**Dockerfile**

* [Dockerfile Benefits](http://tutorials.jenkov.com/docker/dockerfile.html#dockerfile-benefits)
* [Dockerfile Structure](http://tutorials.jenkov.com/docker/dockerfile.html#dockerfile-structure)
* [Docker Base Image](http://tutorials.jenkov.com/docker/dockerfile.html#docker-base-image)
* [MAINTAINER](http://tutorials.jenkov.com/docker/dockerfile.html#maintainer)
* [FROM](http://tutorials.jenkov.com/docker/dockerfile.html#from)
* [CMD](http://tutorials.jenkov.com/docker/dockerfile.html#cmd)
* [COPY](http://tutorials.jenkov.com/docker/dockerfile.html#copy)
* [ADD](http://tutorials.jenkov.com/docker/dockerfile.html#add)
* [ENV](http://tutorials.jenkov.com/docker/dockerfile.html#env)
* [RUN](http://tutorials.jenkov.com/docker/dockerfile.html#run)
* [ARG](http://tutorials.jenkov.com/docker/dockerfile.html#arg)
* [WORKDIR](http://tutorials.jenkov.com/docker/dockerfile.html#workdir)
* [EXPOSE](http://tutorials.jenkov.com/docker/dockerfile.html#expose)
* [VOLUME](http://tutorials.jenkov.com/docker/dockerfile.html#volume)
* [ENTRYPOINT](http://tutorials.jenkov.com/docker/dockerfile.html#entrypoint)
* [HEALTHCHECK](http://tutorials.jenkov.com/docker/dockerfile.html#healthcheck)
  + [Health Check Interval](http://tutorials.jenkov.com/docker/dockerfile.html#health-check-interval)
  + [Health Check Start Period](http://tutorials.jenkov.com/docker/dockerfile.html#health-check-start-period)
  + [Health Check Timeout](http://tutorials.jenkov.com/docker/dockerfile.html#health-check-timeout)
  + [Health Check Retries](http://tutorials.jenkov.com/docker/dockerfile.html#health-check-retries)

A *Docker* *Dockerfile* contains a set of instructions for how to build a Docker image. The Docker build command executes the Dockerfile and builds a Docker image from it. This Dockerfile tutorial explains how to write a Dockerfile and build it into a Docker image.

**Dockerfile Benefits**

A Dockerfile captures the recipe for building a Docker image in writing. A Docker image typically consists of:

* A base Docker image on top of which to build your own Docker image.
* A set of tools and applications to be installed in the Docker image.
* A set of files to be copied into the Docker image (e.g configuration files).
* Possibly a network (TCP / UDP) port (or more) to be opened for traffic in the firewall.
* etc.

First of all, capturing all this in writing in a Dockerfile means that You don't have to remember how to setup a new installation of your application, with OS requirements, applications to install, files to copy, network ports to open up for etc. It's all described in the Dockerfile.

Additionally, letting Docker build your Docker image from a Dockerfile means that you don't have to do this manual, tedious and error prone job. Docker takes of it for you, so you can do something else in the meantime. This is both easier, faster, and less error prone.

Third, it's really easy to share a Dockerfile with someone else, and have them be able to build the Docker image.

Fourth, a Dockerfile can easily be stored in a version control system like Git, so you can track changes to the Dockerfile (the server and application configuration) over time. Version control systems typically also makes it easier for multiple people to collaborate - e.g. on a Dockerfile, as well as share the Dockerfile later on.

**Dockerfile Structure**

A Dockerfile consists of a set of instructions. Each instruction consists of a command followed by arguments to that command, similar to command line executables. Here is a simple example of a Dockerfile:

# The base image

FROM ubuntu:latest

# More instructions here that install software and copy files into the image.

COPY /myapp/target/myapp.jar /myapp/myapp.jar

# The command executed when running a Docker container based on this image.

CMD echo Starting Docker Container

**Docker Base Image**

A Docker image consists of *layers*. Each layer adds something to the final Docker image. Each layer is actually a separate Docker image. Thus, your Docker image consists of one or more underlying Docker images, on top of which you add your own layers.

When you specify your own Docker image via a Dockerfile you typically start with a *Docker base image*. This is another Docker image on top of which you want your own Docker image to be built. The Docker base image you are using may itself consist of multiple layers, and can itself be based on another base image etc., until you get down to the most basic Docker image you can create - a raw Linux container image with no special settings applied.

You specify the base image of a Docker image in your Dockerfile using the FROM command, as explained in the following section.

**MAINTAINER**

The Dockerfile MAINTAINER command is simply used to tell who is maintaining this Dockerfile. Here is an example:

MAINTAINER Joe Blocks <joe@blocks.com>

The MAINTAINER instruction is not often used though, since that kind of information is also often available in GIT repositories and elsewhere.

**FROM**

The Dockerfile FROM command specifies the base image of your Docker images. If you want to start with a bare Linux image, you can use this FROM command:

# The base image

FROM ubuntu:latest

**CMD**

The CMD command specifies the command line command to execute when a Docker container is started up which is based on the Docker image built from this Dockerfile. Here are a few Dockerfile CMD examples:

CMD echo Docker container started.

This example just prints the text Docker container started when the Docker container is started.

The next CMD example runs a Java application:

CMD java -cp /myapp/myapp.jar com.jenkov.myapp.MainClass arg1 arg2 arg3

**COPY**

The Dockerfile COPY command copies one or more files from the Docker host (the computer building the Docker image from the Dockerfile) into the Docker image. The COPY command can copy both a file or a directory from the Docker host to the Docker image. Here is a Dockerfile COPY example:

COPY /myapp/target/myapp.jar /myapp/myapp.jar

This example copies a single file from the Docker host at /myapp/target/myapp.jar to the Docker image at /myapp/myapp.jar . The first argument is the Docker host path (where to copy from) and the second argument is the Docker image path (where to copy to).

You can also copy a directory from the Docker host to the Docker image. Here is an example:

COPY /myapp/config/prod /myapp/config

This example copies the directory /myapp/config/prod from the Docker host to the /myapp/config directory in the Docker image.

You can als copy multiple files into a single directory in the Docker image using the COPY command. Here is an example:

COPY /myapp/config/prod/conf1.cfg /myapp/config/prod/conf2.cfg /myapp/config/

This example copies the two files /myapp/config/prod/conf1.cfg and /myapp/conig/prod/conf2.cfg into the Docker image directory /myapp/config/ . Notice how the destination directory has to end with a / (slash) for this to work.

**ADD**

The Dockerfile ADD instruction works in the same way as the COPY instruction with a few minor differences:

* The ADD instruction can copy and extract TAR files from the Docker host to the Docker image.
* The ADD instruction can download files via HTTP and copy them into the Docker image.

Here are a few Dockerfile ADD examples:

ADD myapp.tar /myapp/

This example will extract the given TAR file from the Docker host into the /myapp/ directory inside the Docker image.

Here is another example:

ADD http://jenkov.com/myapp.jar /myapp/

**ENV**

The Dockerfile ENV command can set an environment variable inside the Docker image. This environment variable is available for the application that is started inside the Docker image with the CMD command. Here is an example:

ENV MY\_VAR 123

This example sets the environment variable MY\_VAR to the value 123 .

**RUN**

The Dockerfile RUN command can execute command line executables within the Docker image. The RUN command is executed during build time of the Docker image, so RUN commands are only executed once. The RUN command can be used to install applications within the Docker image, or extract files, or other command line activities which are necessary to run once to prepare the Docker image for execution.

RUN apt-get install some-needed-app

**ARG**

The Dockerfile ARG instruction lets you define an argument which can be passed to Docker when you build the Docker image from the Dockerfile. Here is an example:

ARG tcpPort

When you run the Docker command to build the Dockerfile containing the above ARG instruction, you can pass an argument to the tcpPort argument like this:

docker build --build-arg tcpPort=8080 .

Notice the --build-arg followed by the tcpPort=8080 . This part sets the tcpPort argument value to 8080.

You can define multiple build arguments using multiple ARG instructions. Here is an example:

ARG tcpPort

ARG useTls

When building the Docker image you must provide values for all of the build arguments. You do so by repeating the --build-arg sections for each argument you want to set. Here is an example:

docker build --build-arg tcpPort=8080 --build-arg useTls=true .

You can set a default value for an ARG so that it becomes optional to provide a value for it when building the Docker image. If the argument is not given a value, it will be given its default value. Here is an example:

ARG tcpPort=8080

ARG useTls=true

If neither the tcpPort nor the useTls argument is set when building the Docker image for the Dockerfile containing the above ARG instructions, their argument values will be set to 8080 and true .

Arguments declared by ARG are typically referenced elsewhere in your Dockerfile. You reference an ARG argument like this:

ARG tcpPort=8080

ARG useTls=true

CMD start-my-server.sh -port ${tcpPort} -tls ${useTls}

Notice the two references ${tcpPort} and ${useTls}. These refer to the declared ARG arguments named tcpPort and useTls .

docker build --build-arg tcpPort=8080

**WORKDIR**

The WORKDIR instruction specifies what the working directory should be inside the Docker image. The working directory will be in effect for all commands following the WORKDIR instruction. Here is an example:

WORKDIR /java/jdk/bin

**EXPOSE**

The Dockerfile EXPOSE instruction opens up network ports in the Docker container to the outside world. For instance, if your Docker container runs a web server, that web server will probably need port 80 open for any client to be able to connect to it. Here is an example of opening a network port using the EXPOSE command:

EXPOSE 8080

You can also set which protocol is allowed to communicate on the opened port. For instance, UDP or TCP. Here is an example of setting the allowed protocol also:

EXPOSE 8080/tcp 9999/udp

If no protocol is set (after the / ) , then the protocol is assumed to be TCP.

**VOLUME**

The Dockerfile VOLUME instruction creates a directory inside the Docker image which you can later mount a volume (directory) to from the Docker host. In other words, you can create a directory inside the docker image, e.g. called /data which can later be mounted to a directory, e.g. called /container-data/container1 in the Docker host. The mounting is done when the container is started up. Here is an example of defining a volume (mountable directory) in a Dockerfile using the VOLUME instruction:

VOLUME /data

**ENTRYPOINT**

The Dockerfile ENTRYPOINT instruction provides an *entrypoint* for Docker containers started from this Docker image. An *entrypoint* is an application or command that is executed when the Docker container is started up. In that way, ENTRYPOINT works similarly to CMD, with the difference being that using ENTRYPOINT the Docker container is shut down when the application executed by ENTRYPOINT finishes. Thus, ENTRYPOINT kind of makes your Docker image an executable command itself, which can be started up and which shut down automatically when finished. Here is a Dockerfile ENTRYPOINT example:

ENTRYPOINT java -cp /apps/myapp/myapp.jar com.jenkov.myapp.Main

This example will execute the Java application main class com.jenkov.myapp.Main when the Docker container is started up, and when the application shuts down, so does the Docker container.

**HEALTHCHECK**

The Dockerfile HEALTHCHECK instruction can execute a *health check* command line command at regular intervals, to monitor the health of the application running inside the Docker container. If the command line command returns a value of 0 when exiting, Docker considers the application and container to be healthy. If the command line command returns a value of 1, Docker consider the application and container to be unhealthy. Here is a Dockerfile HEALTHCHECK example:

HEALTHCHECK java -cp /apps/myapp/healthcheck.jar com.jenkov.myapp.HealthCheck https://localhost/healthcheck

This example configures the Java application com.jenkov.myapp.HealthCheck as the healthcheck command. You can use whatever healthcheck command that makes sense to you here.

**Health Check Interval**

By default Docker executes the HEALTHCHECK command every 30 seconds. However, you can set a custom health check interval if you prefer a different interval length than the default 30 seconds. You specify the health check interval using the --interval argument to the HEALTHCHECK instruction. Here is an example that sets the HEALTHCHECK interval to 60 seconds instead:

HEALTHCHECK **--interval=60s** java -cp /apps/myapp/healthcheck.jar com.jenkov.myapp.HealthCheck https://localhost/healthcheck

**Health Check Start Period**

By default Docker will start checking the health of the Docker container immediately. However, sometimes an application might take some time to start up, so it may not make sense to health check it until after a certain time period has elapsed. This gives the application the chance to startup before Docker starts health checking it. You can set the health check start period using the --start-period argument to the HEALTHCHECK instruction. Here is an example setting the health check start period to 5 minutes, giving the container and application 300 seconds (5 minutes) to start up before Docker starts checking its health:

HEALTHCHECK **--start-period=300s** java -cp /apps/myapp/healthcheck.jar com.jenkov.myapp.HealthCheck https://localhost/healthcheck

**Health Check Timeout**

It is possible for a health check to time out. If the HEALTCHECK command takes more than a given time limit to finish, Docker will consider the health check timed out. You can set the timeout limit using the --timeout argument to the HEALTHCHECK command. Here is an example of setting the health check timeout time limit to 5 seconds:

HEALTHCHECK **--timeout=5s** java -cp /apps/myapp/healthcheck.jar com.jenkov.myapp.HealthCheck https://localhost/healthcheck

Note: If the health check times out, Docker considers the container to be unhealthy too.

**Health Check Retries**

If the HEALTHCHECK fails, either because the HEALTCHECK command returns 1, or if it times out, Docker will retry the HEALTHCHECK command 3 times to see if the Docker container returns to a healthy state, before considering the Docker container unhealthy. You can override the default 3 retries using the --retries argument to the HEALTHCHECK instruction. Here is an example of setting the number of health check retries to 5:

HEALTHCHECK **--retries=5** java -cp /apps/myapp/healthcheck.jar com.jenkov.myapp.HealthCheck https://localhost/healthcheck