Introduction to Containers for Science

Hans Fangohr
European XFEL GmbH
University of Southampton

May 18, 2017

Contents

1 Introduction to Containers for Science

This chapter motivates why containers are a powerful tool for science. If you want to jump straight into the examples, jump to the `Hello world` container in the next chapter.

1.1 Computational science challenge: unusual installation requirements

Computational science projects often relay on unusual libraries, or rely on particular versions of a library, or a combination of particular libraries. Often, these tools have only been developed to be used on a particular operating system (say SuSE, Debian, Ubuntu, ...) and depend on the packages one of the versions of these distribution provides.

This makes the code difficult to port (to other machines), to use in the future (if operating system upgrades are required and the updated operating system breaks one of the requirements of the tool, and difficult to re-use by others.

Containers offer a practical way of allowing installation of such tools inside the container image, execution inside the container, and tus portability (as the container can be run on other machines) and also the ability to run the container in the future.

1.2 Computational science challenge: reproducibility

Reproducibility of computational science has become a widely debated topic. We have made good progress with, for example, the use of versioning tools for software (so that it is possible to go back to particular versions of code that may have been used for a particular study). Github and bitbucket, for example, have helped researches to have cloud hosted version control of their research software.

One area of computational science that in my view is not sufficiently addressed at the moment is that of the computational environment: it is not enough to have the source code, we also need to know (i) how to install it and (ii) which libraries the code may have used to compute a particular result.

Containers offer a good way of documenting this. The file that species how the container image should be built (the `Dockerfile` in the coming chapters), documents *exactly* the compute environment in which the software should execute.

1.3 Computational science challenge: execution of code on cluster

High Performance Computing installations (ofter a Linux cluster) are generally maintained by specialist IT staff, and the scientists are users on the system. As such, the scientists generally have no administrator rights, and IT staff install software on the cluster on behalf of the users. As alluded to above, the users may have quite specific requirements on the particular versions of tools they need, leading to tools to manage different versions of software (`module load ...`) and lengthy processes to install software: for example the operating system is generally fixed, and thus occasionally it is not possible (within the staff time available) to port a tool onto an supercomputer for the user.

Containers have a solution to this problem: the user can provide their own container with the installed software, and thus completely (i) elimenate the need to have multiple versions of libraries of the supercomputer and (ii) reduce IT staff time required to install the software. In the following chapters, I use Docker as an application to run containers. In the High Performance Computing community other container applications are likely to be more widely spread, such as for example 'singularity'.

1.4 So what are these containers?

Containers are a slim version of virtual machines: they allow us to use an operating system of our choice, install software of our choice, and execute this software 'inside' the container. This can be done seamlessly, i.e. it is possible to just run one command inside the container and it doesn't take long to start the container (sub seconds). It is possible to execute multiple different containers on the same machine to execute tasks in their respectively preferred compute environment.

The subsequent chapters give more details on the use of containers, creation of (container) images, mounting of file systems, and sharing of ports between containers and host.

For the quick tour here, we assume the reader has installed Docker [1] on their computer.

[1] http://docker.com

1.5 This document

The document will grow over time. It is available:

- sources at http://github.com/fangohr/containers-for-science
- notebooks (one for each chapter at https://github.com/fangohr/containers-for-science/tree/master/notebooks)
- pdf version at https://github.com/fangohr/containers-for-science/tree/master/pdf/containers-for-science.pdf
- html version at https://github.com/fangohr/containers-for-science/tree/master/html/index.html

The pdf and html versions will be updated from the source occasionally.

If you know the Jupyter notebook, you may want to clone the repository, and execute the notebooks in the `notebooks` chapter to walk through the commands interactively.

2 Hello World programme in container

2.1 Example 1: Executing a bash command inside an Ubuntu container

First, we will *pull* the latest Ubuntu (long term support) image from dockerhub.com. If you have carried out this operation before, it will be very quick, otherwise it depends a little on the bandwidth of your Internet connection:

```
In [1]: # NBVAL_IGNORE_OUTPUT
        !docker pull ubuntu:latest
```

```
latest: Pulling from library/ubuntu
Digest: sha256:382452f82a8bbd34443b2c727650af46aced0f94a44463c62a9848133ecb1aa8
Status: Image is up to date for ubuntu:latest
```

Second, we can now execute commands inside that container. For example, the file `/etc/issue` contains information about the version of the Linux distribution. We use the unix command `cat` to conCATenate the contents of that file to the standard output:

```
In [2]: !docker run ubuntu:latest cat /etc/issue
```

```
Ubuntu 16.04.2 LTS \n \l
```

As you may guess, the structure of the docker command is - `docker`: the Docker application on your host system - `run`: the command for the docker application - we want to run an image - `ubuntu:latest` : the image that we want to start - all remaining arguments (here `cat /etc/issue`) are being executed inside the running container

One way of 'writing' a container hello world program would be this:

```
In [3]: !docker run ubuntu:latest bash -c "echo 'Hello World'"
```

```
Hello World
```

3 Creating our own docker container

To create our own container images, we need to write a file that defines what should be inside the environment. The contents are similar to a bash script, although the syntax is determined by the application to run the container. Here we use docker, and the file is (by default) called `Dockerfile`:

```
In [1]: %%file Dockerfile
        FROM ubuntu:16.04

        RUN apt-get update
        RUN apt-get install -y cowsay

        # cowsay installs into /usr/games. Make avaible in PATH:
        RUN ln -s /usr/games/cowsay /usr/local/bin
```

```
Overwriting Dockerfile
```

Build a docker image based on the `Dockerfile` above (in the current directory). Call it `cowimage`.

```
In [2]: #NBVAL_IGNORE_OUTPUT
        !docker build -t cowimage .
```

```
Sending build context to Docker daemon 124.4 kB
Step 1/4 : FROM ubuntu:16.04
 ---> 0ef2e08ed3fa
Step 2/4 : RUN apt-get update
 ---> Using cache
 ---> 64d0cc85e4a4
Step 3/4 : RUN apt-get install -y cowsay
 ---> Using cache
 ---> e4fb0aeee30a
Step 4/4 : RUN ln -s /usr/games/cowsay /usr/local/bin
 ---> Using cache
 ---> 78da9d9c6dbd
Successfully built 78da9d9c6dbd
```

Then we can use this image to run commands, for example cowsay:

```
In [3]: !docker run cowimage cowsay "Hello"

 -------
< Hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

3.1 Exporting an image to a file

Often, having a Dockefile as shown above is sufficient to create a computational environment, in particular if all support libraries and the main code itself are open source, and ideally available online.

In cases where we need to transport a created container image, we can use the following commands:

```
In [4]: !docker save cowimage > cowimage.tar

In [5]: !ls -hl cowimage.tar

-rw-r--r--  1 fangohr  staff   211M 18 May 16:14 cowimage.tar
```

It's worth trying to compress the image:

```
In [6]: !gzip -f cowimage.tar

In [7]: !ls -hl cowimage.tar.gz

-rw-r--r--  1 fangohr  staff    88M 18 May 16:14 cowimage.tar.gz
```

3.2 Importing an image from a file

```
In [8]: !docker load < cowimage.tar.gz

Loaded image: cowimage:latest
```

Let's tidy up and remove the large file we just created:

```
In [9]: !rm -f cowimage.tar.gz
```

4 Working on data on host system (mount directory)

4.1 Non-persistance of file system changes in container

While the container is a nice environment to provide our tailored installation environment and computational tools, it does not retain any changes to the disk system: as soon as the container process is stopped, all changes to the files within the container are forgotten:

```
In [1]: !docker run ubuntu:16.04 echo "Hello" > hello.txt && ls -l hello.txt

-rw-r--r--  1 fangohr  staff  6 18 May 16:15 hello.txt
```

In the example above, we create the file `hello.txt` and store `Hello` in it. The `&&` means that if that command was executed successfully, we will carry on and also execute the next one. The next command (`ls -l hello.txt`) tries to display the file `hello.txt`, and the command succeeds.

At this point, our container sessions stops, and all changes in the container are forgotten. We can confirm this by trying to use `ls -l` exactly as we did before, but find that the file `hello.txt` is not there:

```
In [2]: !docker run ubuntu:16.04 ls -l hello.txt

ls: cannot access 'hello.txt': No such file or directory
```

While it is possible to create special data containers for persistent data, I find it more straight forward to mount a directory from the host system into the container, and to save any output data on this mounted directory.

4.2 Mounting a directory from the host to be available in the container

Let's create a new container to demonstrate this. We will use `cowsay` as the application we install in the container, and we want it to "say" something that comes from an input file (on the host system) and to produce output in the container, which should be saved to the host file system so we can make use of this when the container execution has completed.

```
In [3]: %%file Dockerfile
        FROM ubuntu:16.04

        RUN apt-get update
        RUN apt-get install -y cowsay

        # cowsay installs into /usr/games. Make avaible in PATH:
        RUN ln -s /usr/games/cowsay /usr/local/bin

        # create directory we use for input and output
        RUN mkdir /io

        # change into that direcotry
        WORKDIR /io

Overwriting Dockerfile
```

```
In [4]: !docker build -t cowimage-mount .

Sending build context to Docker daemon 124.4 kB
Step 1/6 : FROM ubuntu:16.04
 ---> 0ef2e08ed3fa
Step 2/6 : RUN apt-get update
```

```
 ---> Using cache
 ---> 64d0cc85e4a4
Step 3/6 : RUN apt-get install -y cowsay
 ---> Using cache
 ---> e4fb0aeee30a
Step 4/6 : RUN ln -s /usr/games/cowsay /usr/local/bin
 ---> Using cache
 ---> 78da9d9c6dbd
Step 5/6 : RUN mkdir /io
 ---> Using cache
 ---> 762384eee8ce
Step 6/6 : WORKDIR /io
 ---> Using cache
 ---> 000d2e436b25
Successfully built 000d2e436b25
```

Let's check that we start in `/io` if we use the container:

```
In [5]: !docker run cowimage-mount pwd
```

```
/io
```

Now we need to mount our local directory to `/io` when we call docker. Let's first create an itput data file:

```
In [6]: %%file cow-input.txt
        Hello from file
```

```
Overwriting cow-input.txt
```

```
In [7]: !docker run -v `pwd`:/io cowimage-mount  cowsay `cat cow-input.txt` > cow-output.txt
```

Let's first check that this has created our output file `cow-output.txt`, and that the file is available on the host system:

```
In [8]: !ls -l cow-output.txt
```

```
-rw-r--r--  1 fangohr  staff  181 18 May 16:15 cow-output.txt
```

The file exists. What does it contain?

```
In [9]: !cat cow-output.txt
```

```
 _________________
< Hello from file >
 -----------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

This is good. We discuss all parts of the command line now.

The `-v` option tells docker to mount a Volume. In particular, the notation `-v A:B` asks to mount the path `A` from the host file system to the path `B` in the container. As we would like to mount our current directory from the host, we use the `pwd` command (which stands for Print Working Directory). By enclosing `pwd` in backticks (‘), the output of the `pwd` command is used to represent `A`.

The path `B` to which we mount the directory is `/io`. Note that we have asked in the Dockerfile that the process in the container should start in this directory.

The actual command to be executed within the container is

```
cowsay `cat cow-input.txt` > cow-output.txt
```

- `cowsay` is the name of the executable.
- `cow-input.txt` is a file on the host file system, which is available within the container in `/io` because we mounted the directory
- with `` `cat cow-input.txt` ``we, we take the content of the `cow-input.txt` file (which is `hello from file` as we have created the file earlier in this notebook), and pass this content to `cowsay`. As a result, the cow prints this in the speech bubble.
- with `> cow-output.txt` we send the (standard) output from the process into the file `cow-output.txt`. As the file doesn’t exist, it is created (in the container) in our directory `/io`) and as our host directory is mounted to `/io` in the container, the file is actually saved in the host directory. And therefore available after the docker container process has completed.

5 Connecting to a port in the container

Another common use case is that we start some kind of long-running process in the container, and talk to it through a port. That process could be a Jupyter Notebook, for example.

5.1 Example http.server

For demonstration purposes, we will use Python’s in-built web server. To run it from the host, we could use

```
In [ ]: # NBVAL_SKIP
        !python -m http.server
```

This program will show the contents of the current file system in a webbrowser interface at port 8000 of this machine. So typically at one or some of these links http://127.0.0.1:8000 or http://localhost:8000 or http://0.0.0.0:8000).

(If you have executed the above cell by pressing SHIFT+RETURN, you need to interrupt the http.server process to get the control back in the notebook. This can be done by choosing from the menu "Kernel" -> "Interrupt".)

We will now create a container and run this server inside the container. We like to use a web-browser on the host machine to inspect the files.

First, we create the Dockerfile:

```
In [2]: %%file Dockerfile
        FROM ubuntu:16.04

        RUN apt-get -y update
        RUN apt-get -y install python3

        CMD python3 -m http.server
```

```
Overwriting Dockerfile
```

The last line starts the `http.server` when the container is run.

```
In [3]: #NBVAL_IGNORE_OUTPUT
        !docker build -t portdemo .

Sending build context to Docker daemon 124.4 kB
Step 1/4 : FROM ubuntu:16.04
 ---> 0ef2e08ed3fa
Step 2/4 : RUN apt-get -y update
 ---> Using cache
 ---> 977edc4badb5
Step 3/4 : RUN apt-get -y install python3
 ---> Using cache
 ---> 1629941ecac2
Step 4/4 : CMD python3 -m http.server
 ---> Using cache
 ---> 1ec81802325c
Successfully built 1ec81802325c
```

We now need to export the port 8000 in the container. We can do this using:

```
In [ ]: #NBVAL_SKIP
        !docker run -p 8123:8000 portdemo
```

The numbers following reflect the internal port (8000) that should be connected to the port (8123) on the host system.

Once the above command is executing, we should be able to browse the file system in the container by going to the link http://localhost:8123 (or http://127.0.0.1:8123 or http://0.0.0.0:8123) on the host system.

We could have mapped port 8000 in the container to port 8000 on the host as well (‘-p 8000:8000).

5.2 Jupyter Notebook

A common application of exposing ports is to install computational or data analysis software inside the container, and to control it from a jupyter notebook running inside the container, but to use a webbrowser from the host system to interact with the notebook. In that case, the above example of exposing the port is in principle the right way to go, too. However, as this is a common usecase, there are a number of prepared Dockerfiles to install the notebook inside the container available at https://github.com/jupyter/docker-stacks, so tat one can start the Dockerfile with `FROM jupyter/...`, (instead of `FROM ubuntu/...`) and in this way build on the Dockerfiles that the Jupyter team provides already.

6 References

A number of useful references I came across while putting together these notebooks.

- Carl Boettiger

```
@article{DBLP:journals/corr/Boettiger14,
  author    = {Carl Boettiger},
  title     = {An introduction to Docker for reproducible research, with examples
               from the {R} environment},
  journal   = {CoRR},
  volume    = {abs/1410.0846},
  year      = {2014},
  url       = {http://arxiv.org/abs/1410.0846},
  timestamp = {Sun, 02 Nov 2014 11:25:59 +0100},
  biburl    = {http://dblp.uni-trier.de/rec/bib/journals/corr/Boettiger14},
  bibsource = {dblp computer science bibliography, http://dblp.org}
}
```

- Titus Brown

  http://angus.readthedocs.io/en/2016/week3/CTB_docker.html
  http://ivory.idyll.org/blog/2015-docker-and-replicating-papers.html

- David Koop

  http://www.cis.umassd.edu/~dkoop/cis602-2016fa/lectures/lecture15.pdf

```
In [ ]:
```