Creating a Spark Engine with Docker

1. Personal hub.docker.com registry

the URL to my personal hub.docker.com registry where I have pushed the image of my project.

https://hub.docker.com/u/vnpedroso

2. Introduction

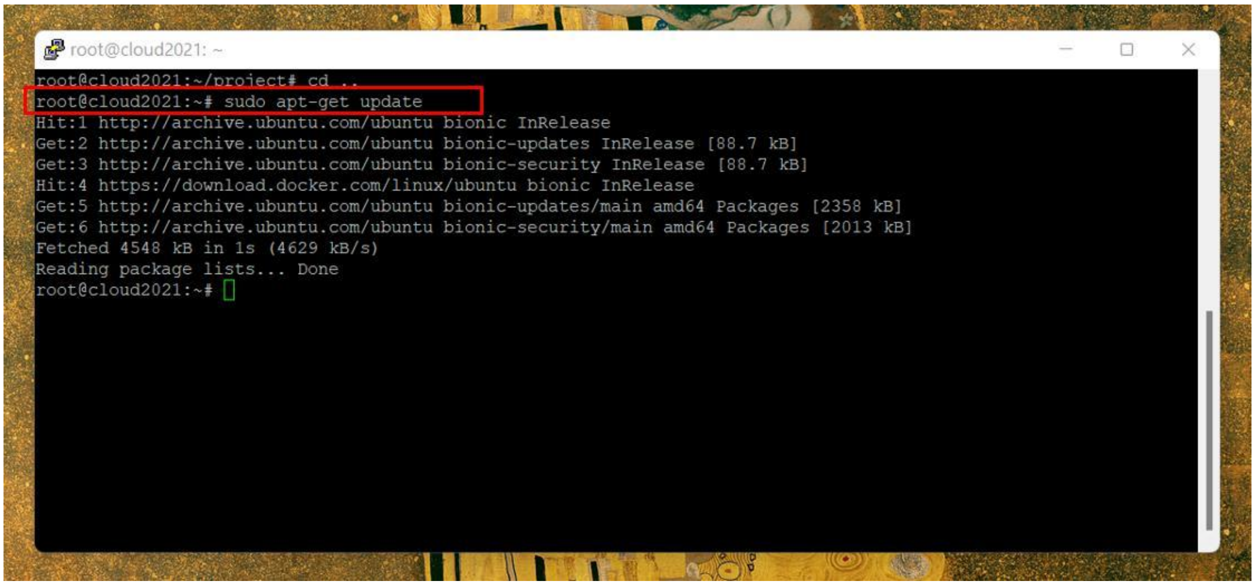
The advantages of parallel processing when dealing with Big Data are countless. Actually it is almost a necessity. Nowadays it is widely accepted that distributed systems (many machines realizing parallel processing) overcome local systems (single machines) in many different modalities. For instance, horizontal scaling is more elastic than vertical scaling: it is more difficult and more limited to keep increasing the CPU and GPU power of one single machine than to increase the quantity of machines, and therefore, the total CPU and GPU power of a system. More than that, distributed systems are fault tolerant: if one machine fails or bugs for any reason whatsoever, no data or actual processing is lost, small chunks of data are replicated and the processing is redistributed among the other fully operational machines. The same cannot be said about local systems.

Apache Spark is one of the most used frameworks for Big Data parallel processing. Spark is known for its flexibility and speed (up to 100x faster) versus the traditional MapReduce used in Big Data. That occurs mostly because it does not need to write data to disk after its famous “transformation” operation, it keeps data in memory, only spilling over to disk if the memory is filled (lazy evaluation). It also accepts a variety of data sources and allows you to manage your datasets as both RDDs (Resilient Distributed Datasets) or Spark DataFrames (DataFrames really similar to Python and R’s DataFrames).

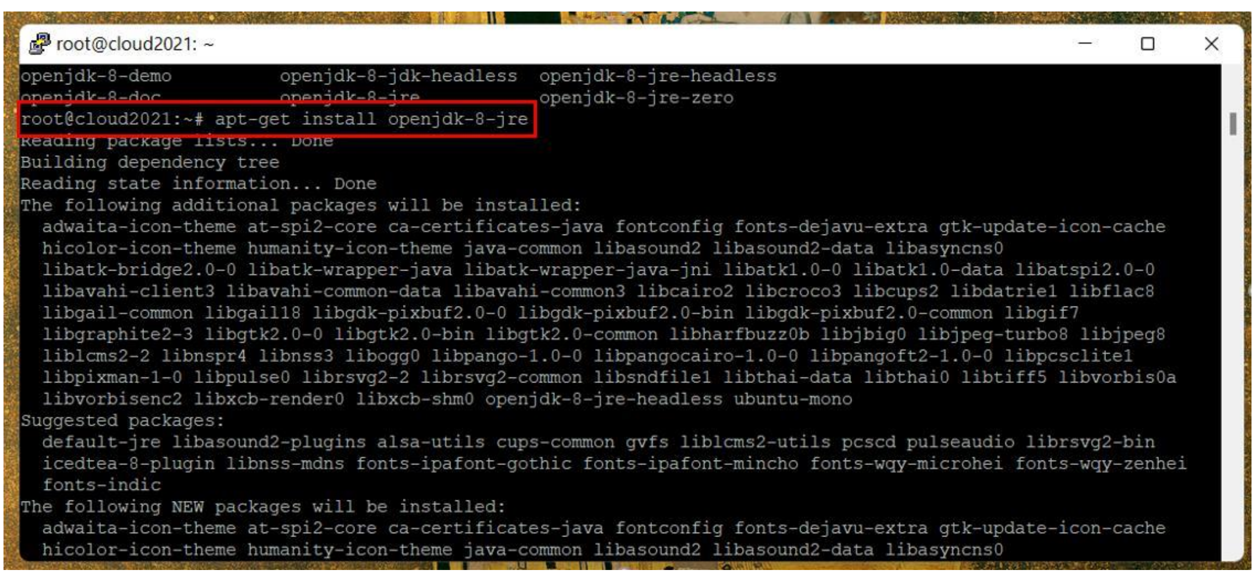
The structure of Apache Spark distributed system is made of a Master Node (the cluster manager), the Slaves or Workers Nodes and a Driver Program (the SparkContext or SparkSession). In this report I will show how to use Docker’s container technology in a Linux host to create a Spark Cluster with one container for each individual node.

OBS - Pre-Setup

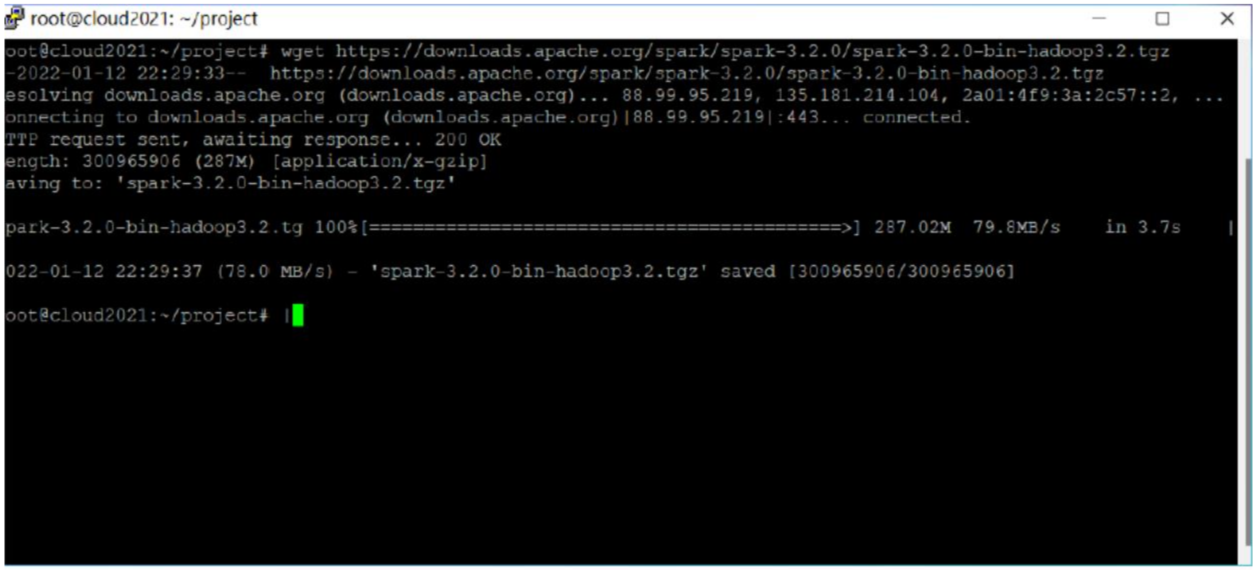
Updating packages and installing Java

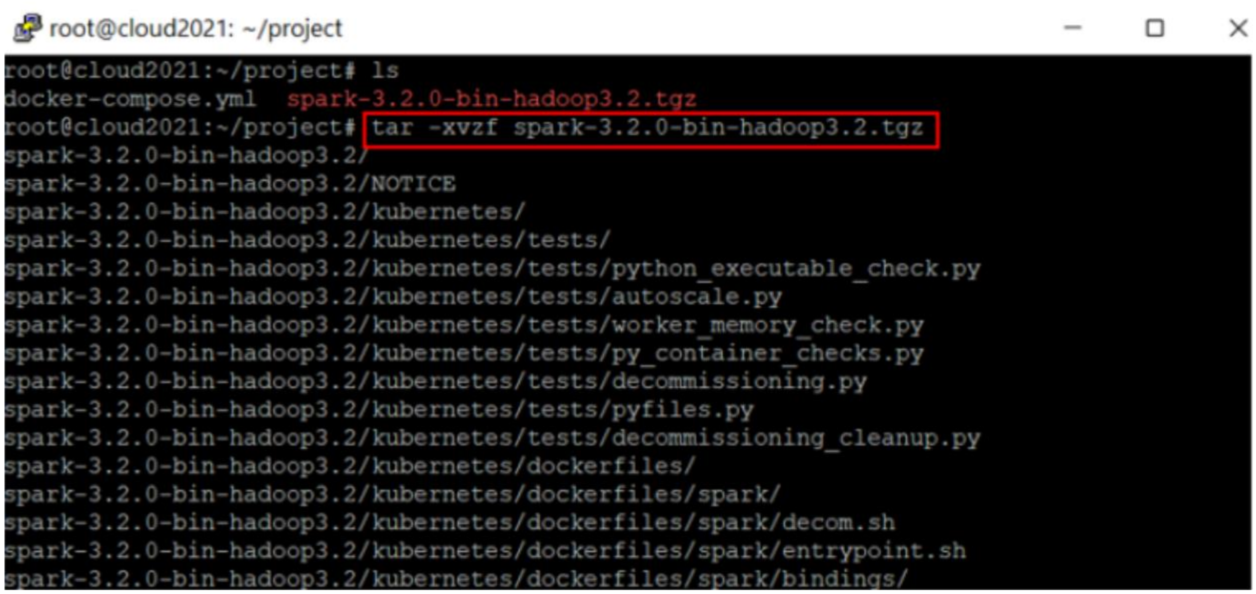


Open your terminal, update your packages and install Java. Apache Spark-Shell depends on a JVM to run. Skip the Java installation if you already have it in your machine.



Downloading and unzipping Spark



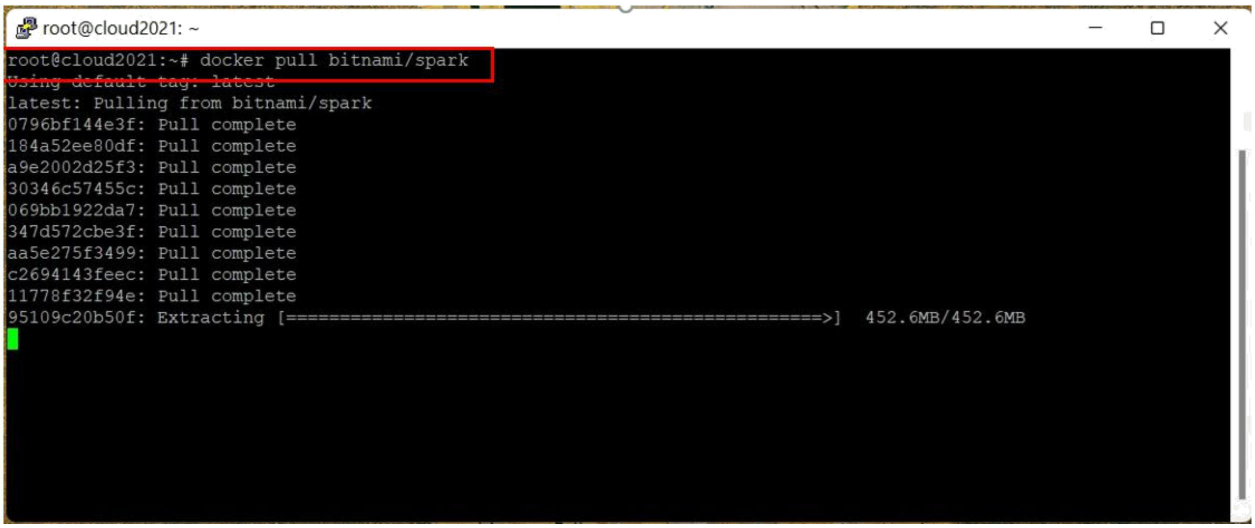


3. IaC

Start docker by executing **docker** command on your terminal. The first step is to pull the spark image from its repository on **Dockerhub.com.** That will pull the image from its online repository into our machine. That can be achieved by executing the following command:

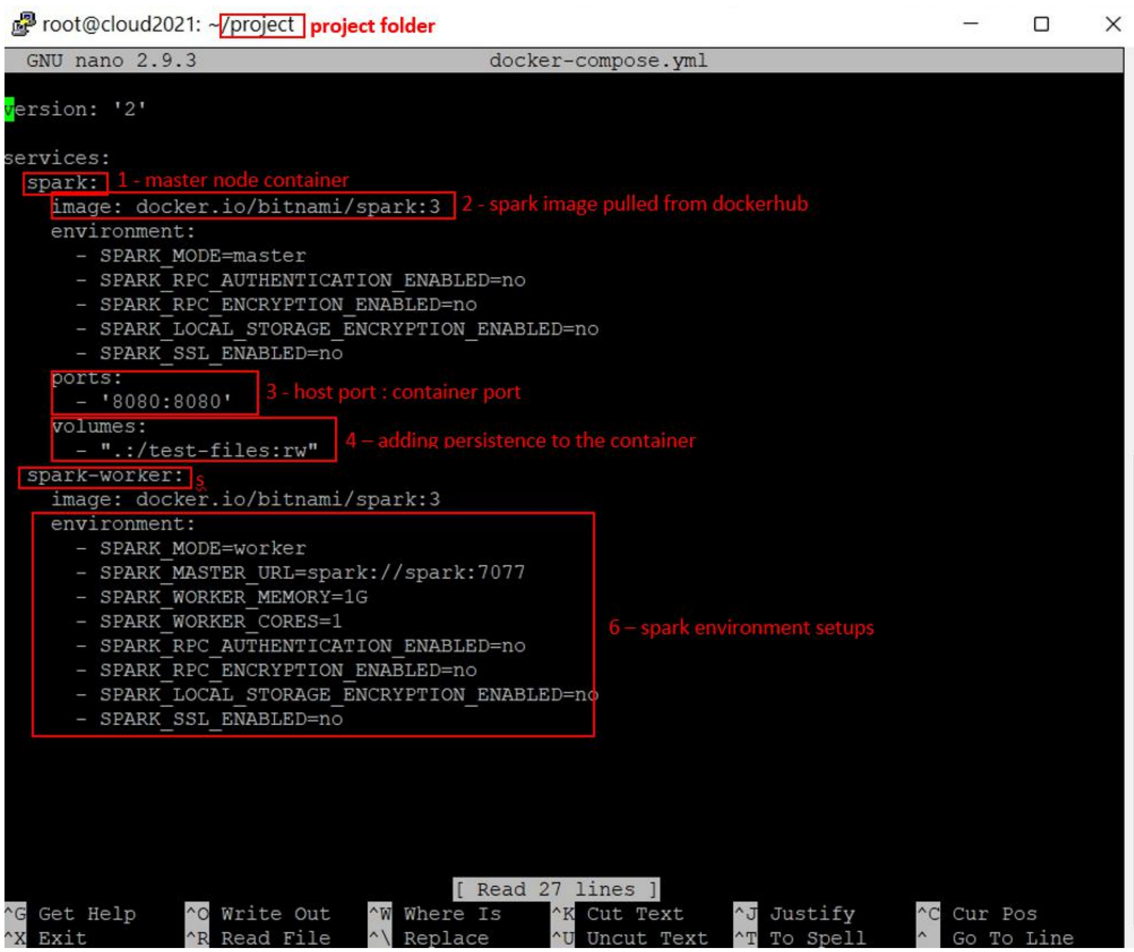
Docker Pull Command

```
docker pull bitnami/spark
```



Creating the docker compose file

Use the following command to create the docker-compose file: **nano docker-compose.yml**



The docker-compose is a file that can set up the whole process of creating and deploying a container. It's file extension is .yml, which stands for YAML, a type of data serialization language, used by docker, that is known for its readability.

In the figure above we are creating two containers with our docker-compose, one for our master node and the other for our slave node. We can create as many slave nodes containers ("spark-worker") as we want to, but only one master node container ("spark").

All the nodes should have as default image the spark image we pulled from dockerhub.

As aforementioned, to realize its parallel computing, Apache Spark uses one master node to control slave nodes. The master node monitors the whole cluster and each slave nodes by managing their tasks.

Notice that the “spark-worker” parameter is at the same level of indentation as the “spark” parameter. Both are containers created from the base image. If you follow the same rules of syntax (in which the indentation is included), you can add as many containers for as many nodes you have! Notice also that we allocated 1GB of memory to our slave node, that environment set-up can be edited within the docker file, it only depends on the total memory of your distributed system and the needs of your analysis. That memory allocation is a clear example of docker’s IaaS (Infrastructure as Code) capabilities.

Returning to our docker-compose file, here follows a little guidebook of its main set-up words:

- **Version -** determines the version of docker that we will be using
- **Services -** determines the services that will be executed
- **Image -** determines the base image that will be used
- **Environment -** determines environment variables of each base image
- **Ports -** determines the connection ports of the host machine and the images’ containers

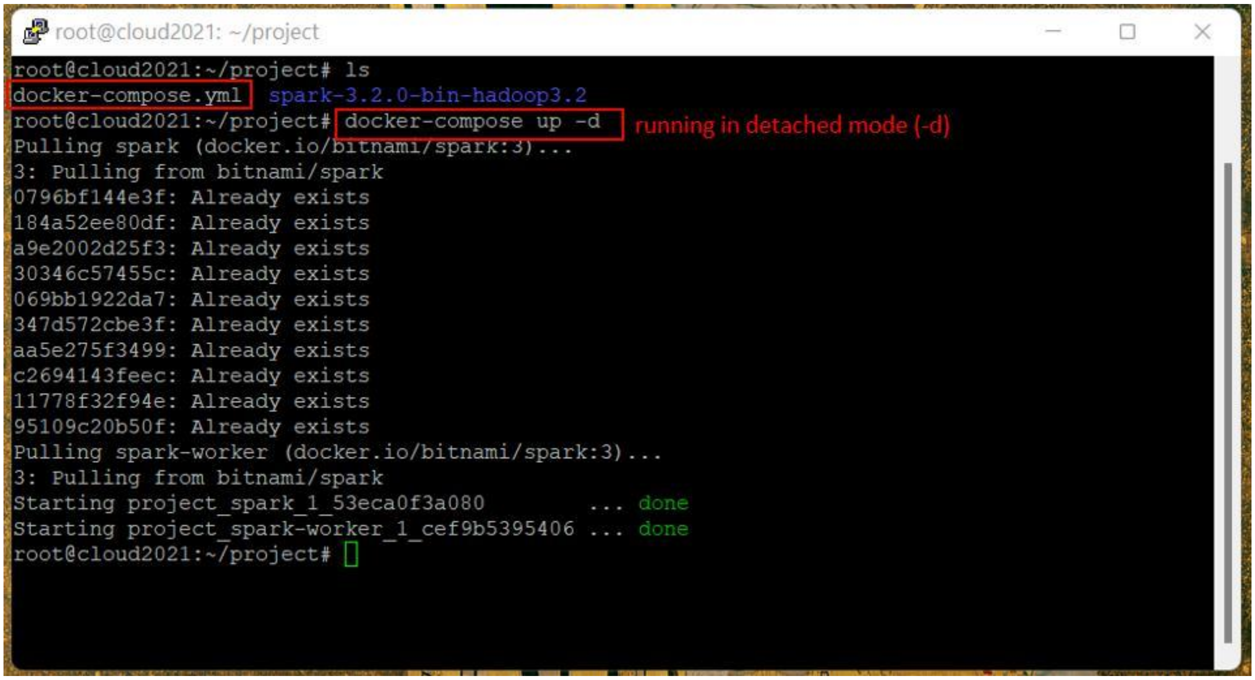
- **Volumes -** creates persistence of data inside the container. It’s specified directory is connected to the host, it’s data exists outside the container and can be accessed from the inside of a container

4. Demonstration

Run the docker-compose file

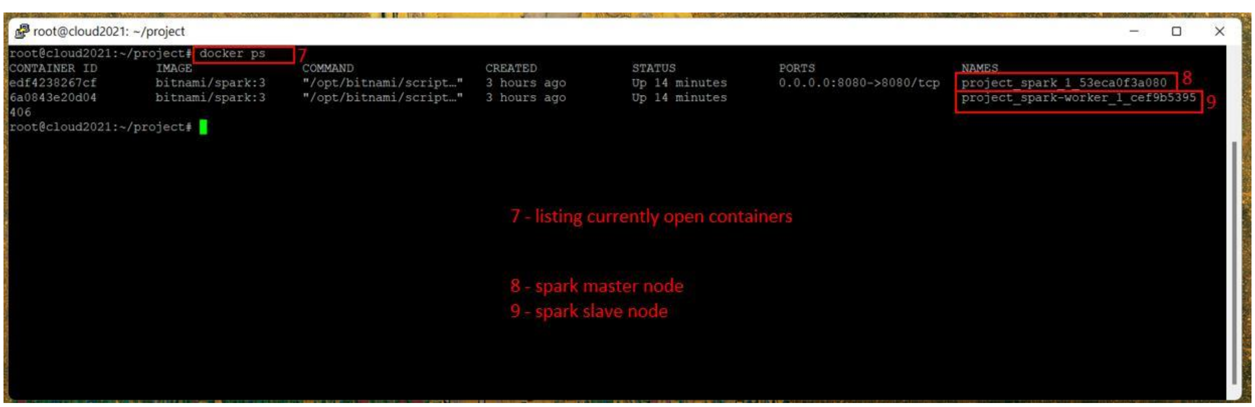
In this section the first step we ought to perform is to run our docker-compose file and check the deployment of our cluster’s containers.

Run the following docker command in order to run it: **docker-compose up -d**



Check the deployment of the cluster

Use the docker command **docker ps** to check on all currently running containers. Information on all the containers specified in the docker-compose file should be displayed after this command.



In our case, our two containers were displayed, sinalizing that now we have a Apache Spark Cluster operating with a master and a slave node, each inside a container.

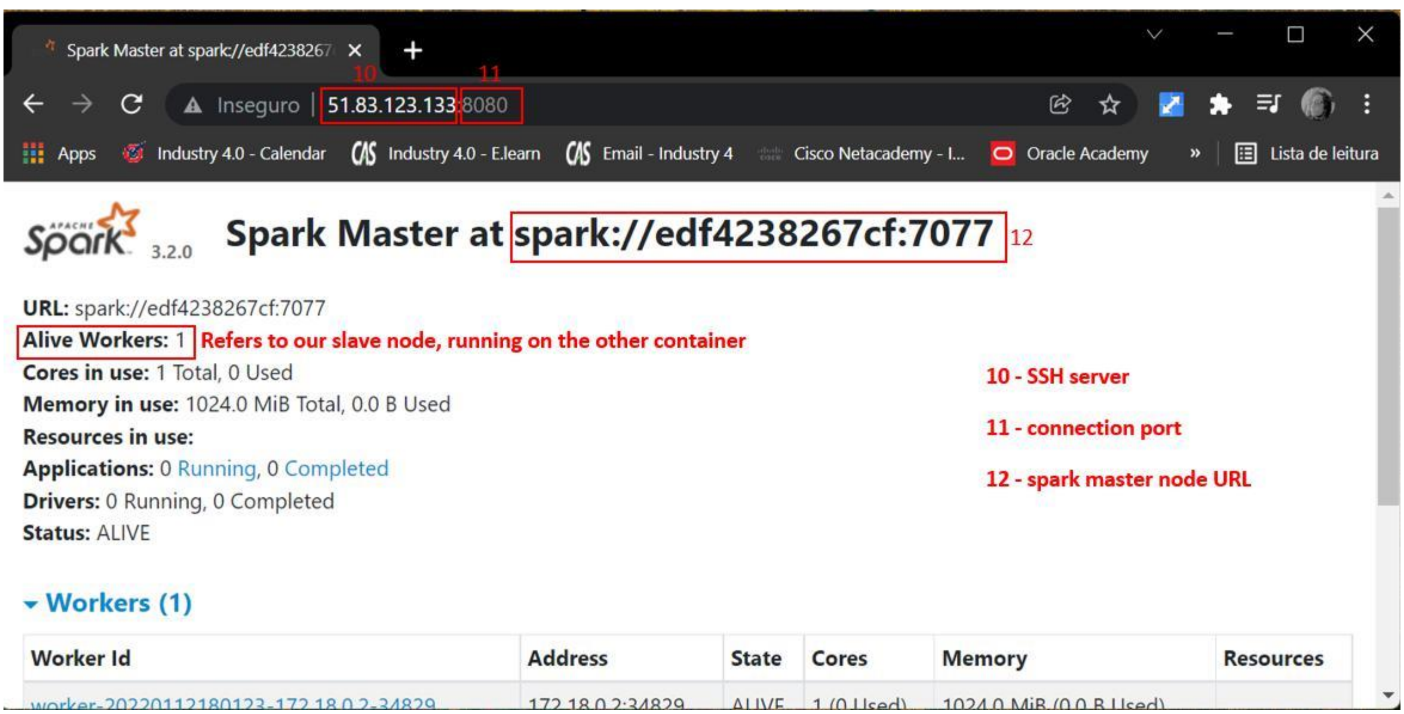
Accessing the Cluster Manager in your browser

After the successful deployment, you can access a cluster manager interface using the host address and the connection port determined in the docker-compose file by type the following in your browser:

http://*host_addres***:***connection_port*

In this report's example, we're using a SSH server to connect to our host machine. Therefore we will use the SSH server address as the host address. The connection port is **8080**, as determined in the docker-compose file. Thus our URL is the following:

http://51.83.123.133:8080

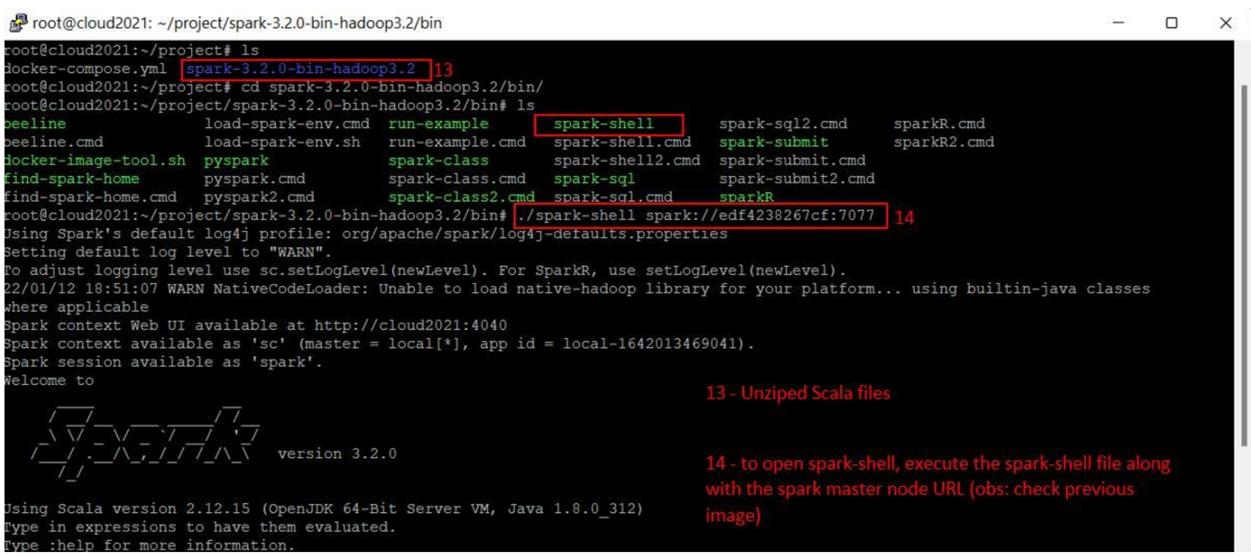


Accessing the spark-shell

Our next test to determine whether our containerized Spark Cluster is functional is to check if it gives us access to the spark-shell, Apache Spark interactive shell in which we can import and treat our data, run machine learning scripts and much more. Spark-shell can run Python (through the API known as PySpark) and Scala code. Keep in mind that to run Scala, a JVM is needed, for the Scala compiler produces JVM bytecode, just like java compiler “javac”. A functional JVM for Scala can be provided with the use of a JDK version of 1.4 or higher.

To access the spark-shell we need to execute the spark-shell file inside the bin directory, which in turn is inside the unzipped directory of our Spark download. You must also provide the Master Node address shown in the figure above. Spark-shell can be accessed both from the host machine and the Master Node Container. From either we connect with the following command:

*spark_unzipped_directory_filepath***/bin/ ./spark-shell spark://***master_node_address*

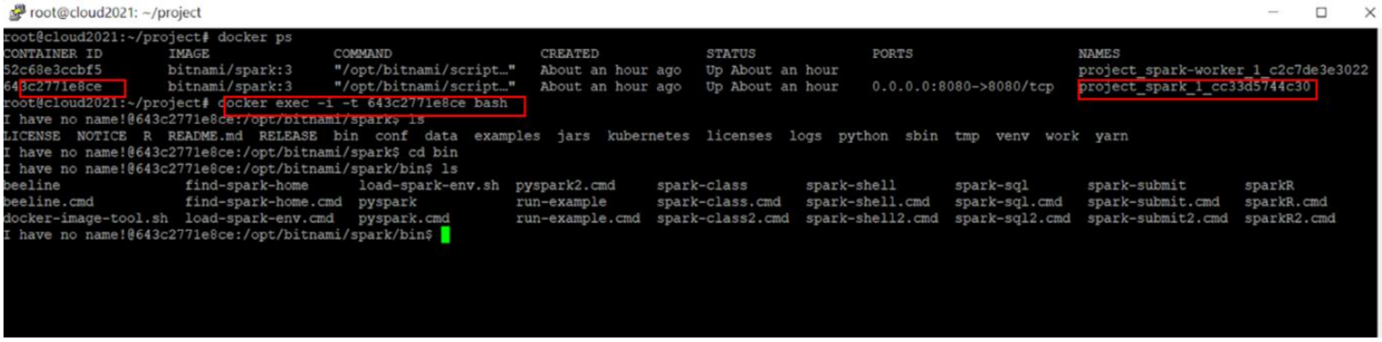


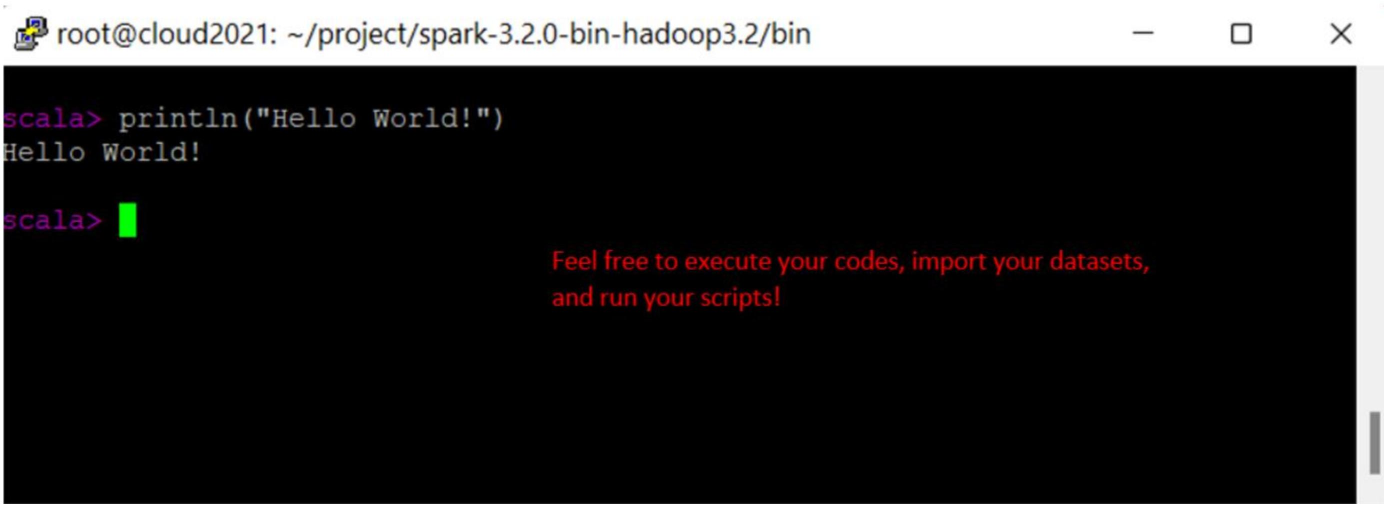
The only difference is that, in order to access the spark-shell from the container we must first run an interactive terminal of the container, which requires one of the variations of the following docker command:

docker exec -i -t *container_id* **bash**

docker exec -i -t *container_name* **bash**

The “exec” stands for execute, the “-i” parameter for interactive, the “-t” parameter for terminal, and the “bash” is the type of terminal that will be run.





Checking Data Persistence

The last test on our Apache Spark Containerized Cluster is to check the data persistence in the containers, that is, if the data generated in the containers can persist its stop. Docker resource for that is the “volumes” in the docker-compose file. As mentioned, it creates a directory connected to another directory in the host machine. The folder in the container does not contain any data by itself, but it can access the host directory data. It can also provide a way of taking data generated by the container and sending it to the host directory. If a file gets erased in one folder, it is also erased in the other, keep that in mind in order not to make mistakes.

To check the data persistence we must know where the volume’s directory was deployed. In our docker-compose example it is in the Master Node Container. We have just learned how to connect to an interactive terminal in the Master Node Container, hence, we can just enter the folder using **cd/test-files.**

Notice that when we use the command **ls** to list the data inside that folder we get the same files and directories that were in our project's directory, the one from which we executed the docker-compose file! With the conclusion of this final test we can be sure of the deployment and functioning of our Containerized Apache Spark Cluster!

5. Conclusions

In this report we navigated through several of docker's main commands, analyzed the structure of a docker-compose file, followed its step-by-step deployment, checked the containers' data persistence and, most importantly, implemented docker technology in a practical and useful solution. Nowadays Machine Learning and Big Data applications, and not only Software Development, are also taking advantage of the light, fast, isolated and uniform environments provided by containerization.