStreetMap data is basically the same as OpenStreetMap data. This is because the OpenSeaMap data is also contained in the OpenStreetMap data.

OpenSeaMap data is used to display nautical data on the map. This nautical data contains locations of lighthouses, buoys, sailing routes etc.

A quick view of what the nautical data looks like on an OpenStreetMap base map is shown in the illustration below:



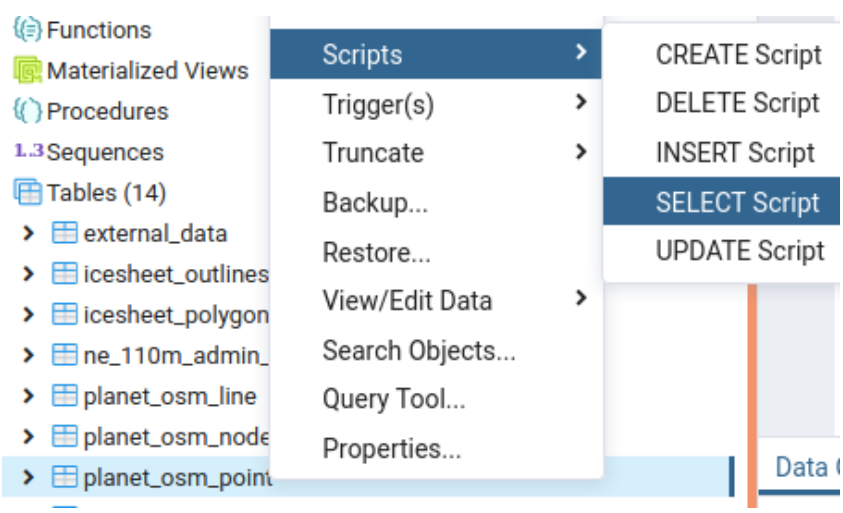
For a full view of what nautical data looks like on the map you can visit the following URL:

https://map.openseamap.org/

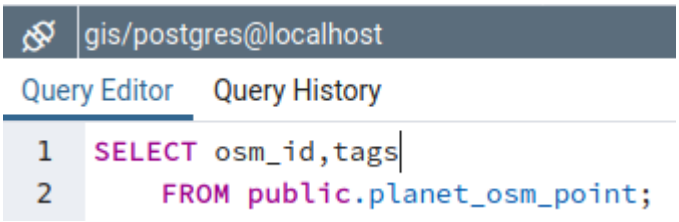
You may have noticed in the previous section that the OpenStreetMap data also contained tags which contained the type: "Seamark". These tags represent nautical data. The problem is that the OpenStreetMap-Carto style sheet does not contain the styling rules related to the Seamark tags so when generating the OpenStreetMap tiles, objects such as buoys and lighthouses are not generated.

If you have not noticed the tags of the type: "Seamark" then take the next 2 steps to see them:

1) Perform a select query on the table: "planet_osm_point" by right-clicking on the table name then go to scripts and the select: 'SELECT Script' as shown in the illustration below:



2) Then remove all the select statements excepts for osm_id and tags so that the query looks the same as shown in the illustration below:



In the result of the query you will see a lot of tags containing seamarks as shown in the illustration below:

Data Output | Explain | Messages | Notifications

	osm_id bigint	tags hstore
13	190186080	"seamark:type"=>"distance_mark", "seamark:distance_mark:units"=>"kilometres", "s...
14	190186073	"seamark:name"=>"Brug Zutendaal", "seamark:type"=>"bridge", "river:waterway_dist...
15	7236813993	"seamark:type"=>"distance_mark", "seamark:distance_mark:units"=>"kilometres", "s...
16	190186066	"seamark:type"=>"distance_mark", "seamark:distance_mark:units"=>"kilometres", "s...
17	190186054	"seamark:type"=>"distance_mark", "seamark:distance_mark:units"=>"kilometres", "s...
18	7243060483	"seamark:name"=>"Brug Eigenbilzen", "seamark:type"=>"bridge", "river:waterway_di...

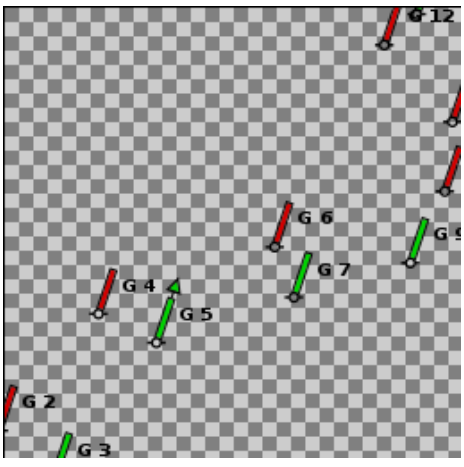
In order to generate the OpenSeaMap tiles you require a tool which basically does the same as Mapnik in combination with OpenStreetMap-Carto but then for the nodes with 'seamark' tags.

The tool you use to create the OpenSeaMap tiles is called: "Searenderer". In section 5.5.3 you already installed and transformed the tool in order to achieve your end-goals.

The goal of this tool is to generate tiles containing the nautical objects (such as buoys, lighthouses and waterways) in the file format PNG with the alpha layer turned on.

Turning on the alpha layer will result in a transparent tile (PNG image) which can be used as an overlay on the OpenStreetMap map.
Using this method you can later toggle the OpenSeaMap layer on or off which is pretty neat!

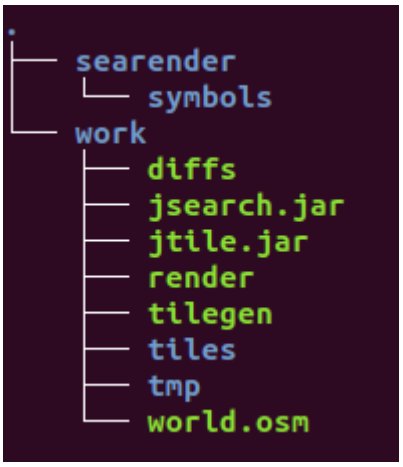
In the illustration below you can see the resulting PNG file of one generated nautical data tile:



5.7.2.4 Downloading and Rendering OpenSeaMap Data

To understand how the process of rendering OpenSeaMap tiles works, it's important to know the file structure of the OpenSeaMap renderer folder.

An illustration of the renderer folder structure is as follows:



A textual representation of the file structure is as follows:

- ➔ The searender folder contains everything related to the styling of the OpenSeaMap tiles:
  - ➢ The symbols folder contains the symbols which need to be linked to nodes in the RAW OSM data (e.g. a buoy symbol or a lighthouse symbol).
- ➔ The work folder contains the scripts and files related to generating the OpenSeaMap tiles:
  - ➢ The tiles folder is going to contain the tiles after they have been generated.
  - ➢ The tmp (temporary) folder is going to contain the files which are generated during the rendering process. These files include the smaller (cut) .osm files and SVG's.
  - ➢ The world.osm file is going to contain all the OSM data that has been rendered before. This file will be compared to the OSM file that contains the data from which you are going to generate the OpenSeaMap Tiles. If there is a difference between these files they will be written to the diffs.txt file.

    NOTE: When you want to render a new part of the world and the new part overlaps with the data that already has been rendered, the data will not be rendered again. If you want to start from scratch you have to clear the contents of the world.osm file (see step 1 on the page below).
  - ➢ The diffs.txt text file is going to contain the differences between the OpenSeaMap tiles that have been generated before and the OpenSeaMap tiles that need to be rendered. If there is **no difference** in the data that has already been generated and the data that needs to be generated, the content of the diffs.txt file will be empty and no rendering will take place.
  - ➢ The jsearch.jar application splits the OSM file (next.osm) that contains the data which needs to be rendered, into smaller OSM files. These smaller OSM files each represent a tile using the X, Y, Z values of a tile.

- The render script checks if there is a difference between the data to add and the data that has already been rendered. This script also executes the jsearch.jar application.

- The tilegen script links the OSM data, that has to be rendered, to the correct symbol using the value of the <tag> data column in a node in the OSM data. An OSM file consists of a lot of nodes. In the illustration below you can find a part of the content of an OSM file. This illustration shows 2 nodes in the dataset.

```
<node id="4161903091" lat="51.6173704" lon="3.6891961" version="1">
        <tag k="source" v="Bing;knowledge"/>
        <tag k="man_made" v="dolphin"/>
        <tag k="seamark:type" v="mooring"/>
        <tag k="seamark:mooring:category" v="dolphin"/>
</node>
<node id="4161903092" lat="51.6174037" lon="3.688885" version="1">
        <tag k="source" v="Bing;knowledge"/>
        <tag k="man_made" v="dolphin"/>
        <tag k="seamark:type" v="mooring"/>
        <tag k="seamark:mooring:category" v="dolphin"/>
</node>
```

  So if a tag value in a node contains the value “buoy" the symbol of the buoy, located in the “searenderer/symbols” folder, will be linked to this node.

- The tilegen script also calls the **jtile.jar** application which generates the actual tiles using the SVG’s which are in turn generated in the **jsearch.jar** application.

- After the tiles are generated, this script will copy the content of the tiles folder to the OpenSeaMap cache subfolder in the TileStache cache folder.

- The file **next.osm** is not included in the file structure after installing the OpenSeaMap renderer. This file is going to represent the OSM data that you want to render.
  - ➔ First you will have to download a new OSM file and then rename it to **next.osm**.
  - ➔ After you have renamed the file, you need to place it in the work folder of the OpenSeaMap renderer.

To generate OpenSeaMap tiles you have to perform the following steps:

1) If you have rendered tiles before and want to reset all the current tiles you should perform the next steps in the renderer/work directory which is located in the tilestache-server folder:
   - ➔ **Clear the contents of the tiles and tmp folders.**
     rm -rfv ~/Geostack/tilestache-server/renderer/work/tiles/* && rm -rfv \
     ~/Geostack/tilestache-server/renderer/work/tmp/*

   - ➔ **Remove all contents of the diffs.txt file.**
     > ~/Geostack/tilestache-server/renderer/work/diffs.txt

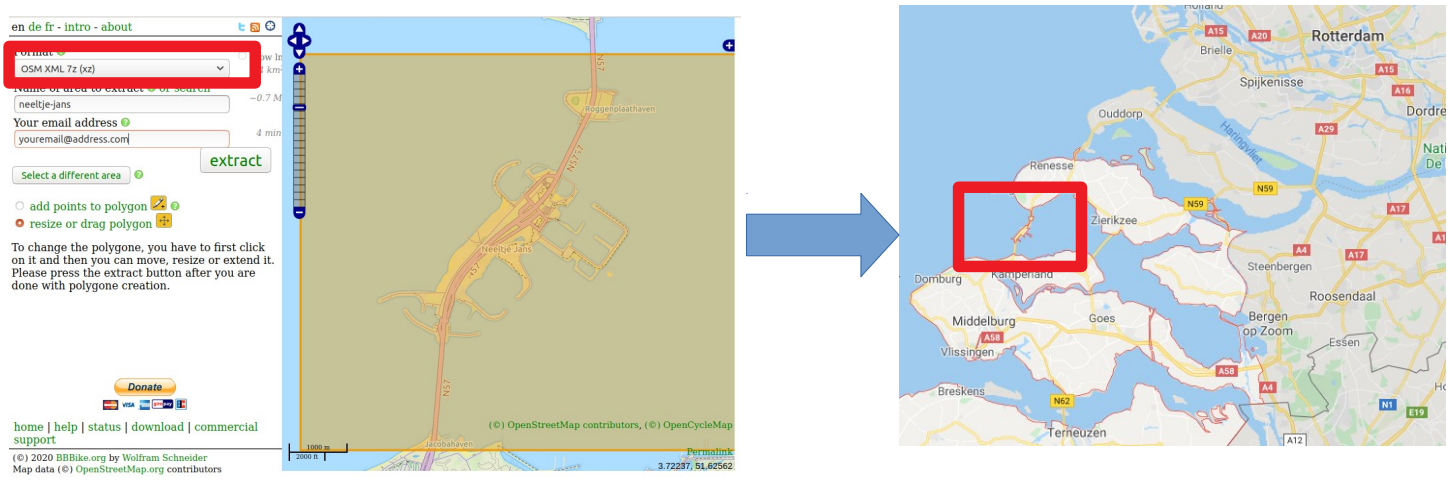
   - ➔ **Remove all contents of the world.osm file.**
     > ~/Geostack/tilestache-server/renderer/work/world.osm

NOTE: the greater sign ‘>’ is a short ‘Linux command’ to empty the files diffs.txt and world.osm. This is called ‘clobbering’ or overwriting a file’ so it is empty and contains zero bytes.

- The command ‘>’ is almost equivalent to the command ‘echo “” > file.ext’ which writes (‘echoes’) an empty string (“”) of 1 character to the file, so the file size is still 1 byte.
- The greater sign itself is called a redirect sign because it redirects the output of a command to another file.
- When used just by itself, the greater sign can redirect nothing to the file because there is no output of any command in front of it which empties the file to a file size of zero bytes.

2) Download a new map extract **IN .OSM** format **NOT OSM.PBF** from one of the sources from which you also downloaded the OSM data (See section 5.5.3.2).

   For testing purposes it's recommend to download a map extract from the website: https://extract.bbbike.org/ since you can select a small portion of a map as shown in the illustration below. The map portion which is downloaded in the illustration below is called: "Neeltje Jans" which is a Dutch delta that was constructed to facilitate the construction of the Oosterscheldedam.



   Also notice the red box in the illustration above. This is where you select the .OSM XML 7Z format in which you want to download the OpenSeaMap data.

   After the download is completed you have to extract the file and copy it to the folder: ~/Geostack/tilestache-server/renderer/work/.

3) Rename the downloaded .osm file in the folder: " ~/Geostack/tilestache-server/renderer/work/" to next.osm.

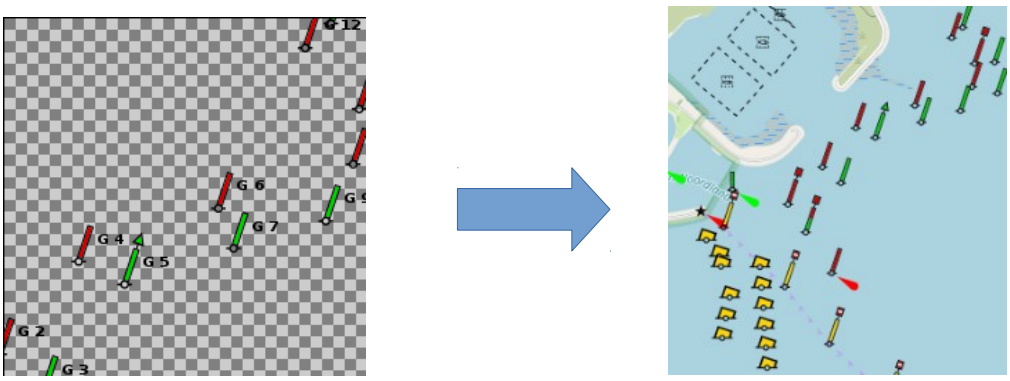
4) Enter the OpenSeaMap renderer folder by running the following command:

   `cd ~/Geostack/tilestache-server/renderer/work/`

5) Run the render and tilegen scripts by entering the command: `./render && ./tilegen`
   The process will look the same as shown in the illustration below.
   NOTE: the OpenSeaMap Tile rendering process can take a long time depending on the size of the OSM file from which the tiles need to be rendered!

```
geostack@geostack-system:~/Geostack/tilestache-server/renderer/work$ ./render &&
 ./tilegen
Started rendering of OSM file please be patient...
Found files to render
Generating SVG's from OSM file:  1044-680-11.osm
Generating SVG's from OSM file:  2088-1359-12.osm
Generating SVG's from OSM file:  2088-1360-12.osm
Generating SVG's from OSM file:  2088-1361-12.osm
Generating SVG's from OSM file:  2089-1359-12.osm
Generating SVG's from OSM file:  2089-1360-12.osm
Generating SVG's from OSM file:  2089-1361-12.osm
Generating SVG's from OSM file:  2090-1359-12.osm
Generating SVG's from OSM file:  2090-1360-12.osm
Generating SVG's from OSM file:  2090-1361-12.osm
Generating SVG's from OSM file:  261-170-9.osm
Generating SVG's from OSM file:  522-340-10.osm
Finished generating Tiles
Starting copying of tiles to Tilestache Cache folder located at:
               ~/Geostack/tilestache-server/cache/openseamap-local
```

After the scripts are finished rendering you will see the generated OpenSeaMap tiles in the folder: “~/Geostack/tilestache-server/cache/openseamap-local”. In the illustrations below you can see an example of one of the generated OpenSeaMap Tiles (PNG) and how it will look in the 2D Map Viewer as overlay on the OpenStreetMap map.



After setting up the TileStache tile server (which you will learn how to do that in the next chapter) the OpenSeaMap tiles will become available via the following URL: http://localhost:8081/openseamap-local/ or if the tile server is running behind the NGINX web server via the URL: http://localhost/tiles/openseamap-local.

NOTE: In section 5.5.3.8 is described how you can create a Python script which automates the generation process of the OpenSeaMap map tiles.

5.7.2.5 Creating the TileStache configuration

1) Create a file called tilestache-configuration.cfg in the tilestache-server directory.

   touch ~/Geostack/tilestache-server/tilestache-configuration.cfg

2) Open the newly created file and add the following code to that file:

```
{
  "index":"entries.html",
  "cache":{
    "name":"Disk",
    "path":"cache",
    "umask":"0000"
  },
  "layers":{
    "openstreetmap-local":{
      "provider":{
        "name":"mapnik",
        "mapfile":"openstreetmap-carto/style.xml"
      },
      "projection":"spherical mercator"
    },
    "openstreetmap-web":{
      "allowed origin":"*",
      "provider":{
        "name":"proxy",
        "url":"http://tile.openstreetmap.de/{Z}/{X}/{Y}.png"
      }
    },
    "openseamap-local":{
      "allowed origin":"*",
      "provider":{
        "name":"proxy",
        "url": "file:////home/geostack/Geostack/tilestache-server/cache/openseamap-local/{Z}/{X}/{Y}.png"
      }
    },
    "openseamap-web":{
      "allowed origin":"*",
      "provider":{
        "name":"proxy",
        "url": "http://tiles.openseamap.org/seamark/{Z}/{X}/{Y}.png"
      }
    },
    "landscape-map":{
      "allowed origin":"*",
      "provider":{
        "name":"proxy",
        "url":"https://b.tile.thunderforest.com/landscape/{Z}/{X}/{Y}.png?apikey=YourAPIKey"
      }
    }
  }
}
```

It's not possible to add any comments to this file since it will throw error's when adding comments to a configuration file.

The configuration file above contains 5 different TileStache entries which are as follows: 1) a local OpenStreetMap version which uses the RAW OSM data in the PostgreSQL database, 2) a web version of OpenStreetMap which proxies the WMS from OSM, 3) a local version of OpenSeaMap which uses the generated tiles in the Tilestache cache, 4) a web version of OpenSeaMap which proxies the WMS from OpenSeaMap and 5) a WMS from Thunderforest.

Some explanation related to the TileStache configuration is given below.

The first part of the TileStache configuration is related to the settings of our TileStache tile server map tile caching. These settings are as follows:

```
{
 # Below we define the location and name of the landing page of our Tilestache server.
 # This file is going to contain all the entries between which we want to switch in our Angular applications.
 # In the Flask API we are going to scrape the contents of this.
 "index":"entries.html",
 # Below we define the settings related to the cache
 "cache":{
   # Below we define the name of the cache
   "name":"Disk",
   # Below we define the path location of the cache
   "path":"cache",
  # Below we define the permissions that the items in the cache will have.
  # "0000" means that the files have full permissions
   "umask":"0000"
 },
```

The entry which is used to provide our local OpenStreetMap Tiles is as follows:

```
  # Below we set the name of the Tilestache entry
  "openstreetmap-local":{
     # Below we set the type of provider to Mapnik.
     # Mapnik is a tool for generating Tiles using a stylesheet
     "provider":{
       "name":"mapnik",
       # Here we set the location of the style.xml which contains our database information and
       # the query's related to the styling of the OpenStreetMap Tiles
       "mapfile":"openstreetmap-carto/style.xml"
     },
    # Below we define the projection of the Tiles
    "projection":"spherical mercator"
  },
```

An example of an entry in our TileStache configuration using an external WMS (Web map server) is shown in the illustration below. This entry is related to the Thunderforest WMS,

The following is required to create a **PROXY** layer entry:

- ✔ The name of the TileStache entry. You can decide yourself what you want the name of the entry is going to be, just make sure it's related to the WMS. (Red)
- ✔ The allowed Origin Type (Blue): This is related to the HTML-headers in the HTTP requests. (* means every type of HTTP request is allowed)
- ✔ The provider (Green): This is where you declare the configuration of entry you are adding.
- ✔ The provider name (Purple): Here you declare the entry type which is proxy in this case.
- ✔ The URL (Orange): Since the type of this entry is a proxy (name substitution) for a WMS located on a web link, you need to declare the URL on which the digital topographical map is located.
- ✔ In the URL you also declare the {Z}{X}{Y} notation for the requested tiles. (Yellow)

NOTE: for the Thunderforest WMS to work, you need to register a free account and obtain an API key. This can be done at the following URL: https://www.thunderforest.com/pricing/ **Once you have obtained an API key you should paste it where it say's "YourAPIKey" in the TileStache configuration file.**

3) Create an HTML file in which you will place the layer names of the TileStache entries:

```
touch ~/Geostack/tilestache-server/entries.html
```

4) Add the names of the entries in the TileStache Configuration to the HTML file, by adding following lines:

```
<p>openstreetmap-local</p>
<p>openstreetmap-web</p>
<p>landscape-map</p>
```

The Flask API will scrape this HTML file and return all the items between the <p> tags to the 2D and 3D map viewers (Our Angular applications).

NOTE: If you add more entries to the TileStache configuration file (and you also want them to be accessible in the Angular applications) you should also add it to the entries.html file.

5) To validate whether the configuration is working correctly you run the following command. You set the amount of CPU workers to 4 and set the timeout to 100.
This means that after 100 seconds of working (rendering) the worker will fail en restart.
Set the timeout value higher if you want to render bigger maps.

```
gunicorn3 --workers 4 --timeout 100 -b :8081 \
"TileStache:WSGITileServer('/home/geostack/Geostack/tilestache-server/\
tilestache-configuration.cfg')"
```

6) Create a desktop shortcut which is used to start the tile server service.

```
touch ~/Desktop/Start-Tileserver.desktop
```

7) Add the following code to the file that is created on the desktop and save it afterwards:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=gunicorn3 --workers 4 --timeout 100 -b :8081
"TileStache:WSGITileServer('/home/geostack/Geostack/tilestache-server/tilestache-configuration.cfg')"
Icon=gnome-panel-launcher
Name[en_US]=Start-Tileserver
```

8) Create a desktop shortcut which is used to stop the tile server service.

```
touch ~/Desktop/Stop-Tileserver.desktop
```

9) Add the following code to the file that is created on the desktop:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "fuser -k 8081/tcp"
Icon=gnome-panel-launcher
Name[en_US]=Stop-Tileserver
```

10) Add the tile server to our local NGINX web server configuration by editing the configuration file of our **LOCAL** NGINX web server instance.

```
nano ~/Geostack/nginx-modsecurity/nginx-local.conf
```

11) Add the following below the upstream server for the Flask API:

```
# Creating the upstream server for the tilestache server
upstream tilestache-server{
   server localhost:8081;
}
```

12) Add the following below the location of the Flask API:

```
# The location of the entries.html file which is used to scrape the entries in
# our TileStache configuration file.
location /tiles/ {
  proxy_pass http://tilestache-server/;
}

# The location of the local OpenStreetMap TileStache entry.
location /tiles/openstreetmap-local/ {
   proxy_pass http://tilestache-server/openstreetmap-local/;
}

# The location of the web version of the OpenStreetMap TileStache entry.
location /tiles/openstreetmap-web/ {
   proxy_pass http://tilestache-server/openstreetmap-web/;
}

# The location of the local version of the OpenSeaMap TileStache entry.
location /tiles/openseamap-local/{
   proxy_pass http://tilestache-server/openseamap-local/;
}

# The location of the web version of the OpenSeaMap TileStache entry.
location /tiles/openseamap-web/{
   proxy_pass http://tilestache-server/openseamap-web/;
}

# The location of the landscape map TileStache entry.
location /tiles/landscape-map/ {
   proxy_pass http://tilestache-server/landscape-map/;
}
```

6) Save the configuration files by pressing the following keys on your keyboard: Ctrl + S

7) Copy and rename the NGINX configuration to the correct folder in the system.

   sudo cp ~/Geostack/nginx-modsecurity/nginx-local.conf /etc/nginx/nginx.conf

8) Restart the NGINX service for the changes to take effect by running the following command: sudo service nginx restart

   That's it! Now when you navigate to one of the locations specified in the NGINX Configuration (e.g. http://localhost/tiles/openstreetmap-web/0/0/0.png), you should be greeted with a map served by the tile server which is running behind the NGINX web server. **Remember to first start the tile server using the desktop shortcut!**

5.7.2.6 Dockerizing the TileStache Tile server

Dockerizing the TileStache tile server is done by performing the following steps:

1) Create a file called: “Dockerfile” in the folder ~/Geostack/tilestache-tileserver/ by running the following command: `touch ~/Geostack/tilestache-server/Dockerfile`

2) Open this file and add the following to the file and save it:

```
## Use ubuntu 18.04 as base image
FROM ubuntu:18.04

## Install git pip npm NodeJS Mapnik and other required packeges.
RUN apt-get update \
 && DEBIAN_FRONTEND="noninteractive" apt-get install -y -f git nodejs npm mapnik-utils python3-mapnik python3-pip mapnik-utils python3 gdal-bin \
 && rm -rf /var/lib/apt/lists/*

## Create folder structure
RUN mkdir tilestache && mkdir tilestache/cache

# Install the required python modules and libraries
COPY requirements.txt /
RUN pip3 install  -r requirements.txt

# Clone the Openstreetmap carto github repo
RUN cd tilestache && git clone git://github.com/gravitystorm/openstreetmap-carto.git

# Install openstreetmap carto in the container
RUN cd /tilestache/openstreetmap-carto && npm install -g carto

# Copy the project.mml file to the correct location in the container.
COPY project-docker.mml /tilestache/openstreetmap-carto/project.mml

# RUN carto to create our OSM style file used to style the generated Tiles.
RUN carto /tilestache/openstreetmap-carto/project.mml > /tilestache/openstreetmap-carto/style.xml

# Copy tilestache configuration file to the correct location in the container
COPY tilestache-configuration.cfg /tilestache/

# Copy the index.html to the correct location in the container
COPY entries.html /tilestache/
```

3) Copy and rename the project.mml file from the OpenStreetMap-Carto folder to the tilestache-tileserver folder by using the following command:

```
cp ~/Geostack/tilestache-server/openstreetmap-carto/project.mml \
~/Geostack/tilestache-server/project-docker.mml
```

4) Change the following section in the project-docker.mml file:

```
osm2pgsql: &osm2pgsql
  # This line defines the type of Geospatial extension added to the database
  type: "postgis"
  # This line defines the name of the database containing the RAW OSM data.
  dbname: "gis"
  # This line defines the server on which the OSM database is running.
  host: "localhost"
  # This line defines the name of the owner of the OSM database.
  user: "geostack"
  # This line defines the password of the OSM database.
  password: "geostack"
  # This line defines any key fields in the database. (Not necessary in our case)
  key_field: ""
  # This line defines the name of the field that contains the geometry of the datarow.
  geometry_field: "way"
  # This line defines the  extend of the map.
  extent: "-20037508,-20037508,20037508,20037508"
```

To the following:

```
osm2pgsql: &osm2pgsql
  type: "postgis"
  dbname: "gis"
  # We pass the name of the PostgreSQL datastore service, from the docker-compose.yml file, as host.
  host: "postgresql-datastore"
  user: "geostack"
  password: "geostack"
  key_field: ""
  geometry_field: "way"
  extent: "-20037508,-20037508,20037508,20037508"
```

This makes sure our PostgreSQL docker container is used instead of our local PostgreSQL Database instance.

5) Create a requirements.txt file which is going contain the Python packages that need to be installed in the TileStache tile server Docker container with the following command:
`touch ~/Geostack/tilestache-server/requirements.txt`
6) Add the following to the requirements.txt file and save it:

```
gunicorn
TileStache
Pillow
pyyaml
requests
psycopg2-binary
```

Add the following service to the docker-compose.yml below the Flask API service:

```
# Here we define the name of the tilestache-server
 tilestache-server:
   # Here we define the name of the tilestache-server
  container_name: tilestache-server
  # The line below makes sure the Tileserver container restarts when stopping accidentally
  restart: always
   # Here we set the directory in which the Dockerfile is located
  build: ./tilestache-server
  # Here we add the cache as volume so the local cache is shared with the docker cache
  volumes:
    - ./tilestache-server/cache:/tilestache/cache
  # Here we set the port on which the Tileserver will be running
  ports:
    - '8081'
  # Here we define the command that will run when the docker container is started
  command: gunicorn --workers 4 -b :8081 --timeout 100  "TileStache:WSGITileServer('/tilestache/tilestache-
configuration.cfg')"
```

7) Build the new service you just added in the docker-compose.yml file, using the following command: `cd ~/Geostack && docker-compose build tilestache-server`

8) Add the following, below the line where you defined the Flask API upstream server, to the nginx-docker.conf file located in the folder: "~/Geostack/nginx-modsecurity".

```
# Creating the upstream server for the tilestache server
upstream tilestache-server{
   server tilestache-server:8081;
}
```

9) Add the following to the nginx-docker.conf file located in the folder: "~/Geostack/nginx-modsecurity". Do this below the line where you defined the Flask API location.

```
# The location of the entries.html file
location /tiles/ {
  proxy_pass http://tilestache-server/;
}

# The location of the local openstreetmap .
location /tiles/openstreetmap-local/ {
   proxy_pass http://tilestache-server/openstreetmap-local/;
}

# The location of the web version of openstreetmap.
location /tiles/openstreetmap-web/ {
   proxy_pass http://tilestache-server/openstreetmap-web/;
}

# The location of the local openseamap.
location /tiles/openseamap-local/{
   proxy_pass http://tilestache-server/openseamap-local/;
}

# The location of the web version of openseamap
location /tiles/openseamap-web/{
   proxy_pass http://tilestache-server/openseamap-web/;
}

# The location of the landscape map.
location /tiles/landscape-map/ {
   proxy_pass http://tilestache-server/landscape-map/;
}
```

10) Rebuild the nginx-webserver service by running the following command:

`cd ~/Geostack && docker-compose build nginx-webserver`

5.7.2.7 Importing OSM data in the PostgreSQL Docker container

To import RAW OSM data in the 'gis' database running in the PostgreSQL docker container, you have to perform the following steps:

1) Retrieve the Docker IP of the PostgreSQL Docker container by running the command:

   ```
   docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
   postgresql-datastore
   ```

2) Run osm2pgsql using the IP address obtained in the previous step:

   ```
   osm2pgsql -d gis -H {the IP adress} -P 5432 -U postgres -W --create --slim -G --hstore --tag-transform-script ~/Geostack/tilestache-server/openstreetmap-carto/openstreetmap-carto.lua -C 2500 --number-processes 1 -S ~/Geostack/tilestache-server/openstreetmap-carto/openstreetmap-carto.style {path to osm.pbf}
   ```

   Where {The IP address} is the IP address obtained in step 2 and {Path to osm.pbf} is the location where the file osm.pbf is located which you want to import in the gis database.

3) Then enter the password: "geostack" and let the import process finish.

You can also import OSM data in the docker instance of the PostgreSQL database by simply shutting down the local instance of the PostgreSQL database and then make the Dockerized version of the PostgreSQL database available to the LOCALHOST by editing the docker-compose.yml file (see chapter 5.4.1.2).

5.7.2.8 Automating the OSM data import and generation process

Now that you know how to manually import OpenStreetMap data in a PostgreSQL Database and how to manually generate OpenSeaMap tiles, it's time to make a Python script which automates these processes. This script will do the following:

- ➔ Select .osm.pbf and .osm files using a simple GUI;
- ➔ Give the user the option to import OpenStreetMap data in the 'gis' database;
- ➔ Give the user the option to generate OpenSeaMap tiles and place them in the TileStache Cache.

So let's start with creating a Python script called: "osm-data-import.py" in the folder: "~/Geostack/import-utilities/" by running the following command:

```
touch ~/Geostack/import-utilities/osm-data-import.py
```

NOTE: the following source code text cannot be copied and pasted because the code blocks are images. The source code is explained using inline comments. Please, do read and learn! You can find the complete script in the file "osm-data-import-v-course.py" provided in the folder: "Building-the-VM-using-the-installation-scripts/Geostack/import-utilities".

Open the file and add the following module imports at the top of the script:

```
# The os module is used to execute bash commands in the Python script.
import os
# The tkinter module is used to create a GUI.
import tkinter
# The filedialog module is used to show a file selection GUI.
from tkinter import filedialog
```

Next you need to create the main function which is triggered when running the Python script. This function can be seen as the “main” function that let’s you choose between an OpenStreetMap data import or the OpenSeaMap tile renderer.
This function is called: “import_tool()”. Add the code for it below the transform_data function.

NOTE: The following illustration is divided in 2 parts. The last line of the first illustration is the same as the first line of the second illustration. You don’t need to add this line twice.

```
def import_tool():

    # Print the start text in the terminal.
    print('---->>WELCOME TO THE OSM IMPORT TOOL<<----')

    # Create a new instance of a Tkinter GUI.
    GUI = tkinter.Tk()
    # This line makes sure the GUI is instantiated in the background.
    GUI.withdraw()

    # Here we create the options which the user can select to choose between
    # importing OpenStreetMap data or generating OpenSeaMap tiles.
    ostreetm = {'1','ostreetm','openstreetmap'}
    oseam = {'2','oseam','openseamap'}

    # Here we ask the user to choose between 2 options. We assign the result
    # of the user input to a variable called:"u_input"
    u_input = input('What do you want to do?\
     \n[1] Import OpenStreetMap data\
      \n[2] Generate OpenSeaMap Tiles \n')

    # Here we define the code which is executed depending on the user input.
    # If the user input, which is assigned to a variable called: "u_input" is
    # in the list of words assigned to variable:"ostreetm" the following code
    # will be executed.
    if u_input in ostreetm:

        # Here we add the code logic required to open the selection GUI.
        # The following parameters are passed in this code:
        # - parent = the instance of the Tkinter GUI
        # - filetypes = specifies the extensions which are allowed to be selected
        #               since the user chose to import OpenStreetMap data,
        #               we are only allowing osm.pbf files to be selected.
        # - title = the text displayed at the top of the GUI.
        input_file = filedialog.askopenfilename(
        parent=GUI,filetypes=[('OSM.PBF file','*.osm.pbf')],
                              title='Choose an osm.pbf file')
```

```
    parent=GUI,filetypes=[('OSM.PBF file','*.osm.pbf')],
                          title='Choose an osm.pbf file')

    # Here we check if the selected file is not equal to None.
    # If this is the case the following code is executed.
    if input_file != None:

        # Print that the file is valid.
        print("Selected file is valid!")

        # Call the function:"import_osteetm" and pass the selected file
        # as parameter.
        import_osteetm(input_file)

# Here we define the code which is executed depending on the user input.
# If the user input, which is assigned to a variable called: "u_input" is
# in the list of words assigned to variable:"oseam" the following code
# will be executed.
elif u_input in oseam:

    # Here we add the code logic required to open the selection GUI.
    # Here we pass .osm as the filetypes parameter.
    input_file = filedialog.askopenfilename(
    parent=GUI,filetypes=[('.OSM file','*.osm')],title='Choose an osm file')

    # Here we check if the selected file is not equal to None.
    # If this is the case the following code is executed.
    if input_file != None:

        # Print that the file is valid.
        print("Selected file is valid!")

        # Call the function:"import_oseam" and pass the selected file
        # as parameter.
        import_oseam(input_file)
```

As you can see in the illustrations above you used 2 functions (“import_ostreetm()” and “import_oseam()”) which have not been created yet. So now create these functions.
You start of by creating the function which will be triggered when the user chooses to import RAW OpenStreetMap data in the PostgreSQL database.

How this is done is shown on the next page.

```
# Here we create a function called:"import_osteetm".
# This function takes an input file as input parameter.
def import_osteetm(input_file):

    # Print that the import process has started.
    print("Executing the local import command for file: "+ input_file)

    # Here we create the osm2pgsql command. For more info on this command
    # read section 5.5.3.1.
    command = 'osm2pgsql -d gis -H localhost -P 5432 -U geostack -W --create\
     --slim -G --hstore --tag-transform-script\
      ~/Geostack/tilestache-server/openstreetmap-carto/openstreetmap-carto.lua\
       -C 3500 --number-processes 4 -S\
        ~/Geostack/tilestache-server/openstreetmap-carto/openstreetmap-carto.style\
         '+ input_file

    # Here we execute the command which we created above
    os.system(command)
    # Print when done importing
    os.system('echo "done importing the data"')
    # Restart the application
    import_tool()
```

Now let's create the function which is used to generate the OpenSeaMap tiles.
How this is done is shown in the illustration below:

```
# Here we create a function called:"import_oseam".
# This function takes an input file as input parameter.
def import_oseam(input_file):
    print("Running Generation command for file: " + input_file)

    # Create the command to rename the input_file to next.osm and move the
    # file to the /renderer/work directory.
    command = 'cp '+ input_file + '\
     ~/Geostack/tilestache-server/renderer/work/next.osm'

    # Execute the command which we created above.
    os.system(command)

    # Print that the command above was executed succesfully.
    print("Copied and renamed:" + input_file + " to next.osm")

    # Print that the tile rendering process has started.
    print("Executing the rendering scripts. Please be patient!")

    # Create the command that runs the render and tilegen scripts.
    command = 'cd /home/geostack/Geostack/tilestache-server/renderer/work\
     && ./render && ./tilegen'

    # execute the command which we created above.
    os.system(command)
    # Print when done
    os.system('echo "done generating tiles"')
    # Restart the application
    import_tool()
```

The last thing you need to do is adding the code which makes sure the Python script is executed and the import_tool function is triggered.
Do this by adding the following last line at the bottom of the script:

```
import_tool()
```

Now create a desktop shortcut which runs the Python script when it's clicked. Do this by performing the following steps:

1) Create a desktop shortcut to start the Python script, using the following command:

   touch ~/Desktop/Import-OSM-Data.desktop

2) Add the following code to the file that is created on the desktop:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c 'python3 ~/Geostack/import-utilities/osm-data-import.py'
Icon=gnome-panel-launcher
Name[en_US]=Import-OSM-Data
```

3) Make sure the shortcut is trusted by running the following command:

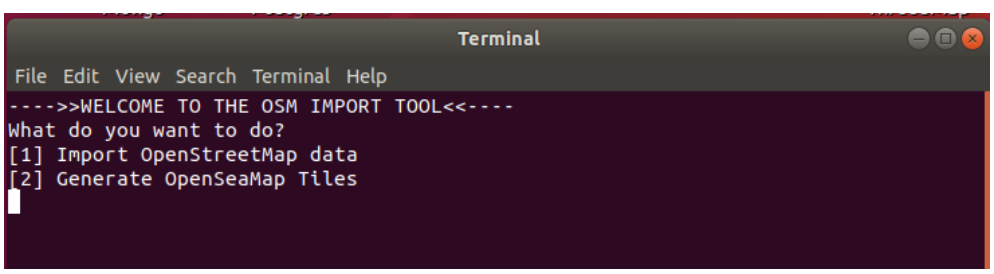
   for i in ~/Desktop/*.desktop; do gio set "$i" "metadata::trusted" yes ;done

4) Make sure the shortcut is launch-able by running the following command:

   sudo chmod +x ~/Desktop/Import-OSM-Data.desktop

Now when you click the desktop icon, shown in the illustration below, you can easily import and render OpenStreetMap and OpenSeaMap tiles.

This will open a terminal as shown in the illustration below:



Now you are able to easily import OpenStreetMap Data and generate OpenSeaMap tiles.

5.7.3 Installing the Cesium Terrain Server

Now that you have most of the middleware components in place it's time to create the Cesium Terrain Server (CTS) which is used to generate terrain files and serve elevation (height) maps to our 3D Map Viewer application. Look at the CTS as the tile server for 3D maps if you will.

NOTE: how to program a web app for Cesium is detailed in the programming manual 'Creating an 3 Dimensional Map Viewer'!

5.7.3.1 Introduction to Terrain files and DEM files

The terrain files for the CTS contain information related to the height of the ground at a given point. By using the terrain files you can visualize local elevation maps in your web application.

A terrain file is created from a DEM file. A DEM (Digital Elevation Model) can be represented as a raster (a grid of squares, also known as a height map when representing elevation) or as a vector-based Triangular Irregular Network (TIN).

DEM files are commonly built using data collected with remote sensing techniques, but they may also be built from land surveying. A DTM, DTED or DEM file resembles an image file, where pixels are mapped to specific geodetic coordinates.

Digital Terrain Models (**DTM**) are often called **Digital Elevation Models** (DEM) and they are a topographic **model** of the bare Earth surface that can be manipulated by computer programs.

Digital Surface Models (**DSM**) are files created using Airborne Light Detection and Ranging (LiDAR) LiDAR uses pulses of light that travel to the **ground**. When the pulse of light bounces off its target and returns to the sensor, it gives the distance to the Earth's surface.

For more information related to DTM (DEM) and DSM files you should visit the following URLs:

- https://gisgeography.com/dem-dsm-dtm-differences/
- https://en.wikipedia.org/wiki/Digital_elevation_model

These data files contain the **elevation** (height) data of the **terrain** in a **digital** format which relates to a rectangular grid of measuring points which is called a point cloud.

DEM files are very large files because they have a huge amount of measuring points! Resolutions vary from the Space Shuttle datasets with 30 meter resolution to the European Union project with 1 – 2 meter resolution to national datasets like the height profile of the Netherlands with a resolution of less than a meter.

- https://www2.jpl.nasa.gov/srtm/ (NASA Shuttle Radar Topography Mission; world wide DEM files)
- https://www.eea.europa.eu/data-and-maps/data/copernicus-land-monitoring-service-eu-dem (European Union; EU DEM files)
- https://www.ahn.nl/ (Country example: AHN is the project for the Dutch DEM files)

The file format for DEM files is most often the GeoTIFF version to store the point cloud in the file container of the TIFF image file format. Read more here: https://en.wikipedia.org/wiki/GeoTIFF
For download these TIFF files (.tif file extension) are often compressed in the ZIP file format.

NOTE: remember DEM files are very large files and converting the content into terrain files with the 3D map tiles will take a lot of computing time too! Think hundreds of Gigabytes er even a Terabyte or more of disk storage, 16 – 32 GB RAM and a fast CPU! Still the conversion requires many minutes and even hours of computing time to generate the terrain files! Start small!!!

5.7.3.2 Downloading DSM or DTM files

To download the DSM or DTM files you should visit the following URL:
https://downloads.pdok.nl/ahn3-downloadpage/

You will then be greeted with a map as shown in the illustration below.

On the map you can see a lot of small rectangular boxes which each represent an area of a topographical map segment for which you can download a DEM file.

- When you zoom in on the map you can select a map segment by clicking on it which will turn the color to purple to indicate it is selected as is shown in the illustration below.

KAARTBLAD: 46GZ1		
INHOUD	FORMAAT	LINK
0,5 meter raster dsm	GeoTIFF (gezipt)	Download
0,5 meter raster dtm	GeoTIFF (gezipt)	Download
5 meter raster dsm	GeoTIFF (gezipt)	Download
5 meter raster dtm	GeoTIFF (gezipt)	Download
Puntenwolk	LAZ	Download

You can choose from the following options:

- ➔ 0.5M Raster DSM which is a very precise (up to 0.5m) dataset which also includes the height data of tree's and houses etc.
- ➔ 0.5M Raster DTM which is a very precise (up to 0.5m) dataset which does **NOT** include the height data of tree's and houses etc. (**NOTE**: these files are used in the GeoStack Course!)
- ➔ 5M Raster DSM which is a is less precise (up to 5m) dataset which also includes the height data of tree's and houses etc.
  5M Raster DTM which is a less precise (up to 5m) dataset which does **NOT** include the height data of tree's and houses etc.
- ➔ A Point Cloud file which is the most precise. You are not going to use this types of datasets in the GeoStack Course because these are very large files!

When you have decided what type of DEM file you want to download, you should click on the Download link next to the file you want to download. (RED in the illustration above).
This will download a ZIP file which will give you a TIF file with elevation data after unzipping.

5.7.3.3 Rendering Digital Terrain Models for Cesium

To render terrain files, used in the 3D Map Viewer, you should perform the following steps:

1) Create a new Cesium server directory and its sub-folders in the Geostack folder:

   ```
   mkdir -p ~/Geostack/cesium-server/data/tilesets/terrain
   ```

2) Place the downloaded .TIF file in the ~/Geostack/cesium-server/data/ directory

3) Transform the projection of the .tiff file to WGS84 by using the GDALWarp Docker container.

   ```
   docker run -v ~/Geostack/cesium-server/data:/data geodata/gdal gdalwarp -t_srs \
   EPSG:3857 /data/M_52EZ2.TIF /data/M_52EZ23857.TIF
   ```

   The command above will first download the GDAL docker image after which it will transform the projection of the downloaded .TIF file. This process takes about 1 minute an should look similar to the illustration below.

   ```
   Creating output file that is 10095P x 12600L.
   Processing input file /data/i09bz1.tif.
   Using internal nodata values (e.g. -3.40282e+38) for image /data/i09bz1.tif.
   Copying nodata values from source /data/i09bz1.tif to destination /data/i09bz13857.tif.
   0...10...20...30...40...50...60...70...80...90...100 - done.
   ```

4) Create a new directory in the folder ~/Geostack/cesium-server/data/tilesets/terrain which is going to contain the rendered .terrain files. It's recommend to give this folder the same name as the .tiff file. Once you have created the folder you need to move the rendered terrain files to this folder.

   For example: the downloaded .tiff file, which is used in this example, has the name: "M_52EZ2" so it's recommended to use the same name for the folder.
   The terrain files will then be available via the URL: http://localhost:8080/tilesets/M_52EZ2 or the URL: http://localhost/terrain/M_52EZ2 in case the Cesium server is running behind the NGINX web server.

   You can create a new directory by running the following command:

   ```
   mkdir -p ~/Geostack/cesium-server/data/tilesets/terrain/M_52EZ2
   ```

5) In the 3D Map viewer you are going to add a layer which uses the terrain data served by the cesium-terrain-server.
   This will be done by using the URL: http://localhost:8080/tilesets/{name of the folder} or the URL: http://localhost/terrain/{name of the folder} in case the Cesium server is running behind the NGINX web server.

6) Navigate to the cesium-server directory by using the following command:

   ```
   cd ~/Geostack/cesium-server
   ```

7) Run the .terrain file generator using the folder created in the previous step as output directory with the following command:

   ```
   docker run -it -v ~/Geostack/cesium-server/data:/data tumgis/ctb-quantized-mesh \
   ctb-tile -f Mesh -C -o /data/tilesets/terrain/M_52EZ2 /data/M_52EZ23857.TIF
   ```

   The command above will first download the quantized-mesh docker container after which it will start the terrain file generation process.
   The generation process of this file (500MB) takes around 10 minutes to complete.

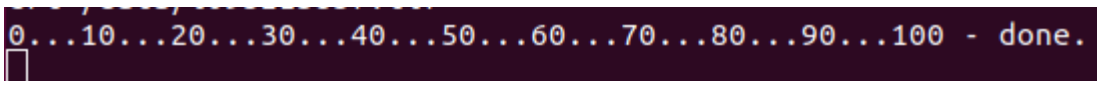
8) Create a TileJSON file which is used for representing map metadata. This file will be called layer.json.

   - The CesiumTerrainProvider Class in Cesium requires that a layer.json resource is present describing the terrain tileset.
   - The ctb-tile utility does not create this file.
   - If a layer.json file is present in the root directory of the tileset then this file will be returned by the server when the client requests it.
   - If the file is not found the server will return a default resource.

   Since you don't want a default resource you are going to create the layer.json file using the following command:

   ```
   docker run -it -v ~/Geostack/cesium-server/data:/data tumgis/ctb-quantized-mesh \
   ctb-tile -l -f Mesh -C -o /data/tilesets/terrain/M_52EZ2 /data/M_52EZ23857.TIF
   ```

   This process takes about 5 seconds to complete and should look similar to the illustration below.



   Now the new terrain files are generated you can run the Cesium Terrain Server and check if the terrain files are accessible by requesting the metadata file: layer.json

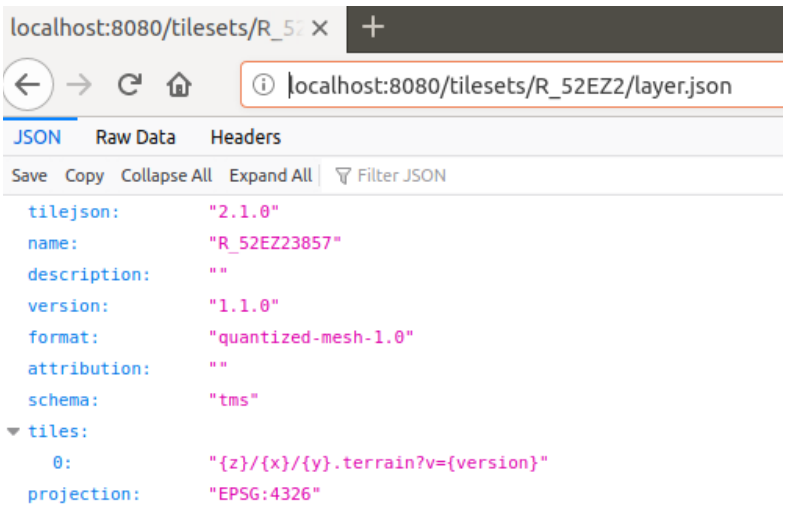
9) Run the Cesium Terrain Server with the following command:

```
cd ~/Geostack/cesium-server && docker run -p 8080:8000 -e 'SERVE_STATIC=0' -v \
/home/geostack/Geostack/cesium-server/data/tilesets/terrain:/data/tilesets/terrain \
geodata/cesium-terrain-server
```

The environment variable: “SERVE_STATIC=0” makes sure that the Cesium application, created by the Cesium Server, is not served by the Cesium Terrain Server. This will speed up the serving process of the .terrain files.

NOTE: The first time you run this command the cesium-terrain-server docker image will be downloaded so the startup process takes about 2 minutes to complete depending on your network speed.

10) Now when you navigate to the URL: http://localhost:8080/tilesets/M_52EZ2/layer.json you will be greeted with the layer metadata of the newly generated terrain files as shown in the illustration below.



NOTE: if you get a permission denied error, you should create the terrain folder again and copy the contents of the old folder to the new terrain folder!

5.7.3.4 Caching the Terrain files with Memcached

To be able to cache the generated .terrain files, you are going to use a software product called Memcached which serves as a temporary in-memory data cache. You are going to use a Docker container for Memcached. This is done by running the following command:

```
docker run --rm --name cesium-memcache -p 11211:11211 memcached -m 200
```

This command will first download the MemCached Docker image after which it creates a new container called: “cesium-memcache” which is available on port 11211. The -m parameter indicates how much memory can be used by the memcache container.

To use Memcached with the Cesium Terrain Server you need to link the memcached container to our Cesium Terrain Server container. This is done by running the following command:

```
cd ~/Geostack/cesium-server && docker run -p 8080:8000 -v \
/home/geostack/Geostack/cesium-server/data/tilesets/terrain:/data/tilesets/terrain \
--link cesium-memcache:memcached geodata/cesium-terrain-server
```

As you can see you used the syntax: “- -link cesium-memcache:memcached ”. This syntax links the running memcached container to your Cesium Terrain Server.

To check whether the memcached container is working you can perform the following steps:

1) Start the memcached Docker container (If it’s not running already) by running the following command:

   ```
   docker run --rm --name cesium-memcache -p 11211:11211 memcached -m 200
   ```

2) Start the Cesium Terrain Server with the memcached docker container by running the following command in a new terminal (Ctrl + Alt + T):

   ```
   cd ~/Geostack/cesium-server && docker run -p 8080:8000 -v \
   /home/geostack/Geostack/cesium-server/data/tilesets/terrain:/data/tilesets/terrain \
   --link cesium-memcache:memcached geodata/cesium-terrain-server
   ```

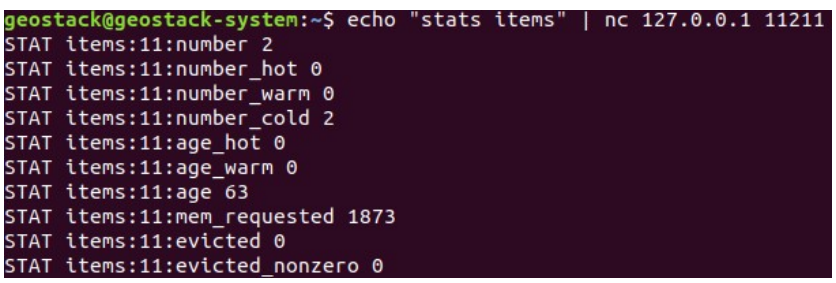
3) Load the Terrain Layer metadata file by navigating to the following URL:

   http://localhost:8080/tilesets/M_52EZ2/layer.json

4) List the current items in the memcached storage by connecting to the container using telnet. This is done by running the following command in a new terminal (Ctrl + Alt + T):

   ```
   echo "stats items" | nc 127.0.0.1 11211
   ```

   The output should be similar to the output shown in the illustration below:

   If the result is similar to the one shown in the output above the Memcached cache is working correctly.

Now let's create a desktop shortcut which can be used to start the Memcached docker container. This is done by performing the following steps:

1) Create a new desktop shortcut, used to start the Memcached docker container, by running the following command: `touch ~/Desktop/Start-Memcached.desktop`

2) Add the following code to the file that is created on the desktop.

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "docker run --rm --name cesium-memcache -p 11211:11211 memcached -m 200"
Icon=gnome-panel-launcher
Name[en_US]=Start-Memcached
```

Now let's create a desktop shortcut which can be used to start the Cesium Terrain Server. This is done by performing the following steps:

3) Create a new desktop shortcut, used to start the Cesium Terrain Server, by running the following command: `touch ~/Desktop/Start-Cesium-Server.desktop`

4) Add the following code to the file that is created on the desktop and save it afterwards:

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "cd ~/Geostack/cesium-server && docker run -p 8080:8000 -v
/home/geostack/Geostack/cesium-server/data/tilesets/terrain:/data/tilesets/
terrain --link cesium-memcache:memcached geodata/cesium-terrain-server"
Icon=gnome-panel-launcher
Name[en_US]=Start-Cesium-Server
```

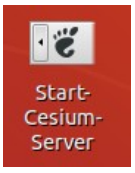
5) Make sure the desktop shortcuts are set to trusted by running the following command:

```
for i in ~/Desktop/*.desktop; do  gio set "$i" "metadata::trusted" yes ;done
```

6) Make the shortcuts launch-able by running the following command:

```
sudo chmod +x ~/Desktop/Start-Memcached.desktop && sudo chmod +x \
~/Desktop/Start-Cesium-Server.desktop
```

Now you have 2 new desktop icons which can be used to start the Memcached cache and the Cesium-Terrain Server as shown in the illustrations below:



NOTE: when running the Cesium Terrain Server remember to first start the Memcached cache using the shortcut which you created above!

5.7.3.5 Automating the Cesium Terrain file generation process

To be able to quickly render new Terrain files, you can create a script by performing the following steps:

1) In the folder: ”~/Geostack/import-utilities/” add a new file called: cesium-terrain-import.sh by running the command:

   touch ~/Geostack/import-utilities/cesium-terrain-import.sh

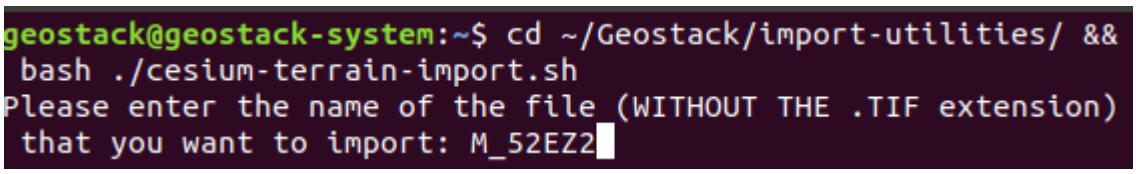
2) Then add the following code to this script:

```
#!/bin/sh

# Ask for user input.
echo -n "Please enter the name of the file (WITHOUT THE .TIF extension) that you want to import: "
# Read the user input.
read VAR

# Check if the user input is not empty
if [[ $VAR != '' ]]
then
        # Create a new directory for the Terrain files.
        echo "Creating a new directory"
        mkdir -p ~/Geostack/cesium-server/data/tilesets/terrain/$VAR

        # Tranform the file to WGS84 projection.
        echo "Transforming tif file to WGS84"
        docker run -v ~/Geostack/cesium-server/data:/data geodata/gdal gdalwarp -t_srs EPSG:3857\
 /data/$VAR.TIF /data/$VAR+3857.TIF

        # Generate the terrain files using the CTB-Quantized-Mesh tool.
        echo " Running 'tumgis/ctb-quantized-mesh', to generate the terrain files"
        docker run -it -v ~/Geostack/cesium-server/data:/data tumgis/ctb-quantized-mesh\
 ctb-tile -f Mesh -C -o /data/tilesets/terrain/$VAR /data/$VAR+3857.TIF

        # Generate the terrain files metadata using the -l parameter.
        echo "Creating the layer.json file for Cesium"
        docker run -it -v ~/Geostack/cesium-server/data:/data tumgis/ctb-quantized-mesh\
 ctb-tile -l -f Mesh -C -o /data/tilesets/terrain/$VAR /data/$VAR+3857.TIF

        # Print when process is done
        echo "Done, you can now access the new terrain files via the URL:
http://localhost/terrain/$VAR/0/0/0.terrain or http://localhost/tilesets/terrain/$VAR/0/0/0.terrain "
        echo "NOTE: The cesium server should be running when trying to access the terrain files"
fi
```

3) Start the script by running the following command:

   cd ~/Geostack/import-utilities/ && bash ./cesium-terrain-import.sh

4) Then make sure the .TIF file, from which you want to generate the .terrain files, is located in the folder: “~/Geostack/cesium-server/data/” and enter the name of the file in the terminal.
   NOTE: enter the name without the .TIF extension as shown in the illustration below.



5.7.3.6 Adding the Cesium Terrain Server to the NGINX configuration

To make sure the Cesium Terrain Server is accessible via the NGINX web server you have to perform the following steps:

3) Open the **LOCAL** NGINX configuration file called: “nginx-local.conf” located in the folder: “/Geostack/nginx-modsecurity/”.

4) Add the following upstream server below the part where you specified the upstream server for the TileStache tile server:

```
# Here the upstream server for our Cesium terrain server is created
upstream cesium-terrain-server{
      server localhost:8080;
}
```

5) Add the following below the part where you specified the location of the Flask API:

```
# Location of our Cessium .terrain files.
location /terrain/ {
        proxy_pass http://cesium-terrain-server/tilesets/;
}
```

3) Save the configuration files by pressing the following keys on your keyboard: Ctrl + S

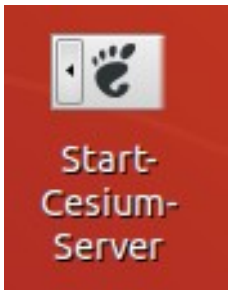
4) Copy and rename the NGINX configuration to the correct folder in the system by running the following command:

   sudo cp ~/Geostack/nginx-modsecurity/nginx-local.conf /etc/nginx/nginx.conf

5) Restart the NGINX service for the changes to take effect by running the following command: sudo service nginx restart

That’s it! Now when you navigate to http://localhost/terrain/M_52EZ2/layer.json, you should be greeted with the terrain files metadata served by the Cesium Terrain Server which is running behind the NGINX web server.

Remember to first start the Memcached cache and the Cesium Terrain Server using the desktop shortcuts shown in the illustration below.



5.7.3.7 Dockerizing the Cesium Terrain Server

Now that you have finished setting up the local version of the Cesium Terrain Server you can Dockerize it. This is done by performing the following steps:

1) Add the cesium-terrain-server service to the docker-compose.yml file located in the GeoStack root folder.

```
memcached-cache:
  # Here we set the name of the memcached container
  container_name: memcached-cache
  # Here we define the image that is used for this container
  image: memcached:latest
  # The line below makes sure the container restarts when stopping accidentally
  restart: always
  # Here we define the ports on which the container is available
  ports:
   - “11211"
cesium-terrain-server:
  # Here we set the name of the container
  container_name: cesium-terrain-server
  # Here we define the image that is used for this container
  image: geodata/cesium-terrain-server
  # The line below makes sure the container restarts when stopping accidentally
  restart: always
  # Here we define the ports on which the container is available
  ports:
   - "8080:8000"
   #- "8080"
  volumes:
  # Here we add the terrain folder as volume used in the container.
   - ./cesium-server/data/tilesets/terrain:/data/tilesets/terrain
  # Here we create a container link to the memcached container
  links:
   - memcached-cache:memcached
  # Below we define the services on which the Cesium Server depends.
  # These services will be started before the Cesium Server starts.
  depends_on:
   - memcached-cache
```

2) Add the following upstream server below the upstream server for the TileStache server:

```
# Here the upstream server for our Cesium terrain server is created
upstream cesium-terrain-server{
    server cesium-terrain-server:8000;
}
```

3) Add the following below the line where you specified the location of the last WMS from the tile server. (The landscape map WMS location):

```
# Location of our Cessium .terrain files.
location /terrain/ {
      proxy_pass http://cesium-terrain-server/tilesets/;
}
```

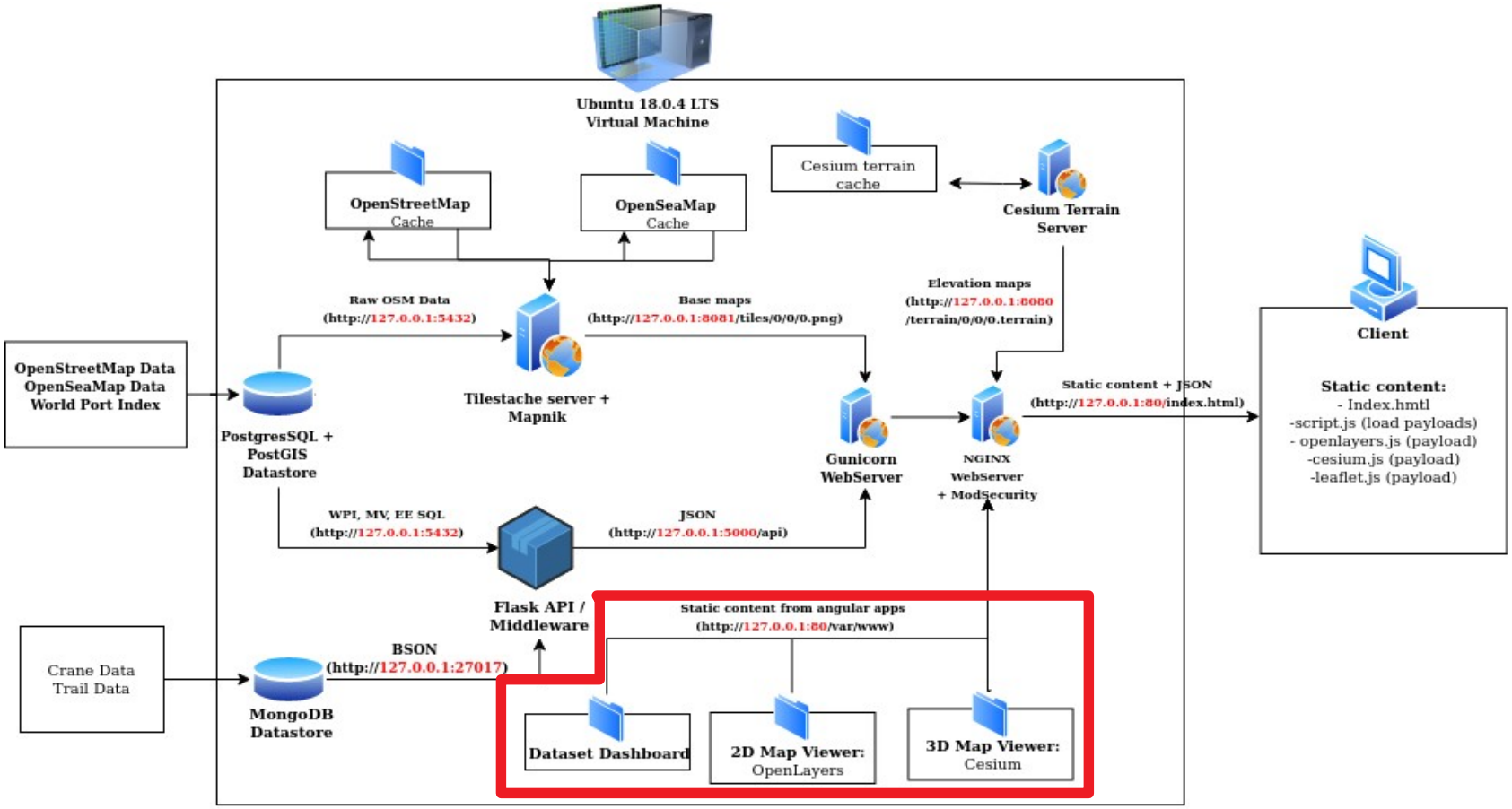
4) Save the configuration files by pressing the keys on your keyboard: Ctrl + S

5) Rebuild the NGINX service for the changes to take effect:
cd ~/Geostack && docker-compose build nginx-webserver

That’s it! Now you have ‘Dockerized’ your Cesium Terrain Server.
In Chapter 6 you will be running the GeoStack as Docker containers and you will be able to see the end result but now first continue to create the frontend of your GeoStack!

5.8 Installing the frontend software

At this point you have all the backend software for data storage and middleware software for web services and web servers in place. You also have modeled, indexed and imported the data in the corresponding datastores. The data now has to be visualized by creating the 3 web applications: Dataset Dashboard, 2D Map Viewer and 3D Map Viewer. First you need to install the software that is required to program these web applications.



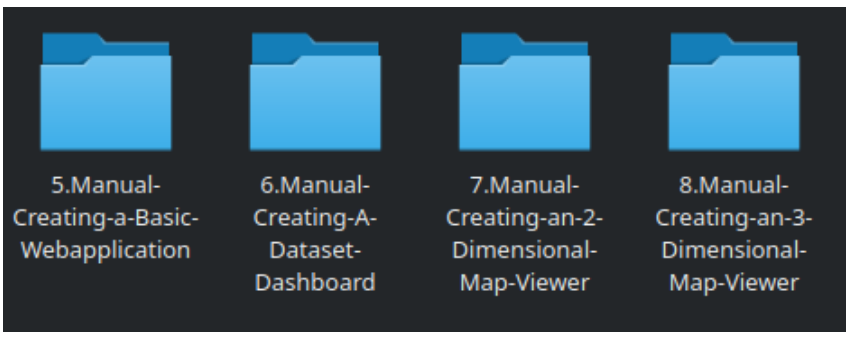
Installing the Angular CLI is done by running the following command:

`sudo npm install -g @angular/cli`

Now you create a folder called: “angular-apps” in the root GeoStack folder. This folder is going to contain all the web applications which you are going to program.
To create this folder you open a terminal (with Ctrl + Alt + T) and execute the following command: `mkdir ~/Geostack/angular-apps`

At this point you have the Angular CLI (command line interface) installed. This software product will be used to create the web applications.

Each of the web applications has it’s own programming manual which can be found in the folder: “Building-the-VM-from-scratch-using-the-manuals” as shown in the illustration below.



These manuals are intended to enable you to learn the following:

- **Manual Creating a Basic Web application:**
  In this manual you will learn how to create the base of all the applications in our GeoStack. The other programming manuals are an extension of this manual.
  This manual also contains some information related to the JavaScript framework Angular and how to use it with the TypeScript programming language as glue code for it.
- **Manual Creating a Dataset Dashboard:**
  In this manual you will learn how to create a Dataset Dashboard which shows all the datasets in our MongoDB datastore. The dashboard contains interactive graphs and tables. You will also learn how to be able to generate data profiles using Pandas-Profiling.
- **Manual Creating a 2D Map Viewer:**
  In this manual you will learn how to create a 2D Map Viewer using the geospatial framework OpenLayers. This application is used to visualize data in 2D.
  The functionality of this application can be found in the table related to the 2D Map Viewer found in chapter 1 of this document.
- **Manual Creating a 3D Map Viewer:**
  In this manual you will learn how to create a 3D Map Viewer using the geospatial framework Cesium. This application is used to visualize data in 3D and displaying height maps. The functionality of this application can be found in the table related to the 3D Map Viewer found in chapter 1 of this document.

You should start by reading the manual related to creating the Basic web application. After you have finished creating this application you can choose which application you want to create next. All the code created during the programming manuals can be found in the POC folder which is located in the same folder as the Manual in question. It’s highly recommended to use this source code when creating the applications using the manuals. After you have finished creating the applications you can come back to this cookbook to start reading the next chapters.

5.8.1 Installing and updating an Angular Project

When you want to distribute an Angular project you should remove the Node_Modules folder which is located in the root folder of the Angular project which you want to distribute. This folder contains a lot of small files which will slow down the process of zipping or downloading the project significantly.
During this section you will be installing and updating the base web application which you created by following the programming manual: "**Creating a basic web application**".

So, now navigate to the root folder of the base application by running the following command:

`cd ~/Geostack/angular-apps/base-application/`

The Node Modules can be installed using the following command from the root folder of the Angular project: `sudo npm install`

The result of this command will look similar to the one shown in the illustration below:

```
added 1013 packages from 533 contributors and audited 16050 packages in 25.667s

26 packages are looking for funding
  run `npm fund` for details

found 71 vulnerabilities (70 low, 1 moderate)
  run `npm audit fix` to fix them, or `npm audit` for details
```

As you can see the installer found some vulnerabilities which have to be fixed.
Run the following command to get an overview of the vulnerabilities which have to be fixed:
`sudo npm audit fix`

This will result in a long list of dependencies which have to be fixed. A part of the output is shown in the illustration below:

Low	Prototype Pollution
Package	minimist
Patched in	>=0.2.1 <1.0.0 || >=1.2.3
Dependency of	@angular-devkit/build-angular [dev]
Path	@angular-devkit/build-angular > webpack-dev-server > chokidar > fsevents > node-pre-gyp > rc > minimist
More info	https://npmjs.com/advisories/1179

```
found 71 vulnerabilities (70 low, 1 moderate) in 16050 scanned packages
  run `npm audit fix` to fix 62 of them.
  3 vulnerabilities require semver-major dependency updates.
  6 vulnerabilities require manual review. See the full report for details.
```

Now fix these vulnerabilities by updating the Node Modules.

- Updating an Angular project or app can be a tricky process because some dependencies only work together when certain versions of these dependencies are installed.

The first step is to update the modules which won't break the application by running the command: `sudo npm audit fix`

Now let this fixing process finish. The output of this command is shown in the illustration below:

```
added 1 package from 3 contributors, removed 1 package and updated 5 packages in 6.49s

26 packages are looking for funding
  run `npm fund` for details

fixed 62 of 71 vulnerabilities in 16050 scanned packages
  6 vulnerabilities required manual review and could not be updated
  1 package update for 3 vulnerabilities involved breaking changes
  (use `npm audit fix --force` to install breaking changes; or refer to `npm audit` for steps to fix these manually)
```

As you can see 62 of the 71 vulnerabilities which were found are now fixed and 6 of the vulnerabilities have to be reviewed manually since they could not be updated. One package update will break the application if it's performed.

If you are VERY sure the update will not break the application you can run the command: `sudo npm audit fix –force`

- It's not recommended to run this command since it could break your Angular Project and thus your application!
- When running the command: 'sudo npm audit' again you will be greeted with a list of dependencies which have to be reviewed manually.
- To update these packages you should search the internet to see what's the best option.

To be sure your application is still working you should run the command: `sudo npm start`

- When the output of the command is the same as shown in the illustration below you have successfully updated your Angular Project and thus your application.

```
chunk {main} main.js, main.js.map (main) 48.6 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 268 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 2.33 MB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.87 MB [initial] [rendered]
Date: 2020-04-12T15:52:57.028Z - Hash: 9e8b71f9df2d0f85b53b - Time: 11766ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://l
ocalhost:4200/ **
i ｢wdm｣: Compiled successfully.
```

5.8.2 Serving an Angular app on the NGINX Web server

To serve an Angular application from behind an NGINX web server you need to take some steps.

- The first step is to compile your Angular App into a set of plain HTML/Javascript/CSS files.
- Because you created the Angular apps by using the Angular-CLI (@angular/cli) this can be done with the command: ng build --prod

Here is the example of how to build the 2D Map Viewer web application as a set of static files and to run the app from that set of files behind the NGINX web server in 5 simple steps:

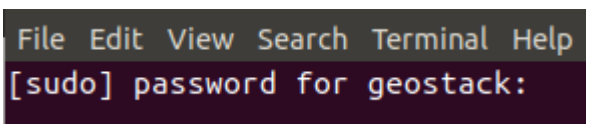
1) Enter the 2D Map Viewer application root directory by running the following command: `cd ~/Geostack/angular-apps/2d-map-viewer/`
2) Build the production version of our 2D Map Viewer by running the following command: `ng build --prod --base-href /2d-map-viewer/`

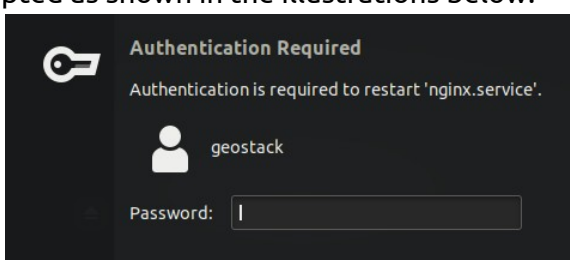
   The "base-href" flag is used to specify the location on which the 2D Map Viewer application will be accessible via the NGINX web server (http://localhost/2d-map-viewer).

3) Now you need to specify a new location in your NGINX configuration file called: “nginx-local.conf” which is located in the folder “~/Geostack/nginx-modsecurity/”.
Open this file and add the following lines below the first specified location in the configuration file (the “/” location).

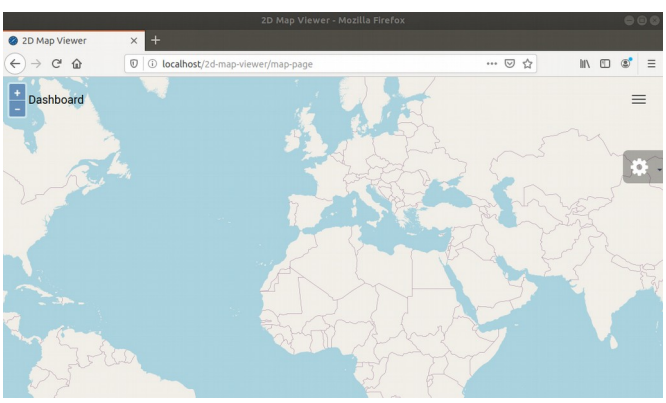
```
# Below we specify the location on which the 2D Map Viewer will
# be accessible
location /2d-map-viewer/ {

        # Here we tell NGINX to include media types when handling request
        # Media types are also known as a Multipurpose Internet Mail
        # Extensions or MIME types (CSS, TXT, IMG).
        include       mime.types;

        # Here we set the root folder to the location of the dist folder which
        # was generated when running the command: ng prod --build.
        root   /home/geostack/Geostack/angular-apps/2d-map-viewer/dist/;

        # Here we tell NGINX to search for an index.html file located in the
        # dist folder from which the location was specified above.
        try_files $uri /2d-map-viewer/index.html;
}
```

4) Save the NGINX configuration file and click on the desktop shortcut called: “Restart-NGINX” which will first copy the updated NGINX configuration file to the correct location in the system (“/etc/nginx/nginx.conf”) and then reload the NGINX web server for the changes to take effect.
Enter the password: “geostack” when prompted as shown in the illustrations below:





5) Now when you navigate to the following URL: http://localhost/2d-map-viewer/ you will be greeted by the 2D Map Viewer which is served as static content by the NGINX web server.
NOTE: make sure the Flask API and the TileStache tile server are running for the data and the OSM map to be able to load.



Since the process of serving an Angular app via NGINX is the same for all web apps, you now know how to do the same for the Base-Application, Dataset-Dashboard and 3D Map Viewer.

5.8.3 Installing the geospatial frameworks OpenLayers and Cesium

Since the geospatial frameworks are used in the web applications next to the generic web app framework Angular, their installation is explained in the programming manuals.

The programming manual for the 2D Map Viewer explains the installation of OpenLayers and the in the manual for the 3D Map Viewer the installation of Cesium is explained.

For OpenLayers both the standard package installation with the npm package manager is explained next to the manual installation which is used in the GeoStack Course to learn how you can have version control over the framework version you want to use in your web apps.

The Cesium manual not only focuses on the required conversion of DEM files to '3D map tiles' in the terrain files for the Cesium Terrain Server but it also explains the differences in application development for web apps about drawing 2D maps with OpenLayers and 3D maps with Cesium.

To run OpenLayers locally without an internet connection a local installation of the OpenLayers JavaScript source code package is all you need and the same goes for Cesium.

Of course you still will need online access to the web map servers to get the topographical maps from OpenStreetMap or a connection to the Cesium-ION web services for the terrain height files and imagery for 3D maps!

Challenge --> Run the GeoStack completely off line in 'airplane mode'!

If you want to run completely locally 'in airplaine' mode without any internet connection then of course you will need some serious data storage and computing power!

Running in airplane mode requires vast amounts of data storage for the raw topographical map data, like your own local copy of OpenStreetMap and if you need satellite and air imagery that will take up an even much larger amount of disk space and if you want 3D maps the DEM files require massive storage space too.

That being said, once you completed the GeoStack Course it is not that difficult and also very educational to run an entire GeoStack locally on your laptop or small server.

- Tip 1: choose a small geographical area like a big city, a province or a small country!
- Tip 2: a fast extra SSD of 1 or 2 TB helps!
- Tip 3: first simple 2D maps with OpenLayers and then the more complex 3D with Cesium.

Have fun!

5.9 Running the GeoStack as Docker containers

5.9.1 Checking the docker-compose.yml file

If you did everything correct, you should end up with the source code as presented in this section in the file docker-compose.yml which is located in the folder: "~/Geostack".

Remember you created this configuration file in the section 'Installing the Docker virtualization software'.

The goal of this configuration file is to start all the docker containers and volumes of the GeoStack in one (1) command with Docker-compose but remember this will take something like 10 – 15 minutes to get everything started.

In this section the source code of the configuration file is listed to enable you to check if you got your multi-container Docker configuration coded correct!

Check when you read it back if you still remember what all the configuration specifications do because it has been a while since you created the file and started adding all the GeoStack components to it!

NOTE: the source code is split in 3 sections over the next 3 pages for readability!

--> The rest of this page is intentionally left blank! <--

```
#Define the docker compose version
version: '3.7'
#Defining the services
services:
  postgresql-datastore:
    # Here we define the name which the PostgreSQL container is going to have.
    container_name: postgresql-datastore
    # Set the directory in which the dockerfile is located
    build: ./postgresql-datastore
    # Add the data volume of the docker container
    volumes:
      - ./postgresql-datastore/data:/var/lib/postgresql/data
     # Set the port on which the docker container is available to port 5432
     # Since we set it to port 5432:5432, the docker container will also  be accessible on
     # our host system via localhost:5432
    ports:
      - '5432:5432'
    # Here we add the environment variable which allows connections without a pass.
    environment:
      POSTGRES_HOST_AUTH_METHOD: "trust"

  # Here we define the name of the MongoDB datastore Docker service
  mongodb-datastore:
    # Here we define the name which the MongoDB container is going to have
    container_name: mongodb-datastore
    # Here we define the image that is used for this container
    image: mongo:latest
    # Add the data volume of the docker container
    volumes:
      - ./mongodb-datastore/data:/data/db
     # Set the port on which the docker container is available to port 27017
     # Since we set it to port 27017:27017, the docker container will also  be
     # accessible on our host system via localhost:27017
    ports:
      - '27017:27017'

  # Below we define the service for the Middleware components in Geostack
  nginx-webserver:
    container_name: nginx-webserver
    # Set the directory in which the dockerfile is located
    build: ./nginx-modsecurity
    # The line below makes sure the NGINX container restarts when stopping
    # accidentally
    restart: always
     # Set the port on which the docker container is available to port 80
     # Since we set it to port 80:80  the docker container will also be accessible
     # on our host system via localhost:80 or just localhost
    ports:
      - "80:80"
    # Below we define the services on which the NGINX webserver depends.
    depends_on:
      - flask-api
      - tilestache-server
      - cesium-terrain-server
```

```
 flask-api:
  # Here we set the name of the Flask-API / App container
  container_name: flask-api
  # The line below makes sure the Flask container restarts when stopping
  # accidentally
  restart: always
  # Set the directory in which the dockerfile is located
  build: ./flask-gunicorn
  # Here we set the port on which the Tileserver will be running
  ports:
   - "5000"
  # Here we set the command that will be executed when the Flask-API
  # service starts.
  command: gunicorn -b :5000 app:app
  # Here we add the Downloads volume of the docker container
  volumes:
   - .:/downloads
  # Below we define the services on which the Flask-API depends.
  # These services will be started before the Flask-API starts.
  depends_on:
   - mongodb-datastore
   - postgresql-datastore
   - tilestache-server

 # Here we define the name of the tilestache-server service
 tilestache-server:
   # Here we define the name of the tilestache-server container
  container_name: tilestache-server
  # The line below makes sure the Tileserver container restarts when stopping accidentally
  restart: always
   # Here we set the directory in which the Dockerfile is located
  build: ./tilestache-server
  # Here we add the cache as volume so the local cache is shared with the docker cache
  volumes:
   - ./tilestache-server/cache:/tilestache/cache
  # Here we set the port on which the Tileserver will be running
  ports:
   - '8081'
  # Here we define the command that will run when the docker container is started
  command: gunicorn --workers 4 --timeout 100 -b :8081
"TileStache:WSGITileServer('/tilestache/tilestache-configuration.cfg')"

 memcached-cache:
  # Here we set the name of the memcached container
  container_name: memcached-cache
  # Here we define the image that is used for this container
  image: memcached:latest
  # The line below makes sure the container restarts when stopping accidentally
  restart: always
  # Here we define the ports on which the container is available
  ports:
   - "11211"
```

```
cesium-terrain-server:
  # Here we set the name of the container
  container_name: cesium-terrain-server
  # Here we define the image that is used for this container
  image: geodata/cesium-terrain-server
  # The line below makes sure the container restarts when stopping accidentally
  restart: always
  # Here we define the ports on which the container is available
  ports:
   - "8080:8000"
   #- "8080"
  volumes:
  # Here we add the terrain folder as volume used in the container.
   - ./cesium-server/data/tilesets/terrain:/data/tilesets/terrain
  # Here we create a container link to the memcached container
  links:
   - memcached-cache:memcached
  # Below we define the services on which the Cesium Server depends.
  # These services will be started before the Cesium Server starts.
  depends_on:
   - memcached-cache
```

Before running the Dockerized GeoStack you first have to make sure the Local NGINX web server, TileStache tile server, Flask API, MongoDB datastore and PostgreSQL datastore are not running, otherwise the ports of the local instances of the GeoStack components will conflict with the ports of our running Docker containers. To stop the components listed above you can click on the desktop shortcuts that you created.

Now you can run the following command to run all the docker containers in our docker compose file: `cd ~/Geostack && docker-compose up`

NOTE: If this is the first time the docker containers are build the process of starting the containers will take a while.

NOTE 2: Before the Dockerized tile server will work you first need to download and import the OpenStreetMap-Carto Base Map Shapefiles in the Dockerized PostgreSQL 'gis ' database. This is done by opening a terminal (with Ctrl + Alt + T) and running the following command after all the Docker containers are started and running:

```
docker exec -it tilestache-server python3 tilestache/openstreetmap-carto/scripts/get-external-data.py -H postgresql-datastore -U geostack -c /tilestache/openstreetmap-carto/external-data.yml
```

This command makes sure the get-external-data.py script in the tilestache-server Docker container is executed. The -H flag is used to specify the Dockerized PostgreSQL instance, the -U flag is used to specify the PostgreSQL user and the -c flag is used to specify the external-data configuration file.

5.9.2 Create a Desktop Shortcut Configuration File for Docker Compose

Now let's create a new desktop shortcut which can be used to start the entire Dockerized GeoStack with one (1) command and stop the local instances of the GeoStack components.

First you need to create a configuration file for the desktop shortcut.
You do this by running the following command: touch ~/Desktop/Start-Geostack-Docker.desktop

Add the following lines to the new file which was created on the desktop:

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "service mongodb stop && service postgresql stop && service nginx stop && cd ~/Geostack && docker-compose up"
Icon=gnome-panel-launcher
Name[en_US]=Start-Geostack-Docker
```

The command that is executed, when running this desktop shortcut, does the following:

- ➔ Stop the Local MongoDB instance, using the command: service mongodb stop;
- ➔ Stop the Local PostgreSQL instance, using the command: service postgresql stop;
- ➔ Stop the Local NGINX instance, using the command: service nginx stop;
- ➔ Navigate to the Geostack directory;
- ➔ Run the GeoStack Docker services, using the command: docker-compose up.

5.9.3 Create the Desktop Shortcut for Docker Compose

Create a new desktop shortcut, used to stop the GeoStack docker version and restart the local instances of the GeoStack components, by running the following command:

touch ~/Desktop/Stop-Geostack-Docker.desktop

Add the following to the new desktop file which was created:

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "cd ~/Geostack && docker-compose down && service mongodb start && service postgresql start && service nginx start"
Icon=gnome-panel-launcher
Name[en_US]=Stop-Geostack-Docker
```

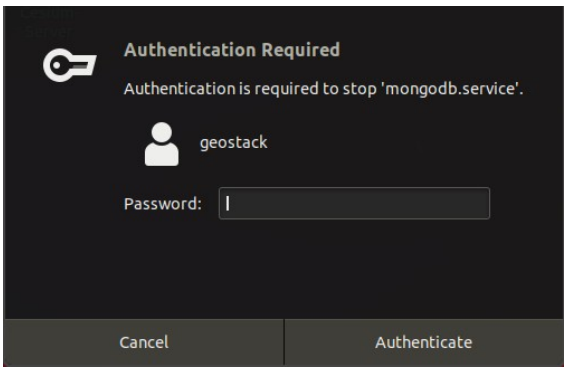
The command that is executed, when running this desktop shortcut, does the following:

- ➔ Navigate to the Geostack directory;
- ➔ Run docker-compose down to stop all the Docker services;
- ➔ Start the Local MongoDB instance, using the command: service mongodb start;
- ➔ Start the Local PostgreSQL instance, using the command: service postgresql start;
- ➔ Start the Local NGINX instance, using the command: service nginx start.

5.9.4 Start the entire GeoStack with Docker Compose

Now you can run the Dockerized GeoStack by clicking on the desktop shortcut shown in the illustration below.

When clicking on the desktop shortcut 3 screens will pop up asking you for a password. One of these screens is shown in the illustration below. Enter the password: “geostack”.



If everything is working correctly, the following output will be shown in the terminal.

```
Creating network "geostack_default" with the default driver
Creating postgresql-datastore  ... done
Creating mongodb-datastore     ... done
Creating tilestache-server     ... done
Creating memcached-cache       ... done
Creating cesium-terrain-server ... done
Creating flask-api             ... done
Creating nginx-webserver       ... done
```

5.9.5 Exporting Docker images and volumes

After all the GeoStack Docker containers are build you can run the following command to list all the current images which are located in our Docker system: `docker images`

The output of this command will be similar to the one shown in the illustration below:

```
geostack@geostack-VirtualBox:~/Geostack$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
geostack_nginx-webserver            latest              9d0bc49b3f27        2 days ago          799MB
geostack_flask-api                  latest              fcf426993324        2 days ago          1.49GB
geostack_tilestache-server          latest              56f138aca6ac        2 days ago          2.56GB
geostack_postgresql-datastore       latest              54697f2434f3        2 days ago          444MB
tumgis/ctb-quantized-mesh           latest              5a47ee3d7463        2 days ago          905MB
memcached                           latest              e310fbc8b97a        3 days ago          82.3MB
postgres                            11                  aa8042237034        2 weeks ago         283MB
mongo                               latest              c5e5843d9f5f        3 weeks ago         387MB
ubuntu                              latest              4e5021d210f6        3 weeks ago         64.2MB
owasp/modsecurity                   3.0-nginx           2862e1502c7f        11 months ago       729MB
python                              3.7.2               2053ca75899e        12 months ago       929MB
geodata/gdal                        latest              1e1929f80f44        2 years ago         1.29GB
geodata/cesium-terrain-server       latest              417b66fcb988        4 years ago         979MB
geostack@geostack-VirtualBox:~/Geostack$
```

Sometimes it's useful to be able to distribute the Docker images / containers and their data volumes to other systems for flexibility in hosting, backup and to share them with others!

So imagine you have another computer on which you want to run the PostgreSQL Datastore Docker container and the data volume containing the World Port Index dataset and the RAW OpenstreetMap data.

You can export a specific image to easily distribute it to other systems. To do this you use the command: `docker save {image name} > {output directory name}.tar`

- Note: Docker ONLY uses export files in the TAR (Tape ARchive) file format for Containers!

So export the PostgreSQL Docker image to a TAR file by running the following command: `docker save geostack_postgresql-datastore > geostack_postgresql-datastore.tar`

This will give you a TAR file with the '.tar' file extension as shown in the illustration below.

Now export the Docker Volume (a Docker virtual disk for a Docker container) containing the World Port Index dataset and the RAW OpenStreetMap data as a ZIP file.

- So yes, it's TAR for Containers and ZIP for Volumes in Docker!

Export the Docker Volume by performing the following steps:

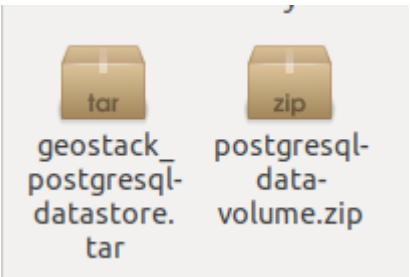
1) Change the file permissions of the PostgreSQL Docker container data volume by running the following command: `sudo chown -R $USER ~/Geostack/postgresql-datastore/data`
2) Zip the data volume by running the following command:
   `zip -r postgresql-data-volume.zip ~/Geostack/postgresql-datastore/data`

   Let this zipping process finish which takes around 20 seconds.

After the zipping process is finished the output should look similar to the output shown in the illustration below:

```
adding: home/geostack/Geostack/postgresql-datastore/data/pg_multixact/offsets/0000 (deflated 100%)
adding: home/geostack/Geostack/postgresql-datastore/data/pg_multixact/members/ (stored 0%)
adding: home/geostack/Geostack/postgresql-datastore/data/pg_multixact/members/0000 (deflated 100%)
```

Now you ended up with a TAR file containing the PostgreSQL Docker image and a ZIP file containing the PostgreSQL Docker data volume as shown in the illustration below:



Now you can distribute these files to another computer by placing them on an external disk drive or transfer them through a cloud drive or to a cloud server if you have online disk space and some or much up- and download time to spare, especially if it's quite a few gigabytes to transfer!

Imagine the other computer or VM on which you want to run the PostgreSQL Docker container with data volume has a docker-compose.yml file containing the following code:

```
#Define the docker compose version
version: '3.7'
#Defining the services
services:
  postgresql-datastore:
    # Here we define the name which the PostgreSQL container is going to have.
    container_name: postgresql-datastore
    # Set the directory in which the dockerfile is located
    build: ./postgresql-datastore
    # Add the data volume of the docker container
    volumes:
      - ./postgresql-datastore/data:/var/lib/postgresql/data
     # Set the port on which the docker container is available to port 5432
     # Since we set it to port 5432:5432, the docker container will also  be accessible on
     # our host system via localhost:5432
    ports:
      - '5432:5432'
    # Here we add the environment variable which allows connections without a pass.
    environment:
      POSTGRES_HOST_AUTH_METHOD: "trust"
```

This is the same PostgreSQL container setup as in your GeoStack. The docker-compose.yml file expects a folder called:"postgresql-datastore" inside the folder which contains the docker-compose.yml file ("~/GeoStack/" in your case).

When the files are on the new computer you need to extract the postgresql-data-volume.zip file by running the following command: `unzip postgresql-data-volume.zip`

Now you will end up with the postgresql-datastore folder which contains the PostgreSQL data volume that was exported (copied) from your source computer system.

Now load (import) the PostgreSQL Docker container which you copied by running the following command: `docker load --input geostack_postgresql-datastore.tar`

This will load the PostgreSQL Docker container from your other system into your new system. The output of the command which you used above should be similar to the one shown in the illustration below:

```
java@java-VirtualBox:~$ docker load --input geostack_postgresql-datastore.tar
0632b4d712bb: Loading layer  58.48MB/58.48MB
b83557c279b6: Loading layer  10.44MB/10.44MB
e863880e5bbb: Loading layer  339.5kB/339.5kB
60ca2c79f388: Loading layer  4.068MB/4.068MB
2febf793865d: Loading layer   17.1MB/17.1MB
10875082519c: Loading layer  1.426MB/1.426MB
16bac8cf620e: Loading layer  1.536kB/1.536kB
5ccbf3488556: Loading layer  9.216kB/9.216kB
ff627b786feb: Loading layer  198.3MB/198.3MB
f4535c863bdf: Loading layer  56.32kB/56.32kB
1464888a40e8: Loading layer  2.048kB/2.048kB
353ce8ed7c7b: Loading layer  3.072kB/3.072kB
6d43c8ccb009: Loading layer  14.34kB/14.34kB
5d8431511db7: Loading layer  1.536kB/1.536kB
3accbe6aeebb: Loading layer  162.3MB/162.3MB
5e3fec960fe6: Loading layer   2.56kB/2.56kB
e24b4cfe7bfd: Loading layer   2.56kB/2.56kB
Loaded image: geostack_postgresql-datastore:latest
```

Now when you run the following command you can see the Docker container is correctly imported: `docker images`

The output of this command should be similar to the one shown in the illustration below:

```
java@java-VirtualBox:~$ docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
geostack_postgresql-datastore   latest              54697f2434f3        6 days ago          444MB
```

Now when you run the following command from the folder in which the docker-compose.yml file is located, you will have a working PostgreSQL Docker container copied from our orginal GeoStack virtual machine: `docker compose up`

You can perform the steps described in the section ‘Managing the PostgreSQL Databases’ to check whether the data volume is working accordingly.

Do that as an extra exercise and check the World Port Index database and the OpenStreetMap GIS database are shown in the list of PostgreSQL databases as shown in the illustration below.

```
World_Port_Index_Database | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
gis                       | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
```

6 How to continue the GeoStack Course?

Read this if you did the Automatic Installation of the GeoStack!

If you used the Automatic Installation of the GeoStack you have got your Virtual Machine up and running the fast and easy way. Good for you!
Probably because you wanted to learn first about how the GeoStack VM is build or because you wanted to focus fast on programming the web applications and that's fine too! Save some time!

- The advice is to first globally read through chapter 5 'Installing the GeoStack MANUALLY' to get an idea of what exactly was installed by the installations scripts and how things work internally.
  No need to get into the details but if you run into something that doesn't work or something you don't understand, at least you know where to find the information!
- Now decide if you want to continue the manual installation to learn more about the installation of the server software or if you want to start programming the web apps.

Read this if you did the Manual Installation of the GeoStack!

If you worked your way through the Manual Installation of the GeoStack up to this point you have come a long way and learned a lot! Congratulations!
The server side of things now works and you also know how and why!

How to finish this cookbook?

Read chapter 7 'Useful Tips & Tricks' if you didn't already!

- There is some good stuff here for newbie users of Linux, VirtualBox, Atom etc.!

Explore chapter 8 'Useful Weblinks' if you didn't already!

- See if there are any weblinks to read up on some subjects to expand your knowledge!

Overview of the GeoStack Course up to this point!

In this cookbook 'Creating the GeoStack Course VM' you have already worked through the following cookbooks:

1. Cookbook 'ETL Process with Datasets'
2. Cookbook 'Data Modeling in MongoDB'
3. Cookbook 'Creating a Python – Flask Web Application'
4. Cookbook 'A Secure NGINX Web Server with ModSecurity'

How to continue the GeoStack Course?

The server and datasets are there and then the next step is programming the web applications by following these 4 programming manuals in this sequence of increasing complexity:

1. Programming Manual 'Creating a Basic Web Application' (with Angular + TypeScript)
2. Programming Manual 'Creating a Dataset Dashboard' (with Angular + TypeScript)
3. Programming Manual 'Creating an 2 Dimensional Map Viewer' (with OpenLayers)
4. Programming Manual 'Creating an 3 Dimensional Map Viewer' (with Cesium)

7 Useful Tips & Tricks

In this chapter some useful tips and tricks are discussed. These tips and tricks can increase the workflow speed during and after completing the GeoStack Course.

7.1 Linux Stuff

7.1.1 Creating desktop shortcuts

During this cookbook multiple desktop shortcuts are created. Creating Desktop shortcuts in Ubuntu 18.04 is a bit different from Ubuntu 19.10 and 20.04. The way a desktop shortcut is created (in Ubuntu 18.04) is as follows:

1) First you created a new empty text file on your desktop. This file should have the file extension: ".desktop" and is can be created using the following command:

   `touch ~/Desktop/{shortcut name}.desktop`

2) Then you should open the file, add the following code and save it afterwards:

```
# The line below makes sure that the system knows that the text file is
executable.
#!/usr/bin/env xdg-open

# The line below makes sure that the system knows that the file is a desktop
shortcut.
[Desktop Entry]

# The Line below defines the version of the desktop shortcut.
Version=1.0

# The line below defines what type of shortcut the shortcut is.  Set it to
# application if you want the shortcut to be an application launcher.
Type=Application

# The line below defines whether a terminal should be opened when the
# shortcut is launched. It's recommended to set this value to True.
Terminal=true

# The line below defines the command that should be executed when the
# desktop shortcut is clicked. Using the syntax: "sh -c "command" " tells the
# system that the command needs to be executed as root user.
Exec=sh -c "{your command}"

# The line below defines what type of icon should be displayed on the desktop.
# This value can be set to another type of icon if you want it to.
Icon=gnome-panel-launcher

# The line below defines the name of the Desktop shortcut.
Name[en_US]={The shortcut name}
```

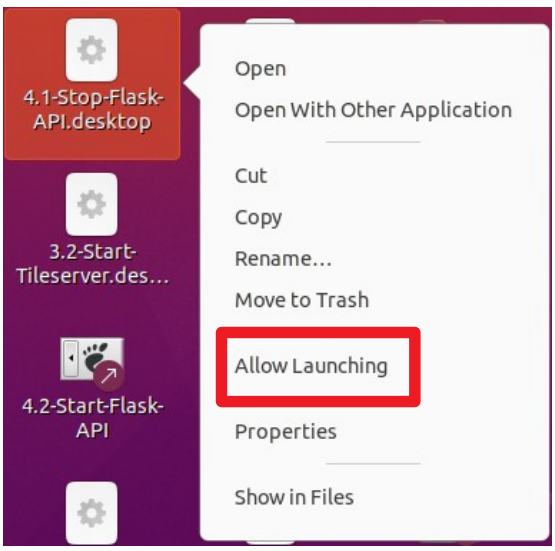
After you have saved the file you want to make sure the desktop shortcut is trusted. This is done by running the following command:

```
for i in ~/Desktop/*.desktop; do  gio set "$i" "metadata::trusted" yes ;done
```

Finally you need to make sure the shortcut is launch-able. This is done by running the following command: `sudo chmod +x ~/Desktop/{shortcut name}.desktop`

Now if you are using Ubuntu 19.10 or higher you should perform one more step before a desktop shortcut can be used.

If you are on **Ubuntu 19.10 or 20.04** you need to allow the desktops shortcuts to be executable. This is done by right-clicking a desktop shortcut and the selecting Allow Launching in the menu that pops up as shown in the illustration below:



That's it!

Now the shortcut should be executable and can be run by double clicking on the shortcut which you just created.

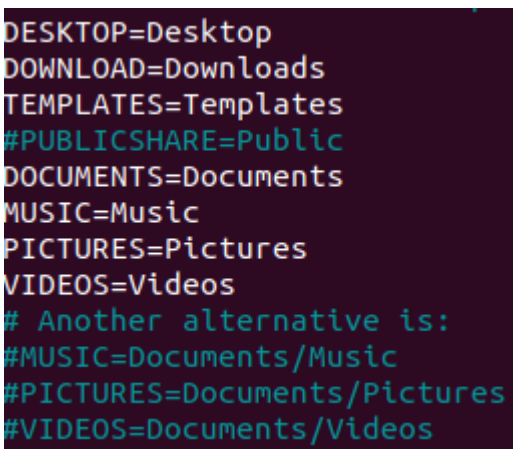
7.1.2 Removing DEFAULT folders from Nautilus

If you want to remove unused **DEFAULT** folders from the Nautilus folder viewer you should perform the following steps:

1) Edit the user directory defaults file by opening it, using the following command:

   `sudo nano /etc/xdg/user-dirs.defaults`

2) Put a "#" in front of the folders that you want to remove, as shown in the illustration below.

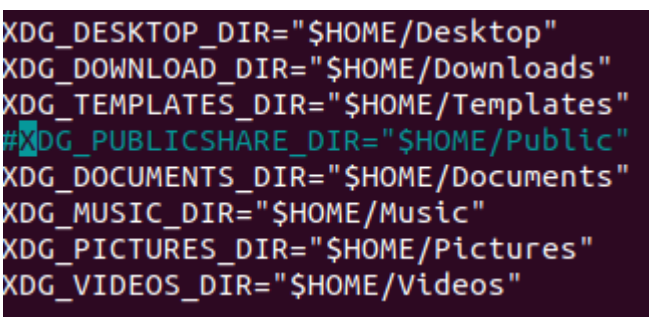


3) Save the file by pressing Ctrl + S on you keyboard.

4) Edit the user directory configuration file by opening it, using the following command:

   `sudo nano ~/.config/user-dirs.dirs`

5) Put a "#" in front of the folders that you want to remove, as shown in the illustration below.



6) Save the file by pressing Ctrl + S on you keyboard.

7) Stop nautilus by running the following command: `pkill nautilus`

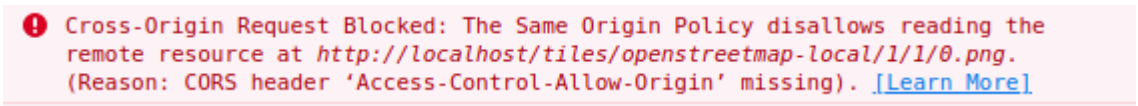
8) Restart nautilus by running the following command:

   `nautilus-desktop > /dev/null 2>&1 & echo "Restarted"`

Now the folder which you removed should be removed from the Nautilus folder viewer.
If this is not the case you should restart you Virtual Machine.

7.1.3 Allow Cross-Origin Resource Sharing (CORS)

During the creation of a software stack there is a strong possibility you will encounter an error similar to the one shown in the illustration below:

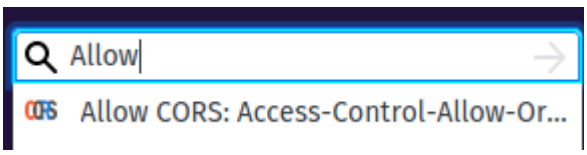


These error's occur when trying to obtain resources from multiple web servers at the same time. In the example above this error occurred when trying to obtain OpenStreetMap Tiles from the TileStache tile server (running on "localhost:8081") in the 3D Map Viewer running on localhost:4200.

For more information related to the specifics of CORS you should read the following URL:
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

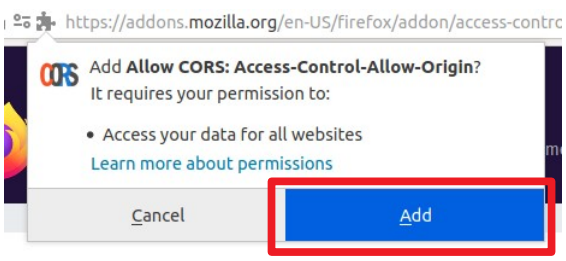
To solve this problem you need to install a Firefox extension called: "Allow CORS". This is done by performing the following steps:

1) Navigate to the Addons website of Firefox by entering the following URL in your browser: https://addons.mozilla.org/en-US/firefox/
2) Type: "Allow CORS" in the search bar in the top right of the page and click on the first option that pops up, as shown in the illustration below:

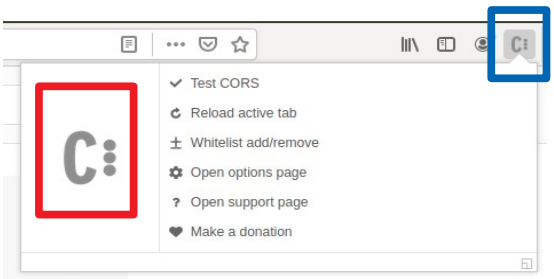


3) On the next screen select: "Add to Firefox" as shown in the illustration below:

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



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

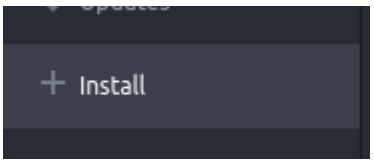


7.2 Applications

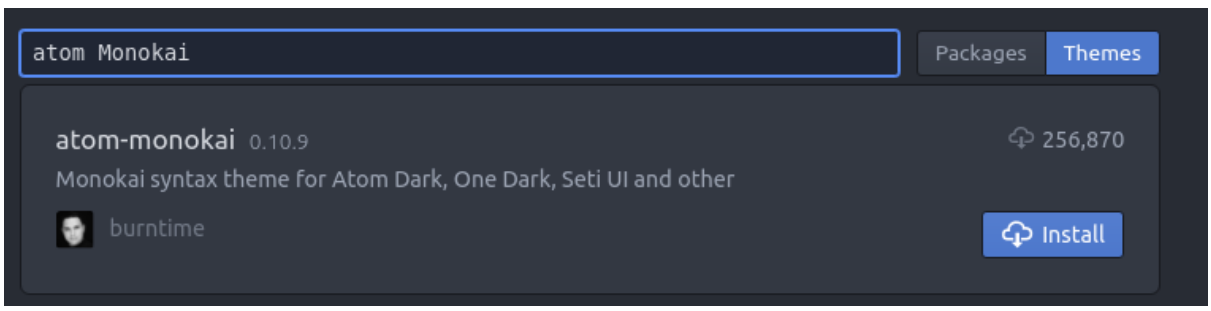
7.2.1 Editing Atom theme settings

You can change the theme of your Atom editor by performing the following steps:

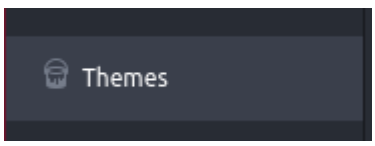
1) At the top of the Atom window go to Edit → Preferences.
2) Select Install in the left menu bar as shown in the illustration below.



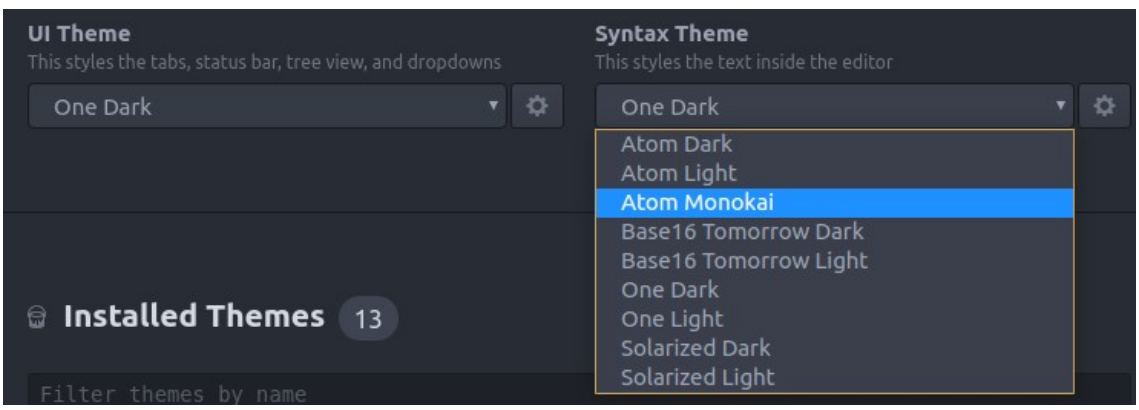
3) Select themes next to the search box, search for Atom Monokai and press install as shown in the illustration below.



4) Select Themes in the left menu bar as shown in the illustration below.



5) Set the syntax theme to Atom Monokai as shown in the illustration below.



6) At the top of the Atom window go to Edit → Stylesheet.
7) Add the following line to the stylesheet and save the file:

```
atom-text-editor .syntax--comment { color: cadetblue; }
```

8) Restart Atom for the changes to take effect.

   This will change the color of the Inline comments to make them more readable.

7.2.2 Cheat Sheet - Useful Docker commands

Some useful Docker commands to learn are listed here in this cheat sheet (tip: print it!):

#	Description	Command
1)	**Find the IP address of a Docker container**	docker inspect -f '{{range .NetworkSettings.Networks}} {{.IPAddress}}{{end}}' {container name or ID}
2)	**List all docker images:**	docker images
3)	**List all running docker containers**	docker ps
4)	**Remove docker image:**	docker image rm {container name or ID} -F
5)	**List all docker containers:**	docker container ls
6)	**Kill a docker container:**	docker stop {container ID}
7)	**Clean docker system:**	docker system prune
8)	**Run docker-compose file:**	docker-compose up
9)	**Run a specific service in the docker-compose file**	docker-compose up {service name}
10)	**Build docker-compose file**	docker-compose up
11)	**Build a specific service in the docker-compose file**	docker-compose build {service name}
12)	**Open a terminal in a running docker container**	docker exec -it {container name or ID} bash
13)	**Show Memcached Docker data**	echo "stats cachedump 15 4" | nc 127.0.0.1 11211
14)	Login to Docker PostgreSQL user	docker exec -it postgresql-datastore bash su postgres psql \l

7.3 VirtualBox

7.3.1 3D Acceleration in your Virtual Machine

On some platforms, and in some circumstances, the wrong renderers may be used by the guest OS which results in very slow 3d performance of the guest. To check whether you Virtual Machine is correctly configured you should perform the following steps:

1) Install NUX Tools, which is used to perform some tests on your Guest system. This is done by running the following command: `sudo apt install nux-tools`

2) Now run the following command to see whether your system is correctly configured:

   `/usr/lib/nux/unity_support_test -p`

   This command should print the following output:

   If this is the case, your system is correctly configured. If this is not the case and you have something similar to the output shown in the illustration below, you should continue with the following steps.

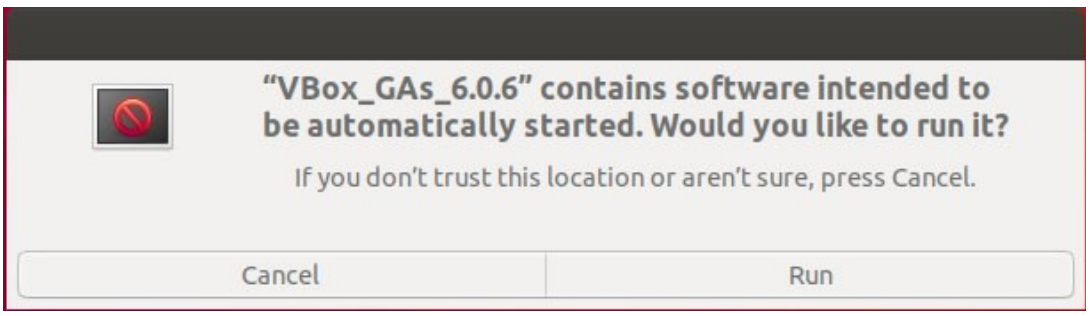
3) Install the required packages for building kernel module by running the following command:

   `sudo apt install dkms build-essential module-assistant`

4) Prepare your system to build kernel module by running the following command:

   `sudo m-a prepare`

5) In VirtualBox menu bar, select **Devices → Insert Guest Additions CD image**; at this point you'll be asked to run the software contained in it, click Run button:



6) After the installation process is done you should rerun the command from step 2 and check if everything is set to yes. If this is not the case you should continue with the following step.

7) Try installing xorg vmware drivers on your Ubuntu guest Virtual Machine by running the following command:

   `sudo apt-get install xserver-xorg-video-vmware-hwe-18.04`

You can also install GL2MARK to see your systems 3D Performance. This is done by running the following command: `sudo apt install glmark2`

Then run GL2Mark by entering the command: `gl2mark`

After the application is done running you will get an score which represents the speed of your graphics adapter.

7.3.2 VirtualBox Graphics Adapters

VirtualBox offers a variety of Graphics Adapters. Depending on the Graphics card in your system you should chose the adapter which should be used. Below you can find some information related to the available graphics adapters.

1) VBoxVGA

This emulates a graphics adapter specific to VirtualBox, the same as in previous versions (<6.0.0).

- This is the default for images created for previous versions of VirtualBox (<6.0.0) and for Windows guests before Windows 7.
- It has some form of 3D pass through, but – if I remember correctly – uses an insecure approach that just lets the guest dump any and all commands to the host GPU.
- Using it on a Linux guest requires installing the guest additions because this adapter is not (yet) supported by the mainline Linux kernel.
- Only supports OpenGL 1.1 on 64bit Windows 10 and all Linux guests.

This option likely exists just to provide continuity – after upgrading to 6.0, all old VMs have this mode selected automatically so there's no unexpected change in behavior; you don't lose whatever acceleration you *previously* had.

2) VMSVGA

This emulates the VMware Workstation graphics adapter with the 'VMware SVGA 3D' acceleration method.

- Contrary to what the manual says, this is currently the default for Linux guests.
- It is supposed to provide better performance and security than the old method.
- This is supported by the mainline Linux kernel using the SVGA drivers.
- Supports OpenGL 2.1 on all Windows and Linux guests.
- It might also have the advantage of supporting old operating systems which had VMware guest additions available but not VirtualBox guest additions.

3) VBoxSVGA

This provides a hybrid device that works like VMSVGA (including its new 3D acceleration capabilities), but reports the same old PCI VID:PID as VBoxVGA.

- This is the default for Windows guests.
- The advantage of this mode is that you can upgrade existing VMs (which previously used VBoxVGA and had the VirtualBox Video driver installed) and they don't lose their graphics in the process – they still see the same device, until you upgrade the 'Guest Additions' at any later time to enable 3D acceleration.
- Also, because it's still VMware SVGA *emulated* by VirtualBox, choosing this option and using the VirtualBox driver may still have advantages over the VMware one, e.g. allow to make use of VirtualBox-specific additional features.

Source: https://superuser.com/questions/1403123/what-are-differences-between-vboxvga-vmsvga-and-vboxsvga-in-virtualbox

7.4 Recording your Virtual Machine screen

To record the screen of your Virtual Machine, use a simple tool called: “SimpleScreenRecorder”. This application was used to record the video clips for the YouTube Channel of this project!

First you need to install SimpleScreenRecorder. Then you are going to increase the size of the cursor which comes in handy when recording the screen and it also helps the viewer follow what you do! After the installation a short description is given on how to use SimpleScreenRecorder.

7.4.1 Installing SimpleScreenRecorder

SimpleScreenRecorder is installed by performing the following steps:

1) Add the SimpleScreenRecorder ppa to the systems repository list by running the following command:

   `sudo add-apt-repository ppa:maarten-baert/simplescreenrecorder`

2) Update the local package database by using the following command:

   `sudo apt update`

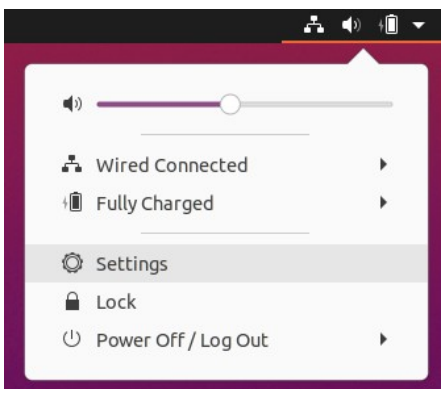
3) Install SimpleScreenRecorder using the following command:

   `sudo apt install simplescreenrecorder`

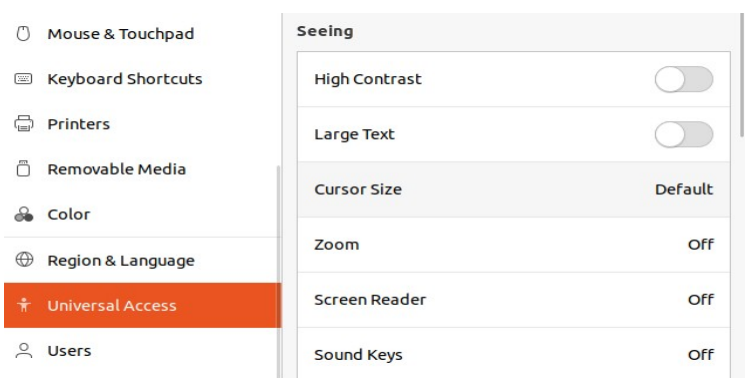
7.4.2 Increasing the cursor size

Before recording the screen of you Virtual Machine, it’s recommended to increase the size of your cursor. This is done by performing the following steps:

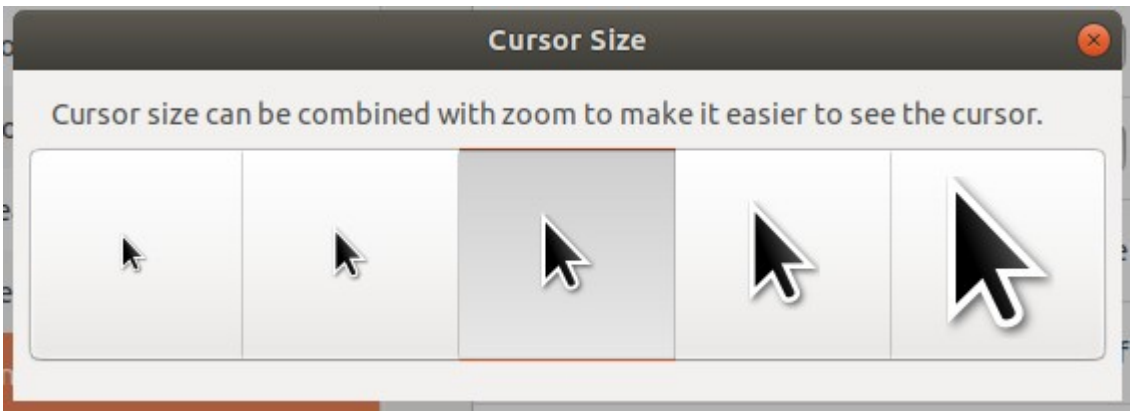
1) Open the settings menu in the top right of you Ubuntu Virtual Machine as shown in the illustration below.



2) Select Universal Access in the left menu of the screen that pops up. Then select Cursor Size as shown in the illustration below.



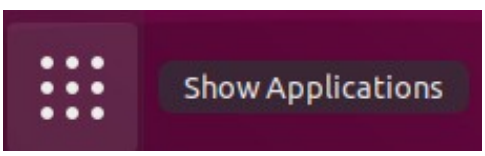
3) Then select the size of the cursor which suits your needs as shown in the illustration below.



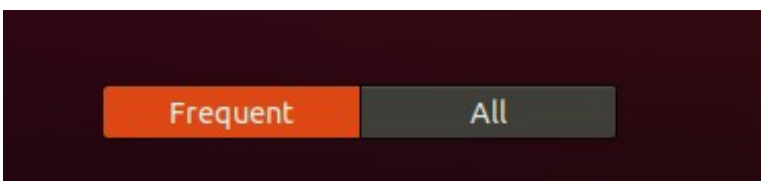
7.4.3 Using SimpleScreenRecorder

Opening the SimpleScreenRecorder application is done by performing the following steps:

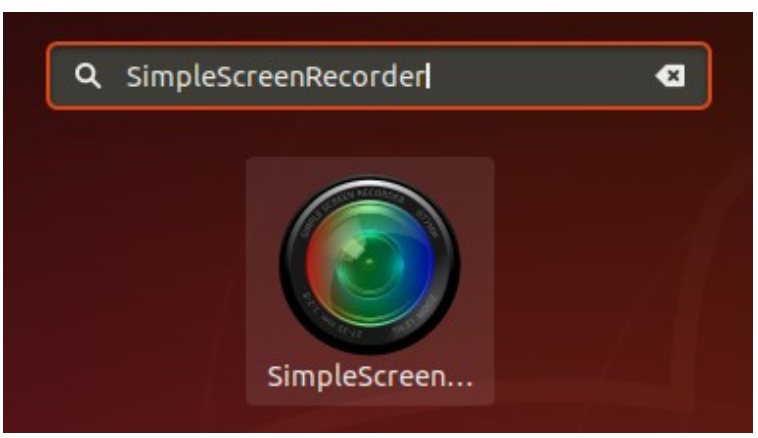
1) Open the show application menu on the bottom left of the desktop. The icon is shown in the illustration below.



2) Then select All at the bottom of the screen that pops up, as shown in the illustration below.

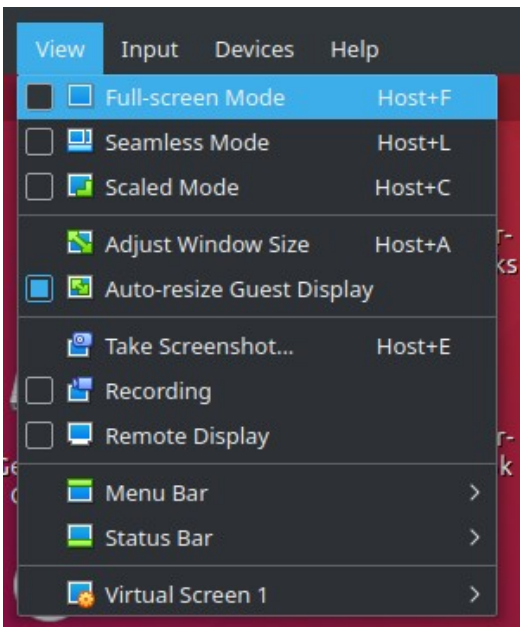


3) Enter “SimpleScreenRecorder” in the search bar and click on the icon that shows up, as shown in the illustration below.

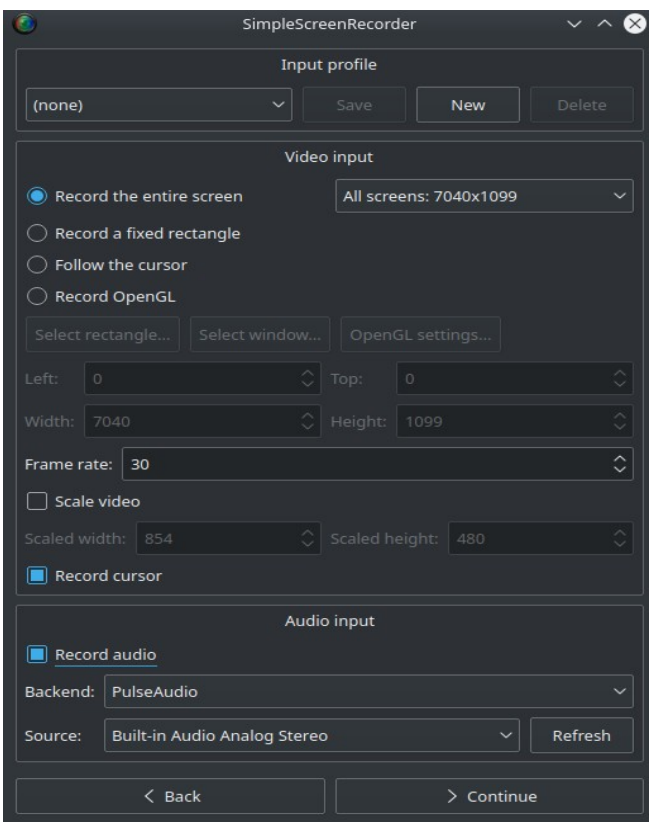


To use the SimpleScreenRecorder application you should perform the following steps:

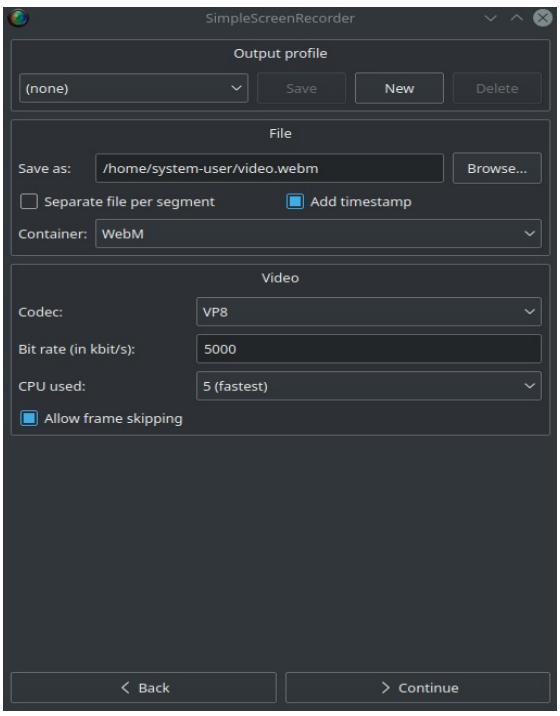
1) When recording your Virtual Machine window it's best to set your Virtual Machine window to full screen. This is done by clicking on View → Full-screen Mode in the VirtualBox window. How this is done is shown in the illustration below.



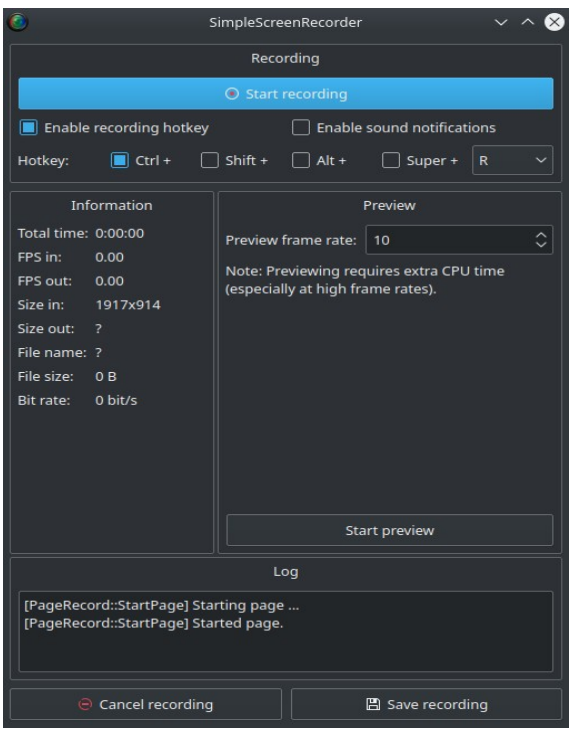
2) Edit the settings as you see fit. It's recommended to set the frame rate to 60 Frames Per Second. If you want to record with audio you should check the box: "Record Audio". Select continue after editing the settings.



3) In the next screen you also should edit the settings as you see fit. Here you can select where you want to save the video file after the recording process if finished. You can also set the extension type of the video. It's recommended to set it to: "webm" since this is a small video format which can be played in a web browser.



4) Next click on: "Start Recording" which will start the recording process. If you are done recording you can press on Pause recording and then on save recording to save the video.



When the screen is being recorded the red 'recording light' icon shown in the illustration below will pop up in the top right of your screen.

8 Useful Weblinks

8.1 Data analyses links and tools

Information related to the ETL-Process:
https://nl.wikipedia.org/wiki/Extraction,_Transformation_and_Load
Converting CSV to GeoJSON
http://www.convertcsv.com/csv-to-geojson.htm
Converting GPX to GeoJSON
https://mygeodata.cloud/converter/gpx-to-geojson
Converting JSON to GeoJSON
https://stackoverflow.com/questions/5620696/convert-lat-long-into-geojson-object

8.2 MongoDB Links

The MongoDB website:
https://www.mongodb.com/
The MongoDB documentation:
https://docs.mongodb.com/
An example of Python with MongoDB and MongoAtlas
https://www.youtube.com/watch?v=3ZS7LEH_XBg
An example of Python-Flask with MongoDB and Docker
https://www.youtube.com/watch?v=AAPOCB1U1kg

8.3 PostgreSQL / PostGIS and GeoServer Links

Installing PostgreSQL with Postgis and PGAdmin4 with GUI:
https://freegistutorial.com/how-to-install-postgis-on-ubuntu-18-04/
Installing PostgreSQL with Postgis and PGAdmin4 without GUI:
https://www.paulshapley.com/2018/11/how-to-install-postgresql-10-and.html?m=1
Installing GeoServer and running it as service:
https://github.com/jncc/web-mapper-core/wiki/Tips-for-installing-geoserver-on-Ubuntu-16.04
Installing GeoServer and running it as service, and managing the service:
https://gismentor.com/index.php/2019/05/08/installing-geoserver-in-linux-ubuntu-gismentor/
An example of Python-Flask with PostgreSQL and PostGIS
https://github.com/ryanj/flask-postGIS/tree/master/templates
An example of Python-Flask with PostgreSQL and Docker
http://fuzzytolerance.info/blog/2018/12/04/Postgres-PostGIS-in-Docker-for-production/

8.4 NGINX ModSecurity Links

Installing the latest version of NGINX:
https://www.linuxbabe.com/ubuntu/install-nginx-latest-version-ubuntu-18-04
Installing ModSecurity for NGINX:
https://www.linuxjournal.com/content/modsecurity-and-nginx
Installing the ModSecurity Core Rule Set
https://www.linuxjournal.com/content/modsecurity-and-nginx
The ModSecurity Github Repository:
https://github.com/SpiderLabs/ModSecurity

8.5 Python-Flask Links

An example of Python-Flask with PostgreSQL and Docker
http://fuzzytolerance.info/blog/2018/12/04/Postgres-PostGIS-in-Docker-for-production/
An example of Python with Folium, Leaflet and Pandas
https://www.youtube.com/watch?v=4RnU5qKTfYY
An example of Python-Flask with PostgreSQL and PostGIS
https://github.com/ryanj/flask-postGIS/tree/master/templates
An example of a Full-Stack Python Flask application with Docker and React:
https://medium.com/@riken.mehta/full-stack-tutorial-flask-react-docker-ee316a46e876
An example of Python with Leaflet and D3
http://adilmoujahid.com/posts/2016/08/interactive-data-visualization-geospatial-d3-dc-leaflet-python/
An example of Python Flask with Leaflet
https://github.com/adwhit/flask-leaflet-demo/tree/master/templates
An example of Python-Flask with MongoDB and PyMongo
https://devinpractice.com/2019/03/25/flask-mongodb-tutorial/
An example of Python with MongoDB, PyMongo and MongoEngine
https://pythonise.com/feed/python/mongodb-python-mongoengine-pt1
An example of Python with GeoJson, Folium and Leaflet
https://youtu.be/clP6W7W79MM
An example of Python-Flask with Leaflet:
https://www.freelancer.com/projects/javascript/flask-leaflet-application-that-displays/
2 websites with very useful information related to visualizations with Python:
https://pyviz.org/ and http://holoviz.org/

8.6 TileStache Tile server Links

The official TileStache documentation:

http://tilestache.org/doc/

An example of TileStache with Gunicorn and NGINX:

https://digital-geography.com/set-tileserver-using-tilestache-gunicorn-nginx/

8.7 Cesium Terrain Server Links

An example of creating a Cesium Terrain Server:

https://bertt.wordpress.com/2016/12/08/visualizing-terrains-with-cesium/

Installing Memcached for NGINX:

https://websiteforstudents.com/setup-memcached-on-ubuntu-18-04-16-04-with-nginx-and-php-7-2/

Creating an Cesium Terrain Server example:

https://www.aiwebnetwork.com/blog-detail/install-setup-cesium-terrain-server-on-ubuntu

8.8 Angular Links

Official AngularJS website:

https://angularjs.org/

Creating an Angular Application example:

https://angular.io/tutorial

Cleaning your Angular Project:

https://itnext.io/clean-code-checklist-in-angular-%EF%B8%8F-10d4db877f74

8.9 OpenLayers Links

Official OpenLayers website:

https://openlayers.org/

Official OpenLayers documentation:

https://openlayers.org/en/latest/doc/

Running OpenLayers Locally example:

https://wiki.openstreetmap.org/wiki/OpenLayers_Local_Tiles_Example

Running OpenLayers with OpenStreetMap and OpenSeaMap Locally:

https://openlayers.org/en/latest/examples/localized-openstreetmap.html

Example OpenLayers with a tile server

https://switch2osm.org/using-tiles/getting-started-with-openlayers/

OpenLayers Cesium implementation:

https://openlayers.org/ol-cesium/

8.10 Cesium Links

Official Cesium website:

https://cesium.com/

Official Cesium documentation:

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

Cesium coding examples:

https://sandcastle.cesium.com/

8.11 Leaflet Links

Creating a Leaflet application using a tile server:

https://switch2osm.org/using-tiles/getting-started-with-leaflet/

Creating a Leaflet slider example:

https://github.com/dwilhelm89/LeafletSlider

An example of Python with Leaflet and D3:

http://adilmoujahid.com/posts/2016/08/interactive-data-visualization-geospatial-d3-dc-leaflet-python/

An example of Python Flask with Leaflet:

https://github.com/adwhit/flask-leaflet-demo/tree/master/templates

An example of Python with GeoJson, Folium and Leaflet:

https://youtu.be/clP6W7W79MM

An example of Python-Flask with Leaflet:

https://www.freelancer.com/projects/javascript/flask-leaflet-application-that-displays/