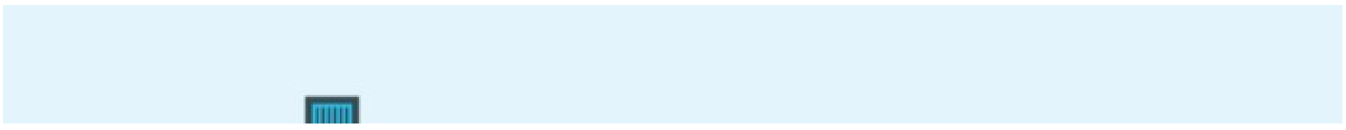
Creating a big data development platform using docker-compose

Mohammad Semnani
Nov 25, 2019 · 6 min read ★

We are at the age of Data! Big Big Data! Machines and applications are all around the world, generating tons of logs and data, changing the way of doing business, treating patients, navigating in streets, deciding, and in one sentence, changing the way of living.

Nowadays, many concepts and technologies have emerged to harness this fast paced horse and to store and manage data and computation over the cluster of servers (ex: Distributed File Systems, MapReduce, NoSql), changed and still changing how we develop software projects, and how we deploy them on the production environments (ex: Containers, Cloud computing, ServerLess).

One of the first steps of developing such technologies is to simulate the production environment on a local machine. In this article, I will discuss one way of preparing such an infrastructure using docker-compose and tackle the problems and challenges that may occur in front. Specifically, I will spin up HDFS, Hive, Spark, Hue, Zeppelin, Kafka, Zookeeper, and Streamsets on some Docker containers, using docker-compose. There are also some other options or alternatives to make such a platform out there. For example, you can spin up some VMs on a virtualization tool like VirtualBox or KVM, using Vagrant. Or you may use Kubernetes as your container orchestration instead. It mostly depends on your production environment and each one has it's own pros and cons which is outside the scope of this article.



In this article, it is assumed that you are already familiar with Docker and the technologies that are used in this article. It worth mentioning that the source code is available in Github.

. . .

Docker Compose

Regarding docker compose official website:

> *Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.*

For creating a docker-compose file, it is better to have these concerns in your mind:

- For using docker images in the compose file, you can either create your own docker image from scratch, or you can use a ready-to-use image from Docker Hub (Docker Hub is the world's largest library and community for container images). Albeit it is faster to find a working image on the docker hub.
- For finding a good image, it worth looking at its number of downloads. It shows how popular this image is.
- It is also worth looking at the last update date of the image in the Docker hub. It shows how new this image is!

- Also, pay attention to use compatible versions of each component in your docker-compose file. For example, if you are using Hadoop version 2.7, then for adding Spark, you need to add a compatible image spark-hadoop2.7. Or if you are using some components which are written in Scala, then use those components with the same Scala version. **It is really hard to resolve the problems that come from library incompatibilities!**
- Use specific version numbers for images. Repositories are always updating and new versions are available. So if the version number is not specified, the latest version will be used. In this case, maybe there were some changes in the latest version of your image which can break your platform.

In proceeding sections, we will add each component to the “docker-compose.yml” file, one by one.

. . .

HDFS

Here we won’t spin up Yarn as well, but just HDFS. For our Spark jobs, just stand-alone management is used.

For running HDFS, we used a predefined image from here. All the configuration is defined in hadoop-hive.env file. Also, a volume for keeping data is defined. The data in these mounted folders will not be removed when the docker containers are down or restarted.

Besides them, a docker network is defined named “pet_met”. This network is used to assign a static IP to docker containers. Some of its benefits will be presented in the proceeding sections. The default driver is of bridge type. In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. Different types of drivers are **bridge**, **host**, **overlay**, **macvlan**, and **none**. You can consult docker networking to get more information about them.

Visit http://localhost:50070/ to see NameNode UI.

```
version: '3'
```

```
version: "3"
services:

  namenode:
    image: bde2020/hadoop-namenode:2.0.0-hadoop2.7.4-java8
    container_name: namenode
    volumes:
      - /tmp/hdfs/namenode:/hadoop/dfs/name
    environment:
      - CLUSTER_NAME=test
    env_file:
      - ./hadoop-hive.env
    ports:
      - "50070:50070"
    networks:
      net_pet:
        ipv4_address: 172.27.1.5

  datanode:
    image: bde2020/hadoop-datanode:2.0.0-hadoop2.7.4-java8
    container_name: datanode
    volumes:
      - /tmp/hdfs/datanode:/hadoop/dfs/data
      - ./bank:/bank
    env_file:
      - ./hadoop-hive.env
    environment:
      SERVICE_PRECONDITION: "namenode:50070"
    depends_on:
      - namenode
    ports:
      - "50075:50075"
    networks:
      net_pet:
        ipv4_address: 172.27.1.6

networks:
  net_pet:
    ipam:
      driver: default
      config:
        - subnet: 172.27.0.0/16
```

docker-compose.yml_hdfs hosted with ♡ by GitHub view raw

• • •

Hive

For saving Hive metadata, you can use either an internal in-memory database or use an external DB docker image.

For the sake of simplicity, an internal in-memory database is, of course, the first choice. But when you restart your docker, all the data and metastore that you saved in your container will be lost! No surprise, This feature is intrinsic to docker.

But what is the solution? The solution is using an external database docker image and thanks to docker volumes, mount its data folder to somewhere in your local machine and it’s done! Here we used Postgresql as an external one.

```
hive-server:
    image: bde2020/hive:2.3.2-postgresql-metastore
    container_name: hive-server
    env_file:
      - ./hadoop-hive.env
    environment:
      HIVE_CORE_CONF_javax_jdo_option_ConnectionURL: "jdbc:postgresql://hive-metastore/metastor
      SERVICE_PRECONDITION: "hive-metastore:9083"
    ports:
      - "10000:10000"
    depends_on:
      - hive-metastore
    networks:
      net_pet:
        ipv4_address: 172.27.1.7

  hive-metastore:
    image: bde2020/hive:2.3.2-postgresql-metastore
    container_name: hive-metastore
    env_file:
      - ./hadoop-hive.env
    command: /opt/hive/bin/hive --service metastore
    environment:
      SERVICE_PRECONDITION: "namenode:50070 datanode:50075 hive-metastore-postgresql:5432"
    ports:
      - "9083:9083"
    depends_on:
      - hive-metastore-postgresql
    networks:
      net_pet:
        ipv4_address: 172.27.1.8

  hive-metastore-postgresql:
    image: bde2020/hive-metastore-postgresql:2.3.0
    container_name: hive-metastore-postgresql
    networks:
      net_pet:
        ipv4_address: 172.27.1.9
```

docker-compose.yml hive hosted with ♡ by GitHub view raw

. . .

Hue

Like Hive, for Hue, An external database image was used. This part of the code came from the Hue repository.

I had a problem while I wanted to use Hue's in-memory database. It kept complaining by message "Database is locked". After some googling, I came to the solution to use an external database.

```
  hue:
      image: gethue/hue:20191107-135001
      hostname: hue
      container_name: hue
      dns: 8.8.8.8
      ports:
      - "8888:8888"
      volumes:
        - ./hue-overrides.ini:/usr/share/hue/desktop/conf/z-hue.ini
      depends_on:
      - "database"
      networks:
        net_pet:
          ipv4_address: 172.27.1.13

  database:
      image: mysql:5.7
      container_name: database
      ports:
          - "33061:3306"
      command: --init-file /data/application/init.sql
      volumes:
          - /tmp/mysql/data:/var/lib/mysql
          - ./init.sql:/data/application/init.sql
      environment:
          MYSQL_ROOT_USER: root
          MYSQL_ROOT_PASSWORD: secret
          MYSQL_DATABASE: hue
          MYSQL_USER: root
          MYSQL_PASSWORD: secret
      networks:
        net_pet:
          ipv4_address: 172.27.1.14
```

docker-compose.yml_hue hosted with ♡ by GitHub view raw

But wait, you need to define a grant for Hue image to access external MySql image. If not, MySql will complain about an illegal access! That's what the init.sql file tries to resolve.

```
CREATE DATABASE IF NOT EXISTS hue;
grant all on hue.* to 'hue'@'172.27.1.13' identified by 'secretpassword';
grant all on hue.* to 'root'@'172.27.1.13' identified by 'secret';
flush privileges;
```

docker-compose.yml_init_sql hosted with ♡ by GitHub view raw

This is one place that defining static IP is handy! we can grant the access of hue user, from the Hue docker image, using the Hue static IP: 172.27.1.13. You can not use localhost or 127.0.0.1 or hue(As it is the hue container name) here. As you may know, images inside the docker-compose network are accessible by their names. For example, you can ping Hue from MySql container using this command: "ping hue". But when Hue wants to access MySql, It will do it using its IP.

visit http://localhost:8888/ to see Hue UI.

. . .

Apache Spark

As mentioned before, here we use standalone for our spark cluster management. For more information on different types of cluster managers, please refer to spark docs here.

```
spark-master:
    image: bde2020/spark-master:2.4.0-hadoop2.7
    image: bde2020/spark-master:2.4.0-hadoop2.7
    container_name: spark-master
    ports:
      - 8080:8080
      - 7077:7077
    environment:
      - CORE_CONF_fs_defaultFS=hdfs://namenode:8020
    env_file:
      - ./hadoop-hive.env
    networks:
      net_pet:
        ipv4_address: 172.27.1.10

  spark-worker:
    image: bde2020/spark-worker:2.4.0-hadoop2.7
    container_name: spark-worker
    depends_on:
      - spark-master
    environment:
      - SPARK_MASTER=spark://spark-master:7077
      - CORE_CONF_fs_defaultFS=hdfs://namenode:8020
      - HIVE_CORE_CONF_javax_jdo_option_ConnectionURL=jdbc:postgresql://hive-metastore/metastor
    depends_on:
      - spark-master
    ports:
      - 8081:8081
    env_file:
      - ./hadoop-hive.env
    networks:
      net_pet:
        ipv4_address: 172.27.1.11
```

docker-compose.yml_spark hosted with ♡ by **GitHub** **view raw**

One master and one worker is defined. Also, the address of NameNode is specified. If you want to add more workers, just add the worker snippet as many as you prefer. But remember to take care of the names (spark-worker-2 …).

visit http://localhost:8080/ to see SparkMaster UI.

...

Apache Zeppelin

Spark is the base engine for Zeppelin, so it is important to set Spark master url in Zeppelin. Also for connecting to HDFS and Hive, the NameNode URL should be specified as well.

visit http://localhost:19090/ to see Zeppelin UI.

```
 zeppelin:
   image: openkbs/docker-spark-bde2020-zeppelin
   container_name: zeppelin
   environment:
     CORE_CONF_fs_defaultFS: "hdfs://namenode:8020"
     SPARK_MASTER: "spark://spark-master:7077"
     MASTER: "spark://spark-master:7077"
     SPARK_MASTER_URL: spark://spark-master:7077
     ZEPPELIN_PORT: 8080
     ZEPPELIN_JAVA_OPTS:
       -Dspark.driver.memory=1g
       -Dspark.executor.memory=2g
   ports:
     - 19090:8080
   env_file:
     - ./hadoop-hive.env
   volumes:
     - /tmp/simple-demo/zeppelin/data:/usr/lib/zeppelin/data:rw
     - /tmp/simple-demo/zeppelin/notebook:/usr/lib/zeppelin/notebook:rw
   command: /usr/lib/zeppelin/bin/zeppelin.sh
   networks:
     net_pet:
       ipv4_address: 172.27.1.12
```

docker-compose.yml_zeppelin hosted with ♡ by GitHub view raw

. . .

Streamsets

It is pretty straight forward. Visit http://localhost:18630/ to see Streamsets UI.

```
streamsets:
  image: streamsets/datacollector:3.13.0-latest
  ports:
    - "18630:18630"
  networks:
    net_pet:
        ipv4_address: 172.27.1.17
```

docker-compose.yml_streamsets hosted with ♡ by GitHub view raw

. . .

Apache Kafka

To spin up Kafka, we need to run Zookeeper as well. You can bring up Zookeeper in the same container. But is is a best practice that each container be responsible for one single component.

The tricky part of running Kafka images is to set "KAFKA_ADVERTISED_HOST_NAME" correctly. Here is another place that our Static IP definition came to help.

```
zookeeper:
  image: wurstmeister/zookeeper:3.4.6
  ports:
    - "2181:2181"
  networks:
    net_pet:
      ipv4_address: 172.27.1.15

kafka:
  image: wurstmeister/kafka:2.12-2.3.0
  ports:
    - "9092:9092"
  environment:
    KAFKA_ADVERTISED_HOST_NAME: 172.27.1.16
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
  networks:
    net_pet:
      ipv4_address: 172.27.1.16
```

docker-compose.yml_kafka hosted with ♡ by GitHub view raw

. . .

Troubleshooting

- If some of your containers won't start, you may need to increase your docker resources (specifically the memory!). Having this number of containers up and running needs more resources than usual.
- If after a while, one of the containers stops working with no reason or changes in the source code, one reason could be a messy mounted volume! by clearing the mounted folder, it may back to a normal state.

. . .

Altogether

Congratulations! You have a working docker-compose that connects your Big data components together. You may stream your data to Kafka by StreamSets, then using Zeppelin, define some streaming jobs to move data to HDFS and Hive, and after doing some calculation by SparkSql, generate fancy reports in Zeppelin. Just run "docker-compose up" and enjoy!

Thanks to Kiarash Irandoust and Yves Sinkgraven.

Docker Docker Compose Big Data Hdfs Spark