Singularity Container Documentation

Release 3.0

User Docs

Mar 25, 2019

CONTENTS

CHAPTER ONE

QUICK START

This guide is intended for running Singularity on a computer where you have root (administrative) privileges.

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

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

1.1 Quick Installation Steps

You will need a Linux system to run Singularity.

See the *installation page* for information about installing older versions of Singularity.

1.1.1 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
```

Note: Note that `squashfs-tools` is an image build dependency only and is not required for Singularity `build` and `run` commands.

1.1.2 Install Go

Singularity 3.0 is written primarily in Go, and you will need Go installed to compile it from source.

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

First, visit the Go download page and pick the appropriate Go archive (>=1.11.1). Copy the link address and download with `wget` like so:

```
$ export VERSION=1.11 OS=linux ARCH=amd64
$ cd /tmp
$ wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz
```

Then extract the archive to `/usr/local`

```
$ sudo tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Finally, set up your environment for Go

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

1.1.3 Clone the Singularity repository

Go is a bit finicky about where things are placed. Here is the correct way to build Singularity from source.

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

1.1.4 Install Go dependencies

Dependencies are managed using Dep. You can use go get to install it like so:

```
$ go get -u -v github.com/golang/dep/cmd/dep
```

1.1.5 Compile the Singularity binary

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

```
$ cd $GOPATH/src/github.com/sylabs/singularity
$ ./mconfig
$ make -C builddir
$ sudo make -C builddir install
```

Singularity must be installed as root to function properly.

1.2 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:
```

(continues on next page)

(continued from previous page)

```
  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
  -q, --quiet              suppress normal output
  -s, --silent             only print errors
  -t, --tokenfile string   path to the file holding your sylabs
                           authentication token (default
                           "/home/david/.singularity/sylabs-token")
  -v, --verbose            print additional information

Available Commands:
  build       Build a new Singularity container
  capability  Manage Linux capabilities on containers
  exec        Execute a command within container
  help        Help about any command
  inspect     Display metadata for container if available
  instance    Manage containers running in the background
  keys        Manage OpenPGP key stores
  pull        Pull a container from a URI
  push        Push a container to a Library URI
  run         Launch a runscript within container
  run-help    Display help for container if available
  search      Search the library
  shell       Run a Bourne shell within container
  sign        Attach cryptographic signatures to container
  test        Run defined tests for this particular container
  verify      Verify cryptographic signatures on container
  version     Show application version

Examples:
  $ singularity help <command>
      Additional help for any Singularity subcommand can be seen by appending
      the subcommand name to the above command.


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 on container

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
```

(continues on next page)

(continued from previous page)

```
  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:
  -g, --groupid uint32   group ID to be verified
  -h, --help             help for verify
  -i, --id uint32        descriptor ID to be verified
  -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 passed 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 `capabilities` command group like so:

```
$ singularity capability list --user 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 help container.sif  # See the container's help, if provided

$ singularity help --app foo container.sif  # See the help for foo, if provided
```

1.3 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'
```

(continues on next page)

(continued from previous page)

```
    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.

1.4 Interact with images

You can interact with images in several ways. It is not actually necessary to `pull` or `build` an image to interact with it. The commands listed here will work with image URIs in addition to accepting a local path to an image.

For these examples we will use a `lolcow_latest.sif` image that can be pulled from the Container Library like so.

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

1.4.1 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
```

1.4.2 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 |
                ||     ||
```

1.4.3 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 |
                ||     ||
```

1.4.4 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 |
                ||     ||
```

1.5 Build images from scratch

Singularity v3.0 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).

For more details about the different build options and best practices, read about the Singularity flow.

1.5.1 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
```

1.5.2 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.

1.5.3 Singularity Definition Files

For a reproducible, 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 do things like install software, setup the environment, and copy files into the container from the host system.

Here is an example of a definition file:

```
BootStrap: library
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

%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/

CHAPTER

TWO

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!

2.1 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!

2.1.1 Singularity Google Group

If you have been using Singularity and having good luck with it, join our Google Group and help out other users!

2.1.2 Singularity on Slack

Many of our users come to Slack for quick help with an issue. You can find us at singularity-container.

2.2 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

2.3 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.

2.4 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.

2.4.1 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/
```

2.4.2 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`.

2.4.3 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.

2.4.4 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
```

2.4.5 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.

2.4.6 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

THREE

INSTALLATION

This document will guide you through the process of installing Singularity >= 3.0.0 via several different methods. (For instructions on installing earlier versions of Singularity please see earlier versions of the docs.)

3.1 Overview

Singularity runs on Linux natively and can also be run on Windows and Mac through virtual machines (VMs). Here we cover several different methods of installing Singularity (>=v3.0.0) on Linux and also give methods for downloading and running VMs with singularity pre-installed from Vagrant Cloud.

3.2 Install on Linux

Linux is the only operating system that can support containers because of kernel features like namespaces. You can use these methods to install Singularity on bare metal Linux or a Linux VM.

3.2.1 Before you begin

If you have an earlier version of Singularity installed, you should *remove it* before executing the installation commands. You will also need to install some dependencies and install Go.

3.2.1.1 Install Dependencies

Install these dependencies with `apt-get` or `yum/rpm` as shown below or similar with other package managers.

`apt-get`

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

`yum`

```
$ sudo yum update -y && \
    sudo yum groupinstall -y 'Development Tools' && \
    sudo yum install -y \
    openssl-devel \
    libuuid-devel \
    libseccomp-devel \
    wget \
    squashfs-tools
```

3.2.1.2 Install Go

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

Visit the Go download page and pick a package archive to download. Copy the link address and download with wget. Then extract the archive to `/usr/local` (or use other instructions on go installation page).

```
$ export VERSION=1.11 OS=linux ARCH=amd64 && \
    wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz && \
    sudo tar -C /usr/local -xzvf go$VERSION.$OS-$ARCH.tar.gz && \
    rm go$VERSION.$OS-$ARCH.tar.gz
```

Then, set up your environment for Go.

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

If you are installing Singularity v3.0.0 you will also need to install `dep` for dependency resolution.

```
$ go get -u github.com/golang/dep/cmd/dep
```

3.2.2 Install from source

The following commands will install Singularity from the GitHub repo to `/usr/local`. This method will work for >=v3.0.0. To install an older tagged release see older versions of the docs.

When installing from source, you can decide to install from either a **tag**, a **release branch**, or from the **master branch**.

- **tag**: GitHub tags form the basis for releases, so installing from a tag is the same as downloading and installing a specific release. Tags are expected to be relatively stable and well-tested.
- **release branch**: A release branch represents the latest version of a minor release with all the newest bug fixes and enhancements (even those that have not yet made it into a point release). For instance, to install v3.0 with the latest bug fixes and enhancements checkout `release-3.0`. Release branches may be less stable than code in a tagged point release.
- **master branch**: The `master` branch contains the latest, bleeding edge version of Singularity. This is the default branch when you clone the source code, so you don't have to check out any new branches to install it. The `master` branch changes quickly and may be unstable.

3.2.2.1 Download Singularity repo (and optionally check out a tag or branch)

To ensure that the Singularity source code is downloaded to the appropriate directory use these commands.

```
$ go get -d github.com/sylabs/singularity
```

Go will complain that there are no Go files, but it will still download the Singularity source code to the appropriate directory within the `$GOPATH`.

Now checkout the version of Singularity you want to install.

```
$ export VERSION=v3.0.3 # or another tag or branch if you like && \
    cd $GOPATH/src/github.com/sylabs/singularity && \
    git fetch && \
    git checkout $VERSION # omit this command to install the latest bleeding edge␣
↪code from master
```

3.2.2.2 Download and install Singularity from a release

You can also install Singularity from one of our releases. For this, you can simply download a release from <https://github.com/sylabs/singularity/releases>`_. After that you can just run the following commands to proceed with the installation.

Note: Make sure to update the release version before running the following commands.

```
$ export VERSION=3.0.3 && # adjust this as necessary \
    mkdir -p $GOPATH/src/github.com/sylabs && \
    cd $GOPATH/src/github.com/sylabs && \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/
↪singularity-${VERSION}.tar.gz && \
    tar -xzf singularity-${VERSION}.tar.gz && \
    cd ./singularity && \
    ./mconfig
```

3.2.2.3 Compile Singularity

Singularity uses a custom build system called `makeit`. `mconfig` is called to generate a `Makefile` and then `make` is used to compile and install.

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

By default Singularity will be installed in the `/usr/local` directory hierarchy. You can specify a custom directory with the `--prefix` option, to `mconfig` like so:

```
$ ./mconfig --prefix=/opt/singularity
```

This option can be useful if you want to install multiple versions of Singularity, install a personal version of Singularity on a shared system, or if you want to remove Singularity easily after installing it.

For a full list of `mconfig` options, run `mconfig --help`. Here are some of the most common options that you may need to use when building Singularity from source.

- `--sysconfdir`: Install read-only config files in sysconfdir. This option is important if you need the `singularity.conf` file or other configuration files in a custom location.

- `--localstatedir`: Set the state directory where containers are mounted. This is a particularly important option for administrators installing Singularity on a shared file system. The `--localstatedir` should be set to a directory that is present on each individual node.
- `-b`: Build Singularity in a given directory. By default this is `./builddir`.

3.2.2.4 Source bash completion file

To enjoy bash completion with Singularity commands and options, source the bash completion file like so. Add this command to your *~/.bashrc* file so that bash completion continues to work in new shells. (Obviously adjust this path if you installed the bash completion file in a different location.)

```
$ . /usr/local/etc/bash_completion.d/singularity
```

3.2.3 Build and install an RPM

Building and installing a Singularty RPM allows the installation be more easily managed, upgraded and removed. In Singularity >=v3.0.1 you can build an RPM directly from the release tarball.

Note: Be sure to download the correct asset from the GitHub releases page. It should be named *singularity-<version>.tar.gz*.

After installing the *dependencies* and installing *Go* as detailed above, you are ready download the tarball and build and install the RPM.

```
$ export VERSION=3.0.3 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/
↪singularity-${VERSION}.tar.gz && \
    rpmbuild -tb singularity-${VERSION}.tar.gz && \
    sudo rpm -ivh ~/rpmbuild/RPMS/x86_64/singularity-$VERSION-1.el7.x86_64.rpm && \
    rm -rf ~/rpmbuild singularity-$VERSION*.tar.gz
```

Options to `mconfig` can be passed using the familiar syntax to `rpmbuild`. For example, if you want to force the local state directory to `/mnt` (instead of the default `/var`) you can do the following:

```
rpmbuild -tb --define='_localstatedir /mnt' singularity-$VERSION.tar.gz
```

Note: It is very important to set the local state directory to a directory that physically exists on nodes within a cluster when installing Singularity in an HPC environment with a shared file system. Thus the `_localstatedir` option should be of considerable interest to HPC admins.

3.2.4 Remove an old version

When you run `sudo make install`, the command lists files as they are installed. They must all be removed in order to completely remove Singularity.

For example, in a standard installation of Singularity 3.0.1 (when building from source) you must remove all of these files and directories to completely remove Singularity.

Obviously, this list of files may differ depending on how you install Singularity or with newer versions of Singularity released following the writing of this document.

```
$ sudo rm -rf \
    /usr/local/libexec/singularity \
    /usr/local/var/singularity \
    /usr/local/etc/singularity \
    /usr/local/bin/singularity \
    /usr/local/bin/run-singularity \
    /usr/local/etc/bash_completion.d/singularity
```

If you anticipate needing to remove Singularity, it might be easier to install it in a custom directory using the `--prefix` option to `mconfig`. In that case Singularity can be uninstalled simply by deleting the parent directory. Or it may be useful to install Singularity *using a package manager* so that it can be updated and/or uninstalled with ease in the future.

3.2.5 Distribution packages of Singularity

Note: Packaged versions of Singularity in Linux distribution repos are maintained by community members. They (necessarily) tend to be older releases of Singularity. For the latest upstream versions of Singularity it is recommended that you build from source using one of the methods detailed above.

3.2.5.1 Install the Debian/Ubuntu package using `apt`

Singularity is available on Debian and derivative distributions starting with Debian stretch and the Ubuntu 16.10 releases. The package is called `singularity-container`. For more recent releases of singularity and backports for older Debian and Ubuntu releases, it is recommended that you use the NeuroDebian repository.

Enable the NeuroDebian repository following instructions on the NeuroDebian site. Use the dropdown menus to find the best mirror for your operating system and location. For example, after selecting Ubuntu 16.04 and selecting a mirror in CA, you are instructed to add these lists:

```
$ sudo wget -O- http://neuro.debian.net/lists/xenial.us-ca.full | sudo tee /etc/apt/
↪sources.list.d/neurodebian.sources.list && \
    sudo apt-key adv --recv-keys --keyserver hkp://pool.sks-keyservers.net:80␣
↪0xA5D32F012649A5A9 && \
    sudo apt-get update
```

Now singularity can be installed like so:

```
sudo apt-get install -y singularity-container
```

During the above, if you have a previously installed configuration, you might be asked if you want to define a custom configuration/init, or just use the default provided by the package, eg:

```
Configuration file '/etc/singularity/init'

  ==> File on system created by you or by a script.
  ==> File also in package provided by package maintainer.
    What would you like to do about it ?  Your options are:
      Y or I  : install the package maintainer's version
      N or O  : keep your currently-installed version
        D     : show the differences between the versions
        Z     : start a shell to examine the situation
The default action is to keep your current version.
```

(continues on next page)

(continued from previous page)

```
*** init (Y/I/N/O/D/Z) [default=N] ? Y

Configuration file '/etc/singularity/singularity.conf'
  ==> File on system created by you or by a script.
  ==> File also in package provided by package maintainer.
    What would you like to do about it ?  Your options are:
      Y or I  : install the package maintainer's version
      N or O  : keep your currently-installed version
        D     : show the differences between the versions
        Z     : start a shell to examine the situation
The default action is to keep your current version.
*** singularity.conf (Y/I/N/O/D/Z) [default=N] ? Y
```

Most users should accept these defaults. For cluster admins, we recommend that you read the admin docs to get a better understanding of the configuration file options available to you.

After following this procedure, you can check the Singularity version like so:

```
$ singularity --version
    2.5.2-dist
```

If you need a backport build of the recent release of Singularity on those or older releases of Debian and Ubuntu, you can see all the various builds and other information here.

3.2.5.2 Install the CentOS/RHEL package using `yum`

The epel (Extra Packages for Enterprise Linux) repos contain Singularity. The singularity package is actually split into two packages called `singularity-runtime` (which simply contains the necessary bits to run singularity containers) and `singularity` (which also gives you the ability to build Singularity containers).

To install Singularity from the epel repos, first install the repos and then install Singularity. For instance, on CentOS6/7 do the following:

```
$ sudo yum update -y && \
    sudo yum install -y epel-release && \
    sudo yum update -y && \
    sudo yum install -y singularity-runtime singularity
```

After following this procedure, you can check the Singularity version like so:

```
$ singularity --version
    2.6.0-dist
```

3.3 Install on Windows or Mac

Linux containers like Singularity cannot run natively on Windows or Mac because of basic incompatibilities with the host kernel. (Contrary to a popular misconception, Mac does not run on a Linux kernel. It runs on a kernel called Darwin originally forked from BSD.)

For this reason, the Singularity community maintains a set of Vagrant Boxes via Vagrant Cloud, one of Hashicorp's open source tools. The current versions can be found under the sylabs organization.

3.3.1 Setup

First, install the following software:

3.3.1.1 Windows

Install the following programs:

- Git for Windows
- VirtualBox for Windows
- Vagrant for Windows
- Vagrant Manager for Windows

3.3.1.2 Mac

You need to install several programs. This example uses Homebrew but you can also install these tools using the GUI.

First, optionally install Homebrew.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

Next, install Vagrant and the necessary bits (either using this method or by downloading and installing the tools manually).

```
$ brew cask install virtualbox && \
    brew cask install vagrant && \
    brew cask install vagrant-manager
```

3.3.2 Singularity Vagrant Box

Run GitBash (Windows) or open a terminal (Mac) and create and enter a directory to be used with your Vagrant VM.

```
$ mkdir vm-singularity && \
    cd vm-singularity
```

If you have already created and used this folder for another VM, you will need to destroy the VM and delete the Vagrantfile.

```
$ vagrant destroy && \
    rm Vagrantfile
```

Then issue the following commands to bring up the Virtual Machine. (Substitute a different value for the `$VM` variable if you like.)

```
$ export VM=sylabs/singularity-3.0-ubuntu-bionic64 && \
    vagrant init $VM && \
    vagrant up && \
    vagrant ssh
```

You can check the installed version of Singularity with the following:

```
vagrant@vagrant:~$ singularity version
3.0.3-1
```

Of course, you can also start with a plain OS Vagrant box as a base and then install Singularity using one of the above methods for Linux.

3.4 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
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/2.5.2/user-guide/introduction.html#security-and-
↪privilege-escalation

    - Installation:
    https://www.sylabs.io/guides/3.0/user-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 to get a human response (sylabs.io/contact). I can do
my best to facilitate this interaction if help is needed.
```

(continues on next page)

(continued from previous page)

```
Thank you kindly for considering this request!

Best,

User
```

As is stated in the sample message above, you can always reach out to us for additional questions or support.

CHAPTER

FOUR

COMMAND LINE INTERFACE

Below are links to the automatically generated CLI docs

CHAPTER

FIVE

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.

5.1 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.

5.2 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.simg library://sylabs-jms/testing/lolcow
```

The first argument (lolcow.simg) 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.

5.3 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
```

5.4 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/
```

5.5 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.

5.6 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
```

(continues on next page)

(continued from previous page)

```
%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`.

5.7 Build options

5.7.1 --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`.

5.7.2 --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.

5.7.3 --force

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

5.7.4 --json

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

5.7.5 --library

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

5.7.6 `--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.

5.7.7 `--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".)

5.7.8 `--sandbox`

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

5.7.9 `--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.

5.7.10 `--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.

5.8 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

CHAPTER

SIX

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.

6.1 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

6.2 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.

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:

- *library* (images hosted on the Container Library)
- *docker* (images hosted on Docker Hub)
- *shub* (images hosted on Singularity Hub)
- *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)
- *arch* (Arch Linux)
- *busybox* (BusyBox)
- *zypper* (zypper based systems such as Suse and OpenSuse)

6.3 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, the order of the sections in the def file is unimportant and 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

%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"
```

(continues on next page)

(continued from previous page)

```
    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

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

6.3.1 %setup

Commands in the `%setup` section are 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 priviledges. 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.

6.3.2 %files

The `%files` section allows you to copy files from your host system into the container with greater safety than using the `%setup` section. Each line is a `<source>` and `<destination>` pair, where the source is a path on your host system, and the destination is a path in the container. The `<destination>` specification can be omitted and will be assumed to be the same path as the `<source>` specification.

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

6.3.3 %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 (see below).

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` (see below) 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.

6.3.4 %post

Commands in the `%post` section are executed within the container after the base OS has been installed at build time. This is where you will 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. Note that variables set in `%post` take precedence over those set in the `%environment` section as explained above.

6.3.5 %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
```

6.3.6 %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)
```

6.3.7 %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
        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.
```

6.3.8 %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
```

The easiest way to see labels is to inspect the image:

```
$ singularity inspect my_container.sif

{
    "Author": "d@sylabs.io",
    "Version": "v0.0.1",
    "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*.

6.3.9 %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.
```

6.4 Apps

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)

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"

##############################
# foo
##############################

```

(continues on next page)

(continued from previous page)

```
%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.

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.

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
```

(continues on next page)

(continued from previous page)

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

6.5 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.

CHAPTER

SEVEN

BUILD ENVIRONMENT

7.1 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.

7.2 Cache Folders

To make downloading images for build and pull faster and less redundant, Singularity uses a caching strategy. By default, Singularity will create a set of folders in your `$HOME` directory for docker layers, Cloud library images, and metadata, respectively:

```
$HOME/.singularity/cache/library
$HOME/.singularity/cache/oci
$HOME/.singularity/cache/oci-tmp
```

If you want to cache in a different directory, set `SINGULARITY_CACHEDIR` to the desired path. By using the `-E` option with the `sudo` command, `SINGULARITY_CACHEDIR` will be passed along to root's environment and respected during the build. Remember that when you run commands as root images will be cached in root's home at `/root` and not your user's home.

7.3 Temporary Folders

Singularity uses a temporary directory to build the squashfs filesystem, and this temp space needs to be large enough to hold the entire resulting Singularity image. By default this happens in `/tmp` but the location can be configured by setting `SINGULARITY_TMPDIR` to the full path where you want the sandbox and squashfs temp files to be stored. Remember to use `-E` option to pass the value of `SINGULARITY_TMPDIR` to root's environment when executing the `build` command with `sudo`.

When you run one of the action commands (i.e. `run`, `exec`, or `shell`) with a container from the container library or an OCI registry, Singularity builds the container in the temporary directory caches it and runs it from the cached location.

Consider the following command:

```
singularity exec docker://busybox /bin/sh
```

This container is first built in `/tmp`. Since all the oci blobs are converted into SIF format, by default a temporary runtime directory is created in:

```
$HOME/.singularity/cache/oci-tmp/<sha256-code>/busybox_latest.sif
```

In this case, the `SINGULARITY_TMPDIR` and `SINGULARITY_CACHEDIR` variables will also be respected.

7.4 Pull Folder

For details about customizing the output location of pull, see the pull docs. You have the similar ability to set it to be something different, or to customize the name of the pulled image.

7.5 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.

7.5.1 Defaults

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

7.5.1.1 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.

7.5.1.2 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.

CHAPTER

EIGHT

SUPPORT FOR DOCKER AND OCI

8.1 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.

8.2 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)\_______
            (__)\       )\/\
```

(continues on next page)

(continued from previous page)

```
                 ||----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.

8.3 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
WARNING: Authentication token file not found : Only pulls of public images will␣
↪succeed
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*.

Note: The above authentication warning originates from a check for the existence of `${HOME}/.singularity/sylabs-token`. It can be ignored when making use of Docker Hub, or silenced by issuing `touch ${HOME}/.singularity/sylabs-token` once.

`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.

8.4 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.

8.4.1 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.

8.4.2 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.

8.5 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
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.

8.6 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*.

8.6.1 Working from the Singularity Command Line

8.6.1.1 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
```

(continues on next page)

(continued from previous page)

```
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
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`.

8.6.1.2 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, store it in the file `$HOME/.singularity/sylabs-token`.

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
 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

8.6.1.3 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
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*.

8.6.1.4 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
```

(continues on next page)

(continued from previous page)

```
-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.

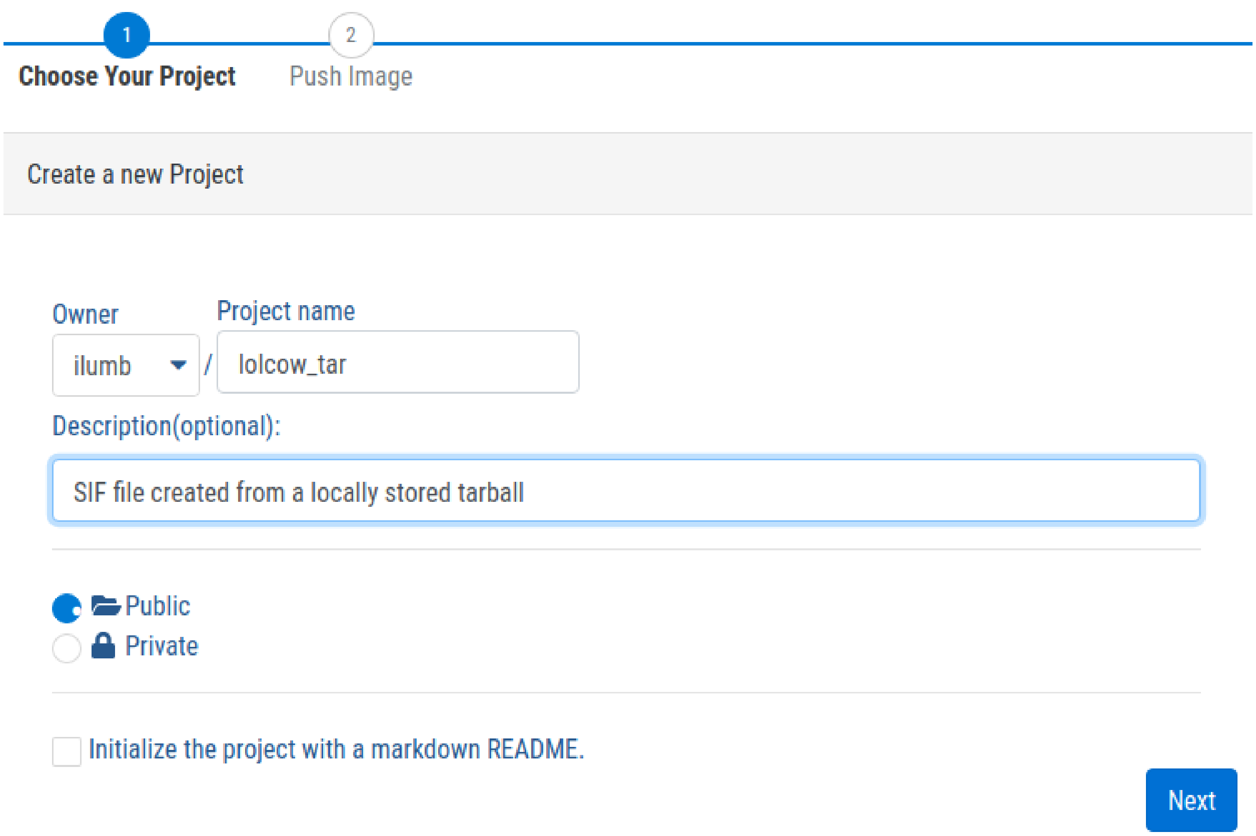
Note: The `docker-archive` bootstrap agent handles archives (`.tar` files) as well as compressed archives (`.tar.gz`) when containers are built for Singularity via its `build` command.

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*.

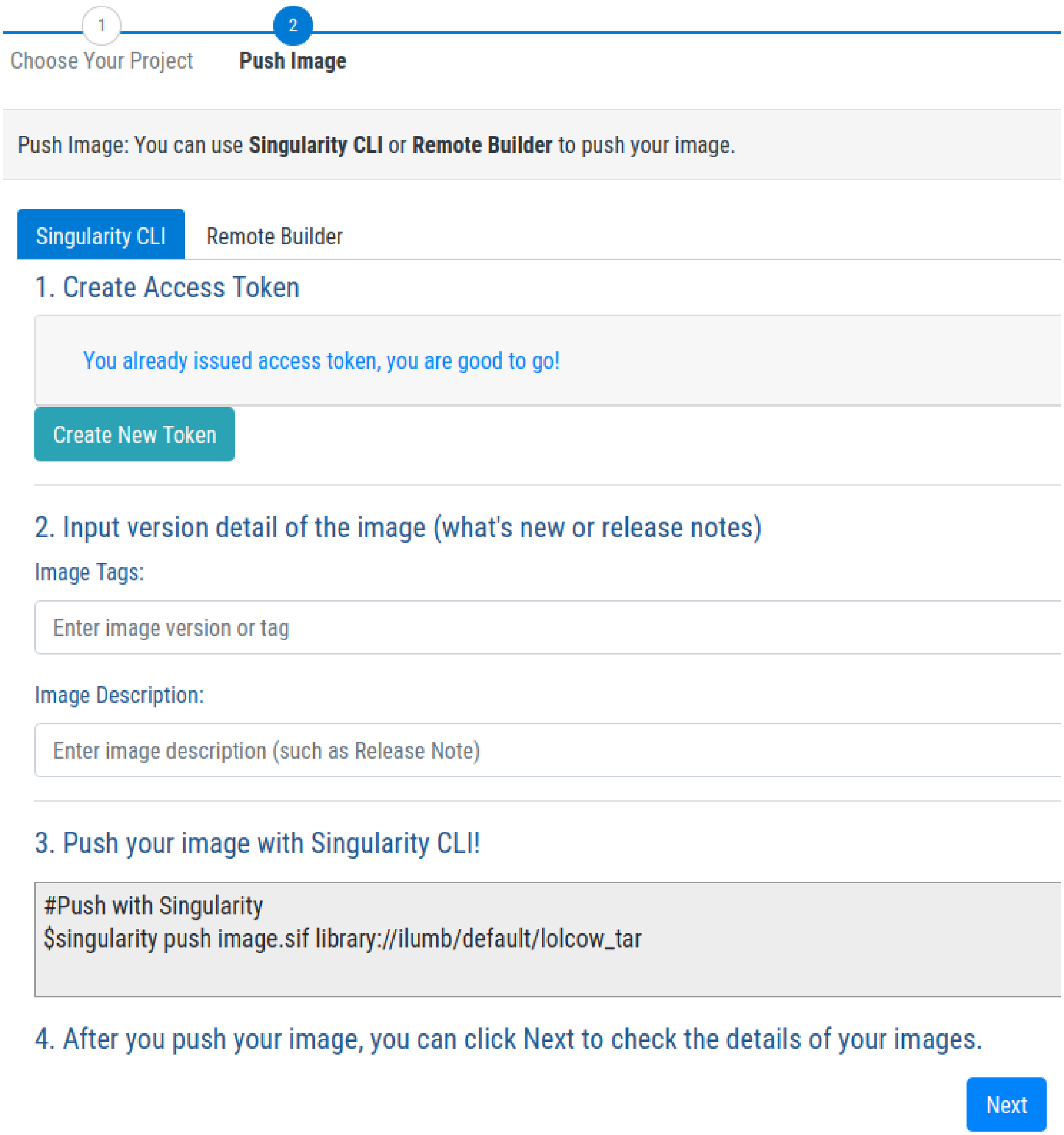
8.6.1.5 Pushing Locally Available Images to a Library

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:



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:



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.

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.

8.6.2 Working with Definition Files

8.6.2.1 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.

8.6.2.2 Remotely Bootstrapped and Built Containers

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

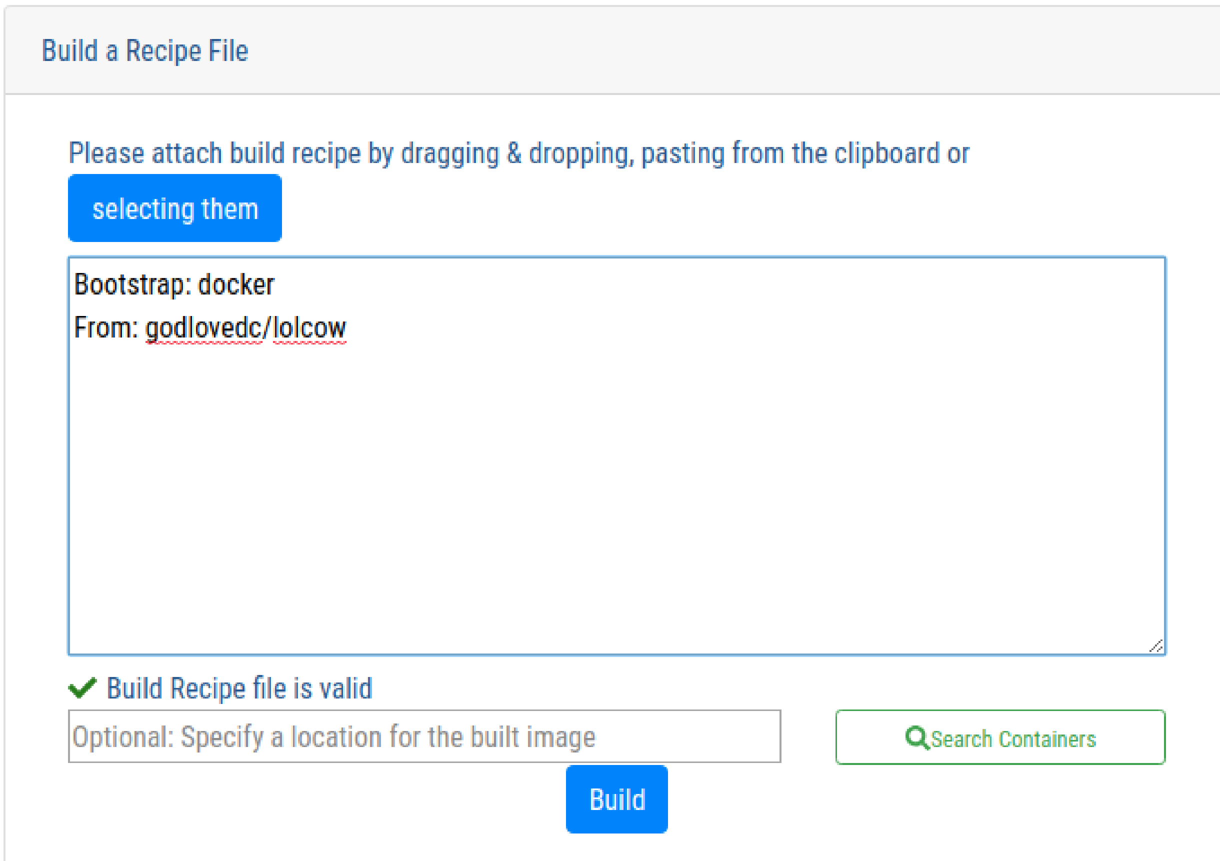
```
Bootstrap: docker
From: godlovedc/lolcow
```

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
```

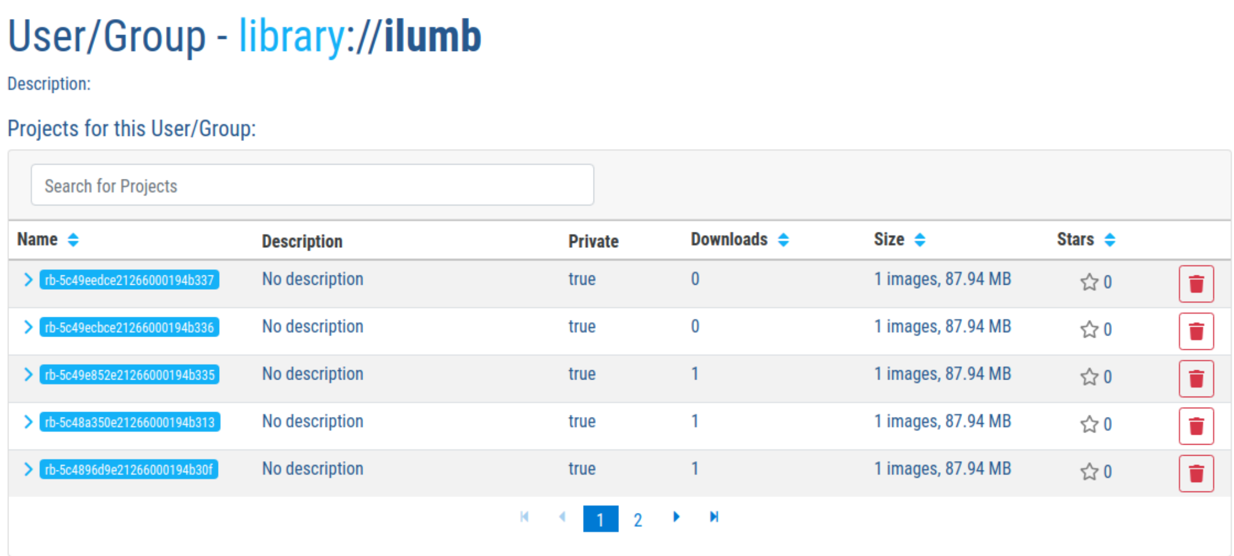
In the above, `--remote` has been added as the `build` option that causes use of the Remote Builder service. A much more subtle change, however, is the *absence* of `sudo` ahead of `singularity build`. Though subtle here, this absence is notable, as users can build containers via the Remote Builder with *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.



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).

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



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.

8.6.2.3 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* .

8.6.2.4 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`.

8.6.2.5 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.

8.6.2.6 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}
```

(continues on next page)

(continued from previous page)

```
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

8.6.2.7 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
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*.

8.7 OCI Image Support

8.7.1 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; 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."

8.7.1.1 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
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.

8.7.1.2 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
        │       ├── 9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2
        │       ├── 9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118
        │       ├── d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e
        │       └── f2a852991b0a36a9f3d6b2a33b98a461e9ede8393482f0deb5287afcbae2ce10
        ├── index.json
        └── oci-layout

4 directories, 10 files
```

8.7.1.3 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.

8.7.1.4 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"
      }
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

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",
      "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/
↪docker-clean \t&& echo 'Dir::Cache::pkgcache \"\"; Dir::Cache::srcpkgcache \"\";' >>
↪ /etc/apt/apt.conf.d/docker-clean \t\t&& echo 'Acquire::Languages \"none\";' > /etc/
↪apt/apt.conf.d/docker-no-languages \t\t&& echo 'Acquire::GzipIndexes \"true\";␣
↪Acquire::CompressionTypes::Order:: \"gz\";' > /etc/apt/apt.conf.d/docker-gzip-
↪indexes \t\t&& echo 'Apt::AutoRemove::SuggestsImportant \"false\";' > /etc/apt/apt.
↪conf.d/docker-autoremove-suggests"
    },
    {
      "created": "2017-09-18T23:31:38.788043199Z",
      "created_by": "/bin/sh -c rm -rf /var/lib/apt/lists/*"
    },
```

(continues on next page)

(continued from previous page)


```
    {
      "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.

8.7.2 Building Containers for Singularity from OCI Images

8.7.2.1 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.

8.7.2.2 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.

8.7.2.3 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*.

8.7.2.4 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...
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
```

(continues on next page)

(continued from previous page)

```
 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`.

8.7.2.5 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"
}
```

8.8 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.

8.9 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.

CHAPTER

NINE

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.

9.1 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.

9.2 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 directories `$HOME` , `/tmp` , `/proc` , `/sys` , `/dev`, and `$PWD` are among the system-defined bind paths.

9.3 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.

9.3.1 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.

9.3.2 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.

CHAPTER TEN

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.

10.1 Overview

A persistent overlay is a directory that "sits on top" of your compressed, immutable SIF container. When you install new software or create and modify files the overlay directory stores the changes.

If you want to use a SIF container as though it were writable, you can create a directory to use as a persistent overlay. Then you can specify that you want to use the directory as an overlay at runtime with the `--overlay` option.

You can use a persistent overlays with the following commands:

- `run`
- `exec`
- `shell`
- `instance.start`

10.2 Usage

To use a persistent overlay, you must first have a container.

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

Then you must create a directory. (You can also use the `--overlay` option with a legacy writable ext3 image.)

```
$ mkdir my_overlay
```

Now you can use this overlay directory with your container. Note that it is necessary to be root to use an overlay directory.

```
$ sudo singularity shell --overlay my_overlay/ ubuntu.sif

Singularity ubuntu.sif:~> touch /foo

Singularity ubuntu.sif:~> apt-get update && apt-get install -y vim

Singularity ubuntu.sif:~> which vim
```

(continues on next page)

(continued from previous page)

```
/usr/bin/vim

Singularity ubuntu.sif:~> exit
```

You will find that your changes persist across sessions as though you were using a writable container.

```
$ sudo singularity shell --overlay my_overlay/ ubuntu.sif

Singularity ubuntu.sif:~> ls /foo
/foo

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.

```
$ sudo singularity shell ubuntu.sif

Singularity ubuntu.sif:~> ls /foo
ls: cannot access 'foo': No such file or directory

Singularity ubuntu.sif:~> which vim

Singularity ubuntu.sif:~> exit
```

CHAPTER

ELEVEN

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.

11.1 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.

11.2 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      IMAGE
instance1        12715    /home/ysub/alpine_latest.sif
```

Note: The instances are linked with your user. So make sure to run *all* the instance commands either with or without the `sudo` privilege. If you `start` an instance with sudo and 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      IMAGE
instance1        12715    /home/ysub/alpine_latest.sif
instance2        12795    /home/ysub/alpine_latest.sif
instance3        12837    /home/ysub/alpine_latest.sif
```

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.

11.3 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>
```

(continues on next page)

(continued from previous page)

```
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Visit localhost on your browser, you should see a Welcome message!

11.4 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
```

11.4.1 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
```

(continues on next page)

(continued from previous page)

```
    # 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
```

11.4.2 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
```

11.4.3 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 Standard Container Integration Format (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
```

CHAPTER

TWELVE

ENVIRONMENT AND METADATA

Singularity containers support environment variables and labels that you can add to your container during the build process. If you are looking for environment variables to set up the environment on the host system during build time, see the *build environment section*.

12.1 Overview

Environment variables can be included in your container by adding them in your definition file:

- In the %environment section of your definition file.

```
Bootstrap: library
From: library/alpine

%environment
    VARIABLE_ONE = hello
    VARIABLE_TWO = world
    export VARIABLE_ONE VARIABLE_TWO
```

- Or in the %post section of your definition file.

```
Bootstrap: library
From: library/alpine

%post
    echo 'export VARIABLE_NAME=variable_value' >>$SINGULARITY_ENVIRONMENT
```

You can also add labels to your container using the %labels section like so:

```
Bootstrap: library
From: library/alpine

%labels
    OWNER = Joana
```

To view the labels within your container you use the inspect command:

```
$  singularity inspect mysifimage.sif
```

This will give you the following output:

```
{
    "OWNER": "Joana",
    "org.label-schema.build-date": "Monday_07_January_2019_0:01:50_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"
}
```

Many of these labels are created by default, but you can also see the custom label that was added in the example above.

The `inspect` command has *additional options* that are useful for viewing the container's metadata.

12.2 Environment

If you build a container from the Container Library or Docker Hub, the environment will be included with the container at build time. You can also define new environment variables in your definition file as follows:

```
Bootstrap: library
From: library/alpine

%environment
    #First define the variables
    VARIABLE_PATH=/usr/local/bootstrap
    VARIABLE_VERSION=3.0
    #Then export them
    export VARIABLE_PATH VARIABLE_VERSION
```

You may need to add environment variables to your container during the `%post` section. For instance, maybe you will not know the appropriate value of a variable until you have installed some software. To add variables to the environment during `%post` you can use the `$SINGULARITY_ENVIRONMENT` variable with the following syntax:

```
%post
    echo 'export VARIABLE_NAME=variable_value' >>$SINGULARITY_ENVIRONMENT
```

Text in the `%environment` section will be appended to the file `/.singularity.d/env/90-environment.sh` while text redirected to `$SINGULARITY_ENVIRONMENT` will appear in the file `/.singularity.d/env/91-environment.sh`. If nothing is redirected to `$SINGULARITY_ENVIRONMENT` in the `%post` section, the file `/.singularity.d/env/91-environment.sh` will not exist.

Because files in `/.singularity.d/env` are sourced in alpha-numerical order, variables added using `$SINGULARITY_ENVIRONMENT` take precedence over those added via the `%environment` section.

If you need to define a variable in the container at runtime, when you execute Singularity pass a variable prefixed with `SINGULARITYENV_`. These variables will be transposed automatically and the prefix will be stripped. For example, let's say we want to set the variable `HELLO` to have value `world`. We can do that as follows:

```
$ SINGULARITYENV_HELLO=world singularity exec centos7.img env | grep HELLO
HELLO=world
```

The `--cleanenv` option can be used to remove the host environment and execute a container with a minimal environment.

```
$ singularity exec --cleanenv centos7.img env
LD_LIBRARY_PATH=:/usr/local/lib:/usr/local/lib64
SINGULARITY_NAME=test.img
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/home/gmk/git/singularity
LANG=en_US.UTF-8
SHLVL=0
SINGULARITY_INIT=1
SINGULARITY_CONTAINER=test.img
```

Without the `--cleanenv` flag, the environment on the host system will be present within the container at run time.

If you need to change the `$PATH` of your container at run time there are a few special environmental variables you can use:

- `SINGULARITYENV_PREPEND_PATH=/good/stuff/at/beginning` to prepend directories to the beginning of the `$PATH`
- `SINGULARITYENV_APPEND_PATH=/good/stuff/at/end` to append directories to the end of the `$PATH`
- `SINGULARITYENV_PATH=/a/new/path` to override the `$PATH` within the container

12.3 Labels

Your container stores metadata about its build, along with Docker labels, and custom labels that you define during build in a `%labels` section.

For containers that are generated with Singularity version 3.0 and later, labels are represented using the rc1 Label Schema. For example:

```
$ 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"
    }
```

You will notice that the one label doesn't belong to the label schema, `OWNER`. This was a user provided label during bootstrap.

You can add custom labels to your container in a bootstrap file:

```
Bootstrap: docker
From: ubuntu: latest

%labels
  OWNER Joana
```

The `inspect` command is useful for viewing labels and other container meta-data. The next section will detail its various options.

12.4 The `inspect` command

The `inspect` command gives you the ability to print out the labels and/or other metadata that was added to your container using the definition file.

12.4.1 `--labels`

This flag corresponds to the default behavior of the `inspect` command. When you run a `singularity inspect <your-container.sif>` you will get output like this.

```
$ singularity inspect --labels jupyter.sif

{
    "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"
}
```

This is the same as running `singularity inspect jupyter.sif`.

12.4.2 `--deffile`

This flag gives you the def file(s) that was used to create 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
```

(continues on next page)

(continued from previous page)

```
    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
    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 a definition file for a `jupyter.sif` container.

12.4.3 `--runscript`

This flag 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
```

(continues on next page)

(continued from previous page)

```
# 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
```

12.4.4 --test

This flag shows the test script for the image.

```
$ singularity inspect --test jupyter.sif
```

This will output the corresponding `%test` section from the definition file.

12.4.5 --environment

This flag shows the environment settings for the image. The respective environment variables set in `%environment` section (So the ones in `90-environment.sh`) and `SINGULARITY_ENV` variables set at runtime (that are located in‘‘91-environment.sh‘‘) will be printed out.

```
$ 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

==91-environment.sh==
export RANDOM=123456
```

As you can see, the `JUP_PORT` and `JUP_IPNAME` were previously defined in the `%environment` section of the defintion file, while the RANDOM variable shown regards to the use of `SINGULARITYENV_` variables, so in this case `SINGULARITYENV_RANDOM` variable was set and exported at runtime.

12.4.6 --helpfile

This 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
```

12.4.7 --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}"
        },
        "type": "container"
}
```


12.5 Container Metadata

Inside of the container, metadata is stored in the `/.singularity.d` directory. You probably shouldn't edit any of these files directly but it may be helpful to know where they are and what they do:

```
/.singularity.d/

├── actions
│   ├── exec
│   ├── run
│   ├── shell
│   ├── start
│   └── test
├── env
│   ├── 01-base.sh
|   ├── 10-docker2singularity.sh
│   ├── 90-environment.sh
```

(continues on next page)

(continued from previous page)

```
│   ├── 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.
- **env**: All *.sh files in this directory are sourced in alpha-numeric order when the container is initiated. For legacy purposes there is a symbolic link called `/environment` that points to `/.singularity.d/env/90-environment.sh`.
- **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.

CHAPTER

THIRTEEN

SIGNING AND VERIFYING CONTAINERS

Singularity 3.0 introduces the abilities 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.

13.1 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) Click the "Manage my API tokens" button from the "Account Management" page.
6) Click "Create".
7) Click "Copy token to Clipboard" from the "New API Token" page.
8) Paste the token string into your `~/.singularity/sylabs-token` file.

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

```
$ singularity pull library://alpine

$ singularity verify alpine_latest.sif
Verifying image: alpine_latest.sif
Data integrity checked, authentic and signed by:
    Sylabs Admin <support@sylabs.io>, KeyID 51BE5020C508C7E9
```

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

13.2 Signing your own containers

13.2.1 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 keys newpair
Enter your name (e.g., John Doe) : Dave Godlove
Enter your email address (e.g., john.doe@example.com) : d@sylabs.io
Enter optional comment (e.g., development keys) : demo
Generating Entity and OpenPGP Key Pair... Done
Enter encryption passphrase :
```

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

```
$ singularity keys list
Public key listing (/home/david/.singularity/sypgp/pgp-public):

0) U: Dave Godlove (demo) <d@sylabs.io>
   C: 2018-10-08 15:25:30 -0400 EDT
   F: 135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A
   L: 4096
   --------
```

In the output above, the letters stand for the following:

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

After generating your key you can optionally push it to the Keystore using the fingerprint like so:

```
$ singularity keys push 135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A
public key `135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A` pushed to server successfully
```

This will allow others to verify images that you have signed.

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

```
$ singularity keys search Godlove
Search results for 'Godlove'

Type bits/keyID     Date       User ID
--------------------------------------------------------------------------------
pub  4096R/8EE0DC4A 2018-10-08 Dave Godlove (demo) <d@sylabs.io>
--------------------------------------------------------------------------------

$ singularity keys pull 8EE0DC4A
1 key(s) fetched and stored in local cache /home/david/.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).

13.2.2 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
Data integrity checked, authentic and signed by:
    Dave Godlove (demo) <d@sylabs.io>, KeyID FED5BBA38EE0DC4A
```

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

```
$ rm ~/.singularity/sypgp/*

$ singularity verify my_container.sif
Verifying image: my_container.sif
INFO:    key missing, searching key server for KeyID: FED5BBA38EE0DC4A...
INFO:    key retreived successfully!
Store new public key 135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A? [Y/n] y
Data integrity checked, authentic and signed by:
    Dave Godlove (demo) <d@sylabs.io>, KeyID FED5BBA38EE0DC4A
```

Answering yes at the interactive prompt will store the Public key locally so you will not have to contact the Keystore again the next time you verify your container.

CHAPTER

FOURTEEN

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.

14.1 Linux Capabilities

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 capabilities to open raw sockets so that they can use `ping` in a container where the binary is controlled via capabilities (i.e. a recent version of CentOS).

To do so, the admin would issue a command such as this:

```
$ sudo singularity capability add --user david CAP_NET_RAW
```

This means the user `david` has just been granted permissions (through Linux capabilities) to open raw sockets within Singularity containers.

The admin can check that this change is in effect with the `capability list` command.

```
$ sudo singularity capability list --user david
CAP_NET_RAW
```

To take advantage of this new capability, the user `david` must also request the capability when executing a container with the `--add-caps` flag like so:

```
$ singularity exec --add-caps CAP_NET_RAW 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=18.3 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 18.320/18.320/18.320/0.000 ms
```

If the admin decides that it is no longer necessary to allow the user `dave` to open raw sockets within Singularity containers, they can revoke the appropriate Linux capability like so:

```
$ sudo singularity capability drop --user david CAP_NET_RAW
```

The `capabiltiy add` and `drop` subcommands will also accept the case insensitive keyword `all` to grant or revoke all Linux capabilities to a user or group. Similarly, the `--add-caps` option will accept the `all` keyword. Of course appropriate caution should be exercised when using this keyword.

14.2 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.

14.2.1 `--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.

14.2.2 `--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
```

14.2.3 `--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
```

14.2.4 `--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.

14.2.5 `--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)
```

CHAPTER

FIFTEEN

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.

15.1 --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
```

15.2 --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
```

15.3 --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

```

(continues on next page)

(continued from previous page)

```
$ sudo singularity exec --net my_container.sif hostname -I
10.22.0.4
```

15.4 --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

Administrators can also define custom network configurations and place them in the same directory for the benefit of users.

15.5 --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.

CHAPTER

SIXTEEN

LIMITING CONTAINER RESOURCES WITH CGROUPS

Starting in Singularity 3.0, users have the ability to limit container resources using cgroups.

16.1 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`. 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.

16.2 Examples

16.2.1 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/`.

16.2.2 Limiting CPU

Limit CPU resources using one of the following strategies. The `cpu` section of the configuration file can limit memory with the following:

16.2.2.1 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.

16.2.2.2 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
```

16.2.2.3 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

16.2.3 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

16.2.3.1 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.

CHAPTER

SEVENTEEN

APPENDIX

17.1 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.

17.1.1 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)

17.1.2 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.

17.1.3 C

1. **SINGULARITY_CACHEDIR**: Specifies the directory for image downloads to be cached in.
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.

17.1.4 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_DNS**: A list of the DNS server addresses separated by commas to be added in `resolv.conf`.
5. **SINGULARITY_DOCKER_LOGIN**: To specify the interactive prompt for docker authentication.
6. **SINGULARITY_DOCKER_USERNAME**: To specify a username for docker authentication.
7. **SINGULARITY_DOCKER_PASSWORD**: To specify the password for docker authentication.
8. **SINGULARITY_DROP_CAPS**: To specify a list (comma separated string) of capabilities to be dropped. Default is an empty string.

17.1.5 E

1. **SINGULARITY_ENVIRONMENT**: Contains all the environment variables that have been exported in your container.
2. **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*.
3. **SINGULARITYENV_APPEND_PATH**: Used to append directories to the end of the `$PATH` environment variable. You can see more in detail on how to use this variable in our *environment and metadata section*.
4. **SINGULARITYENV_PATH**: A specified path to override the `$PATH` environment variable within the container. You can see more in detail on how to use this variable in our *environment and metadata section*.
5. **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*.

17.1.6 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.

17.1.7 G

1. **SINGULARITY_GROUP**: Used to specify a string of capabilities for the given group.

17.1.8 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.

17.1.9 I

1. **SINGULARITY_IMAGE**: Filename of the container.

17.1.10 J

1. **SINGULARITY_JSON**: Specifies the structured json of the def file, every node as each section in the def file.

17.1.11 K

1. **SINGULARITY_KEEP_PRIVS**: To let root user keep privileges in the container. Default is set to false.

17.1.12 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.

17.1.13 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.

17.1.14 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.

17.1.15 P

1. **SINGULARITY_PWD** and **SINGULARITY_TARGET_PWD**: The initial working directory for payload process inside the container.

17.1.16 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.

17.1.17 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.

17.1.18 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.

17.1.19 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.

17.1.20 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.

17.2 Build Modules

17.2.1 `library` bootstrap agent

17.2.1.1 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.

17.2.1.2 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`.

17.2.2 `docker` bootstrap agent

17.2.2.1 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.

17.2.2.2 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.

17.2.2.3 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*.

17.2.3 `shub` bootstrap agent

17.2.3.1 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.

17.2.3.2 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.

17.2.3.3 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.

17.2.4 `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.

17.2.4.1 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.

17.2.4.2 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.

17.2.4.3 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.

17.2.5 `yum` bootstrap agent

This module allows you to build a Red Hat/CentOS/Scientific Linux style container from a mirror URI.

17.2.5.1 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.

17.2.5.2 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, than 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.

17.2.5.3 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 `debootstrap` 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
```

17.2.6 `debootstrap` build agent

This module allows you to build a Debian/Ubuntu style container from a mirror URI.

17.2.6.1 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.

17.2.6.2 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.

17.2.6.3 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
```

17.2.7 `arch` bootstrap agent

This module allows you to build a Arch Linux based container.

17.2.7.1 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).

17.2.7.2 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.

17.2.7.3 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.

17.2.8 `busybox` bootstrap agent

This module allows you to build a container based on BusyBox.

17.2.8.1 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.

17.2.8.2 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.

17.2.8.3 Notes

You can build a fully functional BusyBox container that only takes up ~600kB of disk space!

17.2.9 `zypper` bootstrap agent

This module allows you to build a Suse style container from a mirror URI.

17.2.9.1 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.

17.2.9.2 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.