Container

Thomas Juettemann, EMBL-EBI
aqua_faang_course@ebi.ac.uk

May 10-12, 2021

1 Introduction

2 Docker Hub

Docker Hub is the world's largest repository of container images, at the time of this course it hosts 8.3 million repositories. In this section we explore a couple of repositories and run our first container.

2.1 Hello World

Because that is how every IT adventure begins.

Task

1. On Dockerhub find the official **hello-world** image and click on it.
2. In the description page, find and execute the docker run command in the terminal.

Solution

```
https://hub.docker.com/_/hello-world
docker run hello-world
```

2.2 Versions

Dockerhub offers to host several versions of a software. Here we take a quick look at the similarity and differences using PostgreSQL as an example.

Task

1. On Dockerhub find the official **postgres** image and click on it.
2. Which is the latest version?
3. What happens when you click on *latest*?
4. What is the difference between the first two tags in the first line?
5. What is the major difference (hint: *FROM*) between the first tag in line 1 and the first tag in line 2?

Solution

```
https://hub.docker.com/_/postgres
# Latest version is 13.2
# It is a link to the corresponding dockerfile which is hosted on
   GitHub
# They are different tags for the same dockerfile (=same image).
   Each line refers to the same image.
# They use a different Linux version, debian:buster-slim vs alpine
   :3.13.
```

3 BioContainers

BioContainers is a community-driven project that provides the infrastructure and basic guidelines to create, manage and distribute bioinformatics packages (e.g conda) and containers (e.g docker, singularity). Many tools used by computational biologists on a daily basis are readily available. A mayor difference to Docker Hub is that users can request container to be created.

3.1 Exploring

In this section we will search for a specific tool on BioContainers, find the newest version, download the image and run it locally in different ways. The tool of choice is BLAST, and we will use it to create a blast database and then run an alignment.

3.1.1 Finding an image

As a community driven project, BioContainers does not have the notion of an "official image", and therefore no such filter exists. Those images are usually used (=pulled) much more often than others. We exploit that fact to find the "official" NCBI version of BLAST.

Task

1. Got to https://biocontainers.pro/
2. Click on Registry (wait for the search interface to load)
3. Search for **blast**
4. Below the search bar, change the sorting from "Default" to "Pull No" and the sort order from "Asc" to "Desc".
5. The NCBI image is the first or second hit, click on it

Solution

```
# Final destination is:
https://biocontainers.pro/tools/blast
```

3.1.2 Pulling a specfic version

The landing page contains generic information about the tool and different way of utilising it: Docker, Singularity and Conda. The version used in the various commands is not necessarily the most recent one.

Task

1. Click on "Packages and Containers"
2. Find the version **quay.io/biocontainers/blast:2.11.0–pl526he19e7b1_0**
3. Copy the **docker pull** command
4. Open a terminal
5. Paste and execute the cached **docker pull** command
6. Store (**=export**) the image name in an environment variable called *blast_c*
7. List all images

Solution

```
docker pull quay.io/biocontainers/blast:2.11.0--pl526he19e7b1_0
export blast_c='quay.io/biocontainers/blast:2.11.0--
   pl526he19e7b1_0'
docker images #alternative: docker image ls
```

3.1.3 Inspecting an image

Time to play with the image! and see if in the *config section* **WorkingDir** or **Entrypoint** are set, and which **command** is executed when the container starts?

Task

1. *Inspect* the image
2. In the **Config** section, what are the values for
   (a) **WorkingDir**
   (b) **Entrypoint**
3. Which command (**Cmd**) is executed when the container is run?

Solution

```
docker inspect $blast_c
docker inspect -f '{{json .Config.WorkingDir}} {{json .Config.
   Entrypoint}} {{json .Config.Cmd}}' $blast_C
# "" null ["/bin/sh"]
```

3.1.4 From image to container

A docker image is build from the instructions in a docker file. The image is then used as a template (or base), which we can copy and use to run an application. The application needs an isolated environment in which to run: a container. From the BLAST image that we downloaded, we will create a container, and using that container we will create a BLAST database, and then query this database using blastp.
The */train-aquafaang-bioinf/container/data* directory contains a FASTA file with zebrafish proteins in FASTA format. Using the blast container, we will create a BLAST database from the FASTA file. makeblastdb takes 2 arguments:

1. -in path/to/input/fasta
2. -dbtype ()prot or nucl)
3. Documentation: https://ncbi.github.io/magicblast/cook/blastdb.html

Keep in mind that docker can not access any files in our local filesystem, unless it is mounted. Remember that mount takes 3 arguments:

1. type (usually bind)
2. source (absolute path to the directory outside the container)
3. target (absolute path to the directory inside the container)
4. Documentation: https://docs.docker.com/storage/

Task

1. Using the BLAST container (=$blast_c), create a BLAST database with **makeblastdb**

Solution

```
docker run \
--mount type=bind,source=$HOME/train-aquafaang-bioinf/container/
  data,target=/mnt  \
$blast_c \
makeblastdb -in /mnt/zebrafish.1.protein.faa -dbtype prot
```

3.1.5 Life inside a container

To gain a better understanding of a container, we will run the actual alignment inside the container. If you are not sure about the parameters, remember that docker offers help for each sub command (e.g. docker run –help). We want to know which user you are inside a container, and where all executables are located.

Task

1. **run** the BLAST container **interactive**, using a **pseudo tty** and bash as **command**.
2. Inside the container, run **whoami** and **which blastp**
3. List the contents of /mnt

Solution

```
docker run  -it  $blast_c bash
# bash-4.2# whoami
# root
# bash-4.2# which blastp
# /usr/local/bin/blastp
# bash-4.2# ls -l /mnt
# total 0
exit
```

Now let us explore the effect of mounting.

Task

1. Run the same command as before, but also **mount** the **container/data** directory to **/mnt**

Solution

```
docker run -it \
--mount type=bind,source=$HOME/train-aquafaang-bioinf/container/
   data,target=/mnt  \
$blast_c bash
```

Explore a running container

Task

1. Open a second terminal
2. List all running container
3. Copy the container ID of the BLAST container
4. Inspect it
5. What is the **Cmd** in **Config**?
6. What do you see in the **Mounts** section?

Solution

```
docker container ls
docker inspect <container_id>
OR
docker inspect -f '{{json .Config.Cmd}} {{json .Mounts}}' <
   container_id>
```

Execute a command in a running container

The docker exec command runs a new command in a running container. Using exec on the running container, execute a ls -l and a ls -l /mnt. What do you see?

Task

1. Using the container ID, execute a ls -l
2. Using the container ID, execute a ls -l /mnt

Solution

```
docker exec <container_id> ls -l /
docker exec <container_id> ls -l /mnt
```

Alignment

There are 3 ways of running the alignment:

1. Using the container the same way as we did with makeblastdb (swap makeblastdb for blastp). This is the most common solution in a workflow / pipeline or ad-hoc situation.
2. Using docker exec while the first container is still running. This creates a basic version of a BLAST server that can be accessible to other users on the same server.
3. Running blastp inside the first container. Mostly used for debugging or development.

Task

1. Run the alignment in all 3 different ways.
2. Use different names for the out file
3. To make the mounting short, create a environment variable: *export docker_mount="type=bind,source=$HOME/train-aquafaang-bioinf/container/data,target=/mnt"*
4. Exit the running container

Solution

```
export docker_mount="type=bind,source=$HOME/train-aquafaang-bioinf
   /container/data,target=/mnt"
docker run --mount $docker_mount $blast\_c blastp -query /mnt/
   P04156.fasta -db /mnt/zebrafish.1.protein.faa -out /mnt/result-
   1.txt
docker exec <container_id> blastp -query /mnt/P04156.fasta -db /
   mnt/zebrafish.1.protein.faa -out /mnt/result-2.txt
blastp -query /mnt/P04156.fasta -db /mnt/zebrafish.1.protein.faa -
   out /mnt/result-3.txt
exit
```

4 Cleaning up

In the last section we will be looking at how to stop rouge containers and how to get rid of unused containers and images.

4.0.1 Stop or get killed

Containers are meant to be replaceable. If a container becomes inactive, it is easy to replace it with a new one. But how to get rid of a hanging container? Similar to Unix, also Docker has a *kill* command.

Task

1. Download the latest alpine image
2. Run an alpine container using the -i options, nothing else
3. In a second terminal, list all running container
4. Copy the <*container ID*> of the alpine container
5. Stop or Kill the container

Solution

```
docker pull alpine
docker run -i alpine
# New terminal
docker container ls
#copy \textit{<container ID>}
docker stop \textit{<container ID>} # Alternative: docker kill \
   textit{<container ID>}
```

4.0.2 Spring cleaning

Docker takes a conservative approach to cleaning up unused objects (often referred to as "garbage collection"), such as images, containers, volumes, and networks: these objects are generally not removed unless you explicitly ask Docker to do so. This can cause Docker to use extra disk space. For each type of object, Docker provides a prune command. In addition, you can use docker system prune to clean up multiple types of objects at once.

Task

1. Prune images
2. Prune all images which are not associated to at least one container
3. Prune container
4. Prune everything

Solution

```
docker image prune
docker image prune -a
docker container prune
docker system prune
```