D12.2 - Container-as-a-service Technical Documentation

Deliverable No.:	D12.2
Deliverable Name:	Container-as-a-service Technical Documentation
Contractual Submission Date:	30/04/2019
Actual Submission Date:	30/04/2019
Version:	v1.1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730897.

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**HPC-Europa3**
Project Full Name:	Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing
Deliverable No.:	D12.2
Document name:	Container-as-a-service Technical Documentation
Nature (R, P, D, O):	R
Dissemination Level (PU, PP, RE, CO):	PU
Version:	v1.1
Actual Submission Date:	30/04/2019
Author, Institution: E-Mail:	Niall Wilson, Irish Centre for High End Computing niall.wilson@ichec.ie
Other contributors	Oleksandr Rudyy : Barcelona Supercomputing Center Atte Sillanpää : CSC

ABSTRACT: This document contains technical documentation related to the installation and usage of a number of container technologies used for HPC applications. It also demonstrates how container images can be created and used on different systems.

KEYWORD LIST:
High Performance Computing, HPC, Virtualization, Containers, Docker, Singularity

MODIFICATION CONTROL			
Version	**Date**	**Status**	**Author**
1.0	5/04/2019	Draft	Niall Wilson
1.1	26/04/2019	Final	Niall Wilson

TABLE OF CONTENTS

1 Executive Summary

The purpose of this document is to gather together technical documentation concerned with implementing the proposed use of containers in HPC to help researchers develop portable applications. The technology and software underpinning this "container-as-a-service" is evolving at a very rapid pace which is testament to its success but also makes it difficult to document comprehensive, up to date information. Hence the information contained here is meant to act as an introduction and guide to more detailed and specific documentation freely available on the web. It should be used by both HPC centre system administrators and HPC end users to get an overview of how this service was implemented on a test basis for this project and it is hoped that some of the guidance included will enable a more direct and quicker implementation of these services locally. Firstly, the installation procedures, configuration and usage guidelines as well as security considerations are described for each of the main technologies examined. This will enable users to install these systems on their own laptops and hence perform the first step in the proposed workflow of creating applications locally and then migrating them to HPC systems. Next, details are provided on image registries and an example is given of the process of containerising an application, exporting the image to a remote registry and then importing the image on a HPC system and running it there.

The other deliverables of the JRA will go into more detail on particular aspects of container usage including:

D12.3 – "Using containers technologies to improve portability of applications in HPC" which provides more details from the perspective of the HPC user on how to effectively containerise applications
D12.5 – "**Workload collocation based on containers technologies to improve isolation**" and
D12.6 – "Benchmarking" which both examine the performance implications of running applications in containers

2 Architecture Proposal Review

The main high level components of the container as a service architecture are as follows:

1. **Application Container Images**
   The enabling feature of application portability is being also to encapsulate all required software into a single portable image.
   This component includes tools to help package applications into images as well as a central repository to enable users to store and retrieve images. Basic authentication and search capabilities are also required. Technology choice here involves Docker and Singularity image formats and it is possible to convert Docker images to Singularity images.
2. **Container runtime technology**
   Different software implementations exist which enable the execution of containers on a Linux system. Each will have different requirements, advantages and disadvantages. The proposed architecture will allow each site or system to provide as many of these as they wish. Local security policies and requirements as well as software dependencies will determine which container runtime software is exposed to the users. Examples of requirements include the ability of the container runtime to interact with tools for workload co-location. Docker is the first technology with widespread adoption but other mature technologies like Singularity may be more suitable for HPC applications and environments, especially due to Docker's strong requirements on running with system privileges, which can jeopardize security in big infrastructures. Other HPC specific variants which can be made available include Shifter and Charliecloud.
3. **Workflow Manager**
   Different applications are frequently part of larger workflows and so containerised applications must be able to integrate with workflow managers which can chain together numerous applications and their corresponding input and output datasets. HPC systems remain predominantly batch oriented. As such, the execution of containerised applications will need to conform to the constraints of existing batch workload managers (eg Slurm, Torque, PBSPro) in a majority of sites. But more container-native orchestration tools such as Kubernetes are also of interest where cloud-like platforms are in use and should be included.

From a user perspective the workflow is as follows (Fig. 1):

1. User optionally creates an application container image and pushes it to either a public image repository or directly to a HPC system.
2. User logs onto HPC system and optionally pulls application container image from an image repository if image isn't already present on system. An image repository isn't mandatory - images can be manually copied or created on the system itself also.

3. User writes workflow job which could be as simple as a single application requiring just 1 node or multiple applications, chaining together input and output datasets run across multiple nodes. This job is then submitted to the workload manager.
4. The workflow manager is responsible for allocating the required hardware nodes and initiating the startup of the containerised applications. Data staging between network file systems and local node file systems may be performed here also.

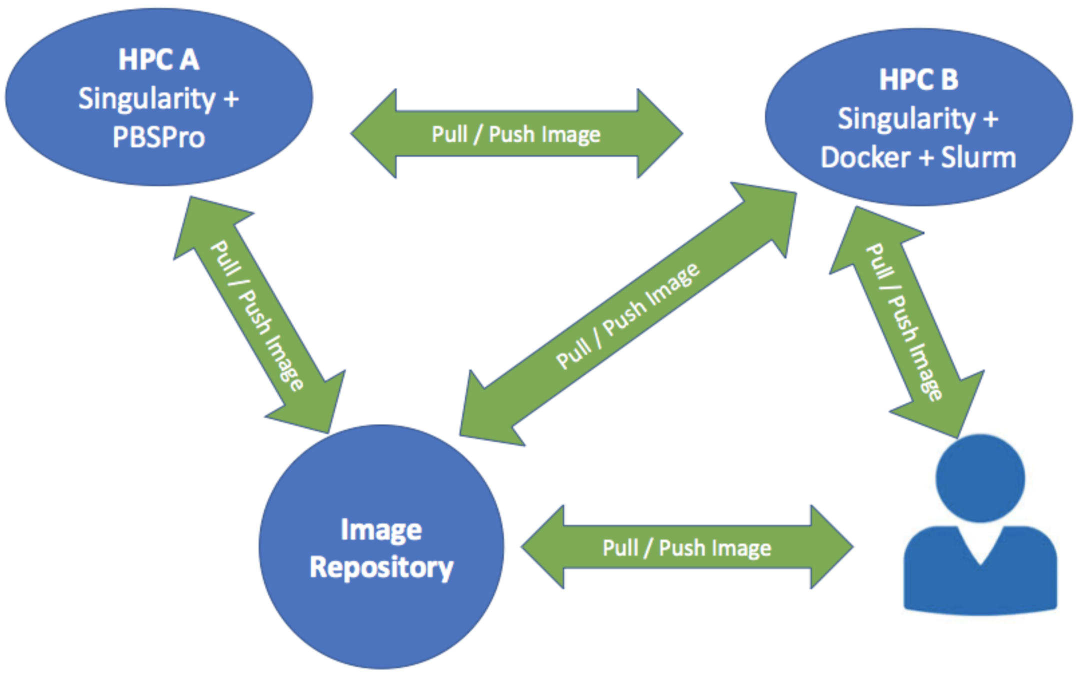


Figure 1: Workflow example from user perspective

Each HPC site or cluster within a particular site is free to support any one or combination of the above technologies. So a single cluster could support both Docker and Singularity as well as bare-metal jobs which is ideal for benchmarking purposes. Each cluster may also have a different workload manager but it is highly unlikely that multiple options would exist on the same cluster. The requirements and characteristics of each container runtime technology will necessitate different approaches to running applications. For example, Singularity can natively run images from parallel file systems such as Lustre which are common on HPC clusters but this is not true of Docker, and so additional work will be required to stage images into and out of compute node local storage.

So the proposed architecture to be implemented is:

1. An image repository to contain Docker and/or Singularity images accessible for read/write. This could also be a set of complementary repositories including Docker Hub, Singularity Hub and a dedicated HPC Europa 3 repository. Central to the concept though is that these repositories are available over the Internet to use at any HPC centre.

2. A number of clusters with at least one (and preferably more) container runtime implementations installed with a workload manager, multiple nodes and a parallel file system available to run both native bare-metal and containerised applications on the same hardware.
3. The ability for a user to create an application image locally on their laptop or home HPC system, and publish it for use by others or themselves on different remote HPC systems.
4. To run an application, the user logs into a specific HPC system, pulls the application image from the repository and submits a batch job to the workload manager to execute the set of containers required.

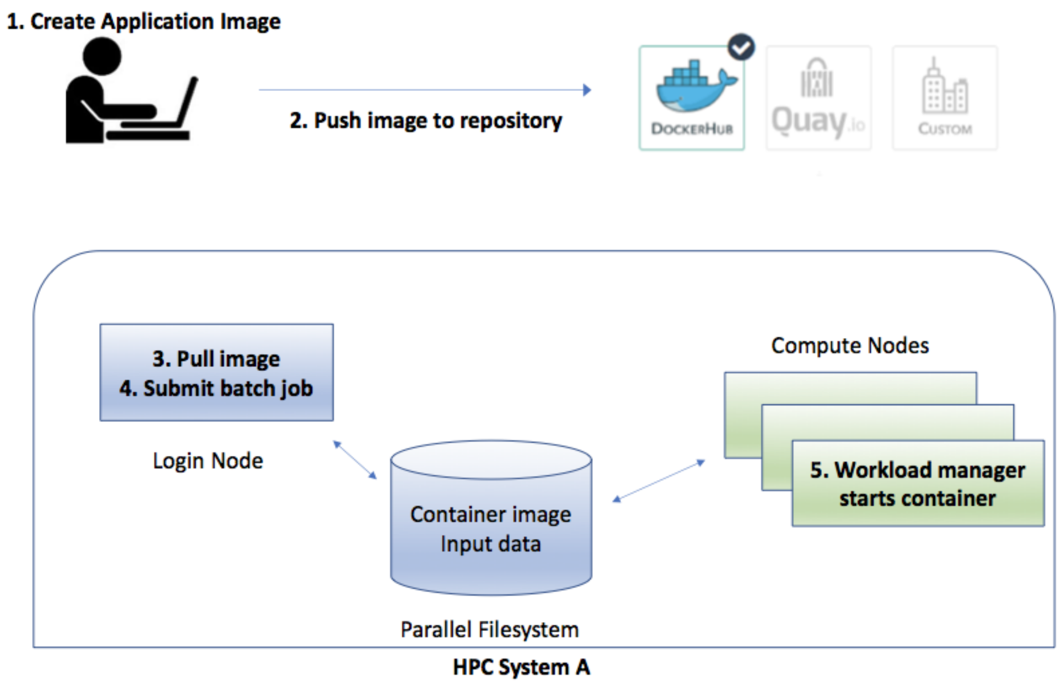


Figure 2: Architecture proposal diagram

3 Installation and configuration details

3.1 Docker

The documentation for Docker[1] is very comprehensive and so rather than reproduce detailed installation instructions here, only selected important details relevant to HPC will be discussed. Docker is available in two editions - Community Edition and Enterprise Edition. Most sites requiring just the core features of Docker will only require the Community Edition which is freely available through the major Linux distro package repositories or via Docker's own yum and apt repositories. One of the major benefits of containerisation is of course the ability to develop rich and diverse software environments locally on one's own laptop prior to pushing these container images out to larger systems for production work. In this respect Docker is an essential tool for local development work. Docker Desktop for Mac[2] and Docker Desktop for Windows[3] are packaged versions of Docker plus associated tools which use the native hypervisor technology of the respective platforms to install a Linux VM and then install and configure the Docker tools so that it works just like a native application. This enables users to build and test HPC Linux applications on their Apple or Windows laptop prior to moving to a HPC system for production runs.

3.1.1 Installation

As was discussed in D12.1, the specific architecture of how docker engine runs prevents it from being easily deployed on HPC clusters. For example, one major difficulty with deploying docker on a cluster of HPC nodes is that there is no storage driver for Lustre or GPFS which means that all container images must be stored on the local node disk with all the staging and synchronisation issues that that entails. Additionally, the docker daemon must be installed and run on all compute nodes along with a lot of custom configuration to enable privileged access by users and to incorporate HPC hardware such as GPUs and high performance networks.

However, having taken into account these caveats it is very straightforward to install Docker on individual nodes. Most major distributions include it in their package repositories or you can add Docker's own repository. So for example with a CentOS host the steps are:

1. Install necessary required packages
   ```
   $ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
   ```

2. Add the stable repository
   ```
   $ sudo yum-config-manager \
       --add-repo \
       https://download.docker.com/linux/centos/docker-ce.repo
   ```

3. Install Docker Community Edition
   ```
   $ sudo yum install docker-ce docker-ce-cli containerd.io
   ```

4. Start docker engine

```
$ sudo systemctl start docker
```

3.1.2 Configuration

There are two ways to configure the Docker daemon:

1. Use a JSON configuration file. This is the preferred option, since it keeps all configurations in a single place.
2. Use flags when starting dockerd

The JSON configuration file is usually found at `/etc/docker/daemon.json` on Linux systems.

3.1.3 Usage

The `docker` client command is used to issue instructions to the docker daemon. By default it tries to connect to a daemon on the same host but this can be changed in the config file or using the `--host` or `-H` option. The scope of subcommands and usage is illustrated via the output of `docker --help`:

```
Usage:      docker COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default
"/root/.docker")
  -D, --debug              Enable debug mode
      --help               Print usage
  -H, --host list          Daemon socket(s) to connect to (default [])
  -l, --log-level string   Set the logging level ("debug", "info", "warn",
"error", "fatal") (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default
"/root/.docker/ca.pem")
      --tlscert string     Path to TLS certificate file (default
"/root/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default
"/root/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit

Management Commands:
  container   Manage containers
  image       Manage images
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  volume      Manage volumes

Commands:
  attach      Attach to a running container
  build       Build an image from a Dockerfile
  commit      Create a new image from a containers changes
  cp          Copy files/folders between a container and the local
filesystem
  create      Create a new container
  diff        Inspect changes on a containers filesystem
  events      Get real time events from the server
  exec        Run a command in a running container
  export      Export a containers filesystem as a tar archive
  history     Show the history of an image
  images      List images
  import      Import the contents from a tarball to create a filesystem
image
  info        Display system-wide information
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  login       Log in to a Docker registry
  logout      Log out from a Docker registry
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  ps          List containers
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  run         Run a command in a new container
  save        Save one or more images to a tar archive (streamed to STDOUT
by default)
  search      Search the Docker Hub for images
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage
statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  version     Show the Docker version information
  wait        Block until one or more containers stop, then print their
exit codes

Run 'docker COMMAND --help' for more information on a command.
```

3.1.4 Security

As has been discussed in the original D12.1 document, Docker presents HPC centres with significant security issues in comparison to the alternative technologies, largely because of how it was designed and intended to be used (i.e not on shared, multi-user HPC clusters). The requirement to enable root privileges for all users of docker for normal functionality is an unacceptable security risk for most HPC centres. Additionally, a recently published exploit (CVE-2019-5736[4]) demonstrated the ability for malicious container images to overwrite the host runc binary and thus gain root-level code execution on the host.

3.2 Singularity

Documentation for Singularity[5] is very comprehensive. It is a much smaller and simpler toolset than Docker. Initial development was for Linux only but in March 2019 an Alpha version of Singularity Desktop for MacOS[6] was released which enabled Linux-based Singularity containers to be designed, built, tested, and signed/verified on macOS. However, at the time of writing it is only possible to build images for Singularity on MacOS by making use of the Remote Builder service from the Sylabs Cloud.
Unlike Docker, there is no daemon process with Singularity and it uses a single regular file for images rather than layers in a union filesystem and so it can be installed once on a shared filesystem such as Lustre and then used on all nodes in the cluster.

3.2.1 Installation

There are two common ways to install Singularity, from source code and via binary packages provided through standard RPM and apt repositories. The installation from source code is illustrated below but it is important to note two major caveats:

1. You can install Singularity into any directory of your choosing, but you must ensure that the location you select supports programs running as **SUID**. It is common to disable **SUID** with the mount option `nosuid` for various network mounted file systems so this should be checked.
2. The `make install` step must be run as root to have Singularity properly installed. Failure to install as root will cause Singularity to not function properly or have limited functionality when run by a non-root user.

Firstly, the necessary dependencies must be installed.
On Debian-based systems:

```
$ sudo apt-get update && \
  sudo apt-get install -y build-essential \
  libssl-dev uuid-dev libgpgme11-dev libseccomp-dev pkg-config squashfs-tools
```

On CentOS/RHEL:

```
$ sudo yum groupinstall -y 'Development Tools' && \
  sudo yum install -y epel-release && \
  sudo yum install -y golang openssl-devel libuuid-devel libseccomp-devel squashfs-
tools
```

Singularity requires the Go language when building from source and even if golang is already installed as with the CentOS instructions above, the Singularity build process requires a copy of the Go source.

```
$ go version
```

```
go version go1.11.4 linux/amd64

$ export VERSION=1.11.4 OS=linux ARCH=amd64
$ wget -O /tmp/go${VERSION}.${OS}-${ARCH}.tar.gz
https://dl.google.com/go/go${VERSION}.${OS}-${ARCH}.tar.gz && \
  sudo tar -C /usr/local -xzf /tmp/go${VERSION}.${OS}-${ARCH}.tar.gz
```

Finally, set up your environment for Go:

```
$ echo 'export GOPATH=${HOME}/go' >> ~/.bashrc && \
  echo 'export PATH=/usr/local/go/bin:${PATH}:${GOPATH}/bin' >> ~/.bashrc && \
  source ~/.bashrc
```

Now we are ready to clone a copy of the Singularity source code from Github

```
$ mkdir -p ${GOPATH}/src/github.com/sylabs && \
  cd ${GOPATH}/src/github.com/sylabs && \
  git clone https://github.com/sylabs/singularity.git && \
  cd singularity
```

To build a stable version of Singularity, check out a release tag before compiling:

```
$ git checkout v3.1.0
```

You can now build Singularity using the following commands:

```
$ cd ${GOPATH}/src/github.com/sylabs/singularity && \
  ./mconfig && \
  cd ./builddir && \
  make && \
  sudo make install
```

Singularity is now installed for all users. As a simple test you can check which version is installed by running:

```
$ singularity version
```

3.2.2 Configuration

The main Singularity configuration file is called `singularity.conf` and is where most configuration options are set. The default location is
`/usr/local/etc/singularity/singularity.conf`
While most default options defined in this file will not need to be changed, it is highly likely that some configuration changes will be required in order to reflect the local filesystem and storage mounting setup. The relevant section and associated comments are included here from the default config file:

```
 MOUNT HOME: [BOOL]
# DEFAULT: yes
# Should we automatically determine the calling user's home directory and
# attempt to mount it's base path into the container? If the --contain
option
# is used, the home directory will be created within the session directory
or
# can be overridden with the SINGULARITY_HOME or SINGULARITY_WORKDIR
# environment variables (or their corresponding command line options).
mount home = yes

# MOUNT TMP: [BOOL]
# DEFAULT: yes
# Should we automatically bind mount /tmp and /var/tmp into the container?
If
# the --contain option is used, both tmp locations will be created in the
# session directory or can be specified via the  SINGULARITY_WORKDIR
# environment variable (or the --workingdir command line option).
mount tmp = yes

# MOUNT HOSTFS: [BOOL]
# DEFAULT: no#
# Probe for all mounted file systems that are mounted on the host, and bind
# those into the container?
mount hostfs = no

# BIND PATH: [STRING]
# DEFAULT: Undefined
# Define a list of files/directories that should be made available from
within
# the container. The file or directory must exist within the container on
# which to attach to. you can specify a different source and destination
# path (respectively) with a colon; otherwise source and dest are the same.
```

```
#bind path = /etc/singularity/default-nsswitch.conf:/etc/nsswitch.conf
#bind path = /opt
#bind path = /scratch
bind path = /etc/localtime
bind path = /etc/hosts

# USER BIND CONTROL: [BOOL]
# DEFAULT: yes
# Allow users to influence and/or define bind points at runtime? This will
allow
# users to specify bind points, scratch and tmp locations. (note: User bind
# control is only allowed if the host also supports PR_SET_NO_NEW_PRIVS)
user bind control = yes
```

3.2.3 Usage

In contrast to Docker, Singularity does not use a daemon and so it can be installed on a shared filesystem and used on any node mounting this filesystem. As with Docker, it uses a single executable with a series of subcommands:

```
$ singularity --help
USAGE: singularity [global options...] <command> [command options...] ...

GLOBAL OPTIONS:
    -d|--debug    Print debugging information
    -h|--help     Display usage summary
    -s|--silent   Only print errors
    -q|--quiet    Suppress all normal output
       --version  Show application version
    -v|--verbose  Increase verbosity +1
    -x|--sh-debug Print shell wrapper debugging information

GENERAL COMMANDS:
    help       Show additional help for a command or container
    selftest   Run some self tests for singularity install

CONTAINER USAGE COMMANDS:
    exec       Execute a command within container
    run        Launch a runscript within container
    shell      Run a Bourne shell within container
    test       Launch a testscript within container
```

```
CONTAINER MANAGEMENT COMMANDS:
    apps        List available apps within a container
    bootstrap   *Deprecated* use build instead
    build       Build a new Singularity container
    check       Perform container lint checks
    inspect     Display container's metadata
    mount       Mount a Singularity container image
    pull        Pull a Singularity/Docker container to $PWD

COMMAND GROUPS:
    image       Container image command group
    instance    Persistent instance command group


CONTAINER USAGE OPTIONS:
    see singularity help <command>

For any additional help or support visit the Singularity
website: https://www.sylabs.io/
```

3.2.4 Security

The Singularity security model is to enable untrusted users (those who don’t have root access) to run untrusted containers (those that have not been vetted by admins) safely. To do this, Singularity’s design forces a user to have the same UID and GID context inside and outside of the container. This is accomplished by dynamically writing entries to `/etc/passwd` and `/etc/groups` at runtime. This design has the additional benefit of making it easy for a user inside the container to safely read and write data to the host system with correct the ownership and permissions.

Secondly, Singularity mounts the container file system with the `nosuid` flag and executes processes within the container with the `PR_SET_NO_NEW_PRIVS` bit set. Combined with the fact that the user is the same inside and outside of the container, this prevents a user from escalating privileges.

A malicious container may not be able to damage the system, but it could still do harm in the user’s space without escalating privileges. Starting in Singularity 3.0, containers may be cryptographically signed when they are built and verified at runtime via PGP keys. This allows a user to ensure that a container is a bit-for-bit reproduction of the container produced by the original author before they run it. As long as the user trusts the creator of the container, they can run it safely.

3.3 Charliecloud

Charliecloud[7] is a very small codebase maintained and published via Github with good documentation[8]. It requires a relatively recent Linux kernel (> v4.4) with User Namespaces enabled and is intended to be used with Docker images which it converts to flat tar archives. There are no prepackaged Mac OS or Windows versions available so to use it on these platforms requires the explicit provisioning of a Linux VM followed by installation and configuration of Charliecloud within the VM. Charliecloud uses Docker to build local images and pull images from Dockerhub so Docker must be installed and the user must have sufficient privileges to run the underlying Docker commands if these conveniences are required in addition to just executing containers.

3.3.1 Installation

While prebuilt packages are available the preferred method of installation is to clone the git repository and build from source:

```
$ zgrep CONFIG_USER_NS /proc/config.gz
CONFIG_USER_NS=y

$ git clone --recursive https://github.com/hpc/charliecloud.git

$ cd charliecloud

$ make

$ make install PREFIX=$HOME/cc
```

The installation is comprised of a number of executables installed under the bin subdirectory wherever Charliecloud was installed:

```
$ ls $HOME/cc/bin
ch-build  ch-build2dir  ch-docker2tar  ch-fromhost  ch-pull2dir  ch-pull2tar
ch-run  ch-ssh  ch-tar2dir
```

3.3.2 Configuration

Charliecloud does not run as a daemon and doesn't use a configuration file so all configuration options which depart from the default must be supplied on the commandline. The most frequently used commandline option is `--bind` or `-b` which allows the user to bind mount additional host directories into the container. Several host directories are always bind-mounted into the container. These include system directories such as `/dev`, `/proc`, `/sys`, `/tmp`; Charliecloud's `ch-ssh` command in `/usr/bin`; and the invoking user's home directory unless `--no-home` is specified. To add additional directories, simply add one or more `-b` arguments when launching the container (the source and destination directory names can be separated by a ":" or if omitted, the default is to mount under /mnt/0, mnt/1, etc inside the container). For example, to mount the local host directory `/work/project1` under `/mnt` inside the container:

```
ch-run -b /work/project1:/mnt /var/tmp/my_app -- bash
```

This is also used to try to ensure that user and group IDs are the same inside the container as on the host by bind mounting `/etc/passwd` and `/etc/group`. The `--uid` option can be used to override this but the user namespace kernel feature will map it back to your regular uid before any host interactions are performed and so normal permissions still apply even if you launch a container with the root uid 0.

3.3.3 Usage

Many of the Charliecloud commands such as `ch-build`,`ch-build2dir`,`ch-docker2tar`, `ch-fromhost`,`ch-pull2dir`,`ch-pull2ta`,`ch-tar2dir` are used to build, fetch and manipulate Docker images. The main command used to execute containers is `ch-run`

```
Usage: ch-run [OPTION...] NEWROOT CMD [ARG...]

Run a command in a Charliecloud container.

  -b, --bind=SRC[:DST]   mount SRC at guest DST (default /mnt/0, /mnt/1,
etc.)
  -c, --cd=DIR           initial working directory in container
  -g, --gid=GID          run as GID within container
  -j, --join             use same container as peer ch-run
      --join-ct=N        number of ch-run peers (implies --join)
      --join-pid=PID     join a namespace using a PID
      --join-tag=TAG     label for peer group (implies --join)
      --no-home          do not bind-mount your home directory
      --set-env=FILE     set environment variables in FILE
  -t, --private-tmp      use container-private /tmp
  -u, --uid=UID          run as UID within container
      --unset-env=GLOB   unset environment variable(s)
  -v, --verbose          be more verbose (debug if repeated)
  -w, --write            mount image read-write
  -?, --help             Give this help list
      --usage            Give a short usage message
  -V, --version          print version and exit

Example:

  $ ch-run /data/foo -- echo hello
  hello
```

3.3.4 Security

The Charliecloud security model does not use setuid capability but instead uses the newer USER namespace kernel capability. The goal of USER is to give unprivileged processes access to traditionally privileged functionality in specific contexts when doing so is safe. This feature is relatively new in the Linux kernel (introduced in 2013) so systems should be kept up to date with new kernel security patches.

3.4 Shifter

With the current status of Shifter's documentation it is not easy to find manuals about its installation, Command-Line-Interface (CLI) calls or its configuration. Although you can find over the internet many webpages with Shifter tutorials or manuals, there is only one official documentation intended to general users in its GitHub repository[9], whose documentation is scarce and some sections are unfinished.

Shifter containers needs Linux's kernel *chroot* and *namespaces* capabilities to run. As soon as your computer uses Linux operating system you should be able to run Shifter on it. Shifter is made of 4 components: the ImageGateway, the CLI, the *udiRoot* and the Workload Manager. The Workload Manager component is only designed to integrate SLURM with Shifter and provide useful functionalities for the container deployment. Despite Shifter runtime does not need any daemon to run containers, its ImageGateway component depends of a MongoDB database and MUNGE authentication service.

3.4.1 Installation

The only way to install Shifter is from the source code available in NERSC/shifter GitHub repository. You can install Shifter through the conventional process using a Makefile or through RPM packages built by your own. Shifter leverages **SUID** file permission to run its containers, therefore you must check that your system enables this option and you install Shifter runtime as root. In addition, your system must be able to leverage loop devices and have support for the squashfs file system. But before all, you need to install Shifter's dependencies in your distribution. The set of packages Shifter requires is the following:

```
rpm-build gcc glibc-devel munge libcurl-devel json-c json-c-devel pam-devel
munge-devel libtool autoconf automake gcc-c++ python-pip xfsprogs squashfs-
tools python-devel libcap-devel python-flask python-gunicorn python-pymongo
```

Now you can download the sources and begin with the installation.

```
git clone https://github.com/NERSC/shifter.git
```

To install Shifter through RPM you will need to generate the appropriate tarball:

```
VERSION=$(grep Version: shifter/shifter.spec | awk '{print $2}')
cp -rp shifter "shifter-$VERSION"
tar cf "shifter-$VERSION.tar.gz" "shifter-$VERSION"
```

Then, you can generate Shifter's *.rpm* package with:

```
rpmbuild -tb "shifter-$VERSION.tar.gz"
```

This command will check if your system satisfies Shifter's requirements and will store the generated RPM packages in your home directory. In case you missed some dependency it will notify you with an error.

It is important to remark that during the RPM building Shifter might try to download and install some packages on its own. If your cluster does not have a connection to the internet, you will be forced to hack a little its sources files. For this, before generating your tarball, you will have to get into *`shifter/dep`* directory and modify the *`build_mount.sh`* and *`cppustest.sh`* scripts to not to download the packages. As a result, you will need to install manually the packages defined in that two scripts.

Once you have generated the RPM packages the following steps are divided in

1. The installation of the ImageGateway
2. The installation of Shifter's runtime

It is only necessary to run the ImageGateway service in one node (e.g., your login node). All your other nodes must have access to that service through the network. For this, your nodes must share the same MUNGE key and have access to the *imagegwapi* port (typically 5000) on the ImageGateway node. Shifter's runtime, on the contrary, must be installed in all your nodes.

To install the Image Manager through RPMs:

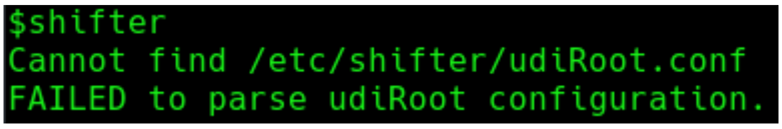
```
sudo rpm -i /path/to/rpmbuild/RPMS/x86_64/shifter-imagegw-$VERSION.rpm
```

Right now your system posses the ImageGateway component of Shifter, yet it is not functional as it needs to be configured and to have available MongoDB and Munge services, which we will discuss in the next section.

Now, you can install Shifter's runtime:

```
sudo rpm -i /path/to/rpmbuild/RPMS/x86_64/shifter-runtime-$VERSION.rpm
```

After this you should have access to `shifter` and `shifterimg` commands, but if you try to execute them you will get an error like this:


```
$shifter
Cannot find /etc/shifter/udiRoot.conf
FAILED to parse udiRoot configuration.
```

This is because you still have to configure Shifter.

If you would rather install Shifter with the conventional Makefile workflow you should execute:

```
git clone https://github.com/NERSC/shifter.git
cd shifter
./autogen
./configure --prefix=/usr
make
sudo make install
```

3.4.2 Configuration

After the installation Shifter needs its ImageGateway and runtime to be configured. However, before starting with the ImageGateway configuration we will set up the required MongoDB and MUNGE services.

First, we will launch MongoDB:

```
mkdir -p /data/db
mongod --smallfiles &
```

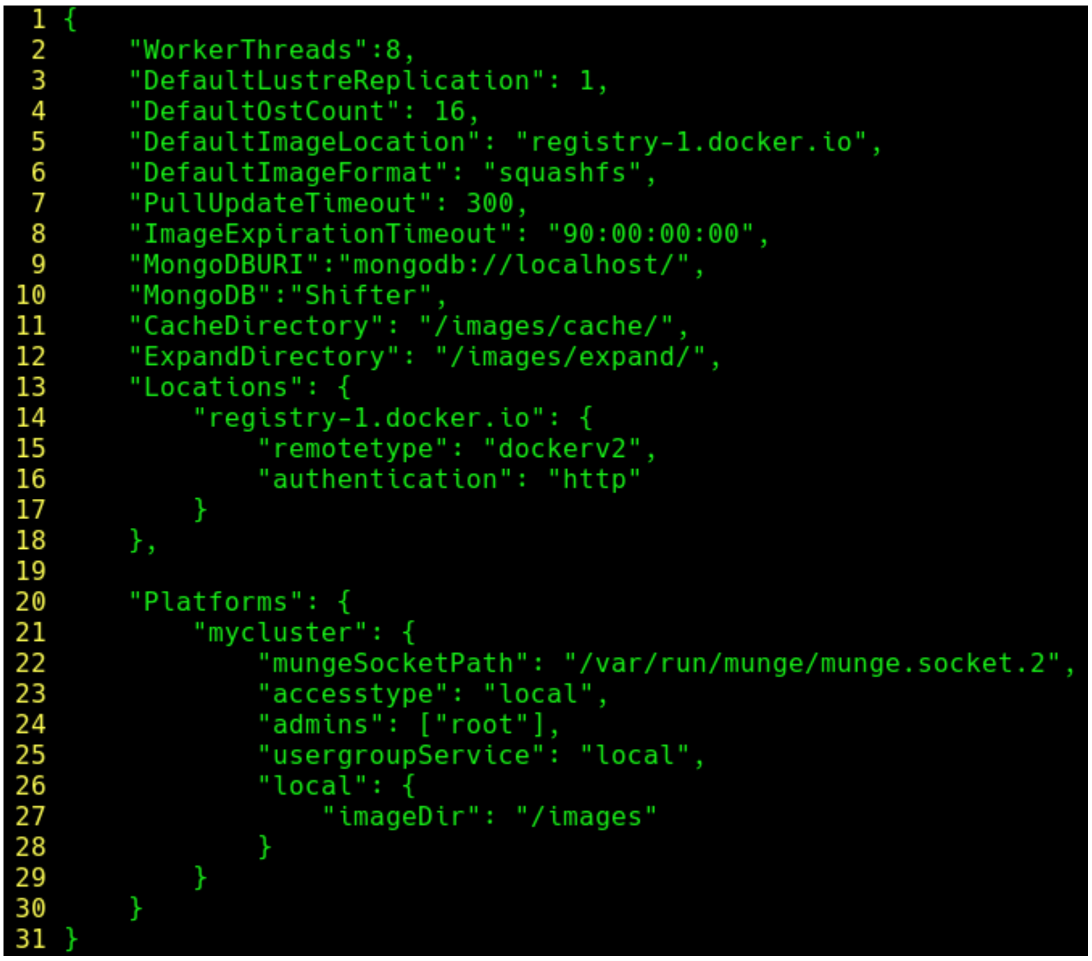
This will create a folder where Mongo will store its data and start mongod daemon.

Next, we will start MUNGE by setting the key we want to use and starting the service. The following key is only an example and you should not use the same.

```
echo "abcdefghijklkmnopqrstuvwxyz0123456" > /etc/munge/munge.key
chown munge.munge /etc/munge/munge.key
chmod 600 /etc/munge/munge.key
systemctl enable munge
systemctl start munge
```

The `systemctl enable munge` will ensure that MUNGE service is executed after each boot.

In order to configure the ImageGateway, you will find an example of its configuration file in the folder **/etc/shifter** named *imagemanager.json.example*:



```
{
    "WorkerThreads":8,
    "DefaultLustreReplication": 1,
    "DefaultOstCount": 16,
    "DefaultImageLocation": "registry-1.docker.io",
    "DefaultImageFormat": "squashfs",
    "PullUpdateTimeout": 300,
    "ImageExpirationTimeout": "90:00:00:00",
    "MongoDBURI":"mongodb://localhost/",
    "MongoDB":"Shifter",
    "CacheDirectory": "/images/cache/",
    "ExpandDirectory": "/images/expand/",
    "Locations": {
        "registry-1.docker.io": {
            "remotetype": "dockerv2",
            "authentication": "http"
        }
    },

    "Platforms": {
        "mycluster": {
            "mungeSocketPath": "/var/run/munge/munge.socket.2",
            "accesstype": "local",
            "admins": ["root"],
            "usergroupService": "local",
            "local": {
                "imageDir": "/images"
            }
        }
    }
}
```

You must copy this file as *imagemanager.json* and at minimum check that:

- "MongoDBURI" is the correct URL to shifter imagegw mongodb server
- "CacheDirectory" exists (semi-permanent storage for docker layers)
- "ExpandDirectory" exists (temporary storage for converting images)
- The "imageDir" is set correctly for your system (storage for Shifter images)

CacheDirectory and ExpandDirectory need only to be visible in the system where you installed the ImageGateway. The imageDir directory, contrarily, must be accessible to all your nodes since it is where definitive Shifter images will be stored.

Finally, you can initiate the ImageGateway and an ImageGateway worker for your cluster:

```
gunicorn -b 0.0.0.0:5000 --backlog 2048 shifter_imagegw.api:app
```

In order to configure Shifter runtime, you also will find in **/etc/shifter** the file *udiRoot.conf.example* which you must copy as *udiRoot.conf.* You will notice that this config file have a lot of options. You are free to tune it as you wish, but at least you should change:

- The value of your system's name so it matches the platform name from *imagemanager.json*.
- Set the URL for ImageGateway to match your imagegw machine, no trailing slash. For example, if the hostname of the machine containing the Image Manager is "foo123" you

should set the URL as http://foo123:5000, if it is your localhost, you should set it to http://localhost:5000.

- Set the **etcPath** field so it points to a directory containing the *group* and *passwd* files you wish your container to use.
- Set the **imagePath** field pointing to your imageDir from *imagemanager.json*.

After these steps, you should be able to run `shifter` and `shifterimg` commands without problems.

3.4.3 Usage

Shifter's CLI consists of 2 commands:

- **shifter**: offers the ability to summon containers given an image. It possesses options to define the container's environment, entry point or working directory. It also offers the "volume" functionality to attach host directories within the container. `shifter -h` will display its help with all the details.

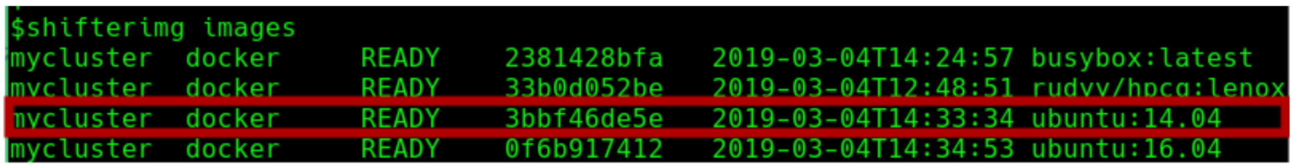
- **shifterimg**: this command interacts with the ImageGateway and offers 3 functionalities.
  - List the available Shifter images in the system.
  - Lookup for the identifier of a stored image in your imageDir.
  - Pull and convert images from Docker repositories.

As an important remark: **shifterimg** does allow the pulling of new images, however, it does not offer the capacity to neither remove or rename pulled images. If you have in your system Shifter images that you wish to delete, you will have to remove them manually from the image directory defined in your configuration files.

As a use-case example, imagine that you want to download Ubuntu's 14.04 image from Docker Hub and run it with Shifter. To achieve this you should first download the image:

```
shifterimg pull ubuntu:14.04
```

To check whether your download was successful you can list the available images:



```
$shifterimg images
mycluster  docker     READY    2381428bfa   2019-03-04T14:24:57 busybox:latest
mycluster  docker     READY    33b0d052be   2019-03-04T12:48:51 rudyy/hpcg:lenox
mycluster  docker     READY    3bbf46de5e   2019-03-04T14:33:34 ubuntu:14.04
mycluster  docker     READY    0f6b917412   2019-03-04T14:34:53 ubuntu:16.04
```

To run your container you can simply execute:

```
shifter --image=ubuntu:14.04
```

3.4.4 Security

Shifter's aim is to leverage the already existing Docker infrastructure but in a way that it becomes a suitable container implementation for HPC systems. To achieve this, Shifter addresses to Docker security issues, more specifically, to the Docker's root owned daemon. While with Docker users launch their containers with root capabilities through the Docker daemon, Shifter binary is setuid-root to execute all the privileged system calls needed for the container's environment set up before dropping root capabilities. Once the container's basic infrastructure is setup, the container adopts the privileges of the user who started it. Within containers, the only possible existing users and groups are those defined within the *group* and *passwd* files in **etcPath**. In the end, Shifter's goal is to ensure that the user running inside the container is the same user who invoked it. In addition, to avoid user-escalation within containers Shifter activates the "nosuid" mount option.

One Shifter issue is that it uses loop devices to mount container file systems, which are directly managed by the kernel who has privileged access. Consequently, you should forbid users write into loop devices directly. In a similar way, directly importing squashfs images represents a potential security risk.

Another security problem consists of Linux ability to attach security attributes to individuals files from file systems. For example, within a file system with the "no-suid" it is possible to possess binaries with the SUID flag activated and operative. To solve this, Shifter's ImageGateway implements methods to prevent these dangerous attributes as well as its runtime to reduce the risks.

Regarding the ImageGateway, it is not necessary to run it with root privileges. In fact, is advisable to start it as a non-root user to prevent having images with Linux security capabilities embedded. The access to the imagegw API is controlled via Munge authentication. Munge is a light-weight, shared key authentication system that allows a privileged daemon process to authenticate the user who is calling the process.

3.5 Image Registry

Efficient usage of containers requires them to be easily available for download and deployment. The standard solution is to manage them in an image registry. Your organization might have one available and in that case it makes sense to use it. Some registries can only be accessed with certain accounts or from a certain IP range and therefore customized solutions may be needed. The following chapters describe two such cases.

3.5.1 OpenShift registry

This project utilized the OpenShift registry at CSC. The OpenShift internal registry is installed with built-in RedHat provided OpenShift installer, which leverages Ansible and OpenStack Heat orchestration tool. The end result of the installation is a regular OpenShift application, albeit a one with elevated access rights, running in a Pod controlled by a DeploymentConfig. The container images are stored in a persistent volume which is GlusterFS on top of CEPH in the case of Rahti.

3.5.2 Harbor container image registry

In case you don't have an OpenShift platform available, a separate image registry can be installed on a cloud platform or some other (virtual) machine providing services. At CSC, a general IaaS cloud service branded as cPouta based on OpenStack is available for research purposes. The following steps show how to install the harbor container image registry on a Vanilla ubuntu-18.04 virtual machine in cPouta OpenStack cluster, but the steps should be rather general.

1. Follow the official instructions (`https://github.com/goharbor/harbor/blob/master/docs/installation_guide.md`)
   Download the online installer https://github.com/goharbor/harbor/releases
   `https://storage.googleapis.com/harbor-releases/release-1.7.0/harbor-online-installer-v1.7.5.tgz`

2. Install prerequisites:
   - `Python 2.7`
   - `Docker engine > 1.10`
   - `Docker compose > 1.6.0`
   - `OpenSSL ~ latest`

3. Get certificate and DNS

   - open `0.0.0.0/0 port 80` security group
   - Get a DNS: `<dns>`
   - Get certificates (e.g. certbot and letsencrypt.org)
   - close `0.0.0.0/0 port 80` security group

4. Edit `harbor.cfg`

   - `hostname: <dns>`

- `ssl_cert, ssl_cert_key`
- `ui_url_protocol: https`
- edit `harbor_admin_password` for admins first time password

5. Run `install.sh --with-clair --with-notary`

6. Configure OpenStack security groups as appropriate

Pros and cons of Harbor

The steps above will install an image registry to be used for managing containers. Naturally, there are also other options and while this setup will work, it is not an optimal solution for production work without additional configuration. The following lists provide things to considering when selecting a registry.

Advantages

- Manual installation using bash script
- In OpenStack network security control is easy
  - Whitelist HPC cluster IPs only for https access
- Parts 1 and 2 can be packed as a base image
- Harbor is in incubating state in Cloud Native Computing Foundation
- Harbor is free
- Could be completely automated with ansible & heat
- Can be re-deployed with new harbor.cfg while keeping stored images and other user side configuration intact with install.sh

Disadvantages

- Manual installation using bash script
- Not tested with external access management
- Not analyzed thoroughly if harbor is a good piece of software
- More suitable base image might be some CentOS
- Harbor is not "enterprise" software
- Using whitelist complexifies admin work
- Online installer requires internet access
  - Offline installer should mitigate this
- Does not scale horizontally
- Container images are stored within the VM volume and not to external volume causing the virtual machine image to bloat when new container images are uploaded

4 Image workflow and usage

As an example of the full end to end workflow we will illustrate here how a user can build and test an application using a standard software environment on their laptop and then export an image of this container to a remote repository from where it can be pulled to a HPC system and executed. The process here will use Docker to build an image locally on a laptop and then push this image to Dockerhub. It will then use Singularity on the HPC system to pull the Docker image, convert it to a Singularity image and run the application via the batch scheduling system. This choice of tools reflects those most commonly encountered and is therefore representative of the typical workflow which researchers will use.

For the purposes of illustrating a non-trivial, real-world example of containerising an application, we will use the WRF[10] (Weather Research and Forecasting) code which is very widely used and famously difficult to build. Due to the number of library dependencies required to build WRF we will leverage another standard scientific programming tool - EasyBuild[11] to install and manage the environment for all dependencies. EasyBuild itself can be used to install applications and has some recipies for older versions of WRF but here we choose to use it just to install the dependencies and to build the latest release of WRF from source.

4.1 Image Creation

Typically there are two methods to create a new container image:

1. Modify and existing image by launching it in a container, installing or modifying software, configs or files within the container and then saving this container as a new image.
2. Build the image using a well known base image and a reproducible, scripted set of operations. The use of Dockerfiles and Singularityfiles defined by these respective technologies enables this ability to explicitly script how an image is built. This method is preferable due to being transparent, reproducible, self documenting and extensible.

To illustrate the image creation on a users laptop we will use Docker. Docker provides very easy to install and use packages for end users across all of the main platforms. For example, even though Docker requires a Linux operating system by definition, Docker for Mac is a package which automatically installs Docker in a Linux virtual machine running on the native Mac hypervisor such that a user on an Apple macOS laptop can use Docker in exactly the same way that a Linux user can. Similar packages also exist for Windows. The image creation illustrated below was performed on an Apple laptop using Docker Desktop.

The Dockerfile below contains everything to build the latest version of WRF within an EasyBuild environment on top of a minimal base distribution of Centos Linux.

```
FROM centos:latest

# install dependencies
RUN  yum update -y && \
     yum install -y wget && \
     wget https://dl.fedoraproject.org/pub/epel/7/x86_64/Packages/e/epel-
release-7-11.noarch.rpm && \
     rpm -Uvh epel-release-7-11.noarch.rpm && \
     rm -f /epel-release-7-11.noarch.rpm && \
     yum update -y && \
     yum install -y unzip tcsh sudo bzip2 patch mlocate tcl tcl-devel \
        time lua lua-posix lua-filesystem lua-devel openssl-devel \
        libibverbs-devel which python-pip && \
     pip install --upgrade pip && \
     pip install --upgrade setuptools && \
     yum groupinstall -y 'Development Tools'

# install Lmod
RUN  wget https://downloads.sourceforge.net/project/lmod/Lmod-7.8.tar.bz2
&& \
     tar xvf Lmod-7.8.tar.bz2 && \
     cd Lmod-7.8 && \
     ./configure --prefix=/opt/apps && \
     make install && \
     cp /opt/apps/lmod/lmod/init/profile /etc/profile.d/profile.sh && \
     rm -rf /Lmod-7.8* && \
     cd && \
     source /etc/profile.d/profile.sh

# install EasyBuild
RUN  cd / && \
     wget https://raw.githubusercontent.com/hpcugent/easybuild-
framework/develop/easybuild/scripts/bootstrap_eb.py && \
     PREFIX=/opt/apps/easybuild && \
     chmod 777 /opt /opt/apps && \
     useradd easybuild && \
     mkdir /easybuild && \
     chown easybuild:easybuild /easybuild && \
     runuser easybuild -c "source /etc/profile.d/profile.sh && python
/bootstrap_eb.py $PREFIX" && \
     chmod 755 /opt /opt/apps && \
     echo "easybuild ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers

# set env
# must still source /etc/profile at start
ENV MODULEPATH /opt/apps/easybuild/modules/all:${MODULEPATH}
ENV EASYBUILD_INSTALLPATH /opt/apps/easybuild

RUN runuser easybuild -c "source /etc/profile.d/profile.sh && module load
```

```
EasyBuild && \
        find /opt/apps -name Doxygen-1.8.14-GCCcore-7.3.0.eb && \
        sed -i
's/http:\/\/ftp\.stack\.nl\/pub\/users\/dimitri/http:\/\/doxygen\.nl\/files
/g'  /opt/apps/easybuild/software/EasyBuild/3.8.0/lib/python2.7/site-
packages/easybuild_easyconfigs-3.8.0-
py2.7.egg/easybuild/easyconfigs/d/Doxygen/Doxygen-1.8.14-GCCcore-7.3.0.eb
&& \
        sed -i 's/^runtest.*//g'
/opt/apps/easybuild/software/EasyBuild/3.8.0/lib/python2.7/site-
packages/easybuild_easyconfigs-3.8.0-
py2.7.egg/easybuild/easyconfigs/f/FFTW/FFTW-3.3.8-gompi-2018b.eb && \
        eb netCDF-Fortran-4.4.4-foss-2018b.eb --robot "

# Build WRF using Easybuild modules
ENV NETCDF_classic 1
ENV NETCDF /opt/apps/easybuild/software/netCDF-Fortran/4.4.4-foss-2018b
RUN runuser easybuild -c "source /etc/profile.d/profile.sh && module load
netCDF-Fortran/4.4.4-foss-2018b && \
        module list && sudo chown easybuild:easybuild /opt/apps && \
        cd /opt/apps && wget
http://www2.mmm.ucar.edu/wrf/src/WRFV4.0.TAR.gz && \
        tar xzf WRFV4.0.TAR.gz && \
        cd WRF && ./configure <<< $'34\r1\r' && \
        /bin/csh ./compile em_fire"
```

The main parts of the Dockerfile are:

1. Starting from the well known standard centos base image, run a package upgrade and then install the required packages which EasyBuild depends upon.
2. Install Lmod - the environment modules package which EasyBuild uses for managing environment variables.
3. Install EasyBuild itself using a non-root user.
4. Use EasyBuild to install the netCDF-Fortran-4.4.4-foss-2018b.eb recipe which will install all of the compilers, tools and libraries which WRF requires. Some bug fixes are required in this section using sed to edit some of the recipe files.
5. Finally, load the appropriate EasyBuild environment, download the WRF source code, run the configure scripts and build a particular test case.

To build the image from this Dockerfile, simply execute the following command from within the same directory as the Dockerfile:

```
docker build -t wrf_em_fire:v1 .
```

Due to the large number of dependencies to be built, this particular image takes many hours to build even on a modern fast laptop. But eventually, a 2.97GB image is created locally called wrf_em_fire with a tag of v1.
We can then test the application by launching a container with Docker and running WRF locally:

```
$ docker run -ti wrf_em_fire:v1 /bin/bash
[root@99a95c8fe29f /]# cd /opt/apps/WRF
[root@99a95c8fe29f WRF]# source /etc/profile.d/profile.sh && module load
netCDF-Fortran/4.4.4-foss-2018b
[root@99a95c8fe29f WRF]# cd test/em_fire/
[root@99a95c8fe29f em_fire]# ls
create_links.sh  input_sounding_hill_simple  namelist.fire_hill_simple
namelist.input_hill_simple  README.txt
ideal.exe        input_sounding_two_fires    namelist.fire_two_fires
namelist.input_two_fires    wrf.exe
input_sounding   namelist.fire               namelist.input
README.namelist
[root@99a95c8fe29f em_fire]# ./ideal.exe
 starting wrf task            0  of           1
[root@99a95c8fe29f em_fire]# ./wrf.exe
 starting wrf task            0  of           1
```

4.2 Image Export

Docker images are composed of layers in a union filesystem and so to copy this image to another system we can either use Dockerhub to transparently push the layers to or we can export the layers to a tarfile (via `docker save`) and manually copy that. As part of the typical workflow outlined, we will now push this newly created image to our own private image registry. Public registries such as Dockerhub operate in the same manner. For all registries you will typically require a username and password to authenticate.

```
$ docker tag wrf_em_fire:v1 docker-registry.rathi.csc.fi:8443/niwilson/eb-
wrf:v1
$ docker push docker-registry.rathi.csc.fi:8443/niwilson/eb-wrf:v1
```

4.3 Image import

Once published in a registry or copied over to a local filesystem, it is straightforward for each of docker, singularity, charliecloud or shifter to import, convert or download (pull) an image for use locally. The most direct method is to pull the image from a registry and the example below represents a very common use case whereby singularity pulls an image from Dockerhub and converts it to a singularity format image locally.

```
$ singularity pull docker://ubuntu:latest
WARNING: pull for Docker Hub is not guaranteed to produce the
WARNING: same image on repeated pull. Use Singularity Registry
WARNING: (shub://) to pull exactly equivalent images.
Docker image path: index.docker.io/library/ubuntu:latest
Cache folder set to /ichec/home/staff/nwilson/.singularity/docker
[4/4] |===================================| 100.0%
Importing: base Singularity environment
Exploding layer:
sha256:898c46f3b1a1f39827ed135f020c32e2038c87ae0690a8fe73d94e5df9e6a2d6.tar
.gz
Exploding layer:
sha256:63366dfa0a5076458e37ebae948bc7823bab256ca27e09ab94d298e37df4c2a3.tar
.gz
Exploding layer:
sha256:041d4cd74a929bc4b66ee955ab5b229de098fa389d1a1fb9565e536d8878e15f.tar
.gz
Exploding layer:
sha256:6e1bee0f8701f0ae53a5129dc82115967ae36faa30d7701b195dfc6ec317a51d.tar
.gz
Exploding layer:
sha256:c6a9ef4b9995d615851d7786fbc2fe72f72321bee1a87d66919b881a0336525a.tar
.gz
WARNING: Building container as an unprivileged user. If you run this
container as root
WARNING: it may be missing some functionality.
Building Singularity image...
Singularity container built: ./ubuntu-latest.simg
Cleaning up...
Done. Container is at: ./ubuntu-latest.simg
```

5 Example runs

Using the image created in Section 3.1 as an example we will now demonstrate how we can use containers to run the exact same application on multiple systems. At the end of Section 3.1 we illustrated using Docker on a Mac how we could launch the WRF container and then run some of the sample tests within the container. The next step would be to run this same test case on a HPC system for greater speed and scalability.

First, we must push the local docker image to a central registry such as Docker Hub:

```
$ docker tag eb-wrf:v2 ngwilson/eb-wrf:v2
$ docker push ngwilson/eb-wrf:v2
The push refers to repository [docker.io/ngwilson/eb-wrf]
749266c926cc: Pushed
11b19f5f80d0: Pushed
fe25f6104ad9: Pushed
596cf35da5c2: Pushed
dab3058cb0e8: Pushed
071d8bd76517: Pushed
v2: digest:
sha256:002ea21c3a0a1e8cfee488dd942c8b1bc605d5673b4faf39be6cc2bbb71cc06f
size: 1589
```

Next, we log into the HPC system and use singularity to fetch the docker image and convert it to a singularity image:

```
$ module load singularity
$ singularity pull docker://ngwilson/eb-wrf:v2
WARNING: pull for Docker Hub is not guaranteed to produce the
WARNING: same image on repeated pull. Use Singularity Registry
WARNING: (shub://) to pull exactly equivalent images.
Docker image path: index.docker.io/ngwilson/eb-wrf:v2
Cache folder set to /ichec/home/staff/nwilson/.singularity/docker
Importing: base Singularity environment
Exploding layer:
sha256:a02a4930cb5d36f3290eb84f4bfa30668ef2e9fe3a1fb73ec015fc58b9958b17.tar
.gz
WARNING: Warning reading tar header: Ignoring malformed pax extended
attribute
WARNING: Warning reading tar header: Ignoring malformed pax extended
attribute
```

```
WARNING: Warning reading tar header: Ignoring malformed pax extended
attribute
Exploding layer:
sha256:cf8dde8a52caf911fa92b62c2419e04717c2ec2dd50498224109860cd09ad936.tar
.gz
Exploding layer:
sha256:0c0f9445ae5844090925379ca08b9fa2dbab56474bb624bb26ad5a1d31c63395.tar
.gz
Exploding layer:
sha256:560929b63b67b563f446d0de1f3cc8659910f16893cba2278e8330867ebb4db7.tar
.gz
Exploding layer:
sha256:d767f47e4502dfcced87d4f82afcbd2c95daee8b9f4ce6a4fd812d92bab718d9.tar
.gz
Exploding layer:
sha256:493ec8f41bf3b824c6e0cd327c97c91bfb11cda06e573197d44b06544b9d450b.tar
.gz
Exploding layer:
sha256:9569bfbee9e65a41837d7bc31dd677e3cbaf3d66b49b59eb6a6ceb6b8a45a7c8.tar
.gz
WARNING: Building container as an unprivileged user. If you run this
container as root
WARNING: it may be missing some functionality.
Building Singularity image...
Singularity container built: ./eb-wrf-v2.simg
Cleaning up...
Done. Container is at: ./eb-wrf-v2.simg
```

Now we can test some basic functionality and look inside the singularity image by starting a container and executing an interactive shell within it:

```
$ cat /etc/redhat-release
CentOS Linux release 7.4.1708 (Core)

$ singularity shell eb-wrf-v2.simg
Singularity: Invoking an interactive shell within container...

Singularity eb-wrf-v2.simg:~/work/singularity_images> cat /etc/redhat-
release
CentOS Linux release 7.6.1810 (Core)

Singularity eb-wrf-v2.simg:~/work/singularity_images> ls /opt/apps/WRF/
```

```
arch  clean    configure      doc     dyn_nmm   frame  inc    Makefile
README        run  test   var  chem  compile  configure.wrf  dyn_em
external    hydro  main  phys       Registry  share    tools  wrftladj
```

The next step is to actually run our test application. The typical way in which applications are executed on HPC systems is through a batch scheduling workload manager. For a description of the usage of containers with workload managers and the potential performance implications, please refer to Chapter 7 of D12.3 - Using containers technologies to improve portability of applications in HPC. Here we will execute our application container via a Slurm batch script. For this example we will use a simple 3D fire model test case which comes with the WRF source distribution under `test/em_fire`. We will copy all necessary input files into a dedicated folder:

```
$ ls -l
total 324
-rw-r--r--  1 nwilson ichec 206689 Apr  3 19:32 fuels.m
-rw-r--r--. 1 nwilson ichec   1799 Apr  3 17:22 input_sounding
-rw-r--r--. 1 nwilson ichec   2559 Apr  3 17:32 namelist.fire
-rw-r--r--. 1 nwilson ichec   9615 Apr  3 17:22 namelist.input
-rw-r--r--  1 nwilson ichec  91199 Apr  3 19:32 namelist.output
-rwxr--r--. 1 nwilson ichec    153 Apr  3 18:14 run_wrf.sh*
-rw-r--r--. 1 nwilson ichec    194 Apr  3 19:32 submit.sh
```

Apart from the WRF input files we also have the Slurm submit script `submit.sh` and a script which is subsequently executed by singularity within a container - `run_wrf.sh`

```
$ cat run_wrf.sh
#!/bin/bash
source /etc/profile.d/profile.sh
module load  netCDF-Fortran/4.4.4-foss-2018b
/opt/apps/WRF/main/ideal.exe
mpirun /opt/apps/WRF/main/wrf.exe
```

This script runs within the container and simply loads the appropriate environment modules then executes the WRF preprocessing step `ideal.exe` before launching the main MPI application. Finally to run the full workflow we just submit a Slurm script which calls singularity to exeute this `run_wrf.sh` script. Recall that Singularity bind mounts the users home directory by default and so all files on the host system will also be accessible to singularity.

```
$ cat submit.sh
#!/bin/sh
#SBATCH -N 1
```

```
#SBATCH -t 1:00:00
#SBATCH -A sys_test
#SBATCH -p ProdQ
module load singularity
time singularity exec /ichec/work/staff/nwilson/singularity_images/eb-
wrf.img ./run_wrf.sh
```

We then submit the job via:

```
$ sbatch submit.sh
Submitted batch job 72075
```

This results in singularity running a container on a 40 core backend compute node and using all 40 cores to run an MPI build of WRF successfully. We can see once the job is finished that the individual outputs from each MPI rank is present along with the WRF output files:

```
$ tail rsl.out.0026
FIRE:Time    3599.500 s Latent heat output     0.401E+08 W
FIRE:Time    3599.500 s Max latent heat flux   0.424E+04 W/m^2
FIRE:Time    3600.000 s Average wind           0.999E+00 m/s
FIRE:Time    3600.000 s Maximum wind           0.113E+01 m/s
FIRE:Time    3600.000 s Fire area              0.584E+06 m^2
FIRE:Time    3600.000 s Heat output            0.440E+09 W
FIRE:Time    3600.000 s Max heat flux          0.685E+05 W/m^2
FIRE:Time    3600.000 s Latent heat output     0.404E+08 W
FIRE:Time    3600.000 s Max latent heat flux   0.629E+04 W/m^2
d01 0001-01-01_01:00:00 wrf: SUCCESS COMPLETE WRF
```

6 References

1. Docker Documentation. *Docker Documentation* (2019). Available at: https://docs.docker.com/. (Accessed: 6th February 2019)
2. Get started with Docker Desktop for Mac. *Docker Documentation* (2019). Available at: https://docs.docker.com/docker-for-mac/. (Accessed: 6th February 2019)
3. Get started with Docker for Windows. *Docker Documentation* (2019). Available at: https://docs.docker.com/docker-for-windows/. (Accessed: 6th February 2019)
4. Sarai, A. oss-sec: CVE-2019-5736: runc container breakout (all versions). Available at: https://seclists.org/oss-sec/2019/q1/119. (Accessed: 22nd March 2019)
5. Singularity Documentation - Sylabs.io. *Sylabs.io* Available at: https://www.sylabs.io/docs/. (Accessed: 6th February 2019)
6. Singularity Desktop for macOS (Alpha Preview) Support Page - Sylabs.io. *Sylabs.io* Available at: https://www.sylabs.io/singularity-desktop-macos/. (Accessed: 14th March 2019)
7. hpc. hpc/charliecloud. *GitHub* Available at: https://github.com/hpc/charliecloud. (Accessed: 4th March 2019)
8. Overview — Charliecloud 0.9.8 documentation. Available at: https://hpc.github.io/charliecloud. (Accessed: 15th March 2019)
9. NERSC. NERSC/shifter. *GitHub* Available at: https://github.com/NERSC/shifter. (Accessed: 4th March 2019)

10. Weather Research and Forecasting Model | MMM: Mesoscale & Microscale Meteorology Laboratory. Available at: https://www.mmm.ucar.edu/weather-research-and-forecasting-model. (Accessed: 5th February 2019)
11. EasyBuild documentation — EasyBuild v3.8.1 documentation (release 20190129.0). Available at: https://easybuild.readthedocs.io/en/latest/. (Accessed: 5th February 2019)