

# Creating a Spark Engine with Docker

<b>Project requirements</b>	<b>2</b>
<b>Technical instructions</b>	<b>3</b>
<b>1. Personal hub.docker.com registry</b>	<b>4</b>
<b>2. Introduction</b>	<b>5</b>
<b>3. IaC</b>	<b>6</b>
<b>4. Demonstration</b>	<b>7</b>
<b>5. Conclusions</b>	<b>8</b>

## 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

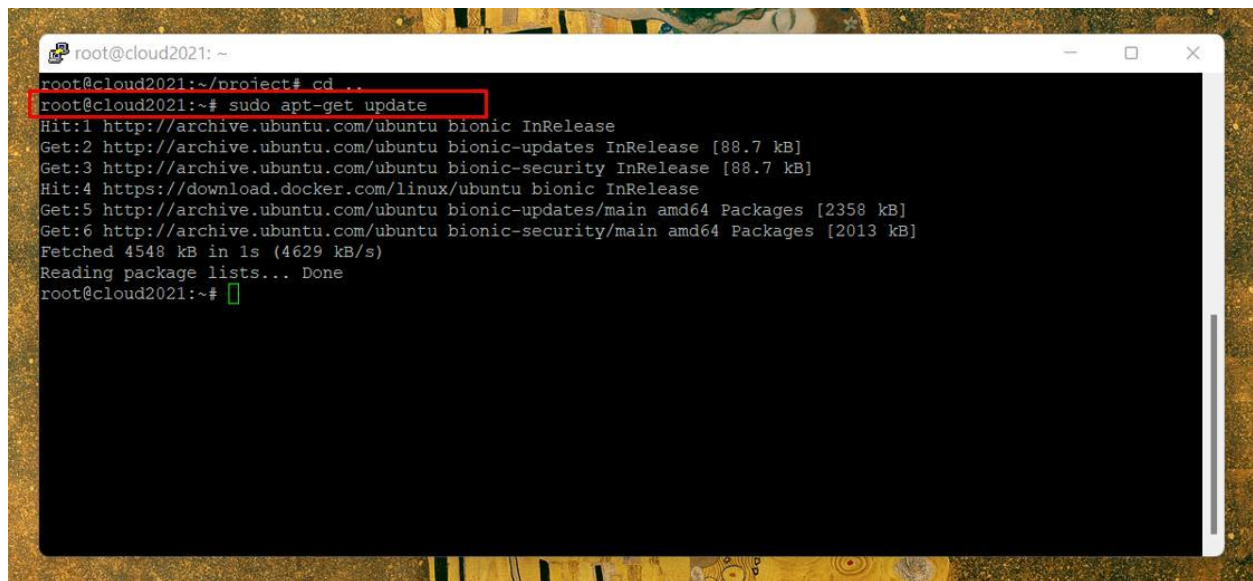
The advantages of parallel processing when dealing with Big Data are countless. Actually it is almost a necessity. Nowadays it is widely accepted that distributed systems (many machines realizing parallel processing) overcome local systems (single machines) in many different modalities. For instance, 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.

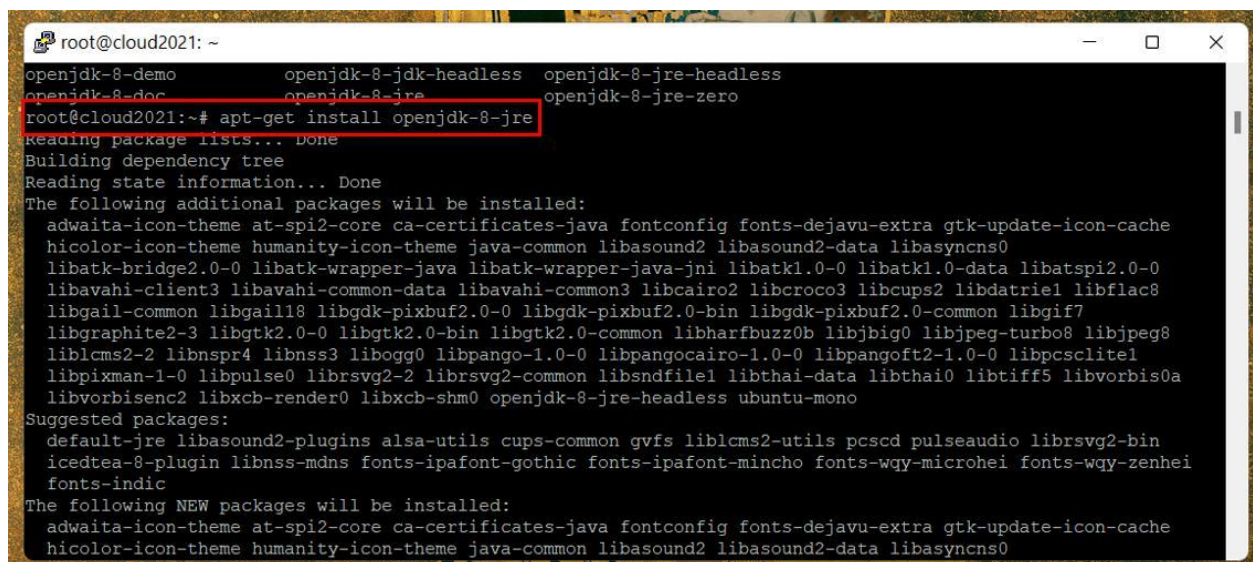
## OBS - Pre-Setup

### Updating packages and installing Java

A terminal window titled 'root@cloud2021: ~' showing the execution of 'sudo apt-get update'. The command is highlighted with a red box. The output shows several package lists being updated from various sources, including archive.ubuntu.com and download.docker.com. The process concludes with 'Reading package lists... Done' and a prompt for the next command.

```
root@cloud2021:~/project# cd ..
root@cloud2021:~# sudo apt-get update
Hit:1 http://archive.ubuntu.com/ubuntu bionic InRelease
Get:2 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:3 http://archive.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Hit:4 https://download.docker.com/linux/ubuntu bionic InRelease
Get:5 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [2358 kB]
Get:6 http://archive.ubuntu.com/ubuntu bionic-security/main amd64 Packages [2013 kB]
Fetched 4548 kB in 1s (4629 kB/s)
Reading package lists... Done
root@cloud2021:~#
```

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.

A terminal window titled 'root@cloud2021: ~' showing the execution of 'apt-get install openjdk-8-jre'. The command is highlighted with a red box. The output lists several additional packages to be installed along with the requested Java package. The process concludes with 'Reading state information... Done' and a list of suggested and new packages.

```
root@cloud2021:~# apt-get install openjdk-8-jre
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  adwaita-icon-theme at-spi2-core ca-certificates-java fontconfig fonts-dejavu-extra gtk-update-icon-cache
  hicolor-icon-theme humanity-icon-theme java-common libasound2 libasound2-data libasound2-plugins
  libatk-bridge2.0-0 libatk-wrapper-java libatk-wrapper-java-jni libatk1.0-0 libatk1.0-data libatspi2.0-0
  libavahi-client3 libavahi-common-data libavahi-common3 libcairo2 libcups2 libdatrie1 libflac8
  libgail-common libgail18 libgdk-pixbuf2.0-0 libgdk-pixbuf2.0-bin libgdk-pixbuf2.0-common libgif7
  libgraphite2-3 libgtk2.0-0 libgtk2.0-bin libgtk2.0-common libharfbuzz0b libjpeg8 libjs-jquery libjs-jquery-ui
  liblcms2-2 liblensfun0 libnss3 libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0 libpcsclite1
  libpixman-1-0 libpulse0 librsvg2-2 librsvg2-common libsndfile1 libthai-data libthai0 libtiff5 libvorbis0a
  libvorbisenc2 libxcb-render0 libxcb-shm0 openjdk-8-jre-headless ubuntu-mono
Suggested packages:
  default-jre libasound2-plugins alsa-utils cups-common gvfs liblcms2-utils pcsd pulseaudio librsvg2-bin
  icedtea-8-plugin libnss-mdns fonts-ipafont-gothic fonts-ipafont-mincho fonts-wqy-microhei fonts-wqy-zenhei
  fonts-indic
The following NEW packages will be installed:
  adwaita-icon-theme at-spi2-core ca-certificates-java fontconfig fonts-dejavu-extra gtk-update-icon-cache
  hicolor-icon-theme humanity-icon-theme java-common libasound2 libasound2-data libasound2-plugins
  libatk-bridge2.0-0 libatk-wrapper-java libatk-wrapper-java-jni libatk1.0-0 libatk1.0-data libatspi2.0-0
  libavahi-client3 libavahi-common-data libavahi-common3 libcairo2 libcups2 libdatrie1 libflac8
  libgail-common libgail18 libgdk-pixbuf2.0-0 libgdk-pixbuf2.0-bin libgdk-pixbuf2.0-common libgif7
  libgraphite2-3 libgtk2.0-0 libgtk2.0-bin libgtk2.0-common libharfbuzz0b libjpeg8 libjs-jquery libjs-jquery-ui
  liblcms2-2 liblensfun0 libnss3 libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0 libpcsclite1
  libpixman-1-0 libpulse0 librsvg2-2 librsvg2-common libsndfile1 libthai-data libthai0 libtiff5 libvorbis0a
  libvorbisenc2 libxcb-render0 libxcb-shm0 openjdk-8-jre-headless ubuntu-mono
```

## Downloading and unzipping Spark

```
root@cloud2021: ~/project
root@cloud2021:~/project# wget https://downloads.apache.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop3.2.tgz
--2022-01-12 22:29:33-- https://downloads.apache.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop3.2.tgz
resolving downloads.apache.org (downloads.apache.org)... 88.99.95.219, 135.181.214.104, 2a01:4f9:3a:2c57::2, ...
connecting to downloads.apache.org (downloads.apache.org)|88.99.95.219|:443... connected.
HTTP request sent, awaiting response... 200 OK
length: 300965906 (287M) [application/x-gzip]
saving to: 'spark-3.2.0-bin-hadoop3.2.tgz'

spark-3.2.0-bin-hadoop3.2.tgz 100%[=====>] 287.02M 79.8MB/s in 3.7s

2022-01-12 22:29:37 (78.0 MB/s) - 'spark-3.2.0-bin-hadoop3.2.tgz' saved [300965906/300965906]

root@cloud2021:~/project#
```

```
root@cloud2021: ~/project
root@cloud2021:~/project# ls
docker-compose.yml  spark-3.2.0-bin-hadoop3.2.tgz
root@cloud2021:~/project# tar -xvzf spark-3.2.0-bin-hadoop3.2.tgz
spark-3.2.0-bin-hadoop3.2/
spark-3.2.0-bin-hadoop3.2/NOTICE
spark-3.2.0-bin-hadoop3.2/kubernetes/
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/python_executable_check.py
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/autoscale.py
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/worker_memory_check.py
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/py_container_checks.py
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/decommissioning.py
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/pyfiles.py
spark-3.2.0-bin-hadoop3.2/kubernetes/tests/decommissioning_cleanup.py
spark-3.2.0-bin-hadoop3.2/kubernetes/dockerfiles/
spark-3.2.0-bin-hadoop3.2/kubernetes/dockerfiles/spark/
spark-3.2.0-bin-hadoop3.2/kubernetes/dockerfiles/spark/decom.sh
spark-3.2.0-bin-hadoop3.2/kubernetes/dockerfiles/spark/entrypoint.sh
spark-3.2.0-bin-hadoop3.2/kubernetes/dockerfiles/spark/bindings/
```

## 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
```



```
root@cloud2021: ~  
root@cloud2021:~# docker pull bitnami/spark  
Using default tag: latest  
latest: Pulling from bitnami/spark  
0796bf144e3f: Pull complete  
184a52ee80df: Pull complete  
a9e2002d25f3: Pull complete  
30346c57455c: Pull complete  
069bb1922da7: Pull complete  
347d572cbe3f: Pull complete  
aa5e275f3499: Pull complete  
c2694143feec: Pull complete  
11778f32f94e: Pull complete  
95109c20b50f: Extracting [=====>] 452.6MB/452.6MB  
█
```

## Creating the docker compose file

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

```
root@cloud2021: ~/project project folder
GNU nano 2.9.3 docker-compose.yml
version: '2'

services:
  spark: 1 - master node container
    image: docker.io/bitnami/spark:3 2 - spark image pulled from dockerhub
    environment:
      - SPARK_MODE=master
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
    ports:
      - '8080:8080' 3 - host port : container port
    volumes:
      - ".:test-files:rw" 4 - adding persistence to the container
  spark-worker: 5
    image: docker.io/bitnami/spark:3
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark:7077
      - SPARK_WORKER_MEMORY=1G
      - SPARK_WORKER_CORES=1
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no 6 - spark environment setups

[ Read 27 lines ]
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify    ^C Cur Pos
^X Exit      ^R Read File  ^\ Replace   ^U Uncut Text ^T To Spell   ^_ Go To Line
```

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**

```
root@cloud2021: ~/project
root@cloud2021:~/project# ls
docker-compose.yml  spark-3.2.0-bin-hadoop3.2
root@cloud2021:~/project# docker-compose up -d
Pulling spark (docker.io/bitnami/spark:3)...
3: Pulling from bitnami/spark
0796bf144e3f: Already exists
184a52ee80df: Already exists
a9e2002d25f3: Already exists
30346c57455c: Already exists
069bb1922da7: Already exists
347d572cbe3f: Already exists
aa5e275f3499: Already exists
c2694143feec: Already exists
11778f32f94e: Already exists
95109c20b50f: Already exists
Pulling spark-worker (docker.io/bitnami/spark:3)...
3: Pulling from bitnami/spark
Starting project_spark_1_53eca0f3a080 ... done
Starting project_spark-worker_1_cef9b5395406 ... done
root@cloud2021:~/project#
```

## 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.

```
root@cloud2021: ~/project
root@cloud2021:~/project# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
edf4238267cf   bitnami/spark:3  "/opt/bitnami/script..." 3 hours ago    Up 14 minutes  0.0.0.0:8080->8080/tcp    project_spark_1_53eca0f3a080
6a0843e20d04   bitnami/spark:3  "/opt/bitnami/script..." 3 hours ago    Up 14 minutes                    project_spark-worker_1_cef9b5395406
```

7 - listing currently open containers

8 - spark master node

9 - spark slave node

In our case, our two containers were displayed, signaling 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\\_address:connection\\_port](http://host_address: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>

Spark Master at [51.83.123.133:8080](http://51.83.123.133:8080)

URL: [spark://edf4238267cf:7077](http://spark://edf4238267cf:7077)

Alive Workers: 1 **Refers to our slave node, running on the other container**

Cores in use: 1 Total, 0 Used

Memory in use: 1024.0 MiB Total, 0.0 B Used

Resources in use:

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20220112180123-172.18.0.2-34829	172.18.0.2:34829	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	

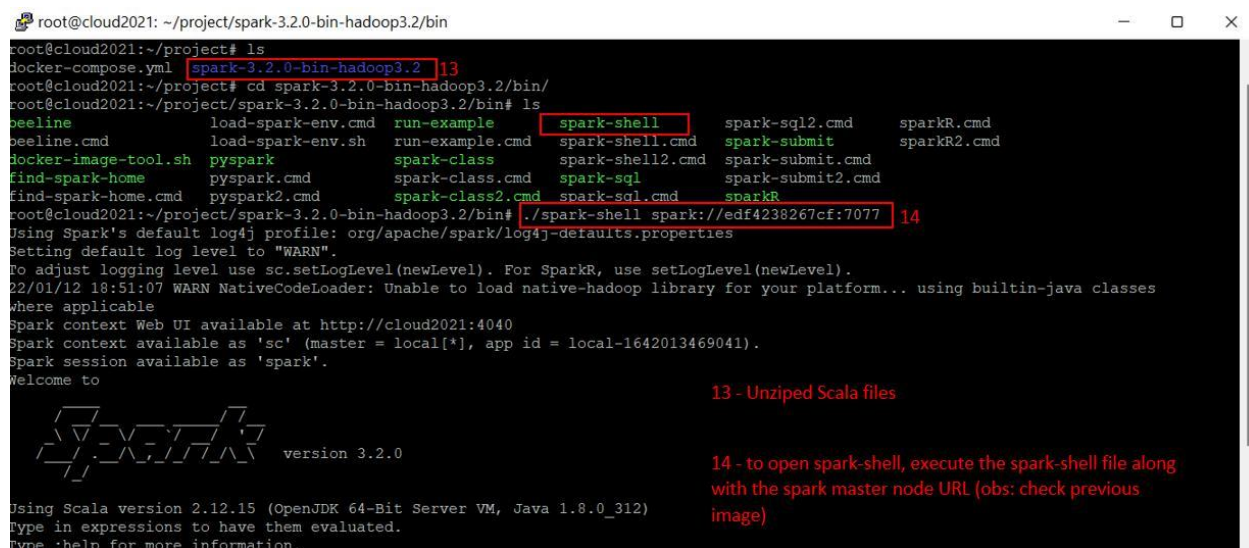
## Accessing the spark-shell

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`



```
root@cloud2021: ~/project/spark-3.2.0-bin-hadoop3.2/bin
root@cloud2021:~/project# ls
docker-compose.yml spark-3.2.0-bin-hadoop3.2 13
root@cloud2021:~/project# cd spark-3.2.0-bin-hadoop3.2/bin/
root@cloud2021:~/project/spark-3.2.0-bin-hadoop3.2/bin# ls
beeline load-spark-env.cmd run-example spark-shell spark-submit sparkR
beeline.cmd load-spark-env.sh run-example.cmd spark-shell.cmd spark-submit sparkR2.cmd
docker-image-tool.sh pyspark spark-class spark-shell2.cmd spark-submit2.cmd
find-spark-home pyspark.cmd spark-class2.cmd spark-submit2.cmd
find-spark-home.cmd pyspark2.cmd spark-class2.cmd spark-submit2.cmd
root@cloud2021:~/project/spark-3.2.0-bin-hadoop3.2/bin# ./spark-shell spark://edf4238267cf:7077 14
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/01/12 18:51:07 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
Spark context Web UI available at http://cloud2021:4040
Spark context available as 'sc' (master = local[*], app id = local-1642013469041).
Spark session available as 'spark'.
Welcome to

  13 - Unzipped Scala files

  14 - to open spark-shell, execute the spark-shell file along
with the spark master node URL (obs: check previous
image)

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 1.8.0_312)
Type in expressions to have them evaluated.
Type :help for more information.
```

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.

```
root@cloud2021: ~/project
root@cloud2021:~/project# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
62c68e3ecbf5       bitnami/spark:3    "/opt/bitnami/script..." About an hour ago   Up About an hour   0.0.0.0:8080->8080/tcp   project_spark-worker_1_c2c7de3e3022
643c2771e8ce       bitnami/spark:3    "/opt/bitnami/script..." About an hour ago   Up About an hour   0.0.0.0:8080->8080/tcp   project_spark_1_cc33d5744c30

root@cloud2021:~/project# docker exec -i -t 643c2771e8ce bash
I have no name!@643c2771e8ce:/opt/bitnami/spark$ ls
LICENSE NOTICE R README.md RELEASE bin conf data examples jars kubernetes licenses logs python sbin tmp venv work yarn
I have no name!@643c2771e8ce:/opt/bitnami/spark$ cd bin
I have no name!@643c2771e8ce:/opt/bitnami/spark/bin$ ls
baseline      find-spark-home      load-spark-env.sh  pyspark2.cmd      spark-class      spark-shell      spark-sql      spark-submit      sparkR
baseline.cmd  find-spark-home.cmd  pyspark            run-example       spark-class.cmd  spark-shell.cmd  spark-sql.cmd  spark-submit.cmd  sparkR.cmd
docker-image-tool.sh  load-spark-env.cmd  pyspark.cmd       run-example.cmd   spark-class2.cmd spark-shell2.cmd  spark-sql2.cmd spark-submit2.cmd sparkR2.cmd
I have no name!@643c2771e8ce:/opt/bitnami/spark/bin$
```

```
root@cloud2021: ~/project/spark-3.2.0-bin-hadoop3.2/bin

scala> println("Hello World!")
Hello World!

scala>
```

Feel free to execute your codes, import your datasets,  
and run your scripts!

## 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**.

```
root@cloud2021: ~/project
root@cloud2021:~/project# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
edf4238267cf        bitnami/spark:3     "/opt/bitnami/script..." 4 hours ago         Up About an hour    0.0.0.0:8080->8080/tcp    project_spark_1_53eca0f3a080
8a0843e20d04        bitnami/spark:3     "/opt/bitnami/script..." 4 hours ago         Up About an hour    0.0.0.0:8080->8080/tcp    project_spark-worker_1_cef9b5
895406

root@cloud2021:~/project# docker exec -i -t edf4238267cf bash 15
I have no name!@edf4238267cf:/opt/bitnami/spark$ ls
LICENSE NOTICE README.md RELEASE bin conf data examples jars kubernetes licenses logs python sbin tmp venv work yarn
I have no name!@edf4238267cf:/opt/bitnami/spark$ cd /test-files/ 16
I have no name!@edf4238267cf:/test-files$ ls
docker-compose.yml spark-3.2.0-bin-hadoop3.2 17
I have no name!@edf4238267cf:/test-files$
```

15 – This executes the docker container of the spark master node and opens a terminal in it

16 – From that terminal we can enter the "test-files" written in the docker-compose file.

17 - Notice that the test-files folder has the same files of the project folder of our machine. Keep in mind that any alteration in these files happens in both folders!

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.