Building on and extending FRIBDAQ containers

The fribdaq container images we create and publish at FRIB are complete for use at the FRIB. You may find, however that you need some package or set of packages that are not in the container image nor in the /usr/opt tarballs we publish at sf.net.

This document describes, in a general way, what you need to do to make additional software packages available to our running container images.

We consider two types of packages:

- Packages that must be compiled, built and installed by you (e.g. GEANT4 or versions of ROOT that are not in our /usr/opt tarballs.
- Packages that are available via debian's package installer (apt).

Packages you must compile

These can be built while running the container. Suggested sequence of operations:

- Build a directory that will hold these packages.
- Add that directory to the –bind options on your container start scripts, binding it to where you want to see these packages.
- Use your start script to star the container you want to see that package.
- Build the package into the directory while running the container.

For example; Suppose you want to have a package called apackage that comes in a tarball and uses the ./configure; make; make install scheme to build/install. Suppose we want those packages to appear in the buster image in /mysoftware/apackage.

- In the host system say create a top level directory for this sort of software for all of the containers you will use let's say that's /mysoftware
- Make the subdirectory buster so that you can have parallel directory trees for other container images if necessary later.
- Add –bind /mysoftware/buster:/mysoftware to the singularity command that starts your container
- Run your containers start script.
- Verify that /mysoftware now is a visible directory in the container
- Build the software as you would on your host system specifying that it must install in /mysoftware/apackage

Debian packages installed with apt.

To make distribution and publication easy, we've published docker images and ask you to use **`singularity build`** to create a singularity image from that docker image. The running singularity image is read-only so you can't just **`apt-get install some-package`** to that image. What you can do, is use Docker to create and publish a new image that uses our images as a starting point.

Here's the rough set of steps to do that (all of these steps are performed on your host system, not in a container):

1. If you don't already have an account on hub.docker.com create one. The examples will use the fictitious account `myhub`.
2. Install the docker build ecosystem on your host system. Add the account you will use to generate the images to the `docker` group. For Debian systems, the docker build/run system is in the packages: docker-ce and docker-ce-cli
3. Create an empty directory which you are going to use to create your new container
4. Create a Dockerfile that describes which container image you're starting from and how to modify it.
5. Build the docker container
6. Tag it for inclusion in the docker hub. And push it to the docker hub.
7. Use it to build a singularity image.

We're going to start the detailed part of our example from step 4. As an example, we're going to add the `isympy3` (Python3 symbolic math package) to our container. We're going to base this on the FRIB image fribdaq/frib-buster:v2.5 (Version 2.5 of the fribdaq buster image).

Create a Dockerfile

Docker uses a description file which must be named `Dockerfile` in an image build directory. This file specifies:

- A starting docker image.
- A working directory in the new image
- How to modify the old image to create the new image.

While all of our changes could be made in the Dockerfile, we're going to put the apt-get install commands into another file in the same directory so that this example can be easily extended. There is nothing to stop us from doing everything in the Dockerfile if that's what you want. Here is the contents of the sample Dockerfile:

```
FROM fribdaq/frib-buster:v2.5
WORKDIR /build
COPY . /build
ARG DEBIAN_FRONTEND=noninteractive
RUN chmod a+x /build/packages
RUN /build/packages
```

Example 1 Sample Dockerfile

Here's the contents of the packages file (in the same directory):

```
apt-get install -y isympy3d
```

Example 2 Contents of packages file.

The contents of the packages file is straightforward. The contents of Dockerfile maybe not:

1. The FROM directive establishes a starting point for the image (in this case the FRIB buster image version 2.5
2. WORKDIR establishes a working directory. This directory is created in the new image.
3. COPY copies stuff into a directory in the image. In this case the entire contents of the directory containing the Dockerfile gets copied to /build our working directory. This pulls the packages file into /build in the image.
4. RUN runs commands in the container image. We use this first to make packages runnable and then to run it which actually does the installation of our additional packages.

Build the docker image:

You'll need to chose an image name. We'll choose `testimage` you should choose something different. The command to build the local Docker image is:

```
docker build . -t testimage
```

This command will build an image known to docker on the local system as testimage:latest.

Tag and push the docker image to hub.docker.com

A bit about tags first. Tags are used to identify images and, in some cases where they go. A docker image specification has the form repository/image-name:tag-name. We need to tag our image in a way that makes it known to docker that it belongs in the dockerhub repository our account there owns and give it a tag (normally tags specify versions but they need not). Remember our docker user name is myhub. That gives us a repository named myhub. If you want to be fancy you can create an organization which can have a repositor as well (fribdaq is an organization/repository)

```
docker tag testimage myhub/testimage:v1.0
```

Does this. Note that in the first parameter `:latest` is implied if no tag is given. The command tags the image we just made as living in the myhub repository in the image named testimage and tagged as v1.0. For your images, you'll want to use the correct repository and a more descriptive image.

Next, we need to provide login information to docker hub so that the push is properly authenticated in docker hub. What I type in the dialog below is in red:

```
docker login

Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to
create one.

Username:myhub

Password:myhub-password

WARNING! Your password will be stored unencrypted in
/home/deployer/.docker/config.json.

Configure a credential helper to remove this warning. See

https://docs.docker.com/engine/reference/commandline/login/#credential
s-store


Login Succeeded
```

Several things to note:

1. The password you type will not be echoed.
2. Since credentials are stored in clear text, you'll want to ensure they are cleared before finishing.

Now you can push the image:

```
Docker push myhub/testimage:v1.0
```

In the command above, naturally use the name you tagged the image with.

Building a singularity image from your docker image.

The form of the command to build a singularity image from a docker container image is:

```
singularity build image-filename docker://repository/image-name:tag
```

This in our example, we can build a singularity image using the command:

```
Singularity build testimage.img docker://myhub/testimage:v1.0
```

Which will eventually result in a file, testimage.img that is our new singularity image. We can build an image start script, just as we did with the base NSCLDAQ container images as long as we specify the name of our new image in the `singularity shell` command.