Singularity Container Documentation

Release 3.0

User Docs

Dec 17, 2018

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

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.)

2.1 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.

2.2 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.

2.2.1 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.

2.2.1.1 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
```

2.2.1.2 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
```

2.2.2 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.

2.2.2.1 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=v3.0.1 # 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
```

2.2.2.2 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`.

2.2.2.3 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
```

2.2.3 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=3.0.1 && # 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.

2.2.4 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.

2.2.5 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.

2.2.5.1 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.
*** 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.

2.2.5.2 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
```

2.3 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.

2.3.1 Setup

First, install the following software:

2.3.1.1 Windows

Install the following programs:

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

2.3.1.2 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
```

2.3.2 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-ubuntu-bionic64 && \
    vagrant init $VM && \
    vagrant up && \
    vagrant ssh
```

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

```
vagrant@ubuntu-bionic:~$ singularity version
    3.0.1
```

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.

2.4 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
```

(continues on next page)

(continued from previous page)

```
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.

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

THREE

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.

3.1 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.

3.2 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.

3.3 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
```

3.4 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/
```

3.5 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.

3.6 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.

3.7 Build options

3.7.1 `--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`.

3.7.2 `--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.

3.7.3 `--force`

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

3.7.4 `--json`

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

3.7.5 `--library`

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

3.7.6 `--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.

3.7.7 `--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".)

3.7.8 `--sandbox`

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

3.7.9 `--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.

3.7.10 `--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.

3.8 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

FOUR

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.

4.1 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

4.2 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)

4.3 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.
```

4.3.1 %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.

4.3.2 %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.

4.3.3 %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.

4.3.4 %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.

4.3.5 %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
```

4.3.6 %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)
```

4.3.7 %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.
```

4.3.8 %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.

4.3.9 %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.
```

4.4 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
```

4.5 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

FIVE

BIND PATHS AND MOUNTS

If enabled by the system administrator, Singularity allows you to map directories on your host system to directories within your container using bind mounts. This allows you to read and write data on the host system with ease.

5.1 Overview

When Singularity 'swaps' the host operating system for the one inside your container, the host file systems becomes inaccessible. But you may want to read and write files on the host system from within the container. To enable this functionality, Singularity will bind directories back into the container via two primary methods: system-defined bind paths and user-defined bind paths.

5.2 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.

5.3 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.

5.3.1 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.

5.3.2 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

SIX

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.

6.1 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`

6.2 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

SEVEN

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.

7.1 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.

7.2 Signing your own containers

7.2.1 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).

7.2.2 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

EIGHT

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.

8.1 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.

8.2 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.

8.2.1 `--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.

8.2.2 `--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
```

8.2.3 `--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
```

8.2.4 `--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.

8.2.5 `--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

NINE

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.

9.1 `--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
```

9.2 `--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
```

9.3 `--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
```

9.4 --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.

9.5 --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

TEN

LIMITING CONTAINER RESOURCES WITH CGROUPS

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

10.1 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.

10.2 Examples

10.2.1 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/`.

10.2.2 Limiting CPU

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

10.2.2.1 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.

10.2.2.2 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
```

10.2.2.3 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

10.2.3 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

10.2.3.1 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

ELEVEN

APPENDIX

11.1 `library` bootstrap agent

11.1.1 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.

11.1.2 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://cloud.sylabs.io/library`.

11.2 `docker` bootstrap agent

11.2.1 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.

11.2.2 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.

11.2.3 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.

11.3 `shub` bootstrap agent

11.3.1 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.

11.3.2 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.

11.3.3 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.

11.4 `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.

11.4.1 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.

11.4.2 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.

11.4.3 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.

11.5 `yum` bootstrap agent

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

11.5.1 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.

11.5.2 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.

11.5.3 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
```

11.6 `debootstrap` build agent

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

11.6.1 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.

11.6.2 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.

11.6.3 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
```

11.7 `arch` bootstrap agent

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

11.7.1 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).

11.7.2 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.

11.7.3 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.

11.8 `busybox` bootstrap agent

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

11.8.1 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.

11.8.2 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.

11.8.3 Notes

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

11.9 `zypper` bootstrap agent

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

11.9.1 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.

11.9.2 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.