Singularity Container Documentation

Release 3.6

User Docs

Oct 13, 2020

CONTENTS

1 Getting Started & Background Information **3**
- 1.1 Introduction to Singularity . . . 3
- 1.2 Quick Start . . . 5
- 1.3 Security in Singularity . . . 15

2 Building Containers **19**
- 2.1 Build a Container . . . 19
- 2.2 Definition Files . . . 23
- 2.3 Build Environment . . . 33
- 2.4 Support for Docker and OCI . . . 37
- 2.5 Fakeroot feature . . . 78

3 Signing & Encryption **81**
- 3.1 Signing and Verifying Containers . . . 81
- 3.2 Key commands . . . 86
- 3.3 Encrypted Containers . . . 88

4 Sharing & Online Services **93**
- 4.1 Remote Endpoints . . . 93
- 4.2 Cloud Library . . . 96

5 Advanced Usage **101**
- 5.1 Bind Paths and Mounts . . . 101
- 5.2 Persistent Overlays . . . 104
- 5.3 Running Services . . . 107
- 5.4 Environment and Metadata . . . 118
- 5.5 OCI Runtime Support . . . 128
- 5.6 Plugins . . . 143
- 5.7 Security Options . . . 144
- 5.8 Network virtualization . . . 148
- 5.9 Limiting container resources with cgroups . . . 151
- 5.10 Singularity and MPI applications . . . 154
- 5.11 GPU Support (NVIDIA CUDA & AMD ROCm) . . . 160

6 Get Involved **165**
- 6.1 Contributing . . . 165

7 Reference **169**
- 7.1 Appendix . . . 169
- 7.2 Command Line Interface . . . 183
- 7.3 License . . . 251

Welcome to the Singularity User Guide!

This guide aims to give an introduction to Singularity, brief installation instructions, and cover topics relevant to users building and running containers.

For a detailed guide to installation and configuration, please see the separate Admin Guide for this version of Singularity at https://sylabs.io/guides/3.6/admin-guide/.

CHAPTER

ONE

GETTING STARTED & BACKGROUND INFORMATION

1.1 Introduction to Singularity

Singularity is a *container* platform. It allows you to create and run containers that package up pieces of software in a way that is portable and reproducible. You can build a container using Singularity on your laptop, and then run it on many of the largest HPC clusters in the world, local university or company clusters, a single server, in the cloud, or on a workstation down the hall. Your container is a single file, and you don't have to worry about how to install all the software you need on each different operating system.

1.1.1 Why use Singularity?

Singularity was created to run complex applications on HPC clusters in a simple, portable, and reproducible way. First developed at Lawrence Berkeley National Laboratory, it quickly became popular at other HPC sites, academic sites, and beyond. Singularity is an open-source project, with a friendly community of developers and users. The user base continues to expand, with Singularity now used across industry and academia in many areas of work.

Many container platforms are available, but Singularity is focused on:

- Verifiable reproducibility and security, using cryptographic signatures, an immutable container image format, and in-memory decryption.
- Integration over isolation by default. Easily make use of GPUs, high speed networks, parallel filesystems on a cluster or server by default.
- Mobility of compute. The single file SIF container format is easy to transport and share.
- A simple, effective security model. You are the same user inside a container as outside, and cannot gain additional privilege on the host system by default. Read more about *Security in Singularity*.

1.1.2 Why use containers?

A Unix operating system is broken into two primary components, the kernel space, and the user space. The Kernel talks to the hardware, and provides core system features. The user space is the environment that most people are most familiar with. It is where applications, libraries and system services run.

Traditionally you use an operating system that has a fixed combination of kernel and user space. If you have access to a machine running CentOS then you cannot install software that was packaged for Ubuntu on it, because the user space of these distributions is not compatible. It can also be very difficult to install multiple versions of the same software, which might be needed to support reproducibility in different workflows over time.

Containers change the user space into a swappable component. This means that the entire user space portion of a Linux operating system, including programs, custom configurations, and environment can be independent of whether your system is running CentOS, Fedora etc., underneath. A Singularity container packages up whatever you need into a single, verifiable file.

Software developers can now build their stack onto whatever operating system base fits their needs best, and create distributable runtime environments so that users never have to worry about dependencies and requirements, that they might not be able to satisfy on their systems.

1.1.3 Use Cases

BYOE: Bring Your Own Environment!

Engineering work-flows for research computing can be a complicated and iterative process, and even more so on a shared and somewhat inflexible production environment. Singularity solves this problem by making the environment flexible.

Additionally, it is common (especially in education) for schools to provide a standardized pre-configured Linux distribution to the students which includes all of the necessary tools, programs, and configurations so they can immediately follow along.

Reproducible science

Singularity containers can be built to include all of the programs, libraries, data and scripts such that an entire demonstration can be contained and either archived or distributed for others to replicate no matter what version of Linux they are presently running.

Commercially supported code requiring a particular environment

Some commercial applications are only certified to run on particular versions of Linux. If that application was installed into a Singularity container running the version of Linux that it is certified for, that container could run on any Linux host. The application environment, libraries, and certified stack would all continue to run exactly as it is intended.

Additionally, Singularity blurs the line between container and host such that your home directory (and other directories) exist within the container. Applications within the container have full and direct access to all files you own thus you can easily incorporate the contained commercial application into your work and process flow on the host.

Static environments (software appliances)

Fund once, update never software development model. While this is not ideal, it is a common scenario for research funding. A certain amount of money is granted for initial development, and once that has been done the interns, grad students, post-docs, or developers are reassigned to other projects. This leaves the software stack un-maintained, and even rebuilds for updated compilers or Linux distributions can not be done without unfunded effort.

Legacy code on old operating systems

Similar to the above example, while this is less than ideal it is a fact of the research ecosystem. As an example, I know of one Linux distribution which has been end of life for 15 years which is still in production due to the software stack which is custom built for this environment. Singularity has no problem running that operating system and application stack on a current operating system and hardware.

Complicated software stacks that are very host specific

There are various software packages which are so complicated that it takes much effort in order to port, update and qualify to new operating systems or compilers. The atmospheric and weather applications are a good example of this. Porting them to a contained operating system will prolong the use-fullness of the development effort considerably.

Complicated work-flows that require custom installation and/or data

Consolidating a work-flow into a Singularity container simplifies distribution and replication of scientific results. Making containers available along with published work enables other scientists to build upon (and verify) previous scientific work.

1.2 Quick Start

This guide is intended for running Singularity on a computer where you have root (administrative) privileges, and will install Singularity from source code. Other installation options, including building an RPM package and installing Singularity without root privileges are discussed in the installation section of the admin guide.

If you need to request an installation on your shared resource, see the *requesting an installation section* for information to send to your system administrator.

For any additional help or support contact the Sylabs team: https://www.sylabs.io/contact/

1.2.1 Quick Installation Steps

You will need a Linux system to run Singularity natively. Options for using Singularity on Mac and Windows machines, along with alternate Linux installation options are discussed in the installation section of the admin guide.

Install system dependencies

You must first install development libraries to your host. Assuming Ubuntu (apply similar to RHEL derivatives):

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

Note: Note that `squashfs-tools` is only a dependency for commands that build images. The `build` command obviously relies on `squashfs-tools`, but other commands may do so as well if they are ran using container images from Docker Hub for instance.

There are 3 broad steps to installing Singularity:

1. *Installing Go*
2. *Downloading Singularity*
3. *Compiling Singularity Source Code*

Install Go

Singularity v3 and above is written primarily in Go, so you will need Go installed to compile it from source.

This is one of several ways to install and configure Go.

Note: If you have previously installed Go from a download, rather than an operating system package, you should remove your `go` directory, e.g. `rm -r /usr/local/go` before installing a newer version. Extracting a new version of Go over an existing installation can lead to errors when building Go programs, as it may leave old files, which have been removed or replaced in newer versions.

Visit the Go Downloads page and pick a package archive suitable to the environment you are in. Once the Download is complete, extract the archive to `/usr/local` (or use other instructions on go installation page). Alternatively, follow the commands here:

```
$ export VERSION=1.13 OS=linux ARCH=amd64 && \  # Replace the values as needed
  wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz && \ # Downloads the␣
↪required Go package
  sudo tar -C /usr/local -xzvf go$VERSION.$OS-$ARCH.tar.gz && \ # Extracts the archive
  rm go$VERSION.$OS-$ARCH.tar.gz    # Deletes the ``tar`` file
```

Set the Environment variable `PATH` to point to Go:

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

Download Singularity from a release

You can download Singularity from one of the releases. To see a full list, visit the GitHub release page. After deciding on a release to install, you can run the following commands to proceed with the installation.

```
$ export VERSION=3.6.4 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/
↪singularity-${VERSION}.tar.gz && \
    tar -xzf singularity-${VERSION}.tar.gz && \
    cd singularity
```

Compile the Singularity source code

Now you are ready to build Singularity. Dependencies will be automatically downloaded. You can build Singularity using the following commands:

```
$ ./mconfig && \
    make -C builddir && \
    sudo make -C builddir install
```

Singularity must be installed as root to function properly.

1.2.2 Overview of the Singularity Interface

Singularity's *command line interface* allows you to build and interact with containers transparently. You can run programs inside a container as if they were running on your host system. You can easily redirect IO, use pipes, pass arguments, and access files, sockets, and ports on the host system from within a container.

The `help` command gives an overview of Singularity options and subcommands as follows:

```
$ singularity help

Linux container platform optimized for High Performance Computing (HPC) and
Enterprise Performance Computing (EPC)

Usage:
  singularity [global options...]

Description:
  Singularity containers provide an application virtualization layer enabling
  mobility of compute via both application and environment portability. With
  Singularity one is capable of building a root file system that runs on any
  other Linux system where Singularity is installed.

Options:
  -d, --debug     print debugging information (highest verbosity)
  -h, --help      help for singularity
      --nocolor   print without color output (default False)
  -q, --quiet     suppress normal output
  -s, --silent    only print errors
  -v, --verbose   print additional information

Available Commands:
  build       Build a Singularity image
  cache       Manage the local cache
  capability  Manage Linux capabilities for users and groups
  exec        Run a command within a container
  help        Help about any command
  inspect     Show metadata for an image
  instance    Manage containers running as services
  key         Manage OpenPGP keys
  oci         Manage OCI containers
  plugin      Manage singularity plugins
  pull        Pull an image from a URI
  push        Upload image to the provided library (default is "cloud.sylabs.io")
  remote      Manage singularity remote endpoints
  run         Run the user-defined default command within a container
  run-help    Show the user-defined help for an image
```

(continues on next page)

(continued from previous page)

```
  search      Search a Container Library for images
  shell       Run a shell within a container
  sif         siftool is a program for Singularity Image Format (SIF) file␣
↪manipulation
  sign        Attach a cryptographic signature to an image
  test        Run the user-defined tests within a container
  verify      Verify cryptographic signatures attached to an image
  version     Show the version for Singularity

Examples:
  $ singularity help <command> [<subcommand>]
  $ singularity help build
  $ singularity help instance start


For additional help or support, please visit https://www.sylabs.io/docs/
```

Information about subcommand can also be viewed with the `help` command.

```
$ singularity help verify
Verify cryptographic signatures attached to an image

Usage:
  singularity verify [verify options...] <image path>

Description:
  The verify command allows a user to verify cryptographic signatures on SIF
  container files. There may be multiple signatures for data objects and
  multiple data objects signed. By default the command searches for the primary
  partition signature. If found, a list of all verification blocks applied on
  the primary partition is gathered so that data integrity (hashing) and
  signature verification is done for all those blocks.

Options:
  -a, --all                verify all objects
  -g, --group-id uint32    verify objects with the specified group ID
  -h, --help               help for verify
  -j, --json               output json
      --legacy-insecure    enable verification of (insecure) legacy signatures
  -l, --local              only verify with local keys
  -i, --sif-id uint32      verify object with the specified ID
  -u, --url string         key server URL (default "https://keys.sylabs.io")


Examples:
  $ singularity verify container.sif


For additional help or support, please visit https://www.sylabs.io/docs/
```

Singularity uses positional syntax (i.e. the order of commands and options matters). Global options affecting the behavior of all commands follow the main `singularity` command. Then sub commands are followed by their options and arguments.

For example, to pass the `--debug` option to the main `singularity` command and run Singularity with debugging messages on:

```
$ singularity --debug run library://sylabsed/examples/lolcow
```

To pass the `--containall` option to the `run` command and run a Singularity image in an isolated manner:

```
$ singularity run --containall library://sylabsed/examples/lolcow
```

Singularity 2.4 introduced the concept of command groups. For instance, to list Linux capabilities for a particular user, you would use the `list` command in the `capability` command group like so:

```
$ singularity capability list dave
```

Container authors might also write help docs specific to a container or for an internal module called an `app`. If those help docs exist for a particular container, you can view them like so.

```
$ singularity inspect --helpfile container.sif  # See the container's help, if␣
↪provided

$ singularity inspect --helpfile --app=foo foo.sif  # See the help for foo, if␣
↪provided
```

1.2.3 Download pre-built images

You can use the `search` command to locate groups, collections, and containers of interest on the Container Library .

```
$ singularity search alp
No users found for 'alp'

Found 1 collections for 'alp'
    library://jchavez/alpine

Found 5 containers for 'alp'
    library://jialipassion/official/alpine
            Tags: latest
    library://dtrudg/linux/alpine
            Tags: 3.2 3.3 3.4 3.5 3.6 3.7 3.8 edge latest
    library://sylabsed/linux/alpine
            Tags: 3.6 3.7 latest
    library://library/default/alpine
            Tags: 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 latest
    library://sylabsed/examples/alpine
            Tags: latest
```

You can use the pull and build commands to download pre-built images from an external resource like the Container Library or Docker Hub.

When called on a native Singularity image like those provided on the Container Library, `pull` simply downloads the image file to your system.

```
$ singularity pull library://sylabsed/linux/alpine
```

You can also use `pull` with the `docker://` uri to reference Docker images served from a registry. In this case `pull` does not just download an image file. Docker images are stored in layers, so `pull` must also combine those layers into a usable Singularity file.

```
$ singularity pull docker://godlovedc/lolcow
```

Pulling Docker images reduces reproducibility. If you were to pull a Docker image today and then wait six months and pull again, you are not guaranteed to get the same image. If any of the source layers has changed the image will be altered. If reproducibility is a priority for you, try building your images from the Container Library.

You can also use the `build` command to download pre-built images from an external resource. When using `build` you must specify a name for your container like so:

```
$ singularity build ubuntu.sif library://ubuntu

$ singularity build lolcow.sif docker://godlovedc/lolcow
```

Unlike `pull`, `build` will convert your image to the latest Singularity image format after downloading it. `build` is like a "Swiss Army knife" for container creation. In addition to downloading images, you can use `build` to create images from other images or from scratch using a *definition file*. You can also use `build` to convert an image between the container formats supported by Singularity. To see a comparison of Singularity definition file with Dockerfile, please see: *this section*.

1.2.4 Interact with images

You can interact with images in several ways, each of which can accept image URIs in addition to a local image path.

For demonstration, we will use a `lolcow_latest.sif` image that can be pulled from the Container Library:

```
$ singularity pull library://sylabsed/examples/lolcow
```

Shell

The shell command allows you to spawn a new shell within your container and interact with it as though it were a small virtual machine.

```
$ singularity shell lolcow_latest.sif

Singularity lolcow_latest.sif:~>
```

The change in prompt indicates that you have entered the container (though you should not rely on that to determine whether you are in container or not).

Once inside of a Singularity container, you are the same user as you are on the host system.

```
Singularity lolcow_latest.sif:~> whoami
david

Singularity lolcow_latest.sif:~> id
uid=1000(david) gid=1000(david) groups=1000(david),4(adm),24(cdrom),27(sudo),30(dip),
↪46(plugdev),116(lpadmin),126(sambashare)
```

`shell` also works with the `library://`, `docker://`, and `shub://` URIs. This creates an ephemeral container that disappears when the shell is exited.

```
$ singularity shell library://sylabsed/examples/lolcow
```

Executing Commands

The exec command allows you to execute a custom command within a container by specifying the image file. For instance, to execute the `cowsay` program within the `lolcow_latest.sif` container:

```
$ singularity exec lolcow_latest.sif cowsay moo
 _____
< moo >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

`exec` also works with the `library://`, `docker://`, and `shub://` URIs. This creates an ephemeral container that executes a command and disappears.

```
$ singularity exec library://sylabsed/examples/lolcow cowsay "Fresh from the library!"
 _________________________
< Fresh from the library! >
 -------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Running a container

Singularity containers contain *runscripts*. These are user defined scripts that define the actions a container should perform when someone runs it. The runscript can be triggered with the run command, or simply by calling the container as though it were an executable.

```
$ singularity run lolcow_latest.sif
 _____________________________________
/ You have been selected for a secret \
\ mission.                            /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

$ ./lolcow_latest.sif
 ____________________________________
/ Q: What is orange and goes "click, \
\ click?" A: A ball point carrot.    /
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

`run` also works with the `library://`, `docker://`, and `shub://` URIs. This creates an ephemeral container that runs and then disappears.

```
$ singularity run library://sylabsed/examples/lolcow
 ____________________________________
/ Is that really YOU that is reading \
\ this?                              /
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

1.2.5 Working with Files

Files on the host are reachable from within the container.

```
$ echo "Hello from inside the container" > $HOME/hostfile.txt

$ singularity exec lolcow_latest.sif cat $HOME/hostfile.txt

Hello from inside the container
```

This example works because `hostfile.txt` exists in the user's home directory. By default Singularity bind mounts `/home/$USER`, `/tmp`, and `$PWD` into your container at runtime.

You can specify additional directories to bind mount into your container with the `--bind` option. In this example, the `data` directory on the host system is bind mounted to the `/mnt` directory inside the container.

```
$ echo "Drink milk (and never eat hamburgers)." > /data/cow_advice.txt

$ singularity exec --bind /data:/mnt lolcow_latest.sif cat /mnt/cow_advice.txt
Drink milk (and never eat hamburgers).
```

Pipes and redirects also work with Singularity commands just like they do with normal Linux commands.

```
$ cat /data/cow_advice.txt | singularity exec lolcow_latest.sif cowsay
 ________________________________________
< Drink milk (and never eat hamburgers). >
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

1.2.6 Build images from scratch

Singularity v3.0 and above produces immutable images in the Singularity Image File (SIF) format. This ensures reproducible and verifiable images and allows for many extra benefits such as the ability to sign and verify your containers.

However, during testing and debugging you may want an image format that is writable. This way you can `shell` into the image and install software and dependencies until you are satisfied that your container will fulfill your needs. For these scenarios, Singularity also supports the `sandbox` format (which is really just a directory).

Sandbox Directories

To build into a `sandbox` (container in a directory) use the `build --sandbox` command and option:

```
$ sudo singularity build --sandbox ubuntu/ library://ubuntu
```

This command creates a directory called `ubuntu/` with an entire Ubuntu Operating System and some Singularity metadata in your current working directory.

You can use commands like `shell`, `exec` , and `run` with this directory just as you would with a Singularity image. If you pass the `--writable` option when you use your container you can also write files within the sandbox directory (provided you have the permissions to do so).

```
$ sudo singularity exec --writable ubuntu touch /foo

$ singularity exec ubuntu/ ls /foo
/foo
```

Converting images from one format to another

The `build` command allows you to build a container from an existing container. This means that you can use it to convert a container from one format to another. For instance, if you have already created a sandbox (directory) and want to convert it to the default immutable image format (squashfs) you can do so:

```
$ singularity build new-sif sandbox
```

Doing so may break reproducibility if you have altered your sandbox outside of the context of a definition file, so you are advised to exercise care.

Singularity Definition Files

For a reproducible, verifiable and production-quality container you should build a SIF file using a Singularity definition file. This also makes it easy to add files, environment variables, and install custom software, and still start from your base of choice (e.g., the Container Library).

A definition file has a header and a body. The header determines the base container to begin with, and the body is further divided into sections that perform things like software installation, environment setup, and copying files into the container from host system, etc.

Here is an example of a definition file:

```
BootStrap: library
From: ubuntu:16.04

```

(continues on next page)

(continued from previous page)

```
%post
    apt-get -y update
    apt-get -y install fortune cowsay lolcat

%environment
    export LC_ALL=C
    export PATH=/usr/games:$PATH

%runscript
    fortune | cowsay | lolcat

%labels
    Author GodloveD
```

To build a container from this definition file (assuming it is a file named lolcow.def), you would call build like so:

```
$ sudo singularity build lolcow.sif lolcow.def
```

In this example, the header tells Singularity to use a base Ubuntu 16.04 image from the Container Library.

- The `%post` section executes within the container at build time after the base OS has been installed. The `%post` section is therefore the place to perform installations of new applications.
- The `%environment` section defines some environment variables that will be available to the container at runtime.
- The `%runscript` section defines actions for the container to take when it is executed.
- And finally, the `%labels` section allows for custom metadata to be added to the container.

This is a very small example of the things that you can do with a *definition file*. In addition to building a container from the Container Library, you can start with base images from Docker Hub and use images directly from official repositories such as Ubuntu, Debian, CentOS, Arch, and BusyBox. You can also use an existing container on your host system as a base.

If you want to build Singularity images but you don't have administrative (root) access on your build system, you can build images using the Remote Builder.

This quickstart document just scratches the surface of all of the things you can do with Singularity!

If you need additional help or support, contact the Sylabs team: https://www.sylabs.io/contact/

Singularity on a shared resource

Perhaps you are a user who wants a few talking points and background to share with your administrator. Or maybe you are an administrator who needs to decide whether to install Singularity.

This document, and the accompanying administrator documentation provides answers to many common questions.

If you need to request an installation you may decide to draft a message similar to this:

```
Dear shared resource administrator,

We are interested in having Singularity (https://www.sylabs.io/docs/)
installed on our shared resource. Singularity containers will allow us to
build encapsulated environments, meaning that our work is reproducible and
we are empowered to choose all dependencies including libraries, operating
system, and custom software. Singularity is already in use on many of the
```

(continues on next page)

(continued from previous page)

```
top HPC centers around the world. Examples include:

    Texas Advanced Computing Center
    GSI Helmholtz Center for Heavy Ion Research
    Oak Ridge Leadership Computing Facility
    Purdue University
    National Institutes of Health HPC
    UFIT Research Computing at the University of Florida
    San Diego Supercomputing Center
    Lawrence Berkeley National Laboratory
    University of Chicago
    McGill HPC Centre/Calcul Québec
    Barcelona Supercomputing Center
    Sandia National Lab
    Argonne National Lab

Importantly, it has a vibrant team of developers, scientists, and HPC
administrators that invest heavily in the security and development of the
software, and are quick to respond to the needs of the community. To help
learn more about Singularity, I thought these items might be of interest:

    - Security: A discussion of security concerns is discussed at
    https://www.sylabs.io/guides/{adminversion}/admin-guide/admin_quickstart.html

    - Installation:
    https://www.sylabs.io/guides/{adminversion}/admin-guide/installation.html

If you have questions about any of the above, you can email the open source
list (singularity@lbl.gov), join the open source slack channel
(singularity-container.slack.com), or contact the organization that supports
Singularity directly (sylabs.io/contact). I can do
my best to facilitate this interaction if help is needed.

Thank you kindly for considering this request!

Best,

User
```

1.3 Security in Singularity

Containers are popular for many good reasons. They are light weight, easy to spin-up and require reduced IT management resources as compared to hardware VM environments. More importantly, container technology facilitates advanced research computing by granting the ability to package software in highly portable and reproducible environments encapsulating all dependencies, including the operating system. But there are still some challenges to container security.

Singularity addresses some core missions of containers : Mobility of Compute, Reproducibility, HPC support, and **Security**. This section gives an overview of security features supported by Singularity, especially where they differ from other container runtimes.

1.3.1 Security Policy

Security is not a check box that one can tick and forget. Ensuring security is a ongoing process that begins with software architecture, and continues all the way through to ongoing security practices. In addition to ensuring that containers are run without elevated privileges where appropriate, and that containers are produced by trusted sources, users must monitor their containers for newly discovered vulnerabilities and update when necessary just as they would with any other software. Sylabs is constantly probing to find and patch vulnerabilities within Singularity, and will continue to do so.

If you suspect you have found a vulnerability in Singularity, please follow the steps in our published Security Policy.

so that it can be disclosed, investigated, and fixed in an appropriate manner.

1.3.2 Singularity PRO - Long Term Support & Security Patches

Security patches for Singularity are applied to the latest open-source version, so it is important to follow new releases and upgrade when neccessary.

SingularityPRO is a professionally curated and licensed version of Singularity that provides added security, stability, and support beyond that offered by the open source project. Security and bug-fix patches are backported to select versions of Singularity PRO, so that they can be deployed long-term where required. PRO users receive security fixes (without specific notification or detail) prior to public disclosure, as detailed in the Sylabs Security Policy.

1.3.3 Singularity Runtime & User Privilege

The Singularity Runtime enforces a unique security model that makes it appropriate for *untrusted users* to run *untrusted containers* safely on multi-tenant resources. When you run a container, the processes in the container will run as your user account. Singularity dynamically writes UID and GID information to the appropriate files within the container, and the user remains the same *inside* and *outside* the container, i.e., if you're an unprivileged user while entering the container you'll remain an unprivileged user inside the container.

Additional blocks are in place to prevent users from escalating privileges once they are inside of a container. The container file system is mounted using the `nosuid` option, and processes are started with the `PR_NO_NEW_PRIVS` flag set. This means that even if you run *sudo* inside your container, you won't be able to change to another user, or gain root priveleges by other means. This approach provides a secure way for users to run containers and greatly simplifies things like reading and writing data to the host system with appropriate ownership.

It is also important to note that the philosophy of Singularity is *Integration* over *Isolation*. Most container run times strive to isolate your container from the host system and other containers as much as possible. Singularity, on the other hand, assumes that the user's primary goals are portability, reproducibility, and ease of use and that isolation is often a tertiary concern. Therefore, Singularity only isolates the mount namespace by default, and will bind mount several host directories such as `$HOME` and `/tmp` into the container at runtime. If needed, additional levels of isolation can be achieved by passing options causing Singularity to enter any or all of the other kernel namespaces and to prevent automatic bind mounting. These measures allow users to interact with the host system from within the container in sensible ways.

1.3.4 Singularity Image Format (SIF)

Ensuring container security as a continuous process. Singularity provides ways to ensure integrity throughout the lifecyle of a container, i.e. at rest, in transit and while running. The SIF Singularity Image Format has been designed to achieve these goals.

A SIF file is an immutable container image that packages the container environment into a single file. SIF supports security and integrity through the ability to cryptographically sign a container, creating a signature block within the SIF file which can guarantee immutability and provide accountability as to who signed it. Singularity follows the OpenPGP standard to create and manage these signatures, and the keys used to create them. After building an image with Singularity, a user can `singularity sign` the container and push it to the Library along with its public PGP key (stored in *Keystore*). The signature can be verified (`singularity verify`) while pulling or downloading the image. *This feature* makes it easy to to establish trust in collaborations within and between teams.

In Singularity 3.4 and above, the root file system of a container (stored in the squashFS partition of SIF) can be encrypted. As a result, everything inside the container becomes inaccessible without the correct key or passphrase. Other users on the system will be able to look inside your container files. The content of the container is private, even if the SIF file is shared in public.

Unlike other container platforms where execution requires a number of layers to be extracted to a rootfs directory on the host, Singularity executes containers in a single step, directly from the immutable `.sif`. This reduces the attack surface and allows the container to be easily verified at runtime, to ensure it has not been tampered with.

1.3.5 Admin Configurable Files

System administrators who manage Singularity can use configuration files, to set security restrictions, grant or revoke a user's capabilities, manage resources and authorize containers etc.

For example, the ecl.toml file allows blacklisting and whitelisting of containers.

Configuration files and their parameters are documented for administrators documented here.

cgroups support

Starting with v3.0, Singularity added native support for `cgroups`, allowing users to limit the resources their containers consume without the help of a separate program like a batch scheduling system. This feature can help to prevent DoS attacks where one container seizes control of all available system resources in order to stop other containers from operating properly. To use this feature, a user first creates a cgroups configuration file. An example configuration file is installed by default with Singularity as a guide. At runtime, the `--apply-cgroups` option is used to specify the location of the configuration file to apply to the container and cgroups are configured accordingly. More about cgroups support here.

`--security` options

Singularity supports a number of methods for further modifying the security scope and context when running Singularity containers. Flags can be passed to the action commands; `shell`, `exec`, and `run` allowing fine grained control of security. Details about them are documented here.

1.3.6 Security in the Sylabs Cloud

Sylabs Cloud consists of a Remote Builder, a Container Library, and a Keystore. Together, theses services provide an end-to-end solution for packaging and distributing applications in secure and trusted containers.

Remote Builder

As mentioned earlier, the Singularity runtime prevents executing code with root-level permissions on the host system. However, building a container requires elevated privileges that most shared environments do not grant their users. The Build Service aims to address this by allowing unprivileged users to build containers remotely, with root level permissions inside the secured service. System administrators can use the system to monitor which users are building containers, and the contents of those containers. The Singularity CLI has native integration with the Build Service from version 3.0 onwards. In addition, a browser interface to the Build Service also exists, which allows users to build containers using only a web browser.

Note: Please also see the *Fakeroot feature* which is a secure option for admins in multi-tenant HPC environments and similar use cases where they might want to grant a user special privileges inside a container.

Fakeroot has some limitations, and requires unpriveleged user namespace support in the host kernel.

Container Library

The Container Library allows users to store and share Singularity container images in the Singularity Image Format (SIF). A web front-end allows users to create new projects within the Container Library, edit documentation associated with container images, and discover container images published by their peers.

Key Store

The Key Store is a key management system offered by Sylabs that uses an OpenPGP implementation to permit sharing and discovery of PGP public keys used to sign and verify Singularity container images. This service is based on the OpenPGP HTTP Keyserver Protocol (HKP), with several enhancements:

- The Service requires connections to be secured with Transport Layer Security (TLS).
- The Service implements token-based authentication, allowing only authenticated users to add or modify PGP keys.
- A web front-end allows users to view and search for PGP keys using a web browser.

Authentication and encryption

1. Communication between users, the authentication service other services is secured via TLS encryption.
2. The services support authentication of users via signed and encrypted authentication tokens.
3. There is no implicit trust relationship between each service. Each request between the services is authenticated using the authentication token supplied by the user in the associated request.

CHAPTER

TWO

BUILDING CONTAINERS

2.1 Build a Container

`build` is the "Swiss army knife" of container creation. You can use it to download and assemble existing containers from external resources like the Container Library and Docker Hub. You can use it to convert containers between the formats supported by Singularity. And you can use it in conjunction with a Singularity definition file to create a container from scratch and customized it to fit your needs.

2.1.1 Overview

The `build` command accepts a target as input and produces a container as output.

The target defines the method that `build` uses to create the container. It can be one of the following:

- URI beginning with **library://** to build from the Container Library
- URI beginning with **docker://** to build from Docker Hub
- URI beginning with **shub://** to build from Singularity Hub
- path to a **existing container** on your local machine
- path to a **directory** to build from a sandbox
- path to a Singularity definition file

`build` can produce containers in two different formats that can be specified as follows.

- compressed read-only **Singularity Image File (SIF)** format suitable for production (default)
- writable **(ch)root directory** called a sandbox for interactive development (`--sandbox` option)

Because `build` can accept an existing container as a target and create a container in either supported format you can convert existing containers from one format to another.

2.1.2 Downloading an existing container from the Container Library

You can use the build command to download a container from the Container Library.

```
$ sudo singularity build lolcow.sif library://sylabs-jms/testing/lolcow
```

The first argument (`lolcow.sif`) specifies a path and name for your container. The second argument (`library://sylabs-jms/testing/lolcow`) gives the Container Library URI from which to download. By default the container will be converted to a compressed, read-only SIF. If you want your container in a writable format use the `--sandbox` option.

2.1.3 Downloading an existing container from Docker Hub

You can use `build` to download layers from Docker Hub and assemble them into Singularity containers.

```
$ sudo singularity build lolcow.sif docker://godlovedc/lolcow
```

2.1.4 Creating writable `--sandbox` directories

If you wanted to create a container within a writable directory (called a sandbox) you can do so with the `--sandbox` option. It's possible to create a sandbox without root privileges, but to ensure proper file permissions it is recommended to do so as root.

```
$ sudo singularity build --sandbox lolcow/ library://sylabs-jms/testing/lolcow
```

The resulting directory operates just like a container in a SIF file. To make changes within the container, use the `--writable` flag when you invoke your container. It's a good idea to do this as root to ensure you have permission to access the files and directories that you want to change.

```
$ sudo singularity shell --writable lolcow/
```

2.1.5 Converting containers from one format to another

If you already have a container saved locally, you can use it as a target to build a new container. This allows you convert containers from one format to another. For example if you had a sandbox container called `development/` and you wanted to convert it to SIF container called `production.sif` you could:

```
$ sudo singularity build production.sif development/
```

Use care when converting a sandbox directory to the default SIF format. If changes were made to the writable container before conversion, there is no record of those changes in the Singularity definition file rendering your container non-reproducible. It is a best practice to build your immutable production containers directly from a Singularity definition file instead.

2.1.6 Building containers from Singularity definition files

Of course, Singularity definition files can be used as the target when building a container. For detailed information on writing Singularity definition files, please see the *Container Definition docs*. Let's say you already have the following container definition file called `lolcow.def`, and you want to use it to build a SIF container.

```
Bootstrap: docker
From: ubuntu:16.04

%post
    apt-get -y update
    apt-get -y install fortune cowsay lolcat

%environment
    export LC_ALL=C
    export PATH=/usr/games:$PATH

%runscript
    fortune | cowsay | lolcat
```

You can do so with the following command.

```
$ sudo singularity build lolcow.sif lolcow.def
```

The command requires `sudo` just as installing software on your local machine requires root privileges.

Note: Beware that it is possible to build an image on a host and have the image not work on a different host. This could be because of the default compressor supported by the host. For example, when building an image on a host in which the default compressor is `xz` and then trying to run that image on a CentOS 6 node, where the only compressor available is `gzip`.

2.1.7 Building encrypted containers

Beginning in Singularity 3.4.0 it is possible to build and run encrypted containers. The containers are decrypted at runtime entirely in kernel space, meaning that no intermediate decrypted data is ever present on disk or in memory. See *encrypted containers* for more details.

2.1.8 Build options

`--builder`

Singularity 3.0 introduces the option to perform a remote build. The `--builder` option allows you to specify a URL to a different build service. For instance, you may need to specify a URL to build to an on premises installation of the remote builder. This option must be used in conjunction with `--remote`.

`--detached`

When used in combination with the `--remote` option, the `--detached` option will detach the build from your terminal and allow it to build in the background without echoing any output to your terminal.

`--encrypt`

Specifies that Singularity should use a secret saved in either the `SINGULARITY_ENCRYPTION_PASSPHRASE` or `SINGULARITY_ENCRYPTION_PEM_PATH` environment variable to build an encrypted container. See *encrypted containers* for more details.

`--fakeroot`

Gives users a way to build containers completely unprivileged. See *the fakeroot feature* for details.

`--force`

The `--force` option will delete and overwrite an existing Singularity image without presenting the normal interactive prompt.

`--json`

The `--json` option will force Singularity to interpret a given definition file as a json.

`--library`

This command allows you to set a different library. (The default library is "https://library.sylabs.io")

`--notest`

If you don't want to run the `%test` section during the container build, you can skip it with the `--notest` option. For instance, maybe you are building a container intended to run in a production environment with GPUs. But perhaps your local build resource does not have GPUs. You want to include a `%test` section that runs a short validation but you don't want your build to exit with an error because it cannot find a GPU on your system.

`--passphrase`

This flag allows you to pass a plaintext passphrase to encrypt the container file system at build time. See *encrypted containers* for more details.

`--pem-path`

This flag allows you to pass the location of a public key to encrypt the container file system at build time. See *encrypted containers* for more details.

`--remote`

Singularity 3.0 introduces the ability to build a container on an external resource running a remote builder. (The default remote builder is located at "https://cloud.sylabs.io/builder".)

`--sandbox`

Build a sandbox (chroot directory) instead of the default SIF format.

`--section`

Instead of running the entire definition file, only run a specific section or sections. This option accepts a comma delimited string of definition file sections. Acceptable arguments include `all`, `none` or any combination of the following: `setup`, `post`, `files`, `environment`, `test`, `labels`.

Under normal build conditions, the Singularity definition file is saved into a container's meta-data so that there is a record showing how the container was built. Using the `--section` option may render this meta-data useless, so use care if you value reproducibility.

`--update`

You can build into the same sandbox container multiple times (though the results may be unpredictable and it is generally better to delete your container and start from scratch).

By default if you build into an existing sandbox container, the `build` command will prompt you to decide whether or not to overwrite the container. Instead of this behavior you can use the `--update` option to build _into_ an existing container. This will cause Singularity to skip the header and build any sections that are in the definition file into the existing container.

The `--update` option is only valid when used with sandbox containers.

2.1.9 More Build topics

- If you want to **customize the cache location** (where Docker layers are downloaded on your system), specify Docker credentials, or any custom tweaks to your build environment, see *build environment*.
- If you want to make internally **modular containers**, check out the getting started guide here
- If you want to **build your containers** on the Remote Builder, (because you don't have root access on a Linux machine or want to host your container on the cloud) check out this site
- If you want to **build a container with an encrypted file system** look *here*.

2.2 Definition Files

A Singularity Definition File (or "def file" for short) is like a set of blueprints explaining how to build a custom container. It includes specifics about the base OS to build or the base container to start from, software to install, environment variables to set at runtime, files to add from the host system, and container metadata.

2.2.1 Overview

A Singularity Definition file is divided into two parts:

1. **Header**: The Header describes the core operating system to build within the container. Here you will configure the base operating system features needed within the container. You can specify, the Linux distribution, the specific version, and the packages that must be part of the core install (borrowed from the host system).
2. **Sections**: The rest of the definition is comprised of sections, (sometimes called scriptlets or blobs of data). Each section is defined by a `%` character followed by the name of the particular section. All sections are optional, and a def file may contain more than one instance of a given section. Sections that are executed at build time are executed with the `/bin/sh` interpreter and can accept `/bin/sh` options. Similarly, sections that produce scripts to be executed at runtime can accept options intended for `/bin/sh`

For more in-depth and practical examples of def files, see the Sylabs examples repository

For a comparison between Dockerfile and Singularity definition file, please see: *this section*.

2.2.2 Header

The header should be written at the top of the def file. It tells Singularity about the base operating system that it should use to build the container. It is composed of several keywords.

The only keyword that is required for every type of build is `Bootstrap`. It determines the *bootstrap agent* that will be used to create the base operating system you want to use. For example, the `library` bootstrap agent will pull a container from the Container Library as a base. Similarly, the `docker` bootstrap agent will pull docker layers from Docker Hub as a base OS to start your image.

Starting with Singularity 3.2, the `Bootstrap` keyword needs to be the first entry in the header section. This breaks compatibility with older versions that allow the parameters of the header to appear in any order.

Depending on the value assigned to `Bootstrap`, other keywords may also be valid in the header. For example, when using the `library` bootstrap agent, the `From` keyword becomes valid. Observe the following example for building a Debian container from the Container Library:

```
Bootstrap: library
From: debian:7
```

A def file that uses an official mirror to install Centos-7 might look like this:

```
Bootstrap: yum
OSVersion: 7
MirrorURL: http://mirror.centos.org/centos-%{OSVERSION}/%{OSVERSION}/os/$basearch/
Include: yum
```

Each bootstrap agent enables its own options and keywords. You can read about them and see examples in the *appendix section*:

Preferred bootstrap agents

- *library* (images hosted on the Container Library)
- *docker* (images hosted on Docker Hub)
- *shub* (images hosted on Singularity Hub)
- *oras* (images from supporting OCI registries)
- *scratch* (a flexible option for building a container from scratch)

Other bootstrap agents

- *localimage* (images saved on your machine)
- *yum* (yum based systems such as CentOS and Scientific Linux)
- *debootstrap* (apt based systems such as Debian and Ubuntu)
- *oci* (bundle compliant with OCI Image Specification)
- *oci-archive* (tar files obeying the OCI Image Layout Specification)
- *docker-daemon* (images managed by the locally running docker daemon)

- *docker-archive* (archived docker images)
- *arch* (Arch Linux)
- *busybox* (BusyBox)
- *zypper* (zypper based systems such as Suse and OpenSuse)

2.2.3 Sections

The main content of the bootstrap file is broken into sections. Different sections add different content or execute commands at different times during the build process. Note that if any command fails, the build process will halt.

Here is an example definition file that uses every available section. We will discuss each section in turn. It is not necessary to include every section (or any sections at all) within a def file. Furthermore, multiple sections of the same name can be included and will be appended to one another during the build process.

```
Bootstrap: library
From: ubuntu:18.04
Stage: build

%setup
    touch /file1
    touch ${SINGULARITY_ROOTFS}/file2

%files
    /file1
    /file1 /opt

%environment
    export LISTEN_PORT=12345
    export LC_ALL=C

%post
    apt-get update && apt-get install -y netcat
    NOW=`date`
    echo "export NOW=\"${NOW}\"" >> $SINGULARITY_ENVIRONMENT

%runscript
    echo "Container was created $NOW"
    echo "Arguments received: $*"
    exec echo "$@"

%startscript
    nc -lp $LISTEN_PORT

%test
    grep -q NAME=\"Ubuntu\" /etc/os-release
    if [ $? -eq 0 ]; then
        echo "Container base is Ubuntu as expected."
    else
        echo "Container base is not Ubuntu."
    fi

%labels
    Author d@sylabs.io
    Version v0.0.1
```

(continues on next page)

(continued from previous page)

```
%help
    This is a demo container used to illustrate a def file that uses all
    supported sections.
```

Although, the order of the sections in the def file is unimportant, they have been documented below in the order of their execution during the build process for logical understanding.

%setup

During the build process, commands in the `%setup` section are first executed on the host system outside of the container after the base OS has been installed. You can reference the container file system with the `$SINGULARITY_ROOTFS` environment variable in the `%setup` section.

Note: Be careful with the `%setup` section! This scriptlet is executed outside of the container on the host system itself, and is executed with elevated privileges. Commands in `%setup` can alter and potentially damage the host.

Consider the example from the definition file above:

```
%setup
    touch /file1
    touch ${SINGULARITY_ROOTFS}/file2
```

Here, `file1` is created at the root of the file system **on the host**. We'll use `file1` to demonstrate the usage of the `%files` section below. The `file2` is created at the root of the file system **within the container**.

In later versions of Singularity the `%files` section is provided as a safer alternative to copying files from the host system into the container during the build. Because of the potential danger involved in running the `%setup` scriptlet with elevated privileges on the host system during the build, it's use is generally discouraged.

%files

The `%files` section allows you to copy files into the container with greater safety than using the `%setup` section. Its general form is:

```
%files [from <stage>]
    <source> [<destination>]
    ...
```

Each line is a `<source>` and `<destination>` pair. The `<source>` is either:

1. A valid path on your host system
2. A valid path in a previous stage of the build

while the `<destination>` is always a path into the current container. If the `<destination>` path is omitted it will be assumed to be the same as `<source>`. To show how copying from your host system works, let's consider the example from the definition file above:

```
%files
    /file1
    /file1 /opt
```

`file1` was created in the root of the host file system during the `%setup` section (see above). The `%files` scriptlet will copy `file1` to the root of the container file system and then make a second copy of `file1` within the container in `/opt`.

Files can also be copied from other stages by providing the source location in the previous stage and the destination in the current container.

```
%files from stage_name
  /root/hello /bin/hello
```

The only difference in behavior between copying files from your host system and copying them from previous stages is that in the former case symbolic links are always followed during the copy to the container, while in the latter symbolic links are preserved.

Files in the `%files` section are always copied before the `%post` section is executed so that they are available during the build and configuration process.

%app*

In some circumstances, it may be redundant to build different containers for each app with nearly equivalent dependencies. Singularity supports installing apps within internal modules based on the concept of Standard Container Integration Format (SCI-F) All the apps are handled by Singularity at this point. More information on Apps *here*.

%post

This section is where you can download files from the internet with tools like `git` and `wget`, install new software and libraries, write configuration files, create new directories, etc.

Consider the example from the definition file above:

```
%post
    apt-get update && apt-get install -y netcat
    NOW=`date`
    echo "export NOW=\"${NOW}\"" >> $SINGULARITY_ENVIRONMENT
```

This `%post` scriptlet uses the Ubuntu package manager `apt` to update the container and install the program `netcat` (that will be used in the `%startscript` section below).

The script is also setting an environment variable at build time. Note that the value of this variable cannot be anticipated, and therefore cannot be set during the `%environment` section. For situations like this, the `$SINGULARITY_ENVIRONMENT` variable is provided. Redirecting text to this variable will cause it to be written to a file called `/.singularity.d/env/91-environment.sh` that will be sourced at runtime.

%test

The `%test` section runs at the very end of the build process to validate the container using a method of your choice. You can also execute this scriptlet through the container itself, using the `test` command.

Consider the example from the def file above:

```
%test
    grep -q NAME=\"Ubuntu\" /etc/os-release
    if [ $? -eq 0 ]; then
        echo "Container base is Ubuntu as expected."
    else
```

(continues on next page)

(continued from previous page)

```
        echo "Container base is not Ubuntu."
    fi
```

This (somewhat silly) script tests if the base OS is Ubuntu. You could also write a script to test that binaries were appropriately downloaded and built, or that software works as expected on custom hardware. If you want to build a container without running the `%test` section (for example, if the build system does not have the same hardware that will be used on the production system), you can do so with the `--notest` build option:

```
$ sudo singularity build --notest my_container.sif my_container.def
```

Running the test command on a container built with this def file yields the following:

```
$ singularity test my_container.sif
Container base is Ubuntu as expected.
```

Now, the following sections are all inserted into the container filesystem in single step:

%environment

The `%environment` section allows you to define environment variables that will be set at runtime. Note that these variables are not made available at build time by their inclusion in the `%environment` section. This means that if you need the same variables during the build process, you should also define them in your `%post` section. Specifically:

- **during build**: The `%environment` section is written to a file in the container metadata directory. This file is not sourced.
- **during runtime**: The file in the container metadata directory is sourced.

You should use the same conventions that you would use in a `.bashrc` or `.profile` file. Consider this example from the def file above:

```
%environment
    export LISTEN_PORT=12345
    export LC_ALL=C
```

The `$LISTEN_PORT` variable will be used in the `%startscript` section below. The `$LC_ALL` variable is useful for many programs (often written in Perl) that complain when no locale is set.

After building this container, you can verify that the environment variables are set appropriately at runtime with the following command:

```
$ singularity exec my_container.sif env | grep -E 'LISTEN_PORT|LC_ALL'
LISTEN_PORT=12345
LC_ALL=C
```

In the special case of variables generated at build time, you can also add environment variables to your container in the `%post` section.

At build time, the content of the `%environment` section is written to a file called `/.singularity.d/env/90-environment.sh` inside of the container. Text redirected to the `$SINGULARITY_ENVIRONMENT` variable during `%post` is added to a file called `/.singularity.d/env/91-environment.sh`.

At runtime, scripts in `/.singularity/env` are sourced in order. This means that variables in the `%post` section take precedence over those added via `%environment`.

See *Environment and Metadata* for more information about the Singularity container environment.

%startscript

Similar to the `%runscript` section, the contents of the `%startscript` section are written to a file within the container at build time. This file is executed when the `instance start` command is issued.

Consider the example from the def file above.

```
%startscript
    nc -lp $LISTEN_PORT
```

Here the netcat program is used to listen for TCP traffic on the port indicated by the `$LISTEN_PORT` variable (set in the `%environment` section above). The script can be invoked like so:

```
$ singularity instance start my_container.sif instance1
INFO:    instance started successfully

$ lsof | grep LISTEN
nc        19061                vagrant    3u     IPv4              107409      0t0     ␣
↪   TCP *:12345 (LISTEN)

$ singularity instance stop instance1
Stopping instance1 instance of /home/vagrant/my_container.sif (PID=19035)
```

%runscript

The contents of the `%runscript` section are written to a file within the container that is executed when the container image is run (either via the `singularity run` command or by executing the container directly as a command). When the container is invoked, arguments following the container name are passed to the runscript. This means that you can (and should) process arguments within your runscript.

Consider the example from the def file above:

```
%runscript
    echo "Container was created $NOW"
    echo "Arguments received: $*"
    exec echo "$@"
```

In this runscript, the time that the container was created is echoed via the `$NOW` variable (set in the `%post` section above). The options passed to the container at runtime are printed as a single string (`$*`) and then they are passed to echo via a quoted array (`$@`) which ensures that all of the arguments are properly parsed by the executed command. The `exec` preceding the final `echo` command replaces the current entry in the process table (which originally was the call to Singularity). Thus the runscript shell process ceases to exist, and only the process running within the container remains.

Running the container built using this def file will yield the following:

```
$ ./my_container.sif
Container was created Thu Dec  6 20:01:56 UTC 2018
Arguments received:

$ ./my_container.sif this that and the other
Container was created Thu Dec  6 20:01:56 UTC 2018
Arguments received: this that and the other
this that and the other
```

%labels

The `%labels` section is used to add metadata to the file `/.singularity.d/labels.json` within your container. The general format is a name-value pair.

Consider the example from the def file above:

```
%labels
    Author d@sylabs.io
    Version v0.0.1
    MyLabel Hello World
```

Note that labels are defined by key-value pairs. To define a label just add it on the labels section and after the first space character add the correspondent value to the label.

On the previous example, the first label name is `Author`` with a value of `d@sylabs.io`. The second label name is `Version` with a value of `v0.0.1`. Finally, the last label named `MyLabel` has the value of `Hello World`.

To inspect the available labels on your image you can do so by running the following command:

```
$ singularity inspect my_container.sif

{
  "Author": "d@sylabs.io",
  "Version": "v0.0.1",
  "MyLabel": "Hello World",
  "org.label-schema.build-date": "Thursday_6_December_2018_20:1:56_UTC",
  "org.label-schema.schema-version": "1.0",
  "org.label-schema.usage": "/.singularity.d/runscript.help",
  "org.label-schema.usage.singularity.deffile.bootstrap": "library",
  "org.label-schema.usage.singularity.deffile.from": "ubuntu:18.04",
  "org.label-schema.usage.singularity.runscript.help": "/.singularity.d/runscript.help
↪",
  "org.label-schema.usage.singularity.version": "3.0.1"
}
```

Some labels that are captured automatically from the build process. You can read more about labels and metadata *here*.

%help

Any text in the `%help` section is transcribed into a metadata file in the container during the build. This text can then be displayed using the `run-help` command.

Consider the example from the def file above:

```
%help
    This is a demo container used to illustrate a def file that uses all
    supported sections.
```

After building the help can be displayed like so:

```
$ singularity run-help my_container.sif
    This is a demo container used to illustrate a def file that uses all
    supported sections.
```

2.2.4 Multi-Stage Builds

Starting with Singularity v3.2 multi-stage builds are supported where one environment can be used for compilation, then the resulting binary can be copied into a final environment. This allows a slimmer final image that does not require the entire development stack.

```
Bootstrap: docker
From: golang:1.12.3-alpine3.9
Stage: devel

%post
  # prep environment
  export PATH="/go/bin:/usr/local/go/bin:$PATH"
  export HOME="/root"
  cd /root

  # insert source code, could also be copied from host with %files
  cat << EOF > hello.go
  package main
  import "fmt"

  func main() {
    fmt.Printf("Hello World!\n")
  }
EOF

  go build -o hello hello.go


# Install binary into final image
Bootstrap: library
From: alpine:3.9
Stage: final

# install binary from stage one
%files from devel
  /root/hello /bin/hello
```

The names of stages are arbitrary. Each of these sections will be executed in the same order as described for single stage build except the files from the previous stage are copied before `%setup` section of the next stage. Files can only be copied from stages declared before the current stage in the definition. E.g., the `devel` stage in the above definition cannot copy files from the `final` stage, but the `final` stage can copy files from the `devel` stage.

2.2.5 Apps

The `%app*` sections can exist alongside any of the primary sections (i.e. `%post`, `%runscript`, `%environment`, etc.). As with the other sections, the ordering of the `%app*` sections isn't important.

The following runscript demonstrates how to build 2 different apps into the same container using SCI-F modules:

```
Bootstrap: docker
From: ubuntu

%environment
    GLOBAL=variables
    AVAILABLE="to all apps"
```

(continues on next page)

(continued from previous page)

```
##############################
# foo
##############################

%apprun foo
    exec echo "RUNNING FOO"

%applabels foo
   BESTAPP FOO

%appinstall foo
   touch foo.exec

%appenv foo
    SOFTWARE=foo
    export SOFTWARE

%apphelp foo
    This is the help for foo.

%appfiles foo
   foo.txt

##############################
# bar
##############################

%apphelp bar
    This is the help for bar.

%applabels bar
   BESTAPP BAR

%appinstall bar
    touch bar.exec

%appenv bar
    SOFTWARE=bar
    export SOFTWARE
```

An `%appinstall` section is the equivalent of `%post` but for a particular app. Similarly, `%appenv` equates to the app version of `%environment` and so on.

After installing apps into modules using the `%app*` sections, the `--app` option becomes available allowing the following functions:

To run a specific app within the container:

```
% singularity run --app foo my_container.sif
RUNNING FOO
```

The same environment variable, `$SOFTWARE` is defined for both apps in the def file above. You can execute the following command to search the list of active environment variables and `grep` to determine if the variable changes depending on the app we specify:

```
$ singularity exec --app foo my_container.sif env | grep SOFTWARE
SOFTWARE=foo

$ singularity exec --app bar my_container.sif env | grep SOFTWARE
SOFTWARE=bar
```

2.2.6 Best Practices for Build Recipes

When crafting your recipe, it is best to consider the following:

1. Always install packages, programs, data, and files into operating system locations (e.g. not `/home`, `/tmp` , or any other directories that might get commonly binded on).
2. Document your container. If your runscript doesn't supply help, write a `%help` or `%apphelp` section. A good container tells the user how to interact with it.
3. If you require any special environment variables to be defined, add them to the `%environment` and `%appenv` sections of the build recipe.
4. Files should always be owned by a system account (UID less than 500).
5. Ensure that sensitive files like `/etc/passwd`, `/etc/group`, and `/etc/shadow` do not contain secrets.
6. Build production containers from a definition file instead of a sandbox that has been manually changed. This ensures greatest possibility of reproducibility and mitigates the "black box" effect.

2.3 Build Environment

2.3.1 Overview

You may wish to customize your build environment by doing things such as specifying a custom cache directory for images or sending your Docker Credentials to the registry endpoint. Here we will discuss these and other topics related to the build environment.

2.3.2 Cache Folders

Singularity will cache SIF container images generated from remote sources, and any OCI/docker layers used to create them. The cache is created at `$HOME/.singularity/cache` by default. The location of the cache can be changed by setting the `SINGULARITY_CACHEDIR` environment variable.

Note: When you run builds as root, using `sudo`, images will be cached in root's home at `/root` and not your user's home. Use the `-E` option to sudo to pass through a `SINGULARITY_CACHEDIR` environment variable.

If you change the value of `SINGULARITY_CACHEDIR` be sure to choose a location that is:

- Unique to you. Permissions are set on the cache so that private images cached for one user are not exposed to another. This means that `SINGULARITY_CACHEDIR` cannot be shared.
- Located on a filesystem with sufficient space for the number and size of container images anticipated.
- Located on a filesystem that supports atomic rename, if possible.

Warning: If you are not certain that your `$HOME` or `SINGULARITY_CACHEDIR` filesytems support atomic rename, do not run Singularity in parallel using remote container URLs. Instead use `singularity pull` to create a local SIF image, and then run this SIF image in a parallel step. An alternative is to use the `--disable-cache` option, but this will result in each Singularity instance independently fetching the container from the remote source, into a temporary location.

Inside the cache location you will find separate directories for the different kinds of data that are cached:

```
$HOME/.singularity/cache/blob
$HOME/.singularity/cache/library
$HOME/.singularity/cache/net
$HOME/.singularity/cache/oci-tmp
$HOME/.singularity/cache/shub
```

You can safely delete these directories, or content within them. Singularity will re-create any directories and data that are needed in future runs.

You should not add any additional files, or modify files in the cache, as this may cause checksum / integrity errors when you run or build containers. If you experience problems use `singularity cache clean` to reset the cache to a clean, empty state.

2.3.3 Cache commands

The `cache` command for Singularity allows you to view and clean up your cache, without manually inspecting the cache directories.

Note: If you have built images as root, directly or via `sudo`, the cache location for those builds is `/root/.singularity`. You will need to use `sudo` when running `cache clean` or `cache list` to manage these cache entries.

Listing Cache

To view a summary of cache usage, use `singularity cache list`:

```
$ singularity cache list
There are 4 container file(s) using 59.45 MB and 23 oci blob file(s) using 379.10 MB␣
↪of space
Total space used: 438.55 MB
```

To view detailed information, use `singularity cache list -v`:

```
$ singularity cache list -v
NAME                     DATE CREATED           SIZE             TYPE
0ed5a98249068fe0592edb   2020-05-27 12:57:22    192.21 MB        blob
1d9cd1b99a7eca56d8f2be   2020-05-28 15:19:07    0.35 kB          blob
219c332183ec3800bdfda4   2020-05-28 12:22:13    0.35 kB          blob
2adae3950d4d0f11875568   2020-05-27 12:57:16    51.83 MB         blob
376057ac6fa17f65688c56   2020-05-27 12:57:12    50.39 MB         blob
496548a8c952b37bdf149a   2020-05-27 12:57:14    10.00 MB         blob
5a63a0a859d859478f3046   2020-05-27 12:57:13    7.81 MB          blob
5efaeecfa72afde779c946   2020-05-27 12:57:25    0.23 kB          blob
```

(continues on next page)

(continued from previous page)

```
6154df8ff9882934dc5bf2   2020-05-27 08:37:22    0.85 kB          blob
70d0b3967cd8abe96c9719   2020-05-27 12:57:24    26.61 MB         blob
8f5af4048c33630473b396   2020-05-28 15:19:07    0.57 kB          blob
95c3f3755f37380edb2f8f   2020-05-28 14:07:20    2.48 kB          blob
96878229af8adf91bcbf11   2020-05-28 14:07:20    0.81 kB          blob
af88fdb253aac46693de78   2020-05-28 12:22:13    0.58 kB          blob
bb94ffe723890b4d62d742   2020-05-27 12:57:23    6.15 MB          blob
c080bf936f6a1fdd2045e3   2020-05-27 12:57:25    1.61 kB          blob
cbdbe7a5bc2a134ca8ec91   2020-05-28 12:22:13    2.81 MB          blob
d51af753c3d3a984351448   2020-05-27 08:37:21    28.56 MB         blob
d9cbbca60e5f0fc028b13c   2020-05-28 15:19:06    760.85 kB        blob
db8816f445487e48e1d614   2020-05-27 12:57:25    1.93 MB          blob
fc878cd0a91c7bece56f66   2020-05-27 08:37:22    32.30 kB         blob
fee5db0ff82f7aa5ace634   2020-05-27 08:37:22    0.16 kB          blob
ff110406d51ca9ea722112   2020-05-27 12:57:25    7.78 kB          blob
sha256.02ee8bf9dc335c2   2020-05-29 13:45:14    28.11 MB         library
sha256.5111f59250ac94f   2020-05-28 13:14:39    782.34 kB        library
747d2dbbaaee995098c979   2020-05-28 14:07:22    27.77 MB         oci-tmp
9a839e63dad54c3a6d1834   2020-05-28 12:22:13    2.78 MB          oci-tmp

There are 4 container file(s) using 59.45 MB and 23 oci blob file(s) using 379.10 MB␣
↪of space
Total space used: 438.55 MB
```

All cache entries are named using a content hash, so that identical layers or images that are pulled from different URIs do not consume more space than needed.

Entries marked `blob` are OCI/docker layers and manifests, that are used to create SIF format images in the `oci-tmp` cache. Other caches are named for the source of the image e.g. `library` and `oras`.

You can limit the cache list to a specific cache type with the `-type` / `-t` option.

Cleaning the Cache

To reclaim space used by the Singularity cache, use `singularity cache clean`.

By default `singularity cache clean` will remove all cache entries, after asking you to confirm:

```
$ singularity cache clean
This will delete everything in your cache (containers from all sources and OCI blobs).
Hint: You can see exactly what would be deleted by canceling and using the --dry-run␣
↪option.
Do you want to continue? [N/y] n
```

Use the `--dry-run` / `-n` option to see the files that would be deleted, or the `--force` / `-f` option to clean without asking for confirmation.

If you want to leave your most recent cached images in place, but remove images that were cached longer ago, you can use the `--days` / `-d` option. E.g. to clean cache entries older than 30 days:

```
$ singularity cache clean --days 30
```

To remove only a specific kind of cache entry, e.g. only library images, use the `type` / `-T` option:

```
$ singularity cache clean --type library
```

2.3.4 Temporary Folders

When building a container, or pulling/running a Singularity container from a Docker/OCI source, a temporary working space is required. The container is constructed in this temporary space before being packaged into a Singularity SIF image. Temporary space is also used when running containers in unprivileged mode, and performing some operations on filesystems that do not fully support `--fakeroot`.

The location for temporary directories defaults to `/tmp`. Singularity will also respect the environment variable `TMPDIR`, and both of these locations can be overridden by setting the environment variable `SINGULARITY_TMPDIR`.

The temporary directory used during a build must be on a filesystem that has enough space to hold the entire container image, uncompressed, including any temporary files that are created and later removed during the build. You may need to set `SINGULARITY_TMPDIR` when building a large container on a system which has a small `/tmp` filesystem.

Remember to use `-E` option to pass the value of `SINGULARITY_TMPDIR` to root's environment when executing the `build` command with `sudo`.

Warning: Many modern Linux distributions use an in-memory `tmpfs` filesystem for `/tmp` when installed on a computer with a sufficient amount of RAM. This may limit the size of container you can build, as temporary directories under `/tmp` share RAM with runniing programs etc. A `tmpfs` also uses default mount options that can interfere with some container builds.

Set `SINGULARITY_TMPDIR` to a disk location, or disable the `tmpfs /tmp` mount on your system if you experience problems.

2.3.5 Encrypted Containers

Beginning in Singularity 3.4.0 it is possible to build and run encrypted containers. The containers are decrypted at runtime entirely in kernel space, meaning that no intermediate decrypted data is ever present on disk or in memory. See *encrypted containers* for more details.

2.3.6 Environment Variables

1. If a flag is represented by both a CLI option and an environment variable, and both are set, the CLI option will always take precedence. This is true for all environment variables except for `SINGULARITY_BIND` and `SINGULARITY_BINDPATH` which is combined with the `--bind` option, argument pair if both are present.
2. Environment variables overwrite default values in the CLI code
3. Any defaults in the CLI code are applied.

Defaults

The following variables have defaults that can be customized by you via environment variables at runtime.

Docker

SINGULARITY_DOCKER_LOGIN Used for the interactive login for Docker Hub.

SINGULARITY_DOCKER_USERNAME Your Docker username.

SINGULARITY_DOCKER_PASSWORD Your Docker password.

RUNSCRIPT_COMMAND Is not obtained from the environment, but is a hard coded default ("/bin/bash"). This is the fallback command used in the case that the docker image does not have a CMD or ENTRYPOINT. **TAG** Is the default tag, `latest`.

SINGULARITY_NOHTTPS This is relevant if you want to use a registry that doesn't have https, and it speaks for itself. If you export the variable `SINGULARITY_NOHTTPS` you can force the software to not use https when interacting with a Docker registry. This use case is typically for use of a local registry.

Library

SINGULARITY_BUILDER Used to specify the remote builder service URL. The default value is our remote builder.

SINGULARITY_LIBRARY Used to specify the library to pull from. Default is set to our Cloud Library.

SINGULARITY_REMOTE Used to build an image remotely (This does not require root). The default is set to false.

Encryption

SINGULARITY_ENCRYPTION_PASSPHRASE Used to pass a plaintext passphrase to encrypt a container file system (with the `--encrypt` flag). The default is empty.

SINGULARITY_ENCRYPTION_PEM_PATH Used to specify the location of a public key to use for container encryption (with the `--encrypt` flag). The default is empty.

2.4 Support for Docker and OCI

2.4.1 Overview

Effort has been expended in developing Docker containers. Deconstructed into one or more compressed archives (typically split across multiple segments, or **layers** as they are known in Docker parlance) plus some metadata, images for these containers are built from specifications known as `Dockerfiles`. The public Docker Hub, as well as various private registries, host images for use as Docker containers. Singularity has from the outset emphasized the importance of interoperability with Docker. As a consequence, this section of the Singularity User Docs first makes its sole focus interoperabilty with Docker. In so doing, the following topics receive attention here:

- Application of Singularity action commands on ephemeral containers derived from public Docker images
- Converting public Docker images into Singularity's native format for containerization, namely the Singularity Image Format (SIF)
- Authenticated application of Singularity commands to containers derived from private Docker images
- Authenticated application of Singularity commands to containers derived from private Docker images originating from private registries
- Building SIF containers for Singularity via the command line or definition files from a variety of sources for Docker images and image archives

The second part of this section places emphasis upon Singularity's interoperability with open standards emerging from the Open Containers Initiative (OCI). Specifically, in documenting Singularity interoperability as it relates to the OCI Image Specification, the following topics are covered:

- Compliance with the OCI Image Layout Specification
- OCI-compliant caching in Singularity
- Acquiring OCI images and image archives via Singularity
- Building SIF containers for Singularity via the command line or definition files from a variety of sources for OCI images and image archives

The section closes with a brief enumeration of emerging best practices plus consideration of troubleshooting common issues.

2.4.2 Running action commands on public images from Docker Hub

`godlovedc/lolcow` is a whimsical example of a publicly accessible image hosted via Docker Hub. Singularity can execute this image as follows:

```
$ singularity run docker://godlovedc/lolcow
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
 45.33 MiB / 45.33 MiB [=====================================================] 1s
Copying blob sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
 848 B / 848 B [============================================================] 0s
Copying blob sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
 621 B / 621 B [============================================================] 0s
Copying blob sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
 853 B / 853 B [============================================================] 0s
Copying blob sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
 169 B / 169 B [============================================================] 0s
Copying blob sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
 53.75 MiB / 53.75 MiB [=====================================================] 2s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [=======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: /home/vagrant/.singularity/cache/oci-tmp/
↪a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb/lolcow_latest.sif
INFO:    Image cached as SIF at /home/vagrant/.singularity/cache/oci-tmp/
↪a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb/lolcow_latest.sif
 ___________________________________
/ Repartee is something we think of \
| twenty-four hours too late.       |
|                                   |
\ -- Mark Twain                     /
 -----------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Here `docker` is prepended to ensure that the `run` command of Singularity is instructed to boostrap container creation based upon this Docker image, thus creating a complete URI for Singularity. Singularity subsequently downloads *all the OCI blobs that comprise this image*, and converts them into a *single* SIF file - the native format for Singularity containers. Because this image from Docker Hub is cached locally in the `$HOME/.singularity/cache/oci-tmp/<org.opencontainers.image.ref.name>/lolcow_latest.sif` directory, where `<org.opencontainers.image.ref.name>` will be replaced by the appropriate hash for the container, the image does not need to be downloaded again (from Docker Hub) the next time a Singularity `run` is executed. In other words, the cached copy is sensibly reused:

```
$ singularity run docker://godlovedc/lolcow
 _________________________________________
/ Soap and education are not as sudden as \
| a massacre, but they are more deadly in |
| the long run.                           |
|                                         |
\ -- Mark Twain                           /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note: Image caching is *documented in detail below*.

Note: Use is made of the `$HOME/.singularity` directory by default to *cache images*. To cache images elsewhere, use of the environment variable `SINGULARITY_CACHEDIR` can be made.

As the runtime of this container is encapsulated as a single SIF file, it is possible to

```
cd /home/vagrant/.singularity/cache/oci-tmp/
↪a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb/
```

and then execute the SIF file directly:

```
./lolcow_latest.sif
 _______________________________________
/ The secret source of humor is not joy \
| but sorrow; there is no humor in      |
| Heaven.                               |
|                                       |
\ -- Mark Twain                         /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note: SIF files abstract Singularity containers as a single file. As with any executable, a SIF file can be executed directly.

`fortune | cowsay | lolcat` is executed by *default* when this container is `run` by Singularity. Singularity's `exec` command allows a different command to be executed; for example:

```
$ singularity exec docker://godlovedc/lolcow fortune
Don't go around saying the world owes you a living.  The world owes you
nothing.  It was here first.
        -- Mark Twain
```

Note: The *same* cached copy of the `lolcow` container is reused here by Singularity `exec`, and immediately below here by `shell`.

Note: Execution defaults are documented below - see *Directing Execution* and *Container Metadata*.

In addition to non-interactive execution of an image from Docker Hub, Singularity provides support for an *interactive* `shell` session:

```
$ singularity shell docker://godlovedc/lolcow
Singularity lolcow_latest.sif:~> cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.3 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.3 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
Singularity lolcow_latest.sif:~>
```

From this it is evident that use is being made of Ubuntu 16.04 *within* this container, whereas the shell *external* to the container is running a more recent release of Ubuntu (not illustrated here).

`inspect` reveals the metadata for a Singularity container encapsulated via SIF; *Container Metadata* is documented below.

Note: `singularity search [search options...] <search query>` does *not* support Docker registries like Docker Hub. Use the search box at Docker Hub to locate Docker images. Docker `pull` commands, e.g., `docker pull godlovedc/lolcow`, can be easily translated into the corresponding command for Singularity. The Docker `pull` command is available under "DETAILS" for a given image on Docker Hub.

2.4.3 Making use of public images from Docker Hub

Singularity can make use of public images available from the Docker Hub. By specifying the `docker://` URI for an image that has already been located, Singularity can `pull` it - e.g.:

```
$ singularity pull docker://godlovedc/lolcow
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
 45.33 MiB / 45.33 MiB [====================================================] 2s
Copying blob sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
 848 B / 848 B [============================================================] 0s
Copying blob sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
 621 B / 621 B [============================================================] 0s
Copying blob sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
 853 B / 853 B [============================================================] 0s
Copying blob sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
 169 B / 169 B [============================================================] 0s
Copying blob sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
 53.75 MiB / 53.75 MiB [====================================================] 3s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_latest.sif
```

This `pull` results in a *local* copy of the Docker image in SIF, the Singularity Image Format:

```
$ file lolcow_latest.sif
lolcow_latest.sif: a /usr/bin/env run-singularity script executable (binary data)
```

In converting to SIF, individual layers of the Docker image have been *combined* into a single, native file for use by Singularity; there is no need to subsequently `build` the image for Singularity. For example, you can now `exec`, `run` or `shell` into the SIF version via Singularity, *as described above*.

`inspect` reveals metadata for the container encapsulated via SIF:

```
$ singularity inspect lolcow_latest.sif

{
    "org.label-schema.build-date": "Thursday_6_December_2018_17:29:48_UTC",
    "org.label-schema.schema-version": "1.0",
    "org.label-schema.usage.singularity.deffile.bootstrap": "docker",
    "org.label-schema.usage.singularity.deffile.from": "godlovedc/lolcow",
    "org.label-schema.usage.singularity.version": "3.0.1-40.g84083b4f"
}
```

Note: *Container Metadata* is documented below.

SIF files built from Docker images are *not* crytographically signed:

```
$ singularity verify lolcow_latest.sif
Verifying image: lolcow_latest.sif
ERROR:   verification failed: error while searching for signature blocks: no␣
↪signatures found for system partition
```

The `sign` command allows a cryptographic signature to be added. Refer to *Signing and Verifying Containers* for details. But caution should be exercised in signing images from Docker Hub because, unless you build an image from scratch (OS mirrors) you are probably not really sure about the complete contents of that image.

Note: `pull` is a one-time-only operation that builds a SIF file corresponding to the image retrieved from Docker Hub. Updates to the image on Docker Hub will *not* be reflected in the *local* copy.

In our example `docker://godlovedc/lolcow`, `godlovedc` specifies a Docker Hub user, whereas `lolcow` is the name of the repository. Adding the option to specifiy an image tag, the generic version of the URI is `docker://<user>/<repo-name>[:<tag>]`. Repositories on Docker Hub provides additional details.

2.4.4 Making use of private images from Docker Hub

After successful authentication, Singularity can also make use of *private* images available from the Docker Hub. The two means available for authentication follow here. Before describing these means, it is instructive to illustate the error generated when attempting access a private image *without* credentials:

```
$ singularity pull docker://ilumb/mylolcow
INFO:    Starting build...
FATAL:   Unable to pull docker://ilumb/mylolcow: conveyor failed to get: Error␣
↪reading manifest latest in docker.io/ilumb/mylolcow: errors:
denied: requested access to the resource is denied
unauthorized: authentication required
```

In this case, the `mylolcow` repository of user `ilumb` **requires** authentication through specification of a valid username and password.

Authentication via Interactive Login

Interactive login is the first of two means provided for authentication with Docker Hub. It is enabled through use of the `--docker-login` option of Singularity's `pull` command; for example:

```
$ singularity pull --docker-login docker://ilumb/mylolcow
Enter Docker Username: ilumb
Enter Docker Password:
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:7b8b6451c85f072fd0d7961c97be3fe6e2f772657d471254f6d52ad9f158a580
Skipping fetch of repeat blob␣
↪sha256:ab4d1096d9ba178819a3f71f17add95285b393e96d08c8a6bfc3446355bcdc49
Skipping fetch of repeat blob␣
↪sha256:e6797d1788acd741d33f4530106586ffee568be513d47e6e20a4c9bc3858822e
Skipping fetch of repeat blob␣
↪sha256:e25c5c290bded5267364aa9f59a18dd22a8b776d7658a41ffabbf691d8104e36
Skipping fetch of repeat blob␣
↪sha256:258e068bc5e36969d3ba4b47fd3ca0d392c6de465726994f7432b14b0414d23b
Copying config sha256:8a8f815257182b770d32dffff7f185013b4041d076e065893f9dd1e89ad8a671
 3.12 KiB / 3.12 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: mylolcow_latest.sif
```

After successful authentication, the private Docker image is pulled and converted to SIF as described above.

Note: For interactive sessions, `--docker-login` is *recommended* as use of plain-text passwords in your environment is *avoided*. Encoded authentication data is communicated with Docker Hub via secure HTTP.

Authentication via Environment Variables

Environment variables offer an alternative means for authentication with Docker Hub. The **required** exports are as follows:

```
export SINGULARITY_DOCKER_USERNAME=ilumb
export SINGULARITY_DOCKER_PASSWORD=<redacted>
```

Of course, the `<redacted>` plain-text password needs to be replaced by a valid one to be of practical use.

Based upon these exports, `$ singularity pull docker://ilumb/mylolcow` allows for the retrieval of this private image.

Note: This approach for authentication supports both interactive and non-interactive sessions. However, the requirement for a plain-text password assigned to an envrionment variable, is the security compromise for this flexibility.

Note: When specifying passwords, 'special characters' (e.g., `$`, `#`, `.`) need to be 'escaped' to avoid interpretation by the shell.

2.4.5 Making use of private images from Private Registries

Authentication is required to access *private* images that reside in Docker Hub. Of course, private images can also reside in **private registries**. Accounting for locations *other* than Docker Hub is easily achieved.

In the complete command line specification

```
docker://<registry>/<user>/<repo-name>[:<tag>]
```

`registry` defaults to `index.docker.io`. In other words,

```
$ singularity pull docker://godlovedc/lolcow
```

is functionally equivalent to

```
$ singularity pull docker://index.docker.io/godlovedc/lolcow
```

From the above example, it is evident that

```
$ singularity pull docker://nvcr.io/nvidia/pytorch:18.11-py3
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:18d680d616571900d78ee1c8fff0310f2a2afe39c6ed0ba2651ff667af406c3e
<blob fetching details deleted>
Skipping fetch of repeat blob␣
↪sha256:c71aeebc266c779eb4e769c98c935356a930b16d881d7dde4db510a09cfa4222
```

(continues on next page)

(continued from previous page)

```
Copying config sha256:b77551af8073c85588088ab2a39007d04bc830831ba1eef4127b2d39aaf3a6b1
 21.28 KiB / 21.28 KiB [==================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: pytorch_18.11-py3.sif
```

will retrieve a specific version of the PyTorch platform for Deep Learning from the NVIDIA GPU Cloud (NGC). Because NGC is a private registry, the above `pull` assumes *authentication via environment variables* when the blobs that collectively comprise the Docker image have not already been cached locally. In the NGC case, the required environment variable are set as follows:

```
export SINGULARITY_DOCKER_USERNAME='$oauthtoken'
export SINGULARITY_DOCKER_PASSWORD=<redacted>
```

Upon use, these environment-variable settings allow for authentication with NGC.

Note: `$oauthtoken` is to be taken literally - it is not, for example, an environment variable.

The password provided via these means is actually an API token. This token is generated via your NGC account, and is **required** for use of the service.

For additional details regarding authentication with NGC, and much more, please consult the NGC Getting Started documentation.

Alternatively, for purely interactive use, `--docker-login` is recommended:

```
$ singularity pull --docker-login docker://nvcr.io/nvidia/pytorch:18.11-py3
Enter Docker Username: $oauthtoken
Enter Docker Password:
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:18d680d616571900d78ee1c8fff0310f2a2afe39c6ed0ba2651ff667af406c3e
<blob fetching details deleted>
Skipping fetch of repeat blob␣
↪sha256:c71aeebc266c779eb4e769c98c935356a930b16d881d7dde4db510a09cfa4222
Copying config sha256:b77551af8073c85588088ab2a39007d04bc830831ba1eef4127b2d39aaf3a6b1
21.28 KiB / 21.28 KiB [==================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: pytorch_18.11-py3.sif
```

Authentication aside, the outcome of the `pull` command is the Singularity container `pytorch_18.11-py3.sif` - i.e., a locally stored copy, that has been coverted to SIF.

2.4.6 Building images for Singularity from Docker Registries

The `build` command is used to **create** Singularity containers. Because it is documented extensively *elsewhere in this manual*, only specifics relevant to Docker are provided here - namely, working with Docker Hub via *the Singularity command line* and through *Singularity definition files*.

Working from the Singularity Command Line

Remotely Hosted Images

In the simplest case, `build` is functionally equivalent to `pull`:

```
$ singularity build mylolcow_latest.sif docker://godlovedc/lolcow
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
Skipping fetch of repeat blob␣
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
Skipping fetch of repeat blob␣
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
Skipping fetch of repeat blob␣
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
Skipping fetch of repeat blob␣
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
Skipping fetch of repeat blob␣
↪sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [==================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: mylolcow_latest.sif
```

This `build` results in a *local* copy of the Docker image in SIF, as did `pull` *above*. Here, `build` has named the Singularity container `mylolcow_latest.sif`.

Note: `docker://godlovedc/lolcow` is the **target** provided as input for `build`. Armed with this target, `build` applies the appropriate boostrap agent to create the container - in this case, one appropriate for Docker Hub.

In addition to a read-only container image in SIF (**default**), `build` allows for the creation of a writable (ch)root *directory* called a **sandbox** for interactive development via the `--sandbox` option:

```
$ singularity build --sandbox mylolcow_latest_sandbox docker://godlovedc/lolcow
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
Skipping fetch of repeat blob␣
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
Skipping fetch of repeat blob␣
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
Skipping fetch of repeat blob␣
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
```

(continues on next page)

(continued from previous page)

```
Skipping fetch of repeat blob␣
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
Skipping fetch of repeat blob␣
↪sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating sandbox directory...
INFO:    Build complete: mylolcow_latest_sandbox
```

After successful execution, the above command results in creation of the `mylolcow_latest_sandbox` directory with contents:

```
bin  boot  core  dev  environment  etc  home  lib  lib64  media  mnt  opt  proc  root␣
↪ run  sbin  singularity  srv  sys  tmp  usr  var
```

The `build` command of Singularity allows (e.g., development) sandbox containers to be converted into (e.g., production) read-only SIF containers, and vice-versa. Consult the *Build a container* documentation for the details.

Implicit in the above command-line interactions is use of public images from Docker Hub. To make use of **private** images from Docker Hub, authentication is required. Available means for authentication were described above. Use of environment variables is functionally equivalent for Singularity `build` as it is for `pull`; see *Authentication via Environment Variables* above. For purely interactive use, authentication can be added to the `build` command as follows:

```
singularity build --docker-login mylolcow_latest_il.sif docker://ilumb/mylolcow
```

(Recall that `docker://ilumb/mylolcow` is a private image available via Docker Hub.) See *Authentication via Interactive Login* above regarding use of `--docker-login`.

Building Containers Remotely

By making use of the Sylabs Cloud Remote Builder, it is possible to build SIF containers *remotely* from images hosted at Docker Hub. The Sylabs Cloud Remote Builder is a **service** that can be used from the Singularity command line or via its Web interface. Here use of the Singularity CLI is emphasized.

Once you have an account for Sylabs Cloud, and have logged in to the portal, select Remote Builder. The right-hand side of this page is devoted to use of the Singularity CLI. Self-generated API tokens are used to enable authenticated access to the Remote Builder. To create a token, follow the instructions provided. Once the token has been created, run `singularity remote login` and paste it at the prompt.

The above token provides *authenticated* use of the Sylabs Cloud Remote Builder when `--remote` is *appended* to the Singularity `build` command. For example, for remotely hosted images:

```
$ singularity build --remote lolcow_rb.sif docker://godlovedc/lolcow
searching for available build agent.........INFO:    Starting build...
Getting image source signatures
Copying blob sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
 45.33 MiB / 45.33 MiB  0s
Copying blob sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
 848 B / 848 B  0s
Copying blob sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
 621 B / 621 B  0s
Copying blob sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
```

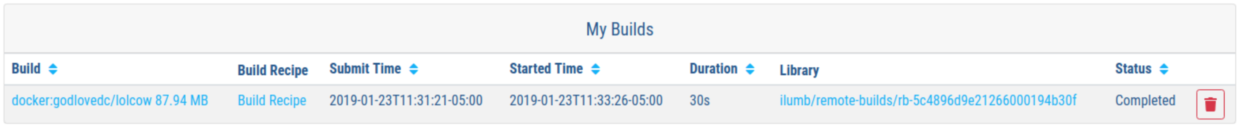
(continues on next page)

(continued from previous page)

```
 853 B / 853 B  0s
Copying blob sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
 169 B / 169 B  0s
Copying blob sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
 53.75 MiB / 53.75 MiB  0s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB  0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: /tmp/image-341891107
INFO:    Now uploading /tmp/image-341891107 to the library
 87.94 MiB / 87.94 MiB  100.00% 38.96 MiB/s 2s
INFO:    Setting tag latest
 87.94 MiB / 87.94 MiB
↪[===========================================================================]
↪100.00% 17.23 MiB/s 5s
```

Note: Elevated privileges (e.g., via `sudo`) are *not* required when use is made of the Sylabs Cloud Remote Builder.

During the build process, progress can be monitored in the Sylabs Cloud portal on the Remote Builder page - as illustrated upon completion by the screenshot below. Once complete, this results in a *local* copy of the SIF file `lolcow_rb.sif`. From the Sylabs Cloud Singularity Library it is evident that the 'original' SIF file remains available via this portal.



My Builds

Build	Build Recipe	Submit Time	Started Time	Duration	Library	Status
docker:godlovedc/lolcow 87.94 MB	Build Recipe	2019-01-23T11:31:21-05:00	2019-01-23T11:33:26-05:00	30s	ilumb/remote-builds/rb-5c4896d9e21266000194b30f	Completed

Locally Available Images: Cached by Docker

Singularity containers can be built at the command line from images cached *locally* by Docker. Suppose, for example:

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
godlovedc/lolcow    latest              577c1fe8e6d8        16 months ago       241MB
```

This indicates that `godlovedc/lolcow:latest` has been cached locally by Docker. Then

```
$ sudo singularity build lolcow_from_docker_cache.sif docker-daemon://godlovedc/
↪lolcow:latest
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:a2022691bf950a72f9d2d84d557183cb9eee07c065a76485f1695784855c5193
 119.83 MiB / 119.83 MiB [==================================================] 6s
Copying blob sha256:ae620432889d2553535199dbdd8ba5a264ce85fcdcd5a430974d81fc27c02b45
 15.50 KiB / 15.50 KiB [====================================================] 0s
Copying blob sha256:c561538251751e3685c7c6e7479d488745455ad7f84e842019dcb452c7b6fecc
 14.50 KiB / 14.50 KiB [====================================================] 0s
Copying blob sha256:f96e6b25195f1b36ad02598b5d4381e41997c93ce6170cab1b81d9c68c514db0
 5.50 KiB / 5.50 KiB [======================================================] 0s
```

(continues on next page)

(continued from previous page)

```
Copying blob sha256:7f7a065d245a6501a782bf674f4d7e9d0a62fa6bd212edbf1f17bad0d5cd0bfc
 3.00 KiB / 3.00 KiB [=====================================================] 0s
Copying blob sha256:70ca7d49f8e9c44705431e3dade0636a2156300ae646ff4f09c904c138728839
 116.56 MiB / 116.56 MiB [=================================================] 6s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [=====================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_from_docker_cache.sif
```

results in `lolcow_from_docker_cache.sif` for native use by Singularity. There are two important differences in syntax evident in the above `build` command:

1. The `docker` part of the URI has been appended by `daemon`. This ensures Singularity seek an image locally cached by Docker to boostrap the conversion process to SIF, as opposed to attempting to retrieve an image remotely hosted via Docker Hub.
2. `sudo` is prepended to the `build` command for Singularity; this is required as the Docker daemon executes as `root`. However, if the user issuing the `build` command is a member of the `docker` Linux group, then `sudo` need not be prepended.

Note: The image tag, in this case `latest`, is **required** when bootstrapping creation of a container for Singularity from an image locally cached by Docker.

Note: The Sylabs Cloud Remote Builder *does not* interoperate with local Docker daemons; therefore, images cached locally by Docker, *cannot* be used to bootstrap creation of SIF files via the Remote Builder service. Of course, a SIF file could be created locally as detailed above. Then, in a separate, manual step, *pushed to the Sylabs Cloud Singularity Library*.

Locally Available Images: Stored Archives

Singularity containers can also be built at the command line from Docker images stored locally as `tar` files.

The `lolcow.tar` file employed below in this example can be produced by making use of an environment in which Docker is available as follows:

1. Obtain a local copy of the image from Docker Hub via `sudo docker pull godlovedc/lolcow`. Issuing the following command confirms that a copy of the desired image is available locally:

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             ␣
↪   SIZE
godlovedc/lolcow    latest              577c1fe8e6d8        17 months ago       ␣
↪   241MB
```

2. Noting that the image identifier above is `577c1fe8e6d8`, the required archive can be created by `sudo docker save 577c1fe8e6d8 -o lolcow.tar`.

Thus `lolcow.tar` is a locally stored archive in the *current* working directory with contents:

```
$ sudo tar tvf lolcow.tar
drwxr-xr-x 0/0               0 2017-09-21 19:37␣
↪02aefa059d08482d344293d0ad27182a0a9d330ebc73abd92a1f9744844f91e9/
-rw-r--r-- 0/0               3 2017-09-21 19:37␣
↪02aefa059d08482d344293d0ad27182a0a9d330ebc73abd92a1f9744844f91e9/VERSION
-rw-r--r-- 0/0            1417 2017-09-21 19:37␣
↪02aefa059d08482d344293d0ad27182a0a9d330ebc73abd92a1f9744844f91e9/json
-rw-r--r-- 0/0       122219008 2017-09-21 19:37␣
↪02aefa059d08482d344293d0ad27182a0a9d330ebc73abd92a1f9744844f91e9/layer.tar
drwxr-xr-x 0/0               0 2017-09-21 19:37␣
↪3762e087ebbb895fd9c38981c1f7bfc76c9879fd3fdadef64df49e92721bb527/
-rw-r--r-- 0/0               3 2017-09-21 19:37␣
↪3762e087ebbb895fd9c38981c1f7bfc76c9879fd3fdadef64df49e92721bb527/VERSION
-rw-r--r-- 0/0             482 2017-09-21 19:37␣
↪3762e087ebbb895fd9c38981c1f7bfc76c9879fd3fdadef64df49e92721bb527/json
-rw-r--r-- 0/0           14848 2017-09-21 19:37␣
↪3762e087ebbb895fd9c38981c1f7bfc76c9879fd3fdadef64df49e92721bb527/layer.tar
-rw-r--r-- 0/0            4432 2017-09-21 19:37␣
↪577c1fe8e6d84360932b51767b65567550141af0801ff6d24ad10963e40472c5.json
drwxr-xr-x 0/0               0 2017-09-21 19:37␣
↪5bad884501c0e760bc0c9ca3ae3dca3f12c4abeb7d18194c364fec522b91b4f9/
-rw-r--r-- 0/0               3 2017-09-21 19:37␣
↪5bad884501c0e760bc0c9ca3ae3dca3f12c4abeb7d18194c364fec522b91b4f9/VERSION
-rw-r--r-- 0/0             482 2017-09-21 19:37␣
↪5bad884501c0e760bc0c9ca3ae3dca3f12c4abeb7d18194c364fec522b91b4f9/json
-rw-r--r-- 0/0            3072 2017-09-21 19:37␣
↪5bad884501c0e760bc0c9ca3ae3dca3f12c4abeb7d18194c364fec522b91b4f9/layer.tar
drwxr-xr-x 0/0               0 2017-09-21 19:37␣
↪81ce2fd011bc8241ae72eaee9146116b7c289e941467ff276397720171e6c576/
-rw-r--r-- 0/0               3 2017-09-21 19:37␣
↪81ce2fd011bc8241ae72eaee9146116b7c289e941467ff276397720171e6c576/VERSION
-rw-r--r-- 0/0             406 2017-09-21 19:37␣
↪81ce2fd011bc8241ae72eaee9146116b7c289e941467ff276397720171e6c576/json
-rw-r--r-- 0/0       125649920 2017-09-21 19:37␣
↪81ce2fd011bc8241ae72eaee9146116b7c289e941467ff276397720171e6c576/layer.tar
drwxr-xr-x 0/0               0 2017-09-21 19:37␣
↪a10239905b060fd8b17ab31f37957bd126774f52f5280767d3b2639692913499/
-rw-r--r-- 0/0               3 2017-09-21 19:37␣
↪a10239905b060fd8b17ab31f37957bd126774f52f5280767d3b2639692913499/VERSION
-rw-r--r-- 0/0             482 2017-09-21 19:37␣
↪a10239905b060fd8b17ab31f37957bd126774f52f5280767d3b2639692913499/json
-rw-r--r-- 0/0           15872 2017-09-21 19:37␣
↪a10239905b060fd8b17ab31f37957bd126774f52f5280767d3b2639692913499/layer.tar
drwxr-xr-x 0/0               0 2017-09-21 19:37␣
↪ab6e1ca3392b2f4dbb60157cf99434b6975f37a767f530e293704a7348407634/
-rw-r--r-- 0/0               3 2017-09-21 19:37␣
↪ab6e1ca3392b2f4dbb60157cf99434b6975f37a767f530e293704a7348407634/VERSION
-rw-r--r-- 0/0             482 2017-09-21 19:37␣
↪ab6e1ca3392b2f4dbb60157cf99434b6975f37a767f530e293704a7348407634/json
-rw-r--r-- 0/0            5632 2017-09-21 19:37␣
↪ab6e1ca3392b2f4dbb60157cf99434b6975f37a767f530e293704a7348407634/layer.tar
-rw-r--r-- 0/0             574 1970-01-01 01:00 manifest.json
```

In other words, it is evident that this 'tarball' is a Docker-format image comprised of multiple layers along with metadata in a JSON manifest.

Through use of the `docker-archive` bootstrap agent, a SIF file (`lolcow_tar.sif`) for use by Singularity can be created via the following `build` command:

```
$ singularity build lolcow_tar.sif docker-archive://lolcow.tar
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:a2022691bf950a72f9d2d84d557183cb9eee07c065a76485f1695784855c5193
 119.83 MiB / 119.83 MiB [==================================================] 6s
Copying blob sha256:ae620432889d2553535199dbdd8ba5a264ce85fcdcd5a430974d81fc27c02b45
 15.50 KiB / 15.50 KiB [====================================================] 0s
Copying blob sha256:c561538251751e3685c7c6e7479d488745455ad7f84e842019dcb452c7b6fecc
 14.50 KiB / 14.50 KiB [====================================================] 0s
Copying blob sha256:f96e6b25195f1b36ad02598b5d4381e41997c93ce6170cab1b81d9c68c514db0
 5.50 KiB / 5.50 KiB [======================================================] 0s
Copying blob sha256:7f7a065d245a6501a782bf674f4d7e9d0a62fa6bd212edbf1f17bad0d5cd0bfc
 3.00 KiB / 3.00 KiB [======================================================] 0s
Copying blob sha256:70ca7d49f8e9c44705431e3dade0636a2156300ae646ff4f09c904c138728839
 116.56 MiB / 116.56 MiB [==================================================] 6s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_tar.sif
```

There are two important differences in syntax evident in the above `build` command:

1. The `docker` part of the URI has been appended by `archive`. This ensures Singularity seek a Docker-format image archive stored locally as `lolcow.tar` to boostrap the conversion process to SIF, as opposed to attempting to retrieve an image remotely hosted via Docker Hub.
2. `sudo` is *not* prepended to the `build` command for Singularity. This is *not* required if the executing user has the appropriate access privileges to the stored file.

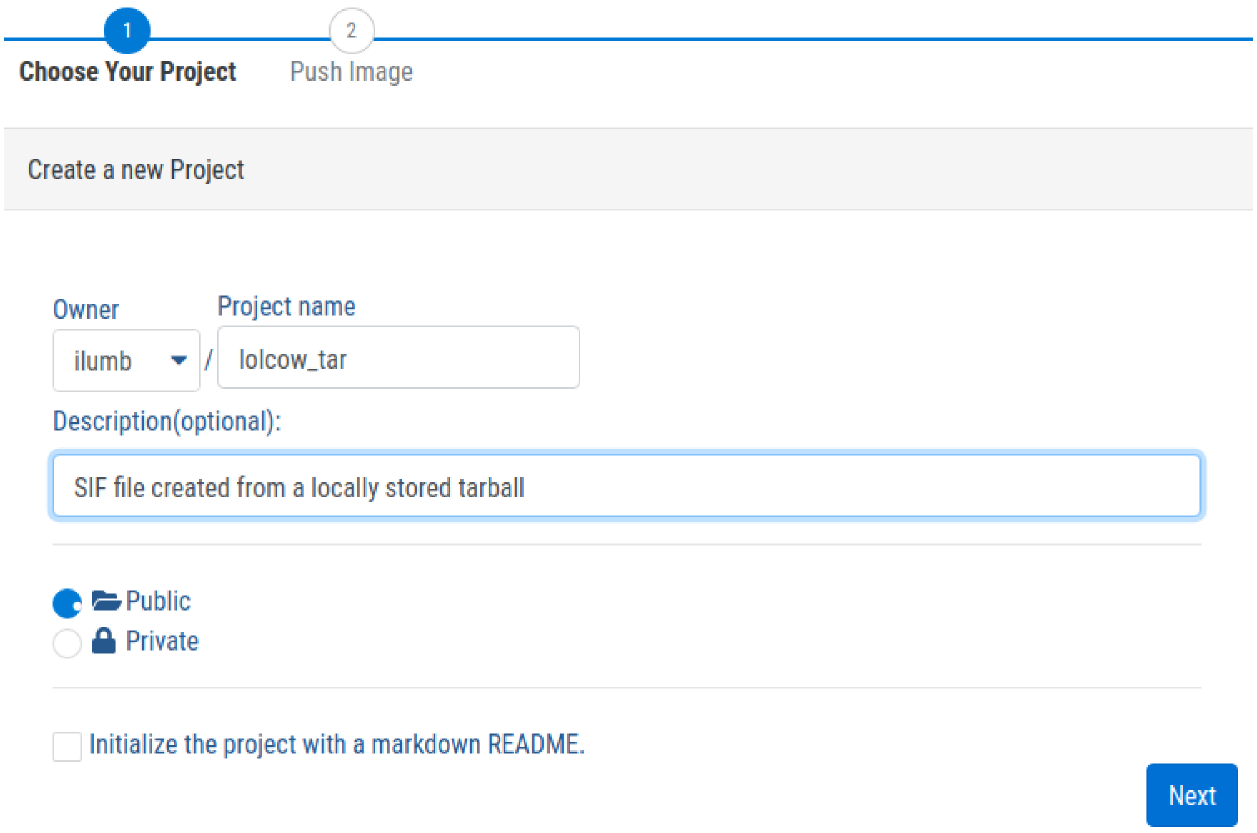
Note: The `docker-archive` bootstrap agent handles archives (`.tar` files) as well as compressed archives (`.tar.gz`) when containers are built for Singularity via its `build` command.

Note: The Sylabs Cloud Remote Builder *does not* interoperate with locally stored Docker-format images; therefore, images cached locally by Docker, *cannot* be used to bootstrap creation of SIF files via the Remote Builder service. Of course, a SIF file could be created locally as detailed above. Then, in a separate, manual step, *pushed to the Sylabs Cloud Singularity Library*.

Pushing Locally Available Images to a Library

The outcome of bootstrapping from an image cached locally by Docker, or one stored locally as an archive, is of course a *locally* stored SIF file. As noted above, this is the *only* option available, as the Sylabs Cloud Remote Builder *does not* interoperate with the Docker daemon or locally stored archives in the Docker image format. Once produced, however, it may be desirable to make the resulting SIF file available through the Sylabs Cloud Singularity Library; therefore, the procedure to `push` a locally available SIF file to the Library is detailed here.

From the Sylabs Cloud Singularity Library, select `Create a new Project`. In this first of two steps, the publicly accessible project is created as illustrated below:



Create a new Project

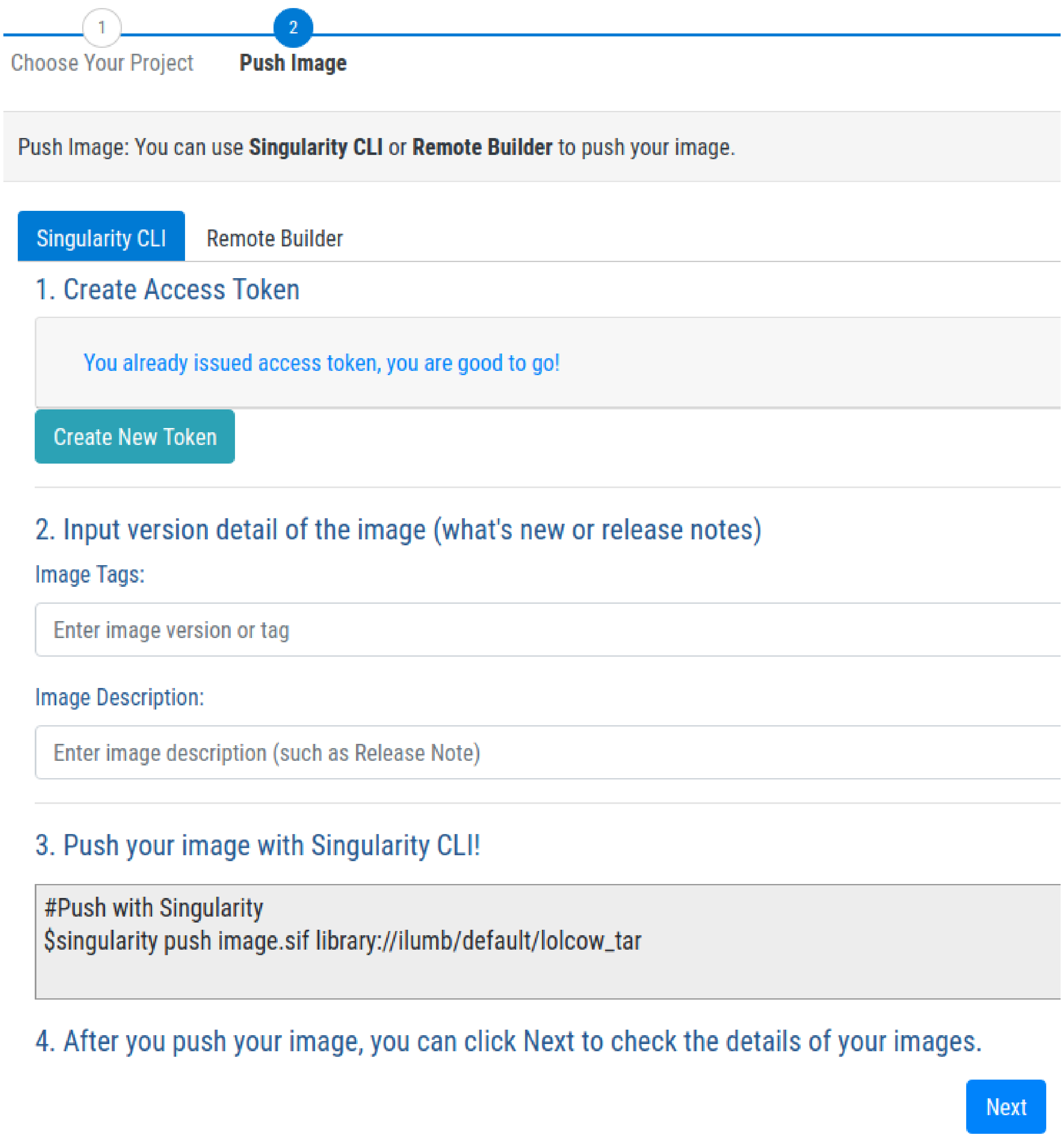
Owner: ilumb / Project name: lolcow_tar

Description(optional): SIF file created from a locally stored tarball

[✓ Public]
Private

Initialize the project with a markdown README.

Because an access token for the cloud service already exists, attention can be focused on the `push` command prototyped towards the bottom of the following screenshot:



Push Image: You can use **Singularity CLI** or **Remote Builder** to push your image.

1. Create Access Token

You already issued access token, you are good to go!

2. Input version detail of the image (what's new or release notes)

Image Tags:

Enter image version or tag

Image Description:

Enter image description (such as Release Note)

3. Push your image with Singularity CLI!

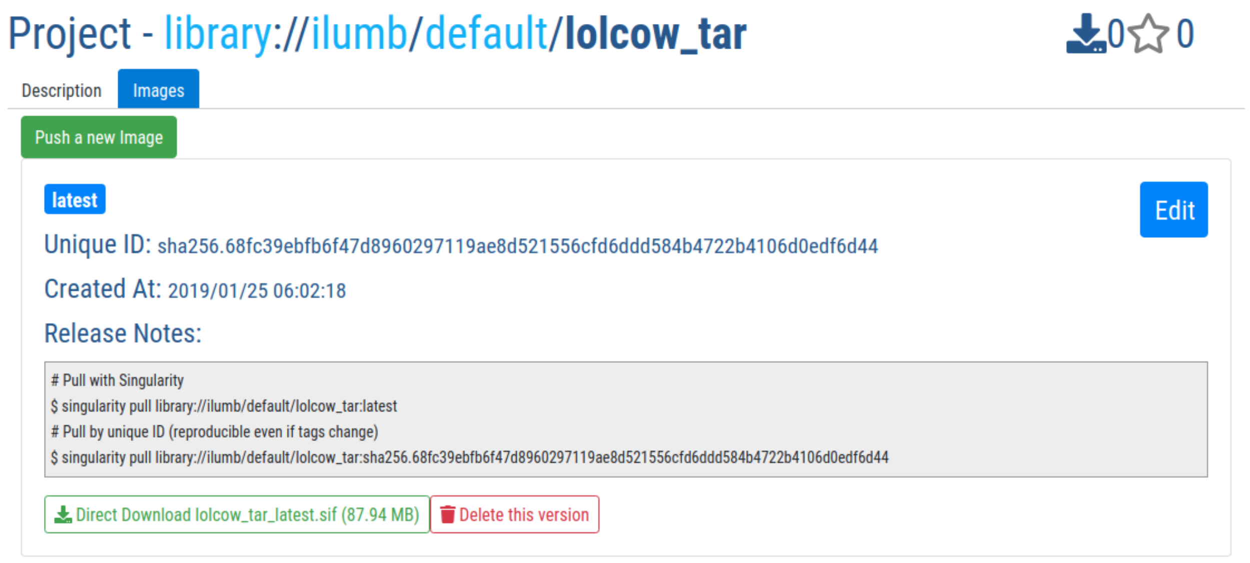
```
#Push with Singularity
$singularity push image.sif library://ilumb/default/lolcow_tar
```

4. After you push your image, you can click Next to check the details of your images.

In fact, by simply replacing `image.sif` with `lolcow_tar.sif`, the following upload is executed:

```
$ singularity push lolcow_tar.sif library://ilumb/default/lolcow_tar
INFO:    Now uploading lolcow_tar.sif to the library
 87.94 MiB / 87.94 MiB␣
↪[=====================================================================] 100.
↪00% 1.25 MiB/s 1m10s
INFO:    Setting tag latest
```

Finally, from the perspective of the Library, the *hosted* version of the SIF file appears as illustrated below. Directions on how to `pull` this file are included from the portal.



Unique ID: sha256.68fc39ebfb6f47d8960297119ae8d521556cfd6ddd584b4722b4106d0edf6d44

Created At: 2019/01/25 06:02:18

Release Notes:

```
# Pull with Singularity
$ singularity pull library://ilumb/default/lolcow_tar:latest
# Pull by unique ID (reproducible even if tags change)
$ singularity pull library://ilumb/default/lolcow_tar:sha256.68fc39ebfb6f47d8960297119ae8d521556cfd6ddd584b4722b4106d0edf6d44
```

Note: The hosted version of the SIF file in the Sylabs Cloud Singularity Library is maintainable. In other words, if the image is updated locally, the update can be pushed to the Library and tagged appropriately.

Working with Definition Files

Mandatory Header Keywords: Remotely Boostrapped

Akin to a set of blueprints that explain how to build a custom container, Singularity definition files (or "def files") are considered in detail elsewhere in this manual. Therefore, only def file nuances specific to interoperability with Docker receive consideration here.

Singularity definition files are comprised of two parts - a **header** plus **sections**.

When working with repositories such as Docker Hub, `Bootstrap` and `From` are **mandatory** keywords within the header; for example, if the file `lolcow.def` has contents

```
Bootstrap: docker
From: godlovedc/lolcow
```

then

```
sudo singularity build lolcow.sif lolcow.def
```

creates a Singularity container in SIF by bootstrapping from the public `godlovedc/lolcow` image from Docker Hub.

In the above definition file, `docker` is one of numerous, possible bootstrap agents; this, and other bootstrap agents receive attention *in the appendix*.

Through *the means for authentication described above*, definition files permit use of private images hosted via Docker Hub. For example, if the file `mylolcow.def` has contents

```
Bootstrap: docker
From: ilumb/mylolcow
```

then

```
sudo singularity build --docker-login mylolcow.sif mylolcow.def
```

creates a Singularity container in SIF by bootstrapping from the *private* `ilumb/mylolcow` image from Docker Hub after successful *interactive authentication*.

Alternatively, if *environment variables have been set as above*, then

```
$ sudo -E singularity build mylolcow.sif mylolcow.def
```

enables authenticated use of the private image.

Note: The `-E` option is required to preserve the user's existing environment variables upon `sudo` invocation - a priviledge escalation *required* to create Singularity containers via the `build` command.

Remotely Bootstrapped and Built Containers

Consider again *the definition file used the outset of the section above*:

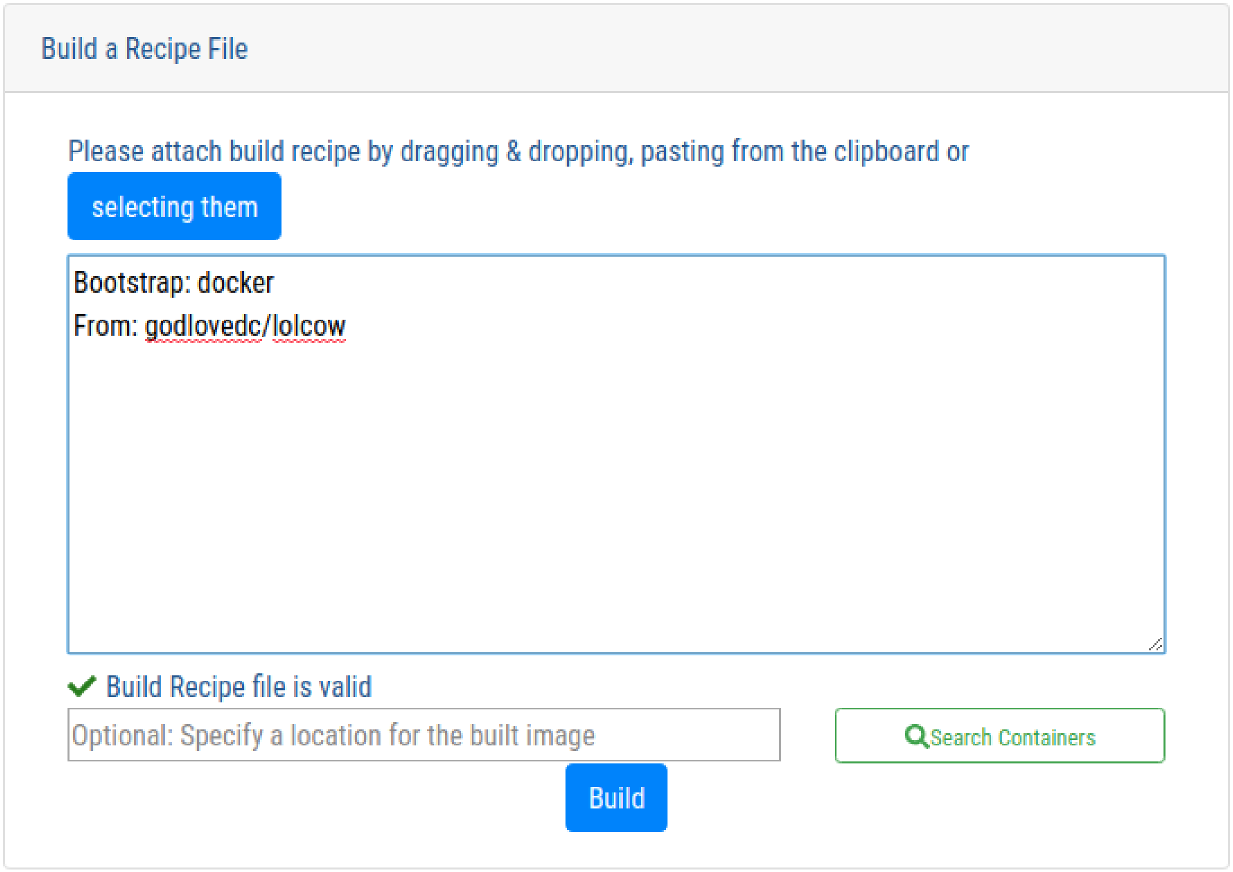
```
Bootstrap: docker
From: godlovedc/lolcow
```

With two small adjustments to the Singularity `build` command, the Sylabs Cloud Remote Builder can be utilized:

```
$ singularity build --remote lolcow_rb_def.sif lolcow.def
searching for available build agent......INFO:    Starting build...
Getting image source signatures
Copying blob sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
 45.33 MiB / 45.33 MiB  0s
Copying blob sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
 848 B / 848 B  0s
Copying blob sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
 621 B / 621 B  0s
Copying blob sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
 853 B / 853 B  0s
Copying blob sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
 169 B / 169 B  0s
Copying blob sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
 53.75 MiB / 53.75 MiB  0s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB  0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: /tmp/image-994007654
INFO:    Now uploading /tmp/image-994007654 to the library
 87.94 MiB / 87.94 MiB  100.00% 41.76 MiB/s 2s
INFO:    Setting tag latest
 87.94 MiB / 87.94 MiB␣
↪[===============================================================================]␣
↪100.00% 19.08 MiB/s 4s
```

In the above, `--remote` has been added as the `build` option that causes use of the Remote Builder service. A much more subtle change, however, is the *absence* of `sudo` ahead of `singularity build`. Though subtle here, this absence is notable, as users can build containers via the Remote Builder with *escalated privileges*; in other words, steps in container creation that *require* `root` access *are* enabled via the Remote Builder even for (DevOps) users *without* admninistrative privileges locally.

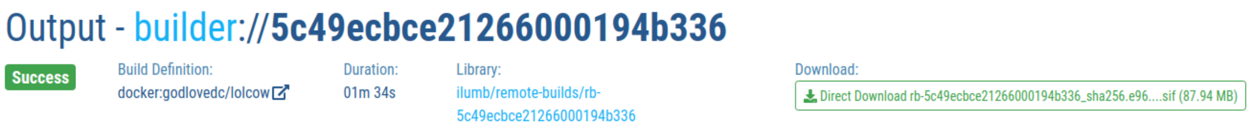
In addition to the command-line support described above, the Sylabs Cloud Remote Builder also allows definition files to be copied and pasted into its Graphical User Interface (GUI). After pasting a definition file, and having that file validated by the service, the build-centric part of the GUI appears as illustrated below. By clicking on the `Build` button, creation of the container is initiated.


Build a Recipe File
Please attach build recipe by dragging & dropping, pasting from the clipboard or
selecting them

```
Bootstrap: docker
From: godlovedc/lolcow
```

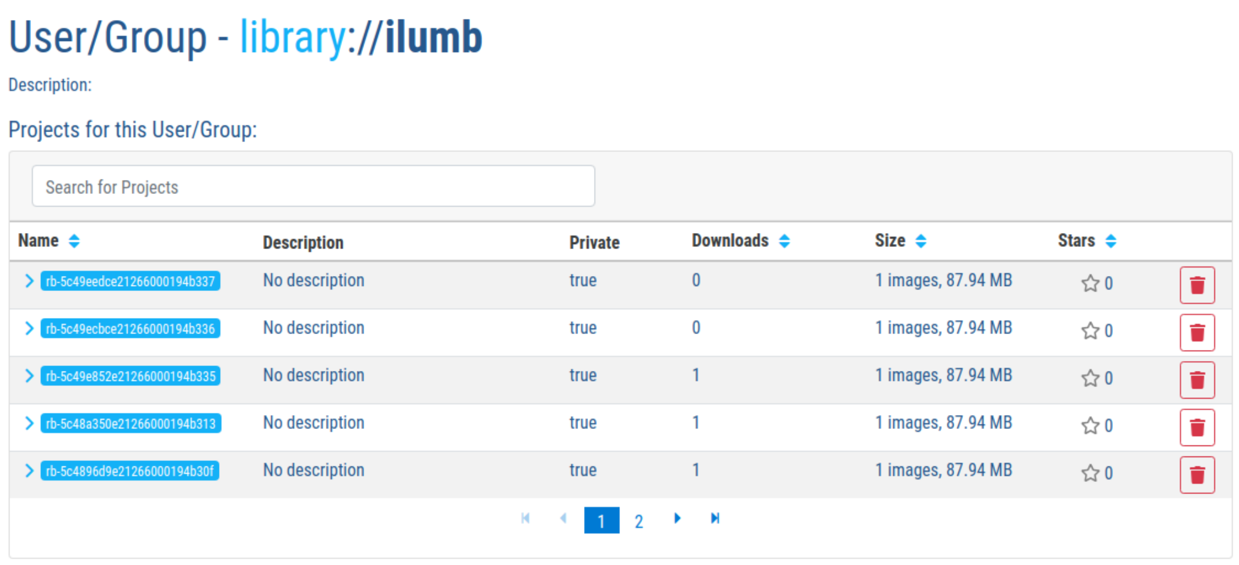
✓ Build Recipe file is valid
Optional: Specify a location for the built image
Search Containers
Build

Once the build process has been completed, the corresponding SIF file can be retrieved from the service - as shown below. A log file for the `build` process is provided by the GUI, and made available for download as a text file (not shown here).


Output - builder://5c49ecbce21266000194b336

Success	Build Definition: docker:godlovedc/lolcow	Duration: 01m 34s	Library: ilumb/remote-builds/rb-5c49ecbce21266000194b336	Download: Direct Download rb-5c49ecbce21266000194b336_sha256.e96....sif (87.94 MB)

A copy of the SIF file created by the service remains in the Sylabs Cloud Singularity Library as illustrated below.


User/Group - library://ilumb

Description:

Projects for this User/Group:

Search for Projects

Name	Description	Private	Downloads	Size	Stars
rb-5c49eedce21266000194b337	No description	true	0	1 images, 87.94 MB	0
rb-5c49ecbce21266000194b336	No description	true	0	1 images, 87.94 MB	0
rb-5c49e852e21266000194b335	No description	true	1	1 images, 87.94 MB	0
rb-5c48a350e21266000194b313	No description	true	1	1 images, 87.94 MB	0
rb-5c4896d9e21266000194b30f	No description	true	1	1 images, 87.94 MB	0

1 2

Note: The Sylabs Cloud is currently available as an Alpha Preview. In addition to the Singularity Library and Remote Builder, a Keystore service is also available. All three services make use of a *freemium* pricing model in supporting Singularity Community Edition. In contrast, all three services are included in SingularityPRO - an enterprise grade subscription for Singularity that is offered for a fee from Sylabs. For addtional details regarding the different offerings available for Singularity, please consult the Sylabs website.

Mandatory Header Keywords: Locally Boostrapped

When `docker-daemon` is the bootstrap agent in a Singularity definition file, SIF containers can be created from images cached locally by Docker. Suppose the definition file `lolcow-d.def` has contents:

```
Bootstrap: docker-daemon
From: godlovedc/lolcow:latest
```

Note: Again, the image tag `latest` is **required** when bootstrapping creation of a container for Singularity from an image locally cached by Docker.

Then,

```
$ sudo singularity build lolcow_from_docker_cache.sif lolcow-d.def
Build target already exists. Do you want to overwrite? [N/y] y
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:a2022691bf950a72f9d2d84d557183cb9eee07c065a76485f1695784855c5193
 119.83 MiB / 119.83 MiB [==================================================] 6s
Copying blob sha256:ae620432889d2553535199dbdd8ba5a264ce85fcdcd5a430974d81fc27c02b45
 15.50 KiB / 15.50 KiB [====================================================] 0s
Copying blob sha256:c561538251751e3685c7c6e7479d488745455ad7f84e842019dcb452c7b6fecc
 14.50 KiB / 14.50 KiB [====================================================] 0s
Copying blob sha256:f96e6b25195f1b36ad02598b5d4381e41997c93ce6170cab1b81d9c68c514db0
 5.50 KiB / 5.50 KiB [======================================================] 0s
Copying blob sha256:7f7a065d245a6501a782bf674f4d7e9d0a62fa6bd212edbf1f17bad0d5cd0bfc
```

(continues on next page)

(continued from previous page)

```
 3.00 KiB / 3.00 KiB [======================================================] 0s
Copying blob sha256:70ca7d49f8e9c44705431e3dade0636a2156300ae646ff4f09c904c138728839
 116.56 MiB / 116.56 MiB [==================================================] 6s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_from_docker_cache.sif
```

In other words, this is the definition-file counterpart to *the command-line invocation provided above*.

Note: The `sudo` requirement in the above `build` request originates from Singularity; it is the standard requirement when use is made of definition files. In other words, membership of the issuing user in the `docker` Linux group is of no consequence in this context.

Alternatively when `docker-archive` is the bootstrap agent in a Singularity definition file, SIF containers can be created from images stored locally by Docker. Suppose the definition file `lolcow-da.def` has contents:

```
Bootstrap: docker-archive
From: lolcow.tar
```

Then,

```
$ sudo singularity build lolcow_tar_def.sif lolcow-da.def
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:a2022691bf950a72f9d2d84d557183cb9eee07c065a76485f1695784855c5193
 119.83 MiB / 119.83 MiB [==================================================] 6s
Copying blob sha256:ae620432889d2553535199dbdd8ba5a264ce85fcdcd5a430974d81fc27c02b45
 15.50 KiB / 15.50 KiB [====================================================] 0s
Copying blob sha256:c561538251751e3685c7c6e7479d488745455ad7f84e842019dcb452c7b6fecc
 14.50 KiB / 14.50 KiB [====================================================] 0s
Copying blob sha256:f96e6b25195f1b36ad02598b5d4381e41997c93ce6170cab1b81d9c68c514db0
 5.50 KiB / 5.50 KiB [======================================================] 0s
Copying blob sha256:7f7a065d245a6501a782bf674f4d7e9d0a62fa6bd212edbf1f17bad0d5cd0bfc
 3.00 KiB / 3.00 KiB [======================================================] 0s
Copying blob sha256:70ca7d49f8e9c44705431e3dade0636a2156300ae646ff4f09c904c138728839
 116.56 MiB / 116.56 MiB [==================================================] 6s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_tar_def.sif
```

through `build` results in the SIF file `lolcow_tar_def.sif`. In other words, this is the definition-file counterpart to *the command-line invocation provided above* .

Optional Header Keywords

In the two-previous examples, the `From` keyword specifies *both* the `user` and `repo-name` in making use of Docker Hub. *Optional* use of `Namespace` permits the more-granular split across two keywords:

```
Bootstrap: docker
Namespace: godlovedc
From: lolcow
```

Note: In their documentation, "Docker ID namespace" and `user` are employed as synonyms in the text and examples, respectively.

Note: The default value for the optional keyword `Namespace` is `library`.

Private Images and Registries

Thus far, use of Docker Hub has been assumed. To make use of a different repository of Docker images the **optional** `Registry` keyword can be added to the Singularity definition file. For example, to make use of a Docker image from the NVIDIA GPU Cloud (NGC) corresponding definition file is:

```
Bootstrap: docker
From: nvidia/pytorch:18.11-py3
Registry: nvcr.io
```

This def file `ngc_pytorch.def` can be passed as a specification to `build` as follows:

```
$ sudo singularity build --docker-login mypytorch.sif ngc_pytorch.def
Enter Docker Username: $oauthtoken
Enter Docker Password: <obscured>
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:18d680d616571900d78ee1c8fff0310f2a2afe39c6ed0ba2651ff667af406c3e
 41.34 MiB / 41.34 MiB [==================================================] 2s
<blob copying details deleted>
Copying config sha256:b77551af8073c85588088ab2a39007d04bc830831ba1eef4127b2d39aaf3a6b1
21.28 KiB / 21.28 KiB [==================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: mypytorch.sif
```

After successful authentication via interactive use of the `--docker-login` option, output as the SIF container `mypytorch.sif` is (ultimately) produced. As above, *use of environment variables* is another option available for authenticating private Docker type repositories such as NGC; once set, the `build` command is as above save for the absence of the `--docker-login` option.

Directing Execution

The `Dockerfile` corresponding to `godlovedc/lolcow` (and available here) is as follows:

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y fortune cowsay lolcat

ENV PATH /usr/games:${PATH}
ENV LC_ALL=C

ENTRYPOINT fortune | cowsay | lolcat
```

The execution-specific part of this `Dockerfile` is the `ENTRYPOINT` - "... an optional definition for the first part of the command to be run ..." according to the available documentation. After conversion to SIF, execution of `fortune | cowsay | lolcat` *within* the container produces the output:

```
$ ./mylolcow.sif
 ______________________________________
/ Q: How did you get into artificial   \
| intelligence? A: Seemed logical -- I |
\ didn't have any real intelligence.   /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

In addition, `CMD` allows an arbitrary string to be *appended* to the `ENTRYPOINT`. Thus, multiple commands or flags can be passed together through combined use.

Suppose now that a Singularity `%runscript` **section** is added to the definition file as follows:

```
Bootstrap: docker
Namespace: godlovedc
From: lolcow

%runscript

    fortune
```

After conversion to SIF via the Singularity `build` command, exection of the resulting container produces the output:

```
$ ./lolcow.sif
This was the most unkindest cut of all.
        -- William Shakespeare, "Julius Caesar"
```

In other words, introduction of a `%runscript` section into the Singularity definition file causes the `ENTRYPOINT` of the `Dockerfile` to be *bypassed*. The presence of the `%runscript` section would also bypass a `CMD` entry in the `Dockerfile`.

To *preserve* use of `ENTRYPOINT` and/or `CMD` as defined in the `Dockerfile`, the `%runscript` section must be *absent* from the Singularity definition. In this case, and to favor execution of `CMD` *over* `ENTRYPOINT`, a non-empty assignment of the *optional* `IncludeCmd` should be included in the header section of the Singularity definition file as follows:

```
Bootstrap: docker
Namespace: godlovedc
From: lolcow
IncludeCmd: yes
```

Note: Because only a non-empty `IncludeCmd` is required, *either* `yes` (as above) or `no` results in execution of `CMD` *over* `ENTRYPOINT`.

To summarize execution precedence:

1. If present, the `%runscript` section of the Singularity definition file is executed
2. If `IncludeCmd` is a non-empty keyword entry in the header of the Singularity definition file, then `CMD` from the `Dockerfile` is executed
3. If present in the `Dockerfile`, `ENTRYPOINT` appended by `CMD` (if present) are executed in sequence
4. Execution of the `bash` shell is defaulted to

Container Metadata

Singularity's `inspect` command displays container metadata - data about data that is encapsulated *within* a SIF file. Default output (assumed via the `--labels` option) from the command was *illustrated above*. `inspect`, however, provides a number of options that are *detailed elsewhere*; in the remainder of this section, Docker-specific use to establish execution precedence is emphasized.

As stated above (i.e., *the first case of execution precedence*), the very existence of a `%runscript` section in a Singularity definition file *takes precedence* over commands that might exist in the `Dockerfile`.

When the `%runscript` section is *removed* from the Singularity definition file, the result is (once again):

```
$ singularity inspect --deffile lolcow.sif

from: lolcow
bootstrap: docker
namespace: godlovedc
```

The runscript 'inherited' from the `Dockerfile` is:

```
$ singularity inspect --runscript lolcow.sif

#!/bin/sh
OCI_ENTRYPOINT='"/bin/sh" "-c" "fortune | cowsay | lolcat"'
OCI_CMD=''
# ENTRYPOINT only - run entrypoint plus args
if [ -z "$OCI_CMD" ] && [ -n "$OCI_ENTRYPOINT" ]; then
    SINGULARITY_OCI_RUN="${OCI_ENTRYPOINT} $@"
fi

# CMD only - run CMD or override with args
if [ -n "$OCI_CMD" ] && [ -z "$OCI_ENTRYPOINT" ]; then
    if [ $# -gt 0 ]; then
        SINGULARITY_OCI_RUN="$@"
    else
        SINGULARITY_OCI_RUN="${OCI_CMD}"
    fi
```

(continues on next page)

(continued from previous page)

```
fi

# ENTRYPOINT and CMD - run ENTRYPOINT with CMD as default args
# override with user provided args
if [ $# -gt 0 ]; then
    SINGULARITY_OCI_RUN="${OCI_ENTRYPOINT} $@"
else
    SINGULARITY_OCI_RUN="${OCI_ENTRYPOINT} ${OCI_CMD}"
fi

eval ${SINGULARITY_OCI_RUN}
```

From this Bourne shell script, it is evident that only an `ENTRYPOINT` is detailed in the `Dockerfile`; thus the `ENTRYPOINT only - run entrypoint plus args` conditional block is executed. In this case then, *the third case of execution precedence* has been illustrated.

The above Bourne shell script also illustrates how the following scenarios will be handled:

- A `CMD` only entry in the `Dockerfile`
- **Both** `ENTRYPOINT` *and* `CMD` entries in the `Dockerfile`

From this level of detail, use of `ENTRYPOINT` *and/or* `CMD` in a Dockerfile has been made **explicit**. These remain examples within *the third case of execution precedence*.

2.4.7 OCI Image Support

Overview

OCI is an acronym for the Open Containers Initiative - an independent organization whose mandate is to develop open standards relating to containerization. To date, standardization efforts have focused on container formats and runtimes; it is the former that is emphasized here. Stated simply, an **OCI blob** is content that can be addressed; in other words, *each* layer of a Docker image is rendered as an OCI blob as illustrated in the (revisited) `pull` example below.

Note: To facilitate interoperation with Docker Hub, the Singularity core makes use of the `containers/image` library - "... a set of Go libraries aimed at working in various way[s] with containers' images and container image registries."

Image Pulls Revisited

After describing various *action commands that could be applied to images hosted remotely via Docker Hub*, the notion of having *a local copy in Singularity's native format for containerization (SIF)* was introduced:

```
$ singularity pull docker://godlovedc/lolcow
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
 45.33 MiB / 45.33 MiB [====================================================] 1s
Copying blob sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
 848 B / 848 B [============================================================] 0s
Copying blob sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
 621 B / 621 B [============================================================] 0s
```

(continues on next page)

(continued from previous page)

```
Copying blob sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
 853 B / 853 B [============================================================] 0s
Copying blob sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
 169 B / 169 B [============================================================] 0s
Copying blob sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
 53.75 MiB / 53.75 MiB [====================================================] 2s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_latest.sif
```

Thus use of Singularity's `pull` command results in the *local* file copy in SIF, namely `lolcow_latest.sif`. Layers of the image from Docker Hub are copied locally as OCI blobs.

Image Caching in Singularity

If the *same* `pull` command is issued a *second* time, the output is different:

```
$ singularity pull docker://godlovedc/lolcow
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
Skipping fetch of repeat blob␣
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
Skipping fetch of repeat blob␣
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
Skipping fetch of repeat blob␣
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
Skipping fetch of repeat blob␣
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
Skipping fetch of repeat blob␣
↪sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_latest.sif
```

As the copy operation has clearly been *skipped*, it is evident that a copy of all OCI blobs **must** be cached locally. Indeed, Singularity has made an entry in its local cache as follows:

```
$ tree .singularity/
.singularity/
└── cache
    └── oci
        ├── blobs
        │   └── sha256
        │       ├── 3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
        │       ├── 73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
        │       ├── 7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
        │       ├── 8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
```

(continues on next page)

(continued from previous page)

```
        │           ├── 9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
        │           ├── 9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
        │           ├── d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
        │           └── f2a852991b0a36a9f3d6b2a33b98a461e9ede8393482f0deb5287afcbae2ce10
        ├── index.json
        └── oci-layout

4 directories, 10 files
```

Compliance with the OCI Image Layout Specification

From the perspective of the directory `$HOME/.singularity/cache/oci`, this cache implementation in Singularity complies with the OCI Image Layout Specification:

- `blobs` directory - contains content addressable data, that is otherwise considered opaque
- `oci-layout` file - a mandatory JSON object file containing both mandatory and optional content
- `index.json` file - a mandatory JSON object file containing an index of the images

Because one or more images is 'bundled' here, the directory `$HOME/.singularity/cache/oci` is referred to as the `$OCI_BUNDLE_DIR`.

For additional details regarding this specification, consult the OCI Image Format Specification.

OCI Compliance and the Singularity Cache

As required by the layout specification, OCI blobs are *uniquely* named by their contents:

```
$ shasum -a 256 ./blobs/sha256/
↪9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118  ./blobs/sha256/
↪9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
```

They are also otherwise opaque:

```
$ file ./blobs/sha256/
↪9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118 ./blobs/sha256/
↪9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118: gzip compressed␣
↪data
```

The content of the `oci-layout` file in this example is:

```
$ cat oci-layout | jq
{
  "imageLayoutVersion": "1.0.0"
}
```

This is as required for compliance with the layout standard.

Note: In rendering the above JSON object files, use has been made of `jq` - the command-line JSON processor.

The index of images in this case is:

```
$ cat index.json | jq
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest":
↪"sha256:f2a852991b0a36a9f3d6b2a33b98a461e9ede8393482f0deb5287afcbae2ce10",
      "size": 1125,
      "annotations": {
        "org.opencontainers.image.ref.name":
↪"a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb"
      },
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ]
}
```

The `digest` blob in this index file includes the details for all of the blobs that collectively comprise the `godlovedc/lolcow` image:

```
$ cat  ./blobs/sha256/
↪f2a852991b0a36a9f3d6b2a33b98a461e9ede8393482f0deb5287afcbae2ce10 | jq
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "digest": "sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
↪",
    "size": 3410
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
↪"sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118",
      "size": 47536248
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
↪"sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a",
      "size": 848
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
↪"sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2",
      "size": 621
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
↪"sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e",
```

(continues on next page)

(continued from previous page)

```
      "size": 853
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
↪"sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9",
      "size": 169
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
↪"sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945",
      "size": 56355961
    }
  ]
}
```

The digest blob referenced in the index.json file references the following configuration file:

```
$ cat ./blobs/sha256/73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82␣
↪| jq
{
  "created": "2017-09-21T18:37:47.278336798Z",
  "architecture": "amd64",
  "os": "linux",
  "config": {
    "Env": [
      "PATH=/usr/games:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "LC_ALL=C"
    ],
    "Entrypoint": [
      "/bin/sh",
      "-c",
      "fortune | cowsay | lolcat"
    ]
  },
  "rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:a2022691bf950a72f9d2d84d557183cb9eee07c065a76485f1695784855c5193",
      "sha256:ae620432889d2553535199dbdd8ba5a264ce85fcdcd5a430974d81fc27c02b45",
      "sha256:c561538251751e3685c7c6e7479d488745455ad7f84e842019dcb452c7b6fecc",
      "sha256:f96e6b25195f1b36ad02598b5d4381e41997c93ce6170cab1b81d9c68c514db0",
      "sha256:7f7a065d245a6501a782bf674f4d7e9d0a62fa6bd212edbf1f17bad0d5cd0bfc",
      "sha256:70ca7d49f8e9c44705431e3dade0636a2156300ae646ff4f09c904c138728839"
    ]
  },
  "history": [
    {
      "created": "2017-09-18T23:31:37.453092323Z",
      "created_by": "/bin/sh -c #(nop) ADD␣
↪file:5ed435208da6621b45db657dd6549ee132cde58c4b6763920030794c2f31fbc0 in / "
    },
    {
      "created": "2017-09-18T23:31:38.196268404Z",
      "created_by": "/bin/sh -c set -xe \t\t&& echo '#!/bin/sh' > /usr/sbin/policy-rc.
↪d \t&& echo 'exit 101' >> /usr/sbin/policy-rc.d \t&& chmod +x /usr/sbin/policy-rc.d␣
↪\t\t&& dpkg-divert --local --rename --add /sbin/initctl \t&& cp -a /usr/sbin/policy-
↪rc.d /sbin/initctl \t&& sed -i 's/^exit.*/exit 0/' /sbin/initctl \t\t&& echo 'force-
↪unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \t\t&& echo 'DPkg::Post-Invoke
↪{ \"rm -f /var/cache/apt/archives/*.deb /var/cache/apt/archives/partial/*.deb /var/
↪cache/apt/*.bin || true\"; };' > /etc/apt/apt.conf.d/docker-clean \t&& echo
↪'APT::Update::Post-Invoke { \"rm -f /var/cache/apt/archives/*.deb /var/cache/apt/
↪archives/partial/*.deb /var/cache/apt/*.bin || true\"; };' >> /etc/apt/apt.conf.d/
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "created": "2017-09-18T23:31:38.788043199Z",
      "created_by": "/bin/sh -c rm -rf /var/lib/apt/lists/*"
    },
    {
      "created": "2017-09-18T23:31:39.411670721Z",
      "created_by": "/bin/sh -c sed -i 's/^#\\s*\\(deb.*universe\\)$/\\1/g' /etc/apt/
↪sources.list"
    },
    {
      "created": "2017-09-18T23:31:40.055188541Z",
      "created_by": "/bin/sh -c mkdir -p /run/systemd && echo 'docker' > /run/systemd/
↪container"
    },
    {
      "created": "2017-09-18T23:31:40.215057796Z",
      "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/bash\"]",
      "empty_layer": true
    },
    {
      "created": "2017-09-21T18:37:46.483638061Z",
      "created_by": "/bin/sh -c apt-get update && apt-get install -y fortune cowsay␣
↪lolcat"
    },
    {
      "created": "2017-09-21T18:37:47.041333952Z",
      "created_by": "/bin/sh -c #(nop)  ENV PATH=/usr/games:/usr/local/sbin:/usr/
↪local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "empty_layer": true
    },
    {
      "created": "2017-09-21T18:37:47.170535967Z",
      "created_by": "/bin/sh -c #(nop)  ENV LC_ALL=C",
      "empty_layer": true
    },
    {
      "created": "2017-09-21T18:37:47.278336798Z",
      "created_by": "/bin/sh -c #(nop)  ENTRYPOINT [\"/bin/sh\" \"-c\" \"fortune |␣
↪cowsay | lolcat\"]",
      "empty_layer": true
    }
  ]
}
```

Even when all OCI blobs are already in Singularity's local cache, repeated image pulls cause *both* these last-two JSON object files, as well as the `oci-layout` and `index.json` files, to be updated.

Building Containers for Singularity from OCI Images

Working Locally from the Singularity Command Line: `oci` Boostrap Agent

The example detailed in the previous section can be used to illustrate how a SIF file for use by Singularity can be created from the local cache - an albeit contrived example, that works because the Singularity cache is compliant with the OCI Image Layout Specification.

Note: Of course, the `oci` bootstrap agent can be applied to *any* **bundle** that is compliant with the OCI Image Layout Specification - not *just* the Singularity cache, as created by executing a Singularity `pull` command.

In this local case, the `build` command of Singularity makes use of the `oci` boostrap agent as follows:

```
$ singularity build ~/lolcow_oci_cache.sif oci://$HOME/.singularity/cache/
↪oci:a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
Skipping fetch of repeat blob␣
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
Skipping fetch of repeat blob␣
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
Skipping fetch of repeat blob␣
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
Skipping fetch of repeat blob␣
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
Skipping fetch of repeat blob␣
↪sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: /home/vagrant/lolcow_oci_cache.sif
```

As can be seen, this results in the SIF file `lolcow_oci_cache.sif` in the user's home directory.

The syntax for the `oci` boostrap agent requires some elaboration, however. In this case, and as illustrated above, `$HOME/.singularity/cache/oci` has content:

```
$ ls
blobs  index.json  oci-layout
```

In other words, it is the `$OCI_BUNDLE_DIR` containing the data and metadata that collectively comprise the image layed out in accordance with the OCI Image Layout Specification *as discussed previously* - the same data and metadata that are assembled into a single SIF file through the `build` process. However,

```
$ singularity build ~/lolcow_oci_cache.sif oci://$HOME/.singularity/cache/oci
INFO:    Starting build...
FATAL:   While performing build: conveyor failed to get: more than one image in oci,␣
↪choose an image
```

does not *uniquely* specify an image from which to bootstrap the `build` process. In other words, there are multiple images referenced via `org.opencontainers.image.ref.name` in the `index.json` file. By appending

:a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb to oci in this example, the image is uniquely specified, and the container created in SIF (as illustrated previously).

Note: Executing the Singularity pull command multiple times on the same image produces multiple org.opencontainers.image.ref.name entries in the index.json file. Appending the value of the unique org.opencontainers.image.ref.name allows for use of the oci boostrap agent.

Working Locally from the Singularity Command Line: oci-archive Boostrap Agent

OCI archives, i.e., tar files obeying the OCI Image Layout Specification *as discussed previously*, can seed creation of a container for Singularity. In this case, use is made of the oci-archive bootstrap agent.

To illustrate this agent, it is convenient to build the archive from the Singularity cache. After a single pull of the godlovedc/lolcow image from Docker Hub, a tar format archive can be generated from the $HOME/.singularity/cache/oci directory as follows:

```
$ tar cvf $HOME/godlovedc_lolcow.tar *
blobs/
blobs/sha256/
blobs/sha256/73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
blobs/sha256/8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
blobs/sha256/9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
blobs/sha256/3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
blobs/sha256/9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
blobs/sha256/d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
blobs/sha256/f2a852991b0a36a9f3d6b2a33b98a461e9ede8393482f0deb5287afcbae2ce10
blobs/sha256/7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
index.json
oci-layout
```

The native container lolcow_oci_tarfile.sif for use by Singularity can be created by issuing the build command as follows:

```
$ singularity build lolcow_oci_tarfile.sif oci-archive://godlovedc_lolcow.tar
Build target already exists. Do you want to overwrite? [N/y] y
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
Skipping fetch of repeat blob␣
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
Skipping fetch of repeat blob␣
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
Skipping fetch of repeat blob␣
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
Skipping fetch of repeat blob␣
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
Skipping fetch of repeat blob␣
↪sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_oci_tarfile.sif
```

This assumes that the `tar` file exists in the current working directory.

Note: Cache maintenance is a manual process at the current time. In other words, the cache can be cleared by **carefully** issuing the command `rm -rf $HOME/.singularity/cache`. Of course, this will clear the local cache of all downloaded images.

Note: Because the layers of a Docker image as well as the blobs of an OCI image are already `gzip` compressed, there is a minimal advantage to having compressed archives representing OCI images. For this reason, the `build` detailed above boostraps a SIF file for use by Singularity from only a `tar` file, and not a `tar.gz` file.

Working from the Singularity Command Line with Remotely Hosted Images

In the previous section, an OCI archive was created from locally available OCI blobs and metadata; the resulting `tar` file served to bootstrap the creation of a container for Singularity in SIF via the `oci-archive` agent. Typically, however, OCI archives of interest are remotely hosted. Consider, for example, an Alpine Linux OCI archive stored in Amazon S3 storage. Because such an archive can be retrieved via secure HTTP, the following `pull` command results in a local copy as follows:

```
$ singularity pull https://s3.amazonaws.com/singularity-ci-public/alpine-oci-archive.
↪tar
 1.98 MiB / 1.98 MiB␣
↪[==================================================================================]␣
↪100.00% 7.48 MiB/s 0s
```

Thus `https` (and `http`) are additional bootstrap agents available to seed development of containers for Singularity.

It is worth noting that the OCI image specfication compliant contents of this archive are:

```
$ tar tvf alpine-oci-archive.tar
drwxr-xr-x 1000/1000         0 2018-06-25 14:45 blobs/
drwxr-xr-x 1000/1000         0 2018-06-25 14:45 blobs/sha256/
-rw-r--r-- 1000/1000       585 2018-06-25 14:45 blobs/sha256/
↪b1a7f144ece0194921befe57ab30ed1fd98c5950db7996719429020986092058
-rw-r--r-- 1000/1000       348 2018-06-25 14:45 blobs/sha256/
↪d0ff39a54244ba25ac7447f19941765bee97b05f37ceb438a72e80c9ed39854a
-rw-r--r-- 1000/1000   2065537 2018-06-25 14:45 blobs/sha256/
↪ff3a5c916c92643ff77519ffa742d3ec61b7f591b6b7504599d95a4a41134e28
-rw-r--r-- 1000/1000       296 2018-06-25 14:45 index.json
-rw-r--r-- 1000/1000        31 2018-06-25 14:45 oci-layout
```

Proceeding as before, for a (now) locally available OCI archive, a SIF file can be produced by executing:

```
$ singularity build alpine_oci_archive.sif oci-archive://alpine-oci-archive.tar
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:ff3a5c916c92643ff77519ffa742d3ec61b7f591b6b7504599d95a4a41134e28
 1.97 MiB / 1.97 MiB [=====================================================] 0s
Copying config sha256:b1a7f144ece0194921befe57ab30ed1fd98c5950db7996719429020986092058
 585 B / 585 B [===========================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: alpine_oci_archive.sif
```

The resulting SIF file can be validated as follows, for example:

```
$ ./alpine_oci_archive.sif
Singularity> cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.7.0
PRETTY_NAME="Alpine Linux v3.7"
HOME_URL="http://alpinelinux.org"
BUG_REPORT_URL="http://bugs.alpinelinux.org"
Singularity>
$
```

Note: The `http` and `https` bootstrap agents can only be used to `pull` OCI archives from where they are hosted.

In working with remotely hosted OCI image archives then, a two-step workflow is *required* to produce SIF files for native use by Singularity:

1. Transfer of the image to local storage via the `https` (or `http`) bootstrap agent. The Singularity `pull` command achieves this.
2. Creation of a SIF file via the `oci-archive` bootstrap agent. The Singularity `build` command achieves this.

Note: Though a frequently asked question, the distribution of OCI images remains out of scope. In other words, there is no OCI endorsed distribution method or registry.

Established with nothing more than a Web server then, any individual, group or organization, *could* host OCI archives. This might be particularly appealing, for example, for organizations having security requirements that preclude access to public registries such as Docker Hub. Other that having a very basic hosting capability, OCI archives need only comply to the OCI Image Layout Specification *as discussed previously*.

Working with Definition Files: Mandatory Header Keywords

Three, new bootstrap agents have been introduced as a consequence of compliance with the OCI Image Specification - assuming `http` and `https` are considered together. In addition to bootstrapping images for Singularity completely from the command line, definition files can be employed.

As above, the OCI image layout compliant Singularity cache can be employed to create SIF containers; the definition file, `lolcow-oci.def`, equivalent is:

```
Bootstrap: oci
From: .singularity/cache/
↪oci:a692b57abc43035b197b10390ea2c12855d21649f2ea2cc28094d18b93360eeb
```

Recall that the colon-appended string in this file uniquely specifies the `org.opencontainers.image.ref.name` of the desired image, as more than one possibility exists in the `index.json` file. The corresponding `build` command is:

```
$ sudo singularity build ~/lolcow_oci_cache.sif lolcow-oci.def
WARNING: Authentication token file not found : Only pulls of public images will␣
↪succeed
Build target already exists. Do you want to overwrite? [N/y] y
INFO:    Starting build...
```

(continues on next page)

(continued from previous page)

```
Getting image source signatures
Copying blob sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
 45.33 MiB / 45.33 MiB [====================================================] 0s
Copying blob sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
 848 B / 848 B [============================================================] 0s
Copying blob sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
 621 B / 621 B [============================================================] 0s
Copying blob sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
 853 B / 853 B [============================================================] 0s
Copying blob sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
 169 B / 169 B [============================================================] 0s
Copying blob sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
 53.75 MiB / 53.75 MiB [====================================================] 0s
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: /home/vagrant/lolcow_oci_cache.sif
```

Required use of `sudo` allows Singularity to `build` the SIF container `lolcow_oci_cache.sif`.

When it comes to OCI archives, the definition file, `lolcow-ocia.def` corresponding to the command-line invocation above is:

```
Bootstrap: oci-archive
From: godlovedc_lolcow.tar
```

Applying `build` as follows

```
$ sudo singularity build lolcow_oci_tarfile.sif lolcow-ocia.def
WARNING: Authentication token file not found : Only pulls of public images will␣
↪succeed
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob␣
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
Skipping fetch of repeat blob␣
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a
Skipping fetch of repeat blob␣
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
Skipping fetch of repeat blob␣
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
Skipping fetch of repeat blob␣
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9
Skipping fetch of repeat blob␣
↪sha256:8e860504ff1ee5dc7953672d128ce1e4aa4d8e3716eb39fe710b849c64b20945
Copying config sha256:73d5b1025fbfa138f2cacf45bbf3f61f7de891559fa25b28ab365c7d9c3cbd82
 3.33 KiB / 3.33 KiB [======================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: lolcow_oci_tarfile.sif
```

results in the SIF container `lolcow_oci_tarfile.sif`.

Working with Definition Files: Additonal Considerations

In working with definition files, the following additional considerations arise:

- In addition to the mandatory header keywords documented above, *optional header keywords* are possible additions to OCI bundle and/or archive bootstrap definition files.
- As distribution of OCI bundles and/or archives is out of the Initiative's scope, so is the authentication required to access private images and/or registries.
- The direction of execution follows along the same lines *as described above*. Of course, the SIF container's metadata will make clear the `runscript` through application of the `inspect` command *as described previously*.
- Container metadata will also reveal whether or not a given SIF file was bootstrapped from an OCI bundle or archive; for example, below it is evident that an OCI archive was employed to bootstrap creation of the SIF file:

```
$ singularity inspect --labels lolcow_oci_tarfile.sif | jq
{
  "org.label-schema.build-date": "Sunday_27_January_2019_0:5:29_UTC",
  "org.label-schema.schema-version": "1.0",
  "org.label-schema.usage.singularity.deffile.bootstrap": "oci-archive",
  "org.label-schema.usage.singularity.deffile.from": "godlovedc_lolcow.tar",
  "org.label-schema.usage.singularity.version": "3.0.3-1"
}
```

2.4.8 Container Caching

To avoid fetching duplicate docker or OCI layers every time you want to `run`, `exec` etc. a `docker://` or `oci://` container directly, Singularity keeps a cache of layer files. The SIF format container that Singularity creates from these layers is also cached. This means that re-running a docker container, e.g. `singularity run docker://alpine` is much faster until the upstream image changes in docker hub, and a new SIF must be built from updated layers.

By default the cache directory is `.singularity/cache` in your `$HOME` directory. You can modify the cache directory by setting the `SINGULARITY_CACHEDIR` environment variable. To disable caching altogether, set the `SINGULARITY_DISABLE_CACHE` environment variable.

The `singularity cache` command can be used to see the content of your cache dir, and clean the cache if needed:

```
$ singularity cache list
There are 10 container file(s) using 4.75 GB and 78 oci blob file(s) using 5.03 GB of␣
↪space
Total space used: 9.78 GB

$ singularity cache clean
This will delete everything in your cache (containers from all sources and OCI blobs).
Hint: You can see exactly what would be deleted by canceling and using the --dry-run␣
↪option.
Do you want to continue? [N/y] y
Removing /home/dave/.singularity/cache/library
Removing /home/dave/.singularity/cache/oci-tmp
Removing /home/dave/.singularity/cache/shub
Removing /home/dave/.singularity/cache/oci
Removing /home/dave/.singularity/cache/net
Removing /home/dave/.singularity/cache/oras
```

For a more complete guide to caching and the `cache` command, see the *Build Environment* page.

2.4.9 Best Practices

Singularity can make use of most Docker and OCI images without complication. However, there exist known cases where complications can arise. Thus a brief compilation of best practices follows below.

1. Accounting for trust

Docker containers *allow for* privilege escalation. In a `Dockerfile`, for example, the `USER` instruction allows for user and/or group settings to be made in the Linux operating environment. The trust model in Singularity is completely different: Singularity allows untrusted users to run untrusted containers in a trusted way. Because Singularity containers embodied as SIF files execute in *user* space, there is no possibility for privilege escalation. In other words, those familiar with Docker, should *not* expect access to elevated user permissions; and as a corollary, use of the `USER` instruction must be *avoided*.

Singularity does, however, allow for fine-grained control over the permissions that containers require for execution. Given that Singularilty executes in user space, it is not surprising that permissions need to be externally established *for* the container through use of the `capability` command. *Detailed elsewhere in this documentation*, Singularity allows users and/or groups to be granted/revoked authorized capabilties. Owing to Singularity's trust model, this fundamental best practice can be stated as follows:

> "Employ `singularity capability` to manage execution privileges for containers"

2. Maintaining containers built from Docker and OCI images

SIF files created by boostrapping from Docker or OCI images are, of course, only as current as the most recent Singularity `pull`. Subsequent retrievals *may* result in containers that are built and/or behave differently, owing to changes in the corresponding `Dockerfile`. A prudent practice then, for maintaining containers of value, is based upon use of Singularity definition files. Styled and implemented after a `Dockerfile` retrieved at some point in time, use of `diff` on subsequent versions of this same file, can be employed to inform maintenance of the corresponding Singularity definition file. Understanding build specifications at this level of detail places container creators in a much more sensible position prior to signing with an encrypted key. Thus the best practice is:

> "Maintain detailed build specifications for containers, rather than opaque runtimes"

3. Working with environment variables

In a `Dockerfile`, environment variables are declared as key-value pairs through use of the `ENV` instruction. Declaration in the build specification for a container is advised, rather than relying upon user (e.g., `.bashrc`, `.profile`) or system-wide configuration files for interactive shells. Should a `Dockerfile` be converted into a definition file for Singularity, as suggested in the container-maintenance best practice above, environment variables can be explicitly represented as `ENV` instructions that have been converted into entries in the `%environment` section, respectively. This best practice can be stated as follows:

"Define environment variables in container specifications, not interactive shells"

4. Installation to `/root`

Docker and OCI container's are typically run as the `root` user; therefore, `/root` (this user's `$HOME` directory) will be the installation target when `$HOME` is specified. Installation to `/root` may prove workable in some circumstances - e.g., while the container is executing, or if read-only access is required to this directory after installation. In general, however, because this is the `root` directory conventional wisdom suggests this practice be avoided. Thus the best practice is:

> "Avoid installations that make use of `/root`."

5. Read-only / filesystem

Singularity mounts a container's / filesystem in read-only mode. To ensure a Docker container meets Singularity's requirements, it may prove useful to execute `docker run --read-only --tmpfs /run --tmpfs /tmp godlovedc/lolcow`. The best practioce here is:

> "Ensure Docker containers meet Singularity's read-only / filesystem requirement"

6. Installation to `$HOME` or `$TMP`

In making use of Singularity, it is common practice for `$USER` to be automatically mounted on `$HOME`, and for `$TMP` also to be mounted. To avoid the side effects (e.g., 'missing' or conflicting files) that might arise as a consequence of executing `mount` commands then, the best practice is:

> "Avoid placing container 'valuables' in `$HOME` or `$TMP`."

A detailed review of the container's build specification (e.g., its `Dockerfile`) may be required to ensure this best practice is adhered to.

7. Current library caches

Irrespective of containers, a common runtime error stems from failing to locate shared libraries required for execution. Suppose now there exists a requirement for symbolically linked libraries *within* a Singularity container. If the builld process that creates the container fails to update the cache, then it is quite likely that (read-only) execution of this container will result in the common error of missing libraries. Upon investigation, it is likely revealed that the library exists, just not the required symbolic links. Thus the best practice is:

> "Ensure calls to `ldconfig` are executed towards the *end* of `build` specifications (e.g., `Dockerfile`), so that the library cache is updated when the container is created."

8. Use of plain-text passwords for authentication

For obvious reasons, it is desireable to completely *avoid* use of plain-text passwords. Therefore, for interactive sessions requiring authentication, use of the `--docker-login` option for Singularity's `pull` and `build` commands is *recommended*. At the present time, the *only* option available for non-interactive use is to *embed plain-text passwords into environment variables*. Because the Sylabs Cloud Singularity Library employs time-limited API tokens for authentication, use of SIF containers hosted through this service provides a more secure means for both interactive *and* non-interactive use. This best practice is:

> "Avoid use of plain-text passwords"

9. Execution ambiguity

Short of converting an *entire* `Dockerfile` into a Singularity definition file, informed specification of the `%runscript` entry in the def file *removes* any ambiguity associated with `ENTRYPOINT` *versus* `CMD` and ultimately *execution precedence*. Thus the best practice is:

> "Employ Singularity's `%runscript` by default to avoid execution ambiguity"

Note that the `ENTRYPOINT` can be bypassed completely, e.g., `docker run -i -t --entrypoint /bin/bash godlovedc/lolcow`. This allows for an interactive session within the container, that may prove useful in validating the built runtime.

Best practices emerge from experience. Contributions that allow additional experiences to be shared as best practices are always encouraged. Please refer to *Contributing* for additional details.

2.4.10 Troubleshooting

In making use of Docker and OCI images through Singularity the need to troubleshoot may arise. A brief compilation of issues and their resolution is provided here.

1. Authentication issues

Authentication is required to make use of Docker-style private images and/or private registries. Examples involving private images hosted by the public Docker Hub were *provided above*, whereas the NVIDIA GPU Cloud was used to *illustrate access to a private registry*. Even if the intended use of containers is non-interactive, issues in authenticating with these image-hosting services are most easily addressed through use of the `--docker-login` option that can be appended to a Singularity `pull` request. As soon as image signatures and blobs start being received, authentication credentials have been validated, and the image `pull` can be cancelled.

2. Execution mismatches

Execution intentions are detailed through specification files - i.e., the `Dockerfile` in the case of Docker images. However, intentions and precedence aside, the reality of executing a container may not align with expectations. To alleviate this mismatch, use of `singularity inspect --runscript <somecontainer>.sif` details the *effective* runscript - i.e., the one that is actually being executed. Of course, the ultimate solution to this issue is to develop and maintain Singularity definition files for containers of interest.

3. More than one image in the OCI bundle directory

As illustrated above, and with respect to the bootstrap agent `oci://$OCI_BUNDLE_DIR`, a fatal error is generated when *more* than one image is referenced in the `$OCI_BUNDLE_DIR/index.json` file. The workaround shared previously was to append the bootstrap directive with the unique reference name for the image of interest - i.e., `oci://$OCI_BUNDLE_DIR:org.opencontainers.image.ref.name`. Because it may take some effort to locate the reference name for an image of interest, an even simpler solution is to ensure that each `$OCI_BUNDLE_DIR` contains at most a single image.

4. Cache maintenance

Maintenance of the Singularity cache (i.e., `$HOME/.singularity/cache`) requires manual intervention at this time. By **carefully** issuing the command `rm -rf $HOME/.singularity/cache`, its local cache will be cleared of all downloaded images.

5. The `http` and `https` are `pull` only boostrap agents

`http` and `https` are the only examples of `pull` only boostrap agents. In other words, when used with Singularity's `pull` command, the result is a local copy of, for example, an OCI archive image. This means that a subsequent step is necessary to actually create a SIF container for use by Singularity - a step involving the `oci-archive` bootstrap agent in the case of an OCI image archive.

Like *best practices*, troubleshooting scenarios and solutions emerge from experience. Contributions that allow additional experiences to be shared are always encouraged. Please refer to *Contributing* for additional details.

2.4.11 Singularity Definition file vs. Dockerfile

On the following table, you can see which are the similarities/differences between a Dockerfile and a Singularity definition file:

Singularity Definition file		Dockerfile	
Section	Description	Section	Description
`Bootstrap`	Defines from which library to build your container from. You are free to choose between `library` (Our cloud library) , `docker` , `shub` and `oras`.	-	Can only bootstrap from Docker Hub.
`From:`	To specify the provider from which to build the container.	`FROM`	Creates a layer from the described docker image. For example, if you got a Dockerfile with the `FROM` section set like: `FROM:ubuntu:18.04`, this means that a layer will be created from the `ubuntu:18.04` **Docker** image. (You cannot choose any other bootstrap provider)
`%setup`	Commands that run outside the container (in the host system) after the base OS has been installed.	-	Not supported.
`%files`	To copy files from your local to the host.	`COPY`	To copy files from your Docker's client current directory.
`%environment`	To declare and set your environment variables.	`ENV`	`ENV` will take the name of the variable and the value and set it.
`%help`	To provide a help section to your container image.	-	Not supported on the Dockerfile.
`%post`	Commands that will	`RUN`	Commands to build your

2.5 Fakeroot feature

2.5.1 Overview

The fakeroot feature (commonly referred as rootless mode) allows an unprivileged user to run a container as a **"fake root"** user by leveraging user namespace UID/GID mapping.

Note: This feature requires a Linux kernel >= 3.8, but the recommended version is >= 3.18

A **"fake root"** user has almost the same administrative rights as root but only **inside the container** and the **requested namespaces**, which means that this user:

- can set different user/group ownership for files or directories they own
- can change user/group identity with su/sudo commands
- has full privileges inside the requested namespaces (network, ipc, uts)

2.5.2 Restrictions/security

Filesystem

A **"fake root"** user can't access or modify files and directories for which they don't already have access or rights on the host filesystem, so a **"fake root"** user won't be able to access root-only host files like `/etc/shadow` or the host `/root` directory.

Additionally, all files or directories created by the **"fake root"** user are owned by `root:root` inside container but as `user:group` outside of the container. Let's consider the following example, in this case "user" is authorized to use the fakeroot feature and can use 65536 UIDs starting at 131072 (same thing for GIDs).

UID inside container	UID outside container
0 (root)	1000 (user)
1 (daemon)	131072 (non-existent)
2 (bin)	131073 (non-existent)
…	…
65536	196607

Which means if the **"fake root"** user creates a file under a `bin` user in the container, this file will be owned by `131073:131073` outside of container. The responsibility relies on the administrator to ensure that there is no overlap with the current user's UID/GID on the system.

Network

Restrictions are also applied to networking, if `singularity` is executed without the `--net` flag, the **"fake root"** user won't be able to use `ping` or bind a container service to a port below 1024.

With `--net` the **"fake root"** user has full privileges in a dedicated container network. Inside the container network they can bind on privileged ports below 1024, use ping, manage firewall rules, listen to traffic, etc. Anything done in this dedicated network won't affect the host network.

Note: Of course an unprivileged user could not map host ports below than 1024 by using: `--network-args="portmap=80:80/tcp"`

Warning: For unprivileged installation of Singularity or if `allow setuid = no` is set in `singularity.conf` users won't be able to use a `fakeroot` network.

2.5.3 Requirements / Configuration

Fakeroot depends on user mappings set in `/etc/subuid` and group mappings in `/etc/subgid`, so your username needs to be listed in those files with a valid mapping (see the admin-guide for details), if you can't edit the files ask an administrator.

In Singularity `3.5` a `singularity config fakeroot` command has been added to allow configuration of the `/etc/subuid` and `/etc/subgid` mappings from the Singularity command line. You must be a root user or run with `sudo` to use `config fakeroot`, as the mapping files are security sensitive. See the admin-guide for more details.

2.5.4 Usage

If your user account is configured with valid `subuid` and `subgid` mappings you work as a fake root user inside a container by using the `--fakeroot` or `-f` option.

The `--fakeroot` option is available with the following singularity commands:

- `shell`
- `exec`
- `run`
- `instance start`
- `build`

Build

With fakeroot an unprivileged user can now build an image from a definition file with few restrictions. Some bootstrap methods that require creation of block devices (like `/dev/null`) may not always work correctly with **"fake root"**, Singularity uses seccomp filters to give programs the illusion that block device creation succeeded. This appears to work with `yum` bootstraps and *may* work with other bootstrap methods, although `debootstrap` is known to not work.

Examples

Build from a definition file:

```
singularity build --fakeroot /tmp/test.sif /tmp/test.def
```

Ping from container:

```
singularity exec --fakeroot --net docker://alpine ping -c1 8.8.8.8
```

HTTP server:

```
singularity run --fakeroot --net --network-args="portmap=8080:80/tcp" -w docker://
↪nginx
```

CHAPTER

THREE

SIGNING & ENCRYPTION

3.1 Signing and Verifying Containers

Singularity 3.0 introduced the ability to create and manage PGP keys and use them to sign and verify containers. This provides a trusted method for Singularity users to share containers. It ensures a bit-for-bit reproduction of the original container as the author intended it.

Note: Singularity 3.6.0 uses a new signature format. Containers signed by 3.6.0 cannot be verifed by older versions of Singularity.

To verify containers signed with older versions of Singularity using 3.6.0 the `--legacy-insecure` flag must be provided to the `singularity verify` command.

3.1.1 Verifying containers from the Container Library

The `verify` command will allow you to verify that a container has been signed using a PGP key. To use this feature with images that you pull from the container library, you must first generate an access token to the Sylabs Cloud. If you don't already have a valid access token, follow these steps:

1) Go to: https://cloud.sylabs.io/
2) Click "Sign in to Sylabs" and follow the sign in steps.
3) Click on your login id (same and updated button as the Sign in one).
4) Select "Access Tokens" from the drop down menu.
5) Enter a name for your new access token, such as "test token"
6) Click the "Create a New Access Token" button.
7) Click "Copy token to Clipboard" from the "New API Token" page.
8) Run `singularity remote login` and paste the access token at the prompt.

Now you can verify containers that you pull from the library, ensuring they are bit-for-bit reproductions of the original image.

```
$ singularity verify alpine_latest.sif

Container is signed by 1 key(s):

Verifying partition: FS:
8883491F4268F173C6E5DC49EDECE4F3F38D871E
```

(continues on next page)

(continued from previous page)

```
[REMOTE]  Sylabs Admin <support@sylabs.io>
[OK]      Data integrity verified

INFO:    Container verified: alpine_latest.sif
```

In this example you can see that **Sylabs Admin** has signed the container.

3.1.2 Signing your own containers

Generating and managing PGP keys

To sign your own containers you first need to generate one or more keys.

If you attempt to sign a container before you have generated any keys, Singularity will guide you through the interactive process of creating a new key. Or you can use the `newpair` subcommand in the `key` command group like so:.

```
$ singularity key newpair

Enter your name (e.g., John Doe) : David Trudgian
Enter your email address (e.g., john.doe@example.com) : david.trudgian@sylabs.io
Enter optional comment (e.g., development keys) : demo
Enter a passphrase :
Retype your passphrase :
Would you like to push it to the keystore? [Y,n] Y
Generating Entity and OpenPGP Key Pair... done
Key successfully pushed to: https://keys.sylabs.io
```

Note that I chose `Y` when asked if I wanted to push my key to the keystore. This will push my public key to whichever keystore has been configured by the `singularity remote` command, so that it can be retrieved by other users running `singularity verify`. If you do not wish to push your public key, say `n` during the `newpair` process.

The `list` subcommand will show you all of the keys you have created or saved locally.`

```
$ singularity key list

Public key listing (/home/dave/.singularity/sypgp/pgp-public):

0) U: David Trudgian (demo) <david.trudgian@sylabs.io>
   C: 2019-11-15 09:54:54 -0600 CST
   F: E5F780B2C22F59DF748524B435C3844412EE233B
   L: 4096
   --------
```

In the output above the index of my key is `0` and the letters stand for the following:

- U: User
- C: Creation date and time
- F: Fingerprint
- L: Key length

If you chose not to push your key to the keystore during the `newpair` process, but later wish to, you can push it to a keystore configured using `singularity remote` like so:

```
$ singularity key push E5F780B2C22F59DF748524B435C3844412EE233B

public key `E5F780B2C22F59DF748524B435C3844412EE233B` pushed to server successfully
```

If you delete your local public PGP key, you can always locate and download it again like so.

```
$ singularity key search Trudgian

Showing 1 results

KEY ID    BITS  NAME/EMAIL
12EE233B  4096  David Trudgian (demo) <david.trudgian@sylabs.io>

$ singularity key pull 12EE233B

1 key(s) added to keyring of trust /home/dave/.singularity/sypgp/pgp-public
```

But note that this only restores the *public* key (used for verifying) to your local machine and does not restore the *private* key (used for signing).

Searching for keys

Singularity allows you to search the keystore for public keys. You can search for names, emails, and fingerprints (key IDs). When searching for a fingerprint, you need to use `0x` before the fingerprint, check the example:

```
# search for key ID:
$ singularity key search 0x8883491F4268F173C6E5DC49EDECE4F3F38D871E

# search for the sort ID:
$ singularity key search 0xF38D871E

# search for user:
$ singularity key search Godlove

# search for email:
$ singularity key search @gmail.com
```

Signing and validating your own containers

Now that you have a key generated, you can use it to sign images like so:

```
$ singularity sign my_container.sif

Signing image: my_container.sif
Enter key passphrase :
Signature created and applied to my_container.sif
```

Because your public PGP key is saved locally you can verify the image without needing to contact the Keystore.

```
$ singularity verify my_container.sif
Verifying image: my_container.sif
[LOCAL]   Signing entity: David Trudgian (Demo keys) <david.trudgian@sylabs.io>
[LOCAL]   Fingerprint: 65833F473098C6215E750B3BDFD69E5CEE85D448
Objects verified:
```

(continues on next page)

(continued from previous page)

```
ID   |GROUP    |LINK     |TYPE
------------------------------------------------
1    |1        |NONE     |Def.FILE
2    |1        |NONE     |JSON.Generic
3    |1        |NONE     |FS
Container verified: my_container.sif
```

If you've pushed your key to the Keystore you can also verify this image in the absence of a local public key. To demonstrate this, first `remove` your local public key, and then try to use the `verify` command again.

```
$ singularity key remove E5F780B2C22F59DF748524B435C3844412EE233B

$ singularity verify my_container.sif
Verifying image: my_container.sif
[REMOTE]   Signing entity: David Trudgian (Demo keys) <david.trudgian@sylabs.io>
[REMOTE]   Fingerprint: 65833F473098C6215E750B3BDFD69E5CEE85D448
Objects verified:
ID   |GROUP    |LINK     |TYPE
------------------------------------------------
1    |1        |NONE     |Def.FILE
2    |1        |NONE     |JSON.Generic
3    |1        |NONE     |FS
Container verified: my_container.sif
```

Note that the `[REMOTE]` message shows the key used for verification was obtained from the keystore, and is not present on your local computer. You can retrieve it, so that you can verify even if you are offline with `singularity key pull`

```
$ singularity key pull E5F780B2C22F59DF748524B435C3844412EE233B

1 key(s) added to keyring of trust /home/dave/.singularity/sypgp/pgp-public
```

Advanced Signing - SIF IDs and Groups

As well as the default behaviour, which signs all objects, fine-grained control of signing is possible.

If you `sif list` a SIF file you will see it is comprised of a number of objects. Each object has an `ID`, and belongs to a `GROUP`.

```
$ singularity sif list my_container.sif

Container id: e455d2ae-7f0b-4c79-b3ef-315a4913d76a
Created on:   2019-11-15 10:11:58 -0600 CST
Modified on:  2019-11-15 10:11:58 -0600 CST
----------------------------------------------------
Descriptor list:
ID    |GROUP    |LINK     |SIF POSITION (start-end)  |TYPE
------------------------------------------------------------------------------
1     |1        |NONE     |32768-32800               |Def.FILE
2     |1        |NONE     |36864-36961               |JSON.Generic
3     |1        |NONE     |40960-25890816            |FS (Squashfs/*System/amd64)
```

I can choose to sign and verify a specific object with the `--sif-id` option to `sign` and `verify`.

```
$ singularity sign --sif-id 1 my_container.sif
Signing image: my_container.sif
Enter key passphrase :
Signature created and applied to my_container.sif

$ singularity verify --sif-id 1 my_container.sif
Verifying image: my_container.sif
[LOCAL]   Signing entity: David Trudgian (Demo keys) <david.trudgian@sylabs.io>
[LOCAL]   Fingerprint: 65833F473098C6215E750B3BDFD69E5CEE85D448
Objects verified:
ID  |GROUP   |LINK    |TYPE
------------------------------------------------
1   |1       |NONE    |Def.FILE
Container verified: my_container.sif
```

Note that running the `verify` command without specifying the specific sif-id gives a fatal error. The container is not considered verified as whole because other objects could have been changed without my knowledge.

```
$ singularity verify my_container.sif
Verifying image: my_container.sif
[LOCAL]   Signing entity: David Trudgian (Demo keys) <david.trudgian@sylabs.io>
[LOCAL]   Fingerprint: 65833F473098C6215E750B3BDFD69E5CEE85D448

Error encountered during signature verification: object 2: object not signed
FATAL:   Failed to verify container: integrity: object 2: object not signed
```

I can sign a group of objects with the `--group-id` option to `sign`.

```
$ singularity sign --groupid 1 my_container.sif
Signing image: my_container.sif
Enter key passphrase :
Signature created and applied to my_container.sif
```

This creates one signature over all objects in the group. I can verify that nothing in the group has been modified by running `verify` with the same `--group-id` option.

```
$ singularity verify --group-id 1 my_container.sif
Verifying image: my_container.sif
[LOCAL]   Signing entity: David Trudgian (Demo keys) <david.trudgian@sylabs.io>
[LOCAL]   Fingerprint: 65833F473098C6215E750B3BDFD69E5CEE85D448
Objects verified:
ID  |GROUP   |LINK    |TYPE
------------------------------------------------
1   |1       |NONE    |Def.FILE
2   |1       |NONE    |JSON.Generic
3   |1       |NONE    |FS
Container verified: my_container.sif
```

Because every object in the SIF file is within the signed group 1 the entire container is signed, and the default `verify` behavior without specifying `--group-id` can also verify the container:

```
$ singularity verify my_container.sif
Verifying image: my_container.sif
[LOCAL]   Signing entity: David Trudgian (Demo keys) <david.trudgian@sylabs.io>
[LOCAL]   Fingerprint: 65833F473098C6215E750B3BDFD69E5CEE85D448
Objects verified:
ID  |GROUP   |LINK    |TYPE
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------
1   |1        |NONE     |Def.FILE
2   |1        |NONE     |JSON.Generic
3   |1        |NONE     |FS
Container verified: my_container.sif
```

3.2 Key commands

Singularity 3.2 introduces the abilities to import, export and remove PGP keys following the OpenPGP standard via GnuPGP (GPG). These commands only modify the local keyring and are not related to the cloud keystore.

3.2.1 Key import command

Singularity 3.2 allows you import keys reading either from binary or armored key format and automatically detect if it is a private or public key and add it to the correspondent local keystore.

To give a quick view on how it works, we will first consider the case in which a user wants to import a secret (private) key to the local keystore.

First we will check what's the status of the local keystore (which keys are stored by the moment before importing a new key).

```
$ singularity key list --secret
```

Note: Remember that using `--secret` flag or `-s` flag will return the secret or private local keyring as output.

The output will look as it follows:

```
Private key listing (/home/joana/.singularity/sypgp/pgp-secret):

0) U: Johnny Cash (none) <cash@sylabs.io>
C: 2019-04-11 22:22:28 +0200 CEST
F: 47282BDC661F58FA4BEBEF47CA576CBD8EF1A2B4
L: 3072
--------
1) U: John Green (none) <john@sylabs.io>
C: 2019-04-11 13:08:45 +0200 CEST
F: 5720799FE7B048CF36FAB8445EE1E2BD7B6342C5
L: 1024
--------
```

Note: Remember that running that same command but with sudo privilege, will give you a totally different list since it will be the correspondent keystore from user `root`

After this, you can simply import the key you need by adding the exact location to the file, let's say you own a gpg key file named `pinkie-pie.asc` which is a secret GPG key you want to import. Then you will just need to run the following command to import your key:

```
$ singularity key import $HOME/pinkie-pie.asc
```

Note: This location is considering your key was located on the `$HOME` directory. You can specify any location to the file.

Since you're importing a private (secret) key, you will need to specify the passphrase related to it and then a new passphrase to be added on your local keystore.

```
Enter your old password :
Enter a new password for this key :
Retype your passphrase :
Key with fingerprint 8C10B902F438E4D504C3ACF689FCFFAED5F34A77 successfully added to␣
↪the keyring
```

After this you can see if that key was correctly added to your local keystore by running `singularity key list -s` command:

```
Private key listing (/home/joana/.singularity/sypgp/pgp-secret):

  0) U: Johnny Cash (none) <cash@sylabs.io>
  C: 2019-04-11 22:22:28 +0200 CEST
  F: 47282BDC661F58FA4BEBEF47CA576CBD8EF1A2B4
  L: 3072
  --------
  1) U: John Green (none) <john@sylabs.io>
  C: 2019-04-11 13:08:45 +0200 CEST
  F: 5720799FE7B048CF36FAB8445EE1E2BD7B6342C5
  L: 1024
  --------
  3) U: Pinkie Pie (Eternal chaos comes with chocolate rain!) <balloons@sylabs.io>
  C: 2019-04-26 12:07:07 +0200 CEST
  F: 8C10B902F438E4D504C3ACF689FCFFAED5F34A77
  L: 1024
  --------
```

You will see the imported key at the bottom of the list. Remember you can also import an `ascii` armored key and this will be automatically detected by the `key import` command (no need to specify the format).

Note: In case you would like to import a public key the process remains the same, as the import command will automatically detect whether this key to be imported is either public or private.

3.2.2 Key export command

The key export command allows you to export a key that is on your local keystore. This key could be either private or public, and the key can be exported on `ASCII` armored format or on binary format. Of course to identify the keystore and the format the syntax varies from the `key import` command.

For example to export a public key in binary format you can run:

```
$ singularity key export 8C10B902F438E4D504C3ACF689FCFFAED5F34A77 $HOME/mykey.asc
```

This will export a public binary key named `mykey.asc` and will save it under the home folder. If you would like to export the same public key but in an `ASCII` armored format, you would need to run the following command:

```
$ singularity key export --armor 8C10B902F438E4D504C3ACF689FCFFAED5F34A77 $HOME/mykey.
↪asc
```

And in the case in which you may need to export a secret key on `ASCII` armored format, you would need to specify from where to find the key, since the fingerprint is the same.

```
$ singularity key export --armor --secret 8C10B902F438E4D504C3ACF689FCFFAED5F34A77
↪$HOME/mykey.asc
```

and on binary format instead:

```
$ singularity key export --secret 8C10B902F438E4D504C3ACF689FCFFAED5F34A77 $HOME/
↪mykey.asc
```

Note: Exporting keys will not change the status of your local keystore or keyring. This will just obtain the content of the keys and save it on a local file on your host.

3.2.3 Key remove command

In case you would want to remove a public key from your public local keystore, you can do so by running the following command:

```
$ singularity key remove 8C10B902F438E4D504C3ACF689FCFFAED5F34A77
```

Note: Remember that this will only delete the public key and not the private one with the same matching fingerprint.

3.3 Encrypted Containers

Users can build a secure, confidential container environment by encrypting the root file system.

3.3.1 Overview

In Singularity >= v3.4.0 a new feature to build and run encrypted containers has been added to allow users to encrypt the file system image within a SIF. This encryption can be performed using either a passphrase or asymmetrically via an RSA key pair in Privacy Enhanced Mail (PEM/PKCS1) format. The container is encrypted in transit, at rest, and even while running. In other words, there is no intermediate, decrypted version of the container on disk. Container decryption occurs at runtime completely within kernel space.

Note: This feature utilizes the Linux `dm-crypt` library and `cryptsetup` utility and requires cryptsetup version of >= 2.0.0. This version should be standard with recent Linux versions such as Ubuntu 18.04, Debian 10 and CentOS/RHEL 7, but users of older Linux versions may have to update.

3.3.2 Encrypting a container

A container can be encrypted either by supplying a plaintext passphrase or a PEM file containing an asymmetric RSA public key. Of these two methods the PEM file is more secure and is therefore recommended for production use.

Note: In Singularity 3.4, the definition file stored with the container will not be encrypted. If it contains sensitive information you should remove it before encryption via `singularity sif del 1 myimage.sif`. Metadata encryption will be addressed in a future release.

An `-e|--encrypt` flag to `singularity build` is used to indicate that the container needs to be encrypted.

A passphrase or a key-file used to perform the encryption is supplied at build time via an environment variable or a command line option.

Encryption Method	Environment Variable	Commandline Option
Passphrase	`SINGULARITY_ENCRYPTION_PASSPHRASE`	`--passphrase`
Asymmetric Key (PEM)	`SINGULARITY_ENCRYPTION_PEM_PATH`	`--pem-path`

The `-e|--encrypt` flag is implicitly set when the `--passphrase` or `--pem-path` flags are passed with the build command. If multiple encryption related flags and/or environment variables are set, the following precedence is respected.

1. `--pem-path`
2. `--passphrase`
3. `SINGULARITY_ENCRYPTION_PEM_PATH`
4. `SINGULARITY_ENCRYPTION_PASSPHRASE`

Passphrase Encryption

Note: Passphrase encryption is less secure than encrypting containers using an RSA key pair (detailed below). Passphrase encryption is provided as a convenience, and as a way for users to familiarize themselves with the encrypted container workflow, but users running encrypted containers in production are encouraged to use asymmetric keys.

In case of plaintext passphrase encryption, a passphrase is supplied by one of the following methods.

Encrypting with a passphrase interactively

```
$ sudo singularity build --passphrase encrypted.sif encrypted.def
Enter encryption passphrase: <secret>
INFO:    Starting build...
```

Using an environment variable

```
$ sudo SINGULARITY_ENCRYPTION_PASSPHRASE=<secret> singularity build --encrypt␣
↪encrypted.sif encrypted.def
Starting build...
```

In this case it is necessary to use the `--encrypt` flag since the presence of an environment variable alone will not trigger the encrypted build workflow.

While this example shows how an environment variable can be used to set a passphrase, you should set the environment variable in a way that will not record your passphrase on the command line. For instance, you could save a plain text passphrase in a file (e.g. `secret.txt`) and use it like so.

```
$ export SINGULARITY_ENCRYPTION_PASSPHRASE=$(cat secret.txt)

$ sudo -E singularity build --encrypt encrypted.sif encrypted.def
Starting build...
```

PEM File Encryption

Singularity currently supports RSA encryption using a public/private key-pair. Keys are supplied in PEM format. The public key is used to encrypt containers that can be decrypted on a host that has access to the secret private key.

You can create a pair of RSA keys suitable for encrypting your container with the `ssh-keygen` command, and then create a PEM file with a few specific flags like so:

```
# Generate a keypair
$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/vagrant/.ssh/id_rsa): rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
[snip...]

# Convert the public key to PEM PKCS1 format
$ ssh-keygen -f ./rsa.pub -e -m pem >rsa_pub.pem

# Rename the private key (already PEM PKCS1) to a nice name
$ mv rsa rsa_pri.pem
```

You would use the `rsa_pub.pem` file to encrypt your container and the `rsa_pri.pem` file to run it.

Encrypting with a command line option

```
$ sudo singularity build --pem-path=rsa_pub.pem encrypted.sif encrypted.def
Starting build...
```

Encrypting with an environment variable

```
$ sudo SINGULARITY_ENCRYPTION_PEM_PATH=rsa_pub.pem singularity build --encrypt␣
↪encrypted.sif encrypted.def
Starting build...
```

In this case it is necessary to use the `--encrypt` flag since the presence of an environment variable alone will not trigger the encrypted build workflow.

3.3.3 Running an encrypted container

To `run`, `shell`, or `exec` an encrypted image, credentials to decrypt the image need to be supplied at runtime either in a key-file or a plaintext passphrase.

Running a container encrypted with a passphrase

A passphrase can be supplied at runtime by either of the ways listed in the sections above.

Running with a passphrase interactively

```
$ singularity run --passphrase encrypted.sif
Enter passphrase for encrypted container: <secret>
```

Running with a passphrase in an environment variable

```
$ SINGULARITY_ENCRYPTION_PASSPHRASE="secret" singularity run encrypted.sif
```

While this example shows how an environment variable can be used to set a passphrase, you should set the environment variable in a way that will not record your passphrase on the command line. For instance, you could save a plain text passphrase in a file (e.g. `secret.txt`) and use it like so.

```
$ export SINGULARITY_ENCRYPTION_PASSPHRASE=$(cat secret.txt)

$ singularity run encrypted.sif
```

Running a container encrypted with a PEM file

A private key is supplied using either of the methods listed in the Encryption section above.

Running using a command line option

```
$ singularity run --pem-path=rsa_pri.pem encrypted.sif
```

Running using an environment variable

```
$ SINGULARITY_ENCRYPTION_PEM_PATH=rsa_pri.pem singularity run encrypted.sif
```

CHAPTER

FOUR

SHARING & ONLINE SERVICES

4.1 Remote Endpoints

4.1.1 Overview

Sylabs introduced the online Sylabs Cloud to enable users to Create, Secure, and Share their container images with others.

The `remote` command group in Singularity allows you to login to an account on the public Sylabs Cloud, or configure Singularity to point to a local installation of Singularity Enterprise, which provides an on-premise private Container Library, Remote Builder and Key Store.

Users can setup and switch between multiple remote endpoints, which are stored in their `~/.singularity/remote.yaml` file. Alternatively, remote endpoints can be set system-wide by an administrator.

Note: The `remote` command group configures Singularity to use and authenticate to the public Sylabs Cloud, a private installation of Singularity Enterprise, or community-developed services that are API compatible.

The `remote` command group *cannot be used* to e.g. configure singularity to store credentials for access to a docker registry. See the *Support for Docker and OCI* guide for information about authenticating to various docker registries.

4.1.2 Public Sylabs Cloud

A fresh, default installation of Singularity is configured to connect to the public cloud.sylabs.io services. If you only want to use the public services you just need to obtain an authentication token, and then `singularity remote login`:

1) Go to: https://cloud.sylabs.io/
2) Click "Sign in to Sylabs" and follow the sign in steps.
3) Click on your login id (same and updated button as the Sign in one).
4) Select "Access Tokens" from the drop down menu.
5) Enter a name for your new access token, such as "test token"
6) Click the "Create a New Access Token" button.
7) Click "Copy token to Clipboard" from the "New API Token" page.
8) Run `singularity remote login` and paste the access token at the prompt.

Once your token is stored, you can check that you are able to connect to the services with the `status` subcommand:

```
$ singularity remote status
INFO:    Checking status of default remote.
SERVICE            STATUS  VERSION
Builder Service    OK      v1.1.4-0-g3ef2555
Consent Service    OK      v1.0.2-0-g2a24b4a
Keystore Service   OK      v1.9.0-0-g112eb0e-dirty
Library Service    OK      v1.0.4-0-g24d3b74
Token Service      OK      v1.0.2-0-g2a24b4a
```

If you see any errors you may need to check if your system requires proxy environment variables to be set, or if a firewall is blocking access to `*.sylabs.io`. Talk to your system adminitrator.

You can interact with the public Sylabs Cloud using various Singularity commands:

pull, push, build –remote, key, search, verify, exec, shell, run, instance

Note: Using `docker://`, `oras://` and `shub://` URIs with these commands does not interact with the Sylabs Cloud.

4.1.3 Managing Remote Endpoints

Generally, users and administrators should manage remote endpoints using the `singularity remote` command, and avoid editing `remote.yaml` configuration files directly.

List and Login to Remotes

To `list` existing remote endpoints, run this:

```
$ singularity remote list

NAME           URI               GLOBAL
[SylabsCloud]  cloud.sylabs.io   YES
```

The `[...]` brackets around the name `SylabsCloud` show that this is the current default remote endpoint.

To `login` to a remote, for the first time or if your token expires or was revoked:

```
# Login to the default remote endpoint
$ singularity remote login

# Login to another remote endpoint
$ singularity remote login <remote_name>

# example...
$ singularity remote login SylabsCloud
singularity remote login SylabsCloud
INFO:    Authenticating with remote: SylabsCloud
Generate an API Key at https://cloud.sylabs.io/auth/tokens, and paste here:
API Key:
INFO:    API Key Verified!
```

Add & Remove Remotes

To `add` a remote endpoint (for the current user only):

```
$ singularity remote add <remote_name> <remote_uri>
```

For example, if you have an installation of Singularity enterprise hosted at enterprise.example.com:

```
$ singularity remote add myremote https://enterprise.example.com

INFO:    Remote "myremote" added.
INFO:    Authenticating with remote: myremote
Generate an API Key at https://enterprise.example.com/auth/tokens, and paste here:
API Key:
```

You will be prompted to setup an API key as the remote is added. The web address needed to do this will always be given.

To `add` a global remote endpoint (available to all users on the system) an administrative user should run:

```
$ sudo singularity remote add --global <remote_name> <remote_uri>

# example..

$ sudo singularity remote add --global company-remote https://enterprise.example.com
[sudo] password for dave:
INFO:    Remote "company-remote" added.
INFO:    Global option detected. Will not automatically log into remote.
```

Note: Global remote configurations can only be modified by the root user and are stored in the `etc/singularity/remote.yaml` file, at the Singularity installation location.

Conversely, to `remove` an endpoint:

```
$ singularity remote remove <remote_name>
```

Use the `--global` option as the root user to remove a global endpoint:

```
$ sudo singularity remote remove --global <remote_name>
```

Set the Default Remote

A remote endpoint can be set as the default to use with commands such as `push`, `pull` etc. via `remote use`:

```
$ singularity remote use <remote_name>
```

The default remote shows up in `[...]` square brackets in the output of `remote list`:

```
$ singularity remote list
NAME            URI                     GLOBAL
[SylabsCloud]   cloud.sylabs.io         YES
company-remote  enterprise.example.com  YES
myremote        enterprise.example.com  NO

```

(continues on next page)

(continued from previous page)

```
$ singularity remote use myremote
INFO:    Remote "myremote" now in use.

$ singularity remote list
NAME            URI                     GLOBAL
SylabsCloud     cloud.sylabs.io         YES
company-remote  enterprise.example.com  YES
[myremote]      enterprise.example.com  NO
```

If you do not want to switch remote with `remote use` you can:

- Make `push` and `pull` use an alternative library server with the `--library` option.
- Make `build --remote` use an alternative remote builder with the `--builder` option.
- Make `keys` use an alternative keyserver with the `-url` option.

4.2 Cloud Library

4.2.1 Overview

The Sylabs Cloud Library is the place to *push* your containers to the cloud so other users can *pull*, *verify*, and use them.

The Sylabs Cloud also provides a *Remote Builder*, allowing you to build containers on a secure remote service. This is convenient so that you can build containers on systems where you do not have root privileges.

4.2.2 Make an Account

Making an account is easy, and straightforward:

1. Go to: https://cloud.sylabs.io/library.
2. Click "Sign in to Sylabs" (top right corner).
3. Select your method to sign in, with Google, GitHub, GitLab, or Microsoft.
4. Type your passwords, and that's it!

4.2.3 Creating a Access token

Access tokens for pushing a container, and remote builder.

To generate a access token, do the following steps:

1) Go to: https://cloud.sylabs.io/
2) Click "Sign in to Sylabs" and follow the sign in steps.
3) Click on your login id (same and updated button as the Sign in one).
4) Select "Access Tokens" from the drop down menu.
5) Enter a name for your new access token, such as "test token"
6) Click the "Create a New Access Token" button.
7) Click "Copy token to Clipboard" from the "New API Token" page.

8) Run `singularity remote login` and paste the access token at the prompt.

Now that you have your token, you are ready to push your container!

4.2.4 Pushing a Container

The `singularity push` command will push a container to the container library with the given URL. Here's an example of a typical push command:

```
$ singularity push my-container.sif library://your-name/project-dir/my-
↪container:latest
```

The `:latest` is the container tag. Tags are used to have different version of the same container.

Note: When pushing your container, theres no need to add a `.sif` (Singularity Image Format) to the end of the container name, (like on your local machine), because all containers on the library are SIF containers.

Let's assume you have your container (v1.0.1), and you want to push that container without deleting your `:latest` container, then you can add a version tag to that container, like so:

```
$ singularity push my-container.sif library://your-name/project-dir/my-container:1.0.1
```

You can download the container with that tag by replacing the `:latest`, with the tagged container you want to download.

4.2.5 Pulling a container

The `singularity pull` command will download a container from the Library (`library://`), Docker Hub (`docker://`), and also Shub (`shub://`).

Note: When pulling from Docker, the container will automatically be converted to a SIF (Singularity Image Format) container.

Here's a typical pull command:

```
$ singularity pull file-out.sif library://alpine:latest

# or pull from docker:

$ singularity pull file-out.sif docker://alpine:latest
```

Note: If there's no tag after the container name, Singularity automatically will pull the container with the `:latest` tag.

To pull a container with a specific tag, just add the tag to the library URL:

```
$ singularity pull file-out.sif library://alpine:3.8
```

Of course, you can pull your own containers. Here's what that will look like:

Pulling your own container

Pulling your own container is just like pulling from Github, Docker, etc…

```
$ singularity pull out-file.sif library://your-name/project-dir/my-container:latest

# or use a different tag:

$ singularity pull out-file.sif library://your-name/project-dir/my-container:1.0.1
```

Note: You *don't* have to specify a output file, one will be created automatically, but it's good practice to always specify your output file.

4.2.6 Verify/Sign your Container

Verify containers that you pull from the library, ensuring they are bit-for-bit reproductions of the original image.

Check out *this page* on how to: *verify a container*, making PGP key, and sign your own containers.

4.2.7 Searching the Library for Containers

When it comes to searching the library, you could always go to: https://cloud.sylabs.io/library and search from there through the web GUI. Or you can use `singularity search <container/user>`, this will search the library for the `<container/user>`.

Using the CLI Search

Here is an example for searching the library for `centos`:

```
$ singularity search centos
No users found for 'centos'

No collections found for 'centos'

Found 6 containers for 'centos'
    library://dtrudg/linux/centos
            Tags: 6 7 centos6 centos7 latest
    library://library/default/centos
            Tags: 6 7 latest
    library://gmk/demo/centos-vim
            Tags: latest
    library://mroche/baseline/centos
            Tags: 7 7.5 7.5.1804 7.6 7.6.1810 latest
    library://gmk/default/centos7-devel
            Tags: latest
    library://emmeff/default/centos7-python36
            Tags: 1.0
```

Notice there are different tags for the same container.

4.2.8 Remote Builder

The remote builder service can build your container in the cloud removing the requirement for root access.

Here's a typical remote build command:

```
$ singularity build --remote file-out.sif docker://ubuntu:18.04
```

Building from a definition file:

This is our definition file. Let's call it `ubuntu.def`:

```
bootstrap: library
from: ubuntu:18.04

%runscript
    echo "hello world from ubuntu container!"
```

Now, to build the container, use the `--remote` flag, and without `sudo`:

```
$ singularity build --remote ubuntu.sif ubuntu.def
```

Note: Make sure you have a *access token*, otherwise the build will fail.

After building, you can test your container like so:

```
$ ./ubuntu.sif
hello world from ubuntu container!
```

You can also use the web GUI to build containers remotely. First, go to https://cloud.sylabs.io/builder (make sure you are signed in). Then you can copy and paste, upload, or type your definition file. When you are finished, click build. Then you can download the container with the URL.

CHAPTER

FIVE

ADVANCED USAGE

5.1 Bind Paths and Mounts

If enabled by the system administrator, Singularity allows you to map directories on your host system to directories within your container using bind mounts. This allows you to read and write data on the host system with ease.

5.1.1 Overview

When Singularity 'swaps' the host operating system for the one inside your container, the host file systems becomes inaccessible. But you may want to read and write files on the host system from within the container. To enable this functionality, Singularity will bind directories back into the container via two primary methods: system-defined bind paths and user-defined bind paths.

5.1.2 System-defined bind paths

The system administrator has the ability to define what bind paths will be included automatically inside each container. Some bind paths are automatically derived (e.g. a user's home directory) and some are statically defined (e.g. bind paths in the Singularity configuration file). In the default configuration, the system default bind points are `$HOME` , `/sys:/sys` , `/proc:/proc`, `/tmp:/tmp`, `/var/tmp:/var/tmp`, `/etc/resolv.conf:/etc/resolv.conf`, `/etc/passwd:/etc/passwd`, and `$PWD`. Where the first path before `:` is the path from the host and the second path is the path in the container.

5.1.3 User-defined bind paths

If the system administrator has enabled user control of binds, you will be able to request your own bind paths within your container.

The Singularity action commands (`run`, `exec`, `shell`, and `instance start` will accept the `--bind/-B` command-line option to specify bind paths, and will also honor the `$SINGULARITY_BIND` (or `$SINGULARITY_BINDPATH`) environment variable. The argument for this option is a comma-delimited string of bind path specifications in the format `src[:dest[:opts]]`, where `src` and `dest` are paths outside and inside of the container respectively. If `dest` is not given, it is set equal to `src`. Mount options (`opts`) may be specified as `ro` (read-only) or `rw` (read/write, which is the default). The `--bind/-B` option can be specified multiple times, or a comma-delimited string of bind path specifications can be used.

Specifying bind paths

Here's an example of using the `--bind` option and binding `/data` on the host to `/mnt` in the container (`/mnt` does not need to already exist in the container):

```
$ ls /data
bar  foo

$ singularity exec --bind /data:/mnt my_container.sif ls /mnt
bar  foo
```

You can bind multiple directories in a single command with this syntax:

```
$ singularity shell --bind /opt,/data:/mnt my_container.sif
```

This will bind `/opt` on the host to `/opt` in the container and `/data` on the host to `/mnt` in the container.

Using the environment variable instead of the command line argument, this would be:

```
$ export SINGULARITY_BIND="/opt,/data:/mnt"

$ singularity shell my_container.sif
```

Using the environment variable `$SINGULARITY_BIND`, you can bind paths even when you are running your container as an executable file with a runscript. If you bind many directories into your Singularity containers and they don't change, you could even benefit by setting this variable in your `.bashrc` file.

A note on using `--bind` with the `--writable` flag

To mount a bind path inside the container, a *bind point* must be defined within the container. The bind point is a directory within the container that Singularity can use as a destination to bind a directory on the host system.

Starting in version 3.0, Singularity will do its best to bind mount requested paths into a container regardless of whether the appropriate bind point exists within the container. Singularity can often carry out this operation even in the absence of the "overlay fs" feature.

However, binding paths to non-existent points within the container can result in unexpected behavior when used in conjuction with the `--writable` flag, and is therefore disallowed. If you need to specify bind paths in combination with the `--writable` flag, please ensure that the appropriate bind points exist within the container. If they do not already exist, it will be necessary to modify the container and create them.

Using `--no-home` and `--containall` flags

`--no-home`

When shelling into your container image, Singularity allows you to mount your current working directory (`CWD`) without mounting your host `$HOME` directory with the `--no-home` flag.

```
$ singularity shell --no-home my_container.sif
```

Note: Beware that if it is the case that your `CWD` is your `$HOME` directory, it will still mount your `$HOME` directory.

--containall

Using the `--containall` (or `-C` for short) flag, `$HOME` is not mounted and a dummy bind mount is created at the `$HOME` point. You cannot use `-B`` (or `--bind`) to bind your `$HOME` directory because it creates an empty mount. So if you have files located in the image at `/home/user`, the `--containall` flag will hide them all.

```
$ singularity shell --containall my_container.sif
```

5.1.4 FUSE mounts

Filesystem in Userspace (FUSE) is an interface to allow filesystems to be mounted using code that runs in userspace, rather than in the Linux Kernel. Unprivileged (non-root) users can mount filesystems that have FUSE drivers. For example, the `fuse-sshfs` package allows you to mount a remote computer's filesystem to your local host, over ssh:

```
$ mount.fuse sshfs#ythel:/home/dave other_host/

# Now mounted to my local machine:
$ ythel:/home/dave on /home/dave/other_host type fuse.sshfs (rw,nosuid,nodev,relatime,
↪user_id=1000,group_id=1000)
```

Singularity 3.6 introduces the `--fusemount` option, which allows you directly expose FUSE filesystems inside a container. The FUSE command / driver that mounts a particular type of filesystem can be located on the host, or in the container.

The FUSE command *must* be based on libfuse3 to work correctly with Singularity `--fusemount`. If you are using an older distribution that provides FUSE commands such as `sshfs` based on FUSE 2 then you can install FUSE 3 versions of the commands you need inside your container.

Note: `--fusemount` functionality was present in a hidden preview state from Singularity 3.4. The behavior has changed for the final supported version introduced in Singularity 3.6.

FUSE mount definitions

A fusemount definition for Singularity consists of 3 parts:

```
--fusemount <type>:<fuse command> <container mountpoint>
```

- **type** specifies how and where the FUSE mount will be run. The options are:
  - `container` - use a FUSE command on the host, to mount a filesystem into the container, with the fuse process attached.
  - `host` - use a FUSE command inside the container, to mount a filesystem into the container, with the fuse process attached.
  - `container-daemon` - use a FUSE command on the host, to mount a filesystem into the container, with the fuse process detached.
  - `host-daemon` - use a FUSE command inside the container, to mount a filesystem into the container, with the fuse process detached.

- **fuse command** specifies the name of the executable that implements the FUSE mount, and any arguments. E.g. `sshfs server:over-there/` for mounting a remote filesystem over SSH, where the remote source is `over-there/` in my home directory on the machine called `server`.
- **container mountpoint** is an *absolute path* at which the FUSE filesystem will be mounted in the container.

FUSE mount with a host executable

To use a FUSE `sshfs` mount in a container, where the `fuse-sshfs` package has been installed on my host, I run with the `host` mount type:

```
$ singularity run --fusemount "host:sshfs server:/ /server" docker://ubuntu
Singularity> cat /etc/hostname
localhost.localdomain
Singularity> cat /server/etc/hostname
server
```

FUSE mount with a container executable

If the FUSE driver / command that you want to use for the mount has been added to your container, you can use the `container` mount type:

```
$ singularity run --fusemount "container:sshfs server:/ /server" sshfs.sif
Singularity> cat /etc/hostname
localhost.localdomain
Singularity> cat /server/etc/hostname
server
```

5.2 Persistent Overlays

Persistent overlay directories allow you to overlay a writable file system on an immutable read-only container for the illusion of read-write access. You can run a container and make changes, and these changes are kept separately from the base container image.

5.2.1 Overview

A persistent overlay is a directory or file system image that "sits on top" of your immutable SIF container. When you install new software or create and modify files the overlay will store the changes.

If you want to use a SIF container as though it were writable, you can create a directory, an ext3 file system image, or embed an ext3 file system image in SIF to use as a persistent overlay. Then you can specify that you want to use the directory or image as an overlay at runtime with the `--overlay` option, or `--writable` if you want to use the overlay embedded in SIF.

If you want to make changes to the image, but do not want them to persist, use the `--writable-tmpfs` option. This stores all changes in an in-memory temporary filesystem which is discarded as soon as the container finishes executing.

You can use persistent overlays with the following commands:

- `run`
- `exec`

- shell
- instance.start

5.2.2 Usage

To use a persistent overlay, you must first have a container.

```
$ sudo singularity build ubuntu.sif library://ubuntu
```

File system image overlay

You can use tools like `dd` and `mkfs.ext3` to create and format an empty ext3 file system image, which holds all changes made in your container within a single file. Using an overlay image file makes it easy to transport your modifications as a single additional file alongside the original SIF container image.

Workloads that write a very large number of small files into an overlay image, rather than a directory, are also faster on HPC parallel filesystems. Each write is a local operation within the single open image file, and does not cause additional metadata operations on the parallel filesystem.

To create an overlay image file with 500MBs of empty space:

```
$ dd if=/dev/zero of=overlay.img bs=1M count=500 && \
    mkfs.ext3 overlay.img
```

Now you can use this overlay with your container, though filesystem permissions still control where you can write, so `sudo` is needed to run the container as `root` if you need to write to `/` inside the container.

```
$ sudo singularity shell --overlay overlay.img ubuntu.sif
```

To manage permissions in the overlay, so the container is writable by unprivileged users you can create a directory structure on your host, set permissions on it as needed, and include it in the overlay with the `-d` option to `mkfs.ext3`:

```
$ mkdir -p overlay/upper overlay/work
$ dd if=/dev/zero of=overlay.img bs=1M count=500 && \
     mkfs.ext3 -d overlay overlay.img
```

Now the container will be writable as the unprivileged user who created the `overlay/upper` and `overlay/work` directories that were placed into `overlay.img`.

```
$ singularity shell --overlay overlay.img ubuntu.sif
Singularity> echo $USER
dtrudg
Singularity> echo "Hello" > /hello
```

Note: The `-d` option to `mkfs.ext3` does not support `uid` or `gid` values >65535. To allow writes from users with larger uids you can create the directories for your overlay with open permissions, e.g. `mkdir -p -m 777 overlay/upper overlay/work`. At runtime files and directories created in the overlay will have the correct `uid` and `gid`, but it is not possible to lock down permissions so that the overlay is only writable by certain users.

Directory overlay

A directory overlay is simpler to use than a filesystem image overlay, but a directory of modifications to a base container image cannot be transported or shared as easily as a single overlay file.

Note: For security reasons, you must be root to use a bare directory as an overlay. ext3 file system images can be used as overlays without root privileges.

Create a directory as usual:

```
$ mkdir my_overlay
```

The example below shows the directory overlay in action.

```
$ sudo singularity shell --overlay my_overlay/ ubuntu.sif

Singularity ubuntu.sif:~> mkdir /data

Singularity ubuntu.sif:~> chown user /data

Singularity ubuntu.sif:~> apt-get update && apt-get install -y vim

Singularity ubuntu.sif:~> which vim
/usr/bin/vim

Singularity ubuntu.sif:~> exit
```

Overlay embedded in SIF

It is possible to embed an overlay image in the SIF file that holds a container. This allows the read-only container image and your modifications to it to be managed as a single file. In order to do this, you must first create a file system image:

```
$ dd if=/dev/zero of=overlay.img bs=1M count=500 && \
    mkfs.ext3 overlay.img
```

Then, you can add the overlay to the SIF image using the `sif` functionality of Singularity.

```
$ singularity sif add --datatype 4 --partfs 2 --parttype 4 --partarch 2 --groupid 1␣
↪ubuntu_latest.sif overlay.img
```

Below is the explanation what each parameter means, and how it can possibly affect the operation:

- `datatype` determines what kind of an object we attach, e.g. a definition file, environment variable, signature.
- `partfs` should be set according to the partition type, e.g. SquashFS, ext3, raw.
- `parttype` determines the type of partition. In our case it is being set to overlay.
- `partarch` must be set to the architecture against you're building. In this case it's `amd64`.
- `groupid` is the ID of the container image group. In most cases there's no more than one group, therefore we can assume it is 1.

All of these options are documented within the CLI help. Access it by running `singularity sif add --help`.

After you've completed the steps above, you can shell into your container with the `--writable` option.

```
$ sudo singularity shell --writable ubuntu_latest.sif
```

Final note

You will find that your changes persist across sessions as though you were using a writable container.

```
$ singularity shell --overlay my_overlay/ ubuntu.sif

Singularity ubuntu.sif:~> ls -lasd /data
4 drwxr-xr-x 2 user root 4096 Apr  9 10:21 /data

Singularity ubuntu.sif:~> which vim
/usr/bin/vim

Singularity ubuntu.sif:~> exit
```

If you mount your container without the `--overlay` directory, your changes will be gone.

```
$ singularity shell ubuntu.sif

Singularity ubuntu.sif:~> ls /data
ls: cannot access 'data': No such file or directory

Singularity ubuntu.sif:~> which vim

Singularity ubuntu.sif:~> exit
```

To resize an overlay, standard Linux tools which manipulate ext3 images can be used. For instance, to resize the 500MB file created above to 700MB one could use the `e2fsck` and `resize2fs` utilities like so:

```
$ e2fsck -f my_overlay && \
    resize2fs my_overlay 700M
```

Hints for creating and manipulating ext3 images on your distribution are readily available online and are not treated further in this manual.

5.3 Running Services

There are *different ways* in which you can run Singularity containers. If you use commands like `run`, `exec` and `shell` to interact with processes in the container, you are running Singularity containers in the foreground. Singularity, also lets you run containers in a "detached" or "daemon" mode which can run different services in the background. A "service" is essentially a process running in the background that multiple different clients can use. For example, a web server or a database. To run services in a Singularity container one should use *instances*. A container instance is a persistent and isolated version of the container image that runs in the background.

5.3.1 Overview

Singularity v2.4 introduced the concept of *instances* allowing users to run services in Singularity. This page will help you understand instances using an elementary example followed by a more useful example running an NGINX web server using instances. In the end, you will find a more detailed example of running an instance of an API that converts URL to PDFs.

To begin with, suppose you want to run an NGINX web server outside of a container. On Ubuntu, you can simply install NGINX and start the service by:

```
$ sudo apt-get update && sudo apt-get install -y nginx

$ sudo service nginx start
```

If you were to do something like this from within a container you would also see the service start, and the web server running. But then if you were to exit the container, the process would continue to run within an unreachable mount namespace. The process would still be running, but you couldn't easily kill or interface with it. This is a called an orphan process. Singularity instances give you the ability to handle services properly.

5.3.2 Container Instances in Singularity

For demonstration, let's use an easy (though somewhat useless) example of alpine_latest.sif image from the container library:

```
$ singularity pull library://alpine
```

The above command will save the alpine image from the Container Library as `alpine_latest.sif`.

To start an instance, you should follow this procedure :

```
[command]                          [image]               [name of instance]

$ singularity instance start   alpine_latest.sif      instance1
```

This command causes Singularity to create an isolated environment for the container services to live inside. One can confirm that an instance is running by using the `instance list` command like so:

```
$ singularity instance list

INSTANCE NAME    PID      IP              IMAGE
instance1        22084                    /home/dave/instances/alpine_latest.sif
```

Note: The instances are linked with your user account. So make sure to run *all* instance commands either with or without the `sudo` privilege. If you `start` an instance with sudo then you must `list` it with sudo as well, or you will not be able to locate the instance.

If you want to run multiple instances from the same image, it's as simple as running the command multiple times with different instance names. The instance name uniquely identify instances, so they cannot be repeated.

```
$ singularity instance start alpine_latest.sif instance2

$ singularity instance start alpine_latest.sif instance3
```

And again to confirm that the instances are running as we expected:

```
$ singularity instance list

INSTANCE NAME    PID      IP              IMAGE
instance1        22084                    /home/dave/instances/alpine_latest.sif
instance2        22443                    /home/dave/instances/alpine_latest.sif
instance3        22493                    /home/dave/instances/alpine_latest.sif
```

You can also filter the instance list by supplying a pattern:

```
$ singularity instance list '*2'

INSTANCE NAME    PID      IP              IMAGE
instance2        22443                    /home/dave/instances/alpine_latest.s
```

You can use the `singularity run/exec` commands on instances:

```
$ singularity run instance://instance1

$ singularity exec instance://instance2 cat /etc/os-release
```

When using `run` with an instance URI, the `runscript` will be executed inside of the instance. Similarly with `exec`, it will execute the given command in the instance.

If you want to poke around inside of your instance, you can do a normal `singularity shell` command, but give it the instance URI:

```
$ singularity shell instance://instance3

Singularity>
```

When you are finished with your instance you can clean it up with the `instance stop` command as follows:

```
$ singularity instance stop instance1
```

If you have multiple instances running and you want to stop all of them, you can do so with a wildcard or the –all flag. The following three commands are all identical.

```
$ singularity instance stop \*

$ singularity instance stop --all

$ singularity instance stop --a
```

Note: Note that you must escape the wildcard with a backslash like this `\*` to pass it properly.

5.3.3 Nginx "Hello-world" in Singularity

The above example, although not very useful, should serve as a fair introduction to the concept of Singularity instances and running services in the background. The following illustrates a more useful example of setting up a sample NGINX web server using instances. First we will create a basic definition file (let's call it nginx.def):

```
Bootstrap: docker
From: nginx
Includecmd: no

%startscript
   nginx
```

This downloads the official NGINX Docker container, converts it to a Singularity image, and tells it to run NGINX when you start the instance. Since we're running a web server, we're going to run the following commands as root.

```
$ sudo singularity build nginx.sif nginx.def

$ sudo singularity instance start --writable-tmpfs nginx.sif web
```

Note: The above `start` command requires `sudo` because we are running a web server. Also, to let the instance write temporary files during execution, you should use `--writable-tmpfs` while starting the instance.

Just like that we've downloaded, built, and run an NGINX Singularity image. And to confirm that it's correctly running:

```
$ curl localhost

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
 body {
     width: 35em;
     margin: 0 auto;
     font-family: Tahoma, Verdana, Arial, sans-serif;
 }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Visit localhost on your browser, you should see a Welcome message!

5.3.4 Putting all together

In this section, we will demonstrate an example of packaging a service into a container and running it. The service we will be packaging is an API server that converts a web page into a PDF, and can be found here. You can build the image by following the steps described below or you can just download the final image directly from Container Library, simply run:

```
$ singularity pull url-to-pdf.sif library://sylabs/doc-examples/url-to-pdf:latest
```

Building the image

This section will describe the requirements for creating the definition file (url-to-pdf.def) that will be used to build the container image. `url-to-pdf-api` is based on a Node 8 server that uses a headless version of Chromium called Puppeteer. Let's first choose a base from which to build our container, in this case the docker image `node:8` which comes pre-installed with Node 8 has been used:

```
Bootstrap: docker
From: node:8
Includecmd: no
```

Puppeteer also requires a slew of dependencies to be manually installed in addition to Node 8, so we can add those into the `post` section as well as the installation script for the `url-to-pdf`:

```
%post

    apt-get update && apt-get install -yq gconf-service libasound2 \
        libatk1.0-0 libc6 libcairo2 libcups2 libdbus-1-3 libexpat1 \
        libfontconfig1 libgcc1 libgconf-2-4 libgdk-pixbuf2.0-0 \
        libglib2.0-0 libgtk-3-0 libnspr4 libpango-1.0-0 \
        libpangocairo-1.0-0 libstdc++6 libx11-6 libx11-xcb1 libxcb1 \
        libxcomposite1 libxcursor1 libxdamage1 libxext6 libxfixes3 libxi6 \
        libxrandr2 libxrender1 libxss1 libxtst6 ca-certificates \
        fonts-liberation libappindicator1 libnss3 lsb-release xdg-utils \
        wget curl && rm -r /var/lib/apt/lists/*
    git clone https://github.com/alvarcarto/url-to-pdf-api.git pdf_server
    cd pdf_server
    npm install
    chmod -R 0755 .
```

And now we need to define what happens when we start an instance of the container. In this situation, we want to run the commands that starts up the url-to-pdf service:

```
%startscript
    cd /pdf_server
    # Use nohup and /dev/null to completely detach server process from terminal
    nohup npm start > /dev/null 2>&1 < /dev/null &
```

Also, the `url-to-pdf` service requires some environment variables to be set, which we can do in the environment section:

```
%environment
    NODE_ENV=development
    PORT=9000
    ALLOW_HTTP=true
    URL=localhost
    export NODE_ENV PORT ALLOW_HTTP URL
```

The complete definition file will look like this:

```
Bootstrap: docker
From: node:8
Includecmd: no

%post

    apt-get update && apt-get install -yq gconf-service libasound2 \
        libatk1.0-0 libc6 libcairo2 libcups2 libdbus-1-3 libexpat1 \
        libfontconfig1 libgcc1 libgconf-2-4 libgdk-pixbuf2.0-0 \
        libglib2.0-0 libgtk-3-0 libnspr4 libpango-1.0-0 \
        libpangocairo-1.0-0 libstdc++6 libx11-6 libx11-xcb1 libxcb1 \
        libxcomposite1 libxcursor1 libxdamage1 libxext6 libxfixes3 libxi6 \
        libxrandr2 libxrender1 libxss1 libxtst6 ca-certificates \
        fonts-liberation libappindicator1 libnss3 lsb-release xdg-utils \
        wget curl && rm -r /var/lib/apt/lists/*
    git clone https://github.com/alvarcarto/url-to-pdf-api.git pdf_server
    cd pdf_server
    npm install
    chmod -R 0755 .

%startscript
    cd /pdf_server
    # Use nohup and /dev/null to completely detach server process from terminal
    nohup npm start > /dev/null 2>&1 < /dev/null &

%environment
    NODE_ENV=development
    PORT=9000
    ALLOW_HTTP=true
    URL=localhost
    export NODE_ENV PORT ALLOW_HTTP URL
```

The container can be built like so:

```
$ sudo singularity build url-to-pdf.sif url-to-pdf.def
```

Running the Service

We can now start an instance and run the service:

```
$ sudo singularity instance start url-to-pdf.sif pdf
```

Note: If there occurs an error related to port connection being refused while starting the instance or while using it later, you can try specifying different port numbers in the `%environment` section of the definition file above.

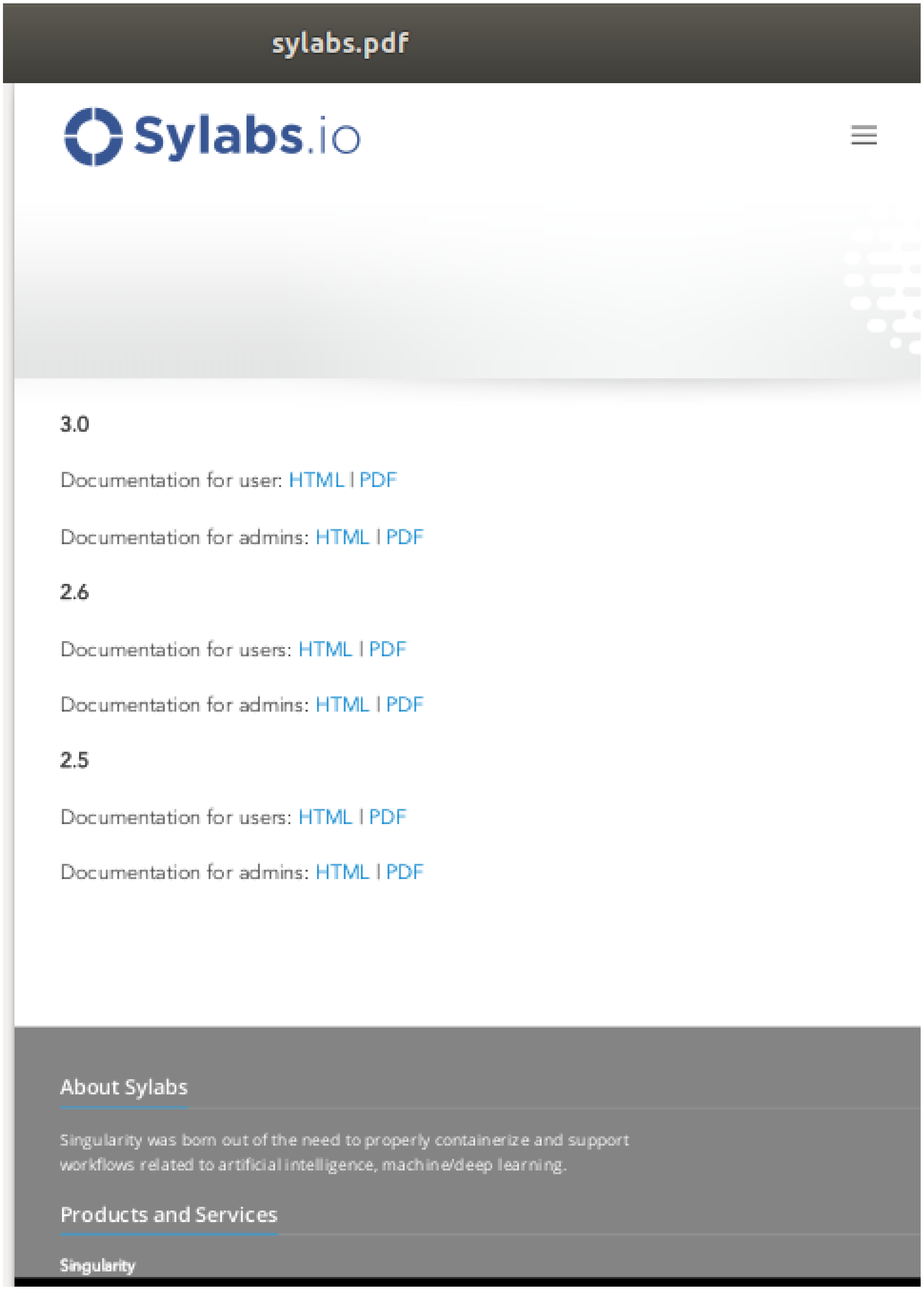
We can confirm it's working by sending the server an http request using curl:

```
$ curl -o sylabs.pdf localhost:9000/api/render?url=http://sylabs.io/docs

% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                         Dload  Upload   Total   Spent    Left  Speed

100 73750  100 73750    0     0  14583      0  0:00:05  0:00:05 --:--:-- 19130
```

You should see a PDF file being generated like the one shown below:



sylabs.pdf

Sylabs.io

3.0

Documentation for user: HTML | PDF

Documentation for admins: HTML | PDF

2.6

Documentation for users: HTML | PDF

Documentation for admins: HTML | PDF

2.5

Documentation for users: HTML | PDF

Documentation for admins: HTML | PDF

About Sylabs

Singularity was born out of the need to properly containerize and support workflows related to artificial intelligence, machine/deep learning.

Products and Services

Singularity

If you shell into the instance, you can see the running processes:

```
$ sudo singularity shell instance://pdf
Singularity: Invoking an interactive shell within container...

Singularity final.sif:/home/ysub> ps auxf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       461  0.0  0.0  18204  3188 pts/1    S    17:58   0:00 /bin/bash --norc
root       468  0.0  0.0  36640  2880 pts/1    R+   17:59   0:00  \_ ps auxf
root         1  0.0  0.1 565392 12144 ?        Sl   15:10   0:00 sinit
root        16  0.0  0.4 1113904 39492 ?       Sl   15:10   0:00 npm
root        26  0.0  0.0   4296   752 ?        S    15:10   0:00  \_ sh -c nodemon --
↪watch ./src -e js src/index.js
root        27  0.0  0.5 1179476 40312 ?       Sl   15:10   0:00      \_ node /pdf_
↪server/node_modules/.bin/nodemon --watch ./src -e js src/index.js
root        39  0.0  0.7 936444 61220 ?        Sl   15:10   0:02          \_ /usr/
↪local/bin/node src/index.js

Singularity final.sif:/home/ysub> exit
```

Making it Fancy

Now that we have confirmation that the server is working, let's make it a little cleaner. It's difficult to remember the exact `curl` command and URL syntax each time you want to request a PDF, so let's automate it. To do that, we can use Scientific Filesystem (SCIF) apps, that are integrated directly into singularity. If you haven't already, check out the Scientific Filesystem documentation to come up to speed.

First off, we're going to move the installation of the url-to-pdf into an app, so that there is a designated spot to place output files. To do that, we want to add a section to our definition file to build the server:

```
%appinstall pdf_server
    git clone https://github.com/alvarcarto/url-to-pdf-api.git pdf_server
    cd pdf_server
    npm install
    chmod -R 0755 .
```

And update our `startscript` to point to the app location:

```
%startscript
    cd /scif/apps/pdf_server/scif/pdf_server
    # Use nohup and /dev/null to completely detach server process from terminal
    nohup npm start > /dev/null 2>&1 < /dev/null &
```

Now we want to define the pdf_client app, which we will run to send the requests to the server:

```
%apprun pdf_client
    if [ -z "${1:-}" ]; then
        echo "Usage: singularity run --app pdf <instance://name> <URL> [output file]"
        exit 1
    fi
    curl -o "${SINGULARITY_APPDATA}/output/${2:-output.pdf}" "${URL}:${PORT}/api/
↪render?url=${1}"
```

As you can see, the `pdf_client` app checks to make sure that the user provides at least one argument.

The full def file will look like this:

```
Bootstrap: docker
From: node:8
Includecmd: no

%post

    apt-get update && apt-get install -yq gconf-service libasound2 \
        libatk1.0-0 libc6 libcairo2 libcups2 libdbus-1-3 libexpat1 \
        libfontconfig1 libgcc1 libgconf-2-4 libgdk-pixbuf2.0-0 \
        libglib2.0-0 libgtk-3-0 libnspr4 libpango-1.0-0 \
        libpangocairo-1.0-0 libstdc++6 libx11-6 libx11-xcb1 libxcb1 \
        libxcomposite1 libxcursor1 libxdamage1 libxext6 libxfixes3 libxi6 \
        libxrandr2 libxrender1 libxss1 libxtst6 ca-certificates \
        fonts-liberation libappindicator1 libnss3 lsb-release xdg-utils \
        wget curl && rm -r /var/lib/apt/lists/*

%appinstall pdf_server
    git clone https://github.com/alvarcarto/url-to-pdf-api.git pdf_server
    cd pdf_server
    npm install
    chmod -R 0755 .

%startscript
    cd /scif/apps/pdf_server/scif/pdf_server
    # Use nohup and /dev/null to completely detach server process from terminal
    nohup npm start > /dev/null 2>&1 < /dev/null &

%environment
    NODE_ENV=development
    PORT=9000
    ALLOW_HTTP=true
    URL=localhost
    export NODE_ENV PORT ALLOW_HTTP URL

%apprun pdf_client
    if [ -z "${1:-}" ]; then
        echo "Usage: singularity run --app pdf <instance://name> <URL> [output file]"
        exit 1
    fi
    curl -o "${SINGULARITY_APPDATA}/output/${2:-output.pdf}" "${URL}:${PORT}/api/
↪render?url=${1}"
```

Create the container as before. The `--force` option will overwrite the old container:

```
$ sudo singularity build --force url-to-pdf.sif url-to-pdf.def
```

Now that we have an output directory in the container, we need to expose it to the host using a bind mount. Once we've rebuilt the container, make a new directory called `/tmp/out` for the generated PDFs to go.

```
$ mkdir /tmp/out
```

After building the image from the edited definition file we simply start the instance:

```
$ singularity instance start --bind /tmp/out/:/output url-to-pdf.sif pdf
```

To request a pdf simply do:

```
$ singularity run --app pdf_client instance://pdf http://sylabs.io/docs sylabs.pdf
```

To confirm that it worked:

```
$ ls /tmp/out/
sylabs.pdf
```

When you are finished, use the instance stop command to close all running instances.

```
$ singularity instance stop --all
```

Note: If the service you want to run in your instance requires a bind mount, then you must pass the `--bind` option when calling `instance start`. For example, if you wish to capture the output of the `web` container instance which is placed at `/output/` inside the container you could do:

```
$ singularity instance start --bind output/dir/outside/:/output/ nginx.sif  web
```

5.3.5 System integration / PID files

If you are running services in containers you may want them to be started on boot, and shutdown gracefully automatically. This is usually performed by an init process, or another supervisor daemon installed on your host. Many init and supervisor daemons support managing processes via pid files.

You can specify a *–pid-file* option to *singularity instance start* to write the PID for an instance to the specifed file, e.g.

```
$ singularity instance start --pid-file /home/dave/alpine.pid alpine_latest.sif␣
↪instanceA

$ cat /home/dave/alpine.pid
23727
```

An example service file for an instance controlled by systemd is below. This can be used as a template to setup containerized services under systemd.

```
[Unit]
Description=Web Instance
After=network.target

[Service]
Type=forking
Restart=always
User=www-data
Group=www-data
PIDFile=/run/web-instance.pid
ExecStart=/usr/local/bin/singularity instance start --pid-file /run/web-instance.pid /
↪data/containers/web.sif web-instance
ExecStop=/usr/local/bin/singularity instance stop web-instance

[Install]
WantedBy=multi-user.target
```

Note that `Type=forking` is required here, since `instance start` starts an instance and then exits.

5.4 Environment and Metadata

Environment variables are values you can set in a session, which can be used to influence the behavior of programs. It's often considered best practice to use environment variables to pass settings to a program in a container, because they are easily set and don't rely on writing and binding in program-specific configuration files. When building a container you may need to set fixed or default environment variables. When running containers you may need to set or override environment variables.

The *metadata* of a container is information that describes the container. Singularity automatically records important information such as the definition file used to build a container. Other details such as the version of Singularity used are present as *labels* on a container. You can also specify your own to be recorded against your container.

5.4.1 Changes in Singularity 3.6

Singularity 3.6 has modified the ways in which environment variables are handled to allow long-term stability and consistency that has been lacking in prior versions. It also introduces new ways of setting environment variables, such as the `--env` and `--env-file` options.

Warning: If you have containers built with Singularity <3.6, and frequently set and override environment variables, please review this section carefully. Some behavior has changed.

Summary of changes

- When building a container, the environment defined in the base image (e.g. a Docker image) is available during the `%post` section of the build.
- An environment variable set in a container image, from the bootstrap base image, or in the `%environment` section of a definition file *will not* be overridden by a host environment variable of the same name. The `--env`, `--env-file`, or `SINGULARITYENV_` methods must be used to explicitly override a environment variable set by the container image.

5.4.2 Environment Overview

When you run a program in a container with Singularity, the environment variables that the program sees are a combination of:

- The environment variables set in the base image (e.g. Docker image) used to build the container.
- The environment variables set in the `%environment` section of the definition file used to build the container.
- *Most* of the environment variables set on your host, which are passed into the container.
- Any variables you set specifically for the container at runtime, using the `--env`, `--env-file` options, or by setting `SINGULARITYENV_` variables outside of the container.
- The `PATH` variable can be manipulated to add entries.
- Runtime variables `SINGULARITY_xxx` set by Singularity to provide information about the container.

The environment variables from the base image or definition file used to build a container always apply, but can be overridden.

You can choose to exclude passing environment variables from the host into the container with the `-e` or `--cleanenv` option.

We'll go through each place environment variables can be defined, so that you can understand how the final environment in a container is created, and can be manipulated.

If you are interested in variables available when you are *building* a container, rather than when running a container, see *build environment section*.

5.4.3 Environment from a base image

When you build a container with Singularity you might *bootstrap* from a library or Docker image, or using Linux distribution bootstrap tools such as `debootstrap`, `yum` etc.

When using `debootstrap`, `yum` etc. you are starting from a fresh install of a Linux distribution into your container. No specific environment variables will be set. If you are using a `library` or `Docker` source then you may inherit environment variables from your base image.

If I build a singularity container from the image `docker://python:3.7` then when I run the container I can see that the `PYTHON_VERSION` variable is set in the container:

```
$ singularity exec python.sif env | grep PYTHON_VERSION
PYTHON_VERSION=3.7.7
```

This happens because the `Dockerfile` used to build that container has `ENV PYTHON_VERSION 3.7.7` set inside it.

You can always override the value of these base image environment variables, if needed. See below.

5.4.4 Environment from a definition file

Environment variables can be included in your container by adding them to your definition file. Use `export` in the `%environment` section of a definition file to set a container environment variable:

```
Bootstrap: library
From: default/alpine

%environment
    export MYVAR="Hello"

%runscript
    echo $MYVAR
```

Now the value of `MYVAR` is `Hello` when the container is launched. The `%runscript` is set to echo the value.

```
$ singularity run env.sif
Hello
```

5.4.5 Environment from the host

If you have environment variables set outside of your container, on the host, then by default they will be available inside the container. Except that:

- The `PS1` shell prompt is reset for a container specific prompt.
- The `PATH` environment variable will be modified to contain default values.
- The `LD_LIBRARY_PATH` is modified to a default `/.singularity.d/libs`, that will include NVIDIA / ROCm libraries if applicable.

Also, an environment variable set on the host *will not* override a variable of the same name that has been set inside the container image.

If you *do not want* the host environment variables to pass into the container you can use the `-e` or `--cleanenv` option. This gives a clean environment inside the container, with a minimal set of environment variables for correct operation of most software.

```
$ singularity exec --cleanenv env.sif env
HOME=/home/dave
LANG=C
LD_LIBRARY_PATH=/.singularity.d/libs
PATH=/startpath:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PROMPT_COMMAND=PS1="Singularity> "; unset PROMPT_COMMAND
PS1=Singularity>
PWD=/home/dave/doc-tesrts
SINGULARITY_COMMAND=exec
SINGULARITY_CONTAINER=/home/dave/doc-tesrts/env.sif
SINGULARITY_ENVIRONMENT=/.singularity.d/env/91-environment.sh
SINGULARITY_NAME=env.sif
TERM=xterm-256color
```

Warning: If you work on a host system that sets a lot of environment variables, e.g. because you use software made available through environment modules / lmod, you may see strange behavior in your container. Check your host environment with `env` for variables such as `PYTHONPATH` that can change the way code runs, and consider using `--cleanenv`.

5.4.6 Environment from the Singularity runtime

It can be useful for a program to know when it is running in a Singularity container, and some basic information about the container environment. Singularity will automatically set a number of environment variables in a container that can be inspected by any program running in the container.

- `SINGULARITY_COMMAND` - how the container was started, e.g. `exec` / `run` / `shell`.
- `SINGULARITY_CONTAINER` - the full path to the container image.
- `SINGULARITY_ENVIRONMENT` - path inside the container to the shell script holding the container image environment settings.
- `SINGULARITY_NAME` - name of the container image, e.g. `myfile.sif` or `docker://ubuntu`.

5.4.7 Overriding environment variables

You can override variables that have been set in the container image, or define additional variables, in various ways as appropriate for your workflow.

`--env` option

New in Singularity 3.6

The `--env` option on the `run/exec/shell` commands allows you to specify environment variables as `NAME=VALUE` pairs:

```
$ singularity run env.sif
Hello

$ singularity run --env MYVAR=Goodbye env.sif
Goodbye
```

Separate multiple variables with commas, e.g. `--env MYVAR=A,MYVAR2=B`, and use shell quoting / shell escape if your variables include special characters.

`--env-file` option

New in Singularity 3.6

The `--env-file` option lets you provide a file that contains environment variables as `NAME=VALUE` pairs, e.g.:

```
$ cat myenvs
MYVAR="Hello from a file"

$ singularity run --env-file myenvs env.sif
Hello from a file
```

`SINGULARITYENV_` prefix

If you export an environment variable on your host called `SINGULARITYENV_xxx` *before* you run a container, then it will set the environment variable `xxx` inside the container:

```
$ singularity run env.sif
Hello

$ export SINGULARITYENV_MYVAR="Overridden"
$ singularity run env.sif
Overridden
```

Manipulating `PATH`

`PATH` is a special environment variable that tells a system where to look for programs that can be run. `PATH` contains multiple filesytem locations (paths) separated by colons. When you ask to run a program `myprog`, the system looks through these locations one by one, until it finds `myprog`.

To ensure containers work correctly, when a host `PATH` might contain a lot of host-specific locations that are not present in the container, Singularity will ensure `PATH` in the container is set to a default.

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

This covers the standard locations for software installed using a system package manager in most Linux distributions. If you have software installed elsewhere in the container, then you can override this by setting `PATH` in the container definition `%environment` block.

If your container depends on things that are bind mounted into it, or you have another need to modify the `PATH` variable when starting a container, you can do so with `SINGULARITYENV_APPEND_PATH` or `SINGULARITYENV_PREPEND_PATH`.

If you set a variable on your host called `SINGULARITYENV_APPEND_PATH` then its value will be appended (added to the end) of the `PATH` variable in the container.

```
$ singularity exec env.sif sh -c 'echo $PATH'
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

$ export SINGULARITYENV_APPEND_PATH="/endpath"
$ singularity exec env.sif sh -c 'echo $PATH'
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/endpath
```

Alternatively you could use the `--env` option to set a `APPEND_PATH` variable, e.g. `--env APPEND_PATH=/endpath`.

If you set a variable on your host called `SINGULARITYENV_PREPEND_PATH` then its value will be prepended (added to the start) of the `PATH` variable in the container.

```
$ singularity exec env.sif sh -c 'echo $PATH'
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

$ export SINGULARITYENV_PREPEND_PATH="/startpath"
$ singularity exec env.sif sh -c 'echo $PATH'
/startpath:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Alternatively you could use the `--env` option to set a `PREPEND_PATH` variable, e.g. `--env PREPEND_PATH=/startpath`.

Evaluating container variables

When setting environment variables with `--env` etc. you can specify an escaped variable name, e.g. `\$PATH` to evaluate the value of that variable in the container.

For example, `--env PATH="\$PATH:/endpath"` would have the same effect as `--env APPEND_PATH="/endpath"`.

Environment Variable Precedence

When a container is run with Singularity 3.6, the container environment is constructed in the following order:

- Clear the environment, keeping just `HOME` and `SINGULARITY_APPNAME`.
- Take Docker defined environment variables, where Docker was the base image source.
- If `PATH` is not defined set the Singularity default `PATH` *or*
- If `PATH` is defined, add any missing path parts from Singularity defaults
- Take environment variables defined explicitly in the image (`%environment`). These can override any previously set values.
- Set SCIF (`--app`) environment variables
- Set base environment essential vars (`PS1` and `LD_LIBRARY_PATH`)
- Inject `SINGULARITYENV_` / `--env` / `--env-file` variables so they can override or modify any previous values:

- Source any remaining scripts from `/singularity.d/env`

5.4.8 Container Metadata

Each Singularity container has metadata describing the container, how it was built, etc. This metadata includes the definition file used to build the container and labels, which are specific pieces of information set automatically or explicitly when the container is built.

For containers that are generated with Singularity version 3.0 and later, default labels are represented using the rc1 Label Schema.

Custom Labels

You can add custom labels to your container using the `%labels` section in a definition file:

```
Bootstrap: docker
From: ubuntu: latest

%labels
  OWNER Joana
```

Inspecting Metadata

The `inspect` command gives you the ability to view the labels and/or other metadata that were added to your container when it was built.

`-l`/ `--labels`

Running inspect without any options, or with the `-l` or `--labels` options will display any labels set on the container

```
$ singularity inspect jupyter.sif
    {
        "OWNER": "Joana"
        "org.label-schema.build-date": "Friday_21_December_2018_0:49:50_CET",
        "org.label-schema.schema-version": "1.0",
        "org.label-schema.usage": "/.singularity.d/runscript.help",
        "org.label-schema.usage.singularity.deffile.bootstrap": "library",
        "org.label-schema.usage.singularity.deffile.from": "debian:9",
        "org.label-schema.usage.singularity.runscript.help": "/.singularity.d/
↪runscript.help",
        "org.label-schema.usage.singularity.version": "3.0.1-236.g2453fdfe"
    }
```

We can easily see when the container was built, the source of the base image, and the exact version of Singularity that was used to build it.

The custom label `OWNER` that we set in our definition file is also visible.

-d / --deffile

The -d or -deffile flag shows the definition file(s) that were used to build the container.

```
$ singularity inspect --deffile jupyter.sif
```

And the output would look like:

```
Bootstrap: library
From: debian:9

%help
    Container with Anaconda 2 (Conda 4.5.11 Canary) and Jupyter Notebook 5.6.0 for␣
↪Debian 9.x (Stretch).
    This installation is based on Python 2.7.15

%environment
    JUP_PORT=8888
    JUP_IPNAME=localhost
    export JUP_PORT JUP_IPNAME

%startscript
    PORT=""
    if [ -n "$JUP_PORT" ]; then
    PORT="--port=${JUP_PORT}"
    fi

    IPNAME=""
    if [ -n "$JUP_IPNAME" ]; then
    IPNAME="--ip=${JUP_IPNAME}"
    fi

    exec jupyter notebook --allow-root ${PORT} ${IPNAME}

%setup
    #Create the .condarc file where the environments/channels from conda are␣
↪specified, these are pulled with preference to root
    cd /
    touch .condarc

%post
    echo 'export RANDOM=123456' >>$SINGULARITY_ENVIRONMENT
    #Installing all dependencies
    apt-get update && apt-get -y upgrade
    apt-get -y install \
    build-essential \
    wget \
    bzip2 \
    ca-certificates \
    libglib2.0-0 \
    libxext6 \
    libsm6 \
    libxrender1 \
    git
    rm -rf /var/lib/apt/lists/*
    apt-get clean
    #Installing Anaconda 2 and Conda 4.5.11
```

(continues on next page)

(continued from previous page)

```
    wget -c https://repo.continuum.io/archive/Anaconda2-5.3.0-Linux-x86_64.sh
    /bin/bash Anaconda2-5.3.0-Linux-x86_64.sh -bfp /usr/local
    #Conda configuration of channels from .condarc file
    conda config --file /.condarc --add channels defaults
    conda config --file /.condarc --add channels conda-forge
    conda update conda
    #List installed environments
    conda list
```

Which is the definition file for the `jupyter.sif` container.

`-r` / `--runscript`

The `-r` or `--runscript` option shows the runscript for the image.

```
$ singularity inspect --runscript jupyter.sif
```

And the output would look like:

```
#!/bin/sh
OCI_ENTRYPOINT=""
OCI_CMD="bash"
# ENTRYPOINT only - run entrypoint plus args
if [ -z "$OCI_CMD" ] && [ -n "$OCI_ENTRYPOINT" ]; then
SINGULARITY_OCI_RUN="${OCI_ENTRYPOINT} $@"
fi

# CMD only - run CMD or override with args
if [ -n "$OCI_CMD" ] && [ -z "$OCI_ENTRYPOINT" ]; then
if [ $# -gt 0 ]; then
    SINGULARITY_OCI_RUN="$@"
else
    SINGULARITY_OCI_RUN="${OCI_CMD}"
fi
fi

# ENTRYPOINT and CMD - run ENTRYPOINT with CMD as default args
# override with user provided args
if [ $# -gt 0 ]; then
    SINGULARITY_OCI_RUN="${OCI_ENTRYPOINT} $@"
else
    SINGULARITY_OCI_RUN="${OCI_ENTRYPOINT} ${OCI_CMD}"
fi

exec $SINGULARITY_OCI_RUN
```

`-t` / `--test`

The `-t` or `--test` flag shows the test script for the image.

```
$ singularity inspect --test jupyter.sif
```

This will output the corresponding `%test` section from the definition file.

`-e` / `--environment`

The `-e` or `--environment` flag shows the environment variables that are defined in the container image. These may be set from one or more environment files, depending on how the container was built.

```
$ singularity inspect --environment jupyter.sif
```

And the output would look like:

```
==90-environment.sh==
#!/bin/sh

JUP_PORT=8888
JUP_IPNAME=localhost
export JUP_PORT JUP_IPNAME
```

`-h` / `--helpfile`

The `-h` or `-helpfile` flag will show the container's description in the `%help` section of its definition file.

You can call it this way:

```
$ singularity inspect --helpfile jupyter.sif
```

And the output would look like:

```
Container with Anaconda 2 (Conda 4.5.11 Canary) and Jupyter Notebook 5.6.0 for Debian␣
↪9.x (Stretch).
This installation is based on Python 2.7.15
```

`-j` / `--json`

This flag gives you the possibility to output your labels in a JSON format.

You can call it this way:

```
$ singularity inspect --json jupyter.sif
```

And the output would look like:

```
{
        "attributes": {
                "labels": "{\n\t\"org.label-schema.build-date\": \"Friday_21_
↪December_2018_0:49:50_CET\",\n\t\"org.label-schema.schema-version\": \"1.0\",\n\t\
↪"org.label-schema.usage\": \"/.singularity.d/runscript.help\",\n\t\"org.label-
↪schema.usage.singularity.deffile.bootstrap\": \"library\",\n\t\"org.label-schema.
↪usage.singularity.deffile.from\": \"debian:9\",\n\t\"org.label-schema.usage.
↪singularity.runscript.help\": \"/.singularity.d/runscript.help\",\n\t\"org.label-
↪schema.usage.singularity.version\": \"3.0.1-236.g2453fdfe\"\n}",
```

(continues on next page)

(continued from previous page)

```
        },
        "type": "container"
}
```

5.4.9 /.singularity.d directory

The `/.singularity.d` directory in a container contains scripts and environment files that are used when a container is executed.

You should not manually modify files under `/.singularity.d`, from your definition file during builds, or directly within your container image. Recent 3.x versions of Singularity replace older action scripts dynamically, at runtime, to support new features. In the longer term, metadata will be moved outside of the container, and stored only in the SIF file metadata descriptor.

```
/.singularity.d/

├── actions
│   ├── exec
│   ├── run
│   ├── shell
│   ├── start
│   └── test
├── env
│   ├── 01-base.sh
|   ├── 10-docker2singularity.sh
│   ├── 90-environment.sh
│   ├── 91-environment.sh
|   ├── 94-appsbase.sh
│   ├── 95-apps.sh
│   └── 99-base.sh
├── labels.json
├── libs
├── runscript
├── runscript.help
├── Singularity
└── startscript
```

- **actions**: This directory contains helper scripts to allow the container to carry out the action commands. (e.g. `exec` , `run` or `shell`). In later versions of Singularity, these files may be dynamically written at runtime, *and should not be modified* in the container.
- **env**: All `*.sh` files in this directory are sourced in alpha-numeric order when the container is started. For legacy purposes there is a symbolic link called `/environment` that points to `/.singularity.d/env/90-environment.sh`. Whenever possible, avoid modifying or creating environment files manually to prevent potential issues building & running containers with future versions of Singularity. Beginning with Singularity 3.6, additional facilities such as `--env` and `--env-file` are available to allow manipulation of the container environment at runtime.
- **labels.json**: The json file that stores a containers labels described above.
- **libs**: At runtime the user may request some host-system libraries to be mapped into the container (with the `--nv` option for example). If so, this is their destination.
- **runscript**: The commands in this file will be executed when the container is invoked with the `run` command or called as an executable. For legacy purposes there is a symbolic link called `/singularity` that points to this file.

- **runscript.help**: Contains the description that was added in the `%help` section.
- **Singularity**: This is the definition file that was used to generate the container. If more than 1 definition file was used to generate the container additional Singularity files will appear in numeric order in a sub-directory called `bootstrap_history`.
- **startscript**: The commands in this file will be executed when the container is invoked with the `instance start` command.

5.5 OCI Runtime Support

5.5.1 Overview

OCI is an acronym for the Open Containers Initiative - an independent organization whose mandate is to develop open standards relating to containerization. To date, standardization efforts have focused on container formats and runtimes. Singularity's compliance with respect to the OCI Image Specification is considered in detail *elsewhere*. It is Singularity's compliance with the OCI Runtime Specification that is of concern here.

Briefly, compliance with respect to the OCI Runtime Specification is addressed in Singularity through the introduction of the `oci` command group. Although this command group can, in principle, be used to provide a runtime that supports end users, in this initial documentation effort, emphasis is placed upon interoperability with Kubernetes; more specifically, interoperability with Kubernetes via the Singularity Container Runtime Interface.

Owing to this restricted focus, a subset of the Singularity `oci` command group receives attention here; specifically:

- Mounting and unmounting OCI filesystem bundles
- Creating OCI compliant container instances

Some context for integration with Kubernetes via the Singularity CRI is provided at the end of the section.

Note: All commands in the `oci` command group require `root` privileges.

5.5.2 Mounted OCI Filesystem Bundles

Mounting an OCI Filesystem Bundle

BusyBox is used here for the purpose of illustration.

Suppose the Singularity Image Format (SIF) file `busybox_latest.sif` exists locally. (Recall:

```
$ singularity pull docker://busybox
INFO:    Starting build...
Getting image source signatures
Copying blob sha256:fc1a6b909f82ce4b72204198d49de3aaf757b3ab2bb823cb6e47c416b97c5985
 738.13 KiB / 738.13 KiB [==================================================] 0s
Copying config sha256:5fffaf1f2c1830a6a8cf90eb27c7a1a8476b8c49b4b6261a20d6257d031ce4f3
 575 B / 575 B [============================================================] 0s
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: busybox_latest.sif
```

This is one way to bootstrap creation of this image in SIF that *retains* a local copy - i.e., a local copy of the SIF file *and* a cached copy of the OCI blobs. Additional approaches and details can be found in the section *Support for Docker and OCI*).

For the purpose of boostrapping the creation of an OCI compliant container, this SIF file can be mounted as follows:

```
$ sudo singularity oci mount ./busybox_latest.sif /var/tmp/busybox
```

By issuing the `mount` command, the root filesystem encapsulated in the SIF file `busybox_latest.sif` is mounted on `/var/tmp/busybox` as an `overlay` file system,

```
$ sudo df -k
Filesystem                   1K-blocks    Used Available Use% Mounted on
udev                            475192       0    475192   0% /dev
tmpfs                           100916    1604     99312   2% /run
/dev/mapper/vagrant--vg-root  19519312 2620740  15883996  15% /
tmpfs                           504560       0    504560   0% /dev/shm
tmpfs                             5120       0      5120   0% /run/lock
tmpfs                           504560       0    504560   0% /sys/fs/cgroup
tmpfs                           100912       0    100912   0% /run/user/900
overlay                       19519312 2620740  15883996  15% /var/tmp/busybox/rootfs
```

with permissions as follows:

```
$ sudo ls -ld /var/tmp/busybox
drwx------ 4 root root 4096 Apr  4 14:30 /var/tmp/busybox
```

Content of an OCI Compliant Filesystem Bundle

The *expected* contents of the mounted filesystem are as follows:

```
$ sudo ls -la /var/tmp/busybox
total 28
drwx------ 4 root root 4096 Apr  4 14:30 .
drwxrwxrwt 4 root root 4096 Apr  4 14:30 ..
-rw-rw-rw- 1 root root 9879 Apr  4 14:30 config.json
drwx------ 4 root root 4096 Apr  4 14:30 overlay
drwx------ 1 root root 4096 Apr  4 14:30 rootfs
```

From the perspective of the OCI runtime specification, this content is expected because it prescribes a

> "… a format for encoding a container as a **filesystem bundle** - a set of files organized in a certain way, and containing all the necessary data and metadata for any compliant runtime to perform all standard operations against it."

Critical to compliance with the specification is the presence of the following *mandatory* artifacts residing locally in a single directory:

1. The `config.json` file - a file of configuration data that must reside in the root of the bundle directory under this name
2. The container's root filesystem - a referenced directory

Note: Because the directory itself, i.e., `/var/tmp/busybox` is *not* part of the bundle, the mount point can be chosen arbitrarily.

The filtered `config.json` file corresponding to the OCI mounted `busybox_latest.sif` container can be detailed as follows via `$ sudo cat /var/tmp/busybox/config.json | jq`:

```
{
  "ociVersion": "1.0.1-dev",
  "process": {
    "user": {
      "uid": 0,
      "gid": 0
    },
    "args": [
      "/.singularity.d/actions/run"
    ],
    "env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "TERM=xterm"
    ],
    "cwd": "/",
    "capabilities": {
      "bounding": [
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
        "CAP_MKNOD",
        "CAP_NET_RAW",
        "CAP_SETGID",
        "CAP_SETUID",
        "CAP_SETFCAP",
        "CAP_SETPCAP",
        "CAP_NET_BIND_SERVICE",
        "CAP_SYS_CHROOT",
        "CAP_KILL",
        "CAP_AUDIT_WRITE"
      ],
      "effective": [
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
        "CAP_MKNOD",
        "CAP_NET_RAW",
        "CAP_SETGID",
        "CAP_SETUID",
        "CAP_SETFCAP",
        "CAP_SETPCAP",
        "CAP_NET_BIND_SERVICE",
        "CAP_SYS_CHROOT",
        "CAP_KILL",
        "CAP_AUDIT_WRITE"
      ],
      "inheritable": [
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
        "CAP_MKNOD",
        "CAP_NET_RAW",
```

(continues on next page)

(continued from previous page)

```
        "CAP_SETGID",
        "CAP_SETUID",
        "CAP_SETFCAP",
        "CAP_SETPCAP",
        "CAP_NET_BIND_SERVICE",
        "CAP_SYS_CHROOT",
        "CAP_KILL",
        "CAP_AUDIT_WRITE"
      ],
      "permitted": [
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
        "CAP_MKNOD",
        "CAP_NET_RAW",
        "CAP_SETGID",
        "CAP_SETUID",
        "CAP_SETFCAP",
        "CAP_SETPCAP",
        "CAP_NET_BIND_SERVICE",
        "CAP_SYS_CHROOT",
        "CAP_KILL",
        "CAP_AUDIT_WRITE"
      ],
      "ambient": [
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
        "CAP_MKNOD",
        "CAP_NET_RAW",
        "CAP_SETGID",
        "CAP_SETUID",
        "CAP_SETFCAP",
        "CAP_SETPCAP",
        "CAP_NET_BIND_SERVICE",
        "CAP_SYS_CHROOT",
        "CAP_KILL",
        "CAP_AUDIT_WRITE"
      ]
    },
    "rlimits": [
      {
        "type": "RLIMIT_NOFILE",
        "hard": 1024,
        "soft": 1024
      }
    ]
  },
  "root": {
    "path": "/var/tmp/busybox/rootfs"
  },
  "hostname": "mrsdalloway",
  "mounts": [
    {
      "destination": "/proc",
```

(continues on next page)

(continued from previous page)

```
      "type": "proc",
      "source": "proc"
    },
    {
      "destination": "/dev",
      "type": "tmpfs",
      "source": "tmpfs",
      "options": [
        "nosuid",
        "strictatime",
        "mode=755",
        "size=65536k"
      ]
    },
    {
      "destination": "/dev/pts",
      "type": "devpts",
      "source": "devpts",
      "options": [
        "nosuid",
        "noexec",
        "newinstance",
        "ptmxmode=0666",
        "mode=0620",
        "gid=5"
      ]
    },
    {
      "destination": "/dev/shm",
      "type": "tmpfs",
      "source": "shm",
      "options": [
        "nosuid",
        "noexec",
        "nodev",
        "mode=1777",
        "size=65536k"
      ]
    },
    {
      "destination": "/dev/mqueue",
      "type": "mqueue",
      "source": "mqueue",
      "options": [
        "nosuid",
        "noexec",
        "nodev"
      ]
    },
    {
      "destination": "/sys",
      "type": "sysfs",
      "source": "sysfs",
      "options": [
        "nosuid",
        "noexec",
        "nodev",
```

(continues on next page)

(continued from previous page)

```
        "ro"
      ]
    }
  ],
  "linux": {
    "resources": {
      "devices": [
        {
          "allow": false,
          "access": "rwm"
        }
      ]
    },
    "namespaces": [
      {
        "type": "pid"
      },
      {
        "type": "network"
      },
      {
        "type": "ipc"
      },
      {
        "type": "uts"
      },
      {
        "type": "mount"
      }
    ],
    "seccomp": {
      "defaultAction": "SCMP_ACT_ERRNO",
      "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
      ],
      "syscalls": [
        {
          "names": [
            "accept",
            "accept4",
            "access",
            "alarm",
            "bind",
            "brk",
            "capget",
            "capset",
            "chdir",
            "chmod",
            "chown",
            "chown32",
            "clock_getres",
            "clock_gettime",
            "clock_nanosleep",
            "close",
            "connect",
```

(continues on next page)

(continued from previous page)

```
            "copy_file_range",
            "creat",
            "dup",
            "dup2",
            "dup3",
            "epoll_create",
            "epoll_create1",
            "epoll_ctl",
            "epoll_ctl_old",
            "epoll_pwait",
            "epoll_wait",
            "epoll_wait_old",
            "eventfd",
            "eventfd2",
            "execve",
            "execveat",
            "exit",
            "exit_group",
            "faccessat",
            "fadvise64",
            "fadvise64_64",
            "fallocate",
            "fanotify_mark",
            "fchdir",
            "fchmod",
            "fchmodat",
            "fchown",
            "fchown32",
            "fchownat",
            "fcntl",
            "fcntl64",
            "fdatasync",
            "fgetxattr",
            "flistxattr",
            "flock",
            "fork",
            "fremovexattr",
            "fsetxattr",
            "fstat",
            "fstat64",
            "fstatat64",
            "fstatfs",
            "fstatfs64",
            "fsync",
            "ftruncate",
            "ftruncate64",
            "futex",
            "futimesat",
            "getcpu",
            "getcwd",
            "getdents",
            "getdents64",
            "getegid",
            "getegid32",
            "geteuid",
            "geteuid32",
            "getgid",
```

(continues on next page)

(continued from previous page)

```
            "getgid32",
            "getgroups",
            "getgroups32",
            "getitimer",
            "getpeername",
            "getpgid",
            "getpgrp",
            "getpid",
            "getppid",
            "getpriority",
            "getrandom",
            "getresgid",
            "getresgid32",
            "getresuid",
            "getresuid32",
            "getrlimit",
            "get_robust_list",
            "getrusage",
            "getsid",
            "getsockname",
            "getsockopt",
            "get_thread_area",
            "gettid",
            "gettimeofday",
            "getuid",
            "getuid32",
            "getxattr",
            "inotify_add_watch",
            "inotify_init",
            "inotify_init1",
            "inotify_rm_watch",
            "io_cancel",
            "ioctl",
            "io_destroy",
            "io_getevents",
            "ioprio_get",
            "ioprio_set",
            "io_setup",
            "io_submit",
            "ipc",
            "kill",
            "lchown",
            "lchown32",
            "lgetxattr",
            "link",
            "linkat",
            "listen",
            "listxattr",
            "llistxattr",
            "_llseek",
            "lremovexattr",
            "lseek",
            "lsetxattr",
            "lstat",
            "lstat64",
            "madvise",
            "memfd_create",
```

(continues on next page)

(continued from previous page)

```
            "mincore",
            "mkdir",
            "mkdirat",
            "mknod",
            "mknodat",
            "mlock",
            "mlock2",
            "mlockall",
            "mmap",
            "mmap2",
            "mprotect",
            "mq_getsetattr",
            "mq_notify",
            "mq_open",
            "mq_timedreceive",
            "mq_timedsend",
            "mq_unlink",
            "mremap",
            "msgctl",
            "msgget",
            "msgrcv",
            "msgsnd",
            "msync",
            "munlock",
            "munlockall",
            "munmap",
            "nanosleep",
            "newfstatat",
            "_newselect",
            "open",
            "openat",
            "pause",
            "pipe",
            "pipe2",
            "poll",
            "ppoll",
            "prctl",
            "pread64",
            "preadv",
            "prlimit64",
            "pselect6",
            "pwrite64",
            "pwritev",
            "read",
            "readahead",
            "readlink",
            "readlinkat",
            "readv",
            "recv",
            "recvfrom",
            "recvmmsg",
            "recvmsg",
            "remap_file_pages",
            "removexattr",
            "rename",
            "renameat",
            "renameat2",
```

(continues on next page)

(continued from previous page)

```
            "restart_syscall",
            "rmdir",
            "rt_sigaction",
            "rt_sigpending",
            "rt_sigprocmask",
            "rt_sigqueueinfo",
            "rt_sigreturn",
            "rt_sigsuspend",
            "rt_sigtimedwait",
            "rt_tgsigqueueinfo",
            "sched_getaffinity",
            "sched_getattr",
            "sched_getparam",
            "sched_get_priority_max",
            "sched_get_priority_min",
            "sched_getscheduler",
            "sched_rr_get_interval",
            "sched_setaffinity",
            "sched_setattr",
            "sched_setparam",
            "sched_setscheduler",
            "sched_yield",
            "seccomp",
            "select",
            "semctl",
            "semget",
            "semop",
            "semtimedop",
            "send",
            "sendfile",
            "sendfile64",
            "sendmmsg",
            "sendmsg",
            "sendto",
            "setfsgid",
            "setfsgid32",
            "setfsuid",
            "setfsuid32",
            "setgid",
            "setgid32",
            "setgroups",
            "setgroups32",
            "setitimer",
            "setpgid",
            "setpriority",
            "setregid",
            "setregid32",
            "setresgid",
            "setresgid32",
            "setresuid",
            "setresuid32",
            "setreuid",
            "setreuid32",
            "setrlimit",
            "set_robust_list",
            "setsid",
            "setsockopt",
```

(continues on next page)

(continued from previous page)

```
                "set_thread_area",
                "set_tid_address",
                "setuid",
                "setuid32",
                "setxattr",
                "shmat",
                "shmctl",
                "shmdt",
                "shmget",
                "shutdown",
                "sigaltstack",
                "signalfd",
                "signalfd4",
                "sigreturn",
                "socket",
                "socketcall",
                "socketpair",
                "splice",
                "stat",
                "stat64",
                "statfs",
                "statfs64",
                "symlink",
                "symlinkat",
                "sync",
                "sync_file_range",
                "syncfs",
                "sysinfo",
                "syslog",
                "tee",
                "tgkill",
                "time",
                "timer_create",
                "timer_delete",
                "timerfd_create",
                "timerfd_gettime",
                "timerfd_settime",
                "timer_getoverrun",
                "timer_gettime",
                "timer_settime",
                "times",
                "tkill",
                "truncate",
                "truncate64",
                "ugetrlimit",
                "umask",
                "uname",
                "unlink",
                "unlinkat",
                "utime",
                "utimensat",
                "utimes",
                "vfork",
                "vmsplice",
                "wait4",
                "waitid",
                "waitpid",
```

(continues on next page)

(continued from previous page)

```
            "write",
            "writev"
          ],
          "action": "SCMP_ACT_ALLOW"
        },
        {
          "names": [
            "personality"
          ],
          "action": "SCMP_ACT_ALLOW",
          "args": [
            {
              "index": 0,
              "value": 0,
              "op": "SCMP_CMP_EQ"
            },
            {
              "index": 0,
              "value": 8,
              "op": "SCMP_CMP_EQ"
            },
            {
              "index": 0,
              "value": 4294967295,
              "op": "SCMP_CMP_EQ"
            }
          ]
        },
        {
          "names": [
            "chroot"
          ],
          "action": "SCMP_ACT_ALLOW"
        },
        {
          "names": [
            "clone"
          ],
          "action": "SCMP_ACT_ALLOW",
          "args": [
            {
              "index": 0,
              "value": 2080505856,
              "op": "SCMP_CMP_MASKED_EQ"
            }
          ]
        },
        {
          "names": [
            "arch_prctl"
          ],
          "action": "SCMP_ACT_ALLOW"
        },
        {
          "names": [
            "modify_ldt"
          ],
```

(continues on next page)

(continued from previous page)

```
            "action": "SCMP_ACT_ALLOW"
          }
        ]
      }
    }
}
```

Furthermore, and through use of `$ sudo cat /var/tmp/busybox/config.json | jq [.root.path]`, the property

```
[
        "/var/tmp/busybox/rootfs"
]
```

identifies `/var/tmp/busybox/rootfs` as the container's root filesystem, as required by the standard; this filesystem has contents:

```
$ sudo ls /var/tmp/busybox/rootfs
bin  dev  environment  etc  home  proc  root  singularity  sys  tmp  usr  var
```

Note: `environment` and `singularity` above are symbolic links to the `.singularity.d` directory.

Beyond `root.path`, the `config.json` file includes a multitude of additional properties - for example:

- `ociVersion` - a mandatory property that identifies the version of the OCI runtime specification that the bundle is compliant with
- `process` - an optional property that specifies the container process. When invoked via Singularity, subproperties such as `args` are populated by making use of the contents of the `.singularity.d` directory, e.g. via `$ sudo cat /var/tmp/busybox/config.json | jq [.process.args]`:

```
[
  [
    "/.singularity.d/actions/run"
  ]
]
```

where `run` equates to the *familiar runscript* for this container. If image creation is bootstrapped via a Docker or OCI agent, Singularity will make use of `ENTRYPOINT` or `CMD` (from the OCI image) to populate `args`; for additional discussion, please refer to *Directing Execution* in the section *Support for Docker and OCI*.

For a comprehensive discussion of all the `config.json` file properties, refer to the implementation guide.

Technically, the `overlay` directory was *not* content expected of an OCI compliant filesystem bundle. As detailed in the section dedicated to Persistent Overlays, these directories allow for the introduction of a writable file system on an otherwise immutable read-only container; thus they permit the illusion of read-write access.

Note: SIF is stated to be an extensible format; by encapsulating a filesystem bundle that conforms with the OCI runtime specification, this extensibility is evident.

5.5.3 Creating OCI Compliant Container Instances

SIF files encapsulate the OCI runtime. By 'OCI mounting' a SIF file (see above), this encapsulated runtime is revealed; please refer to the note below for additional details. Once revealed, the filesystem bundle can be used to bootstrap the creation of an OCI compliant container instance as follows:

```
$ sudo singularity oci create -b /var/tmp/busybox busybox1
```

Note: Data for the `config.json` file exists within the SIF file as a descriptor for images pulled or built from Docker/OCI registries. For images sourced elsewhere, a default `config.json` file is created when the `singularity oci mount ...` command is issued.

Upon invocation, `singularity oci mount ...` also mounts the root filesystem stored in the SIF file on `/bundle/rootfs`, and establishes an overlay filesystem on the mount point `/bundle/overlay`.

In this example, the filesystem bundle is located in the directory `/var/tmp/busybox` - i.e., the mount point identified above with respect to 'OCI mounting'. The `config.json` file, along with the `rootfs` and `overlay` filesystems, are all employed in the bootstrap process. The instance is named `busybox1` in this example.

Note: The outcome of this creation request is truly a container **instance**. Multiple instances of the same container can easily be created by simply changing the name of the instance upon subsequent invocation requests.

The `state` of the container instance can be determined via `$ sudo singularity oci state busybox1`:

```
{
"ociVersion": "1.0.1-dev",
"id": "busybox1",
"status": "created",
"pid": 6578,
"bundle": "/var/tmp/busybox",
"createdAt": 1554389921452964253,
"attachSocket": "/var/run/singularity/instances/root/busybox1/attach.sock",
"controlSocket": "/var/run/singularity/instances/root/busybox1/control.sock"
}
```

Container state, as conveyed via these properties, is in compliance with the OCI runtime specification as detailed here.

The `create` command has a number of options available. Of these, real-time logging to a file is likely to be of particular value - e.g., in deployments where auditing requirements exist.

5.5.4 Unmounting OCI Filesystem Bundles

To unmount a mounted OCI filesystem bundle, the following command should be issued:

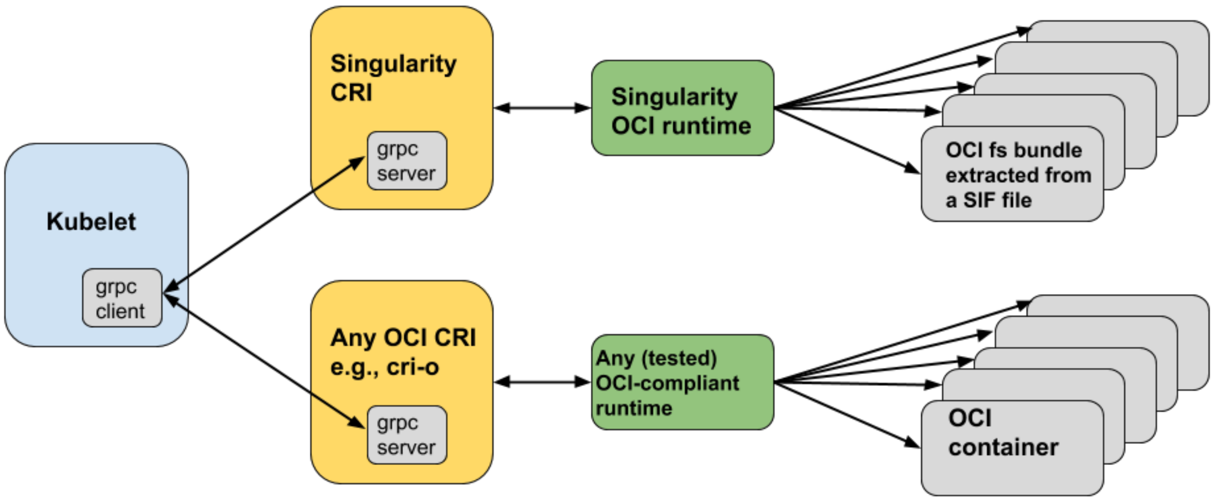
```
$ sudo singularity oci umount /var/tmp/busybox
```

Note: The argument provided to `oci umount` above is the name of the bundle path, `/var/tmp/busybox`, as opposed to the mount point for the overlay filesystem, `/var/tmp/busybox/rootfs`.

5.5.5 Kubernetes Integration

As noted at the *outset here*, in documenting support for an OCI runtime in Singularity, the impetus is initially derived from the requirement to integrate with Kubernetes. Simply stated, Kubernetes is an open-source system for orchestrating containers; developed originally at Google, Kubernetes was contributed as seed technology to the Cloud Native Compute Foundation (CNCF). At this point, Kubernetes is regarded as a Graduated Project by CNCF, and is being used widely in production deployments. Even though Kubernetes emphasizes an orientation around services, it is appealing to those seeking to orchestrate containers having compute-driven requirements. Furthermore, emerging classes of workload in AI for example, appear to have requirements that are best addressed by a combination of service and traditional HPC infrastructures. Thus there is ample existing, as well as emerging, interest in integrating Singularity containers with Kubernetes.

The connection with support for the OCI runtime documented here, within the context of a Singularity-Kubernetes integration, can be best established through an architectural schematic. Dating back to the introduction of a Container Runtime Interface (CRI) for Kubernetes in late 2016, the schematic below is a modified version of the original presented in a Kubernetes blog post. The lower branch of this schematic is essentially a reproduction of the original; it does however, place emphasis on OCI compliance in terms of the CRI and containers (the runtime as well as their instances).



From this schematic it is evident that integrating Singularity containers with Kubernetes requires the following efforts:

1. Implementation of a CRI for Singularity
2. Implementation of an OCI runtime in Singularity

The implementation of a CRI for Singularity is the emphasis of a separate and distinct open source project; the implementation of this CRI is documented here. For the rationale conveyed through the architectural schematic, Singularity CRI's dependence upon Singularity with OCI runtime support is made clear as an installation prerequisite. User-facing documentation for Singularity CRI details usage in a Kubernetes context - usage, of course, that involves orchestration of a Singularity container obtained from the Sylabs Cloud Container Library. Because the entire Kubernetes-based deployment can exist within a single instance of a Singularity container, Singularity CRI can be easily evaluated via Sykube; inspired by Minikube, use of Sykube is included in the documentation for Singularity CRI.

Documenting the implementation of an OCI-compliant runtime for Singularity has been the emphasis here. Although this standalone runtime can be used by end users independent of anything to do with Singularity and Kubernetes, the primary purpose here has been documenting it within this integrated context. In other words, by making use of the OCI runtime presented by Singularity, commands originating from Kubernetes (see, e.g., Basic Usage in the Singularity CRI documentation) have impact ultimately on Singularity containers via the CRI. Singularity CRI is implemented as a gRPC server - i.e., a persistent service available to Kubelets (node agents). Taken together, this integration allows Singularity containers to be manipulated directly from Kubernetes.

5.6 Plugins

5.6.1 Overview

A Singularity plugin is a package that can be dynamically loaded by the Singularity runtime, augmenting Singularity with experimental, non-standard and/or vendor-specific functionality. Currently, plugins are able to add commands and flags to Singularity. In the future, plugins will also be able to interface with more complex subsystems of the Singularity runtime.

5.6.2 Using Plugins

The `list` command prints the currently installed plugins.

```
$ singularity plugin list
There are no plugins installed.
```

Plugins are packaged and distributed as binaries encoded with the versatile Singularity Image Format (SIF). However, plugin authors may also distribute the source code of their plugins. A plugin can be compiled from its source code with the `compile` command. A sample plugin `test-plugin` is included with the Singularity source code.

```
$ singularity plugin compile examples/plugins/test-plugin/
```

Upon successful compilation, a SIF file will appear in the directory of the plugin's source code.

```
$ ls examples/plugins/test-plugin/ | grep sif
test-plugin.sif
```

Note: Currently, **all** plugins must be compiled from the Singularity source code tree.

Also, the plugins mechanism for the Go language that Singularity is written in is quite restrictive - it requires extremely close version matching of packages used in a plugin to the ones used in the program the plugin is built for. Additionally Singularity is using build time config to get the source tree location for `singularity plugin compile` so that you don't need to export environment variables etc, and there isn't mismatch between package path information that Go uses. This means that at present you must:

- Build plugins using the exact same version of the source code, in the same location, as was used to build the Singularity executable.
- Use the exact same version of Go that was used to build the executable when compiling a plugin for it.

Every plugin encapsulates various information such as the plugin's author, the plugin's version, etc. To view this information about a plugin, use the `inspect` command.

```
$ singularity plugin inspect examples/plugins/test-plugin/test-plugin.sif
Name: sylabs.io/test-plugin
Description: This is a short test plugin for Singularity
Author: Michael Bauer
Version: 0.0.1
```

To install a plugin, use the `install` command. This operation requires root privilege.

```
$ sudo singularity plugin install examples/plugins/test-plugin/test-plugin.sif
$ singularity plugin list
ENABLED  NAME
    yes  sylabs.io/test-plugin
```

After successful installation, the plugin will automatically be enabled. Any plugin can be disabled with the `disable` command and re-enabled with the `enable` command. Both of these operations require root privilege.

```
$ sudo singularity plugin disable sylabs.io/test-plugin
$ singularity plugin list
ENABLED  NAME
     no  sylabs.io/test-plugin
$ sudo singularity plugin enable sylabs.io/test-plugin
$ singularity plugin list
ENABLED  NAME
    yes  sylabs.io/test-plugin
```

Finally, to uninstall a plugin, use the `uninstall` command. This operation requires root privilege.

```
$ sudo singularity plugin uninstall sylabs.io/test-plugin
Uninstalled plugin "sylabs.io/test-plugin".
$ singularity plugin list
There are no plugins installed.
```

5.6.3 Writing a Plugin

Developers interested in writing Singularity plugins can get started by reading the Go documentation for the plugin package. Furthermore, reading through the source code for the example test plugin will prove valuable. More detailed plugin development documentation is in the works and will be released at a future date.

5.7 Security Options

Singularity 3.0 introduces many new security related options to the container runtime. This document will describe the new methods users have for specifying the security scope and context when running Singularity containers.

5.7.1 Linux Capabilities

Note: It is extremely important to recognize that **granting users Linux capabilities with the `capability` command group is usually identical to granting those users root level access on the host system**. Most if not all capabilities will allow users to "break out" of the container and become root on the host. This feature is targeted toward special use cases (like cloud-native architectures) where an admin/developer might want to limit the attack surface within a container that normally runs as root. This is not a good option in multi-tenant HPC environments where an admin wants to grant a user special privileges within a container. For that and similar use cases, the *fakeroot feature* is a better option.

Singularity provides full support for granting and revoking Linux capabilities on a user or group basis. For example, let us suppose that an admin has decided to grant a user (named `pinger`) capabilities to open raw sockets so that they can use `ping` in a container where the binary is controlled via capabilities. For information about how to manage capabilities as an admin please refer to the capability admin docs.

To take advantage of this granted capability as a user, `pinger` must also request the capability when executing a container with the `--add-caps` flag like so:

```
$ singularity exec --add-caps CAP_NET_RAW library://sylabs/tests/ubuntu_ping:v1.0␣
↪ping -c 1 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=52 time=73.1 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 73.178/73.178/73.178/0.000 ms
```

If the admin decides that it is no longer necessary to allow the user `pinger` to open raw sockets within Singularity containers, they can revoke the appropriate Linux capability and `pinger` will not be able to add that capability to their containers anymore:

```
$ singularity exec --add-caps CAP_NET_RAW library://sylabs/tests/ubuntu_ping:v1.0␣
↪ping -c 1 8.8.8.8
WARNING: not authorized to add capability: CAP_NET_RAW
ping: socket: Operation not permitted
```

Another scenario which is atypical of shared resource environments, but useful in cloud-native architectures is dropping capabilities when spawning containers as the root user to help minimize attack surfaces. With a default installation of Singularity, containers created by the root user will maintain all capabilities. This behavior is configurable if desired. Check out the capability configuration and root default capabilities sections of the admin docs for more information.

Assuming the root user will execute containers with the `CAP_NET_RAW` capability by default, executing the same container `pinger` executed above works without the need to grant capabilities:

```
# singularity exec library://sylabs/tests/ubuntu_ping:v1.0 ping -c 1 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=52 time=59.6 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 59.673/59.673/59.673/0.000 ms
```

Now we can manually drop the `CAP_NET_RAW` capability like so:

```
# singularity exec --drop-caps CAP_NET_RAW library://sylabs/tests/ubuntu_ping:v1.0␣
↪ping -c 1 8.8.8.8
ping: socket: Operation not permitted
```

And now the container will not have the ability to create new sockets, causing the `ping` command to fail.

The `--add-caps` and `--drop-caps` options will accept the `all` keyword. Of course appropriate caution should be exercised when using this keyword.

5.7.2 Building encrypted containers

Beginning in Singularity 3.4.0 it is possible to build and run encrypted containers. The containers are decrypted at runtime entirely in kernel space, meaning that no intermediate decrypted data is ever present on disk. See *encrypted containers* for more details.

5.7.3 Security related action options

Singularity 3.0 introduces many new flags that can be passed to the action commands; `shell`, `exec`, and `run` allowing fine grained control of security.

`--add-caps`

As explained above, `--add-caps` will "activate" Linux capabilities when a container is initiated, providing those capabilities have been granted to the user by an administrator using the `capability add` command. This option will also accept the case insensitive keyword `all` to add every capability granted by the administrator.

`--allow-setuid`

The SetUID bit allows a program to be executed as the user that owns the binary. The most well-known SetUID binaries are owned by root and allow a user to execute a command with elevated privileges. But other SetUID binaries may allow a user to execute a command as a service account.

By default SetUID is disallowed within Singularity containers as a security precaution. But the root user can override this precaution and allow SetUID binaries to behave as expected within a Singularity container with the `--allow-setuid` option like so:

```
$ sudo singularity shell --allow-setuid some_container.sif
```

`--keep-privs`

It is possible for an admin to set a different set of default capabilities or to reduce the default capabilities to zero for the root user by setting the `root default capabilities` parameter in the `singularity.conf` file to `file` or `no` respectively. If this change is in effect, the root user can override the `singularity.conf` file and enter the container with full capabilities using the `--keep-privs` option.

```
$ sudo singularity exec --keep-privs library://centos ping -c 1 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=128 time=18.8 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 18.838/18.838/18.838/0.000 ms
```

--drop-caps

By default, the root user has a full set of capabilities when they enter the container. You may choose to drop specific capabilities when you initiate a container as root to enhance security.

For instance, to drop the ability for the root user to open a raw socket inside the container:

```
$ sudo singularity exec --drop-caps CAP_NET_RAW library://centos ping -c 1 8.8.8.8
ping: socket: Operation not permitted
```

The `drop-caps` option will also accept the case insensitive keyword `all` as an option to drop all capabilities when entering the container.

--security

The `--security` flag allows the root user to leverage security modules such as SELinux, AppArmor, and seccomp within your Singularity container. You can also change the UID and GID of the user within the container at runtime.

For instance:

```
$ sudo whoami
root

$ sudo singularity exec --security uid:1000 my_container.sif whoami
david
```

To use seccomp to blacklist a command follow this procedure. (It is actually preferable from a security standpoint to whitelist commands but this will suffice for a simple example.) Note that this example was run on Ubuntu and that Singularity was installed with the `libseccomp-dev` and `pkg-config` packages as dependencies.

First write a configuration file. An example configuration file is installed with Singularity, normally at `/usr/local/etc/singularity/seccomp-profiles/default.json`. For this example, we will use a much simpler configuration file to blacklist the `mkdir` command.

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "archMap": [
        {
            "architecture": "SCMP_ARCH_X86_64",
            "subArchitectures": [
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
            ]
        }
    ],
    "syscalls": [
        {
            "names": [
                "mkdir"
            ],
            "action": "SCMP_ACT_KILL",
            "args": [],
            "comment": "",
            "includes": {},
            "excludes": {}
        }
    ]
}
```

We'll save the file at `/home/david/no_mkdir.json`. Then we can invoke the container like so:

```
$ sudo singularity shell --security seccomp:/home/david/no_mkdir.json my_container.sif

Singularity> mkdir /tmp/foo
Bad system call (core dumped)
```

Note that attempting to use the blacklisted `mkdir` command resulted in a core dump.

The full list of arguments accepted by the `--security` option are as follows:

```
--security="seccomp:/usr/local/etc/singularity/seccomp-profiles/default.json"
--security="apparmor:/usr/bin/man"
--security="selinux:context"
--security="uid:1000"
--security="gid:1000"
--security="gid:1000:1:0" (multiple gids, first is always the primary group)
```

5.8 Network virtualization

Singularity 3.0 introduces full integration with cni , and several new features to make network virtualization easy.

A few new options have been added to the action commands (`exec`, `run`, and `shell`) to facilitate these features, and the `--net` option has been updated as well. These options can only be used by root.

5.8.1 `--dns`

The `--dns` option allows you to specify a comma separated list of DNS servers to add to the `/etc/resolv.conf` file.

```
$ nslookup sylabs.io | grep Server
Server:             127.0.0.53

$ sudo singularity exec --dns 8.8.8.8 ubuntu.sif nslookup sylabs.io | grep Server
Server:             8.8.8.8

$ sudo singularity exec --dns 8.8.8.8 ubuntu.sif cat /etc/resolv.conf
nameserver 8.8.8.8
```

5.8.2 `--hostname`

The `--hostname` option accepts a string argument to change the hostname within the container.

```
$ hostname
ubuntu-bionic

$ sudo singularity exec --hostname hal-9000 my_container.sif hostname
hal-9000
```

5.8.3 `--net`

Passing the `--net` flag will cause the container to join a new network namespace when it initiates. New in Singularity 3.0, a bridge interface will also be set up by default.

```
$ hostname -I
10.0.2.15

$ sudo singularity exec --net my_container.sif hostname -I
10.22.0.4
```

5.8.4 `--network`

The `--network` option can only be invoked in combination with the `--net` flag. It accepts a comma delimited string of network types. Each entry will bring up a dedicated interface inside container.

```
$ hostname -I
172.16.107.251 10.22.0.1

$ sudo singularity exec --net --network ptp ubuntu.sif hostname -I
10.23.0.6

$ sudo singularity exec --net --network bridge,ptp ubuntu.sif hostname -I
10.22.0.14 10.23.0.7
```

When invoked, the `--network` option searches the singularity configuration directory (commonly `/usr/local/etc/singularity/network/`) for the cni configuration file corresponding to the requested network type(s). Several configuration files are installed with Singularity by default corresponding to the following network types:

- bridge
- ptp
- ipvlan
- macvlan
- none (must be used alone)

`None` is the only network option that can be used by non-privileged users. It isolates the container network from the host network with a loopback interface.

Administrators can also define custom network configurations and place them in the same directory for the benefit of users.

5.8.5 `--network-args`

The `--network-args` option provides a convenient way to specify arguments to pass directly to the cni plugins. It must be used in conjuction with the `--net` flag.

For instance, let's say you want to start an NGINX server on port 80 inside of the container, but you want to map it to port 8080 outside of the container:

```
$ sudo singularity instance start --writable-tmpfs \
    --net --network-args "portmap=8080:80/tcp" docker://nginx web2
```

The above command will start the Docker Hub official NGINX image running in a background instance called `web2`. The NGINX instance will need to be able to write to disk, so we've used the `--writable-tmpfs` argument to allocate some space in memory. The `--net` flag is necessary when using the `--network-args` option, and specifying the `portmap=8080:80/tcp` argument which will map port 80 inside of the container to 8080 on the host.

Now we can start NGINX inside of the container:

```
$ sudo singularity exec instance://web2 nginx
```

And the `curl` command can be used to verify that NGINX is running on the host port 8080 as expected.

```
$ curl localhost:8080
10.22.0.1 - - [16/Oct/2018:09:34:25 -0400] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0"
↪"-"
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

For more information about cni, check the cni specification.

5.9 Limiting container resources with cgroups

Starting in Singularity 3.0, users have the ability to limit container resources using cgroups.

5.9.1 Overview

Singularity cgroups support can be configured and utilized via a TOML file. An example file is typically installed at `/usr/local/etc/singularity/cgroups/cgroups.toml` (but may also be installed in other locations such as `/etc/singularity/cgroups/cgroups.toml` depending on your installation method). You can copy and edit this file to suit your needs. Then when you need to limit your container resources, apply the settings in the TOML file by using the path as an argument to the `--apply-cgroups` option like so:

```
$ sudo singularity shell --apply-cgroups /path/to/cgroups.toml my_container.sif
```

The `--apply-cgroups` option can only be used with root privileges.

5.9.2 Examples

Limiting memory

To limit the amount of memory that your container uses to 500MB (524288000 bytes), follow this example. First, create a `cgroups.toml` file like this and save it in your home directory.

```
[memory]
    limit = 524288000
```

Start your container like so:

```
$ sudo singularity instance start --apply-cgroups /home/$USER/cgroups.toml \
    my_container.sif instance1
```

After that, you can verify that the container is only using 500MB of memory. (This example assumes that `instance1` is the only running instance.)

```
$ cat /sys/fs/cgroup/memory/singularity/*/memory.limit_in_bytes
524288000
```

After you are finished with this example, be sure to cleanup your instance with the following command.

```
$ sudo singularity instance stop instance1
```

Similarly, the remaining examples can be tested by starting instances and examining the contents of the appropriate subdirectories of `/sys/fs/cgroup/`.

Limiting CPU

Limit CPU resources using one of the following strategies. The `cpu` section of the configuration file can limit memory with the following:

shares

This corresponds to a ratio versus other cgroups with cpu shares. Usually the default value is `1024`. That means if you want to allow to use 50% of a single CPU, you will set `512` as value.

```
[cpu]
    shares = 512
```

A cgroup can get more than its share of CPU if there are enough idle CPU cycles available in the system, due to the work conserving nature of the scheduler, so a contained process can consume all CPU cycles even with a ratio of 50%. The ratio is only applied when two or more processes conflicts with their needs of CPU cycles.

quota/period

You can enforce hard limits on the CPU cycles a cgroup can consume, so contained processes can't use more than the amount of CPU time set for the cgroup. `quota` allows you to configure the amount of CPU time that a cgroup can use per period. The default is 100ms (100000us). So if you want to limit amount of CPU time to 20ms during period of 100ms:

```
[cpu]
    period = 100000
    quota = 20000
```

cpus/mems

You can also restrict access to specific CPUs and associated memory nodes by using `cpus/mems` fields:

```
[cpu]
    cpus = "0-1"
    mems = "0-1"
```

Where container has limited access to CPU 0 and CPU 1.

Note: It's important to set identical values for both `cpus` and `mems`.

For more information about limiting CPU with cgroups, see the following external links:

- Red Hat resource management guide section 3.2 CPU
- Red Hat resource management guide section 3.4 CPUSET
- Kernel scheduler documentation

Limiting IO

You can limit and monitor access to I/O for block devices. Use the `[blockIO]` section of the configuration file to do this like so:

```
[blockIO]
    weight = 1000
    leafWeight = 1000
```

`weight` and `leafWeight` accept values between `10` and `1000`.

`weight` is the default weight of the group on all the devices until and unless overridden by a per device rule.

`leafWeight` relates to weight for the purpose of deciding how heavily to weigh tasks in the given cgroup while competing with the cgroup's child cgroups.

To override `weight/leafWeight` for `/dev/loop0` and `/dev/loop1` block devices you would do something like this:

```
[blockIO]
    [[blockIO.weightDevice]]
        major = 7
        minor = 0
        weight = 100
        leafWeight = 50
    [[blockIO.weightDevice]]
        major = 7
        minor = 1
        weight = 100
        leafWeight = 50
```

You could limit the IO read/write rate to 16MB per second for the `/dev/loop0` block device with the following configuration. The rate is specified in bytes per second.

```
[blockIO]
    [[blockIO.throttleReadBpsDevice]]
        major = 7
        minor = 0
        rate = 16777216
    [[blockIO.throttleWriteBpsDevice]]
        major = 7
        minor = 0
        rate = 16777216
```

To limit the IO read/write rate to 1000 IO per second (IOPS) on `/dev/loop0` block device, you can do the following. The rate is specified in IOPS.

```
[blockIO]
    [[blockIO.throttleReadIOPSDevice]]
        major = 7
        minor = 0
        rate = 1000
    [[blockIO.throttleWriteIOPSDevice]]
        major = 7
        minor = 0
        rate = 1000
```

For more information about limiting IO, see the following external links:

- Red Hat resource management guide section 3.1 blkio
- Kernel block IO controller documentation
- Kernel CFQ scheduler documentation

Limiting device access

You can limit read, write, or creation of devices. In this example, a container is configured to only be able to read from or write to `/dev/null`.

```
[[devices]]
    access = "rwm"
    allow = false
[[devices]]
    access = "rw"
    allow = true
    major = 1
    minor = 3
    type = "c"
```

For more information on limiting access to devices the Red Hat resource management guide section 3.5 DEVICES.

5.10 Singularity and MPI applications

The Message Passing Interface (MPI) is a standard extensively used by HPC applications to implement various communication across compute nodes of a single system or across compute platforms. There are two main open-source implementations of MPI at the moment - OpenMPI and MPICH, both of which are supported by Singularity. The goal of this page is to demonstrate the development and running of MPI programs using Singularity containers.

There are several ways of carrying this out, the most popular way of executing MPI applications installed in a Singularity container is to rely on the MPI implementation available on the host. This is called the *Host MPI* or the *Hybrid* model since both the MPI implementations provided by system administrators (on the host) and in the containers will be used.

Another approach is to only use the MPI implementation available on the host and not include any MPI in the container. This is called the *Bind* model since it requires to bind/mount the MPI version available on the host into the container.

Note: The *bind* model requires to mount storage volumes into the container to use the host MPI from the containers. This file system sharing between the host and containers is sometimes not an option on high-performance computing platforms. This restriction on some HPC systems is due to the fact that mounting a storage volume would either require the execution of privileged operations, potentially compromise the access restrictions to other users' data or go against mount options of the parallel/distributed file system where MPI is installed.

5.10.1 Hybrid model

The basic idea behind the *Hybrid Approach* is when you execute a Singularity container with MPI code, you will call `mpiexec` or a similar launcher on the `singularity` command itself. The MPI process outside of the container will then work in tandem with MPI inside the container and the containerized MPI code to instantiate the job.

The Open MPI/Singularity workflow in detail:

1. The MPI launcher (e.g., `mpirun`, `mpiexec`) is called by the resource manager or the user directly from a shell.
2. Open MPI then calls the process management daemon (ORTED).
3. The ORTED process launches the Singularity container requested by the launcher command.
4. Singularity instantiates the container and namespace environment.
5. Singularity then launches the MPI application within the container.
6. The MPI application launches and loads the Open MPI libraries.
7. The Open MPI libraries connect back to the ORTED process via the Process Management Interface (PMI).

At this point the processes within the container run as they would normally directly on the host.

The advantages of this approach are:

- Integration with resource managers such as Slurm.
- Simplicity since similar to natively running MPI applications.

The drawbacks are:

- The MPI in the container must be compatible with the version of MPI available on the host.
- The configuration of the MPI implementation in the container must be configured for optimal use of the hardware if performance is critical.

Since the MPI implementation in the container must be compliant with the version available on the system, a standard approach is to build your own MPI container, including the target MPI implementation.

To illustrate how Singularity can be used to execute MPI applications, we will assume for a moment that the application is *mpitest.c*, a simple Hello World:

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv) {
        int rc;
        int size;
        int myrank;

        rc = MPI_Init (&argc, &argv);
        if (rc != MPI_SUCCESS) {
                fprintf (stderr, "MPI_Init() failed");
                return EXIT_FAILURE;
        }

        rc = MPI_Comm_size (MPI_COMM_WORLD, &size);
        if (rc != MPI_SUCCESS) {
                fprintf (stderr, "MPI_Comm_size() failed");
                goto exit_with_error;
        }
```

(continues on next page)

(continued from previous page)

```
        rc = MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        if (rc != MPI_SUCCESS) {
                fprintf (stderr, "MPI_Comm_rank() failed");
                goto exit_with_error;
        }

        fprintf (stdout, "Hello, I am rank %d/%d", myrank, size);

        MPI_Finalize();

        return EXIT_SUCCESS;

 exit_with_error:
        MPI_Finalize();
        return EXIT_FAILURE;
}
```

Note: MPI is an interface to a library, so it consists of function calls and libraries that can be used by many programming languages. It comes with standardized bindings for Fortran and C. However, it can support applications in many languages like Python, R, etc.

The next step is to build the definition file which will depend on the MPI implementation available on the host.

If the host MPI is MPICH, a definition file such as the following example can be used:

```
Bootstrap: docker
From: ubuntu:latest

%files
    mpitest.c /opt

%environment
    export MPICH_DIR=/opt/mpich-3.3
    export SINGULARITY_MPICH_DIR=$MPICH_DIR
    export SINGULARITYENV_APPEND_PATH=$MPICH_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=$MPICH_DIR/lib

%post
    echo "Installing required packages..."
    apt-get update && apt-get install -y wget git bash gcc gfortran g++ make

    # Information about the version of MPICH to use
    export MPICH_VERSION=3.3
    export MPICH_URL="http://www.mpich.org/static/downloads/$MPICH_VERSION/mpich-
↪$MPICH_VERSION.tar.gz"
    export MPICH_DIR=/opt/mpich

    echo "Installing MPICH..."
    mkdir -p /tmp/mpich
    mkdir -p /opt
    # Download
    cd /tmp/mpich && wget -O mpich-$MPICH_VERSION.tar.gz $MPICH_URL && tar xzf mpich-
↪$MPICH_VERSION.tar.gz
    # Compile and install
```

(continues on next page)

(continued from previous page)

```
    cd /tmp/mpich/mpich-$MPICH_VERSION && ./configure --prefix=$MPICH_DIR && make␣
↪install
    # Set env variables so we can compile our application
    export PATH=$MPICH_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$MPICH_DIR/lib:$LD_LIBRARY_PATH
    export MANPATH=$MPICH_DIR/share/man:$MANPATH

    echo "Compiling the MPI application..."
    cd /opt && mpicc -o mpitest mpitest.c
```

If the host MPI is Open MPI, the definition file looks like:

```
Bootstrap: docker
From: ubuntu:latest

%files
    mpitest.c /opt

%environment
    export OMPI_DIR=/opt/ompi
    export SINGULARITY_OMPI_DIR=$OMPI_DIR
    export SINGULARITYENV_APPEND_PATH=$OMPI_DIR/bin
    export SINGULARITYENV_APPEND_LD_LIBRARY_PATH=$OMPI_DIR/lib

%post
    echo "Installing required packages..."
    apt-get update && apt-get install -y wget git bash gcc gfortran g++ make file

    echo "Installing Open MPI"
    export OMPI_DIR=/opt/ompi
    export OMPI_VERSION=4.0.1
    export OMPI_URL="https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-
↪$OMPI_VERSION.tar.bz2"
    mkdir -p /tmp/ompi
    mkdir -p /opt
    # Download
    cd /tmp/ompi && wget -O openmpi-$OMPI_VERSION.tar.bz2 $OMPI_URL && tar -xjf␣
↪openmpi-$OMPI_VERSION.tar.bz2
    # Compile and install
    cd /tmp/ompi/openmpi-$OMPI_VERSION && ./configure --prefix=$OMPI_DIR && make␣
↪install
    # Set env variables so we can compile our application
    export PATH=$OMPI_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$OMPI_DIR/lib:$LD_LIBRARY_PATH
    export MANPATH=$OMPI_DIR/share/man:$MANPATH

    echo "Compiling the MPI application..."
    cd /opt && mpicc -o mpitest mpitest.c
```

5.10.2 Bind model

Similarily to the *Hybrid Approach*, the basic idea behind *Bind Approach* is to start the MPI application by calling the MPI launcher (e.g., *mpirun*) from the host. The main difference between the hybrid and bind approach is the fact that with the bind approach, the container usually does not include any MPI implementation. This means that Singularity needs to mount/bind the MPI available on the host into the container.

Technically this requires two steps:

1. Know where the MPI implementation on the host is installed.
2. Mount/bind it into the container in a location where the system will be able to find libraries and binaries.

The advantages of this approach are:

- Integration with resource managers such as Slurm.
- Container images are smaller since there is no need to add an MPI in the containers.

The drawbacks are:

- The MPI used to compile the application in the container must be compatible with the version of MPI available on the host.
- The user must know where the host MPI is installed.
- The user must ensure that binding the directory where the host MPI is installed is possible.
- The user must ensure that the host MPI is compatible with the MPI used to compile and install the application in the container.

The creation of a Singularity container based on the bind model is based on the following steps:

1. Compile your application on a system with the target MPI implementation, as you would do to install your application on any system.
2. Create a definition file that includes the copy of the application from the host to the container image, as well as all required dependencies.
3. Generate the container image.

As already mentioned, the compilation of the application on the host is not different from the installation of your application on any system. Just make sure that the MPI on the system where you create your container is compatible with the MPI available on the platform(s) where you want to run your containers. For example, a container where the application has been compiled with MPICH will not be able to run on a system where only Open MPI is available, even if you mount the directory where Open MPI is installed.

A definition file for a container in bind mode is fairly straight forward. The following example shows the definition file for NetPIPE-5.1.4 compiled on the host in `/tmp/NetPIPE-5.1.4`:

```
Bootstrap: docker
From: ubuntu:disco

%files
      /tmp/NetPIPE-5.1.4/NPmpi /opt

%environment
      MPI_DIR=/opt/mpi
      export MPI_DIR
      export SINGULARITY_MPI_DIR=$MPI_DIR
      export SINGULARITYENV_APPEND_PATH=$MPI_DIR/bin
      export SINGULARITYENV_APPEND_LD_LIBRARY_PATH=$MPI_DIR/lib
```

(continues on next page)

(continued from previous page)

```
%post
      apt-get update && apt-get install -y wget git bash gcc gfortran g++ make file
      mkdir -p /opt/mpi
      apt-get clean
```

In this example, the application, NetPIPE-5.1.4, is copied into `/opt`, as a result, the path to the executable to use on the `mpirun` command is `/opt/NPmpi`. Also, this definition file prepares the environment to have the host MPI mounted in `/opt/mpi`; it sets all the required environment variables (PATH and LD_LIBRARY_PATH) for the system to find all MPI binaries and libraries at run-time.

5.10.3 Execution

The standard way to execute MPI applications with hybrid Singularity containers is to run the native `mpirun` command from the host, which will start Singularity containers and ultimately MPI ranks within the containers.

Assuming your container with MPI and your application is already build, the `mpirun` command to start your application looks like when your container has been built based on the hybrid model:

```
$ mpirun -n <NUMBER_OF_RANKS> singularity exec <PATH/TO/MY/IMAGE> </PATH/TO/BINARY/
↪WITHIN/CONTAINER>
```

Practically, this command will first start a process instantiating `mpirun` and then Singularity containers on compute nodes. Finally, when the containers start, the MPI binary is executed.

For containers built based on the bind model, the command simply needs to include the appropriate bind option:

```
$ mpirun -n <NUMBER_OF_RANKS> singularity exec --bind <PATH/TO/HOST/MPI/DIRECTORY>:
↪<PATH/IN/CONTAINER> <PATH/TO/MY/IMAGE> </PATH/TO/BINARY/WITHIN/CONTAINER>
```

Based on the example presented in the previous sub-section, and assuming MPI is installed in `/opt/openmpi` on the host, the command will look like:

```
$ mpirun -n <NUMBER_OF_RANKS> singularity exec --bind /opt/openmpi:/opt/mpi <PATH/TO/
↪MY/IMAGE> /opt/NPmpi
```

If your target system is setup with a batch system such as SLURM, a standard way to execute MPI applications is through a batch script. The following example illustrates the context of a batch script for Slurm that aims at starting a Singularity container on each node allocated to the execution of the job. It can easily be adapted for all major batch systems available.

```
$ cat my_job.sh
#!/bin/bash
#SBATCH --job-name singularity-mpi
#SBATCH -N $NNODES # total number of nodes
#SBATCH --time=00:05:00 # Max execution time

mpirun -n $NP singularity exec /var/nfsshare/gvallee/mpich.sif /opt/mpitest
```

In fact, the example describes a job that requests the number of nodes specified by the `NNODES` environment variable and a total number of MPI processes specified by the `NP` environment variable. The example is also assuming that the container is based on the hybrid model; if it is based on the bind model, please add the appropriate bind options.

A user can then submit a job by executing the following SLURM command:

```
$ sbatch my_job.sh
```

5.11 GPU Support (NVIDIA CUDA & AMD ROCm)

Singularity natively supports running application containers that use NVIDIA's CUDA GPU compute framework, or AMD's ROCm solution. This allows easy access to users of GPU-enabled machine learning frameworks such as tensorflow, regardless of the host operating system. As long as the host has a driver and library installation for CUDA/ROCm then it's possible to e.g. run tensorflow in an up-to-date Ubuntu 18.04 container, from an older RHEL 6 host.

Applications that support OpenCL for compute acceleration can also be used easily, with an additional bind option.

5.11.1 NVIDIA GPUs & CUDA

Commands that `run`, or otherwise execute containers (`shell`, `exec`) can take an `--nv` option, which will setup the container's environment to use an NVIDIA GPU and the basic CUDA libraries to run a CUDA enabled application. The `--nv` flag will:

- Ensure that the `/dev/nvidiaX` device entries are available inside the container, so that the GPU cards in the host are accessible.
- Locate and bind the basic CUDA libraries from the host into the container, so that they are available to the container, and match the kernel GPU driver on the host.
- Set the `LD_LIBRARY_PATH` inside the container so that the bound-in version of the CUDA libraries are used by applications run inside the container.

Requirements

To use the `--nv` flag to run a CUDA application inside a container you must ensure that:

- The host has a working installation of the NVIDIA GPU driver, and a matching version of the basic NVIDIA/CUDA libraries. The host *does not* need to have an X server running, unless you want to run graphical apps from the container.
- Either a working installation of the `nvidia-container-cli` tool is available on the `PATH` when you run `singularity`, or the NVIDIA libraries are in the system's library search path.
- The application inside your container was compiled for a CUDA version, and device capability level, that is supported by the host card and driver.

These requirements are usually satisified by installing the NVIDIA drivers and CUDA packages directly from the NVIDIA website. Linux distributions may provide NVIDIA drivers and CUDA libraries, but they are often outdated which can lead to problems running applications compiled for the latest versions of CUDA.

Library Search Options

Singularity will find the NVIDIA/CUDA libraries on your host either using the `nvidia-container-cli` tool, or, if it is not available, a list of libraries in the configuration file `etc/singularity/nvbliblist`.

If possible we recommend installing the `nvidia-container-cli` tool from the NVIDIA libnvidia-container website

The fall-back `etc/singularity/nvbliblist` library list is correct at time of release for CUDA 10.1. However, if future CUDA versions split or add library files you may need to edit it. The `nvidia-container-cli` tool will be updated by NVIDIA to always return the appropriate list of libraries.

Example - tensorflow-gpu

Tensorflow is commonly used for machine learning projects but can be diffficult to install on older systems, and is updated frequently. Running tensorflow from a container removes installation problems and makes trying out new versions easy.

The official tensorflow repository on Docker Hub contains NVIDA GPU supporting containers, that will use CUDA for processing. You can view the available versions on the tags page on Docker Hub

The container is large, so it's best to build or pull the docker image to a SIF before you start working with it:

```
$ singularity pull docker://tensorflow/tensorflow:latest-gpu
...
INFO:    Creating SIF file...
INFO:    Build complete: tensorflow_latest-gpu.sif
```

Then run the container with GPU support:

```
$ singularity run --nv tensorflow_latest-gpu.sif

________                               _______________
___  __/__________________________________  ____/__  /________      __
__  /  _  _ \_  __ \_  ___/  __ \_  ___/_  /_   __  /_  __ \_ | /| / /
_  /   /  __/  / / /(__  )/ /_/ /  /   _  __/   _  / / /_/ /_ |/ |/ /
/_/    \___//_/ /_//____/ \____//_/    /_/      /_/  \____/____/|__/


You are running this container as user with ID 1000 and group 1000,
which should map to the ID and group for your user on the Docker host. Great!

Singularity>
```

You can verify the GPU is available within the container by using the tensorflow `list_local_devices()` function:

```
Singularity> python
Python 2.7.15+ (default, Jul  9 2019, 16:51:35)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from tensorflow.python.client import device_lib
>>> print(device_lib.list_local_devices())
2019-11-14 15:32:09.743600: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your␣
↪CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2␣
↪FMA
2019-11-14 15:32:09.784482: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94]␣
↪CPU Frequency: 3292620000 Hz
2019-11-14 15:32:09.787911: I tensorflow/compiler/xla/service/service.cc:168] XLA␣
↪service 0x565246634360 executing computations on platform Host. Devices:
2019-11-14 15:32:09.787939: I tensorflow/compiler/xla/service/service.cc:175]   ␣
↪StreamExecutor device (0): Host, Default Version
2019-11-14 15:32:09.798428: I tensorflow/stream_executor/platform/default/dso_loader.
↪cc:44] Successfully opened dynamic library libcuda.so.1
2019-11-14 15:32:09.842683: I tensorflow/stream_executor/cuda/cuda_gpu_executor.
↪cc:1006] successful NUMA node read from SysFS had negative value (-1), but there␣
↪must be at least one NUMA node, so returning NUMA node zero
2019-11-14 15:32:09.843252: I tensorflow/compiler/xla/service/service.cc:168] XLA␣
↪service 0x5652469263d0 executing computations on platform CUDA. Devices:
2019-11-14 15:32:09.843265: I tensorflow/compiler/xla/service/service.cc:175]   ␣
↪StreamExecutor device (0): GeForce GT 730, Compute Capability 3.5
```

(continues on next page)

(continued from previous page)

```
2019-11-14 15:32:09.843380: I tensorflow/stream_executor/cuda/cuda_gpu_executor.
↪cc:1006] successful NUMA node read from SysFS had negative value (-1), but there␣
↪must be at least one NUMA node, so returning NUMA node zero
2019-11-14 15:32:09.843984: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1618]␣
↪Found device 0 with properties:
name: GeForce GT 730 major: 3 minor: 5 memoryClockRate(GHz): 0.9015
...
```

Multiple GPUs

By default, Singularity makes all host devices available in the container. When the `--contain` option is used a minimal `/dev` tree is created in the container, but the `--nv` option will ensure that all nvidia devices on the host are present in the container.

This behaviour is different to `nvidia-docker` where an `NVIDIA_VISIBLE_DEVICES` environment variable is used to control whether some or all host GPUs are visible inside a container. The `nvidia-container-runtime` explicitly binds the devices into the container dependent on the value of `NVIDIA_VISIBLE_DEVICES`.

To control which GPUs are used in a Singularity container that is run with `--nv` you can set `SINGULARITYENV_CUDA_VISIBLE_DEVICES` before running the container, or `CUDA_VISIBLE_DEVICES` inside the container. This variable will limit the GPU devices that CUDA programs see.

E.g. to run the tensorflow container, but using only the first GPU in the host, we could do:

```
$ SINGULARITYENV_CUDA_VISIBLE_DEVICES=0 singularity run --nv tensorflow_latest-gpu.sif

# or

$ export SINGULARITYENV_CUDA_VISIBLE_DEVICES=0
$ singularity run tensorflow_latest-gpu.sif
```

Troubleshooting

If the host installation of the NVIDIA / CUDA driver and libraries is working and up-to-date there are rarely issues running CUDA programs inside of Singularity containers. The most common issue seen is:

CUDA_ERROR_UNKNOWN when everything seems to be correctly configured

CUDA depends on multiple kernel modules being loaded. Not all of the modules are loaded at system startup. Some portions of the NVIDA driver stack are initialized when first needed. This is done using a setuid root binary, so initializing can be triggered by any user on the host. In Singularity containers, privilege escalation is blocked, so the setuid root binary cannot initialize the driver stack fully.

If you experience `CUDA_ERROR_UNKNOWN` in a container, initialize the driver stack on the host first, by running a CUDA program there or `modprobe nvidia_uvm` as root, and using `nvidia-persistenced` to avoid driver unload.

5.11.2 AMD GPUs & ROCm

Singularity 3.5 adds a `--rocm` flag to support GPU compute with the ROCm framework using AMD Radeon GPU cards.

Commands that `run`, or otherwise execute containers (`shell`, `exec`) can take an `--rocm` option, which will setup the container's environment to use a Radeon GPU and the basic ROCm libraries to run a ROCm enabled application. The `--rocm` flag will:

- Ensure that the `/dev/dri/` device entries are available inside the container, so that the GPU cards in the host are accessible.
- Locate and bind the basic ROCm libraries from the host into the container, so that they are available to the container, and match the kernel GPU driver on the host.
- Set the `LD_LIBRARY_PATH` inside the container so that the bound-in version of the ROCm libraries are used by application run inside the container.

Requirements

To use the `--rocm` flag to run a CUDA application inside a container you must ensure that:

- The host has a working installation of the `amdgpu` driver, and a compatible version of the basic ROCm libraries. The host *does not* need to have an X server running, unless you want to run graphical apps from the container.
- The ROCm libraries are in the system's library search path.
- The application inside your container was compiled for a ROCm version that is compatible with the ROCm version on your host.

These requirements can be satisfied by following the requirements on the ROCm web site

At time of release, Singularity was tested successfully on Debian 10 with ROCm 2.8/2.9 and the upstream kernel driver, and Ubuntu 18.04 with ROCm 2.9 and the DKMS driver.

Example - tensorflow-rocm

Tensorflow is commonly used for machine learning projects, but can be difficult to install on older systems, and is updated frequently. Running tensorflow from a container removes installation problems and makes trying out new versions easy.

The rocm tensorflow repository on Docker Hub contains Radeon GPU supporting containers, that will use ROCm for processing. You can view the available versions on the tags page on Docker Hub

The container is large, so it's best to build or pull the docker image to a SIF before you start working with it:

```
$ singularity pull docker://rocm/tensorflow:latest
...
INFO:    Creating SIF file...
INFO:    Build complete: tensorflow_latest.sif
```

Then run the container with GPU support:

```
$ singularity run --rocm tensorflow_latest.sif
```

You can verify the GPU is available within the container by using the tensorflow `list_local_devices()` function:

```
Singularity> ipython
Python 3.5.2 (default, Jul 10 2019, 11:58:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
>>> from tensorflow.python.client import device_lib
...
>>> print(device_lib.list_local_devices())
...
2019-11-14 16:33:42.750509: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1651]␣
↪Found device 0 with properties:
name: Lexa PRO [Radeon RX 550/550X]
AMDGPU ISA: gfx803
memoryClockRate (GHz) 1.183
pciBusID 0000:09:00.0
...
```

5.11.3 OpenCL Applications

Both the `--rocm` and `--nv` flags will bind the vendor OpenCL implementation libraries into a container that is being run. However, these libraries will not be used by OpenCL applications unless a vendor icd file is available under `/etc/OpenCL/vendors` that directs OpenCL to use the vendor library.

The simplest way to use OpenCL in a container is to `--bind /etc/OpenCL` so that the icd files from the host (which match the bound-in libraries) are present in the container.

Example - Blender OpenCL

The Sylabs examples repository contains an example container definition for the 3D modelling application 'Blender'.

The latest versions of Blender supports OpenCL rendering. You can run Blender as a graphical application that will make use of a local Radeon GPU for OpenCL compute using the container that has been published to the Sylabs library:

```
$ singularity exec --rocm --bind /etc/OpenCL library://sylabs/examples/blender blender
```

Note the *exec* used as the *runscript* for this container is setup for batch rendering (which can also use OpenCL).

CHAPTER

SIX

GET INVOLVED

6.1 Contributing

Singularity is an open source project, meaning we have the challenge of limited resources. We are grateful for any support that you can offer. Helping other users, raising issues, helping write documentation, or contributing code are all ways to help!

6.1.1 Join the community

This is a huge endeavor, and your help would be greatly appreciated! Post to online communities about Singularity, and request that your distribution vendor, service provider, and system administrators include Singularity for you!

Singularity Google Group

If you have been using Singularity and having good luck with it, join our Google Group and help out other users!

Singularity on Slack

Many of our users come to Slack for quick help with an issue. You can find us at singularity-container.

6.1.2 Raise an Issue

For general bugs/issues, you can open an issue at the GitHub repo. However, if you find a security related issue/problem, please email Sylabs directly at security@sylabs.io. More information about the Sylabs security policies and procedures can be found here

6.1.3 Write Documentation

We (like almost all open source software providers) have a documentation dilemma… We tend to focus on the code features and functionality before working on documentation. And there is very good reason for this: we want to share the love so nobody feels left out!

You can contribute to the documentation by raising an issue to suggest an improvement or by sending a pull request on our repository for documentation.

The current documentation is generated with:

- reStructured Text (RST) and ReadTheDocs.

Other dependencies include:

- Python 2.7.
- Sphinx.

More information about contributing to the documentation, instructions on how to install the dependencies, and how to generate the files can be obtained here.

For more information on using Git and GitHub to create a pull request suggesting additions and edits to the docs, see the *section on contributing to the code*. The procedure is identical for contributions to the documentation and the code base.

6.1.4 Contribute to the code

We use the traditional GitHub Flow to develop. This means that you fork the main repo, create a new branch to make changes, and submit a pull request (PR) to the master branch.

Check out our official CONTRIBUTING.md document, which also includes a code of conduct.

Step 1. Fork the repo

To contribute to Singularity, you should obtain a GitHub account and fork the Singularity repository. Once forked, clone your fork of the repo to your computer. (Obviously, you should replace `your-username` with your GitHub username.)

```
$ git clone https://github.com/your-username/singularity.git && \
    cd singularity/
```

Step 2. Checkout a new branch

Branches are a way of isolating your features from the main branch. Given that we've just cloned the repo, we will probably want to make a new branch from master in which to work on our new feature. Lets call that branch `new-feature`:

```
$ git checkout master && \
    git checkout -b new-feature
```

Note: You can always check which branch you are in by running `git branch`.

Step 3. Make your changes

On your new branch, go nuts! Make changes, test them, and when you are happy commit the changes to the branch:

```
$ git add file-changed1 file-changed2...

$ git commit -m "what changed?"
```

This commit message is important - it should describe exactly the changes that you have made. Good commit messages read like so:

```
$ git commit -m "changed function getConfig in functions.go to output csv to fix #2"

$ git commit -m "updated docs about shell to close #10"
```

The tags `close #10` and `fix #2` are referencing issues that are posted on the upstream repo where you will direct your pull request. When your PR is merged into the master branch, these messages will automatically close the issues, and further, they will link your commits directly to the issues they intend to fix. This will help future maintainers understand your contribution, or (hopefully not) revert the code back to a previous version if necessary.

Step 4. Push your branch to your fork

When you are done with your commits, you should push your branch to your fork (and you can also continuously push commits here as you work):

```
$ git push origin new-feature
```

Note that you should always check the status of your branches to see what has been pushed (or not):

```
$ git status
```

Step 5. Submit a Pull Request

Once you have pushed your branch, then you can go to your fork (in the web GUI on GitHub) and submit a Pull Request. Regardless of the name of your branch, your PR should be submitted to the Sylabs `master` branch. Submitting your PR will open a conversation thread for the maintainers of Singularity to discuss your contribution. At this time, the continuous integration that is linked with the code base will also be executed. If there is an issue, or if the maintainers suggest changes, you can continue to push commits to your branch and they will update the Pull Request.

Step 6. Keep your branch in sync

Cloning the repo will create an exact copy of the Singularity repository at that moment. As you work, your branch may become out of date as others merge changes into the upstream master. In the event that you need to update a branch, you will need to follow the next steps:

```
$ git remote add upstream https://github.com/sylabs/singularity.git && # to add a new␣
↪remote named "upstream" \
    git checkout master && # or another branch to be updated \
    git pull upstream master && \
    git push origin master && # to update your fork \
    git checkout new-feature && \
    git merge master
```

CHAPTER SEVEN

REFERENCE

7.1 Appendix

7.1.1 Singularity's environment variables

Singularity 3.0 comes with some environment variables you can set or modify depending on your needs. You can see them listed alphabetically below with their respective functionality.

A

1. **SINGULARITY_ADD_CAPS**: To specify a list (comma separated string) of capabilities to be added. Default is an empty string.
2. **SINGULARITY_ALL**: List all the users and groups capabilities.
3. **SINGULARITY_ALLOW_SETUID**: To specify that setuid binaries should or not be allowed in the container. (root only) Default is set to false.
4. **SINGULARITY_APP** and **SINGULARITY_APPNAME**: Sets the name of an application to be run inside a container.
5. **SINGULARITY_APPLY_CGROUPS**: Used to apply cgroups from an input file for container processes. (it requires root privileges)

B

1. **SINGULARITY_BINDPATH** and **SINGULARITY_BIND**: Comma separated string `source:<dest>` list of paths to bind between the host and the container.
2. **SINGULARITY_BOOT**: Set to false by default, considers if executing `/sbin/init` when container boots (root only).
3. **SINGULARITY_BUILDER**: To specify the remote builder service URL. Defaults to our remote builder.

C

1. **SINGULARITY_CACHEDIR**: Specifies the directory for image downloads to be cached in. See *Cache Folders*.
2. **SINGULARITY_CLEANENV**: Specifies if the environment should be cleaned or not before running the container. Default is set to false.
3. **SINGULARITY_CONTAIN**: To use minimal `/dev` and empty other directories (e.g. `/tmp` and `$HOME`) instead of sharing filesystems from your host. Default is set to false.
4. **SINGULARITY_CONTAINALL**: To contain not only file systems, but also PID, IPC, and environment. Default is set to false.
5. **SINGULARITY_CONTAINLIBS**: Used to specify a string of file names (comma separated string) to bind to the `/.singularity.d/libs` directory.

D

1. **SINGULARITY_DEFFILE**: Shows the Singularity recipe that was used to generate the image.
2. **SINGULARITY_DESC**: Contains a description of the capabilities.
3. **SINGULARITY_DETACHED**: To submit a build job and print the build ID (no real-time logs and also requires `--remote`). Default is set to false.
4. **SINGULARITY_DISABLE_CACHE**: To disable all caching of docker/oci, library, oras, etc. downloads and built SIFs. Default is set to false.
5. **SINGULARITY_DNS**: A list of the DNS server addresses separated by commas to be added in `resolv.conf`.
6. **SINGULARITY_DOCKER_LOGIN**: To specify the interactive prompt for docker authentication.
7. **SINGULARITY_DOCKER_USERNAME**: To specify a username for docker authentication.
8. **SINGULARITY_DOCKER_PASSWORD**: To specify the password for docker authentication.
9. **SINGULARITY_DROP_CAPS**: To specify a list (comma separated string) of capabilities to be dropped. Default is an empty string.

E

1. **SINGULARITY_ENVIRONMENT**: Contains all the environment variables that have been exported in your container.
2. **SINGULARITY_ENCRYPTION_PASSPHRASE**: Used to specify the plaintext passphrase to encrypt the container.
3. **SINGULARITY_ENCRYPTION_PEM_PATH**: Used to specify the path of the file containing public or private key to encrypt the container in PEM format.
4. **SINGULARITYENV_***: Allows you to transpose variables into the container at runtime. You can see more in detail how to use this variable in our *environment and metadata section*.
5. **SINGULARITYENV_APPEND_PATH**: Used to append directories to the end of the `$PATH` environment variable. You can see more in detail on how to use this variable in our *environment and metadata section*.
6. **SINGULARITYENV_PATH**: A specified path to override the `$PATH` environment variable within the container. You can see more in detail on how to use this variable in our *environment and metadata section*.

7. **SINGULARITYENV_PREPEND_PATH**: Used to prepend directories to the beginning of *$PATH`* environment variable. You can see more in detail on how to use this variable in our *environment and metadata section*.

F

1. **SINGULARITY_FAKEROOT**: Set to false by default, considers running the container in a new user namespace as uid 0 (experimental).
2. **SINGULARITY_FORCE**: Forces to kill the instance.

G

1. **SINGULARITY_GROUP**: Used to specify a string of capabilities for the given group.

H

1. **SINGULARITY_HELPFILE**: Specifies the runscript helpfile, if it exists.
2. **SINGULARITY_HOME** : A home directory specification, it could be a source or destination path. The source path is the home directory outside the container and the destination overrides the home directory within the container.
3. **SINGULARITY_HOSTNAME**: The container's hostname.

I

1. **SINGULARITY_IMAGE**: Filename of the container.

J

1. **SINGULARITY_JSON**: Specifies the structured json of the def file, every node as each section in the def file.

K

1. **SINGULARITY_KEEP_PRIVS**: To let root user keep privileges in the container. Default is set to false.

L

1. **SINGULARITY_LABELS**: Specifies the labels associated with the image.
2. **SINGULARITY_LIBRARY**: Specifies the library to pull from. Default is set to our Cloud Library.

N

1. **SINGULARITY_NAME**: Specifies a custom image name.
2. **SINGULARITY_NETWORK**: Used to specify a desired network. If more than one parameters is used, addresses should be separated by commas, where each network will bring up a dedicated interface inside the container.
3. **SINGULARITY_NETWORK_ARGS**: To specify the network arguments to pass to CNI plugins.
4. **SINGULARITY_NOCLEANUP**: To not clean up the bundle after a failed build, this can be helpful for debugging. Default is set to false.
5. **SINGULARITY_NOHTTPS**: Sets to either false or true to avoid using HTTPS for communicating with the local docker registry. Default is set to false.
6. **SINGULARITY_NO_HOME**: Considers not mounting users home directory if home is not the current working directory. Default is set to false.
7. **SINGULARITY_NO_INIT** and **SINGULARITY_NOSHIMINIT**: Considers not starting the `shim` process with `--pid`.
8. **SINGULARITY_NO_NV**: Flag to disable Nvidia support. Opposite of `SINGULARITY_NV`.
9. **SINGULARITY_NO_PRIVS**: To drop all the privileges from root user in the container. Default is set to false.
10. **SINGULARITY_NV**: To enable experimental Nvidia support. Default is set to false.

O

1. **SINGULARITY_OVERLAY** and **SINGULARITY_OVERLAYIMAGE**: To indicate the use of an overlay file system image for persistent data storage or as read-only layer of container.

P

1. **SINGULARITY_PWD** and **SINGULARITY_TARGET_PWD**: The initial working directory for payload process inside the container.

R

1. **SINGULARITY_REMOTE**: To build an image remotely. (Does not require root) Default is set to false.
2. **SINGULARITY_ROOTFS**: To reference the system file location.
3. **SINGULARITY_RUNSCRIPT**: Specifies the runscript of the image.

S

1. **SINGULARITY_SANDBOX**: To specify that the format of the image should be a sandbox. Default is set to false.
2. **SINGULARITY_SCRATCH** and **SINGULARITY_SCRATCHDIR**: Used to include a scratch directory within the container that is linked to a temporary directory. (use -W to force location)
3. **SINGULARITY_SECTION**: To specify a comma separated string of all the sections to be run from the deffile (setup, post, files, environment, test, labels, none)
4. **SINGULARITY_SECURITY**: Used to enable security features. (SELinux, Apparmor, Seccomp)

5. **SINGULARITY_SECRET**: Lists all the private keys instead of the default which display the public ones.
6. **SINGULARITY_SHELL**: The path to the program to be used as an interactive shell.
7. **SINGULARITY_SIGNAL**: Specifies a signal sent to the instance.

T

1. **SINGULARITY_TEST**: Specifies the test script for the image.
2. **SINGULARITY_TMPDIR**: Used with the `build` command, to consider a temporary location for the build. See *Temporary Folders*.

U

1. **SINGULARITY_UNSHARE_PID**: To specify that the container will run in a new PID namespace. Default is set to false.
2. **SINGULARITY_UNSHARE_IPC**: To specify that the container will run in a new IPC namespace. Default is set to false.
3. **SINGULARITY_UNSHARE_NET**: To specify that the container will run in a new network namespace (sets up a bridge network interface by default). Default is set to false.
4. **SINGULARITY_UNSHARE_UTS**: To specify that the container will run in a new UTS namespace. Default is set to false.
5. **SINGULARITY_UPDATE**: To run the definition over an existing container (skips the header). Default is set to false.
6. **SINGULARITY_URL**: Specifies the key server `URL`.
7. **SINGULARITY_USER**: Used to specify a string of capabilities for the given user.
8. **SINGULARITY_USERNS** and **SINGULARITY_UNSHARE_USERNS**: To specify that the container will run in a new user namespace, allowing Singularity to run completely unprivileged on recent kernels. This may not support every feature of Singularity. (Sandbox image only). Default is set to false.

W

1. **SINGULARITY_WORKDIR**: The working directory to be used for `/tmp`, `/var/tmp` and `$HOME` (if `-c` or `--contain` was also used)
2. **SINGULARITY_WRITABLE**: By default, all Singularity containers are available as read only, this option makes the file system accessible as read/write. Default set to false.
3. **SINGULARITY_WRITABLE_TMPFS**: Makes the file system accessible as read-write with non-persistent data (with overlay support only). Default is set to false.

7.1.2 Build Modules

`library` bootstrap agent

Overview

You can use an existing container on the Container Library as your "base," and then add customization. This allows you to build multiple images from the same starting point. For example, you may want to build several containers with the same custom python installation, the same custom compiler toolchain, or the same base MPI installation. Instead of building these from scratch each time, you could create a base container on the Container Library and then build new containers from that existing base container adding customizations in `%post`, `%environment`, `%runscript`, etc.

Keywords

```
Bootstrap: library
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
From: <entity>/<collection>/<container>:<tag>
```

The `From` keyword is mandatory. It specifies the container to use as a base. `entity` is optional and defaults to `library`. `collection` is optional and defaults to `default`. This is the correct namespace to use for some official containers (`alpine` for example). `tag` is also optional and will default to `latest`.

```
Library: http://custom/library
```

The Library keyword is optional. It will default to `https://library.sylabs.io`.

`docker` bootstrap agent

Overview

Docker images are comprised of layers that are assembled at runtime to create an image. You can use Docker layers to create a base image, and then add your own custom software. For example, you might use Docker's Ubuntu image layers to create an Ubuntu Singularity container. You could do the same with CentOS, Debian, Arch, Suse, Alpine, BusyBox, etc.

Or maybe you want a container that already has software installed. For instance, maybe you want to build a container that uses CUDA and cuDNN to leverage the GPU, but you don't want to install from scratch. You can start with one of the `nvidia/cuda` containers and install your software on top of that.

Or perhaps you have already invested in Docker and created your own Docker containers. If so, you can seamlessly convert them to Singularity with the `docker` bootstrap module.

Keywords

```
Bootstrap: docker
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
From: <registry>/<namespace>/<container>:<tag>@<digest>
```

The `From` keyword is mandatory. It specifies the container to use as a base. `registry` is optional and defaults to `index.docker.io`. `namespace` is optional and defaults to `library`. This is the correct namespace to use for some official containers (ubuntu for example). `tag` is also optional and will default to `latest`

See *Singularity and Docker* for more detailed info on using Docker registries.

```
Registry: http://custom_registry
```

The Registry keyword is optional. It will default to `index.docker.io`.

```
Namespace: namespace
```

The Namespace keyword is optional. It will default to `library`.

```
IncludeCmd: yes
```

The IncludeCmd keyword is optional. If included, and if a `%runscript` is not specified, a Docker `CMD` will take precedence over `ENTRYPOINT` and will be used as a runscript. Note that the `IncludeCmd` keyword is considered valid if it is not empty! This means that `IncludeCmd: yes` and `IncludeCmd: no` are identical. In both cases the `IncludeCmd` keyword is not empty, so the Docker `CMD` will take precedence over an `ENTRYPOINT`.

See *Singularity and Docker* for more info on order of operations for determining a runscript.

Notes

Docker containers are stored as a collection of tarballs called layers. When building from a Docker container the layers must be downloaded and then assembled in the proper order to produce a viable file system. Then the file system must be converted to Singularity Image File (sif) format.

Building from Docker Hub is not considered reproducible because if any of the layers of the image are changed, the container will change. If reproducibility is important to your workflow, consider hosting a base container on the Container Library and building from it instead.

For detailed information about setting your build environment see *Build Customization*.

`shub` bootstrap agent

Overview

You can use an existing container on Singularity Hub as your “base,” and then add customization. This allows you to build multiple images from the same starting point. For example, you may want to build several containers with the same custom python installation, the same custom compiler toolchain, or the same base MPI installation. Instead of building these from scratch each time, you could create a base container on Singularity Hub and then build new containers from that existing base container adding customizations in `%post` , `%environment`, `%runscript`, etc.

Keywords

```
Bootstrap: shub
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
From: shub://<registry>/<username>/<container-name>:<tag>@digest
```

The `From` keyword is mandatory. It specifies the container to use as a base. `registry is optional and defaults to ``singularity-hub.org`. `tag` and `digest` are also optional. `tag` defaults to `latest` and `digest` can be left blank if you want the latest build.

Notes

When bootstrapping from a Singularity Hub image, all previous definition files that led to the creation of the current image will be stored in a directory within the container called `/.singularity.d/bootstrap_history`. Singularity will also alert you if environment variables have been changed between the base image and the new image during bootstrap.

`oras` bootstrap agent

Overview

Using, this module, a container from supporting OCI Registries - Eg: ACR (Azure Container Registry), local container registries, etc can be used as your "base" image and later customized. This allows you to build multiple images from the same starting point. For example, you may want to build several containers with the same custom python installation, the same custom compiler toolchain, or the same base MPI installation. Instead of building these from scratch each time, you could make use of `oras` to pull an appropriate base container and then build new containers by adding customizations in `%post` , `%environment`, `%runscript`, etc.

Keywords

```
Bootstrap: oras
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
From: oras://registry/namespace/image:tag
```

The `From` keyword is mandatory. It specifies the container to use as a base. Also,``tag`` is mandatory that refers to the version of image you want to use.

`localimage` bootstrap agent

This module allows you to build a container from an existing Singularity container on your host system. The name is somewhat misleading because your container can be in either image or directory format.

Overview

You can use an existing container image as your "base", and then add customization. This allows you to build multiple images from the same starting point. For example, you may want to build several containers with the same custom python installation, the same custom compiler toolchain, or the same base MPI installation. Instead of building these from scratch each time, you could start with the appropriate local base container and then customize the new container in `%post`, `%environment`, `%runscript`, etc.

Keywords

```
Bootstrap: localimage
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
From: /path/to/container/file/or/directory
```

The `From` keyword is mandatory. It specifies the local container to use as a base.

Notes

When building from a local container, all previous definition files that led to the creation of the current container will be stored in a directory within the container called `/.singularity.d/bootstrap_history`. Singularity will also alert you if environment variables have been changed between the base image and the new image during bootstrap.

`yum` bootstrap agent

This module allows you to build a Red Hat/CentOS/Scientific Linux style container from a mirror URI.

Overview

Use the `yum` module to specify a base for a CentOS-like container. You must also specify the URI for the mirror you would like to use.

Keywords

```
Bootstrap: yum
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
OSVersion: 7
```

The OSVersion keyword is optional. It specifies the OS version you would like to use. It is only required if you have specified a %{OSVERSION} variable in the `MirrorURL` keyword.

```
MirrorURL: http://mirror.centos.org/centos-%{OSVERSION}/%{OSVERSION}/os/$basearch/
```

The MirrorURL keyword is mandatory. It specifies the URI to use as a mirror to download the OS. If you define the `OSVersion` keyword, then you can use it in the URI as in the example above.

```
Include: yum
```

The Include keyword is optional. It allows you to install additional packages into the core operating system. It is a best practice to supply only the bare essentials such that the `%post` section has what it needs to properly complete the build. One common package you may want to install when using the `yum` build module is YUM itself.

Notes

There is a major limitation with using YUM to bootstrap a container. The RPM database that exists within the container will be created using the RPM library and Berkeley DB implementation that exists on the host system. If the RPM implementation inside the container is not compatible with the RPM database that was used to create the container, RPM and YUM commands inside the container may fail. This issue can be easily demonstrated by bootstrapping an older RHEL compatible image by a newer one (e.g. bootstrap a Centos 5 or 6 container from a Centos 7 host).

In order to use the `yum` build module, you must have `yum` installed on your system. It may seem counter-intuitive to install YUM on a system that uses a different package manager, but you can do so. For instance, on Ubuntu you can install it like so:

```
$ sudo apt-get update && sudo apt-get install yum
```

`debootstrap` build agent

This module allows you to build a Debian/Ubuntu style container from a mirror URI.

Overview

Use the `debootstrap` module to specify a base for a Debian-like container. You must also specify the OS version and a URI for the mirror you would like to use.

Keywords

```
Bootstrap: debootstrap
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
OSVersion: xenial
```

The OSVersion keyword is mandatory. It specifies the OS version you would like to use. For Ubuntu you can use code words like `trusty` (14.04), `xenial` (16.04), and `yakkety` (17.04). For Debian you can use values like `stable`, `oldstable`, `testing`, and `unstable` or code words like `wheezy` (7), `jesse` (8), and `stretch` (9).

```
MirrorURL:  http://us.archive.ubuntu.com/ubuntu/
```

The MirrorURL keyword is mandatory. It specifies a URI to use as a mirror when downloading the OS.

```
Include: somepackage
```

The Include keyword is optional. It allows you to install additional packages into the core operating system. It is a best practice to supply only the bare essentials such that the `%post` section has what it needs to properly complete the build.

Notes

In order to use the `debootstrap` build module, you must have `debootstrap` installed on your system. On Ubuntu you can install it like so:

```
$ sudo apt-get update && sudo apt-get install debootstrap
```

On CentOS you can install it from the epel repos like so:

```
$ sudo yum update && sudo yum install epel-release && sudo yum install debootstrap.
↪noarch
```

`arch` bootstrap agent

This module allows you to build a Arch Linux based container.

Overview

Use the `arch` module to specify a base for an Arch Linux based container. Arch Linux uses the aptly named `pacman` package manager (all puns intended).

Keywords

```
Bootstrap: arch
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

The Arch Linux bootstrap module does not name any additional keywords at this time. By defining the `arch` module, you have essentially given all of the information necessary for that particular bootstrap module to build a core operating system.

Notes

Arch Linux is, by design, a very stripped down, light-weight OS. You may need to perform a significant amount of configuration to get a usable OS. Please refer to this README.md and the Arch Linux example for more info.

`busybox` bootstrap agent

This module allows you to build a container based on BusyBox.

Overview

Use the `busybox` module to specify a BusyBox base for container. You must also specify a URI for the mirror you would like to use.

Keywords

```
Bootstrap: busybox
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
MirrorURL: https://www.busybox.net/downloads/binaries/1.26.1-defconfig-multiarch/
↪busybox-x86_64
```

The MirrorURL keyword is mandatory. It specifies a URI to use as a mirror when downloading the OS.

Notes

You can build a fully functional BusyBox container that only takes up ~600kB of disk space!

`zypper` bootstrap agent

This module allows you to build a Suse style container from a mirror URI.

Note: `zypper` version 1.11.20 or greater is required on the host system, as Singularity requires the `--releasever` flag.

Overview

Use the `zypper` module to specify a base for a Suse-like container. You must also specify a URI for the mirror you would like to use.

Keywords

```
Bootstrap: zypper
```

The Bootstrap keyword is always mandatory. It describes the bootstrap module to use.

```
OSVersion: 42.2
```

The OSVersion keyword is optional. It specifies the OS version you would like to use. It is only required if you have specified a %{OSVERSION} variable in the `MirrorURL` keyword.

```
Include: somepackage
```

The Include keyword is optional. It allows you to install additional packages into the core operating system. It is a best practice to supply only the bare essentials such that the `%post` section has what it needs to properly complete the build. One common package you may want to install when using the zypper build module is `zypper` itself.

`docker-daemon` and `docker-archive` bootstrap agents

If you are using docker locally there are two options for creating Singularity images without the need for a repository. You can either build a SIF from a `docker-save` tar file or you can convert any docker image present in docker's daemon internal storage.

Overview

`docker-daemon` allows you to build a SIF from any docker image currently residing in docker's daemon internal storage:

```
$ docker images alpine
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              965ea09ff2eb        7 weeks ago         5.55MB

$ singularity run docker-daemon:alpine:latest
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
Getting image source signatures
Copying blob 77cae8ab23bf done
Copying config 759e71f0d3 done
Writing manifest to image destination
Storing signatures
2019/12/11 14:53:24  info unpack layer:␣
↪sha256:eb7c47c7f0fd0054242f35366d166e6b041dfb0b89e5f93a82ad3a3206222502
INFO:    Creating SIF file...
Singularity>
```

while `docker-archive` permits you to do the same thing starting from a docker image stored in a `docker-save` formatted tar file:

```
$ docker save -o alpine.tar alpine:latest

$ singularity run docker-archive:$(pwd)/alpine.tar
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
Getting image source signatures
Copying blob 77cae8ab23bf done
Copying config 759e71f0d3 done
Writing manifest to image destination
Storing signatures
2019/12/11 15:25:09  info unpack layer:␣
↪sha256:eb7c47c7f0fd0054242f35366d166e6b041dfb0b89e5f93a82ad3a3206222502
INFO:    Creating SIF file...
Singularity>
```

Keywords

The `docker-daemon` bootstrap agent can be used in a Singularity definition file as follows:

```
From: docker-daemon:<image>:<tag>
```

where both `<image>` and `<tag>` are mandatory fields that must be written explicitly. The `docker-archive` bootstrap agent requires instead the path to the tar file containing the image:

```
From: docker-archive:<path-to-tar-file>
```

Note that differently from the `docker://` bootstrap agent both `docker-daemon` and `docker-archive` don't require a double slash `//` after the colon in the agent name.

`scratch` bootstrap agent

The scratch bootstrap agent allows you to start from a completely empty container. You are then responsible for adding any and all executables, libraries etc. that are required. Starting with a scratch container can be useful when you are aiming to minimize container size, and have a simple application / static binaries.

Overview

A minimal container providing a shell can be created by copying the `busybox` static binary into an empty scratch container:

```
Bootstrap: scratch

%setup
    # Runs on host - fetch static busybox binary
    curl -o /tmp/busybox https://www.busybox.net/downloads/binaries/1.31.0-i686-
↪uclibc/busybox
    # It needs to be executable
    chmod +x /tmp/busybox

%files
    # Copy from host into empty container
    /tmp/busybox /bin/sh

%runscript
   /bin/sh
```

The resulting container provides a shell, and is 696KiB in size:

```
$ ls -lah scratch.sif
-rwxr-xr-x. 1 dave dave 696K May 28 13:29 scratch.sif

$ singularity run scratch.sif
WARNING: passwd file doesn't exist in container, not updating
WARNING: group file doesn't exist in container, not updating
Singularity> echo "Hello from a 696KiB container"
Hello from a 696KiB container
```

Keywords

```
Bootstrap: scratch
```

There are no additional keywords for the scratch bootstrap agent.

7.2 Command Line Interface

Below are links to the automatically generated CLI docs

7.2.1 singularity

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Synopsis

Singularity containers provide an application virtualization layer enabling mobility of compute via both application and environment portability. With Singularity one is capable of building a root file system that runs on any other Linux system where Singularity is installed.

```
singularity [global options...]
```

Examples

```
$ singularity help <command> [<subcommand>]
$ singularity help build
$ singularity help instance start
```

Options

```
-c, --config string   specify a configuration file (for root or unprivileged␣
↪installation only) (default "/usr/local/etc/singularity/singularity.conf")
-d, --debug           print debugging information (highest verbosity)
-h, --help            help for singularity
    --nocolor         print without color output (default False)
-q, --quiet           suppress normal output
-s, --silent          only print errors
-v, --verbose         print additional information
```

SEE ALSO

- *singularity build* - Build a Singularity image
- *singularity cache* - Manage the local cache
- *singularity capability* - Manage Linux capabilities for users and groups
- *singularity config* - Manage various singularity configuration (root user only)
- *singularity delete* - Deletes requested image from the library
- *singularity exec* - Run a command within a container
- *singularity inspect* - Show metadata for an image
- *singularity instance* - Manage containers running as services
- *singularity key* - Manage OpenPGP keys
- *singularity oci* - Manage OCI containers
- *singularity plugin* - Manage Singularity plugins
- *singularity pull* - Pull an image from a URI
- *singularity push* - Upload image to the provided URI
- *singularity remote* - Manage singularity remote endpoints
- *singularity run* - Run the user-defined default command within a container
- *singularity run-help* - Show the user-defined help for an image
- *singularity search* - Search a Container Library for images
- *singularity shell* - Run a shell within a container
- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation
- *singularity sign* - Attach digital signature(s) to an image
- *singularity test* - Run the user-defined tests within a container
- *singularity verify* - Verify cryptographic signatures attached to an image
- *singularity version* - Show the version for Singularity

Auto generated by spf13/cobra on 13-Oct-2020

7.2.2 singularity build

Build a Singularity image

Synopsis

IMAGE PATH:

When Singularity builds the container, output can be one of a few formats:

> default: The compressed Singularity read only image format (default) sandbox: This is a read-write container within a directory structure

note: It is a common workflow to use the "sandbox" mode for development of the container, and then build it as a default Singularity image for production use. The default format is immutable.

BUILD SPEC:

The build spec target is a definition (def) file, local image, or URI that can be used to create a Singularity container. Several different local target formats exist:

> def file : This is a recipe for building a container (examples below) directory: A directory structure containing a (ch)root file system image: A local image on your machine (will convert to sif if
>
> > it is legacy format)

Targets can also be remote and defined by a URI of the following formats:

> library:// an image library (default https://cloud.sylabs.io/library) docker:// a Docker registry (default Docker Hub) shub:// a Singularity registry (default Singularity Hub) oras:// a supporting OCI registry

```
singularity build [local options...] <IMAGE PATH> <BUILD SPEC>
```

Examples

```
DEF FILE BASE OS:

    Library:
        Bootstrap: library
        From: debian:9

    Docker:
        Bootstrap: docker
        From: tensorflow/tensorflow:latest
        IncludeCmd: yes # Use the CMD as runscript instead of ENTRYPOINT

    Singularity Hub:
        Bootstrap: shub
        From: singularityhub/centos

    YUM/RHEL:
        Bootstrap: yum
        OSVersion: 7
        MirrorURL: http://mirror.centos.org/centos-%{OSVERSION}/%{OSVERSION}/os/x86_
↪64/
        Include: yum

    Debian/Ubuntu:
        Bootstrap: debootstrap
        OSVersion: trusty
```

(continues on next page)

(continued from previous page)

```
        MirrorURL: http://us.archive.ubuntu.com/ubuntu/

    Local Image:
        Bootstrap: localimage
        From: /home/dave/starter.img

    Scratch:
        Bootstrap: scratch # Populate the container with a minimal rootfs in %setup

DEFFILE SECTIONS:

    %pre
        echo "This is a scriptlet that will be executed on the host, as root before"
        echo "the container has been bootstrapped. This section is not commonly used."

    %setup
        echo "This is a scriptlet that will be executed on the host, as root, after"
        echo "the container has been bootstrapped. To install things into the␣
↪container"
        echo "reference the file system location with $SINGULARITY_ROOTFS."

    %post
        echo "This scriptlet section will be executed from within the container after"
        echo "the bootstrap/base has been created and setup."

    %test
        echo "Define any test commands that should be executed after container has␣
↪been"
        echo "built. This scriptlet will be executed from within the running container
↪"
        echo "as the root user. Pay attention to the exit/return value of this␣
↪scriptlet"
        echo "as any non-zero exit code will be assumed as failure."
        exit 0

    %runscript
        echo "Define actions for the container to be executed with the run command or"
        echo "when container is executed."

    %startscript
        echo "Define actions for container to perform when started as an instance."

    %labels
        HELLO MOTO
        KEY VALUE

    %files
        /path/on/host/file.txt /path/on/container/file.txt
        relative_file.txt /path/on/container/relative_file.txt

    %environment
        LUKE=goodguy
        VADER=badguy
        HAN=someguy
        export HAN VADER LUKE

    %help
```

(continues on next page)

(continued from previous page)

```
        This is a text file to be displayed with the run-help command.

COMMANDS:

    Build a sif file from a Singularity recipe file:
        $ singularity build /tmp/debian0.sif /path/to/debian.def

    Build a sif image from the Library:
        $ singularity build /tmp/debian1.sif library://debian:latest

    Build a base sandbox from DockerHub, make changes to it, then build sif
        $ singularity build --sandbox /tmp/debian docker://debian:latest
        $ singularity exec --writable /tmp/debian apt-get install python
        $ singularity build /tmp/debian2.sif /tmp/debian
```

Options

```
    --arch string       architecture for remote build (default "amd64")
    --builder string    remote Build Service URL, setting this implies --remote␣
↪(default "https://build.sylabs.io")
-d, --detached          submit build job and print build ID (no real-time logs and␣
↪requires --remote)
    --disable-cache     do not use cache or create cache
    --docker-login      login to a Docker Repository interactively
-e, --encrypt           build an image with an encrypted file system
-f, --fakeroot          build using user namespace to fake root user (requires a␣
↪privileged installation)
    --fix-perms         ensure owner has rwX permissions on all container content for␣
↪oci/docker sources
-F, --force             overwrite an image file if it exists
-h, --help              help for build
    --json              interpret build definition as JSON
    --library string    container Library URL (default "https://library.sylabs.io")
    --no-cleanup        do NOT clean up bundle after failed build, can be helpful for␣
↪debugging
    --nohttps           do NOT use HTTPS with the docker:// transport (useful for␣
↪local docker registries without a certificate)
-T, --notest            build without running tests in %test section
    --passphrase        prompt for an encryption passphrase
    --pem-path string   enter an path to a PEM formated RSA key for an encrypted␣
↪container
-r, --remote            build image remotely (does not require root)
-s, --sandbox           build image as sandbox format (chroot directory structure)
    --section strings   only run specific section(s) of deffile (setup, post, files,␣
↪environment, test, labels, none) (default [all])
-u, --update            run definition over existing container (skips header)
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.3 singularity cache

Manage the local cache

Synopsis

Manage your local Singularity cache. You can list/clean using the specific types.

```
singularity cache
```

Examples

```
All group commands have their own help output:

$ singularity cache
$ singularity cache --help
```

Options

```
-h, --help   help for cache
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity cache clean* - Clean your local Singularity cache * *singularity cache list* - List your local Singularity cache

Auto generated by spf13/cobra on 13-Oct-2020

7.2.4 singularity cache clean

Clean your local Singularity cache

Synopsis

This will clean your local cache (stored at $HOME/.singularity/cache if SINGULARITY_CACHEDIR is not set). By default the entire cache is cleaned, use –days and –type flags to override this behavior. Note: if you use Singularity as root, cache will be stored in '/root/.singularity/.cache', to clean that cache, you will need to run 'cache clean' as root, or with 'sudo'.

```
singularity cache clean [clean options...]
```

Examples

```
All group commands have their own help output:

$ singularity help cache clean --days 30
$ singularity help cache clean --type=library,oci
$ singularity cache clean --help
```

Options

```
-D, --days int       remove all cache entries older than specified number of days
-n, --dry-run        operate in dry run mode and do not actually clean the cache
-f, --force          suppress any prompts and clean the cache
-h, --help           help for clean
-T, --type strings   a list of cache types to clean (possible values: library, oci,␣
↪shub, blob, net, oras, all) (default [all])
```

SEE ALSO

- *singularity cache* - Manage the local cache

Auto generated by spf13/cobra on 13-Oct-2020

7.2.5 singularity cache list

List your local Singularity cache

Synopsis

This will list your local cache (stored at $HOME/.singularity/cache if SINGULARITY_CACHEDIR is not set).

```
singularity cache list [list options...]
```

Examples

```
All group commands have their own help output:

$ singularity help cache list
$ singularity help cache list --type=library,oci
$ singularity cache list --help
```

Options

```
-h, --help           help for list
-T, --type strings   a list of cache types to display, possible entries: library, oci,
↪ shub, blob(s), all (default [all])
-v, --verbose        include cache entries in the output
```

SEE ALSO

- *singularity cache* - Manage the local cache

Auto generated by spf13/cobra on 13-Oct-2020

7.2.6 singularity capability

Manage Linux capabilities for users and groups

Synopsis

Capabilities allow you to have fine grained control over the permissions that your containers need to run.

NOTE: capability add/drop commands require root to run. Granting capabilities to users allows them to escalate privilege inside the container and will likely give them a route to privilege escalation on the host system as well. Do not add capabilities to users who should not have root on the host system.

```
singularity capability
```

Examples

```
All group commands have their own help output:

$ singularity help capability add
$ singularity capability add --help
```

Options

```
-h, --help   help for capability
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity capability add* - Add capabilities to a user or group (requires root) * *singularity capability avail* - Show description for available capabilities * *singularity capability drop* - Remove capabilities from a user or group (requires root) * *singularity capability list* - Show capabilities for a given user or group

Auto generated by spf13/cobra on 13-Oct-2020

7.2.7 singularity capability add

Add capabilities to a user or group (requires root)

Synopsis

Add Linux capabilities to a user or group. NOTE: This command requires root to run.

The capabilities argument must be separated by commas and is not case sensitive.

To see available capabilities, type "singularity capability avail" or refer to capabilities manual "man 7 capabilities".

```
singularity capability add [add options...] <capabilities>
```

Examples

```
$ sudo singularity capability add --user nobody AUDIT_READ,chown
$ sudo singularity capability add --group nobody cap_audit_write

To add all capabilities to a user:

$ sudo singularity capability add --user nobody all
```

Options

```
-g, --group string   manage capabilities for a group
-h, --help           help for add
-u, --user string    manage capabilities for a user
```

SEE ALSO

- *singularity capability* - Manage Linux capabilities for users and groups

Auto generated by spf13/cobra on 13-Oct-2020

7.2.8 singularity capability avail

Show description for available capabilities

Synopsis

Show description for available Linux capabilities.

```
singularity capability avail [capabilities]
```

Examples

```
Show description for all available capabilities:

$ singularity capability avail

Show CAP_CHOWN description:

$ singularity capability avail CAP_CHOWN

Show CAP_CHOWN/CAP_NET_RAW description:

$ singularity capability avail CAP_CHOWN,CAP_NET_RAW
```

Options

```
-h, --help   help for avail
```

SEE ALSO

- *singularity capability* - Manage Linux capabilities for users and groups

Auto generated by spf13/cobra on 13-Oct-2020

7.2.9 singularity capability drop

Remove capabilities from a user or group (requires root)

Synopsis

Remove Linux capabilities from a user/group. NOTE: This command requires root to run.

The capabilities argument must be separated by commas and is not case sensitive.

To see available capabilities, type "singularity capability avail" or refer to capabilities manual "man 7 capabilities"

```
singularity capability drop [drop options...] <capabilities>
```

Examples

```
$ sudo singularity capability drop --user nobody AUDIT_READ,CHOWN
$ sudo singularity capability drop --group nobody audit_write

To drop all capabilities for a user:

$ sudo singularity capability drop --user nobody all
```

Options

```
-g, --group string   manage capabilities for a group
-h, --help           help for drop
-u, --user string    manage capabilities for a user
```

SEE ALSO

- *singularity capability* - Manage Linux capabilities for users and groups

Auto generated by spf13/cobra on 13-Oct-2020

7.2.10 singularity capability list

Show capabilities for a given user or group

Synopsis

Show the capabilities for a user or group.

```
singularity capability list [user/group]
```

Examples

```
To list capabilities set for user or group nobody:

$ singularity capability list nobody

To list capabilities for all users/groups:

$ singularity capability list
```

Options

```
-h, --help   help for list
```

SEE ALSO

- *singularity capability* - Manage Linux capabilities for users and groups

Auto generated by spf13/cobra on 13-Oct-2020

7.2.11 singularity config

Manage various singularity configuration (root user only)

Synopsis

The config command allows root user to manage various configuration like fakeroot user mapping entries.

Examples

```
All config commands have their own help output:

$ singularity help config fakeroot
$ singularity config fakeroot --help
```

Options

```
-h, --help   help for config
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity config fakeroot* - Manage fakeroot user mappings entries (root user only) * *singularity config global* - Edit singularity.conf from command line (root user only or unprivileged installation)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.12 singularity config fakeroot

Manage fakeroot user mappings entries (root user only)

Synopsis

The config fakeroot command allow a root user to add/remove/enable/disable fakeroot user mappings.

```
singularity config fakeroot <option> <user>
```

Examples

```
To add a fakeroot user mapping for vagrant user:
$ singularity config fakeroot --add vagrant

To remove a fakeroot user mapping for vagrant user:
$ singularity config fakeroot --remove vagrant

To disable a fakeroot user mapping for vagrant user:
$ singularity config fakeroot --disable vagrant

To enable a fakeroot user mapping for vagrant user:
$ singularity config fakeroot --enable vagrant
```

Options

```
-a, --add       add a fakeroot mapping entry for a user allowing him to use the␣
↪fakeroot feature
-d, --disable   disable a user fakeroot mapping entry preventing him to use the␣
↪fakeroot feature (the user mapping must be present)
-e, --enable    enable a user fakeroot mapping entry allowing him to use the fakeroot␣
↪feature (the user mapping must be present)
-h, --help      help for fakeroot
-r, --remove    remove the user fakeroot mapping entry preventing him to use the␣
↪fakeroot feature
```

SEE ALSO

- *singularity config* - Manage various singularity configuration (root user only)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.13 singularity config global

Edit singularity.conf from command line (root user only or unprivileged installation)

Synopsis

The config global command allow administrators to set/unset/get/reset configuration directives of singularity.conf from command line.

```
singularity config global <option> <directive> [value,...]
```

Examples

```
To add a path to "bind path" directive:
$ singularity config global --set "bind path" /etc/resolv.conf

To remove a path from "bind path" directive:
$ singularity config global --unset "bind path" /etc/resolv.conf

To set "bind path" to the default value:
$ singularity config global --reset "bind path"

To get "bind path" directive value:
$ singularity config global --get "bind path"

To display the resulting configuration instead of writing it to file:
$ singularity config global --dry-run --set "bind path" /etc/resolv.conf
```

Options

```
-d, --dry-run   dump resulting configuration on stdout but doesn't write it to␣
↪singularity.conf
-g, --get       get value of the configuration directive
-h, --help      help for global
-r, --reset     reset the configuration directive value to its default value
-s, --set       set value of the configuration directive (for multi-value directives,␣
↪it will add it)
-u, --unset     unset value of the configuration directive (for multi-value␣
↪directives, it will remove matching values)
```

SEE ALSO

- *singularity config* - Manage various singularity configuration (root user only)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.14 singularity delete

Deletes requested image from the library

Synopsis

The 'delete' command allows you to delete an image from a remote library.

```
singularity delete [arch] <imageRef> [flags]
```

Examples

```
$ singularity delete --arch=amd64 library://username/project/image:1.0
```

Options

```
-A, --arch string      specify requested image arch (default "amd64")
-F, --force            delete image without confirmation
-h, --help             help for delete
    --library string   delete images from the provided library (default "https://
↪library.sylabs.io")
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.15 singularity exec

Run a command within a container

Synopsis

singularity exec supports the following formats:

*.sif Singularity Image Format (SIF). Native to Singularity 3.0+

*.sqsh SquashFS format. Native to Singularity 2.4+

*.img ext3 format. Native to Singularity versions < 2.4.

directory/ sandbox format. Directory containing a valid root file system and optionally Singularity meta-data.

instance://* A local running instance of a container. (See the instance command group.)

library://* A container hosted on a Library (default https://cloud.sylabs.io/library)

docker://* A container hosted on Docker Hub

shub://* A container hosted on Singularity Hub

oras://* A container hosted on a supporting OCI registry

```
singularity exec [exec options...] <container> <command>
```

Examples

```
$ singularity exec /tmp/debian.sif cat /etc/debian_version
$ singularity exec /tmp/debian.sif python ./hello_world.py
$ cat hello_world.py | singularity exec /tmp/debian.sif python
$ sudo singularity exec --writable /tmp/debian.sif apt-get update
$ singularity exec instance://my_instance ps -ef
$ singularity exec library://centos cat /etc/os-release
```

Options

```
    --add-caps string        a comma separated capability list to add
    --allow-setuid           allow setuid binaries in container (root only)
    --app string             set an application to run inside a container
    --apply-cgroups string   apply cgroups from file for container processes (root␣
↪only)
-B, --bind strings           a user-bind path specification.  spec has the format␣
↪src[:dest[:opts]], where src and dest are outside and inside paths.  If dest is not␣
↪given, it is set equal to src.  Mount options ('opts') may be specified as 'ro'␣
↪(read-only) or 'rw' (read/write, which is the default). Multiple bind paths can be␣
↪given by a comma separated list.
-e, --cleanenv               clean environment before running container
-c, --contain                use minimal /dev and empty other directories (e.g. /tmp␣
↪and $HOME) instead of sharing filesystems from your host
-C, --containall             contain not only file systems, but also PID, IPC, and␣
↪environment
    --disable-cache          dont use cache, and dont create cache
    --dns string             list of DNS server separated by commas to add in resolv.
↪conf
    --docker-login           login to a Docker Repository interactively
    --drop-caps string       a comma separated capability list to drop
    --env strings            pass environment variable to contained process
```

(continues on next page)

(continued from previous page)

```
    --env-file string        pass environment variables from file to contained process
-f, --fakeroot               run container in new user namespace as uid 0
    --fusemount strings      A FUSE filesystem mount specification of the form '<type>
↪:<fuse command> <mountpoint>' - where <type> is 'container' or 'host', specifying␣
↪where the mount will be performed ('container-daemon' or 'host-daemon' will run the␣
↪FUSE process detached). <fuse command> is the path to the FUSE executable, plus␣
↪options for the mount. <mountpoint> is the location in the container to which the␣
↪FUSE mount will be attached. E.g. 'container:sshfs 10.0.0.1:/ /sshfs'. Implies --
↪pid.
-h, --help                   help for exec
-H, --home string            a home directory specification.  spec can either be a␣
↪src path or src:dest pair.  src is the source path of the home directory outside␣
↪the container and dest overrides the home directory within the container. (default
↪"/home/dave")
    --hostname string        set container hostname
-i, --ipc                    run container in a new IPC namespace
    --keep-privs             let root user keep privileges in container (root only)
-n, --net                    run container in a new network namespace (sets up a␣
↪bridge network interface by default)
    --network string         specify desired network type separated by commas, each␣
↪network will bring up a dedicated interface inside container (default "bridge")
    --network-args strings   specify network arguments to pass to CNI plugins
    --no-home                do NOT mount users home directory if /home is not the␣
↪current working directory
    --no-init                do NOT start shim process with --pid
    --no-privs               drop all privileges from root user in container)
    --nohttps                do NOT use HTTPS with the docker:// transport (useful␣
↪for local docker registries without a certificate)
    --nonet                  disable VM network handling
    --nv                     enable experimental Nvidia support
-o, --overlay strings        use an overlayFS image for persistent data storage or as␣
↪read-only layer of container
    --passphrase             prompt for an encryption passphrase
    --pem-path string        enter an path to a PEM formated RSA key for an encrypted␣
↪container
-p, --pid                    run container in a new PID namespace
    --pwd string             initial working directory for payload process inside the␣
↪container
    --rocm                   enable experimental Rocm support
-S, --scratch strings        include a scratch directory within the container that is␣
↪linked to a temporary dir (use -W to force location)
    --security strings       enable security features (SELinux, Apparmor, Seccomp)
-u, --userns                 run container in a new user namespace, allowing␣
↪Singularity to run completely unprivileged on recent kernels. This disables some␣
↪features of Singularity, for example it only works with sandbox images.
    --uts                    run container in a new UTS namespace
    --vm                     enable VM support
    --vm-cpu string          number of CPU cores to allocate to Virtual Machine␣
↪(implies --vm) (default "1")
    --vm-err                 enable attaching stderr from VM
    --vm-ip string           IP Address to assign for container usage. Defaults to␣
↪DHCP within bridge network. (default "dhcp")
    --vm-ram string          amount of RAM in MiB to allocate to Virtual Machine␣
↪(implies --vm) (default "1024")
-W, --workdir string         working directory to be used for /tmp, /var/tmp and
↪$HOME (if -c/--contain was also used)
-w, --writable               by default all Singularity containers are available as␣
↪read only. This option makes the file system accessible as read/write.
```

(continues on next page)

(continued from previous page)

```
    --writable-tmpfs         makes the file system accessible as read-write with non␣
↪persistent data (with overlay support only)
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.16 singularity inspect

Show metadata for an image

Synopsis

> Inspect will show you labels, environment variables, apps and scripts associated with the image determined by the flags you pass. By default, they will be shown in plain text. If you would like to list them in json format, you should use the –json flag.

```
singularity inspect [inspect options...] <image path>
```

Examples

```
$ singularity inspect ubuntu.sif

If you want to list the applications (apps) installed in a container (located at
/scif/apps) you should run inspect command with --list-apps <container-image> flag.
( See https://sci-f.github.io for more information on SCIF apps)

The following environment variables are available to you when called
from the shell inside the container. The top variables are relevant
to the active app (--app <app>) and the bottom available for all
apps regardless of the active app. Both sets of variables are also available during␣
↪development (at build time).

ACTIVE APP ENVIRONMENT:
    SCIF_APPNAME       the name for the active application
    SCIF_APPROOT       the installation folder for the application created at /scif/
↪apps/<app>
    SCIF_APPMETA       the application metadata folder
    SCIF_APPDATA       the data folder created for the application at /scif/data/<app>
      SCIF_APPINPUT    expected input folder within data base folder
      SCIF_APPOUTPUT   the output data folder within data base folder

    SCIF_APPENV        points to the application's custom environment.sh file in its␣
↪metadata folder
    SCIF_APPLABELS     is the application's labels.json in the metadata folder
    SCIF_APPBIN        is the bin folder for the app, which is automatically added to␣
↪the $PATH when the app is active
```

(continues on next page)

(continued from previous page)

```
    SCIF_APPLIB        is the application's library folder that is added to the LD_
↪LIBRARY_PATH
    SCIF_APPRUN        is the runscript
    SCIF_APPHELP       is the help file for the runscript
    SCIF_APPTEST       is the testing script (test.sh) associated with the applicatio
    SCIF_APPNAME       the name for the active application
    SCIF_APPFILES      the files section associated with the application that are␣
↪added to


GLOBAL APP ENVIRONMENT:

    SCIF_DATA             scif defined data base for all apps (/scif/data)
    SCIF_APPS             scif defined install bases for all apps (/scif/apps)
    SCIF_APPROOT_<app>    root for application <app>
    SCIF_APPDATA_<app>    data root for application <app>

To list all your apps:

$ singularity inspect --list-apps ubuntu.sif

To list only labels in the json format from an image:

$ singularity inspect --json --labels ubuntu.sif

To verify you own a single application on your container image, use the --app
↪<appname> flag:

$ singularity inspect --app <appname> ubuntu.sif
```

Options

```
    --all             show all available data (imply --json option)
    --app string      inspect a specific app
-d, --deffile         show the Singularity recipe file that was used to generate the␣
↪image
-e, --environment     show the environment settings for the image
-h, --help            help for inspect
-H, --helpfile        inspect the runscript helpfile, if it exists
-j, --json            print structured json instead of sections
-l, --labels          show the labels for the image (default)
    --list-apps       list all apps in a container
-r, --runscript       show the runscript for the image
-s, --startscript     show the startscript for the image
-t, --test            show the test script for the image
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.17 singularity instance

Manage containers running as services

Synopsis

> Instances allow you to run containers as background processes. This can be useful for running services such as web servers or databases.

```
singularity instance
```

Examples

```
All group commands have their own help output:

$ singularity help instance start
$ singularity instance start --help
```

Options

```
-h, --help   help for instance
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity instance list* - List all running and named Singularity instances * *singularity instance start* - Start a named instance of the given container image * *singularity instance stop* - Stop a named instance of a given container image

Auto generated by spf13/cobra on 13-Oct-2020

7.2.18 singularity instance list

List all running and named Singularity instances

Synopsis

The instance list command allows you to view the Singularity container instances that are currently running in the background.

```
singularity instance list [list options...] [<instance name glob>]
```

Examples

```
$ singularity instance list
INSTANCE NAME      PID       IMAGE
test               11963     /home/mibauer/singularity/sinstance/test.sif
test2              11964     /home/mibauer/singularity/sinstance/test.sif
lolcow             11965     /home/mibauer/singularity/sinstance/lolcow.sif

$ singularity instance list 'test*'
INSTANCE NAME      PID       IMAGE
test               11963     /home/mibauer/singularity/sinstance/test.sif
test2              11964     /home/mibauer/singularity/sinstance/test.sif

$ sudo singularity instance list -u mibauer
INSTANCE NAME      PID       IMAGE
test               11963     /home/mibauer/singularity/sinstance/test.sif
test2              16219     /home/mibauer/singularity/sinstance/test.sif
```

Options

```
-h, --help          help for list
-j, --json          print structured json instead of list
-l, --logs          display location of stdout and sterr log files for instances
-u, --user string   if running as root, list instances from "<username>"
```

SEE ALSO

- *singularity instance* - Manage containers running as services

Auto generated by spf13/cobra on 13-Oct-2020

7.2.19 singularity instance start

Start a named instance of the given container image

Synopsis

The instance start command allows you to create a new named instance from an existing container image that will begin running in the background. If a startscript is defined in the container metadata the commands in that script will be executed with the instance start command as well. You can optionally pass arguments to startscript

singularity instance start accepts the following container formats

*.sif Singularity Image Format (SIF). Native to Singularity 3.0+

*.sqsh SquashFS format. Native to Singularity 2.4+

*.img ext3 format. Native to Singularity versions < 2.4.

directory/ sandbox format. Directory containing a valid root file system and optionally Singularity meta-data.

instance://* A local running instance of a container. (See the instance command group.)

library://* A container hosted on a Library (default https://cloud.sylabs.io/library)

docker://* A container hosted on Docker Hub

shub://* A container hosted on Singularity Hub

oras://* A container hosted on a supporting OCI registry

```
singularity instance start [start options...] <container path> <instance name>␣
↪[startscript args...]
```

Examples

```
$ singularity instance start /tmp/my-sql.sif mysql

$ singularity shell instance://mysql
Singularity my-sql.sif> pwd
/home/mibauer/mysql
Singularity my-sql.sif> ps
PID TTY          TIME CMD
  1 pts/0    00:00:00 sinit
  2 pts/0    00:00:00 bash
  3 pts/0    00:00:00 ps
Singularity my-sql.sif>

$ singularity instance stop /tmp/my-sql.sif mysql
Stopping /tmp/my-sql.sif mysql
```

Options

```
    --add-caps string        a comma separated capability list to add
    --allow-setuid           allow setuid binaries in container (root only)
    --apply-cgroups string   apply cgroups from file for container processes (root␣
↪only)
-B, --bind strings           a user-bind path specification.  spec has the format␣
↪src[:dest[:opts]], where src and dest are outside and inside paths.  If dest is not␣
↪given, it is set equal to src.  Mount options ('opts') may be specified as 'ro'␣
↪(read-only) or 'rw' (read/write, which is the default). Multiple bind paths can be␣
↪given by a comma separated list.
    --boot                   execute /sbin/init to boot container (root only)
-e, --cleanenv               clean environment before running container
-c, --contain                use minimal /dev and empty other directories (e.g. /tmp␣
↪and $HOME) instead of sharing filesystems from your host
-C, --containall             contain not only file systems, but also PID, IPC, and␣
↪environment
    --disable-cache          dont use cache, and dont create cache
    --dns string             list of DNS server separated by commas to add in resolv.
↪conf
    --docker-login           login to a Docker Repository interactively
    --drop-caps string       a comma separated capability list to drop
    --env strings            pass environment variable to contained process
    --env-file string        pass environment variables from file to contained process
-f, --fakeroot               run container in new user namespace as uid 0
    --fusemount strings      A FUSE filesystem mount specification of the form '<type>
↪:<fuse command> <mountpoint>' - where <type> is 'container' or 'host', specifying␣
↪where the mount will be performed ('container-daemon' or 'host-daemon' will run the␣
↪FUSE process detached). <fuse command> is the path to the FUSE executable, plus␣
↪options for the mount. <mountpoint> is the location in the container to which the␣
↪FUSE mount will be attached. E.g. 'container:sshfs 10.0.0.1:/ /sshfs'. Implies --
↪pid.
-h, --help                   help for start
-H, --home string            a home directory specification.  spec can either be a␣
↪src path or src:dest pair.  src is the source path of the home directory outside␣
↪the container and dest overrides the home directory within the container. (default
↪"/home/dave")
    --hostname string        set container hostname
    --keep-privs             let root user keep privileges in container (root only)
-n, --net                    run container in a new network namespace (sets up a␣
↪bridge network interface by default)
    --network string         specify desired network type separated by commas, each␣
↪network will bring up a dedicated interface inside container (default "bridge")
    --network-args strings   specify network arguments to pass to CNI plugins
    --no-home                do NOT mount users home directory if /home is not the␣
↪current working directory
    --no-init                do NOT start shim process with --pid
    --no-privs               drop all privileges from root user in container)
    --nohttps                do NOT use HTTPS with the docker:// transport (useful␣
↪for local docker registries without a certificate)
    --nv                     enable experimental Nvidia support
-o, --overlay strings        use an overlayFS image for persistent data storage or as␣
↪read-only layer of container
    --passphrase             prompt for an encryption passphrase
    --pem-path string        enter an path to a PEM formated RSA key for an encrypted␣
↪container
    --pid-file string        write instance PID to the file with the given name
```

(continues on next page)

(continued from previous page)

```
    --rocm                     enable experimental Rocm support
-S, --scratch strings          include a scratch directory within the container that is␣
↪linked to a temporary dir (use -W to force location)
    --security strings         enable security features (SELinux, Apparmor, Seccomp)
-u, --userns                   run container in a new user namespace, allowing␣
↪Singularity to run completely unprivileged on recent kernels. This disables some␣
↪features of Singularity, for example it only works with sandbox images.
    --uts                      run container in a new UTS namespace
-W, --workdir string           working directory to be used for /tmp, /var/tmp and
↪$HOME (if -c/--contain was also used)
-w, --writable                 by default all Singularity containers are available as␣
↪read only. This option makes the file system accessible as read/write.
    --writable-tmpfs           makes the file system accessible as read-write with non␣
↪persistent data (with overlay support only)
```

SEE ALSO

- *singularity instance* - Manage containers running as services

Auto generated by spf13/cobra on 13-Oct-2020

7.2.20 singularity instance stop

Stop a named instance of a given container image

Synopsis

The command singularity instance stop allows you to stop and clean up a named, running instance of a given container image.

```
singularity instance stop [stop options...] [instance]
```

Examples

```
$ singularity instance start my-sql.sif mysql1
$ singularity instance start my-sql.sif mysql2
$ singularity instance stop mysql*
Stopping mysql1 instance of my-sql.sif (PID=23845)
Stopping mysql2 instance of my-sql.sif (PID=23858)

$ singularity instance start my-sql.sif mysql1

Force instance to shutdown
$ singularity instance stop -f mysql1 (may corrupt data)

Send SIGTERM to the instance
$ singularity instance stop -s SIGTERM mysql1
$ singularity instance stop -s TERM mysql1
$ singularity instance stop -s 15 mysql1
```

Options

```
-a, --all              stop all user's instances
-F, --force            force kill instance
-h, --help             help for stop
-s, --signal string    signal sent to the instance
-t, --timeout int      force kill non stopped instances after X seconds (default 10)
-u, --user string      if running as root, stop instances belonging to user
```

SEE ALSO

- *singularity instance* - Manage containers running as services

Auto generated by spf13/cobra on 13-Oct-2020

7.2.21 singularity key

Manage OpenPGP keys

Synopsis

> Manage your trusted, public and private keys in your keyring (default: '~/.singularity/sypgp' if 'SINGU-LARITY_SYPGPDIR' is not set.)

```
singularity key [key options...]
```

Examples

```
All group commands have their own help output:

$ singularity help key newpair
$ singularity key list --help
```

Options

```
-h, --help   help for key
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity key export* - Export a public or private key into a specific file * *singularity key import* - Import a local key into the local keyring * *singularity key list* - List keys in your local keyring * *singularity key newpair* - Create a new key pair * *singularity key pull* - Download a public key from a key server * *singularity key push* - Upload a public key to a key server * *singularity key remove* - Remove a local public key from your keyring * *singularity key search* - Search for keys on a key server

Auto generated by spf13/cobra on 13-Oct-2020

7.2.22 singularity key export

Export a public or private key into a specific file

Synopsis

The 'key export' command allows you to export a key and save it to a file.

```
singularity key export [export options...] <output-file>
```

Examples

```
Exporting a private key:

$ singularity key export --secret ./private.asc

Exporting a public key:

$ singularity key export ./public.asc
```

Options

```
-a, --armor    ascii armored format
-h, --help     help for export
-s, --secret   export a secret key
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.23 singularity key import

Import a local key into the local keyring

Synopsis

The 'key import' command allows you to add a key to your local keyring from a specific file.

```
singularity key import [import options...] <input-key>
```

Examples

```
$ singularity key import ./my-key.asc
```

Options

```
-h, --help           help for import
    --new-password   set a new password to the private key
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.24 singularity key list

List keys in your local keyring

Synopsis

List your local keys in your keyring. Will list public (trusted) keys by default.

```
singularity key list
```

Examples

```
$ singularity key list
$ singularity key list --secret
```

Options

```
-h, --help     help for list
-s, --secret   list private keys instead of the default which displays public ones
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.25 singularity key newpair

Create a new key pair

Synopsis

The 'key newpair' command allows you to create a new key or public/private keys to be stored in the default user local key store location (e.g., $HOME/.singularity/sypgp).

```
singularity key newpair
```

Examples

```
$ singularity key newpair
$ singularity key newpair --password=psk --name=your-name --comment="key comment" --
↪email=mail@email.com --push=false
```

Options

```
-b, --bit-length int    specify key bit length (default 4096)
-C, --comment string    key comment
-E, --email string      key owner email
-h, --help              help for newpair
-N, --name string       key owner name
-P, --password string   key password
-U, --push              specify to push the public key to the remote keystore␣
↪(default true)
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.26 singularity key pull

Download a public key from a key server

Synopsis

The 'key pull' command allows you to connect to a key server look for and download a public key. Key rings are stored into (e.g., $HOME/.singularity/sypgp).

```
singularity key pull [pull options...] <fingerprint>
```

Examples

```
$ singularity key pull 8883491F4268F173C6E5DC49EDECE4F3F38D871E
```

Options

```
-h, --help          help for pull
-u, --url string    specify the key server URL (default "https://keys.sylabs.io")
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.27 singularity key push

Upload a public key to a key server

Synopsis

> The 'key push' command allows you to connect to a key server and upload public keys from the local key store.

```
singularity key push [push options...] <fingerprint>
```

Examples

```
$ singularity key push 8883491F4268F173C6E5DC49EDECE4F3F38D871E
```

Options

```
-h, --help          help for push
-u, --url string    specify the key server URL (default "https://keys.sylabs.io")
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.28 singularity key remove

Remove a local public key from your keyring

Synopsis

The 'key remove' command will remove a local public key from your keyring.

```
singularity key remove <fingerprint>
```

Examples

```
$ singularity key remove D87FE3AF5C1F063FCBCC9B02F812842B5EEE5934
```

Options

```
-h, --help   help for remove
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.29 singularity key search

Search for keys on a key server

Synopsis

The 'key search' command allows you to connect to a key server and look for public keys matching the argument passed to the command line. You can search by name, email, or fingerprint / key ID. (Maximum 100 search entities)

```
singularity key search [search options...] <search_string>
```

Examples

```
$ singularity key search sylabs.io

# search by fingerprint:
$ singularity key search 8883491F4268F173C6E5DC49EDECE4F3F38D871E

# search by key ID:
$ singularity key search F38D871E
```

Options

```
-h, --help          help for search
-l, --long-list     output long list when searching for keys
-u, --url string    specify the key server URL (default "https://keys.sylabs.io")
```

SEE ALSO

- *singularity key* - Manage OpenPGP keys

Auto generated by spf13/cobra on 13-Oct-2020

7.2.30 singularity oci

Manage OCI containers

Synopsis

> Allow you to manage containers from OCI bundle directories.
>
> NOTE: all oci commands requires to run as root

Examples

```
All group commands have their own help output:

$ singularity oci create -b ~/bundle mycontainer
$ singularity oci start mycontainer
```

Options

```
-h, --help   help for oci
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity oci attach* - Attach console to a running container process (root user only) * *singularity oci create* - Create a container from a bundle directory (root user only) * *singularity oci delete* - Delete container (root user only) * *singularity oci exec* - Execute a command within container (root user only) * *singularity oci kill* - Kill a container (root user only) * *singularity oci mount* - Mount create an OCI bundle from SIF image (root user only) * *singularity oci pause* - Suspends all processes inside the container (root user only) * *singularity oci resume* - Resumes all processes previously paused inside the container (root user only) * *singularity oci run* - Create/start/attach/delete a container from a bundle directory (root user only) * *singularity oci start* - Start container process (root user only) * *singularity oci state* - Query state of a container (root user only) * *singularity oci umount* - Umount delete bundle (root user only) * *singularity oci update* - Update container cgroups resources (root user only)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.31 singularity oci attach

Attach console to a running container process (root user only)

Synopsis

Attach will attach console to a running container process running within container identified by container ID.

```
singularity oci attach <container_ID>
```

Examples

```
$ singularity oci attach mycontainer
```

Options

```
-h, --help   help for attach
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.32 singularity oci create

Create a container from a bundle directory (root user only)

Synopsis

Create invoke create operation to create a container instance from an OCI bundle directory

```
singularity oci create -b <bundle_path> [create options...] <container_ID>
```

Examples

```
$ singularity oci create -b ~/bundle mycontainer
```

Options

```
-b, --bundle string        specify the OCI bundle path (required)
    --empty-process        run container without executing container process (eg: for␣
↪POD container)
-h, --help                 help for create
    --log-format string    specify the log file format. Available formats are basic,␣
↪kubernetes and json (default "kubernetes")
-l, --log-path string      specify the log file path
    --pid-file string      specify the pid file
-s, --sync-socket string   specify the path to unix socket for state synchronization
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.33 singularity oci delete

Delete container (root user only)

Synopsis

Delete invoke delete operation to delete resources that were created for container identified by container ID.

```
singularity oci delete <container_ID>
```

Examples

```
$ singularity oci delete mycontainer
```

Options

```
-h, --help   help for delete
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.34 singularity oci exec

Execute a command within container (root user only)

Synopsis

Exec will execute the provided command/arguments within container identified by container ID.

```
singularity oci exec <container_ID> <command> <args>
```

Examples

```
$ singularity oci exec mycontainer id
```

Options

```
-h, --help   help for exec
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.35 singularity oci kill

Kill a container (root user only)

Synopsis

Kill invoke kill operation to kill processes running within container identified by container ID.

```
singularity oci kill [kill options...] <container_ID>
```

Examples

```
$ singularity oci kill mycontainer INT
$ singularity oci kill mycontainer -s INT
```

Options

```
-f, --force             kill container process with SIGKILL
-h, --help              help for kill
-s, --signal string     signal sent to the container (default "SIGTERM")
-t, --timeout uint32    timeout in second before killing container
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.36 singularity oci mount

Mount create an OCI bundle from SIF image (root user only)

Synopsis

Mount will mount and create an OCI bundle from a SIF image.

```
singularity oci mount <sif_image> <bundle_path>
```

Examples

```
$ singularity oci mount /tmp/example.sif /var/lib/singularity/bundles/example
```

Options

```
-h, --help   help for mount
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.37 singularity oci pause

Suspends all processes inside the container (root user only)

Synopsis

Pause will suspend all processes for the specified container ID.

```
singularity oci pause <container_ID>
```

Examples

```
$ singularity oci pause mycontainer
```

Options

```
-h, --help   help for pause
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.38 singularity oci resume

Resumes all processes previously paused inside the container (root user only)

Synopsis

Resume will resume all processes previously paused for the specified container ID.

```
singularity oci resume <container_ID>
```

Examples

```
$ singularity oci resume mycontainer
```

Options

```
-h, --help   help for resume
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.39 singularity oci run

Create/start/attach/delete a container from a bundle directory (root user only)

Synopsis

Run will invoke equivalent of create/start/attach/delete commands in a row.

```
singularity oci run -b <bundle_path> [run options...] <container_ID>
```

Examples

```
$ singularity oci run -b ~/bundle mycontainer

is equivalent to :

$ singularity oci create -b ~/bundle mycontainer
$ singularity oci start mycontainer
$ singularity oci attach mycontainer
$ singularity oci delete mycontainer
```

Options

```
-b, --bundle string        specify the OCI bundle path (required)
-h, --help                 help for run
    --log-format string    specify the log file format. Available formats are basic,␣
↪kubernetes and json (default "kubernetes")
-l, --log-path string      specify the log file path
    --pid-file string      specify the pid file
-s, --sync-socket string   specify the path to unix socket for state synchronization
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.40 singularity oci start

Start container process (root user only)

Synopsis

Start invoke start operation to start a previously created container identified by container ID.

```
singularity oci start <container_ID>
```

Examples

```
$ singularity oci start mycontainer
```

Options

```
-h, --help   help for start
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.41 singularity oci state

Query state of a container (root user only)

Synopsis

State invoke state operation to query state of a created/running/stopped container identified by container ID.

```
singularity oci state <container_ID>
```

Examples

```
$ singularity oci state mycontainer
```

Options

```
-h, --help                 help for state
-s, --sync-socket string   specify the path to unix socket for state synchronization
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.42 singularity oci umount

Umount delete bundle (root user only)

Synopsis

Umount will umount an OCI bundle previously mounted with singularity oci mount.

```
singularity oci umount <bundle_path>
```

Examples

```
$ singularity oci umount /var/lib/singularity/bundles/example
```

Options

```
-h, --help   help for umount
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.43 singularity oci update

Update container cgroups resources (root user only)

Synopsis

Update will update cgroups resources for the specified container ID. Container must be in a RUNNING or CREATED state.

```
singularity oci update [update options...] <container_ID>
```

Examples

```
$ singularity oci update --from-file /tmp/cgroups-update.json mycontainer

or to update from stdin :

$ cat /tmp/cgroups-update.json | singularity oci update --from-file - mycontainer
```

Options

```
-f, --from-file string   specify path to OCI JSON cgroups resource file ('-' to read␣
↪from STDIN)
-h, --help               help for update
```

SEE ALSO

- *singularity oci* - Manage OCI containers

Auto generated by spf13/cobra on 13-Oct-2020

7.2.44 singularity plugin

Manage Singularity plugins

Synopsis

The 'plugin' command allows you to manage Singularity plugins which provide add-on functionality to the default Singularity installation.

```
singularity plugin [plugin options...]
```

Examples

```
All group commands have their own help output:

$ singularity help plugin compile
$ singularity plugin list --help
```

Options

```
-h, --help   help for plugin
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity plugin compile* - Compile a Singularity plugin * *singularity plugin create* - Create a plugin skeleton directory * *singularity plugin disable* - disable an installed Singularity plugin * *singularity plugin enable* - Enable an installed Singularity plugin * *singularity plugin inspect* - Inspect a singularity plugin (either an installed one or an image) * *singularity plugin install* - Install a compiled Singularity plugin * *singularity plugin list* - List installed Singularity plugins * *singularity plugin uninstall* - Uninstall removes the named plugin from the system

Auto generated by spf13/cobra on 13-Oct-2020

7.2.45 singularity plugin compile

Compile a Singularity plugin

Synopsis

The 'plugin compile' command allows a developer to compile a Singularity plugin in the expected environment. The provided host directory is the location of the plugin's source code. A compiled plugin is packed into a SIF file.

```
singularity plugin compile [compile options...] <host_path>
```

Examples

```
$ singularity plugin compile $HOME/singularity/test-plugin
```

Options

```
    --disable-minor-check   disable minor package version check
-h, --help                  help for compile
-o, --out string            path of the SIF output file
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.46 singularity plugin create

Create a plugin skeleton directory

Synopsis

The 'plugin create' command allows a user to creates a plugin skeleton directory structure to start development of a new plugin.

```
singularity plugin create <host_path> <name>
```

Examples

```
$ singularity plugin create ~/myplugin github.com/username/myplugin
$ ls -1 ~/myplugin
go.mod
main.go
singularity_source
```

Options

```
-h, --help   help for create
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.47 singularity plugin disable

disable an installed Singularity plugin

Synopsis

The 'plugin disable' command allows a user to disable a plugin that is already installed in the system and which has been previously enabled.

```
singularity plugin disable <name>
```

Examples

```
$ singularity plugin disable example.org/plugin
```

Options

```
-h, --help   help for disable
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.48 singularity plugin enable

Enable an installed Singularity plugin

Synopsis

> The 'plugin enable' command allows a user to enable a plugin that is already installed in the system and which has been previously disabled.

```
singularity plugin enable <name>
```

Examples

```
$ singularity plugin enable example.org/plugin
```

Options

```
-h, --help   help for enable
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.49 singularity plugin inspect

Inspect a singularity plugin (either an installed one or an image)

Synopsis

The 'plugin inspect' command allows a user to inspect a plugin that is already installed in the system or an image containing a plugin that is yet to be installed.

```
singularity plugin inspect (<name>|<image>)
```

Examples

```
$ singularity plugin inspect sylabs.io/test-plugin
Name: sylabs.io/test-plugin
Description: A test Singularity plugin.
Author: Sylabs
Version: 0.1.0
```

Options

```
-h, --help   help for inspect
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.50 singularity plugin install

Install a compiled Singularity plugin

Synopsis

The 'plugin install' command installs the compiled plugin found at plugin_path into the appropriate directory on the host.

```
singularity plugin install <plugin_path>
```

Examples

```
$ singularity plugin install $HOME/singularity/test-plugin/test-plugin.sif
```

Options

```
-h, --help   help for install
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.51 singularity plugin list

List installed Singularity plugins

Synopsis

The 'plugin list' command lists the Singularity plugins installed on the host.

```
singularity plugin list [list options...]
```

Examples

```
$ singularity plugin list
ENABLED  NAME
    yes  example.org/plugin
```

Options

```
-h, --help   help for list
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.52 singularity plugin uninstall

Uninstall removes the named plugin from the system

Synopsis

The 'plugin uninstall' command removes the named plugin from the system

```
singularity plugin uninstall <name>
```

Examples

```
$ singularity plugin uninstall example.org/plugin
```

Options

```
-h, --help   help for uninstall
```

SEE ALSO

- *singularity plugin* - Manage Singularity plugins

Auto generated by spf13/cobra on 13-Oct-2020

7.2.53 singularity pull

Pull an image from a URI

Synopsis

The 'pull' command allows you to download or build a container from a given URI. Supported URIs include:

library: Pull an image from the currently configured library library://user/collection/container[:tag]

docker: Pull an image from Docker Hub docker://user/image:tag

shub: Pull an image from Singularity Hub shub://user/image:tag

oras: Pull a SIF image from a supporting OCI registry oras://registry/namespace/image:tag

http, https: Pull an image using the http(s?) protocol https://library.sylabs.io/v1/imagefile/library/default/alpine:latest

```
singularity pull [pull options...] [output file] <URI>
```

Examples

```
From Sylabs cloud library
$ singularity pull alpine.sif library://alpine:latest

From Docker
$ singularity pull tensorflow.sif docker://tensorflow/tensorflow:latest

From Shub
$ singularity pull singularity-images.sif shub://vsoch/singularity-images

From supporting OCI registry (e.g. Azure Container Registry)
$ singularity pull image.sif oras://<username>.azurecr.io/namespace/image:tag
```

Options

```
    --arch string      architecture to pull from library (default "amd64")
    --dir string       download images to the specific directory
    --disable-cache    dont use cached images/blobs and dont create them
    --docker-login     login to a Docker Repository interactively
-F, --force            overwrite an image file if it exists
-h, --help             help for pull
    --library string   download images from the provided library (default "https://
↪library.sylabs.io")
    --no-cleanup       do NOT clean up bundle after failed build, can be helpful for␣
↪debugging
    --nohttps          do NOT use HTTPS with the docker:// transport (useful for␣
↪local docker registries without a certificate)
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.54 singularity push

Upload image to the provided URI

Synopsis

The ‘push’ command allows you to upload a SIF container to a given URI. Supported URIs include:

library: library://user/collection/container[:tag]

oras: oras://registry/namespace/repo:tag

NOTE: It’s always good practice to sign your containers before pushing them to the library. An auth token is required to push to the library, so you may need to configure it first with ‘singularity remote’.

```
singularity push [push options...] <image> <URI>
```

Examples

```
To Library
$ singularity push /home/user/my.sif library://user/collection/my.sif:latest

To supported OCI registry
$ singularity push /home/user/my.sif oras://registry/namespace/image:tag
```

Options

```
-U, --allow-unsigned   do not require a signed container
-h, --help             help for push
    --library string   the library to push to (default "https://library.sylabs.io")
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.55 singularity remote

Manage singularity remote endpoints

Synopsis

The 'remote' commands allow you to manage Singularity remote endpoints through its subcommands. These allow you to add, log in, and use endpoints. The remote configuration is stored in $HOME/.singularity/remotes.yaml by default.

Examples

```
All group commands have their own help output:

    $ singularity help remote list
    $ singularity remote list
```

Options

```
-c, --config string   path to the file holding remote endpoint configurations␣
↪(default "/home/dave/.singularity/remote.yaml")
-h, --help            help for remote
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity remote add* - Create a new singularity remote endpoint * *singularity remote list* - List all singularity remote endpoints that are configured * *singularity remote login* - Log into a singularity remote endpoint using an authentication token * *singularity remote remove* - Remove an existing singularity remote endpoint * *singularity remote status* - Check the status of the singularity services at an endpoint * *singularity remote use* - Set a singularity remote endpoint to be actively used

Auto generated by spf13/cobra on 13-Oct-2020

7.2.56 singularity remote add

Create a new singularity remote endpoint

Synopsis

> The 'remote add' command allows you to create a new remote endpoint to be be used for singularity remote services. Authentication with a newly created endpoint will occur automatically.

```
singularity remote add [add options...] <remote_name> <remote_URI>
```

Examples

```
$ singularity remote add SylabsCloud cloud.sylabs.io
```

Options

```
-g, --global             edit the list of globally configured remote endpoints
-h, --help               help for add
    --no-login           skip automatic login step
    --tokenfile string   path to the file holding token
```

SEE ALSO

- *singularity remote* - Manage singularity remote endpoints

Auto generated by spf13/cobra on 13-Oct-2020

7.2.57 singularity remote list

List all singularity remote endpoints that are configured

Synopsis

The 'remote list' command lists all remote endpoints configured for use. If a remote is in use, its name will be encompassed by brackets.

```
singularity remote list
```

Examples

```
$ singularity remote list
```

Options

```
-h, --help   help for list
```

SEE ALSO

- *singularity remote* - Manage singularity remote endpoints

Auto generated by spf13/cobra on 13-Oct-2020

7.2.58 singularity remote login

Log into a singularity remote endpoint using an authentication token

Synopsis

The 'remote login' command allows you to set an authentication token for a specific endpoint. This command will produce a link directing you to the token service you can use to generate a valid token. If no endpoint is specified, it will try the default remote (SylabsCloud).

```
singularity remote login [login options...] <remote_name>
```

Examples

```
$ singularity remote login SylabsCloud
```

Options

```
-h, --help                help for login
    --tokenfile string    path to the file holding token
```

SEE ALSO

- *singularity remote* - Manage singularity remote endpoints

Auto generated by spf13/cobra on 13-Oct-2020

7.2.59 singularity remote remove

Remove an existing singularity remote endpoint

Synopsis

The 'remote remove' command allows you to remove an existing remote endpoint from the list of potential endpoints to use.

```
singularity remote remove [remove options...] <remote_name>
```

Examples

```
$ singularity remote remove SylabsCloud
```

Options

```
-g, --global   edit the list of globally configured remote endpoints
-h, --help     help for remove
```

SEE ALSO

- *singularity remote* - Manage singularity remote endpoints

Auto generated by spf13/cobra on 13-Oct-2020

7.2.60 singularity remote status

Check the status of the singularity services at an endpoint

Synopsis

The 'remote status' command checks the status of the specified remote endpoint and reports the availibility of services and their versions. If no endpoint is specified, it will check the status of the default remote (SylabsCloud).

```
singularity remote status [remote_name]
```

Examples

```
$ singularity remote status SylabsCloud
```

Options

```
-h, --help   help for status
```

SEE ALSO

- *singularity remote* - Manage singularity remote endpoints

Auto generated by spf13/cobra on 13-Oct-2020

7.2.61 singularity remote use

Set a singularity remote endpoint to be actively used

Synopsis

The 'remote use' command sets the remote to be used by default by any command that interacts with Singularity services.

```
singularity remote use [use options...] <remote_name>
```

Examples

```
$ singularity remote use SylabsCloud
```

Options

```
-g, --global   edit the list of globally configured remote endpoints
-h, --help     help for use
```

SEE ALSO

- *singularity remote* - Manage singularity remote endpoints

Auto generated by spf13/cobra on 13-Oct-2020

7.2.62 singularity run

Run the user-defined default command within a container

Synopsis

This command will launch a Singularity container and execute a runscript if one is defined for that container. The runscript is a metadata file within the container that contains shell commands. If the file is present (and executable) then this command will execute that file within the container automatically. All arguments following the container name will be passed directly to the runscript.

singularity run accepts the following container formats:

*.sif Singularity Image Format (SIF). Native to Singularity 3.0+

*.sqsh SquashFS format. Native to Singularity 2.4+

*.img ext3 format. Native to Singularity versions < 2.4.

directory/ sandbox format. Directory containing a valid root file system and optionally Singularity meta-data.

instance://* A local running instance of a container. (See the instance command group.)

library://* A container hosted on a Library (default https://cloud.sylabs.io/library)

docker://* A container hosted on Docker Hub

shub://* A container hosted on Singularity Hub

oras://* A container hosted on a supporting OCI registry

```
singularity run [run options...] <container>
```

Examples

```
# Here we see that the runscript prints "Hello world: "
$ singularity exec /tmp/debian.sif cat /singularity
#!/bin/sh
echo "Hello world: "

# It runs with our inputs when we run the image
$ singularity run /tmp/debian.sif one two three
Hello world: one two three
```

(continues on next page)

(continued from previous page)

```
# Note that this does the same thing
$ ./tmp/debian.sif one two three
```

Options

```
    --add-caps string        a comma separated capability list to add
    --allow-setuid           allow setuid binaries in container (root only)
    --app string             set an application to run inside a container
    --apply-cgroups string   apply cgroups from file for container processes (root␣
↪only)
-B, --bind strings           a user-bind path specification.  spec has the format␣
↪src[:dest[:opts]], where src and dest are outside and inside paths.  If dest is not␣
↪given, it is set equal to src.  Mount options ('opts') may be specified as 'ro'␣
↪(read-only) or 'rw' (read/write, which is the default). Multiple bind paths can be␣
↪given by a comma separated list.
-e, --cleanenv               clean environment before running container
-c, --contain                use minimal /dev and empty other directories (e.g. /tmp␣
↪and $HOME) instead of sharing filesystems from your host
-C, --containall             contain not only file systems, but also PID, IPC, and␣
↪environment
    --disable-cache          dont use cache, and dont create cache
    --dns string             list of DNS server separated by commas to add in resolv.
↪conf
    --docker-login           login to a Docker Repository interactively
    --drop-caps string       a comma separated capability list to drop
    --env strings            pass environment variable to contained process
    --env-file string        pass environment variables from file to contained process
-f, --fakeroot               run container in new user namespace as uid 0
    --fusemount strings      A FUSE filesystem mount specification of the form '<type>
↪:<fuse command> <mountpoint>' - where <type> is 'container' or 'host', specifying␣
↪where the mount will be performed ('container-daemon' or 'host-daemon' will run the␣
↪FUSE process detached). <fuse command> is the path to the FUSE executable, plus␣
↪options for the mount. <mountpoint> is the location in the container to which the␣
↪FUSE mount will be attached. E.g. 'container:sshfs 10.0.0.1:/ /sshfs'. Implies --
↪pid.
-h, --help                   help for run
-H, --home string            a home directory specification.  spec can either be a␣
↪src path or src:dest pair.  src is the source path of the home directory outside␣
↪the container and dest overrides the home directory within the container. (default
↪"/home/dave")
    --hostname string        set container hostname
-i, --ipc                    run container in a new IPC namespace
    --keep-privs             let root user keep privileges in container (root only)
-n, --net                    run container in a new network namespace (sets up a␣
↪bridge network interface by default)
    --network string         specify desired network type separated by commas, each␣
↪network will bring up a dedicated interface inside container (default "bridge")
    --network-args strings   specify network arguments to pass to CNI plugins
    --no-home                do NOT mount users home directory if /home is not the␣
↪current working directory
    --no-init                do NOT start shim process with --pid
    --no-privs               drop all privileges from root user in container)
    --nohttps                do NOT use HTTPS with the docker:// transport (useful␣
↪for local docker registries without a certificate)
```

(continues on next page)

(continued from previous page)

```
    --nonet                      disable VM network handling
    --nv                         enable experimental Nvidia support
-o, --overlay strings            use an overlayFS image for persistent data storage or as␣
↪read-only layer of container
    --passphrase                 prompt for an encryption passphrase
    --pem-path string            enter an path to a PEM formated RSA key for an encrypted␣
↪container
-p, --pid                        run container in a new PID namespace
    --pwd string                 initial working directory for payload process inside the␣
↪container
    --rocm                       enable experimental Rocm support
-S, --scratch strings            include a scratch directory within the container that is␣
↪linked to a temporary dir (use -W to force location)
    --security strings           enable security features (SELinux, Apparmor, Seccomp)
-u, --userns                     run container in a new user namespace, allowing␣
↪Singularity to run completely unprivileged on recent kernels. This disables some␣
↪features of Singularity, for example it only works with sandbox images.
    --uts                        run container in a new UTS namespace
    --vm                         enable VM support
    --vm-cpu string              number of CPU cores to allocate to Virtual Machine␣
↪(implies --vm) (default "1")
    --vm-err                     enable attaching stderr from VM
    --vm-ip string               IP Address to assign for container usage. Defaults to␣
↪DHCP within bridge network. (default "dhcp")
    --vm-ram string              amount of RAM in MiB to allocate to Virtual Machine␣
↪(implies --vm) (default "1024")
-W, --workdir string             working directory to be used for /tmp, /var/tmp and
↪$HOME (if -c/--contain was also used)
-w, --writable                   by default all Singularity containers are available as␣
↪read only. This option makes the file system accessible as read/write.
    --writable-tmpfs             makes the file system accessible as read-write with non␣
↪persistent data (with overlay support only)
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.63 singularity run-help

Show the user-defined help for an image

Synopsis

The help text is from the '%help' section of the definition file. If you are using the '–apps' option, the help text is instead from that app's '%apphelp' section.

```
singularity run-help <image path>
```

Examples

```
$ cat my_container.def
Bootstrap: docker
From: busybox

%help
    Some help for this container

%apphelp foo
    Some help for application 'foo' in this container

$ sudo singularity build my_container.sif my_container.def
Using container recipe deffile: my_container.def
[...snip...]
Cleaning up...

$ singularity run-help my_container.sif

  Some help for this container

$ singularity run-help --app foo my_container.sif

  Some help for application in this container
```

Options

```
    --app string   show the help for an app
-h, --help         help for run-help
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.64 singularity search

Search a Container Library for images

Synopsis

Search a Container Library for users and containers matching the search query. (default cloud.sylabs.io)

```
singularity search [search options...] <search_query>
```

Examples

```
$ singularity search lolcow
$ singularity search centos
```

Options

```
-h, --help              help for search
    --library string    URI for library to search (default "https://library.sylabs.io")
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.65 singularity shell

Run a shell within a container

Synopsis

singularity shell supports the following formats:

*.sif Singularity Image Format (SIF). Native to Singularity 3.0+

*.sqsh SquashFS format. Native to Singularity 2.4+

*.img ext3 format. Native to Singularity versions < 2.4.

directory/ sandbox format. Directory containing a valid root file system and optionally Singularity meta-data.

instance://* A local running instance of a container. (See the instance command group.)

library://* A container hosted on a Library (default https://cloud.sylabs.io/library) docker://* A container hosted on Docker Hub

shub://* A container hosted on Singularity Hub

oras://* A container hosted on a supporting OCI registry

```
singularity shell [shell options...] <container>
```

Examples

```
$ singularity shell /tmp/Debian.sif
Singularity/Debian.sif> pwd
/home/gmk/test
Singularity/Debian.sif> exit

$ singularity shell -C /tmp/Debian.sif
Singularity/Debian.sif> pwd
/home/gmk
Singularity/Debian.sif> ls -l
total 0
Singularity/Debian.sif> exit

$ sudo singularity shell -w /tmp/Debian.sif
$ sudo singularity shell --writable /tmp/Debian.sif

$ singularity shell instance://my_instance

$ singularity shell instance://my_instance
Singularity: Invoking an interactive shell within container...
Singularity container:~> ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
ubuntu       1     0  0 20:00 ?        00:00:00 /usr/local/bin/singularity/bin/sinit
ubuntu       2     0  0 20:01 pts/8    00:00:00 /bin/bash --norc
ubuntu       3     2  0 20:02 pts/8    00:00:00 ps -ef
```

Options

```
    --add-caps string        a comma separated capability list to add
    --allow-setuid           allow setuid binaries in container (root only)
    --app string             set an application to run inside a container
    --apply-cgroups string   apply cgroups from file for container processes (root␣
↪only)
-B, --bind strings           a user-bind path specification.  spec has the format␣
↪src[:dest[:opts]], where src and dest are outside and inside paths.  If dest is not␣
↪given, it is set equal to src.  Mount options ('opts') may be specified as 'ro'␣
↪(read-only) or 'rw' (read/write, which is the default). Multiple bind paths can be␣
↪given by a comma separated list.
-e, --cleanenv               clean environment before running container
-c, --contain                use minimal /dev and empty other directories (e.g. /tmp␣
↪and $HOME) instead of sharing filesystems from your host
-C, --containall             contain not only file systems, but also PID, IPC, and␣
↪environment
    --disable-cache          dont use cache, and dont create cache
    --dns string             list of DNS server separated by commas to add in resolv.
↪conf
```

(continues on next page)

(continued from previous page)

```
    --docker-login           login to a Docker Repository interactively
    --drop-caps string       a comma separated capability list to drop
    --env strings            pass environment variable to contained process
    --env-file string        pass environment variables from file to contained process
-f, --fakeroot               run container in new user namespace as uid 0
    --fusemount strings      A FUSE filesystem mount specification of the form '<type>
↪:<fuse command> <mountpoint>' - where <type> is 'container' or 'host', specifying␣
↪where the mount will be performed ('container-daemon' or 'host-daemon' will run the␣
↪FUSE process detached). <fuse command> is the path to the FUSE executable, plus␣
↪options for the mount. <mountpoint> is the location in the container to which the␣
↪FUSE mount will be attached. E.g. 'container:sshfs 10.0.0.1:/ /sshfs'. Implies --
↪pid.
-h, --help                   help for shell
-H, --home string            a home directory specification.  spec can either be a␣
↪src path or src:dest pair.  src is the source path of the home directory outside␣
↪the container and dest overrides the home directory within the container. (default
↪"/home/dave")
    --hostname string        set container hostname
-i, --ipc                    run container in a new IPC namespace
    --keep-privs             let root user keep privileges in container (root only)
-n, --net                    run container in a new network namespace (sets up a␣
↪bridge network interface by default)
    --network string         specify desired network type separated by commas, each␣
↪network will bring up a dedicated interface inside container (default "bridge")
    --network-args strings   specify network arguments to pass to CNI plugins
    --no-home                do NOT mount users home directory if /home is not the␣
↪current working directory
    --no-init                do NOT start shim process with --pid
    --no-privs               drop all privileges from root user in container)
    --nohttps                do NOT use HTTPS with the docker:// transport (useful␣
↪for local docker registries without a certificate)
    --nonet                  disable VM network handling
    --nv                     enable experimental Nvidia support
-o, --overlay strings        use an overlayFS image for persistent data storage or as␣
↪read-only layer of container
    --passphrase             prompt for an encryption passphrase
    --pem-path string        enter an path to a PEM formated RSA key for an encrypted␣
↪container
-p, --pid                    run container in a new PID namespace
    --pwd string             initial working directory for payload process inside the␣
↪container
    --rocm                   enable experimental Rocm support
-S, --scratch strings        include a scratch directory within the container that is␣
↪linked to a temporary dir (use -W to force location)
    --security strings       enable security features (SELinux, Apparmor, Seccomp)
-s, --shell string           path to program to use for interactive shell
    --syos                   execute SyOS shell
-u, --userns                 run container in a new user namespace, allowing␣
↪Singularity to run completely unprivileged on recent kernels. This disables some␣
↪features of Singularity, for example it only works with sandbox images.
    --uts                    run container in a new UTS namespace
    --vm                     enable VM support
    --vm-cpu string          number of CPU cores to allocate to Virtual Machine␣
↪(implies --vm) (default "1")
    --vm-err                 enable attaching stderr from VM
    --vm-ip string           IP Address to assign for container usage. Defaults to␣
↪DHCP within bridge network. (default "dhcp")
```

(continues on next page)

(continued from previous page)

```
    --vm-ram string          amount of RAM in MiB to allocate to Virtual Machine␣
↪(implies --vm) (default "1024")
-W, --workdir string         working directory to be used for /tmp, /var/tmp and
↪$HOME (if -c/--contain was also used)
-w, --writable               by default all Singularity containers are available as␣
↪read only. This option makes the file system accessible as read/write.
    --writable-tmpfs         makes the file system accessible as read-write with non␣
↪persistent data (with overlay support only)
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.66 singularity sif

siftool is a program for Singularity Image Format (SIF) file manipulation

Synopsis

A set of commands are provided to display elements such as the SIF global header, the data object descriptors and to dump data objects. It is also possible to modify a SIF file via this tool via the add/del commands.

Options

```
-h, --help   help for sif
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC) * *singularity sif add* - Add a data object to a SIF file * *singularity sif del* - Delete a specified object descriptor and data from SIF file * *singularity sif dump* - Extract and output data objects from SIF files * *singularity sif header* - Display SIF global headers * *singularity sif info* - Display detailed information of object descriptors * *singularity sif list* - List object descriptors from SIF files * *singularity sif new* - Create a new empty SIF image file * *singularity sif setprim* - Set primary system partition

Auto generated by spf13/cobra on 13-Oct-2020

7.2.67 singularity sif add

Add a data object to a SIF file

Synopsis

Add a data object to a SIF file

```
singularity sif add [OPTIONS] <containerfile> <dataobjectfile> [flags]
```

Options

```
    --alignment int       set alignment constraint [default: aligned on page size]
    --datatype int        the type of data to add
                          [NEEDED, no default]:
                            1-Deffile,   2-EnvVar,    3-Labels,
                            4-Partition, 5-Signature, 6-GenericJSON
    --filename string     set logical filename/handle [default: input filename]
    --groupid int         set groupid [default: DescrUnusedGroup]
-h, --help                help for add
    --link int            set link pointer [default: DescrUnusedLink]
    --partarch int        the main architecture used (with -datatype 4-Partition)
                          [NEEDED, no default]:
                            1-386,       2-amd64,     3-arm,
                            4-arm64,     5-ppc64,     6-ppc64le,
                            7-mips,      8-mipsle,    9-mips64,
                            10-mips64le, 11-s390x
    --partfs int          the filesystem used (with -datatype 4-Partition)
                          [NEEDED, no default]:
                            1-Squash,    2-Ext3,      3-ImmuObj,
                            4-Raw
    --parttype int        the type of partition (with -datatype 4-Partition)
                          [NEEDED, no default]:
                            1-System,    2-PrimSys,   3-Data,
                            4-Overlay
    --signentity string   the entity that signs (with -datatype 5-Signature)
                          [NEEDED, no default]:
                            example: 433FE984155206BD962725E20E8713472A879943
    --signhash int        the signature hash used (with -datatype 5-Signature)
                          [NEEDED, no default]:
                            1-SHA256,    2-SHA384,    3-SHA512,
                            4-BLAKE2S,   5-BLAKE2B
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.68 singularity sif del

Delete a specified object descriptor and data from SIF file

Synopsis

Delete a specified object descriptor and data from SIF file

```
singularity sif del <descriptorid> <containerfile> [flags]
```

Options

```
-h, --help   help for del
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.69 singularity sif dump

Extract and output data objects from SIF files

Synopsis

Extract and output data objects from SIF files

```
singularity sif dump <descriptorid> <containerfile>
```

Options

```
-h, --help   help for dump
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.70 singularity sif header

Display SIF global headers

Synopsis

Display SIF global headers

```
singularity sif header <containerfile>
```

Options

```
-h, --help   help for header
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.71 singularity sif info

Display detailed information of object descriptors

Synopsis

Display detailed information of object descriptors

```
singularity sif info <descriptorid> <containerfile>
```

Options

```
-h, --help   help for info
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.72 singularity sif list

List object descriptors from SIF files

Synopsis

List object descriptors from SIF files

```
singularity sif list <containerfile>
```

Options

```
-h, --help   help for list
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.73 singularity sif new

Create a new empty SIF image file

Synopsis

Create a new empty SIF image file

```
singularity sif new <containerfile>
```

Options

```
-h, --help   help for new
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.74 singularity sif setprim

Set primary system partition

Synopsis

Set primary system partition

```
singularity sif setprim <descriptorid> <containerfile> [flags]
```

Options

```
-h, --help   help for setprim
```

SEE ALSO

- *singularity sif* - siftool is a program for Singularity Image Format (SIF) file manipulation

Auto generated by spf13/cobra on 13-Oct-2020

7.2.75 singularity sign

Attach digital signature(s) to an image

Synopsis

> The sign command allows a user to add one or more digital signatures to a SIF image. By default, one digital signature is added for each object group in the file.
>
> To generate a keypair, see 'singularity help key newpair'

```
singularity sign [sign options...] <image path>
```

Examples

```
$ singularity sign container.sif
```

Options

```
-g, --group-id uint32   sign objects with the specified group ID
-h, --help              help for sign
-k, --keyidx int        private key to use (index from 'key list')
-i, --sif-id uint32     sign object with the specified ID
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.76 singularity test

Run the user-defined tests within a container

Synopsis

The 'test' command allows you to execute a testscript (if available) inside of a given container

NOTE: For instances if there is a daemon process running inside the container, then subsequent container commands will all run within the same namespaces. This means that the –writable and –contain options will not be honored as the namespaces have already been configured by the 'singularity start' command.

```
singularity test [exec options...] <image path>
```

Examples

```
Set the '%test' section with a definition file like so:
%test
    echo "hello from test" "$@"

$ singularity test /tmp/debian.sif command
    hello from test command

For additional help, please visit our public documentation pages which are
found at:

    https://www.sylabs.io/docs/
```

Options

```
    --add-caps string        a comma separated capability list to add
    --allow-setuid           allow setuid binaries in container (root only)
    --app string             set an application to run inside a container
    --apply-cgroups string   apply cgroups from file for container processes (root␣
↪only)
-B, --bind strings           a user-bind path specification.  spec has the format␣
↪src[:dest[:opts]], where src and dest are outside and inside paths.  If dest is not␣
↪given, it is set equal to src.  Mount options ('opts') may be specified as 'ro'␣
↪(read-only) or 'rw' (read/write, which is the default). Multiple bind paths can be␣
↪given by a comma separated list.
-e, --cleanenv               clean environment before running container
-c, --contain                use minimal /dev and empty other directories (e.g. /tmp␣
↪and $HOME) instead of sharing filesystems from your host
```

(continues on next page)

(continued from previous page)

```
-C, --containall              contain not only file systems, but also PID, IPC, and␣
↪environment
    --disable-cache           dont use cache, and dont create cache
    --dns string              list of DNS server separated by commas to add in resolv.
↪conf
    --docker-login            login to a Docker Repository interactively
    --drop-caps string        a comma separated capability list to drop
    --env strings             pass environment variable to contained process
    --env-file string         pass environment variables from file to contained process
-f, --fakeroot                run container in new user namespace as uid 0
    --fusemount strings       A FUSE filesystem mount specification of the form '<type>
↪:<fuse command> <mountpoint>' - where <type> is 'container' or 'host', specifying␣
↪where the mount will be performed ('container-daemon' or 'host-daemon' will run the␣
↪FUSE process detached). <fuse command> is the path to the FUSE executable, plus␣
↪options for the mount. <mountpoint> is the location in the container to which the␣
↪FUSE mount will be attached. E.g. 'container:sshfs 10.0.0.1:/ /sshfs'. Implies --
↪pid.
-h, --help                    help for test
-H, --home string             a home directory specification.  spec can either be a␣
↪src path or src:dest pair.  src is the source path of the home directory outside␣
↪the container and dest overrides the home directory within the container. (default
↪"/home/dave")
    --hostname string         set container hostname
-i, --ipc                     run container in a new IPC namespace
    --keep-privs              let root user keep privileges in container (root only)
-n, --net                     run container in a new network namespace (sets up a␣
↪bridge network interface by default)
    --network string          specify desired network type separated by commas, each␣
↪network will bring up a dedicated interface inside container (default "bridge")
    --network-args strings    specify network arguments to pass to CNI plugins
    --no-home                 do NOT mount users home directory if /home is not the␣
↪current working directory
    --no-init                 do NOT start shim process with --pid
    --no-privs                drop all privileges from root user in container)
    --nohttps                 do NOT use HTTPS with the docker:// transport (useful␣
↪for local docker registries without a certificate)
    --nonet                   disable VM network handling
    --nv                      enable experimental Nvidia support
-o, --overlay strings         use an overlayFS image for persistent data storage or as␣
↪read-only layer of container
    --passphrase              prompt for an encryption passphrase
    --pem-path string         enter an path to a PEM formated RSA key for an encrypted␣
↪container
-p, --pid                     run container in a new PID namespace
    --pwd string              initial working directory for payload process inside the␣
↪container
    --rocm                    enable experimental Rocm support
-S, --scratch strings         include a scratch directory within the container that is␣
↪linked to a temporary dir (use -W to force location)
    --security strings        enable security features (SELinux, Apparmor, Seccomp)
-u, --userns                  run container in a new user namespace, allowing␣
↪Singularity to run completely unprivileged on recent kernels. This disables some␣
↪features of Singularity, for example it only works with sandbox images.
    --uts                     run container in a new UTS namespace
    --vm                      enable VM support
    --vm-cpu string           number of CPU cores to allocate to Virtual Machine␣
↪(implies --vm) (default "1")
```

(continues on next page)

(continued from previous page)

```
    --vm-err                  enable attaching stderr from VM
    --vm-ip string            IP Address to assign for container usage. Defaults to␣
↪DHCP within bridge network. (default "dhcp")
    --vm-ram string           amount of RAM in MiB to allocate to Virtual Machine␣
↪(implies --vm) (default "1024")
-W, --workdir string          working directory to be used for /tmp, /var/tmp and
↪$HOME (if -c/--contain was also used)
-w, --writable                by default all Singularity containers are available as␣
↪read only. This option makes the file system accessible as read/write.
    --writable-tmpfs          makes the file system accessible as read-write with non␣
↪persistent data (with overlay support only)
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.77 singularity verify

Verify cryptographic signatures attached to an image

Synopsis

The verify command allows a user to verify cryptographic signatures on SIF container files. There may be multiple signatures for data objects and multiple data objects signed. By default the command searches for the primary partition signature. If found, a list of all verification blocks applied on the primary partition is gathered so that data integrity (hashing) and signature verification is done for all those blocks.

```
singularity verify [verify options...] <image path>
```

Examples

```
$ singularity verify container.sif
```

Options

```
-a, --all                  verify all objects
-g, --group-id uint32      verify objects with the specified group ID
-h, --help                 help for verify
-j, --json                 output json
    --legacy-insecure      enable verification of (insecure) legacy signatures
-l, --local                only verify with local key(s) in keyring
-i, --sif-id uint32        verify object with the specified ID
-u, --url string           specify a URL for a key server (default "https://keys.sylabs.
↪io")
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.2.78 singularity version

Show the version for Singularity

Synopsis

Show the version for Singularity

```
singularity version
```

Options

```
-h, --help   help for version
```

SEE ALSO

- *singularity* -

Linux container platform optimized for High Performance Computing (HPC) and Enterprise Performance Computing (EPC)

Auto generated by spf13/cobra on 13-Oct-2020

7.3 License

This documentation is subject to the following 3-clause BSD license:

(continues on next page)

(continued from previous page)