# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 1

## Let’s build Apache Hadoop DFS cluster, bit by bit and see how it is all threaded together.

(20 August 2024)

**Overview**

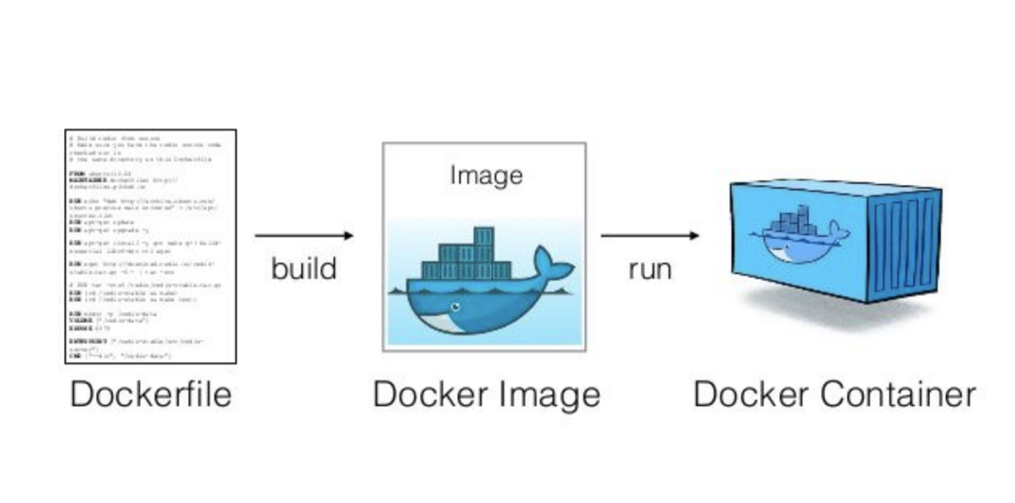
This all started with another [blog](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-1-e7c17d61b9d2) I was writing, where in I was exploring Kafka, kSqlDB, Apache Flink, Apache Hive, Apache Iceberg & Apache Paimon and various other supporting bits and bobs.

During that “rabbit hole” I ended needing to build “some…” docker images, leading to allot of learning along the way and redoing things over and over to make it easier, faster, simpler.

Figured others might be interested, find value out of it.

I’m by no means an expert, this is purely what I learned along the way to make life easier for myself.

We will start by building a basic docker image build ([Ubuntu](https://ubuntu.com/) 20.04), installing some basic OS packages, installing application server (based [OpenJDK 11](https://openjdk.org/projects/jdk/11/)) and [Apache Hadoop DFS cluster](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html), adding required configuration files (directly into the container during build or mounting them at run time) & dealing with environment variables.



During this phase, I will show 2 Dockerfile’s. One where I stage some files locally first and a 2nd where I pull the source files directly from the internet during the build.

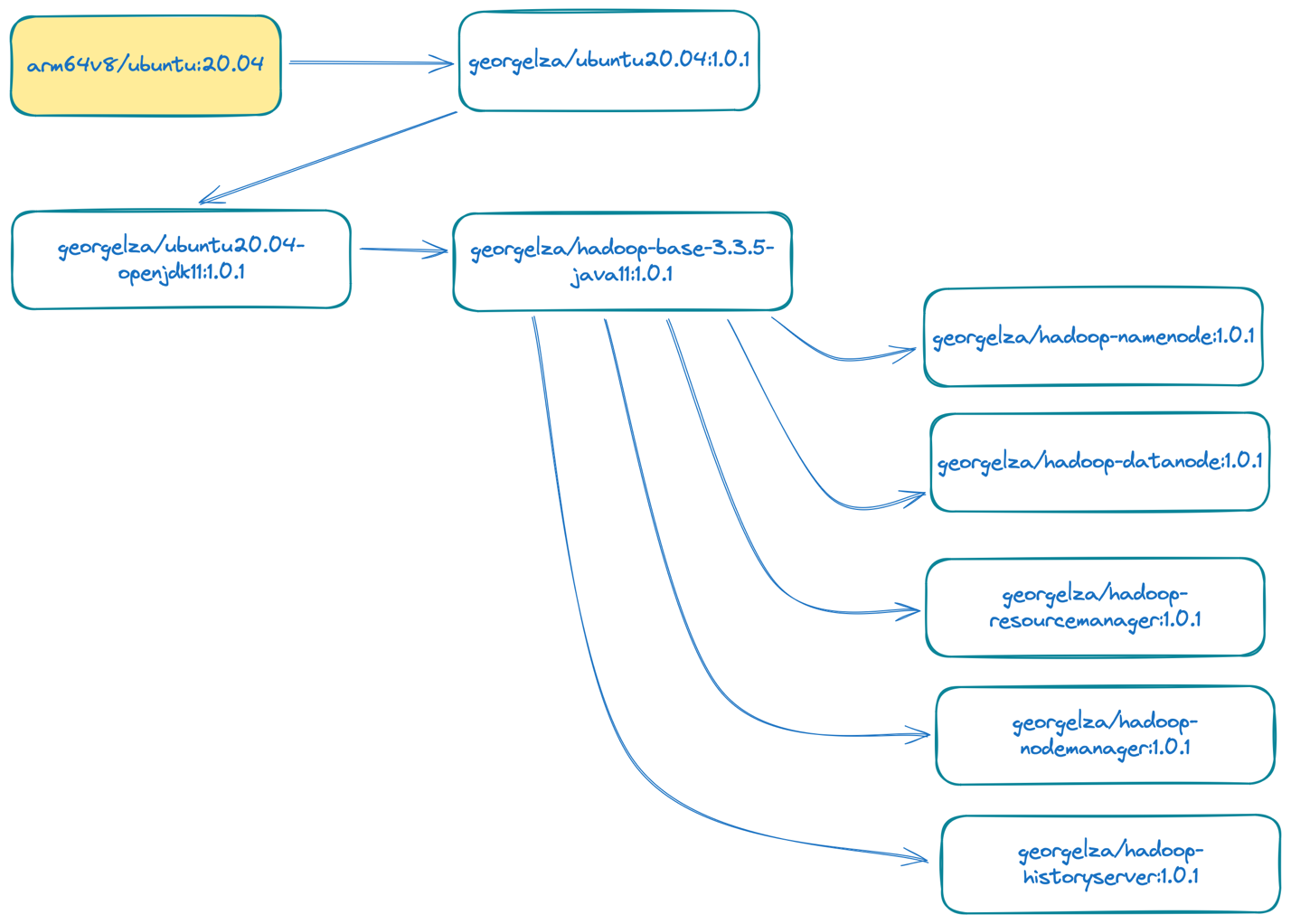
I will additionally discuss the importance of the build order in your [Dockerfile](https://docs.docker.com/reference/dockerfile/#:~:text=A%20Dockerfile%20is%20a%20text,line%20to%20assemble%20an%20image.). As a sidetrack we will also have a look at using [Makefiles](https://makefiletutorial.com/), they are a great way to “package “ all your commands together.

As a separate mini project, I will also show a docker image build using a multistage image build to reduce the image size and attack vectors.

To demonstrate all the above processes, we will build an [Apache Hadoop DFS](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html) cluster based on [Ubuntu 20.04,](https://ubuntu.com/) [OpenJDK 11](https://openjdk.org/projects/jdk/11/) & [Apache Hadoop](https://hadoop.apache.org/) 3.3.5, comprised out of:

* namenode
* nodemanager
* resourcemanager
* 5 datanodes &
* historyserver

In the below diagram I show how we go from our base ubuntu 20.04 image and build the various images out until we end with the HDFS cluster bottom right.



As it stands this will be a 7-part posting, but it’s by no means complete.

NOTE: I work on an Apple MacBook based on their [ARM64 aka AARC64](https://en.wikipedia.org/wiki/AArch64#:~:text=AArch64%20or%20ARM64%20is%20the,MPCore%20big.LITTLE%20CPU%20chip) architecture. Where needed I will point out which lines can be changed to make everything Intel/AMD64 compatible.

Good luck, as always, this is all fraught with rabbit holes, so many and you can disappear so easily… But it’s all fun and you will always discover something new or validate a previous learned skill.

See [Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git) for all the code.

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 2

## Let’s build Apache Hadoop DFS cluster.

(See: [Part 1](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-5-8463f7b25545))

(20 August 2024)

**Overview**

Ok we will start with our base OS image. This will form the foundation for the entire build.

See the build-ubuntu-os-20.04 sub directory.

NOTE: this is one of our ARM64 specific images, to revert to standard AMD64 (Intel), simply remove the arm64v8/ bit from the FROM clause to pull the Intel/AMD64 based image.

First Dockerfile

FROM arm64v8/ubuntu:20.04

# original https://github.com/YanYunNN/hadoop-cluster-docker-m1/blob/main/Dockerfile

WORKDIR /root

RUN echo "--> Install OS dependencies openssh-server & misc tools" && \

build\_deps="openssh-server wget neovim curl unzip net-tools" && \

apt-get update && \

apt-get install -y $build\_deps

RUN echo "--> Purge apt artifacts" && \

apt-get purge -y --auto-remove $build\_deps && \

apt-get clean && \

rm -rf /var/lib/apt/lists/\*

We start our build by specifying our source image, in this case arm64v8/ubuntu:20.04.

Source images are sourced from [Docker hub](http://hub.docker.com/) and [Google Container Registry](https://cloud.google.com/artifact-registry).

Next, we define WORKDIR /root this accomplishes 2 outcomes; it creates the directory if it does not exist, and it changes into the directory.

Following this we execute a docker primitive/command called RUN. I always start the command with echo “some text” to output a description of what’s being done, followed by “&& \” which creates a line continue onto the next line.

RUN echo "--> Install OS dependencies openssh-server & misc tools" && \

build\_deps="openssh-server wget neovim curl unzip net-tools" && \

apt-get update && \

apt-get install -y $build\_deps

With the second line we define a local variable “build\_deps” to which we assign a list of packages that we want to install using the apt-get install command:

build\_deps=”openssh-server wget neovim curl unzip net-tools”

The value of using a variable here is seen in the next RUN command where we clean up after the install and now instruct apt-get to clean up, using: apt-get purge – auto-remove &build\_deps variable/list define previously.

RUN echo "--> Purge apt artifacts" && \

apt-get purge -y --auto-remove $build\_deps && \

apt-get clean && \

rm -rf /var/lib/apt/lists/\*

NOTE: Lesson learned, the package vim during installation request’s physical location which can’t be by passed, neovim on the other hand not.

The build is executed by calling make build in the same directory, this in return issue:

sudo docker build -t ubuntu20.04:$(VERSION) .

To see the output image execute docker images

Ok, that’s all for Part 2, In the next part we will move onto building our base Open JDK 11 application server using our base image.

See [Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git) for all the code.

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 3

## Let’s build Apache Hadoop DFS cluster.

(See: [Part 2](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-5-8463f7b25545))

(20 August 2024)

**Overview**

This is going to be a rather short issue, below is the Dockerfile used to build our OpenJDK11 image.

We start of by using the FROM primitive, but this time we specify our previously build base OS image (georgelza/ubuntu20.04:1.0.1).

As can be seen, we pretty much have the same logic as discussed in part 2, except we changed the value assigned to build\_deps to our openjdk11 package.

The reason we do this as a separate build (vs including the OpenJDK11 as a package in the previous builds build\_deps) is to allow us to use our base OS image from part 2 in other builds, i.e. we might need to build a OpenJDK8 application server also, which then means we can simply re-use the OS image for both, with simply the relevant application server version changed.

The only addition here is we add an environment variable JAVA\_HOME. As this is added to this image all images that will use this image will inherit the variable.

Note: by including the repo owner (georgelza) in this case it allows me to upload mu images to hub.docker.com from where I can pull them when needed…

A Todo that’s now available, multi-platform builds. That’s where a single image name is assigned, but docker builds both AMD64 and ARM64 images… There are more Architectures that can be build. This then allows anyone that issues a pull command to pull the right image for the right architecture, but now that’s a total new Rabbit hole for another day/blog.

FROM georgelza/ubuntu20.04:1.0.1

WORKDIR /root

ENV JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-arm64

# Install some useful tools

RUN echo "--> install Open JDK 11 ARM64" && \

build\_deps="openjdk-11-jdk" && \

apt-get update && \

apt-get install -y $build\_deps

RUN echo "--> Purge apt artifacts" && \

apt-get purge -y --auto-remove $build\_deps && \

apt-get clean && \

rm -rf /var/lib/apt/lists/\*

To execute the build as per previous, execute “make build” in the “build-ubuntu-os-openjdk11” subdirectory. See the Makefile for the actual docker command executed. You will notice it is the same as the OS build except for the value assigned to image\_name variable that has changed.

We now have a base Ubuntu 20.04 OS, with some useful tools and OpenJDK 11 installed. This combination can be used for a large amount of the Apache projects available.

As per the README.md, we could consider using a smaller Ubuntu image like jammy… but that’s an exploration for another day. -> TODO

See you in Part 4 where we will start with the Apache Hadoop build.

**My Repo’s**

All the code used during this article will be available on the below GIT repo.

[Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git)

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 4

## Let’s build Apache Hadoop DFS cluster.

(See: [Part 3](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-5-8463f7b25545))

(20 August 2024)

**Overview**

And this is where it gets real…

We will first build a base image, using the previous Open JDK 11 image we build in Part 3. The base HDFS image is then utilized to build the various HDFS servers. Ye and here I am sure someone will tell me I can rather simplify it via different start up commands, using the same base but with different commands parameter to make it act as the different servers, but that’s for another day. For now, I did it like this, as it also shows some form of inheritance.

For building the base image I will show 2 versions… one where the install software is staged locally, and a 2nd where it is downloaded during the docker build processing using wget in the Dockerfile

We will cover how to pass in environment variables (I will show a docker-compose.yml snippet and a neat way that those are pushed into configuration values stored in files at startup time by the entrypoint.sh we copied into the image at build time, see the base/bin directory for this file.

Ok, let’s start.

Note the Hadoop binary that is utilized here is the ARM64/AARCH64 version. The 2 descriptions ARM64 and AARCH64 are generally used interchangeably by the industry.

The Intel/AMD64 version can be sourced from “ [hadoop-3.3.5.tar.gz](https://archive.apache.org/dist/hadoop/common/hadoop-3.3.5/hadoop-3.3.5.tar.gz) . Pay attention to the JAVA\_HOME… setting. I should probably rather make this more generic and not include -arm64, but it might have value down the line, if you ever echo’d the environment variable, you will immediately know it’s an ARM64/AARCH64 version/platform build.

Our Dockerfile

FROM georgelza/ubuntu20.04-openjdk11:1.0.1

USER root

RUN echo "--> set environment variable"

ENV JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-arm64

ENV HADOOP\_VERSION=3.3.5

ENV HADOOP\_HOME=/opt/hadoop-3.3.5

ENV HADOOP\_CONF\_DIR=/etc/hadoop

ENV MULTIHOMED\_NETWORK=1

ENV PATH=$HADOOP\_HOME/bin/:$PATH

WORKDIR /tmp/

COPY stage/hadoop-3.3.5-aarch64.tar.gz /tmp/

RUN echo "--> install Apache Hadoop 3.3.5" && \

tar -xzvf /tmp/hadoop-3.3.5-aarch64.tar.gz && \

mv /tmp/hadoop-3.3.5 /opt/ && \

rm /tmp/hadoop-\*

# Intel/AMD64 version

# https://archive.apache.org/dist/hadoop/common/hadoop-3.3.5/hadoop-3.3.5.tar.gz

RUN ln -s $HADOOP\_HOME/etc/hadoop $HADOOP\_CONF\_DIR

RUN echo "--> make Apache Hadoop directories" && \

mkdir -p $HADOOP\_HOME/logs && \

mkdir /hadoop-data

ADD bin/entrypoint.sh /entrypoint.sh

RUN chmod a+x /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]

The above COPY command source from the same directory where the Dockerfile is, so as our binary is in a sub directory called stage we copy from stage/<fiile> <target>. The ADD primitive used to copy the entrypoint.sh file is very similar to COPY.

Up next is our Modified Dockerfile where we use wget to download the install media during build time.

I travel allot, and at times during the travelling I’m building. While travelling I don’t always have access to good internet connectivity so what I do is download the install media (\*.tar.gz and \*.jar files) once, stage it into a local stage directory close to my Dockerfile, which then allows me to build and rebuild and rebuild without downloading the files every time… That’s one reason, another could simply be it’s faster, a 3rd and this is for the enterprise guys and their security departments…

They might have a rule that only allows them to build from installation media that the security department have first vetted.

I know a different way to accomplish the same might be to run a private docker image repo which is provisioned with security tooling to scan all images for vulnerabilities and there are loads of those, or it simply might be by staging the installation media on a common shared location anyone needing to build a similar images or rebuild this image can use the exact same install media.

FROM georgelza/ubuntu20.04-openjdk11:1.0.1

USER root

RUN echo "--> set environment variable"

ENV JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-arm64

ENV HADOOP\_VERSION=3.3.5

ENV HADOOP\_HOME=/opt/hadoop-3.3.5

ENV HADOOP\_CONF\_DIR=/etc/hadoop

ENV MULTIHOMED\_NETWORK=1

ENV PATH=$HADOOP\_HOME/bin/:$PATH

WORKDIR /tmp/

RUN echo "--> install Apache Hadoop 3.3.5" && \

wget https://archive.apache.org/dist/hadoop/common/hadoop-3.3.5/hadoop-3.3.5-aarch64.tar.gz && \

tar -xzvf /tmp/hadoop-3.3.5-aarch64.tar.gz && \

mv /tmp/hadoop-3.3.5 /opt/ && \

rm /tmp/hadoop-\*

# Intel/AMD64 version

# https://archive.apache.org/dist/hadoop/common/hadoop-3.3.5/hadoop-3.3.5.tar.gz

RUN ln -s $HADOOP\_HOME/etc/hadoop $HADOOP\_CONF\_DIR

RUN echo "--> make Apache Hadoop directories" && \

mkdir -p $HADOOP\_HOME/logs && \

mkdir /hadoop-data

ADD bin/entrypoint.sh /entrypoint.sh

RUN chmod a+x /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]

Both are built by executing the respective make <command> from our Makefile located in our build-hadoop-openjdk11 directory. For the locally staged software execute make buildbase.

For the wget based version I’ve added “make buildwgbase” to instruct the make to use the alternate Dockerfile called Dockerfile.wget. Notice how the docker command is modified to instruct it to use a specified input file and not the default “Dockerfile”.

I tend to find I prefer wget when the download is a single tar.gz file/bundle. If the file to be downloaded is a single jar file, or multiple jars as per the below then the curl command is more useful.

The below command does a couple of things for us. We start with a description of what are doing via the echo “description text”.

Next up we make a directory via (mkdir -p) under our current directory where we will down the file into and changing into the directory. Next is the curl command with -O telling curl to download the file to the current directory and lastly popd, this basically exits the current directory to where we were before the pushd command.

All the below is executed as a single command via the one RUN primitive and the usage of && \ at the end of each line that create all of this as a single line via the line continuation.

RUN echo "-> Install JARs: Flink's Kafka connector" && \

mkdir -p ./lib/kafka && pushd $\_ && \

curl [https://repo1.maven.org/maven2/org/apache/flink/flink-sql-connector-kafka/3.2.0-1.18/flink-sql-connector-kafka-3.2.0-1.18.jar -O && \](https://repo1.maven.org/maven2/org/apache/flink/flink-sql-connector-kafka/3.2.0-1.18/flink-sql-connector-kafka-3.2.0-1.18.jar%20-O%20&&%20\)

curl <https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-avro-confluent-registry/1.18.1/flink-sql-avro-confluent-registry-1.18.1.jar> -O && \

popd

I will discuss the entrypoint.sh in a later Part. Generally, we want to do the steps that very static as high up in the Dockerfile as possible to take advantage of docker cache’ing.

Think that will be it for part 4… In the next part we will create the various HDFS servers and discuss some of the scripts used to customize each server’s configuration.

**My Repo’s**

All the code used during this article will be available on the below GIT repo.

[Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git)

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 5

## Let’s build Apache Hadoop DFS cluster.

(See: [Part 4](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-5-8463f7b25545))

(20 August 2024)

**Overview**

OK… So we now have a base Ubuntu 20.04 image, we’ve added some useful tooling.

We then used that as a source and installed Open JDK 11.

After that we installed the Hadoop 3.3.5 distribution. This will be our base server from which we now derive our different HDFS servers making up our cluster.

At this point we want to be in build-hadoop-openjdk11-hdfs directory, where we will see the following sub directories.

* namenode
* datanode
* resourcemanager
* nodemanager
* historyserver

To build the various servers we can issue make build from our Makefile. You will notice in the Makefile this basically execute the docker build for each of the above using the following command.

build:

sudo docker build -t hadoop-namenode-$(HADOOP\_VERSION):$(VERSION) ./namenode

sudo docker build -t hadoop-datanode-$(HADOOP\_VERSION):$(VERSION) ./datanode

sudo docker build -t hadoop-resourcemanager-$(HADOOP\_VERSION):$(VERSION) ./resourcemanager

sudo docker build -t hadoop-nodemanager-$(HADOOP\_VERSION):$(VERSION) ./nodemanager

sudo docker build -t hadoop-historyserver-$(HADOOP\_VERSION):$(VERSION) ./historyserver

The above tells docker to build the Dockerfile located in the ./<directory name> directory

Lets take a peek into the namenode Dockerfile as a start.

FROM hadoop-base-3.3.5-java11:1.0.1

RUN echo "--> Build Hadoop HDFS Namenode"

HEALTHCHECK CMD curl -f http://localhost:9870/ || exit 1

ENV HDFS\_CONF\_dfs\_namenode\_name\_dir=file:///hadoop/dfs/warehouse

RUN mkdir -p /hadoop/dfs/warehouse

VOLUME /hadoop/dfs/warehouse

ADD bin/run.sh /run.sh

RUN chmod a+x /run.sh

EXPOSE 9870

CMD ["/run.sh"]

Nothing to special in there, except, we define a HEALTHCECK CMD command, and we expose the port via the EXPOSE 9870 primitive and then like previous we copy in a run.sh file.

Let’s look at another node, say the nodemanager Dockerfile.

FROM hadoop-base-3.3.5-java11:1.0.1

RUN echo "--> Build Hadoop HDFS Nodemanager"

HEALTHCHECK CMD curl -f http://localhost:8042/ || exit 1

ADD bin/run.sh /run.sh

RUN chmod a+x /run.sh

EXPOSE 8042

CMD ["/run.sh"]

As can be seen, very similar… pretty much the only difference is for the nodemanager we don’t define a directory and a volume and the port we’re EXPOSE is different.

One more, let’s look at the datanode Dockerfile.

FROM hadoop-base-3.3.5-java11:1.0.1

RUN echo "--> Build Hadoop HDFS Datanode"

HEALTHCHECK CMD curl -f http://localhost:9864/ || exit 1

ENV HDFS\_CONF\_dfs\_datanode\_data\_dir=file:///hadoop/dfs/warehouse

RUN mkdir -p /hadoop/dfs/warehouse

VOLUME /hadoop/dfs/warehouse

ADD bin/run.sh /run.sh

RUN chmod a+x /run.sh

EXPOSE 9864

CMD ["/run.sh"]

Ok, so this is starting to look like rinse and repeat, surely we can do this with some fancy environment/build variables, (and I’m sure someone with more experience is doing it that way) … so why are we building them different, like above, well the difference for each come in the run.sh file copied into the image.

NOTE: for a real production cluster we will have the /hadoop/dfs/warehouse directory located on dedicated highly available external storage.

Now, let’s think back a bit… remember the base image had a entrypoint.sh, what was the significance of that…

For that lets look at a docker compose.yml file used to stand up the cluster.

# docker-compose -p my-project up -d

#

services:

#### Hadoop / HDFS ####

#

# The Namenode UI can be accessed at http://localhost:9870/⁠ and

# the ResourceManager UI can be accessed at http://localhost:8089/⁠

namenode:

image: hadoop-namenode-3.3.5-java11:1.0.0

container\_name: namenode

volumes:

- ./data/hdfs/namenode:/hadoop/dfs/name

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

env\_file:

- ./hdfs/hadoop.env

ports:

- "9870:9870" # NameNode Web UI

resourcemanager:

image: hadoop-resourcemanager-3.3.5-java11:1.0.0

container\_name: resourcemanager

restart: on-failure

depends\_on:

- namenode

- datanode1

- datanode2

- datanode3

- datanode4

- datanode5

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

env\_file:

- ./hdfs/hadoop.env

ports:

- "8089:8088" # Resource Manager Web UI

historyserver:

image: hadoop-historyserver-3.3.5-java11:1.0.0

container\_name: historyserver

depends\_on:

- namenode

- datanode1

- datanode2

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

volumes:

- ./data/hdfs/historyserver:/hadoop/yarn/timeline

env\_file:

- ./hdfs/hadoop.env

ports:

- "8188:8188"

nodemanager1:

image: hadoop-nodemanager-3.3.5-java11:1.0.0

container\_name: nodemanager1

depends\_on:

- namenode

- datanode1

- datanode2

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

env\_file:

- ./hdfs/hadoop.env

ports:

- "8042:8042" # NodeManager Web UI

datanode1:

image: hadoop-datanode-3.3.5-java11:1.0.0

container\_name: datanode1

depends\_on:

- namenode

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

volumes:

- ./data/hdfs/datanode1:/hadoop/dfs/warehouse

env\_file:

- ./hdfs/hadoop.env

datanode2:

image: hadoop-datanode-3.3.5-java11:1.0.0

container\_name: datanode2

depends\_on:

- namenode

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

volumes:

- ./data/hdfs/datanode2:/hadoop/dfs/warehouse

env\_file:

- ./hdfs/hadoop.env

datanode3:

image: hadoop-datanode-3.3.5-java11:1.0.0

container\_name: datanode3

depends\_on:

- namenode

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

volumes:

- ./data/hdfs/datanode3:/hadoop/dfs/warehouse

env\_file:

- ./hdfs/hadoop.env

datanode4:

image: hadoop-datanode-3.3.5-java11:1.0.0

container\_name: datanode4

depends\_on:

- namenode

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

volumes:

- ./data/hdfs/datanode4:/hadoop/dfs/warehouse

env\_file:

- ./hdfs/hadoop.env

datanode5:

image: hadoop-datanode-3.3.5-java11:1.0.0

container\_name: datanode5

depends\_on:

- namenode

environment:

- CLUSTER\_NAME=${CLUSTER\_NAME}

volumes:

- ./data/hdfs/datanode5:/hadoop/dfs/warehouse

env\_file:

- ./hdfs/hadoop.env

# Without a network explicitly defined, you hit this Hive/Thrift error

# java.net.URISyntaxException Illegal character in hostname

# https://github.com/TrivadisPF/platys-modern-data-platform/issues/231

networks:

default:

name: ${COMPOSE\_PROJECT\_NAME}

Note: The project root has a .env file which populates COMPOSE\_PROJECT\_NAME & CLUSTER\_NAME variables.

First, notice each service defined have an env\_file: specified, pointing to a local ./hdfs/hadoop.env file.

Ok, let’s take a peek into this file:

CORE\_CONF\_fs\_defaultFS=hdfs://namenode:9000

CORE\_CONF\_hadoop\_http\_staticuser\_user=root

CORE\_CONF\_hadoop\_proxyuser\_hue\_hosts=\*

CORE\_CONF\_hadoop\_proxyuser\_hue\_groups=\*

CORE\_CONF\_io\_compression\_codecs=org.apache.hadoop.io.compress.SnappyCodec

CORE\_CONF\_dfs\_replication=2

HDFS\_CONF\_dfs\_webhdfs\_enabled=true

HDFS\_CONF\_dfs\_permissions\_enabled=false

HDFS\_CONF\_dfs\_namenode\_datanode\_registration\_ip\_\_\_hostname\_\_\_check=true

YARN\_CONF\_yarn\_log\_\_\_aggregation\_\_\_enable=true

YARN\_CONF\_yarn\_log\_server\_url=http://historyserver:8188/applicationhistory/logs/

YARN\_CONF\_yarn\_resourcemanager\_recovery\_enabled=true

YARN\_CONF\_yarn\_resourcemanager\_store\_class=org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore

YARN\_CONF\_yarn\_resourcemanager\_scheduler\_class=org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler

YARN\_CONF\_yarn\_scheduler\_capacity\_root\_default\_maximum\_\_\_allocation\_\_\_mb=8192

YARN\_CONF\_yarn\_scheduler\_capacity\_root\_default\_maximum\_\_\_allocation\_\_\_vcores=4

YARN\_CONF\_yarn\_resourcemanager\_fs\_state\_\_\_store\_uri=/rmstate

YARN\_CONF\_yarn\_resourcemanager\_system\_\_\_metrics\_\_\_publisher\_enabled=true

YARN\_CONF\_yarn\_resourcemanager\_hostname=resourcemanager

YARN\_CONF\_yarn\_resourcemanager\_address=resourcemanager:8032

YARN\_CONF\_yarn\_resourcemanager\_scheduler\_address=resourcemanager:8030

YARN\_CONF\_yarn\_resourcemanager\_resource\_\_tracker\_address=resourcemanager:8031

YARN\_CONF\_yarn\_timeline\_\_\_service\_enabled=true

YARN\_CONF\_yarn\_timeline\_\_\_service\_generic\_\_\_application\_\_\_history\_enabled=true

YARN\_CONF\_yarn\_timeline\_\_\_service\_hostname=historyserver

YARN\_CONF\_mapreduce\_map\_output\_compress=true

YARN\_CONF\_mapred\_map\_output\_compress\_codec=org.apache.hadoop.io.compress.SnappyCodec

YARN\_CONF\_yarn\_nodemanager\_resource\_memory\_\_\_mb=16384

YARN\_CONF\_yarn\_nodemanager\_resource\_cpu\_\_\_vcores=8

YARN\_CONF\_yarn\_nodemanager\_disk\_\_\_health\_\_\_checker\_max\_\_\_disk\_\_\_utilization\_\_\_per\_\_\_disk\_\_\_percentage=98.5

YARN\_CONF\_yarn\_nodemanager\_remote\_\_\_app\_\_\_log\_\_\_dir=/app-logs

YARN\_CONF\_yarn\_nodemanager\_aux\_\_\_services=mapreduce\_shuffle

MAPRED\_CONF\_mapreduce\_framework\_name=yarn

MAPRED\_CONF\_mapred\_child\_java\_opts=-Xmx4096m

MAPRED\_CONF\_mapreduce\_map\_memory\_mb=4096

MAPRED\_CONF\_mapreduce\_reduce\_memory\_mb=8192

MAPRED\_CONF\_mapreduce\_map\_java\_opts=-Xmx3072m

MAPRED\_CONF\_mapreduce\_reduce\_java\_opts=-Xmx6144m

MAPRED\_CONF\_yarn\_app\_mapreduce\_am\_env=HADOOP\_MAPRED\_HOME=/opt/hadoop-3.3.5/

MAPRED\_CONF\_mapreduce\_map\_env=HADOOP\_MAPRED\_HOME=/opt/hadoop-3.3.5/

MAPRED\_CONF\_mapreduce\_reduce\_env=HADOOP\_MAPRED\_HOME=/opt/hadoop-3.3.5/

So… what happens here, when the image is started as a container, these values are pushed into the running container and set as environment variables… ok… nice… but why is that special. Well next up is that entrypoint.sh file. See the local base/bin directory for that.

#!/bin/bash

# Set some sensible defaults

export CORE\_CONF\_fs\_defaultFS=${CORE\_CONF\_fs\_defaultFS:-hdfs://`hostname -f`:8020}

echo "fs\_defaultFS:"$CORE\_CONF\_fs\_defaultFS

function addProperty() {

local path=$1

local name=$2

local value=$3

local entry="<property><name>$name</name><value>${value}</value></property>"

local escapedEntry=$(echo $entry | sed 's/\//\\\//g')

sed -i "/<\/configuration>/ s/.\*/${escapedEntry}\n&/" $path

}

function configure() {

local path=$1

local module=$2

local envPrefix=$3

local var

local value

echo "Configuring $module"

for c in `printenv | perl -sne 'print "$1 " if m/^${envPrefix}\_(.+?)=.\*/' -- -envPrefix=$envPrefix`; do

name=`echo ${c} | perl -pe 's/\_\_\_/-/g; s/\_\_/@/g; s/\_/./g; s/@/\_/g;'`

var="${envPrefix}\_${c}"

value=${!var}

echo " - Setting $name=$value"

addProperty $path $name "$value"

done

}

configure /etc/hadoop/core-site.xml core CORE\_CONF

configure /etc/hadoop/hdfs-site.xml hdfs HDFS\_CONF

configure /etc/hadoop/yarn-site.xml yarn YARN\_CONF

configure /etc/hadoop/httpfs-site.xml httpfs HTTPFS\_CONF

configure /etc/hadoop/kms-site.xml kms KMS\_CONF

configure /etc/hadoop/mapred-site.xml mapred MAPRED\_CONF

if [ "$MULTIHOMED\_NETWORK" = "1" ]; then

echo "Configuring for multihomed network"

# HDFS

echo "Configuring HDFS"

addProperty /etc/hadoop/hdfs-site.xml dfs.namenode.rpc-bind-host 0.0.0.0

addProperty /etc/hadoop/hdfs-site.xml dfs.namenode.servicerpc-bind-host 0.0.0.0

addProperty /etc/hadoop/hdfs-site.xml dfs.namenode.http-bind-host 0.0.0.0

addProperty /etc/hadoop/hdfs-site.xml dfs.namenode.https-bind-host 0.0.0.0

addProperty /etc/hadoop/hdfs-site.xml dfs.client.use.datanode.hostname true

addProperty /etc/hadoop/hdfs-site.xml dfs.datanode.use.datanode.hostname true

# YARN

echo "Configuring YARN"

addProperty /etc/hadoop/yarn-site.xml yarn.resourcemanager.bind-host 0.0.0.0

addProperty /etc/hadoop/yarn-site.xml yarn.nodemanager.bind-host 0.0.0.0

addProperty /etc/hadoop/yarn-site.xml yarn.timeline-service.bind-host 0.0.0.0

# MAPRED

echo "Configuring MAPRED"

addProperty /etc/hadoop/mapred-site.xml yarn.nodemanager.bind-host 0.0.0.0

fi

if [ -n "$GANGLIA\_HOST" ]; then

echo "Configuring Ganglia"

mv /etc/hadoop/hadoop-metrics.properties /etc/hadoop/hadoop-metrics.properties.orig

mv /etc/hadoop/hadoop-metrics2.properties /etc/hadoop/hadoop-metrics2.properties.orig

echo "Configuring -module in mapred jvm rpc ugi-"

for module in mapred jvm rpc ugi; do

echo "$module.class=org.apache.hadoop.metrics.ganglia.GangliaContext31"

echo "$module.period=10"

echo "$module.servers=$GANGLIA\_HOST:8649"

done > /etc/hadoop/hadoop-metrics.properties

echo "Configuring -namenode datanode resourcemanager nodemanager mrappmaster jobhistoryserver-"

for module in namenode datanode resourcemanager nodemanager mrappmaster jobhistoryserver; do

echo "$module.sink.ganglia.class=org.apache.hadoop.metrics2.sink.ganglia.GangliaSink31"

echo "$module.sink.ganglia.period=10"

echo "$module.sink.ganglia.supportsparse=true"

echo "$module.sink.ganglia.slope=jvm.metrics.gcCount=zero,jvm.metrics.memHeapUsedM=both"

echo "$module.sink.ganglia.dmax=jvm.metrics.threadsBlocked=70,jvm.metrics.memHeapUsedM=40"

echo "$module.sink.ganglia.servers=$GANGLIA\_HOST:8649"

done > /etc/hadoop/hadoop-metrics2.properties

fi

function wait\_for\_it()

{

local serviceport=$1

local service=${serviceport%%:\*}

local port=${serviceport#\*:}

local retry\_seconds=5

local max\_try=100

let i=1

nc -z $service $port

result=$?

until [ $result -eq 0 ]; do

echo "[$i/$max\_try] check for ${service}:${port}..."

echo "[$i/$max\_try] ${service}:${port} is not available yet"

if (( $i == $max\_try )); then

echo "[$i/$max\_try] ${service}:${port} is still not available; giving up after ${max\_try} tries. :/"

exit 1

fi

echo "[$i/$max\_try] try in ${retry\_seconds}s once again ..."

let "i++"

sleep $retry\_seconds

nc -z $service $port

result=$?

done

echo "[$i/$max\_try] $service:${port} is available."

}

for i in ${SERVICE\_PRECONDITION[@]}

do

wait\_for\_it ${i}

done

exec $@

ok… wow… that’s confusing…

Well, the magic is all in a small section, lets copy that out here and have a look.

configure /etc/hadoop/core-site.xml core CORE\_CONF

configure /etc/hadoop/hdfs-site.xml hdfs HDFS\_CONF

configure /etc/hadoop/yarn-site.xml yarn YARN\_CONF

configure /etc/hadoop/httpfs-site.xml httpfs HTTPFS\_CONF

configure /etc/hadoop/kms-site.xml kms KMS\_CONF

configure /etc/hadoop/mapred-site.xml mapred MAPRED\_CONF

What this does is call the configure function and specifies 3 input variables. I’m not going to discuss the actual code from the configure function, but only high-level touch what is accomplished via it.

The first is an output file (i.e: /etc/hadoop/core-site.xml), the 2nd is a module, and the 3rd is the module prefix. The prefix is the first letters of the variable to be read/extract (i.e: CORE\_CONF, from the environment variables we injected into the container at startup via the env\_file=./hdfs/hadoop.env setting in the docker compose.yml, which is then written by the configure function to the specified output file.

With very little work this can be made to do same for other systems…

By using this our image stays static but allows us to add and remove variables changing the behavior and configuration of our cluster.

Well, there you have it, an [Apache Hadoop](https://hadoop.apache.org/) 3.3.5 DFS (Distributed File System) cluster build on [openJDK11](https://openjdk.org/projects/jdk/11/) on top of a Ubuntu 20.04 OS, with 5 data nodes, all from scratch.

(Sorry, the section was a bit long… but that was luckily primarily because we copied large sections of file into it…)

In the next section I will quickly touch the process of doing multistage Dockerfile’s.

**My Repo’s**

All the code used during this article will be available on the below GIT repo.

[Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git)

------------------------------------------------------------------------------------------------------------------

# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 6

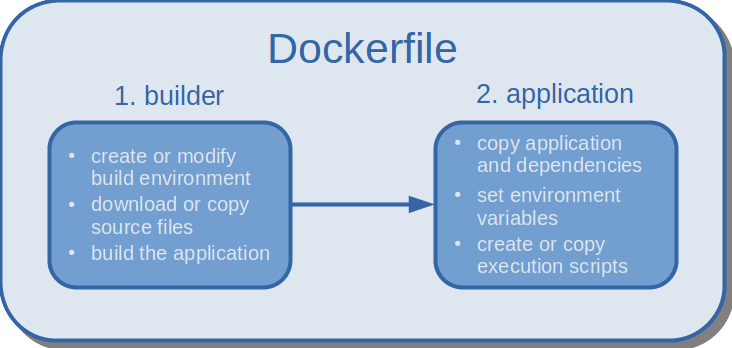
## Multistage docker image build.

(See: [Part 5](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-5-8463f7b25545))

(20 August 2024)

**Overview**

Ok… well this as per norm became a little more than originally planned but here it goes.



In the app directory is the code for a small [Go aka (Golang)](https://go.dev/) application. It’s simple, it’s going to print some information to the screen, to make it just slightly more interesting I’m importing a username and password from a .env file, the file is loaded by docker compose.yml as environment variables which are the pushed into the container at startup.

Next up are 5 Dockerfile’s called Dockerfile.1 -> Dockerfile.5

* First up is a single stage build, using golang:1.22

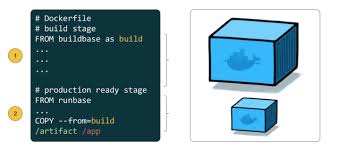
We start by using golang:1.22 as it has all the bits we require to build/compile Golang applications. This has the above advantages, but it also means the image is bloated/big, resulting in the image finally created containing allot of code/binaries/libraries which is not needed when we simply want to run a small compiled application.

Another benefit of using stages, in that it allows the developer to include a layer that say does unit testing, or lint the code. The image itself can also be started and instructed to only start up to a specified stage.

This is most probably what you will see 90% of the time, one stage, start to finish. Nothing wrong with it. It’s at times all that is needed.

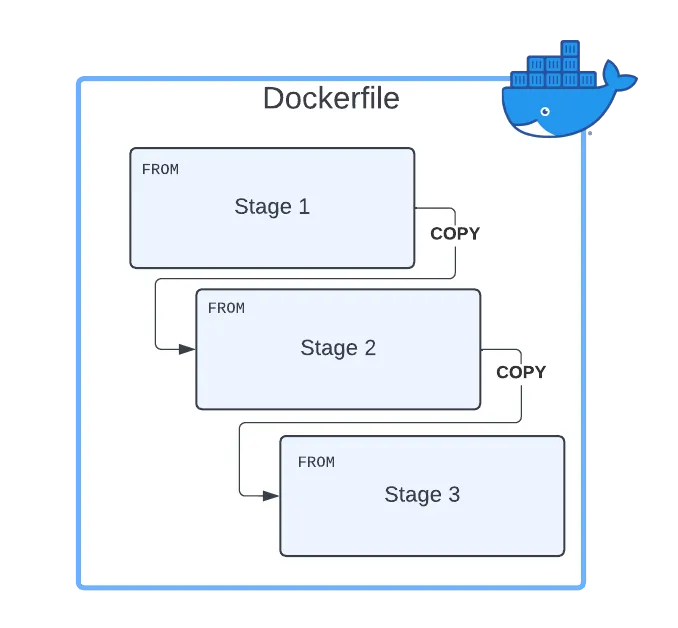
* FROM golang:1.22 AS builder
* ENV GOPROXY=https://proxy.golang.org,direct
* ENV GOSUMDB=sum.golang.org
* WORKDIR /app
* COPY go.mod .
* COPY go.sum .
* RUN go mod download
* COPY cmd/main.go ./cmd/
* RUN go build -v -o ./app/main ./cmd/main.go
* LABEL Author="George Leonard (georgelza@gmail.com)"
* CMD ["/app/main"]
* Next, we introduce the multistage concept. In this example we use the same image for both stages. For this example, I simply reused the same golang:1.22 source image.

During the 2nd stage we use the COPY –FROM=build /app/main . to copy the binary compiled from the builder stage into the production stage into our WORKDIR location.



* FROM golang:1.22 AS builder
* ENV GOPROXY=https://proxy.golang.org,direct
* ENV GOSUMDB=sum.golang.org
* WORKDIR /build
* COPY go.mod .
* COPY go.sum .
* RUN go mod download
* COPY cmd/main.go .
* RUN go build -v -o ./main ./main.go
* ###########START NEW IMAGE###################
* FROM golang:1.22 AS production
* LABEL Author="George Leonard (georgelza@gmail.com)"
* WORKDIR /app
* COPY --from=builder /build/main .
* CMD ["/app/main"]

Below we see that we don’t need to stick to 2 layers… you can chain them together, if it makes sense.



* Next up, we change things up, we start with one base image and then switch to a very general image (ubuntu:20.04) for the production stage/layer.
* FROM golang:1.22 AS builder
* ENV GOPROXY=https://proxy.golang.org,direct
* ENV GOSUMDB=sum.golang.org
* WORKDIR /build
* COPY go.mod .
* COPY go.sum .
* RUN go mod download
* COPY cmd/main.go .
* RUN go build -v -o ./main ./main.go
* ###########START NEW IMAGE###################
* FROM ubuntu:20.04 AS production
* LABEL Author="George Leonard (georgelza@gmail.com)"
* WORKDIR /app
* COPY --from=builder /build/main .
* CMD ["/app/main"]
* Next up, same as above, but we use ubuntu:jammy, a nice small little image, resulting in a nice and compact image compared to our first 2 images created.
* FROM golang:1.22 AS builder
* ENV GOPROXY=https://proxy.golang.org,direct
* ENV GOSUMDB=sum.golang.org
* WORKDIR /build
* COPY go.mod .
* COPY go.sum .
* RUN go mod download
* COPY cmd/main.go .
* RUN go build -v -o ./main ./main.go
* ###########START NEW IMAGE###################
* FROM ubuntu:jammy AS production
* LABEL Author="George Leonard (georgelza@gmail.com)"
* WORKDIR /app
* COPY --from=builder /build/main .
* CMD ["/app/main"]
* And lastly… here I show how we can use one layers output as the source for a next layer in Dockerfile.5.
* FROM golang:1.22 AS builder
* ENV GOPROXY=https://proxy.golang.org,direct
* ENV GOSUMDB=sum.golang.org
* WORKDIR /build
* COPY go.mod .
* COPY go.sum .
* RUN go mod download
* COPY cmd/main.go .
* RUN go build -v -o ./main ./main.go
* ###########START NEW IMAGE###################
* FROM builder AS production
* LABEL Author="George Leonard (georgelza@gmail.com)"
* WORKDIR /app
* COPY --from=builder /build/main .
* CMD ["/app/main"]

The entire project can be built and run by executing make build from our Makefile.

This will pull the source images from hub.docker.com and go through the 5 builds.

Now onto the docker compose.yml file.

I demonstrate two ways to get information into the container.

First, we use configs: this creates a name/alias for each of the 2 source files (app.json and kafka.json) in our case located in the conf/ sub directory under the app directory.

configs:

app\_file:

file: ./conf/app.json

kafka\_file:

file: ./conf/kafka.json

This is then mounted into the container at run time, to keep it interesting you will notice I changed the location and name of the destination files. Using this method allows us to change the variables/settings in the source files and simply restart the container for them to become active.

configs:

- source: app\_file

target: /app/conf/avro\_app.json

mode: 444

- source: kafka\_file

target: /app/conf/avro\_kafka.json

mode: 444

mode 444 \* tells the computer to make the file readable (to all users), but to have no files executable or writeable.

The second method shown is by using environment: setting inside the docker compose.yml file.

environment:

PROJECT\_USERNAME: ${PROJECT\_USERNAME}

PROJECT\_PASSWORD: ${PROJECT\_PASSWORD}

These 2 values themselves are pulled in by docker compose from the .env file located in the same directory as the Dockerfile if present.

Not shown is env\_file=<file> as we used previously during the HDFS builds. This will have the same effect as the environment: method above.

NOTE: make sure you add your .env file to your .gitignore file if you are storing any type of credential/secret etc. so that they are not included in your repo sync.

NOTE: docker compose itself can [manage secrets](https://docs.docker.com/compose/use-secrets/)… I know about it, haven’t looked at it yet, on my Todo list.

Re [Makefile](https://opensource.com/article/18/8/what-how-makefile)… they are a handy tool, you will see I use them inside my build directories, you might also noticed, I have one in the root of the repo, think of that one as the project Makefile which itself in turn can be used to call the individual Makefiles in the sub directories… Neat hey.

And that will be it for this Part… Next up are some take aways, lessons…

**My Repo’s**

All the code used during this article will be available on the below GIT repo.

[Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git)

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose. – Part 7

## Takeaways.

(See: [Part 6](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-5-8463f7b25545))

(20 August 2024)

This part is really really going to be very short.

* We started with a base OS image, that we extended with common tooling,
* We then added/installed an application server stack, reusing our previous image built.
* This was then used to build a working HDFS cluster, not the easiest thing at first.
* And lastly, we quickly covered docker multistage builds.
* With docker… probably the most important thing for me… try and build things that can be repeated,
* Do not copy configuration files into the build at built time, you don’t want the cast in stone, for this I showed multiple ways to get information into a running container,
* Additionally shown was the usage of volumes (see the build-hadoop-openjdk11-hdfs located docker compose.yml file). This ability allows you to copy files into a container and see files created by a container on the mount point, in our case the actual files created/written on the datanode’s for example.
* if you look at my Makefile you will notice allot of variables are used.

**My Repo’s**

All the code used during this article will be available on the below GIT repo.

[Building Docker Images](ttps://github.com/georgelza/dockerimagebuilding.git)

And that’s it for now… Thank you for sticking with me through this exploration. All it did as previous was to create a small little to do list, that itself is growing as I am typing this, of things/subjects I’d like to explore more and blog . Till next time.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

Some more References:

See the master README.md in the root of the repo for some useful references/YouTube videos.