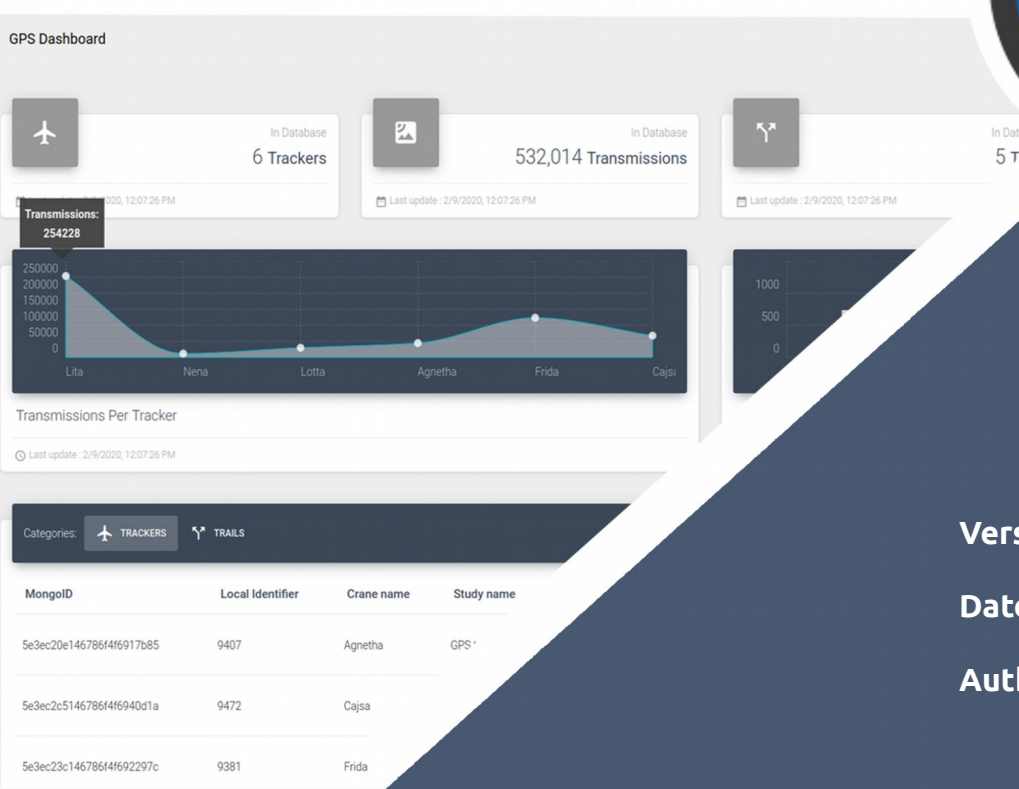
Cookbook

Creating the GeoStack

Course VM



Version : 1.0

Date : 08-04-2020

Author : The GeoStack Project

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.

This cookbook can be seen as a guide to creating the Complete Open Source Geographical Software Stack.

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 Geographical 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.

Prerequisite 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 we are going to create a new Ubuntu Virtual machine in which we are going to install the software and tools required for the GeoStack.

As mentioned before; 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, programming manuals and presentations provided in the folder: “Building-the-VM-from-scratch-using-the-manuals”. Both folders are located in the GeoStack-Course Github Repository which we will be cloning later on!

But more on that later! **Let’s start of with an introduction to the GeoStack.**

Table of Contents

1. Introduction to the GeoStack

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 without an active network connection.

The README.TXT file provided in the root folder of **The GeoStack Project** gives a global description of what you will learn during this course and the functionalities which the Geographical software stack has.

The geographical software stack can be divided in 6 parts which are as follows:

1) **Virtualization software:**

   Each Geostack component is Dockerized and thus can run separately from the other components. The components are also installed in one Virtual Machine so everything can run locally on one machine.

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 we 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 we 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 webservers and an API. The middleware contains the Tilestache Tileserver, Cesium Terrain Server, Flask-API and the NGINX-Webserver. The middleware is where all the “magic” happens.

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.

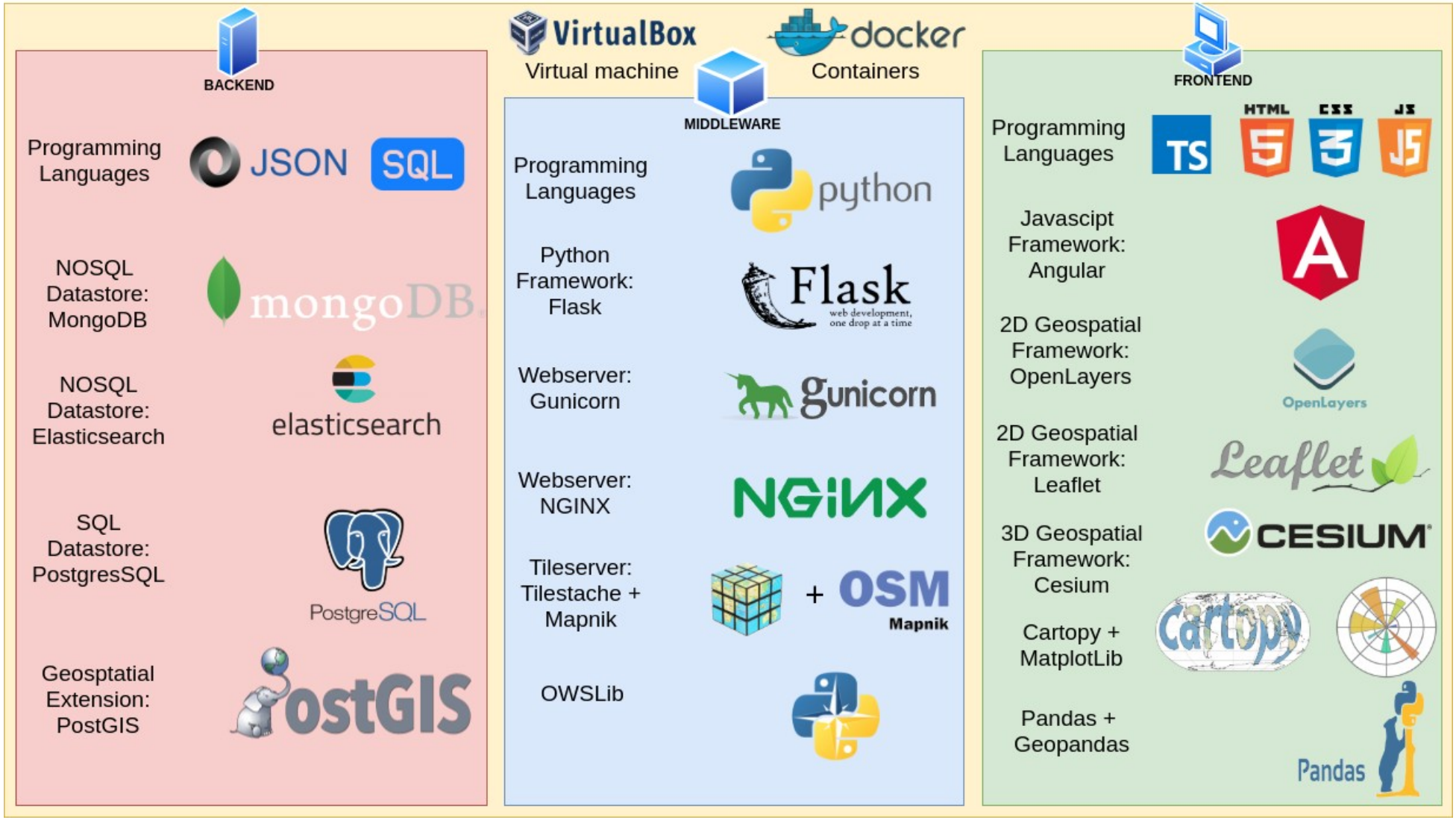
We are going to install the software in the order of the parts mentioned above.

In the table on the next page you can find an overview of existing Geographical software stacks and geospatial software products.

Categorie	Products and Providers
Open Source	❖ OSGeo.org: Publishes the OSGeo live DVD with well-known Open Source Geoproducts. ❖ OpenGeospatial.org: An international consortium of 530 companies around the world for Open Standards in geospatial. ❖ Geonovum: An advice foundation and knowledge center from and for the Dutch Government. ❖ Foss4G.org: An annual conference for Open Source geographic software products. Also called Foss4G.
Comercial	❖ ArcGIS Suite from ESRI: The leading standard in the world of commercial geographic software stacks. ❖ HexagonSpatial Suite: Contains the Luciad Suite, which is now the new competitor to the ArcGIS Suite. ❖ Planet.com: a spatial cloud service which includes the former Open Source suite from Boundless.
Mobile	◆ National Geospatial-Intelligence Agency has 3 SDKs: o AG NGAGeoINT IOS: A Github package for programming geographic applications for IOS. o AG NGAGeoINT Android: A Github package for programming geographic applications for Android. o AG NGAGeoINT Javascript: A Github package for programming geographic applications using Javascript. ◆ Ionic with Cordova: A development environment consisting of TypeScript and Angular.
Geospatial Datastores	◆ Postgres + PostGIS: An Open Source geographic extension for a PostgreSQL database. ◆ SQLite + SpatiaLite: SQLite is a single file datastore. SpatiaLite is the geographical extension of this database. ◆ MariaDB + Microsoft MySQL spatial: A geographical extension of a MariaDB database. ◆ Oracle Spatial DB: A commercial geographic extension of the Oracle SQL commercial relational database. ◆ Microsoft SQL Server: A commercial geographic extension for a MySQL database.
Geospatial SearchEngines	◆ Lucene: This is Apache's Open Source Core Search Engine with spatial search functions. ◆ Solar: This is the Open Source search engine of the Apache foundation used by Lucene. ◆ ElasticSearch: An Open Source geographic search engine.

1.1 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.



1.1.1 The Virtualization software

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.

1.1.2 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.

1.1.3 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**

1.1.4 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.

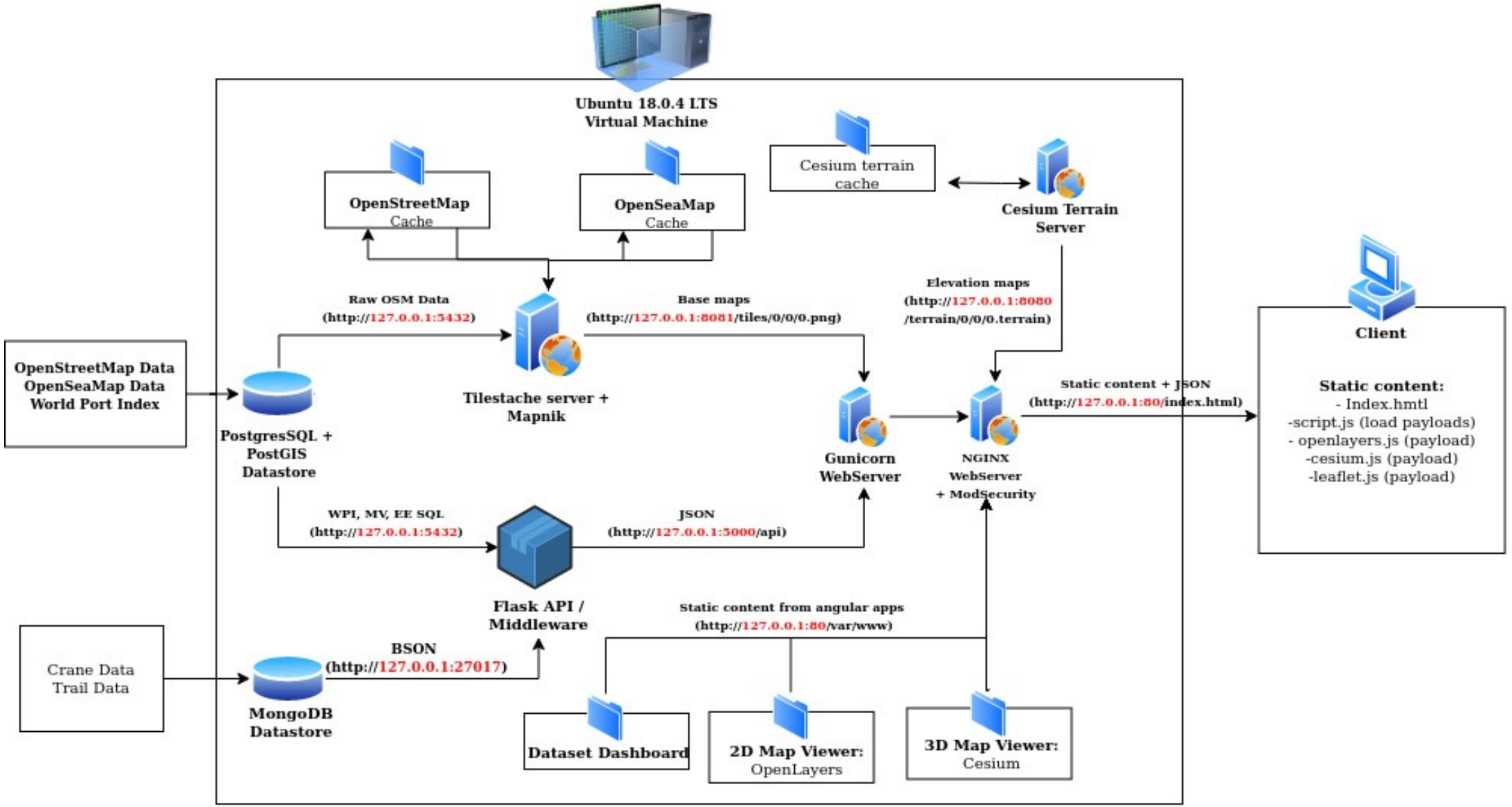
CesiumJS is an open source JavaScript library for creating world-class 3D globes and maps with the best possible performance, precision, visual quality, and ease of use.

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.

1.2 The GeoStack components

The diagram below shows the positioning of each component in the GeoStack. An explanation concerning the diagram and the components shown in the diagram is given during this section. A video is also available which gives a global introduction related to the GeoStack components and the web applications which are created during the GeoStack Course. These videos can be found in the folder: “The-Geostack-Project/GeoStack-Demo-Videos/”. For a better understanding of what your about to creating during this course you should watch these video’s.



Below you can find a textual description, per component, of what you learned in the GeoStack Demo video's.

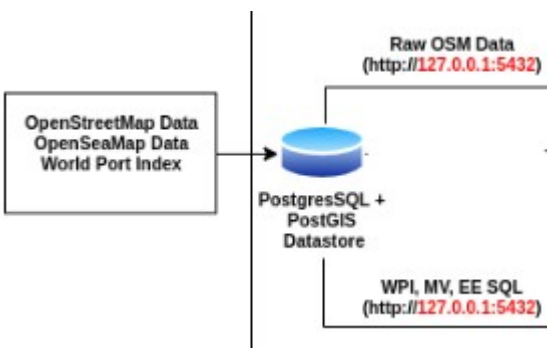
1.2.1 The PostgreSQL datastore

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 our Tileserver.

- **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 ports around the world. The dataset contains data related to the type of port or how big a port is and much more.

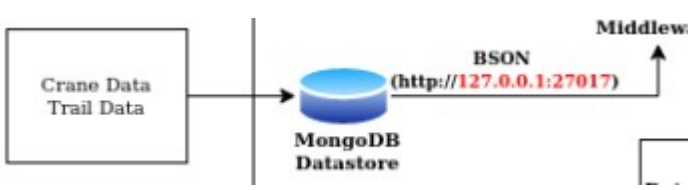


1.2.2 The MongoDB datastore

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.



1.2.3 The Tilestache Tileserver

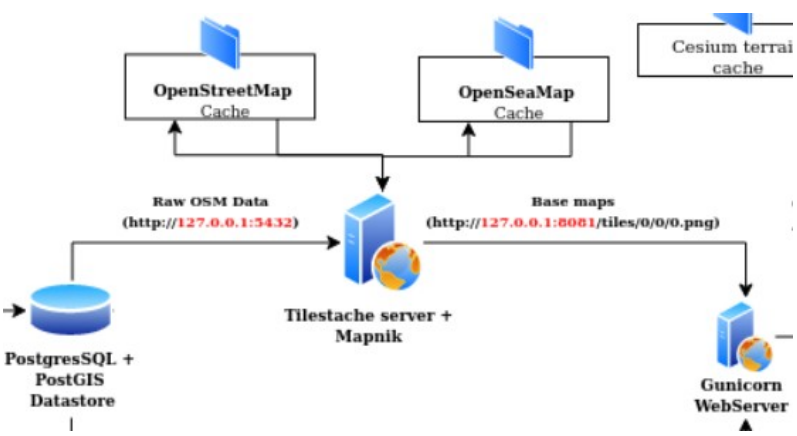
A tile server is the server that generates rendered images (tiles) from a database. There are multiple services online that provide a Tileserver. The problem with using online services is that the Tileserver will not work when you don't have a n active network-connection. This is also the reason why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

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

If you want to create a Tileserver 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/.

We are going to create a Tileserver using the lightweight webserver: "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).

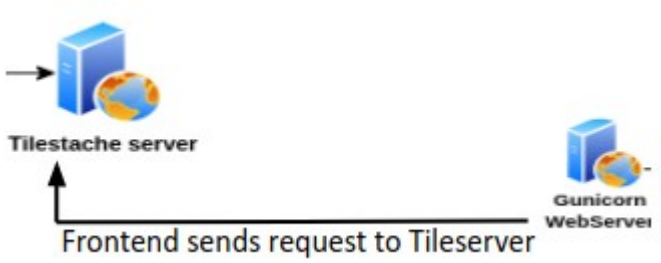
In the diagram below you find the structure and dataflow of the Tileserver in our GeoStack.



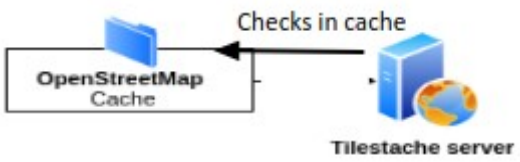
The way the Tilestache Tileserver works is as follows:

1) The frontend sends a so called "{Z}{X}{Y}" request to our Tileserver. 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 the 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.

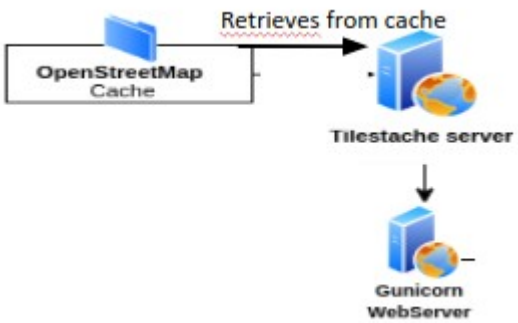
   These values are then used to only return the tiles than need to be displayed in the web application instead of the complete map.



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



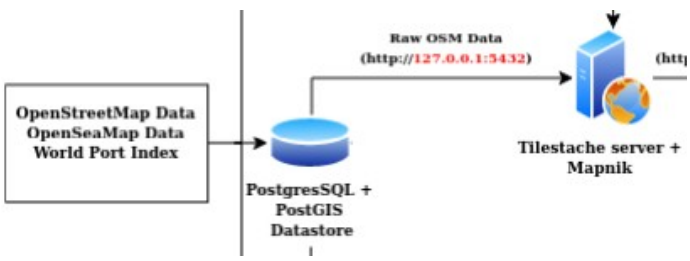
3) If the tiles have been generated before the Tileserver will extract these tiles from the cache and send them back to the frontend (Our web applications).



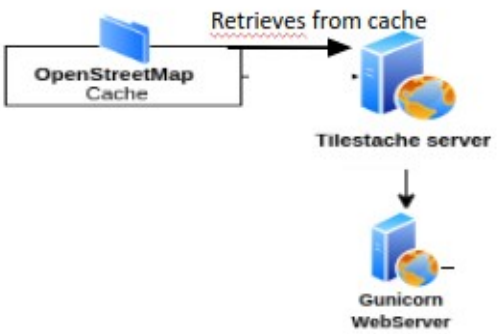
4) If the tiles have **NOT** been generated before the Tileserver will request the RAW OSM data from the PostgreSQL datastore after which Mapnik will generate the requested Tiles using the OpenStreetMap-Carto style sheet.

   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 stylesheet. The style sheet will then for example give all the nodes containing the tag:”water” the color blue.



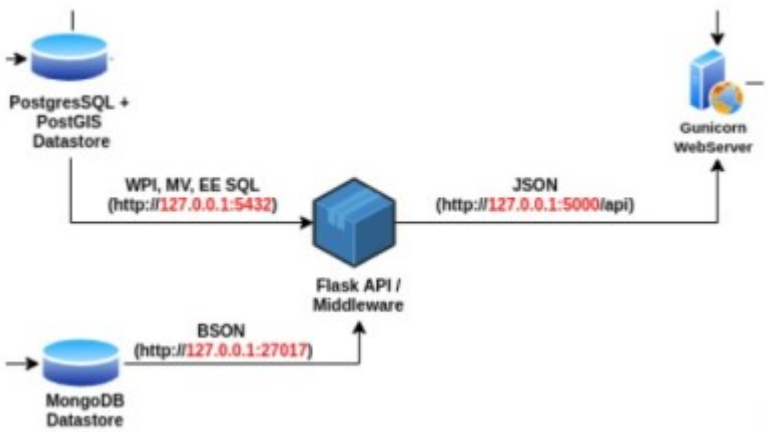
5) After the requested tiles have been generated they will be stored in the cache belonging to the requested map after which they are send to the frontend. When the tiles are requested a second time they don’t have to be generated anymore. This makes the loading of a map a lot faster.



1.2.4 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)
```

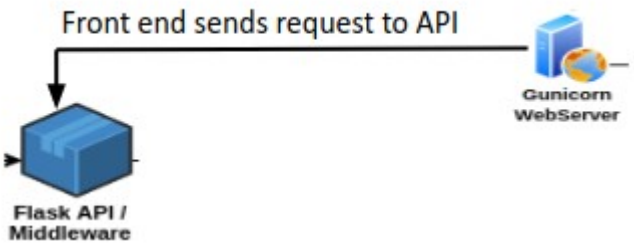
The name of the function in the illustration above is: "get_one_tracker()" and takes an ID as input parameter. The function is bound to the URL: /api/trackers/<id>.

If an request is send to this URL the function: "get_one_tracker()" is triggered and a query is executed on the MongoDB datastore. This query than returns the tracker data belonging to the tracker with the ID passed as parameter in the function.

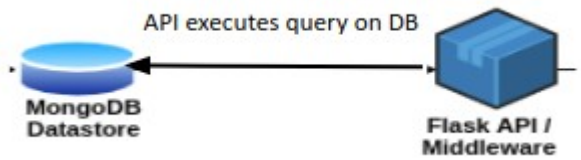
The API then sends the requested and obtained tracker data back to the frontend.

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.

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

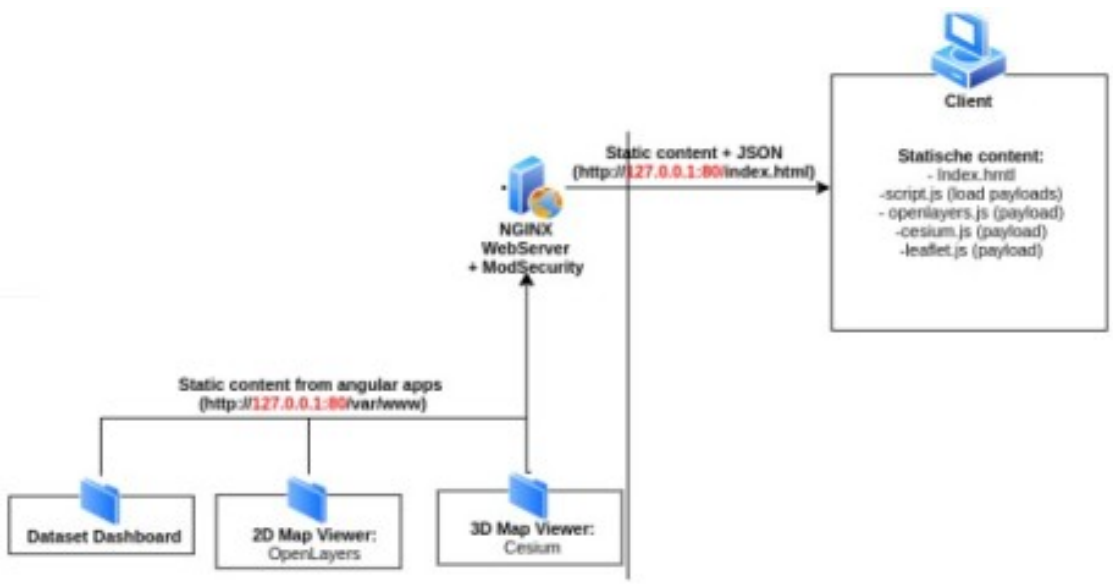
1.2.5 The NGINX Webserver

NGINX, pronounced "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:

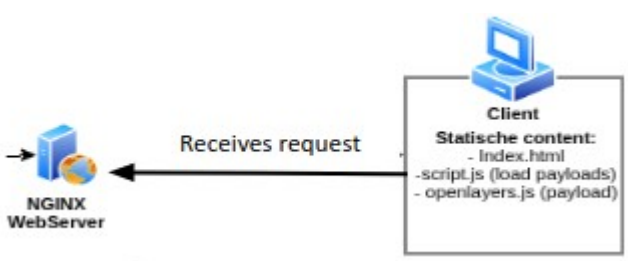
1) **A reverse proxy**
   A reverse proxy is useful if you have multiple web services listening on various ports and you need to reroute requests internally. This allows you to run multiple domain names on port 80 which is the port the web server is running on.
2) **Load balancing**
   Load balancing 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 webserver.

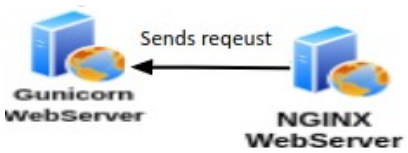


The NGINX webserver works as follows:

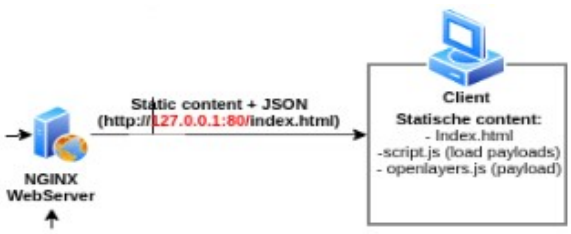
1) The webserver receives request from the client (User). A request could be the following:
   - The location of a Crane.
   - All GPS-Routes in the MongoDB datastore.
   - All Ports in the PostgreSQL datastore.
   - A map from our Tileserver.



2) The webserver then relays the request to the corresponding component in the Geostack. A corresponding component could be:
   - The Tileserver 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 webserver 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.

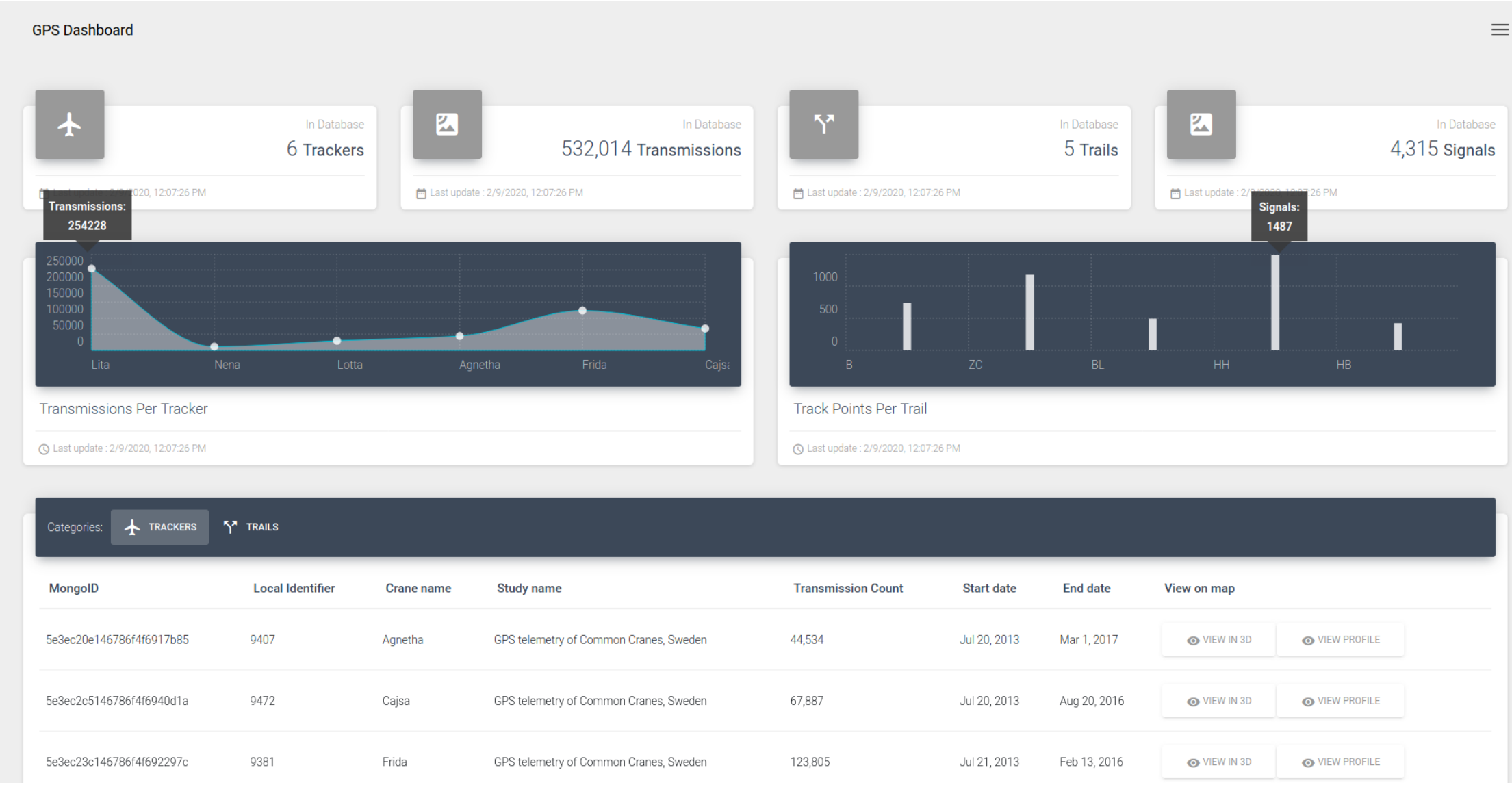


1.2.6 The Dataset Dashboard

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. The dashboard 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 Ports 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.



1.2.7 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 we add data to our datastores the data will automatically be updated.

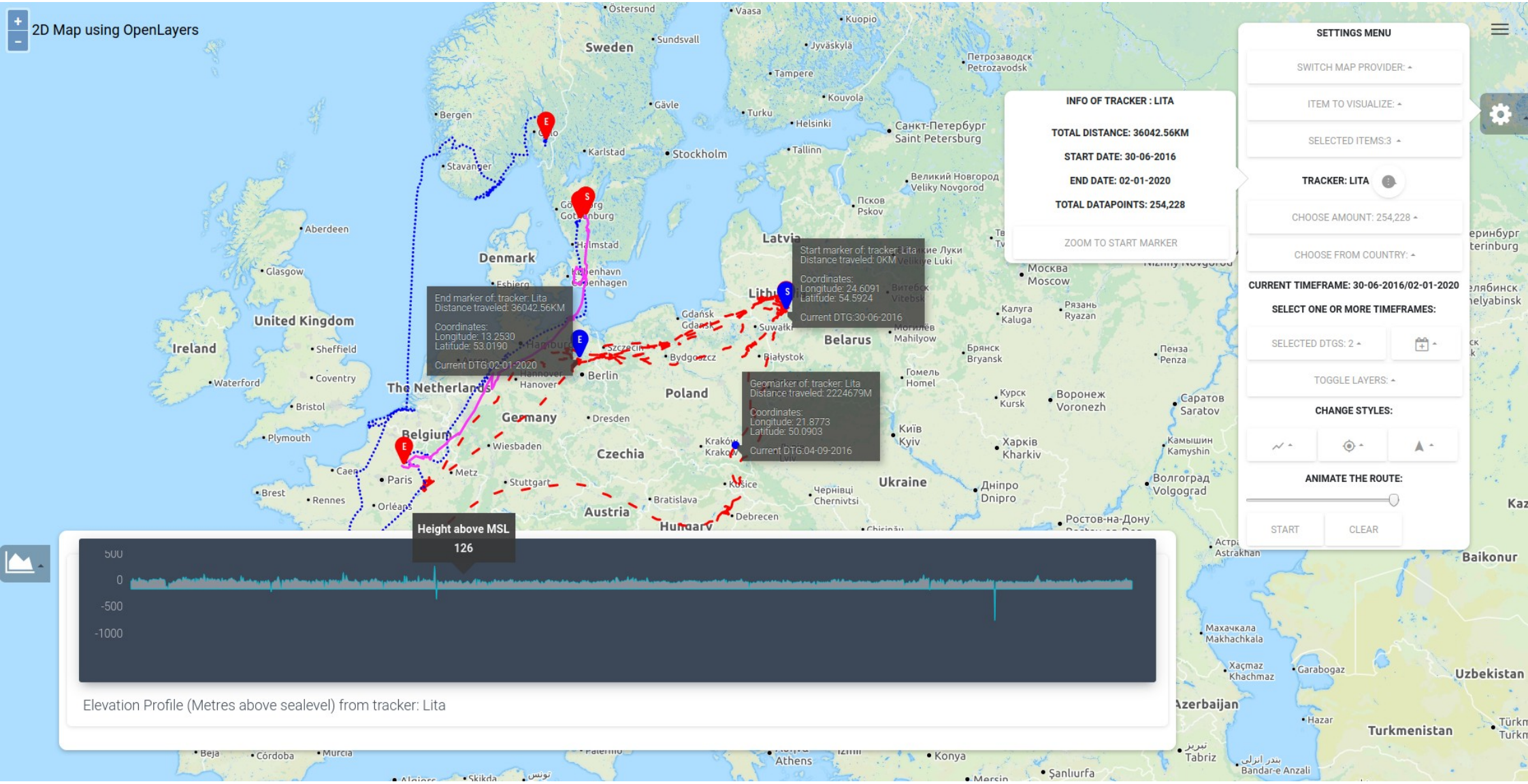
The functionality of this application can be found in the table below.

#	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 Tileserver.
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 a multiple visualized Crane tracker routes using the 2D Map viewer created with the Geospatial Framework: “OpenLayers”.



1.2.8 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.

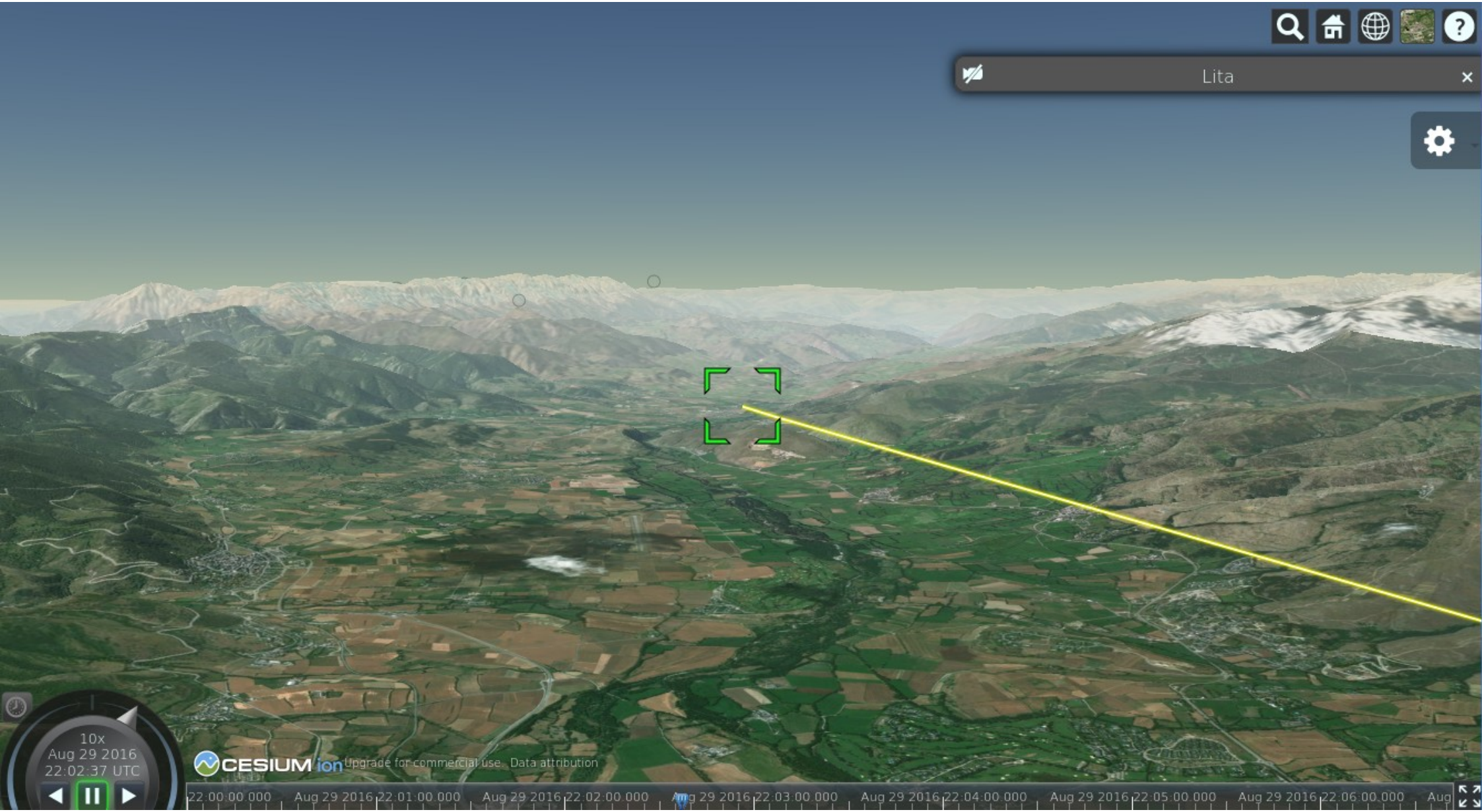
All the data shown in the 3D Map Viewer is “live” which means that when we add data to our datastores the data will automatically be updated.

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 Tileserver.
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 Crane tracker Route 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.

3. Installing the GeoStack

Before we can start with the installation process of the GeoStack software, tools and applications we first have to create a new Virtual Machine and install an Ubuntu Operating System. 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:

- ➔ **Creating a new Virtual Machine and using the installation scripts to install all the software and applications (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.

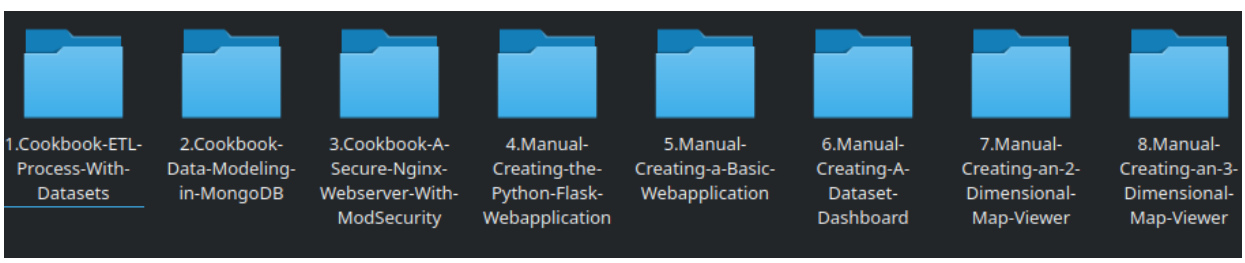
  If you choose to install the Geographical software stack using this technique you have to read this chapter (3) and chapter 4.

- ➔ **Build the Geostack from scratch using this document, the Jupyter Notebooks, the cookbooks, the programming manuals and the presentations (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.

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.

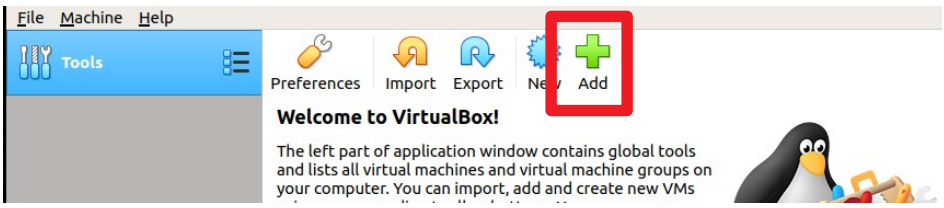
As mentioned above if you choose to install the GeoStack using the installation scripts you should read chapter 4 and If you choose to install the GeoStack manually, you should skip chapter 4 and read from there on.

3.1 Creating the Virtual Machine in VirtualBox

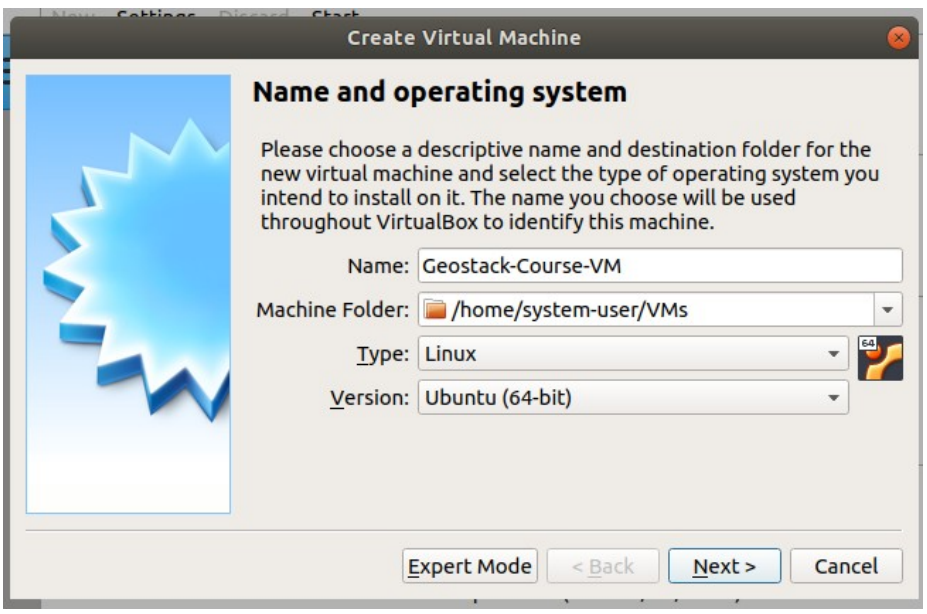
At this point you have VirtualBox on your system. Now we need to install the GeoStack Course Virtual Machine. To be able to install the GeoStack Course VM we first need to create a new Virtual Machine. After we have created the Virtual Machine we are going to install an Ubuntu ISO file.

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

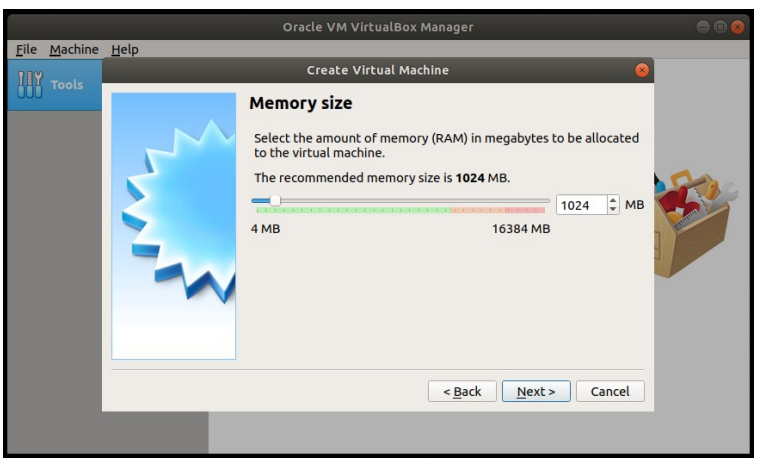
1) Click on the button: “new” in the VirtualBox start screen.



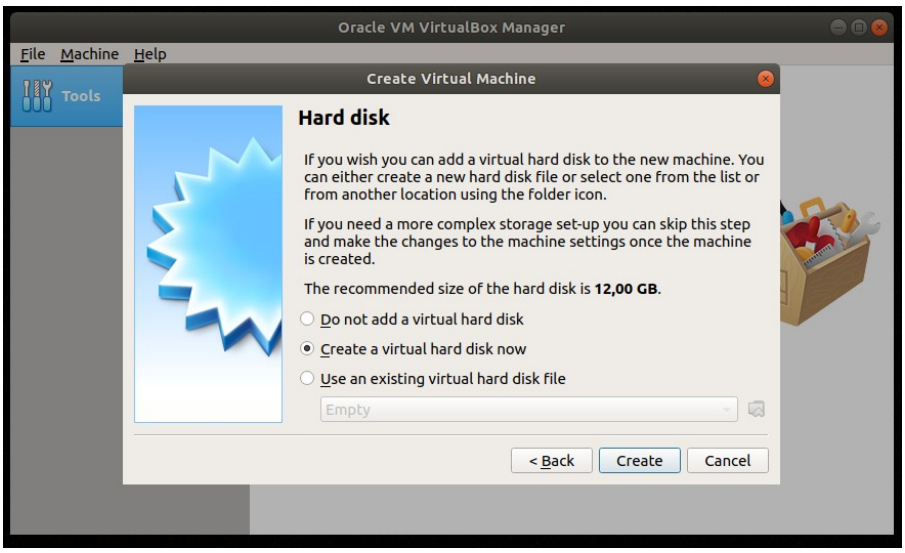
2) Give the Virtual Machine a suitable name in this case we give it the name: “Geostack-Course-VM”.



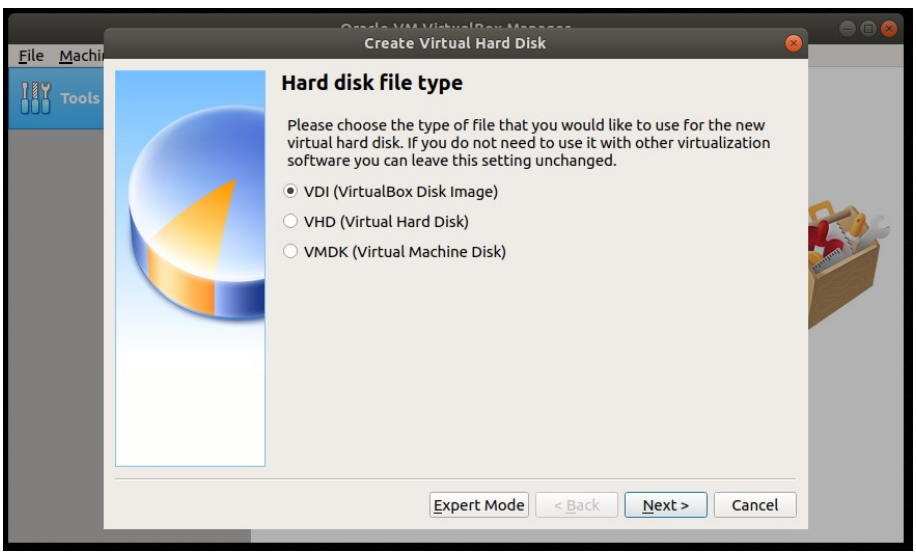
3) Assign the amount of RAM you want the machine to use. the minimal amount of RAM is 1 GB (1024MB) for a VM. If your PC has enough RAM the advice is to assign 3 GB (3072 MB) or 4 GB (4096 MB) RAM because Ubuntu will use up to 2 GB (2048 MB) already for itself.



4) Select create a virtual hard disk.

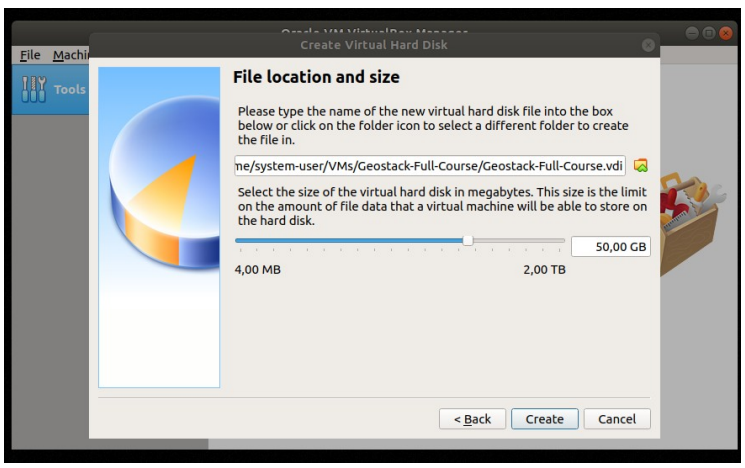


5) Set hard disk file type to VDI (Virtual Disk Image).

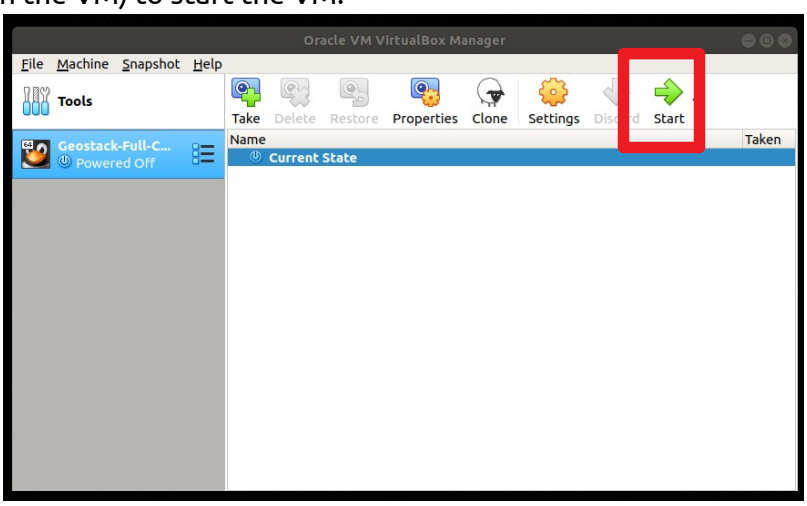


6) Select allocate disk space dynamically (Dynamically allocated). Tip: choose fixed disk size if performance matters to you.

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 we are choosing to set the disk size to 50GB is because increasing the disk size of an VDI file can be a 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.



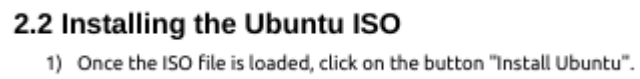
9) Select a startup disk which is the Ubuntu ISO that you have downloaded in the previous cookbook. Click on the yellow folder icon on the right to browse and select the ISO file.

3.2 Installing Ubuntu 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 related to the workshop 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 a newly created Virtual Machine 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**

   This chapter describes how you should connect you Virtual Machine to the network.

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

   This chapter describes how you should create a shared folder between your HOST and GUEST machine. The only difference, when creating a shared folder in for the GeoStack Course VM, is that in step 2 of section 2.4.3 you should select the Geostack-Course folder path instead of the Geostack-Workshop folder path.

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

3.3 Cloning the GeoStack Course Github Repository

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

We are to clone the Github repository, which contains the GeoStack Course files, by performing the following steps:

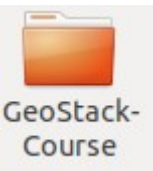
1) Open a terminal by pressing the Ctrl + alt + t on your keyboard.
2) Install GIT by using the following command: `sudo apt install git`
3) Clone The GeoStack-Course Github repository by running the following command:

   `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.



4. Installing GeoStack components AUTOMATICALLY

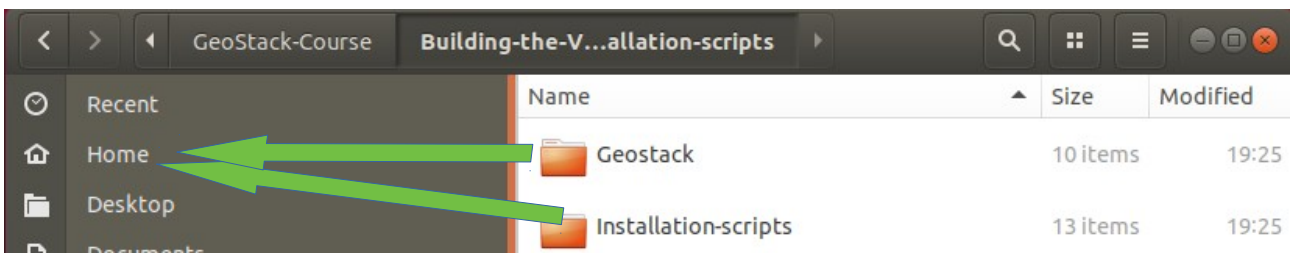
So you chose to install the GeoStack automatically? That's a good choice! If everything goes accordingly you will have a fully working Geographical software stack in less than 3 hours.

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.

- ➔ **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.



Now that we have the required files in our Virtual Machine we can start the installation process. First we need to clear some things up.

NOTE: INSTALLING THE SOFTWARE AUTOMATICALLY, USING THE INSTALLATION SCRIPTS, DOES NOT MEAN YOU DON'T HAVE TO PAY ATTENTION.

Once in a while you are asked for some input such as changing default passwords, accepting updates and adding package sources to our 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.
   ```

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

Now let's 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.

2) Run the first script by entering the command: `~/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 Python2 and Python-pip.
   ➔ Install Curl.
   ➔ Install GIT.
   ➔ Install Docker.
   ➔ Install Docker-compose.
   ➔ Install NodeJS.
   ➔ Install Atom.
   ➔ Add new sidebar shortcuts.

   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 the terminal again and enter the command: `~/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 2 minutes 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 we 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:
`~/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 we 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: `~/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.
   ➔ Clean unnecessary files.

   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 we can move on to the next script for importing all the datasets in the correct datastores. This is done by running the command: `~/Installation-scripts/5-dataset-import.sh`

   This script will do the following:

   ➔ Import, model and index the Crane(Tracker) datasets in MongoDB.
   ➔ 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 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 we can move on to the next script for installing the Flask-API software. This is done by running the command: `~/Installation-scripts/6-middleware-software-Flask.sh`

   This script will do the following:

   ➔ Install Python-Flask.
   ➔ Install Flask-Pymongo.
   ➔ Install Gunicorn3.
   ➔ Install BeautifullSoup 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 we can move on to the next script for installing the Tileserver software. This is done by running the command: `~/Installation-scripts/7-middleware-software-Tileserver.sh`

   This script will do the following:

   ➔ Install Gunicorn.
   ➔ Install Tilestache and Pillow.
   ➔ Install Mapnik.
   ➔ Install OpenStreetMap-Carto and download it's shapefiles.
   ➔ Install OSM2PGSQL.
   ➔ Install OpenSeaMap Renderer.
   ➔ Create the OSM PostgreSQL database: "gis" and add the PostGIS extensions.
   ➔ 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 we can move on to the next script for installing the Cesium Terrain server. This is done by running:
~/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 we can move on to the next script for installing the NGINX webserver. This is done by running:
~/Installation-scripts/9-middleware-software-NGINX.sh

This script will do the following:

➔ Install the latest NGINX version.

➔ Install ModSecurity.

➔ 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 we can move on to the next script for installing the frontend software required for the web applications. This is done running the command:
~/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 follow the complete GeoStack course.

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 we 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: ~/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 we 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".
~/Installation-scripts/12-post-installation-cleanup.sh

This script takes about 3 minutes to complete and clears about 1.7GB of disk space.

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.

5. Installing GeoStack components MANUALLY

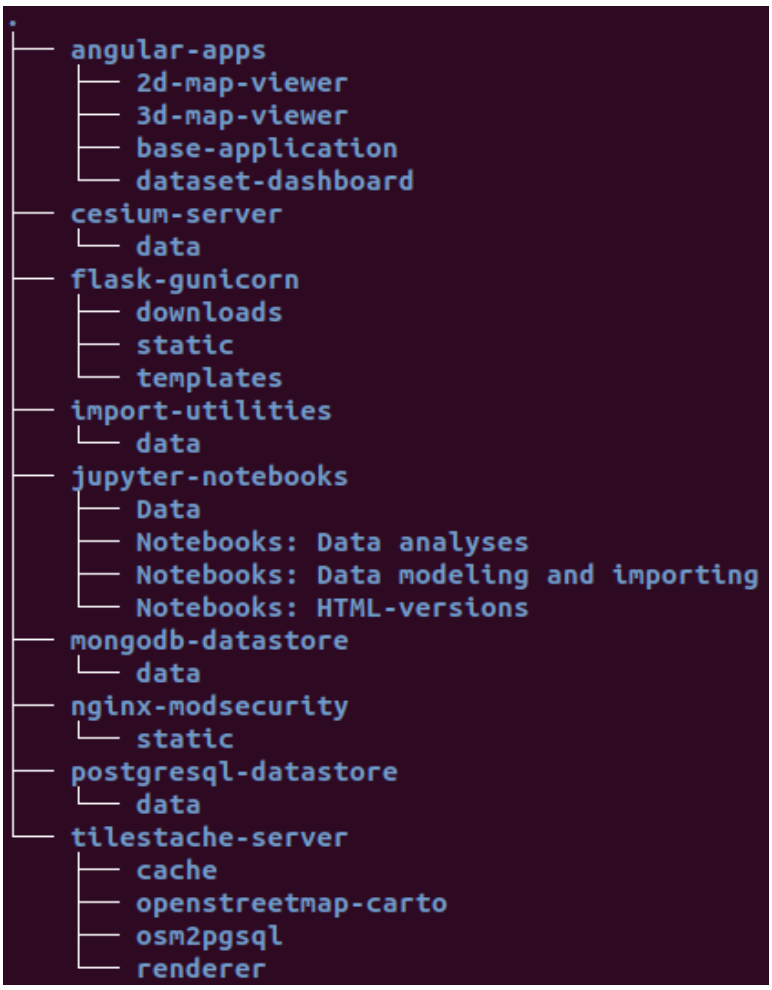
So you chose to install the GeoStack manually? That's a good choice! It's highly recommend you to clear some time in your schedule because his will be a lengthy but fun processes which will result in a lot of knowledge on subjects reaching from data analyses up to and including data visualizations using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

In the folder: "~/GeoStack-Course/Building-the-VM-using-the-installation-scripts/Geostack" you can find all the files which we 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!

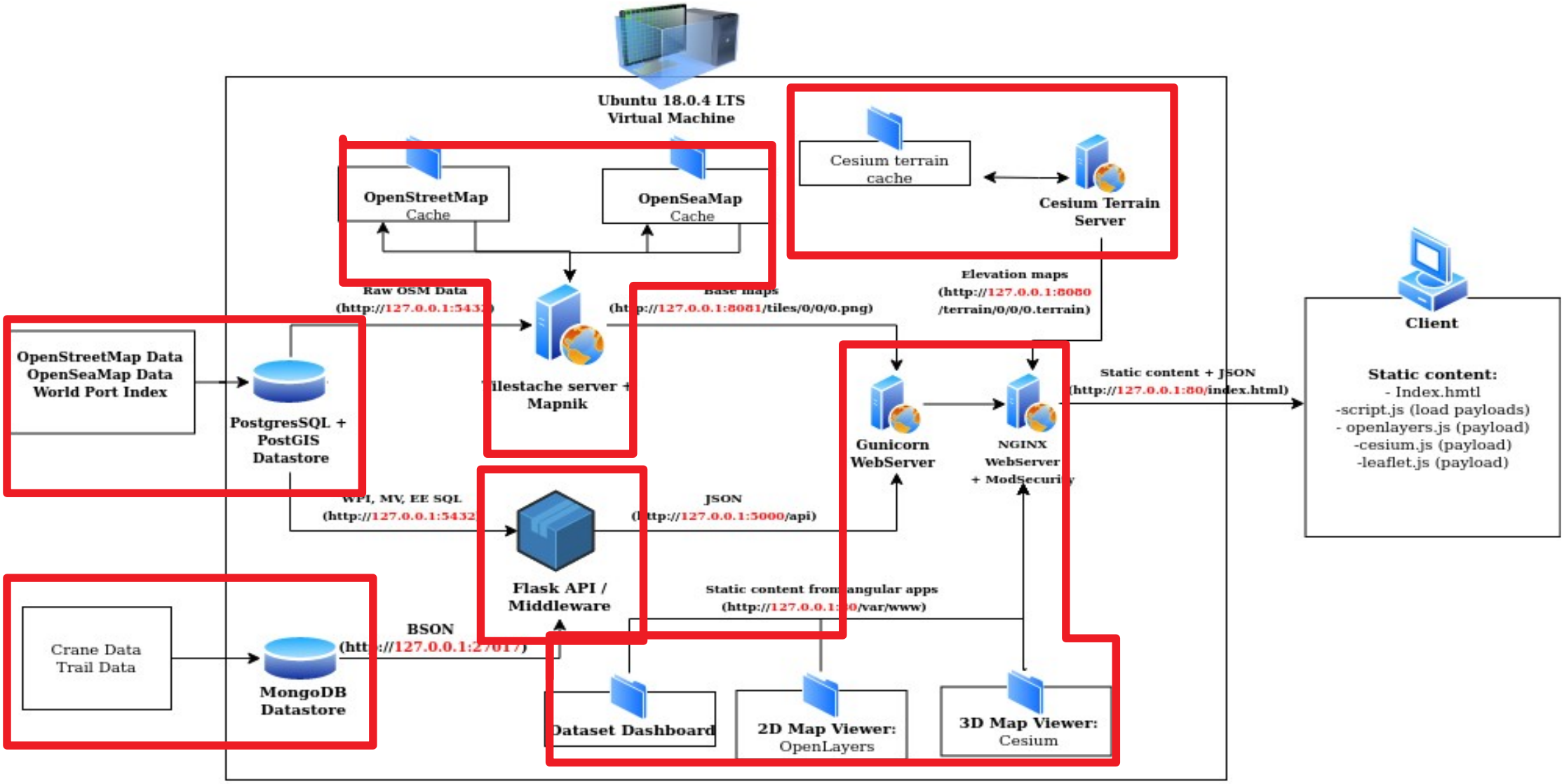
We are going to start of by creating the folder in which we are going to create and add all the GeoStack related files and folder. We create this folder by opening a terminal (Ctrl + alt + t on your keyboard) and running the command: `mkdir ~/Geostack`

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



5.1 Installing 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 compartmentalization you can for example run the PostgreSQL database on a system at home and the Tilestache Tileserver 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 we are going to install the tools required for the virtualization.



So let's install the virtualization software. We start with Installing the VMWare-Tools by running the following command: sudo apt install open-vm-tools-desktop

Install Docker by performing 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. We do 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

In the GeoStack folder, located in the Home folder of the VM, we are going to create a file called: 'docker-compose.yml'. This is done by opening a terminal and running the command:

touch ~/Geostack/docker-compose.yml

In this file we are going to add all the GeoStack components as services after creating them. This file is used to run all the docker containers / components at once by running the command: "docker-compose up" from the directory "Geostack" which is the root directory of the Geographical software stack.

In chapter 6 of this cookbook we are going to run all the GeoStack components as docker containers using the docker-compose.yml file. But more on this later, let's first install some general software and tools which are used to increase our workflow during the **Beginner Course Open Source Geospatial Programming for Data Scientists.**

5.2 Installing the general software

Install Bleachbit, which is a tool for keeping you system clean, by running the following command: sudo apt install bleachbit

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

Install Net Tools (A package for analyzing network statistics) : sudo apt install net-tools

Install LibreOffice Writer : sudo apt install libreoffice-writer

Install Python2 and Python-pip, which are required for the Tileserver, by performing following steps:

1) Install Python 2 by running the following command: sudo apt install python
2) Download the Python-pip install script by running the following command: curl -O -L https://bootstrap.pypa.io/get-pip.py
3) Install python-pip by running the following command: sudo python get-pip.py
4) Remove the install script by running the following command: rm get-pip.py

sudo apt install python-pip

Install Python3 and Python3-pip: sudo apt install python3-pip

Install GIT (A tool for cloning Github repositories) : sudo apt install git

Install Atom, which is the code editor used during the cookbooks, 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

Install NodeJS & NPM, which are used for the Tileserver 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. We do 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. We do this by running the following command: npm -v

5.3 Installing data-analyses software

Now let's install the tools required for the analyses 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 Desktop shortcut to start Jupyter Lab by running the following command:

   `touch ~/Desktop/Jupyter-lab.desktop`

   NOTE: During this course we are going to create a lot of desktop shortcuts which will be used to start and stop the GeoStack components. For more information related to creating desktop shortcuts you should read section 7.1 of this cookbook.

3) Add the following code to the 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`

Install Pandas and GeoPandas : `pip3 install pandas geopandas`

Install Ubuntu libraries required for Cartopy:

`sudo apt install libproj-dev proj-data proj-bin libgeos-dev`

Install Python packages required for Cartopy: `pip3 install cython`

Install MatPlotLib and Scipy: `pip3 install cartopy matplotlib scipy`

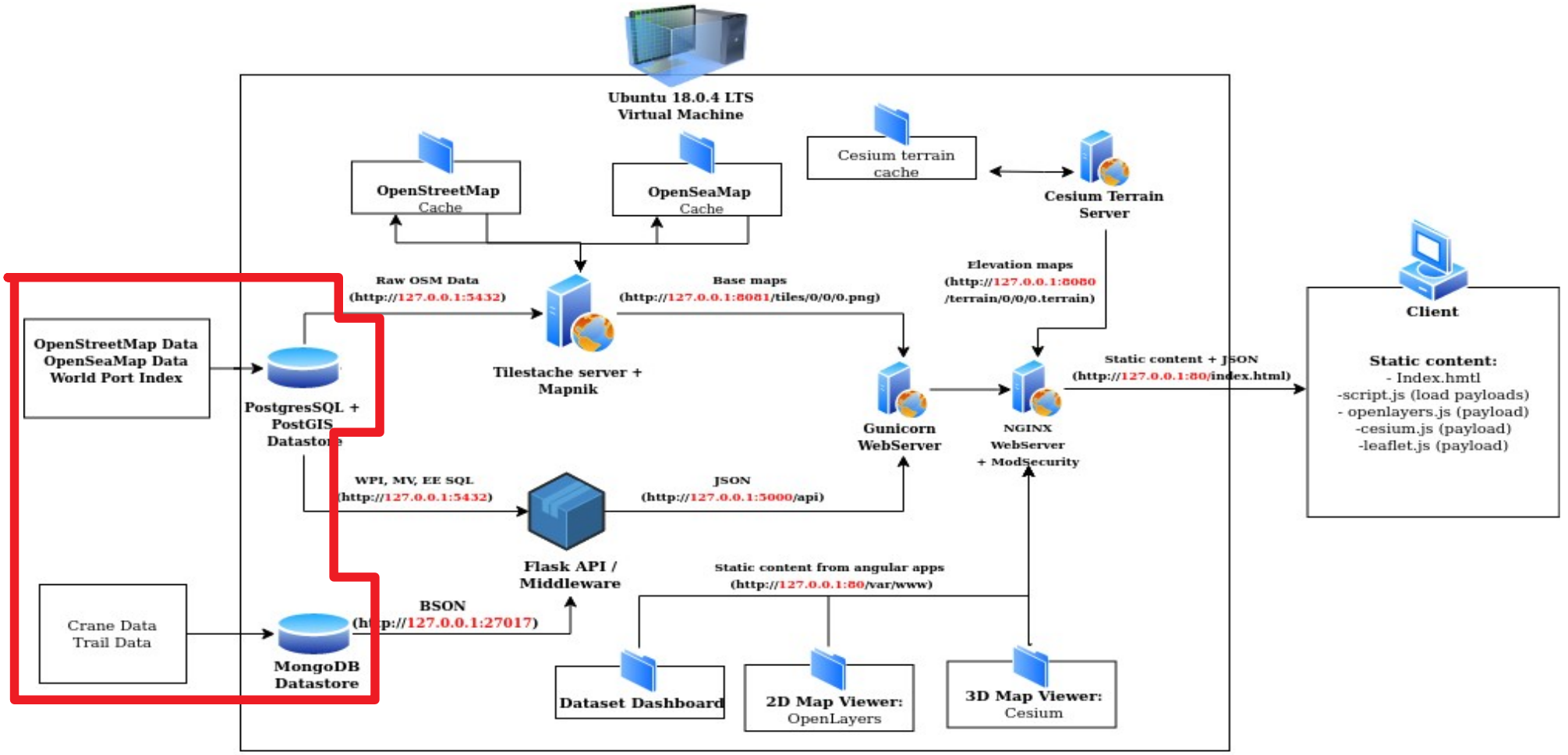
Install Cartopy: `pip3 install git+https://github.com/SciTools/cartopy.git --no-binary cartopy`

Install Python packages required for the GPX file format: `pip3 install gpxpy geopy numpy`

Install Pandas-Profiling: `pip3 install pandas-profiling`

5.4 Installing backend software

During the cookbooks we 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 we need the data analyses and processing software. During this section we 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 we 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: The things you will learn during 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 during the Course.**



5.4.1 Installing PostgreSQL

Install PostgreSQL by performing the following steps:

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

   `wget -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 add-apt-repository 'echo "deb [arch=amd64] http://apt.postgresql.org/pub/repos/apt/ `lsb_release -cs`-pgdg main"`

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

4) Install PostgreSQL using 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 command:

   `\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

installing PostGIS: 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

5.4.1.1 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. We do this by running the following command:

    mkdir ~/Geostack/postgresql-datastore/data

3) Next we are going to create a script which will run when the container is started for the first time. We are going to call this script: "initdb.-postgis.sh". We do this by running the following command: touch ~/Geostack/postgresql-datastore/initdb-postgis.sh
4) Add the following code to the file:

```
#!/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
```

This script 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. We do this by running the following command:

```
touch ~/Geostack/postgresql-datastore/Dockerfile
```

Now open the newly created Docker file which is located in the folder: "~/Geostack/postgresql-datastore/"

6) 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"
```

5.4.1.2 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 we 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.

5.4.1.3 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.

   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!

5.4.1.4 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
---------------------------+----------+----------+-------------+-------------+-----------------------
World_Port_Index_Database | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
gis                       | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
postgres                  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

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.

```
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. We are going to create this database when creating the Tilestache Tileserver 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". We already installed this application which can be started by clicking on the shortcut shown in the illustration below.

5.4.2 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
```

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

5.4.2.1 Dockerizing MongoDB

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, we do not have to create a Dockerfile because we don’t have to add any extra logic when creating the MongoDB Docker container.

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

   `cd ~/Geostack && docker-compose build  mongodb-datastore`

5.4.2.2 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 we 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.

5.4.2.3 Exporting the MongoDB Docker data volume

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

3) 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
   ```

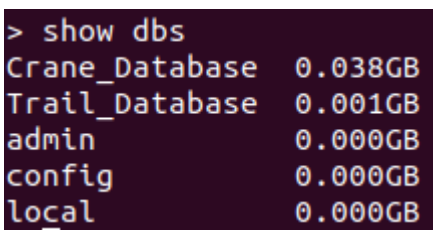
4) Zip the data folder so that the size is decreased which is useful when distributing the MongoDB Docker data volume to other systems.

5.4.2.4 Managing the MongoDB Databases

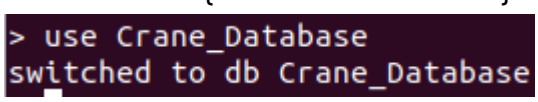
If you want to remove a database from you MongoDB instance you should perform the following steps:

1) Open a terminal and enter the command: mongo. This will open a MongoDB CLI as shown in the illustration below.

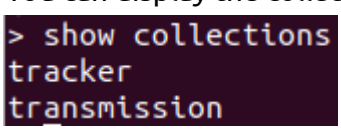
2) Then we want a list of all the databases on our system. This is done by entering the command “show dbs”.



3) Then look for the name of the database which you want to remove and enter the command: use {name of the database}



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



5) Then enter the command: db.dropDatabase() which will then remove the database.

If you want to clean (remove everything from) your local instance of MongoDB you should run the command:

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

WARNING: This will remove all the content and data from you MongoDB which is as follows:

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

Now that we have all the tools, libraries and software products required for our data-analyses we can start performing the data-analyses on the datasets.

The cookbook: “ETL-Process with datasets” gives in-depth information related to performing the ETL-Process using Crane (Tracker) dataset as example. The data analyses steps which we are going to perform are the first 2 steps of the so called: “ETL-Process”. So let’s start reading the cookbook: “ETL-Process with Datasets”.

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

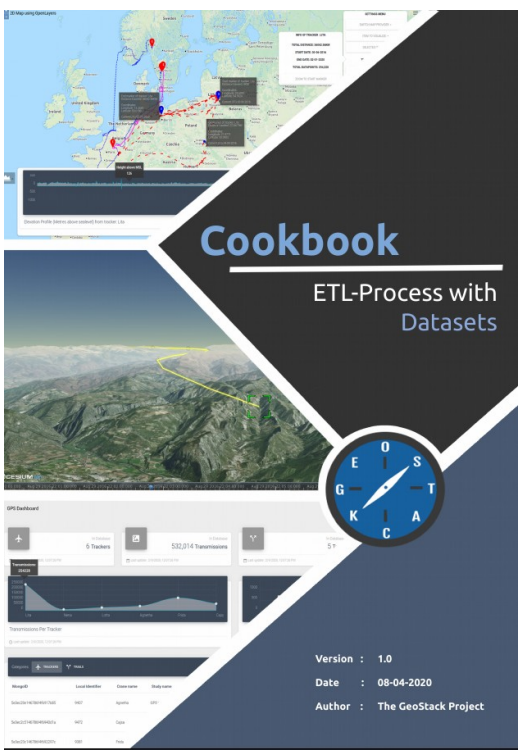
- ➔ How to extract data from a data source;
- ➔ How to filter data for the application goals;
- ➔ How to transform data to the file formats JSON and GeoJSON.

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 related to the data-analyses 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 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.



After you have finished reading this cookbook and the remaining Jupyter notebooks that come with it, you should continue reading this cookbook.

Now that we have analyzed and transformed the datasets, you should start reading 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, which was used in the cookbook: “ETL-Process-with-datasets”, as 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: “Data-Modeling-in-MongoDB-using-MongoEngine”

In the folder of the cookbook you will also find a folder called: “Remaining-Analyses-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.

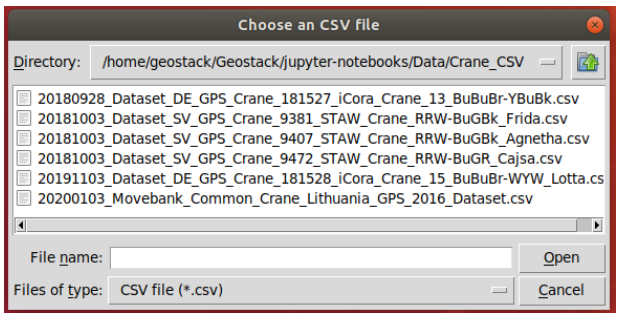
In the folder of the cookbook you will also find a folder called notebooks. This folder contains the Jupyter notebooks containing the process of modeling and importing the Crane datasets, GPS-Route datasets and the World Port Index dataset.

After you have finished reading this cookbook and the Jupyter notebooks that come with it you should continue to read this cookbook.

5.4.2.5 Automating the dataset import process

Now that you know how to manually model, index and import datasets in a MongoDB datastore, it's time to make a Python script which automates this process. This script will do the following:

- ➔ Give you the option to select and import a dataset in either CSV, JSON and GPX format. This is done using 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 we need to install a Python package which is used for the simple GUI to select files. This package is called Tkinter and can be installed using the following command:

`sudo apt-get install python3-tk`

Now we 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 import processes. This is done by entering the following command in a terminal: `mkdir ~/Geostack/import-utilities`

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

Now let's open this file in the Atom code editor. We start of by adding 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 we are need to let the script know it needs to create a Tkinter GUI in the background. We 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 we are going to create 3 functions which each represent a different type of data format. These formats are CSV, JSON and GPX. We start of with creating the function for the CSV import. We do this by adding the following code below the lines where we created the Tkinter GUI.

```
# 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 we need to create the function which is used to import JSON datasets. We 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 we are going to create the function which is used to import GPX datasets, we first need to create a function which creates a dataframe using raw GPX data. 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)
```

Now let's create the function 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 illustrations above, we used the function: “transform_data()”. This function has not been created yet. We are first going to create a function which is used to check whether an input dataframe is valid or not. We call this function: “check_dataframe()”. We do this by adding the following code below the csv_json() 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, we are going to create a function called:”column_selection()”. This function is used to ask the user what the names are of the longitude, latitude, altitude and timestamp columns. This is required since 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 we want to create a function called: “check_columns”. This function is used to check whether the columns, selected by the user in the function:”column_selection”, are valid.

```
# 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 let’s add the function: “transform_data”. This function 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 we will create later. So let’s add the following code below the function: “check_columns’.

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 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)
```

NOTE: The functions for loading the Crane and Trail data have not been created yet. This will be done later. Next we 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 CSV, JSON or GPX dataset. This function is called: “import_tool()”.

To add the main function we need to add the following function below the transform_data function.

NOTE: The following code is explained using inline comments. You can find the complete script in the folder: ”Building-the-VM-using-the-installation-script/Geostack/import-utilities” which can be found in the same folder as this cookbook.

```
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 we want to create the import function for the Crane datasets. Before we can do this we need to create a new Python file which is going to contain the Crane data model. We are going to call this file: “CraneModel.py”. We create this file by running the following command:

```
touch ~/Geostack/import-utilities/CraneModel.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 let’s add the Crane data model to this Python script. We 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 model of the Transmission Metadata embedded document:

```
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 model of the Geometry Embedded document:

```
class Geometry(EmbeddedDocument):

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

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

Next, copy the model of the Speed Embedded document:

```
class Speed(EmbeddedDocument):

    # Speed of the Crane
    ground_speed = FloatField()

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

Then copy the Transmission Document:

```
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)
```

Finally copy the model of the Tracker document:

```
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 we have to model we need to go back to the mongo-data-import script and import the CraneModel in this script. This is done by adding the following line below the import_tool function:

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

Now let's create the function which is used to import the Crane datasets. This function is called: "load_crane_data" and takes a dataframe, a list of columns and the database name as input. This function is basically the same as the one which we created in the cookbook:"Data-modeling-in-MongoDB-using-MongoEngine". There are a few minor differences which are crucial to automating the import the import process. So let's create this function by adding the following code below the line where we imported the CraneModel.

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 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!")
```

That's it! The code required to automatically import Crane datasets is finished. Now we want to do the same for the GPS Route (Trail) datasets. How this is done is shown on the next pages.

Now we want to create the import function for the GPS Route (Trail) datasets. Before we can do this we need to create a new Python file which is going to contain the Trail data model. We are going to call this file: “TrailModel.py”. We 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 let’s add the Trail data model to this Python script. We 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.

We start of by copying the Trail document:

```
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 we want to copy the Signal document:

```
class Signal(Document):

    # Timestamp of signal
    time = DateTimeField()

    # Geometry of signal
    geometry = EmbeddedDocumentField(Geometry)

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

Finally we want to copy the Geometry document:

```
class Geometry(EmbeddedDocument):

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

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

Now that we have to model we need to go back to the mongo-data-import script and import the TrailModel in this script. This is done by adding the following line below the load_crane_data function:

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

Now let's create the function which is used to import the Trail datasets. This function is called: "load_trail_data" and takes a dataframe, a list of columns and the database name as input. This function is basically the same as the one which we created in Jupyter notebook related to the GPS Routes.

So let's create this function by adding the following code below the line where we 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"))
```

The last thing we need to do is adding a line of code which makes sure that when the Python script is executed the import_tool function is triggered. We do this by adding the following at the bottom of the script.

```
import_tool()
```

Now let's create a desktop shortcut which runs the Python script when it's clicked. We do this by performing the following steps:

1) Create a desktop shortcut to start the Python script, using the following command:

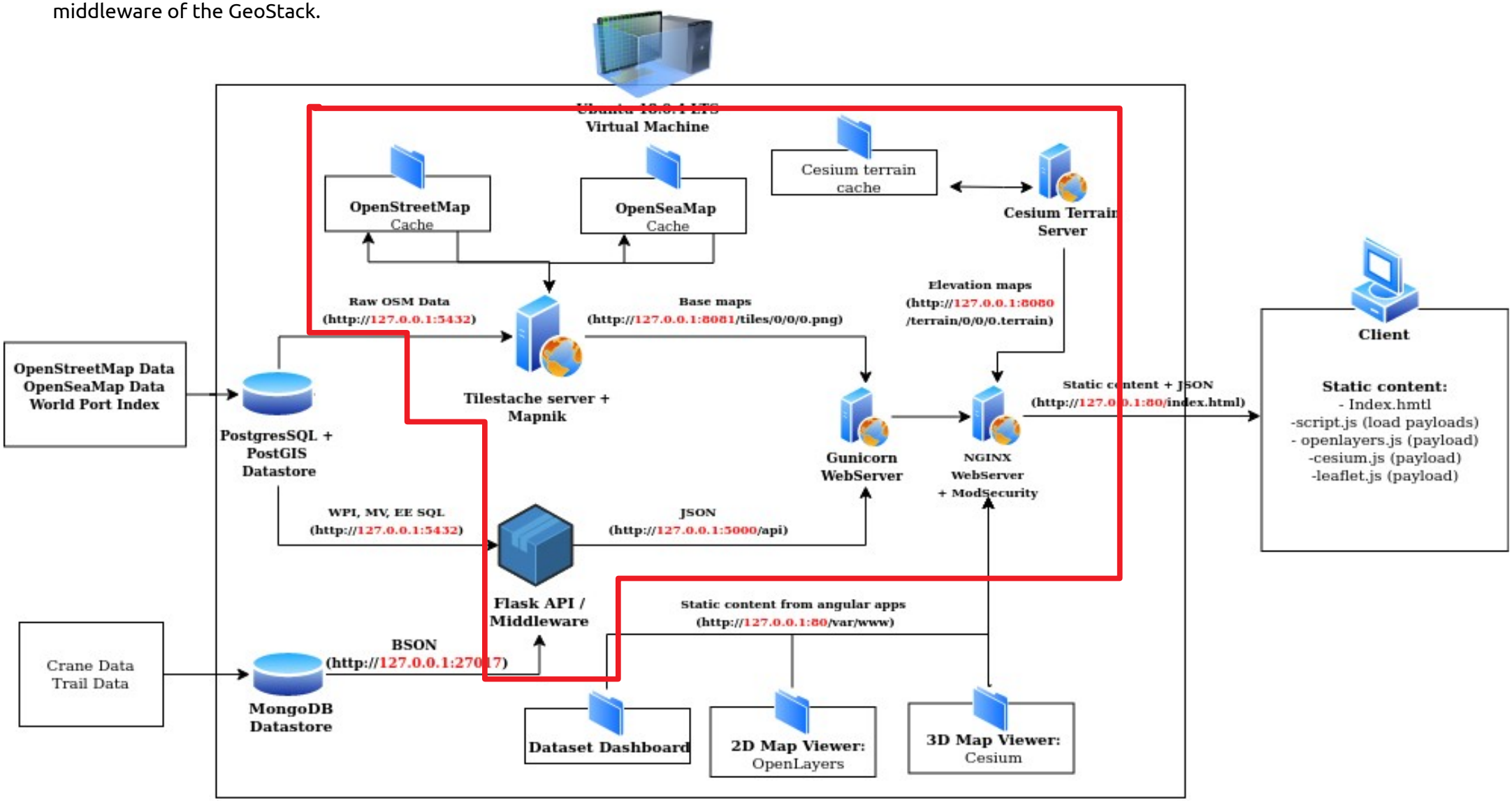
   touch ~/Desktop/Import-Mongo-Data.desktop

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
```

5.5 Installing middleware software

The Middleware consists of multiple webservers and an API. It contains the Tilestache Tileserver, the Cesium Terrain Server, the Flask-API and the NGINX-Webserver. The middleware is where all the “magic” happens. During this section we are going to install everything we need to create the middleware of the GeoStack.



Before we start installing the software required for the middleware of our GeoStack, you should start reading the cookbook: “A secure NGINX webserver with ModSecurity”. This is because you will learn more about the basics of Docker, how you should create separate Docker containers, how to run these containers and how to let containers communicate with eachother.

In this cookbook you will first learn how to create the base Python Flask application and the NGINX-Webserver as Docker containers.

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

- ➔ How to work with Docker containers.
- ➔ Create a basic Flask application and Dockerize it.
- ➔ Creating the secure NGINX webserver with ModSecurity and Dockerize it.
- ➔ Running the Docker containers separately.
- ➔ Linking and running the Docker containers together using Docker-Compose.

After you have finished reading the cookbook: “A secure NGINX webserver with ModSecurity” you should come back to this cookbook (Creating the GeoStack Course VM).

Now that you have finished the cookbook:"A secure NGINX webserver with ModSecurity" you have a folder which contains the base of the Flask-API and the NGINX webserver, The file structure of this folder is the same as shown in the image on the right.

We need to perform some transformations to make sure these files serve the needs of our GeoStack. Let's first copy the nginx-modesecurity folder to the Geostack root folder in which we have all our other components.

Press Ctrl + alt + t to open a terminal and then enter the command:

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

Now we want to remove the file called: "nginx_chapter_8.conf" since we don't need it anymore. We do this by running the following command:

```
rm ~/Geostack/nginx-modsecurity/ nginx_chapter_8.conf
```

Now we need to change the name of the file: "nginx.conf" to "nginx-docker.conf"since we want to keep this configuration for out GeoStack Docker containers. We do this by running the following command:

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

Now we want to 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 Now we need to add the NGINX Webserver to our GeoStack docker-compose.yml file which is located in the GeoStack root folder. So let's 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 we have defined services on which the NGINX webserver depends. These services have not been created yet. We are going to create these later on in the course. Because of this the Docker-compose up and docker-compose build commands will not work. First we are going to install NGINX with ModSecurity locally before we are going to create the flask-API, Tilestache Tileserver, the Cesium Terrain Server and their Docker services.

NOTE: When running the local version of the NGINX webserver, the Dockerized version of the NGINX webserver will not work. This is because both the Local as the Docker version will be running on port 80. To stop the local version of NGINX we will create a desktop shortcut in the next chapter. Use this shortcut to stop the local version of the NGINX webserver when starting the Dockerized version!

So let's install the Local instance of the NGINX webserver, ModSecurity and the ModSecurity Core Rule set. How this is done is shown in the next section.

5.5.1 Installing the NGINX Webserver 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 whether NGINX has been installed correctly 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. We 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
```

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 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) Add the following lines to that file.

```
# Make sure the ModSecurity Configuration is used.
Include /etc/nginx/modsec/modsecurity.conf
# Make sure the 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
# extra 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.

At this point we all the software in place to run our Local instance of the NGINX webserver with ModSecurity. The only thing that's missing at this point is an NGINX configuration file for our Local NGINX instance. How to create the configuration for our 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 Webserver instance by running the following command:

   `sudo 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. We 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 we need to make sure ModSecurity does not only log the attacks, but also blocks malicious requests by returning a “49503 Forbidden error”. We 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:

5.5.2 Installing Python-Flask Locally

Now that we have our Local NGINX webserver up and running we want to transform the base Python Flask application in such a way that it's going to serve the needs of our GeoStack. We first need 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

In the cookbook:"A secure NGINX Webserver with ModSecurity" we have created a basic Flask application. Now we want to extend the Flask-Application to serve the needs of our GeoStack.

To extend the base Python-Flask application you should now read the programming manual: "Creating-the-Python-Flask-Webapplication".

In this manual we are going to transform the Flask application into a Flask-API 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 Tileserver to be able to switch between map providers in our web applications later on.

The functionality of retrieving the Tileserver entries will not work yet since we have not created a Tileserver yet. This will be done in the next chapter, but more on that later! Let's first transform the base Python Flask application to our Flask-API by reading the programming manual for the Flask-API. After you are done you should come back to this cookbook and continue reading from here.

5.5.3 Installing the Tilestache Tileserver

Now that we have a working Flask-API we want to create the TileStache Tileserver. We start by creating a new folder in the GeoStack root folder in which we are going to create the Tilestache Tileserver code and configuration files. We do this by running the following command:

mkdir ~/Geostack/tilestache-server

Installing Gunicorn by running the following command : sudo apt install gunicorn

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

Installing Openstreetmap-Carto which provides our OpenStreetMap tile styling:

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-shapefiles.py script with the following command: scripts/get-shapefiles.py

   This script will download the base OpenStreetMap map. Although most of the data used to create the map is directly from the OpenStreetMap data file that you downloaded above, some shapefiles for things like low-zoom country boundaries are still needed. This 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** : Thisgeosta 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.

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 commands:

   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) Enter 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:
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
```

Now that we have the software in place we should put it to use. We first need to create the PostgreSQL database for our Tilestache Tileserver, which is going to contain our RAW OpenStreetMap data.

We do this by performing the steps described on the next page.

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 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`

5.5.3.1 Downloading and importing OpenStreetMap data in PostgreSQL

At this point we do have a 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 we are going to create a folder which will contain all our downloaded OpenStreetMap data. We do this by running the command: `mkdir ~/Geostack/tilestache-server/osm-data`.

Now we 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 file! This is the only file format that will work with the osm2pgsql OpenStreetMap data import tool.

Now we need to create a style.xml file which is used to link the styling to the nodes in the RAW OpenStreetMap data obtained from the PostgreSQL database: “gis”. This 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 5.5.3.7 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 -H flag is used to specify on which server the database is running.
- ➔ The -P flag is used to specify on which port the database is running.
- ➔ The -U flag is used to specify which user is the owner of the database.
- ➔ 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, will around 50 minutes to process and will take up 25GB of disk space after the processing is finished.

4) Then enter the password: "geostack" when prompted.

NOTE: After each import you should clear the folder containing the local OpenStreetMap cache. This can be done by running the following command:

```
rm -r ~/Geostack/tilestache-server/cache/openstreetmap-local/*
```

This is necessary because otherwise 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.

5.5.3.2 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. The 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 bouy 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 we 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 page 72).
  - ➢ 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 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 which cuts the next.osm file (which contains the OSM data that is going to be rendered) into smaller OSM files that each represent a tile.
  - ➢ The jsearch.jar application splits the OSM file that contains the data that 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 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 a OSM file.

```
<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 the a tag value in a node contains the value "bouy" the symbol of the bouy, located in the “searenderer/symbols” folder, will be linked to this node. The tilegen script also calls the jtile.jar application to generate the actual tiles using the svg’s which are 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 folder in our Tilestache cache.

- The jtile.jar application generates the actual tiles using the SVG's generated by the jsearch.jar application.

- The next.osm file 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 content 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.**

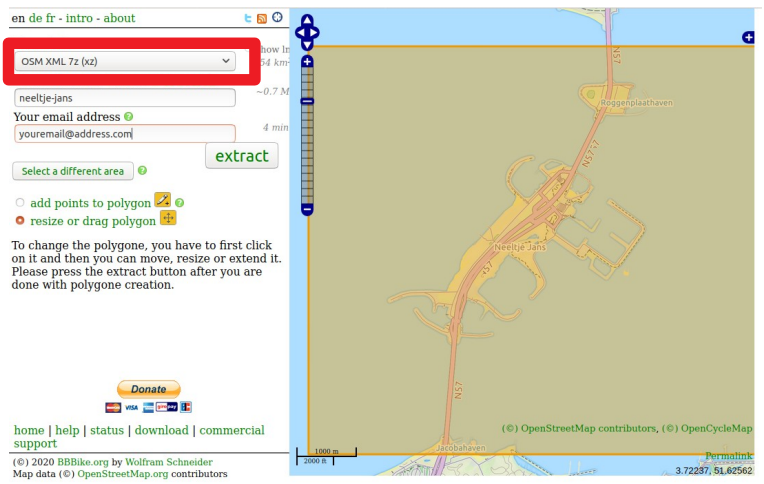
   ```
   > ~/Geostack/tilestache-server/renderer/work/diffs
   ```

   ➔ **Remove all contents of the world.osm file.**

   ```
   > ~/Geostack/tilestache-server/renderer/work/world.osm
   ```

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 chapter 5.5.3.1).

   I recommend to download a map extract from the website: https://extract.bbbike.org/ for testing purposes since you can select a small portion of a map as shown in the illustration below.



   Also notice the red box in the illustration above. This is where you select the .OSM format in which you want to download the OSM 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.

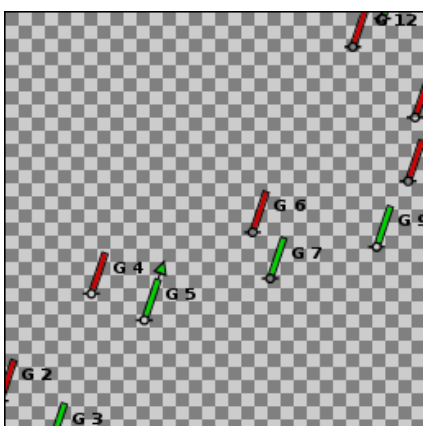
4) Enter the OpenSeaMap renderer folder by running the following command:

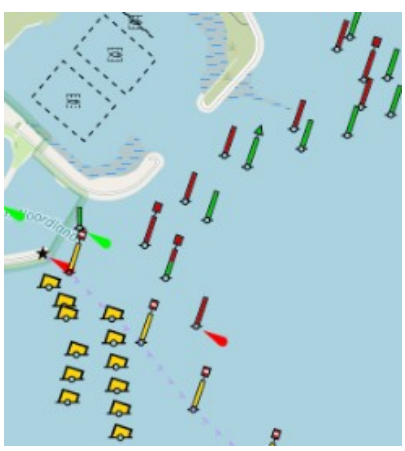
   `cd ~/Geostack/tilestache-server/renderer/work/`

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.

```
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 Tileserver (which we will do in the next chapter) the OpenSeaMap tiles will become available via the following URL: http://localhost:8081/openseamap-local/ or if the Tileserver is running behind the NGINX webserver via the URL: http://localhost/tiles/openseamap-local.

NOTE: In section 5.5.3.7 is described how you can create a Python script which automates the OpenSeaMap tiles generation process.

5.5.3.3 Creating the Tilestache configuration

1) Create a folder called cache in the tilestache-server directory:

   mkdir ~/Geostack/tilestache-server/cache

2) Create a file called tilestache-configuration.cfg in the tilestache-server directory.

   touch ~/Geostack/tilestache-server/tilestache-configuration.cfg

3) 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={your API key}"
      }
    }
  }
}
```

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: A local OpenStreetMap version which uses the RAW OSM data in the PostgreSQL database, A web version of OpenStreetMap which proxies the WMS from OSM, A local version of OpenSeaMap which uses the generated tiles in the Tilestache cache, A web version of OpenSeaMap which proxies the WMS from OpenSeaMap and 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 Tileserver. 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 provide name (Purple): This is where 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, we need to declare the URL on which the digital topographical map is located.
- ✔ In the URL we also declare the {Z}{X}{Y} notation for the requested tiles. (Yellow)

4) Create an HTML file in which we will place the layer names of our tilestache entries:

   touch ~/Geostack/tilestache-server/entries.html

5) 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)

6) Validate whether the configuration is working correctly. We 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 it higher if you are rendering big maps.

   gunicorn3 --workers 4 –timeout 100 -b :8081 "TileStache:WSGITileServer('/home/geostack/Geostack/tilestache-server/tilestache-configuration.cfg')"

7) Create a desktop shortcut which is used to start the Tileserver service.

   touch ~/Desktop/Start-Tileserver.desktop

8) 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 "gunicorn3 --workers 4 --timeout 100 -b :8081
"TileStache:WSGITileServer('/home/geostack/Geostack/tilestache-server/tilestache-
configuration.cfg')""
Name=Start-Tileserver
Icon=gnome-panel-launcher
Name[en_US]=Start-Tileserver
```

9) Create a desktop shortcut which is used to stop the Tileserver service.

   touch ~/Desktop/Stop-Tileserver.desktop

10) 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"
Name=Stop-Tileserver
Icon=gnome-panel-launcher
Name[en_US]=Stop-Tileserver
```

11) Add the Tileserver to our local NGINX webserver configuration by editing the configuration file of our **LOCAL** NGINX webserver instance.

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

12) 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;
}
```

13) 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 Tileserver which is running behind the NGINX webserver. **Remember to first start the Tileserver using the desktop shortcut!**

5.5.3.4 Dockerizing the Tilestache Tileserver

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:

```
FROM ubuntu:18.04

## Install git pip npm nodejs and mapnik
RUN apt-get update \
 && DEBIAN_FRONTEND="noninteractive" apt-get install -y -f git nodejs npm
mapnik-utils python3-mapnik python3-pip mapnik-utils python3 \
 && 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
RUN cd /tilestache/openstreetmap-carto && npm install -g carto

# Download the required shapefiles for Openstreetmap carto
RUN python3 /tilestache/openstreetmap-carto/scripts/get-shapefiles.py

# Copy the project.mml file
COPY project-docker.mml /tilestache/openstreetmap-carto/project.mml

# run carto to create our osm styles from db
RUN carto /tilestache/openstreetmap-carto/project.mml >
/tilestache/openstreetmap-carto/style.xml

# Copy tilestache config
COPY tilestache-configuration.cfg /tilestache/

# Copy the index.html
COPY entries.html /tilestache/
```

3) Copy 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/

4) Change the following section in the project.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 which need to be installed in the Tilestache Tileserver Docker container.

```
touch ~/Geostack/tilestache-server/requirements.txt
```

6) Add the following to the requirements.txt file and save it:

```
gunicorn
TileStache
Pillow
```

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: gunicorn3 --workers 4 -b :8081 --timeout 100  "TileStache:WSGITileServer('/tilestache/tilestache-configuration.cfg')"
```

7) Build the new service we 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 we 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 we 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`

5.5.3.5 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).

5.5.3.6 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 code is explained using inline comments. You can find the complete script in the folder: "Building-the-VM-using-the-installation-script/Geostack/import-utilities" which can be found in the same folder as this cookbook.

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 we 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()”. To add the main function we need to add the following function 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 we used 2 functions which have not be created yet. So let's create these functions. We start of by creating the function to import the 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 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 --append\
     --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)
```

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)
```

The last thing we need to do is adding the code which makes sure the Python script is executed and the import_tool function is triggered. We do this by adding the following at the bottom of the script.

```
import_tool()
```

Now let's create a desktop shortcut which runs the Python script when it's clicked. We do this by performing the following steps:

1) Create a desktop shortcut to start the Python script, using the following command:

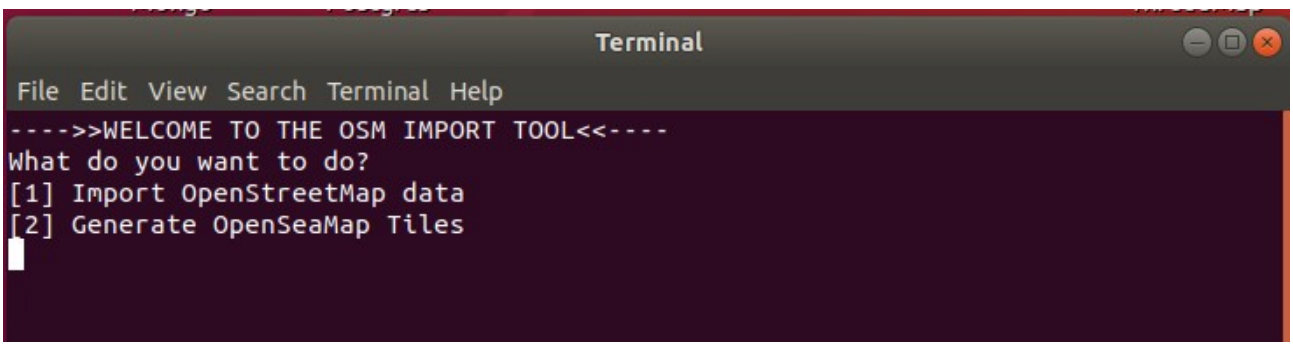
   touch ~/Desktop/Import-OSM-Data.desktop

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
```

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:



5.5.4 Installing the Cesium Terrain Server

Now that we have most of the Middleware components in place we should create the Cesium Terrain server which is used to generate and serve elevation maps to our 3D Map Viewer application. These terrain files contain information related to the height of the ground at a given point. Using the terrain files we can visualize local elevation maps in our web application.

A terrain file is created using a DEM file. A DEM 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 using 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**) sometimes called **Digital Elevation Models** (DEM) is a topographic **model** of the bare Earth that can be manipulated by computer programs. The data files contain the **elevation** data of the **terrain** in a **digital** format which relates to a rectangular grid.

Digital Surface Model (**DSM**) are files created using Airborne Light Detection and Ranging (LiDAR) In a LiDAR systems which use pulses of light traveled to the **ground**. When the pulse of light bounces off its target and returns to the sensor, it gives the range (a variable distance) to the Earth.

For more information related to DTM and DSM files you should visit the following URL:

https://gisgeography.com/dem-dsm-dtm-differences/

5.5.4.1 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 boxes which each represent a downloadable area.

When you zoom in on the location, from which you want to download the DEM file, you can click on the box representing the location which you want to download, as 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 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.
- ➔ 5M Raster DSM which is a is less precise (up to 5m) dataset which also includes the 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. We are not going to use this types of datasets in the GeoStack.

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, if extracted, give you a .TIF file containing elevation data.

5.5.4.2 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:

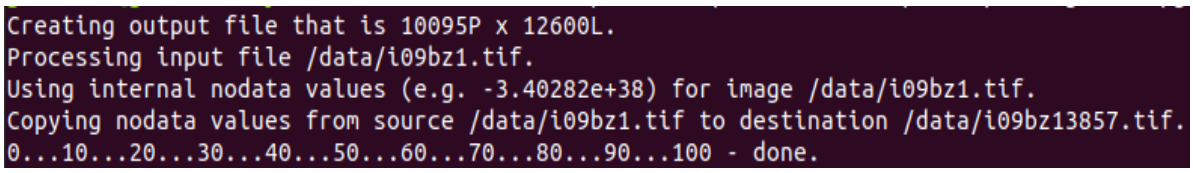
   `mkdir -p ~/Geostack/cesium-server/data/tilesets/terrain`

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`

   This process takes about 10 seconds an should look similar to the illustration below.



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 webserver.

   We can create a new directory by running the following command:

   `mkdir -p ~/Geostack/cesium-server/data/tilesets/terrain/M_52EZ2`

   In the 3D Map viewer we 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 webserver.

5) Navigate to the cesium-server directory by using the following command:

   `cd ~/Geostack/cesium-server`

6) Run the .terrain file generator using the folder created in the previous step as output directory

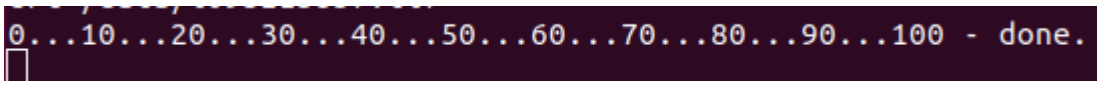
   `docker run -it -v ~/Geostack/cesium-server/data:/data tumgis/ctb-quantized-mesh ctb-tile -f Mesh -C -o /data/tilesets/terrain/M_52EZ2 /data/M_52EZ23857.TIF`

   The generation process of this file (500MB) takes around 10 minutes to complete.

7) 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 we don't want a default resource we are going to create the layer.json 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.



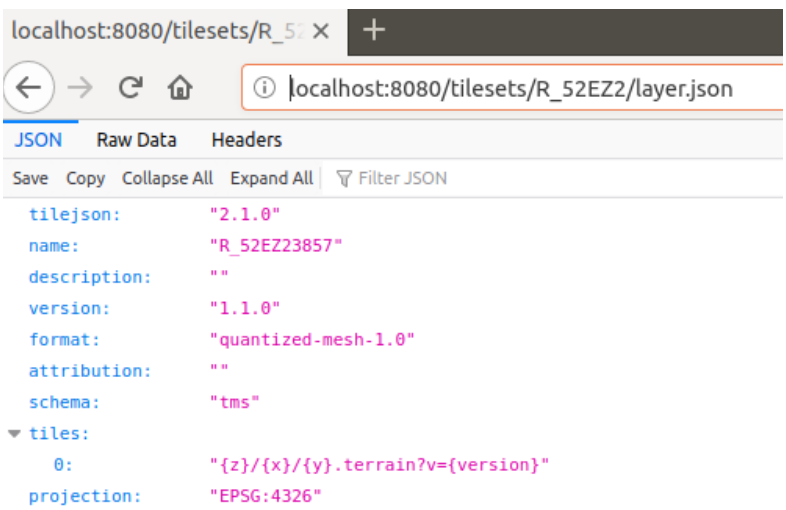
4) Run the Cesium terrain server by running the following command:

```
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 it will download the cesium-terrain-server docker image.

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.

5.5.4.3 Caching the Terrain files with Memcached

To be able to cache the generated .terrain files, we are going to use a software product called Memcached which serves as a temporary in-memory data cache. We are going to use a Memcached Docker container. This is done by running the following command:

```
docker run --rm --name cesium-memcache -p 11211:11211 memcached -m 200
```

This command 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 we 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 we use the syntax: “- -link cesium-memcache:memcached ”. This syntax links the running memcached container to our Cesium Terrain Server.

To check whether the memcached container is working you can perform the following steps:

1) Start the memcached Docker containers by running the following command:

   ```
   docker run --rm --name cesium-memcache -p 11211:11211 memcached -m 200
   ```

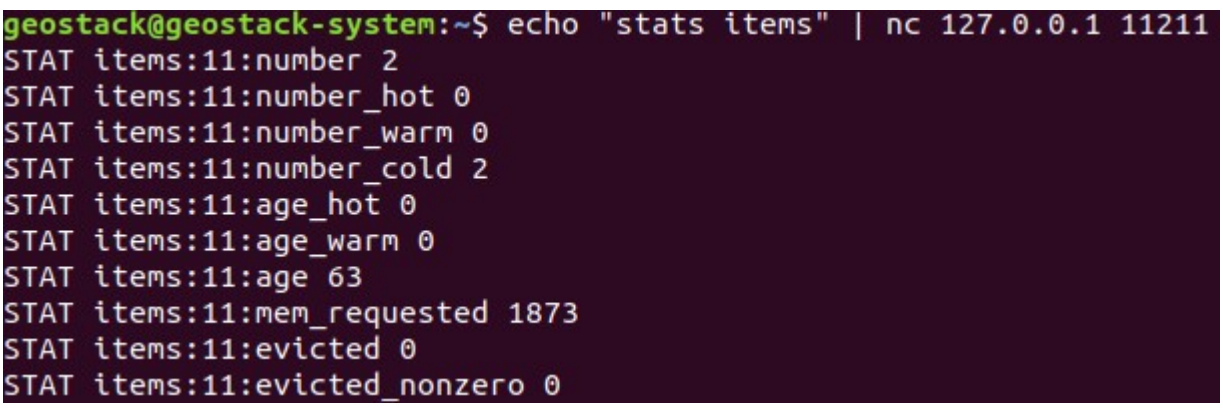
2) Start the Cesium terrain server with the memcached docker container by running the following command:

   ```
   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) List the current items in the memcached storage by connecting to the container using telnet. This is done by running the following command:

   ```
   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"
Name=Start-Memcached
Icon=gnome-panel-launcher
Name[en_US]=Start-Memcached
```

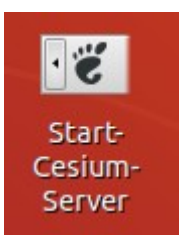
Now let's create a desktop shortcut which can be used to start the Cesium Terrain Server. This is done by performing the following steps:

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.

```
#!/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"
Name=Start-Cesium-Server
Icon=gnome-panel-launcher
Name[en_US]=Start-Cesium-Server
```

Now we have 2 new desktop icons as shown in the illustrations below.



NOTE: When running the Cesium Terrain Server remember to first start the Memcached cache using the shortcut which we created above!

5.5.4.4 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”

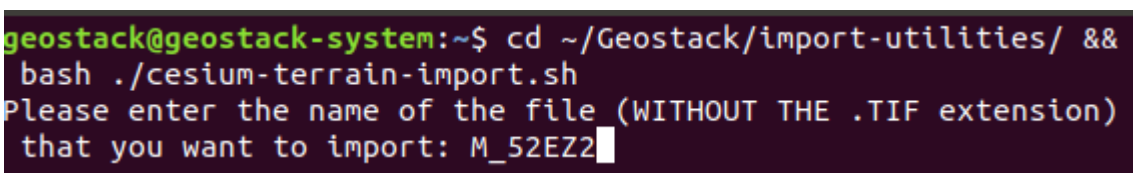
2) Then add the following code to this script:

```
#!/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.



5.5.4.5 Adding the Cesium Terrain server to our NGINX configuration

To make sure the Cesium Terrain Server is accessible via the NGINX webserver 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 we specified the upstream server for the Tilestache 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 we 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 webserver.

Remember to first start the Memcached cache and the Cesium Terrain Server using the desktop shortcuts shown in the illustration below.

5.5.4.6 Dockerizing the Cesium Terrain Server

Now that we have finished setting up the local version of the Cesium Terrain Server we should 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:
   - "11211:11211"
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 we specified the location of the last WMS:

```
# Location of our Cessium .terrain files.
location /terrain/ {
     proxy_pass http://cesium-terrain-server/tilesets/;
}
```

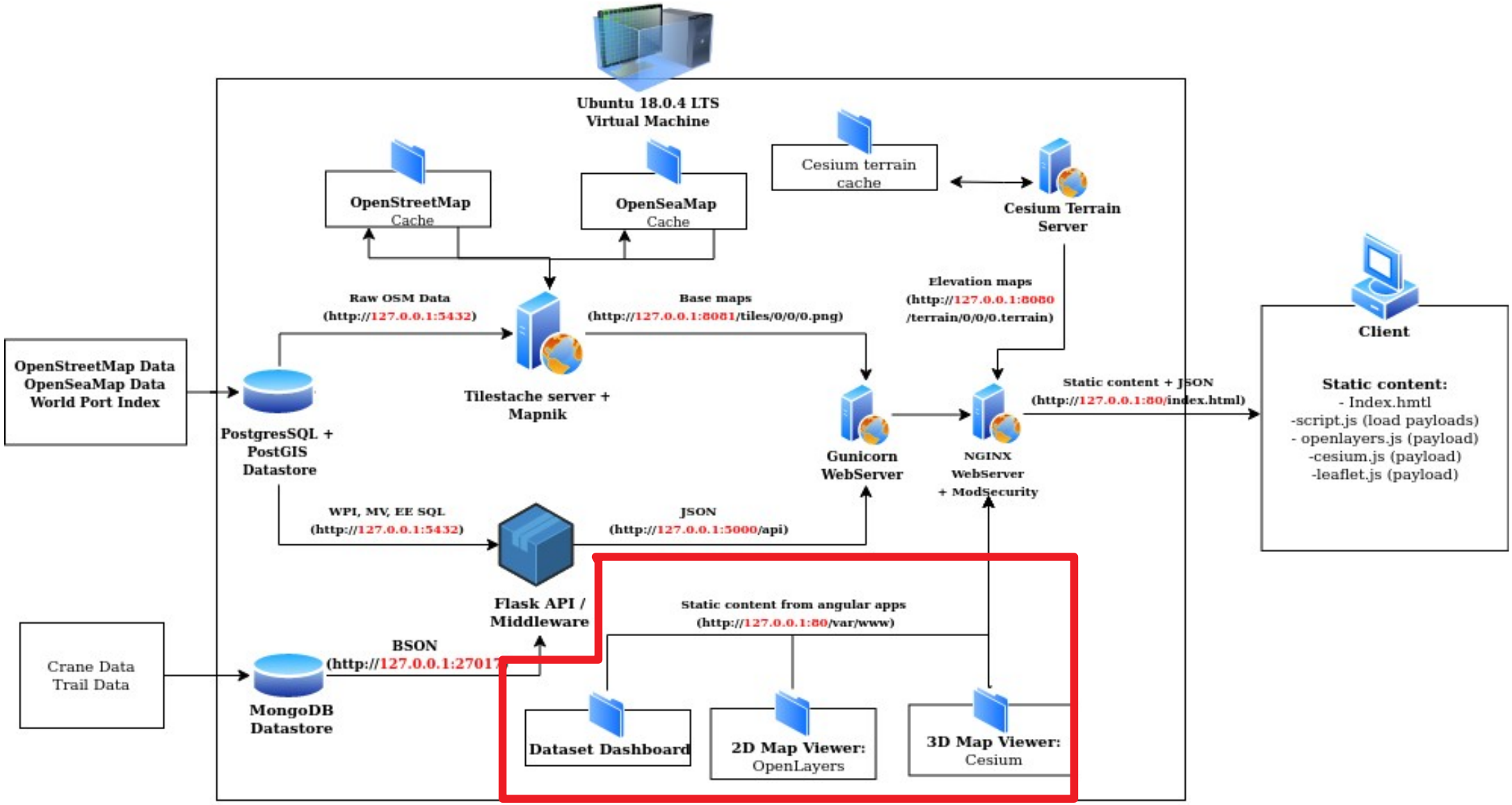
4) Save the configuration files by pressing the keys on your keyboard: ctrl + s

5) Rebuild the NGINX service for the changes to take effect:
cd ~/Geostack && docker-compose build nginx-webserver

That's it! Now we have Dockerized our Cesium Terrain Server. In Chapter 6 we will be running the GeoStack as Docker containers and you will be able to see the end result but let's first create the frontend of our GeoStack.

5.6 Installing the frontend software

At this point we have all the backend and middleware software in place. We have also modeled, indexed and imported the data in the corresponding datastores. This data now has to be visualized. We are going to do this by creating some web applications. First we need to install the software which is required to create the web applications.



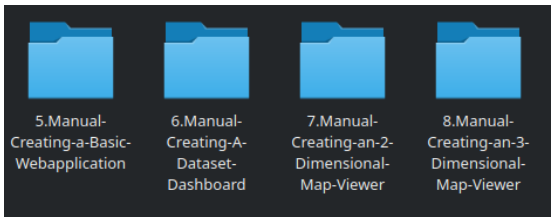
Installing the Angular CLI is done by running the following command:

```
sudo npm install -g @angular/cli
```

Now we want to create a folder called: “angular-apps” in the root GeoStack folder. This folder is going to contain all the web applications which we are going to create. To create this folder we need to execute the following command from a terminal:

```
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 as follows:

- **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 in combination with TypeScript.

- **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.

5.6.1 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 we will be installing and updating the base web application which we created during the programming manual: "**Creating a basic web application**".

So let's 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. When we run the following command to get an overview of the vulnerabilities which have to be fixed:

```
sudo npm audit fix
```

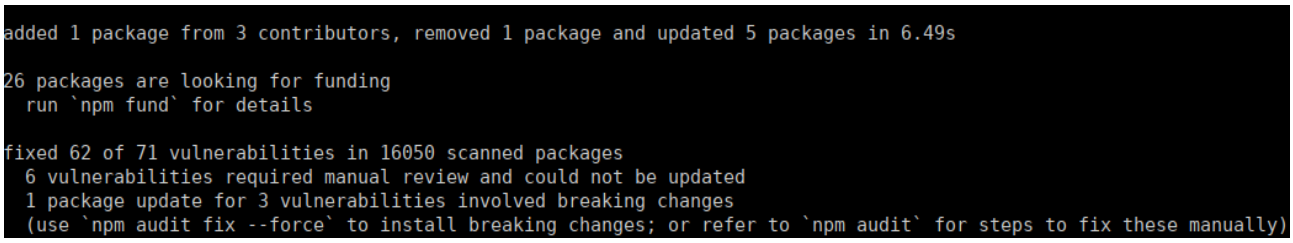
This will result in a long list of dependencies which have to be fixed. A part of the output is shown in the illustration below:

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 let's fix these vulnerabilities by updating the Node Modules. Updating an Angular project / application can be a tricky process. This is because some dependencies only work together when certain versions of these dependencies are installed. Let's first update the modules which won't break the application. This is done by running the command: `sudo npm audit fix`

Now let the process finish. The output of this command is shown in the illustration below:

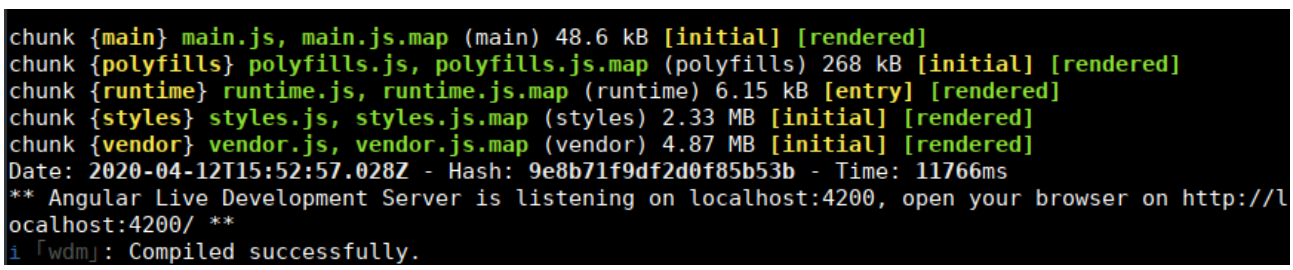


As you can see 62 of the 71 vulnerabilities which were found are now fixed and 6 of the vulnerabilities have to be reviewed manually since they could not be updated. One package update will break the application if it's performed. If you are sure the update will not break the application you can run 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 an list of dependencies which have to be reviewed manually. To update these packages you should inform the internet to see whats 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.



6. Running the GeoStack as Docker containers

If everything is correct you should end up with the following code in the docker-compose.yml file located in the folder: "~/Geostack". The content should be the same as shown below:
NOTE: The code is split in 3 sections.

```
#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:
   - "11211:11211"
```

```
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
```

First you have to make sure the Local NGINX webserver, MongoDB datastore and PostgreSQL datastore are not running, otherwise this will conflict with the ports of our Docker containers.

To stop the components listed above you can click on the desktop shortcuts that we have 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 will take a while.

Create a new desktop shortcut, used to start the GeoStack docker version and stop the local instances of the GeoStack components,by running the following command:

```
touch ~/Desktop/Start-Geostack-Docker.desktop
```

Add the following 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.

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:

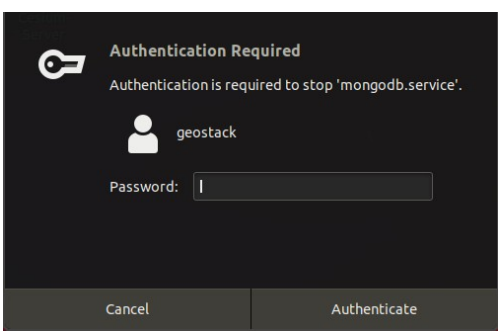
```
#!/usr/bin/env xdg-open

[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.

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
```

6.1 Exporting Docker images and volumes

After all the GeoStack Docker containers are build we can run the following command to list all the current images 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. So let's pretend we have another system somewhere on which we want to run the PostgreSQL Datastore Docker container and the data volume containing the World Port Index dataset and the RAW OpenstreetMap data.

We can export an specific image to easily distribute it to other systems. For this we are going to use the command: `docker save {image name} > {output directory name}.tar`

Let's export the PostgreSQL Docker image to a tar file. We do this by running the following command:

```
docker save geostack_postgresql-datastore > geostack_postgresql-datastore.tar
```

This will give us a .tar file as shown in the illustration below.

Now let's export the data volume containing the World Port Index dataset and the RAW OpenstreetMap data as zip file. We do this 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:

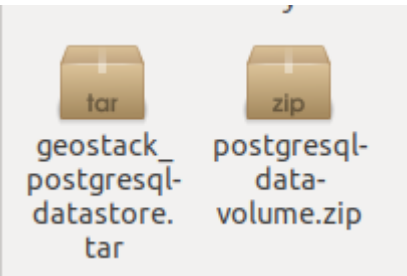
   ```
   zip -r postgresql-data-volume.zip ~/Geostack/postgresql-datastore/data
   ```

   Let this zipping process finish which takes about 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 we end up with a file containing the PostgreSQL Docker image and a file containing the PostgreSQL Docker data volume as shown in the illustration below:



Now let's distribute these files to another system by placing it on a USB drive or something similar. Let's imagine the system om which we 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 our GeoStack. The docker-compose.yml file expects a folder called:"postgresql-datastore" inside the folder with the docker-compose.yml file (~/GeoStack/")

When the files are on the new system you need to extract the postgresql-data-volume.zip file by running the following command:

unzip postgresql-data-volume.zip

Now we will end up with a postgresql-datastore folder which contains the PostgreSQL data volume which we copied from our other system.

Now let's load the PostgreSQL Docker container which we copied from our other system. We do this by running the following command:

```
docker load --input geostack_postgresql-datastore.tar
```

This will load the PostgreSQL Docker container from our other system in our new system. The output of the command which we 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 we run the following command we 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 we run the following command from the folder in which the docker-compose.yml file is located, we will have a working PostgreSQL Docker container copied from our GeoStack virtual machine:

```
docker compose up
```

You can perform the steps described in section 5.4.1.4 to check whether the data volume is working accordingly and the World Port Index database and the 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 |
```

7. Useful tips and tricks

In this chapter some useful tips and tricks are discussed. These tips and tricks can increase the workflow during and after completing the GeoStack Course.

7.1 Creating desktop shortcuts

During this cookbook multiple desktop shortcuts are created. The way a desktop shortcut is created 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 and add the following:

```
# 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 we 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 we need to make sure the shortcut is launch-able. This is done by running the following command: `sudo chmod +x ~/Desktop/{shortcut name}.desktop`

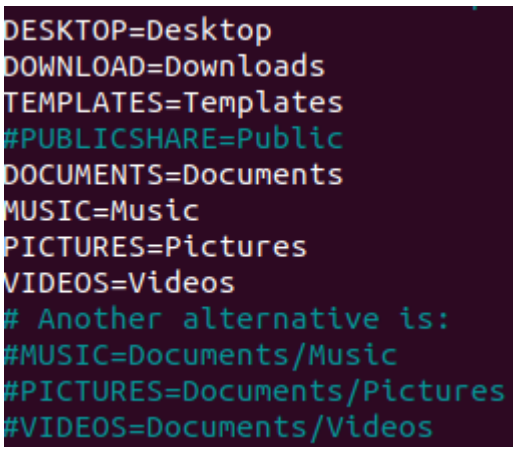
7.2 Removing DEFAULT folders from Nautilus

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.

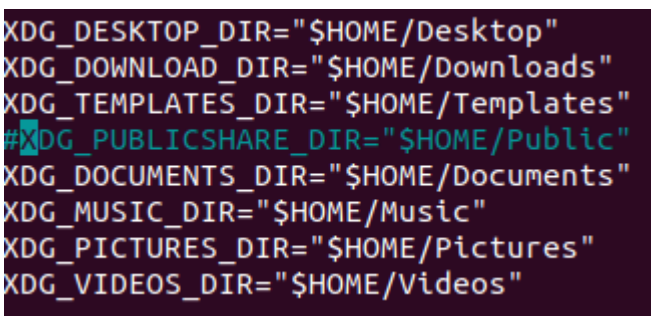


3) Save the file by pressing ctrl + s on you keyboard.

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:

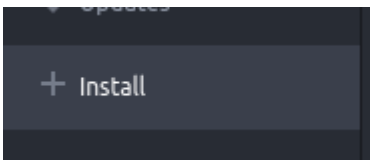
   `nautilus-desktop > /dev/null 2>&1 & echo "Restarted"`

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.

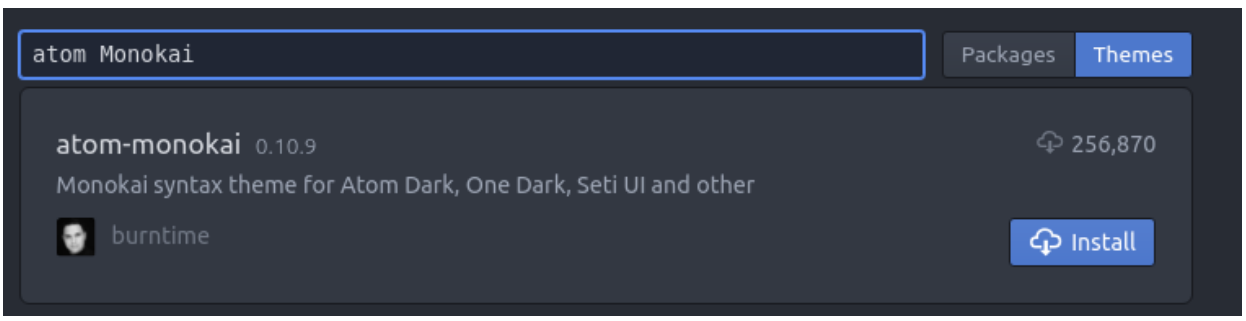
7.3 Editing Atom theme settings

You can change the theme of your Atom editor by performing the following steps:

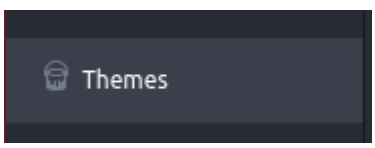
1) At the top of the Atom window go to Edit → Preferences.
2) Select Install in the left menu bar as shown in the illustration below.



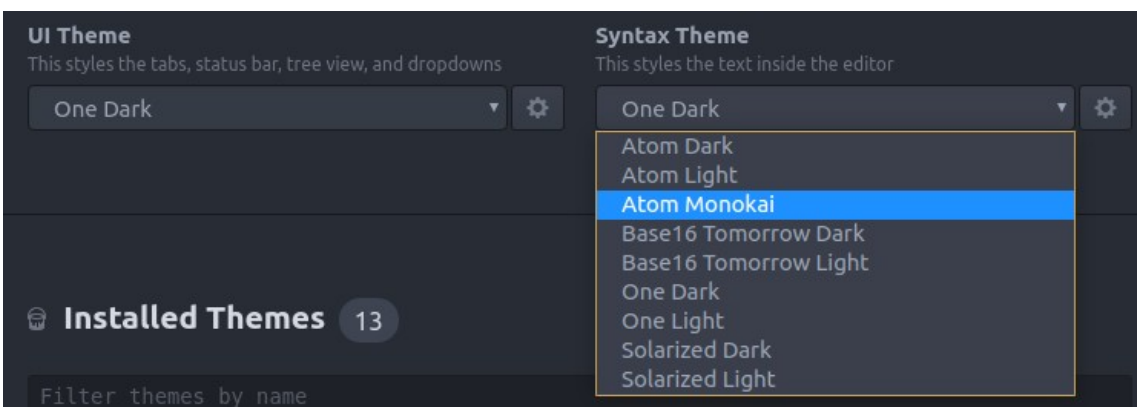
3) Select themes next to the search box, search for Atom Monokai and press install as shown in the illustration below.



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.

7.4 Useful Docker commands

Some useful Docker commands are as follows:

#	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

7.5 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.

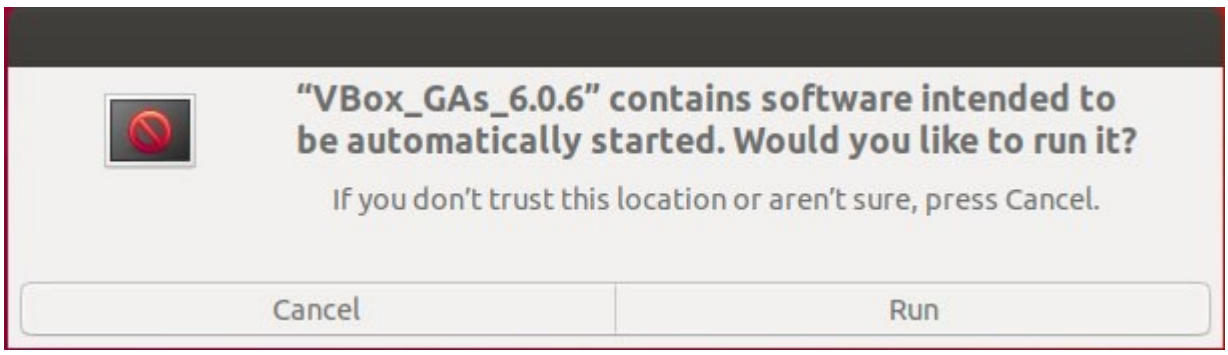
3) Install the required packages for building kernel module by running the following command:

   `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.

7.6 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

7.7 Recording your Virtual Machine screen

To record the screen of your Virtual Machine, you can use a simple tool called: “SimpleScreenRecorder”. First we need to install SimpleScreenRecorder. Then we are going to increase the size of the cursor which comes in handy when recording the screen. Afterwards as small description is given on how to use SimpleScreenRecorder.

7.7.1 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
   ```

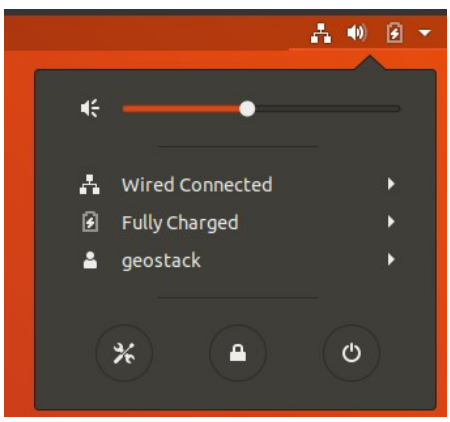
3) Install SimpleScreenRecorder using the following command:

   ```
   sudo apt install simplescreenrecorder
   ```

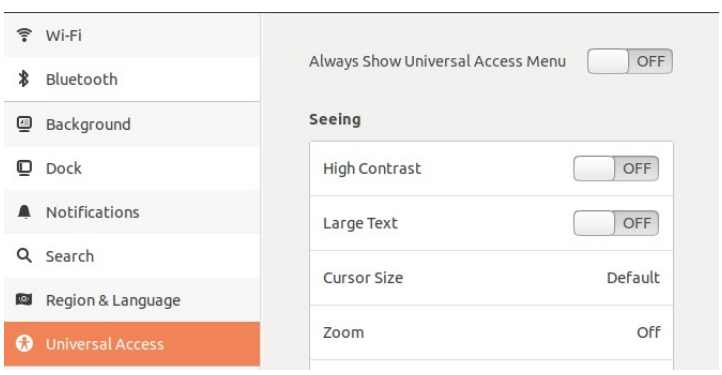
7.7.2 Increasing the cursor size

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:

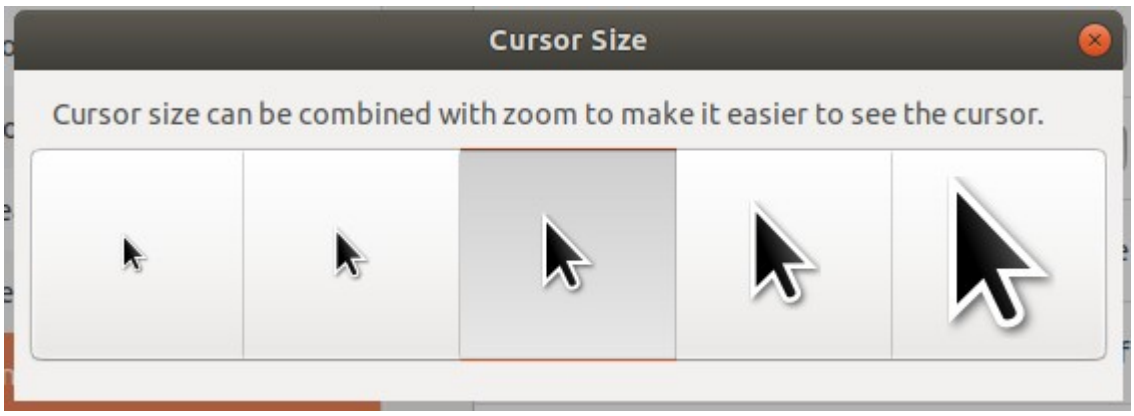
1) Open the settings menu in the top right of you Ubuntu Virtual Machine as shown in the illustration below.



2) Select Universal Access in the left menu of the screen that pops up. Then select Cursor Size as shown in the illustration below.



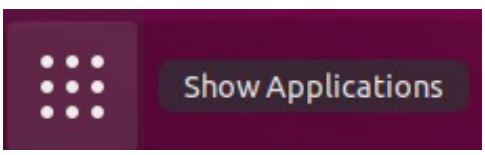
3) Then select the size of the cursor which suits your needs as shown in the illustration below.



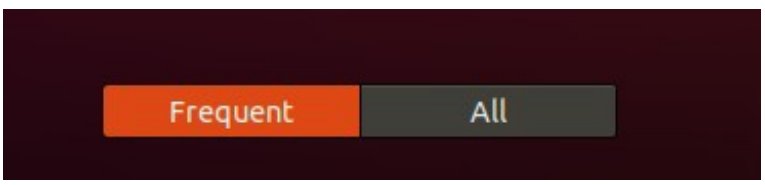
7.7.3 Using SimpleScreenRecorder

Opening the SimpleScreenRecorder application is done by performing the following steps:

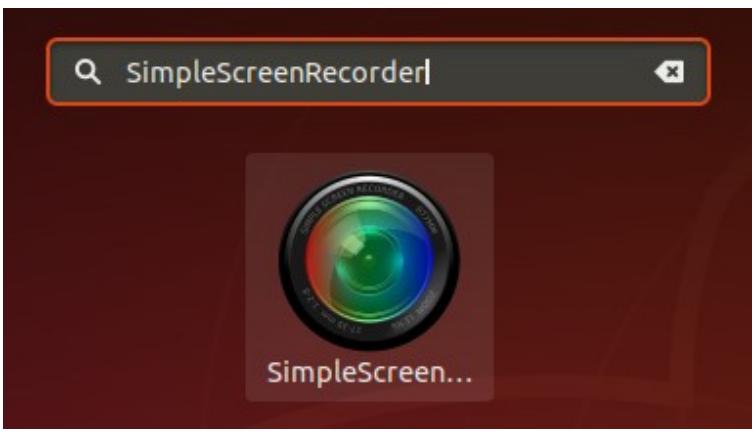
1) Open the show application menu on the bottom left of the desktop. The icon is shown in the illustration below.



2) Then select All at the bottom of the screen that pops up, as shown in the illustration below.

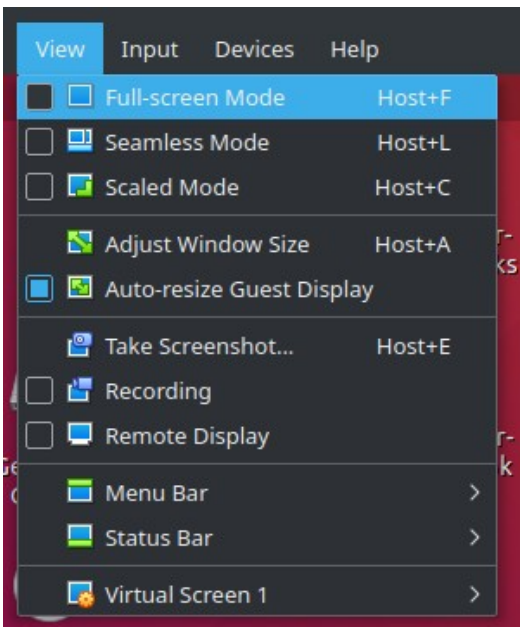


3) Enter “SimpleScreenRecorder” in the search bar and click on the icon that shows up, as shown in the illustration below.

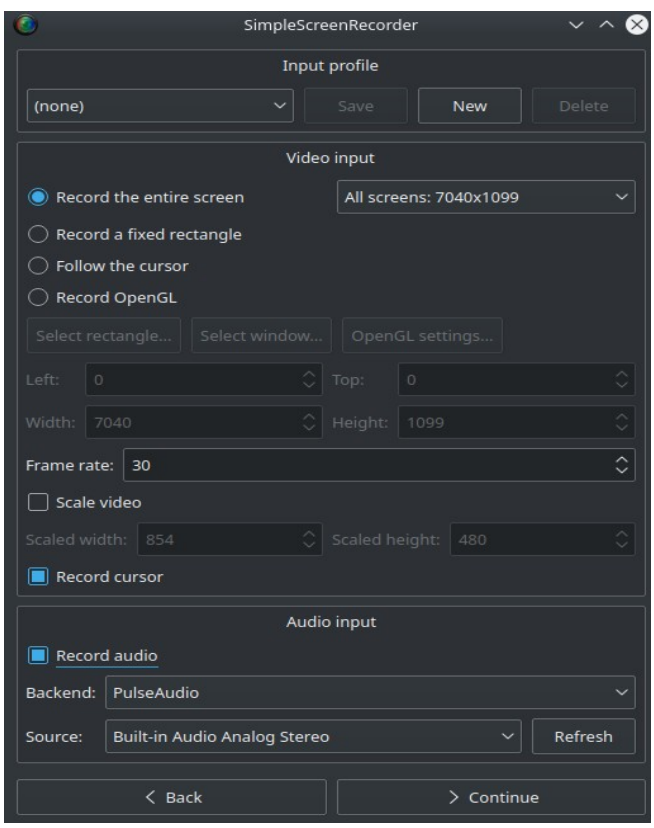


To use the SimpleScreenRecorder application you should perform the following steps:

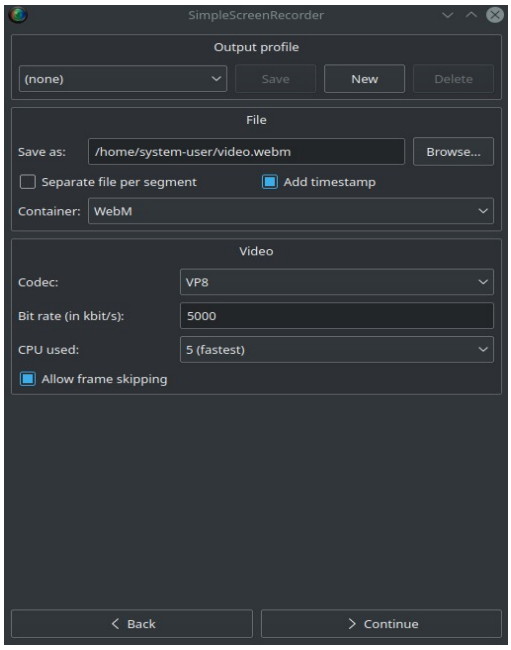
1) When recording your Virtual Machine window it's best to set your Virtual Machine window to full screen. This is done by clicking on View → Full-screen Mode in the VirtualBox window. How this is done is shown in the illustration below.



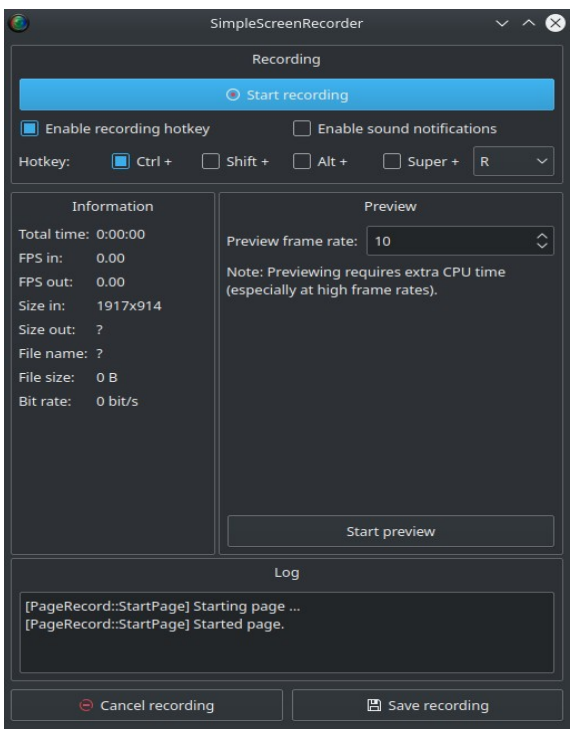
2) Edit the settings as you see fit. It's recommended to set the frame rate to 60 Frames Per Second. If you want to record with audio you should check the box: "Record Audio". Select continue after editing the settings.



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.



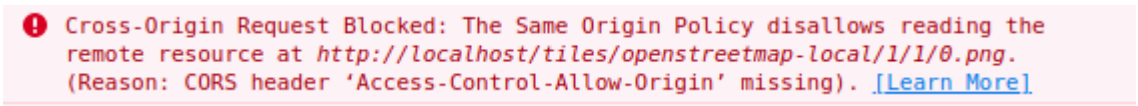
4) Next click on: "Start Recording" which will start the recording process. If you are done recording you can press on Pause recording and then on save recording to save the video.



When the screen is being recorded the icon shown in the illustration below will pop up in the top right of your screen.

7.8 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:



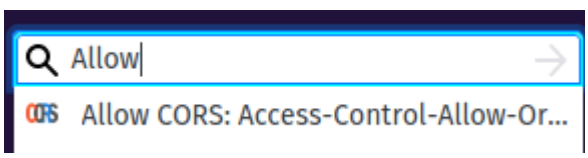
These error's occur when trying to obtain resources from multiple web servers at the same time. In the example above this error occurred when trying to obtain OpenStreetMap Tiles from the Tilestache tileserver (running on "localhost:8081") in the 3D Map Viewer running on localhost:4200.

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

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

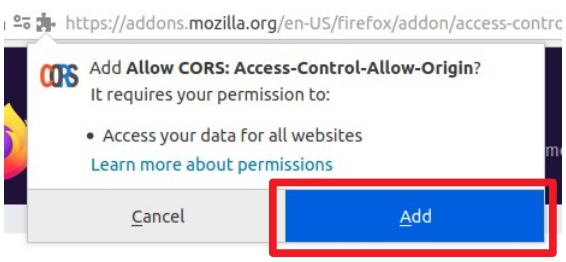
To solve this problem we need to install a Firefox extension called: "Allow CORS". This is done by performing the following steps:

1) Navigate to the Addons website of Firefox by entering the following URL in your browser: https://addons.mozilla.org/en-US/firefox/
2) Type: "Allow CORS" in the search bar in the top right of the page and click on the first option that pops up, as shown in the illustration below:

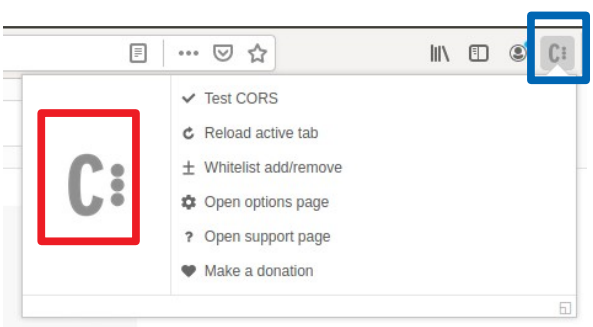


3) On the next screen select: "Add to Firefox" as shown in the illustration below:

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



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



8. Bibliography

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 Tileserver 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 Tileserver

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 Tileserver:

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/

8.12 Remaining Links