docker

WORKSHOP

Barry Evans: using some slides from Stefan Koospal & Mohsen Haghaieghshenasfard

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- **Dockerfile Instructions**
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

Dockerfile Instructions

ADD

Copies files from the build context or remote URLs into the image. If an archive file is added from a local path, it will automatically be unpacked. As the range of functionality covered by ADD is quite large, it’s generally best to prefer the simpler COPY command for copying files and directories in the build context and RUN instructions with curl or wget to download remote resources.

COPY

Used to copy files from the build context into the image. It has two forms, COPY src dest_ and COPY ["src", "dest"], both of which copy the file or directory at src in the build context to dest inside the container. The JSON array format is required if the paths have spaces in them. Wildcards can be used to specify multiple files or directories. Note that you cannot specify src paths outside the build context (e.g., ../another_dir/myfile will not work).

Dockerfile Instructions (cont.)

CMD

Runs the given instruction when the container is started. If an ENTRYPOINT has been defined, the instruction will be interpreted as an argument to the ENTRY POINT (in this case, make sure you use the exec format). The CMD instruction is overridden by any arguments to docker run after the image name. Only the last CMD instruction will have an effect, and any previous CMD instructions will be overridden (including those in base images).

ENTRYPOINT

Sets an executable (and default arguments) to be run when the container starts. Any CMD instructions or arguments to docker run after the image name will be passed as parameters to the executable. ENTRYPOINT instructions are often used to provide “starter” scripts that initialize variables and services before interpreting any given arguments.

Dockerfile Instructions (cont.)

CMD and ENTRYPOINT commands allow us to **set the default command** to run in a container.

Defining a default command

When people run our container, we want to greet them with a nice hello message, and using a custom font. For that, we will execute:

figlet -f script hello

- -f script tells figlet to use a fancy font.
- hello is the message that we want it to display.

Dockerfile Instructions (cont.)

Adding CMD to our Dockerfile

```
FROM ubuntu
RUN apt-get update && apt-get install -y figlet
CMD figlet -f script hello
```

- CMD defines a default command to run when none is given.
- It can appear at any point in the file.
- Each CMD will replace and override the previous one.
- As a result, while you can have multiple CMD lines, it is useless.

Dockerfile Instructions (cont.)

Build and test our image

Let's build it:

~/figlet# **docker build -t figlet .**

And run it:

~/figlet# **docker run -t figlet**

```
 _          _   _
| |        | | | |
| |    _   | | | |  __
|/ \   |/  |/  |/  /  \_
|   |_/|__/|__/|__/\__/
```

Dockerfile Instructions (cont.)

Overriding CMD

If we want to get a shell into our container (instead of running figlet), we just have to specify a different program to run:

~/figlet# **docker run -h figlet -it figlet "/bin/bash"**

root@figlet:/#

- We specified bash.
- It replaced the value of CMD.

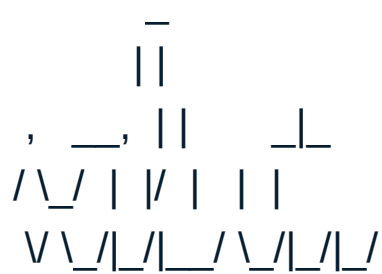
Dockerfile Instructions (cont.)

Using ENTRYPOINT

We want to be able to specify a different message on the command line, while retaining figlet and some default parameters.

In other words, we would like to be able to do this:

docker run figlet salut



Dockerfile Instructions (cont.)

Using CMD and ENTRYPOINT together

What if we want to define a default URL for our container?

Then we will use ENTRYPOINT and CMD together.

- ENTRYPOINT will define the base command for our container.
- CMD will define the default parameter(s) for this command.

Dockerfile Instructions (cont.)

Using the exec format (Json)

```
FROM ubuntu
RUN apt-get update && apt-get install -y figlet
ENTRYPOINT ["figlet","-f","script"]
CMD hello
```

- ENTRYPOINT defines a base command (and its parameters) for the container.
- If we don't specify extra command-line arguments when starting the container, the value of CMD is appended.
- Otherwise, our extra command-line arguments are used instead of CMD.

Dockerfile Instructions (cont.)

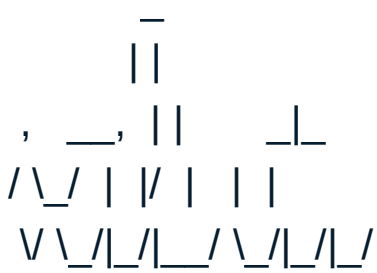
Build and test our image

Let's build it:

~/figlet# **docker build -t figlet .**

And run it:

docker run figlet salut



Dockerfile Instructions (cont.)

Overriding ENTRYPOINT

What if we want to run a shell in our container?

We cannot just do docker run figlet bash because that would just tell figlet to display the word "bash."

We use the --entrypoint parameter:

```
# docker run -it -h figlet --entrypoint "/bin/bash"  figlet
```

Dockerfile Instructions (cont.)

- CMD defines a default command to run when none is given.
- It can appear at any point in the file.
- Each CMD will replace and override the previous one.
- As a result, while you can have multiple CMD lines, it is useless.

Dockerfile Instructions (cont.)

ENV

- Sets environment variables inside the image. These can be referred to in subsequent instructions. For example:

```
FROM ubuntu
ENV MYVERSION 2.7
RUN apt-get update && apt-get install -y figlet
RUN apt-get install -y python${MYVERSION}-minimal
```

- The variables will also be available inside the image.

```
# docker run -it -h figlet --entrypoint "/bin/bash"  figlet
root@figlet:/# echo $MYVERSION
1.1
```

Dockerfile Instructions (cont.)

EXPOSE

Indicates to Docker that the container will have a process listening on the given port or ports. This information is used by Docker when linking containers (see “Linking Containers”) or publishing ports by supplying the -P argument to docker run; by itself the EXPOSE instruction will not affect networking.

FROM

Sets the base image for the Dockerfile; subsequent instructions build on top of this image. The base image is specified as IMAGE:TAG (e.g., debian:wheezy). If the tag is omitted, it is assumed to be latest, but I strongly recommend you always set the tag to a specific version to avoid surprises. Must be the first instruction in a Dockerfile.

Dockerfile Instructions (cont.)

MAINTAINER

Sets the “Author” metadata on the image to the given string. You can retrieve this with docker inspect -f {{.Author}} IMAGE. Normally used to set the name and contact details of the maintainer of the image.

ONBUILD

Specifies an instruction to be executed later, when the image is used as the base layer to another image. This can be useful for processing data that will be added in a child image (e.g., the instruction may copy in code from a chosen directory and run a build script on the data).

RUN

Runs the given instruction inside the container and commits the result.

Dockerfile Instructions (cont.)

USER

Sets the user (by name or UID) to use in any subsequent RUN, CMD, or ENTRYPOINT instructions. Note that UIDs are the same between the host and container, but usernames may be assigned to different UIDs, which can make things tricky when setting permissions.

VOLUME

Declares the specified file or directory to be a volume. If the file or directory already exists in the image, it will copied into the volume when the container is started. If multiple arguments are given, they are interpreted as multiple volume statements. You cannot specify the host directory for a volume inside a Dockerfile for portability and security reasons. For more information, see “Managing Data with Volumes and Data Containers”.

Dockerfile Instructions (cont.)

WORKDIR

Sets the working directory for any subsequent RUN, CMD, ENTRYPOINT, ADD, or COPY instructions. Can be used multiple times. Relative paths may be used and are resolved relative to the previous WORKDIR.

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- **Connecting Containers to the World**
- Common Docker Commands
- Managing Containers
- Practical Section

Connecting Containers to the World

Say you're running a web server inside a container. How do you provide the outside world with access? The answer is to "publish" ports with the -p or -P commands. This command forwards ports on the host to the container. For example:

docker pull nginx

…

docker run -h www --name www -d -p 8000:80 nginx

…

docker ps

```
CONTAINER ID   IMAGE   COMMAND                  CREATED          STATUS          PORTS                    NAMES
4fd7edf272bb   nginx   "nginx -g 'daemon of…"   About a minute ago   Up About a minute   0.0.0.0:8000->80/tcp   www
```

curl localhost:8000

```
<!DOCTYPE html>
<h1>Welcome to nginx!</h1>
```

Connecting Containers to the World (cont.)

The -p 8000:80 argument has told Docker to forward port 8000 on the host to port 80 in the container. Alternatively, the -P argument can be used to tell Docker to automatically select a free port to forward to on the host. For example:

ID=$(docker run -h www --name www -d -P nginx) # **docker port $ID 80**
0.0.0.0:32768

curl localhost:32768
<!DOCTYPE html>
…
<h1>Welcome to nginx!</h1>

The primary advantage of the -P command is that you are no longer responsible for keeping track of allocated ports, which becomes important if you have several containers publishing ports. In these cases you can use the docker port command to discover the port allocated by Docker.

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- **Common Docker Commands**
- Managing Containers
- Practical Section

The run Command

It is the most complex command and supports a large list of potential arguments. The arguments allow users to configure how the image is run, override Dockerfile settings, configure networking, and set privileges and resources for the container.

-a, --attach

Attaches the given stream (STDOUT, etc.) to the terminal. If unspecified, both STDOUT and STDERR are attached. If unspecified and the container is started in interactive mode (-i), STDIN is also attached.

Incompatible with -d

The run Command (cont.)

-d, --detach

Runs the container in “detached” mode. The command will run the container in the background and return the container ID.

-i, --interactive

Keeps STDIN open (even when it’s not attached). Generally used with -t to start an interactive container session. For example:

docker run -h ubuntu -i -t ubuntu "/bin/bash"

root@ubuntu:/# echo "hello world"

hello world

The run Command (cont.)

--restart

Configures when Docker will attempt to restart an exited container. The argument no will never attempt to restart a container, and always will always try to restart, regardless of exit status. The on-failure argument will attempt to restart containers that exit with a nonzero status and can take an optional argument specifying the number of times to attempt to restart before giving up (if not specified, it will retry forever). For example, docker run --restart onfailure:
10 postgres will launch the postgres container and attempt to restart it
10 times if it exits with a nonzero code.

The run Command (cont.)

--rm

Automatically removes the container when it exits. Cannot be used with -d.

-t, --tty

Allocates a pseudo-TTY. Normally used with -i to start an interactive container.

The following options allow setting of container names and variables:

The run Command (cont.)

-e, --env

Sets environment variables inside the container. For example:

docker run -h ubuntu -e var1="hello" -t ubuntu env

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=ubuntu

TERM=xterm

var1=hello

HOME=/root

Also note the --env-file option for passing variables in via a file.

The run Command (cont.)

-h, --hostname

Sets the container's Unix host name to NAME. For example:

$ **docker run -h "myhost“ ubuntu hostname**
myhost

--name NAME

Assigns the name NAME to the container. The name can then be used to address the container in other Docker commands.

The run Command (cont.)

-v, --volume

There are two forms of the argument to set up a volume (a file or directory within a container that is part of the native host filesystem, not the container's union file system). The first form only specifies the directory within the container and will bind to a host directory of Docker's choosing. The second form specifies the host directory to bind to.

--volumes-from

Mounts volumes from the specified container. Often used in association with data containers

The run Command (cont.)

--expose

Equivalent of Dockerfile EXPOSE instruction. Identifies the port or port range as being used in the container but does not open the port. Only really makes sense in association with -P and when linking containers.

--link

Sets up a private network interface to the specified container.

-p, --publish

“Publishes” a port on the container, making it accessible from the host. If the host port is not defined, a random high-numbered port will chosen, which can be discovered by using the **docker port** command. The host interface on which to expose the port may also be specified.

The run Command (cont.)

--expose

Equivalent of Dockerfile EXPOSE instruction. Identifies the port or port range as being used in the container but does not open the port. Only really makes sense in association with -P and when linking containers.

--link

Sets up a private network interface to the specified container.

-p, --publish

“Publishes” a port on the container, making it accessible from the host. If the host port is not defined, a random high-numbered port will chosen, which can be discovered by using the **docker port** command. The host interface on which to expose the port may also be specified.

The run Command (cont.)

-P, --publish-all

Publish all exposed ports on the container to the host. A random high-numbered port will be chosen for each exposed port. The docker port command can be used to see the mapping.

The following options directly override Dockerfile settings:

--entrypoint

Sets the entrypoint for the container to the given argument, overriding any ENTRY POINT instruction in the Dockerfile.

The run Command (cont.)

-u, --user

Sets the user that commands are run under. May be specified as a username or UID. Overrides USER instruction in Dockerfile.

-w, --workdir

Sets the working directory in the container to the provided path. Overrides any value in the Dockerfile.

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- **Managing Containers**
- Practical Section

Managing Containers (cont.)

docker attach [OPTIONS] CONTAINER

The attach command allows the user to view or interact with the main process inside the container. For example:

ID=$(docker run -d ubuntu sh -c "while true; do echo Tick;sleep 1;done")

docker attach $ID

tick

tick

...

Note that using CTRL-C to quit will end the process and cause the container to exit.

Managing Containers (cont.)

docker create

Creates a container from an image but does not start it. Takes most of the same arguments as docker run. To start the container, use docker start.

docker cp

Copies files and directories between a container and the host.

docker exec

Runs a command inside a container. Can be used to perform maintenance tasks or as a replacement for ssh to log in to a container.

Managing Containers (cont.)

docker exec

Runs a command inside a container. Can be used to perform maintenance tasks or as a replacement for ssh to log in to a container. For example:

ID=$(docker run -d ubuntu sh -c "while true; do sleep 1;done")

docker exec $ID echo "Hello“

Hello

docker exec $ID /bin/bash

root@e299debda797:/# ls

bin dev home lib64 mnt proc run srv tmp var

boot etc lib media opt root sbin sys usr

root@e299debda797:/# exit

Managing Containers (cont.)

docker kill

Sends a signal to the main process (PID 1) in a container. By default, sends a SIGKILL, which will cause the container to exit immediately. Alternatively, the signal can be specified with the -s argument. The container ID is returned.

For example:

ID=$(docker run -d ubuntu bash -c "trap 'echo got-sig' 2;while true;do sleep 1;done")

docker kill -s 2 $ID

ca8bf50b4b303fa72c9494503e832a977c6a7121ba08cb2ffe2f144fcaed4ba4

docker logs $ID

got-sig

docker kill $ID

Managing Containers (cont.)

docker pause

Suspends all processes inside the given container. The processes do not receive any signal that they are being suspended and consequently cannot shut down or clean up. The processes can be restarted with docker unpause. docker pause uses the Linux cgroups freezer functionality internally. This command contrasts with docker stop, which stops the processes and sends signals observable by the processes.

docker restart

Restarts one or more containers. Roughly equivalent to calling docker stop followed by docker start on the containers. Takes an optional argument -t that specifies the amount of time to wait for the container to shut down before it is killed with a SIGTERM.

Managing Containers (cont.)

docker rm

Removes one or more containers. Returns the names or IDs of successfully deleted containers. By default, docker rm will not remove any volumes. The -f argument can be used to remove running containers, and the -v argument will remove volumes created by the container (as long as they aren't bind mounted or in use by another container).

For example, to delete all stopped containers:

docker rm -v $(docker ps -aq)

121bedfbc193

fd6c883a46e1

f1e4edf9055d

Managing Containers (cont.)

docker start

Starts a stopped container (or containers). Can be used to restart a container that has exited or to start a container that has been created with docker create but never launched.

docker stop

Stops (but does not remove) one or more containers. After calling docker stop on a container, it will transition to the “exited” state. Takes an optional argument -t which specifies the amount of time to wait for the container to shutdown before it is killed with a SIGTERM.

docker unpause

Restarts a container previously paused with docker pause.

Docker Info

docker info

Prints various information on the Docker system and host.

docker help

Prints usage and help information for the given subcommand. Identical to running a command with the --help flag.

docker version

Prints Docker version information for client and server as well as the version of Go used in compilation.

Docker Info (cont.)

docker info

Prints various information on the Docker system and host.

docker help

Prints usage and help information for the given subcommand. Identical to running a command with the --help flag.

docker version

Prints Docker version information for client and server as well as the version of Go used in compilation.

Container Info

docker diff

Shows changes made to the containers filesystem compared to the image it was launched from. For example:

ID=$(docker run -d ubuntu touch /NEWFILE)

docker diff $ID

A /NEWFILE

docker events

Prints real-time events from the daemon. Use CTRL-C to quit.

Container Info (cont.)

docker inspect

Provides detailed information on given containers or images. The information includes most configuration information and covers network settings and volume mappings. The command can take one argument, -f, which is used to supply a Go template that can be used to format and filter the output.

docker logs

Outputs the “logs” for a container. This is simply everything that has been written to STDERR or STDOUT inside the container.

Container Info (cont.)

docker port

Lists the exposed port mappings for the given container. Can optionally be given the internal container port and protocol to look up. Often used after docker run -P <image> to discover the assigned ports.

For example:

ID=$(docker run -h www --name www -d -P nginx)

docker port $ID

80/tcp -> 0.0.0.0:32769

docker port $ID 80

0.0.0.0:32769

docker port $ID 80/tcp

0.0.0.0:32769

Container Info (cont.)

docker ps

Provides high-level information on current containers, such as the name, ID, and status. Takes a lot of different arguments, notably -a for getting all containers, not just running ones. Also note the -q argument, which only returns the container IDs and is very useful as input to other commands such as docker rm.

Container Info (cont.)

docker top

Provides information on the running processes inside a given container. In effect, this command runs the UNIX ps utility on the host and filters for processes in the given container. For example:

 # docker top $ID

```
UID         PID         PPID        C           STIME       TTY         TIME        CMD
root        5091        5077        0           14:31       ?           00:00:00    nginx: master process nginx -g daemon off;
systemd+    5118        5091        0           14:31       ?           00:00:00    nginx: worker process
```

 # ps -aux |grep 5091

```
root      5091  0.0  0.0  32552  5172 ?        Ss   14:31   0:00 nginx: master process nginx -g daemon off;
```

Docker Images

docker build

Builds an image from a Dockerfile.

docker commit

Creates an image from the specified container. By default, containers are paused prior to commit, but this can be turned off with the --pause=false argument. Takes -a and -m arguments for setting metadata. For example:

ID=$(docker run -d ubuntu touch /NEWFILE)

docker commit -a "Stefan Koospal" -m "Comment" $ID newfile:test

sha256:c9c7833762d4bb8fbaa29c8c82c60230029311c4784b814597c9eb0b822eeb1a

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
newfile	test	c9c7833762d4	About a minute ago	112MB

Docker Images (cont.)

docker history

Outputs information on each of the layers in an image.

docker images

Provides a list of local images, including information such as repository name, tag name, and size. Takes several arguments; in particular, note -q, which only returns the image IDs and is useful as input to other commands such as docker rmi.

docker images

test/figlet	latest	571b6e1056e2	2 hours ago	153MB
test/openjdk-jshell	latest	fbe3a756f5c4	6 days ago	910MB
test/openjdk	latest	fbe3a756f5c4	6 days ago	910MB
test/java	latest	fbe3a756f5c4	6 days ago	910MB
test/cowsay-dockerfile	latest	ac747299997b	6 days ago	197MB

Docker Images (cont.)

docker import

Creates an image from an archive file containing a filesystem, such as that created by docker export. The archive may be identified by a file path or URL or streamed through STDIN (by using the - flag).

docker load

Loads a repository from a tar archive passed via STDIN. The repository may contain several images and tags.

Docker Images (cont.)

docker rmi

Deletes the given image or images. Images are specified by ID or repository and tag name. If a repository name is supplied but no tag name, the tag is assumed to be latest. To delete images that exist in multiple repositories, specify that image by ID and use the -f argument. You will need to run this once per repository.

docker save

Saves the named images or repositories to a tar archive, which is streamed to STDOUT (use -o to write to a file). Images can be specified by ID or as repository:tag. If only a repository name is given, all images in that repository will be saved to the archive, not just the latest tag.

Docker Images (cont.)

docker rmi

Deletes the given image or images. Images are specified by ID or repository and tag name. If a repository name is supplied but no tag name, the tag is assumed to be latest. To delete images that exist in multiple repositories, specify that image by ID and use the -f argument. You will need to run this once per repository.

docker save

Saves the named images or repositories to a tar archive, which is streamed to STDOUT (use -o to write to a file). Images can be specified by ID or as repository:tag. If only a repository name is given, all images in that repository will be saved to the archive, not just the latest tag.

Using the Registry

docker login

Register with, or log in to, the given registry server. If no server is specified, it is assumed to be the Docker Hub. The process will interactively ask for details if required, or they can be supplied as arguments.

docker logout

Logs out from a Docker registry. If no server is specified, it is assumed to be the Docker Hub.

Using the Registry (cont.)

docker pull

Downloads the given image from a registry. Use the -a argument to download all images from a repository.

docker push

Pushes an image or repository to the registry. If no tag is given, this will push all images in the repository to the registry, not just the one marked latest.

docker search

Prints a list of public repositories on the Docker Hub matching the search term. Limits results to 25 repositories.

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- **Practical Section**

Set up a docker container that:

1. hosts a web page using nginx
2. displays the DubJUG logo
3. displays a message “Hello DubJUG!”
4. can be accessed by everyone in the class