:imagesdir: images

= Big Data Processing with Docker and Hadoop

PURPOSE: This chapter explains how to use Docker to create a Hadoop cluster and a Big Data application in Java. It highlights several concepts like service scale, dynamic port allocation, container links, integration tests, debugging, etc.

Big Data applications usually involve distributed processing using tools like Hadoop or Spark. These services can be scaled up, running with several nodes to support more parallelism. Running tools like Hadoop and Spark on Docker makes it easy to scale them up and down. This is very useful to simulate a cluster on development time and also to run integration tests before taking your application to production.

The application on this example reads a file, count how many words are on that file using a MapReduce job implemented on Hadoop and then saves the result on a MongoDB database. In order to do that, we will run a Hadoop cluster and a MongoDB server on Docker.

[NOTE]

Apache Hadoop is an open-source software framework

used for distributed storage and processing of big data sets using the MapReduce programming model. The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster. It then transfers packaged code into nodes to process the data in parallel. The Hadoop framework itself is mostly written in Java.

== Clone the sample application

Clone the project at `https://github.com/fabianenardon/hadoop-docker-demo`

Inspect the `docker/docker-compose.yml` file. It defines a MongoDB service and the services needed to run a Hadoop cluster. It also defines a service for our application. See how the services are linked together.

== Build the application

[source, text]

cd sample
mvn clean install -Papp-docker-image

In the command above, `-Papp-docker-image` will fire up the `app-docker-image` profile, defined in the application `pom.xml`. This profile will create a dockerized version of the application, creating two images:

. `docker-hadoop-example`: docker image used to run the application
. `docker-hadoop-example-tests`: docker image used to run integration tests

== Start all the services

Go to the `sample/docker` folder and start the services:

[source, text]

cd docker

docker-compose up -d

See the logs and wait until everything is up:

[source, text]

docker-compose logs -f

In order to see if everything is up, open `http://localhost:8088/cluster`. You should see 1 active node when everything is up and running.

image::docker-bigdata-03.png[]

== Running the application

This application reads a text file from HDFS and counts how many words it has. The result is saved on MongoDB.

First, create a folder on HDFS. We will save the file to be processed on it:

[source, text]

docker-compose exec yarn hdfs dfs -mkdir /files/

In the command above, we are executing `hdfs dfs -mkdir /files/` on the service `yarn`. This command creates a new folder called `/files/` on HDFS, the distributed file system used by Hadoop.

Put the file we are going to process on HDFS:

[source, text]

docker-compose run docker-hadoop-example
hdfs dfs -put /maven/test-data/text_for_word_count.txt /files/

The `text_for_word_count.txt` file was added to the application image by maven when we built it, so we can use it to test. The command above will transfer the `text_for_word_count.txt` file from the local disk to the `/files/` folder on HDFS, so the Hadoop process can access it.

Run our application

[source, text]

docker-compose run docker-hadoop-example

hadoop jar /maven/jar/docker-hadoop-example-1.0-SNAPSHOT-mr.jar

hdfs://namenode:9000 /files mongo yarn:8050

The command above will run our jar file on the Hadoop cluster. The `hdfs://namenode:9000` parameter is the HDFS address. The `/files` parameter is where the file to process can be found on HDFS. The `mongo` parameter is the MongoDB host address. The `yarn:8050` parameter is the Hadoop yarn address, where the MapReduce job will be deployed. Note that since we are running the Hadoop components (namenode, yarn), MongoDB and our application as Docker services, they can all find each other and we can use the service names as host addresses.

If you go to `http://localhost:8088/cluster`, you can see your job running. When the job finishes, you should see this:

image::docker-bigdata-04.png[]

If everything ran successful, you should be able to see the results on MongoDB.

Connect to the Mongo container:

[source, text]

docker-compose exec mongo mongo

When connected, type:

[source, text]

use mongo_hadoop

db.word_count.find();

You should see the results of running the application. Something like this:

[source, text]

> *db.word_count.find();*
>
> *{ "_id" : "Counts on Sat Mar 18 18:16:20 UTC 2017", "words" : 256 }*

== Scaling the Hadoop cluster

If you want, you can scale your cluster, adding more Hadoop nodes to it:

[source, text]

docker-compose scale nodemanager=2

This means that you want to have 2 nodes in your Hadoop cluster. Go to `http://localhost:8088/cluster` and refresh until you see 2 active nodes.

The trick to scale the nodes is to use dynamically allocated ports and let docker assign a different port to each new nodemanager. See this approach in this snippet of the `docker-compose.yml` file:

[source, text]

nodemanager:

image: tailtarget/hadoop:2.7.2

command: yarn nodemanager

ports:

- “8042” # local port dynamically assigned. allows node to be scaled up and down

links:

- namenode

- datanode

- yarn

hostname: nodemanager

== Stopping the services

Stop all the services

[source, text]

docker-compose down

Note that since our `docker-compose.yml` file defines volume mappings for HDFS and MongoDB, next time you start the services again, your data will still be there.

== Debugging your code

Debugging distributed Hadoop applications can be cumbersome. However, you can configure your environment to use the docker Hadoop cluster and debug your code easily from an IDE.

First, make sure your services are up:

[source, text]

docker-compose up -d

Then, add this to your /etc/hosts:

[source, text]

127.0.0.1 datanode
127.0.0.1 yarn

127.0.0.1 namenode

127.0.0.1 secondarynamenode

127.0.0.1 nodemanager

This configuration will allow you to access the docker Hadoop cluster from your IDE.

Then, open your project on Netbeans (or any other IDE) and run the application file:

image::docker-bigdata-01.png[]

Note that you will be connecting to the docker services at localhost.

You can also set a breakpoint and debug your application:

image::docker-bigdata-02.png[]

== Integration tests

When running integration tests, you want to test your application in an environment as close to production as possible, so you can test interactions between the several components, services, databases, network communication, etc. Fortunately, docker can help you a lot with integration tests.

There are several strategies to run integration tests, but in this application we are going to use the following:

. Start the services with a `docker-compose.yml` file created for testing purposes. This file won't have any volumes mapped, so when the test is over, no state will be saved. The test `docker-compose.yml` file won't publish any port on the host machine, so we can run simultaneous tests.
. Run the application, using the services started with the `docker-compose.yml` test file.
. Run Maven integration tests to check if the application execution produced the expected results. This will be done by checking what was saved on the MongoDB database.
. Stop the services. No state will be stored, so next time you run the integration tests, you will have a clean environment.

Here is how to execute this strategy, step by step:

Start the services with the test configuration:

[source, text]

docker-compose --file src/test/resources/docker-compose.yml up -d

Make sure all services are started and create the folder we need on hdfs to test:

[source, text]

docker-compose --file src/test/resources/docker-compose.yml exec yarn hdfs dfs -mkdir /files/

Put the test file on hdfs:

[source, text]

docker-compose --file src/test/resources/docker-compose.yml

run docker-hadoop-example

hdfs dfs -put /maven/test-data/text_for_word_count.txt /files/

Run the application

[source, text]

docker-compose --file src/test/resources/docker-compose.yml

run docker-hadoop-example

hadoop jar /maven/jar/docker-hadoop-example-1.0-SNAPSHOT-mr.jar

hdfs://namenode:9000 /files mongo yarn:8050

Run our integration tests:

[source, text]

docker-compose --file src/test/resources/docker-compose.yml

run docker-hadoop-example-tests mvn -f /maven/code/pom.xml

-Dmaven.repo.local=/m2/repository -Pintegration-test verify

Stop all the services:

[source, text]

docker-compose --file src/test/resources/docker-compose.yml down

If you want to remote debug tests, run the tests this way instead:

[source, text]

docker run -v ~/.m2:/m2 -p 5005:5005

–link mongo:mongo

–net resources_default

docker-hadoop-example-tests

mvn -f /maven/code/pom.xml

-Dmaven.repo.local=/m2/repository

-Pintegration-test verify

-Dmaven.failsafe.debug

Running with this configuration, the application will wait until an IDE connects for remote debugging on port 5005.

See more about integration tests in the link:./ch09-cicd.adoc[CI/CD using Docker] chapter