Singularity Container Documentation

Release 2.6

User Docs

Feb 15, 2019

CONTENTS

1 Quick Start **1**
1.1 Quick Installation Steps . . . 1
1.2 Overview of the Singularity Interface . . . 1
1.3 Download pre-built images . . . 3
1.4 Interact with images . . . 4
1.4.1 Shell . . . 4
1.4.2 Executing Commands . . . 5
1.4.3 Running a container . . . 5
1.4.4 Working with Files . . . 5
1.5 Build images from scratch . . . 6
1.5.1 Sandbox Directory . . . 6
1.5.2 Writable Image . . . 6
1.5.3 Converting images from one format to another . . . 6
1.5.4 Singularity Recipes . . . 7

2 Introduction **9**
2.1 Welcome to Singularity! . . . 9
2.1.1 Mobility of Compute . . . 9
2.1.2 Reproducibility . . . 9
2.1.3 User Freedom . . . 10
2.1.4 Support on Existing Traditional HPC . . . 10
2.2 A High Level View of Singularity . . . 10
2.2.1 Security and privilege escalation . . . 10
2.2.2 The Singularity container image . . . 11
2.2.2.1 *Copying, sharing, branching, and distributing your image* . . . 11
2.2.2.2 *Supported container formats* . . . 11
2.2.2.3 *Supported Uniform Resource Identifiers (URI)* . . . 11
2.2.3 Name-spaces and isolation . . . 12
2.2.4 Compatibility with standard work-flows, pipes and IO . . . 12
2.2.5 The Singularity Process Flow . . . 13
2.3 The Singularity Usage Workflow . . . 13
2.3.1 Singularity Commands . . . 14
2.4 Support . . . 15
2.5 About . . . 15
2.5.1 Overview . . . 15
2.5.2 Background . . . 15
2.5.2.1 The Singularity Solution . . . 15
2.5.2.2 Portability and Reproducibility . . . 17
2.5.3 Features . . . 18
2.5.3.1 Encapsulation of the environment . . . 18

2.5.3.2 Containers are image based . . . 18
2.5.3.3 No user contextual changes or root escalation allowed . . . 18
2.5.3.4 No root owned daemon processes . . . 18
2.5.4 Use Cases . . . 19
2.5.4.1 BYOE: Bring Your Own Environment . . . 19
2.5.4.2 Reproducible science . . . 19
2.5.4.3 Static environments (software appliances) . . . 19
2.5.4.4 Legacy code on old operating systems . . . 19
2.5.4.5 Complicated software stacks that are very host specific . . . 20
2.5.4.6 Complicated work-flows that require custom installation and/or data . . . 20
2.5.5 License . . . 20
2.5.6 Getting started . . . 20

3 Installation **21**
3.1 Before you begin . . . 21
3.2 Install from a tag . . . 21
3.3 Install a specific release . . . 22
3.4 Install the development branch . . . 22
3.5 Remove an old version . . . 22
3.6 Install on Windows . . . 23
3.6.1 Setup . . . 23
3.6.2 Singularity Vagrant Box . . . 23
3.7 Install on Linux . . . 24
3.7.1 Installation from Source . . . 24
3.7.1.1 Option 1: Download latest stable release . . . 25
3.7.1.2 Option 2: Download the latest development code . . . 25
3.7.1.3 Prefix in special characters . . . 25
3.7.1.4 Updating . . . 26
3.7.2 Debian Ubuntu Package . . . 26
3.7.2.1 Testing first with Docker . . . 26
3.7.2.2 Adding the Mirror and installing . . . 26
3.7.3 Build an RPM from source . . . 28
3.7.4 Build an DEB from source . . . 29
3.7.5 Install on your Cluster Resource . . . 29
3.8 Install on Mac . . . 29
3.8.1 Setup . . . 29
3.8.2 Option 1: Singularity Vagrant Box . . . 30
3.8.3 Option 2: Vagrant Box from Scratch (more advanced alternative) . . . 31
3.9 Requesting an Installation . . . 32
3.9.1 How do I ask for Singularity on my local resource? . . . 32
3.9.2 Information Resources . . . 32
3.9.2.1 Background . . . 32
3.9.2.2 Security . . . 32
3.9.2.3 Presentations . . . 32
3.9.3 Installation Request . . . 32

4 Build a Container **35**
4.1 Overview . . . 35
4.2 Downloading a existing container from Singularity Hub . . . 35
4.3 Downloading a existing container from Docker Hub . . . 37
4.4 Creating `--writable` images and `--sandbox` directories . . . 37
4.4.1 `--writable` . . . 37
4.4.2 `--sandbox` . . . 37
4.5 Converting containers from one format to another . . . 38

4.6 Building containers from Singularity recipe files . . . 38
4.6.1 `--force` . . . 39
4.6.2 `--section` . . . 39
4.6.3 `--notest` . . . 39
4.6.4 `--checks` . . . 39
4.7 More Build topics . . . 40

5 Build Environment **41**
5.1 Cache Folders . . . 41
5.2 Temporary Folders . . . 41
5.3 Pull Folder . . . 42
5.4 Environment Variables . . . 42
5.5 Cache . . . 42
5.5.1 Defaults . . . 43
5.5.1.1 Docker . . . 43
5.5.1.2 Singularity Hub . . . 43
5.5.2 General . . . 44

6 Container Recipes **45**
6.1 Overview . . . 45
6.1.1 Header . . . 45
6.1.2 Sections . . . 46
6.1.2.1 %help . . . 46
6.1.2.2 %setup . . . 47
6.1.2.3 %files . . . 47
6.1.2.4 %labels . . . 49
6.1.2.5 %environment . . . 50
6.1.2.6 %post . . . 51
6.1.2.7 %runscript . . . 52
6.1.2.8 %test . . . 54
6.2 Apps . . . 54
6.3 Examples . . . 57
6.4 Best Practices for Build Recipes . . . 57

7 Singularity Flow **59**
7.1 Building Images . . . 59
7.1.1 The Singularity Flow . . . 59
7.1.2 1. Development Commands . . . 60
7.1.2.1 Sandbox Folder . . . 60
7.1.2.2 Writable Image . . . 62
7.1.3 2. Production Commands . . . 63
7.1.3.1 Recommended Production Build . . . 63
7.1.3.2 Production Build from Sandbox . . . 64

8 Bind Paths and Mounts **65**
8.1 Overview . . . 65
8.1.1 System-defined bind points . . . 65
8.1.2 User-defined bind points . . . 65
8.1.2.1 Specifying Bind Paths . . . 65
8.1.2.2 Binding with Overlay . . . 66

9 Persistent Overlays **69**
9.1 Overview . . . 69
9.2 Usage . . . 69

10 Running Services **71**
10.1 Why container instances? . . . 71
10.2 Container Instances in Singularity . . . 71
10.3 Nginx "Hello-world" in Singularity . . . 73
10.4 Putting all together . . . 74
10.4.1 Building the image . . . 74
10.4.2 Running the Server . . . 76
10.4.3 Making it Pretty . . . 77
10.5 Important Notes . . . 78

11 Container Checks **79**
11.1 Tags and Organization . . . 79
11.2 What checks are available? . . . 80

12 Environment and Metadata **81**
12.1 Environment . . . 81
12.2 Labels . . . 82
12.3 Container Metadata . . . 83

13 Reproducible SCI-F Apps **85**
13.1 Why do we need SCI-F? . . . 85
13.1.1 Mixed up Modules . . . 85
13.1.2 Container Transparency . . . 86
13.1.3 Container Modularity . . . 88
13.1.4 Sections . . . 90
13.1.5 Interaction . . . 90
13.2 SCI-F Example: Cowsay Container . . . 92

14 Singularity and Docker **95**
14.1 TLDR (Too Long Didn't Read) . . . 95
14.2 Import a Docker image into a Singularity Image . . . 95
14.3 Quick Start: The Docker Registry . . . 96
14.4 The Build Specification file, Singularity . . . 97
14.5 How does the runscript work? . . . 98
14.6 How do I specify my Docker image? . . . 99
14.7 Custom Authentication . . . 99
14.7.1 Authentication in the Singularity Build File . . . 100
14.7.2 Authentication in the Environment . . . 100
14.7.3 Testing Authentication . . . 100
14.8 Best Practices . . . 100
14.8.1 1. Installation to Root . . . 101
14.8.2 2. Library Configurations . . . 101
14.8.3 3. Don't install to $HOME or $TMP . . . 101
14.9 Troubleshooting . . . 101

15 Troubleshooting **103**
15.1 No space left on device . . . 103
15.2 Segfault on Bootstrap of Centos Image . . . 103
15.3 How to use Singularity with GRSecurity enabled kernels . . . 104
15.4 The container isn't working on a different host! . . . 104
15.5 Invalid Argument or Unknown Option . . . 105
15.6 Error running Singularity with sudo . . . 105
15.7 How to resolve "Too many levels of symbolic links" error . . . 105

16 Appendix **107**

16.1 build-docker-module . . . 107
16.1.1 Overview . . . 107
16.1.2 Keywords . . . 107
16.1.3 Notes . . . 108
16.2 build-shub . . . 108
16.2.1 Overview . . . 108
16.2.2 Keywords . . . 108
16.2.3 Notes . . . 108
16.3 build-localimage . . . 109
16.3.1 Overview . . . 109
16.3.2 Keywords . . . 109
16.3.3 Notes . . . 109
16.4 build-yum . . . 109
16.4.1 Overview . . . 109
16.4.2 Keywords . . . 109
16.4.3 Notes . . . 110
16.5 build-debootstrap . . . 110
16.5.1 Overview . . . 110
16.5.2 Keywords . . . 110
16.5.3 Notes . . . 111
16.6 build-arch . . . 111
16.6.1 Overview . . . 111
16.6.2 Keywords . . . 111
16.6.3 Notes . . . 111
16.7 build-busybox . . . 111
16.7.1 Overview . . . 112
16.7.2 Keywords . . . 112
16.7.3 Notes . . . 112
16.8 build-zypper . . . 112
16.8.1 Overview . . . 112
16.8.2 Keywords . . . 112
16.9 Singularity Action Flags . . . 113
16.9.1 Examples . . . 113
16.10 Commands . . . 114
16.10.1 Command Usage . . . 114
16.10.1.1 The Singularity command . . . 114
16.10.1.1.1 Options and argument processing . . . 114
16.10.1.1.2 Singularity Help . . . 115
16.10.1.2 Commands Usage . . . 115
16.10.1.3 Support . . . 116
16.10.2 build . . . 117
16.10.2.1 Overview . . . 117
16.10.2.2 Examples . . . 117
16.10.2.2.1 Download an existing container from Singularity Hub or Docker Hub . . . 117
16.10.2.2.2 Create –writable images and –sandbox directories . . . 117
16.10.2.2.3 Convert containers from one format to another . . . 117
16.10.2.2.4 Build a container from a Singularity recipe . . . 117
16.10.3 exec . . . 118
16.10.3.1 Examples . . . 118
16.10.3.1.1 Printing the OS release inside the container . . . 118
16.10.3.1.2 Printing the OS release for a running instance . . . 118
16.10.3.1.3 Runtime Flags . . . 118
16.10.3.1.4 Special Characters . . . 118
16.10.3.1.5 A Python example . . . 119

16.10.3.1.6 A GPU example . . . 120
16.10.4 inspect . . . 122
16.10.4.1 JSON Api Standard . . . 122
16.10.4.2 Inspect Flags . . . 123
16.10.4.2.1 Labels . . . 124
16.10.4.2.2 Runscript . . . 124
16.10.4.2.3 Help . . . 125
16.10.4.2.4 Environment . . . 126
16.10.4.2.5 Test . . . 126
16.10.4.2.6 Deffile . . . 127
16.10.5 pull . . . 128
16.10.5.1 Singularity Hub . . . 128
16.10.5.1.1 How do tags work? . . . 128
16.10.5.1.2 Image Names . . . 128
16.10.5.1.3 Custom Name . . . 129
16.10.5.1.4 Name by commit . . . 129
16.10.5.1.5 Name by hash . . . 129
16.10.5.1.6 Pull to different folder . . . 129
16.10.5.1.7 Pull by commit . . . 130
16.10.5.2 Docker . . . 130
16.10.6 run . . . 132
16.10.6.1 Overview . . . 132
16.10.6.2 Runtime Flags . . . 132
16.10.6.3 Examples . . . 132
16.10.6.3.1 Defining the Runscript . . . 133
16.10.7 shell . . . 134
16.10.7.1 Change your shell . . . 135
16.10.7.1.1 Bash . . . 135
16.10.7.2 Shell Help . . . 135
16.11 Image Command Group . . . 137
16.11.1 image.export . . . 137
16.11.2 image.expand . . . 137
16.11.2.1 Increasing the size of an existing image . . . 137
16.11.3 image.import . . . 138
16.11.4 image.create . . . 138
16.11.4.1 Creating a new blank Singularity container image . . . 139
16.11.4.1.1 Overwriting an image with a new one . . . 140
16.12 Instance Command Group . . . 140
16.12.1 instance.start . . . 140
16.12.1.1 Overview . . . 141
16.12.1.2 Examples . . . 141
16.12.1.2.1 Start an instance called cow1 from a container on Singularity Hub . . . 141
16.12.1.2.2 Start an interactive shell within the instance that you just started . . . 141
16.12.1.2.3 Execute the runscript within the instance . . . 141
16.12.1.2.4 Run a command within a running instance . . . 142
16.12.2 instance.list . . . 142
16.12.2.1 Overview . . . 142
16.12.2.2 Examples . . . 142
16.12.2.2.1 Start a few named instances from containers on Singularity Hub . . . 143
16.12.2.2.2 List running instances . . . 143
16.12.3 instance.stop . . . 143
16.12.3.1 Overview . . . 143
16.12.3.2 Examples . . . 143
16.12.3.2.1 Start a few named instances from containers on Singularity Hub . . . 143

16.12.3.2.2 Stop a single instance . . . 143
16.12.3.2.3 Stop all running instances . . . 144
16.13 Deprecated . . . 144
16.13.1 bootstrap . . . 144
16.13.1.1 Quick Start . . . 144

17 Contributing **147**
17.1 Support Singularity . . . 147
17.1.1 Singularity Google Group . . . 147
17.1.2 Singularity on Slack . . . 147
17.2 Contribute to the code . . . 147
17.2.1 Step 1. Fork the repo . . . 148
17.2.2 Step 2. Set up your config . . . 148
17.2.3 Step 3. Checkout a new branch . . . 150
17.2.4 Step 4. Make your changes . . . 150
17.2.5 Step 5. Push your branch to your fork . . . 151
17.2.6 Step 6. Submit a Pull Request . . . 151
17.2.7 Support, helping and spreading the word! . . . 151
17.3 Contributing to Documentation . . . 151

18 FAQ **153**
18.1 General Singularity Info . . . 153
18.1.1 Why the name "Singularity"? . . . 153
18.1.2 What is so special about Singularity? . . . 153
18.1.3 Which namespaces are virtualized? Is that select-able? . . . 154
18.1.4 What Linux distributions are you trying to get on-board? . . . 154
18.1.5 How do I request an installation on my resource? . . . 154
18.2 Basic Singularity usage . . . 154
18.2.1 Do you need administrator privileges to use Singularity? . . . 154
18.2.2 What if I don't want to install Singularity on my computer? . . . 154
18.2.3 Can you edit/modify a Singularity container once it has been instantiated? . . . 155
18.2.4 Can multiple applications be packaged into one Singularity Container? . . . 155
18.2.5 How are external file systems and paths handled in a Singularity Container? . . . 155
18.2.6 How does Singularity handle networking? . . . 155
18.2.7 Can Singularity support daemon processes? . . . 155
18.2.8 Can a Singularity container be multi-threaded? . . . 155
18.2.9 Can a Singularity container be suspended or check-pointed? . . . 156
18.2.10 Are there any special requirements to use Singularity through an HPC job scheduler? . . . 156
18.2.11 Does Singularity work in multi-tenant HPC cluster environments? . . . 156
18.2.12 Can I run X11 apps through Singularity? . . . 156
18.2.13 Can I containerize my MPI application with Singularity and run it properly on an HPC system? 156
18.2.14 Why do we call 'mpirun' from outside the container (rather than inside)? . . . 156
18.2.15 Does Singularity support containers that require GPUs? . . . 157
18.3 Container portability . . . 157
18.3.1 Are Singularity containers kernel-dependent? . . . 157
18.3.2 Can a Singularity container resolve GLIBC version mismatches? . . . 157
18.3.3 What is the performance trade off when running an application native or through Singularity? 157
18.4 Misc . . . 158
18.4.1 Are there any special security concerns that Singularity introduces? . . . 158

CHAPTER

ONE

QUICK START

This guide is intended for running Singularity on a computer where you have root (administrative) privileges. If you are learning about Singularity on a system where you lack root privileges, you can still complete the steps that do not require the `sudo` command. If you need to request an installation on your shared resource, check out our *requesting an installation help page* for information to send to your system administrator.

1.1 Quick Installation Steps

There are many ways to *install Singularity* but this quick start guide will only cover one. You will need `git` to download the source code and the appropriate tools and libraries. Create and move to a working directory and enter the following steps.

```
git clone https://github.com/sylabs/singularity.git

cd singularity

git fetch --all

git checkout 2.6.1

./autogen.sh

./configure --prefix=/usr/local

make

sudo make 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` option gives an overview of Singularity options and subcommands as follows:

```
$ singularity --help

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

(continues on next page)

(continued from previous page)

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

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

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

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

COMMAND GROUPS:
    image      Container image command group
```

(continues on next page)

(continued from previous page)

```
    instance   Persistent instance command group


CONTAINER USAGE OPTIONS:

    see singularity help <command>

For any additional help or support visit the Singularity

website: https://github.com/sylabs/singularity
```

For any additional help or support visit the Singularity website: https://www.sylabs.io/contact/ Singularity uses positional syntax (i.e. where the option is on the command line matters). Global options follow the `singularity` invocation and affect the way that Singularity runs any command. Then commands are passed followed by their options. For example, to pass the `--debug` option to the main `singularity` command and run Singularity with debugging messages on:

```
$ singularity --debug run shub://GodloveD/lolcow
```

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

```
$ singularity run --containall shub://GodloveD/lolcow
```

To learn more about a specific Singularity command, type one of the following:

```
$ singularity help <command>

$ singularity --help <command>

$ singularity -h <command>

$ singularity <command> --help

$ singularity <command> -h
```

Users can 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.simg            # See the container's help, if provided

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

1.3 Download pre-built images

You can use the *pull* and *build* commands to download pre-built images from an external resource like Singularity Hub or Docker Hub. When called on a native Singularity images like those provided on Singularity Hub, `pull` simply downloads the image file to your system.

```
$ singularity pull shub://vsoch/hello-world   # pull with default name, vsoch-hello-
↪world-master.simg

$ singularity pull --name hello.simg shub://vsoch/hello-world   # pull with custom␣
↪name
```

Singularity images can also be pulled and named by an associated GitHub commit or content hash. 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  # with default name

$ singularity pull --name funny.simg docker://godlovedc/lolcow # with custom name
```

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 Singularity Hub. 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 hello-world.simg shub://vsoch/hello-world

$ singularity build lolcow.simg 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 *recipe file <container-recipes>*. You can also use `build` to convert an image between the 3 major container formats supported by Singularity. We discuss those image formats below in the *Build images from scratch* section.

1.4 Interact with images

Once you have an image, you can interact with it in several ways. For these examples we will use a `hello-world.simg` image that can be downloaded from Singularity Hub like so.

```
$ singularity pull --name hello-world.simg shub://vsoch/hello-world
```

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 hello-world.simg

Singularity: Invoking an interactive shell within container...


# I am the same user inside as outside!

Singularity hello-world.simg:~/Desktop> whoami

vanessa


Singularity hello-world.simg:~/Desktop> id

uid=1000(vanessa) gid=1000(vanessa) groups=1000(vanessa),4(adm),24,27,30(tape),46,113,
↪128,999(input)
```

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

```
$ singularity shell shub://vsoch/hello-world
```

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 list the root (/) of our hello-world.simg image, we could do the following:

```
$ singularity exec hello-world.simg ls /

anaconda-post.log  etc   lib64       mnt   root  singularity  tmp

bin         home  lost+found  opt   run   srv          usr

dev         lib   media       proc  sbin  sys          var
```

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

```
$ singularity exec shub://singularityhub/ubuntu cat /etc/os-release
```

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 hello-world.simg

$ ./hello-world.simg
```

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

```
$ singularity run shub://GodloveD/lolcow
```

1.4.4 Working with Files

Files on the host are reachable from within the container.

```
$ echo "Hello World" > $HOME/hello-kitty.txt

$ singularity exec vsoch-hello-world-master.simg cat $HOME/hello-kitty.txt

Hello World
```

This example works because `hello-kitty.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 "I am your father" >/data/vader.sez

$ ~/sing-dev/bin/singularity exec --bind /data:/mnt hello-world.simg cat /mnt/vader.
↪sez

I am your father
```

1.5 Build images from scratch

As of Singularity v2.4 by default `build` produces immutable images in the squashfs file format. This ensures reproducible and verifiable images. 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 supports two other image formats: a `sandbox` format (which is really just a chroot directory), and a `writable` format (the ext3 file system that was used in Singularity versions less than 2.4).

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

1.5.1 Sandbox Directory

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

```
$ sudo singularity build --sandbox ubuntu/ docker://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. You can also write files to this directory from within a Singularity session (provided you have the permissions to do so). These files will be ephemeral and will disappear when the container is finished executing. However if you use the `--writable` option the changes will be saved into your directory so that you can use them the next time you use your container.

1.5.2 Writable Image

If you prefer to have a writable image file, you can `build` a container with the `--writable` option.

```
$ sudo singularity build --writable ubuntu.img docker://ubuntu
```

This produces an image that is writable with an ext3 file system. Unlike the sandbox, it is a single image file. Also by convention this file name has an ".img" extension instead of ".simg" . When you want to alter your image, you can use commands like `shell`, `exec`, `run`, with the `--writable` option. Because of permission issues it may be necessary to execute the container as root to modify it.

```
$ sudo singularity shell --writable ubuntu.img
```

1.5.3 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-squashfs sandbox
```

Doing so may break reproducibility if you have altered your sandbox outside of the context of a recipe file, so you are advised to exercise care. You can use `build` to convert containers to and from `writable`, `sandbox`, and default (squashfs) file formats via any of the six possible combinations.

1.5.4 Singularity Recipes

For a reproducible, production-quality container, we recommend that you build a container with the default (squashfs) file format using a Singularity recipe 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., Singularity Hub). A recipe file has a header and a body. The header determines what kind of base container to begin with, and the body is further divided into sections (called scriptlets) 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 recipe file:

```
Bootstrap: shub

From: singularityhub/ubuntu


%runscript

    exec echo "The runscript is the containers default runtime command!"


%files

   /home/vanessa/Desktop/hello-kitty.txt        # copied to root of container

   /home/vanessa/Desktop/party_dinosaur.gif     /opt/the-party-dino.gif #


%environment

    VARIABLE=MEATBALLVALUE

    export VARIABLE


%labels

   AUTHOR vsochat@stanford.edu


%post

    apt-get update && apt-get -y install python3 git wget

    mkdir /data

    echo "The post section is where you can install, and configure your container."
```

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

```
$ sudo singularity build ubuntu.simg Singularity
```

In this example, the header tells singularity to use a base Ubuntu image from Singularity Hub. The `%runscript` section defines actions for the container to take when it is executed (in this case a simple message). The `%files` section copies some files into the container from the host system at build time. The `%environment` section defines some environment variables that will be available to the container at runtime. The `%labels` section allows for custom metadata to be added to the container. And finally 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 custom apps. This is a very small example of the things that you can do with a *recipe file* . In addition to building a container from Singularity Hub, you can start with base images from Docker Hub, use images directly from official repositories such as Ubuntu, Debian, CentOS, Arch, and BusyBox, use an existing container on your host system as a base, or even take a snapshot of the host system itself and use that as a base image. If you want to build Singularity images without having singularity installed in a build environment, you can build images using Singularity Hub instead. If you want a more detailed rundown and examples for different build options, see our *singularity flow* page.

CHAPTER

TWO

INTRODUCTION

This document will introduce you to Singularity, If you are viewing this on the web there should be links in the bar to the left that will direct you to other important topics. If you want to get a quick overview, see our *quickstart*. If you want to understand which commands are a best fit for your use case, see our build flow section. There is also a separate Singularity Administration Guide that targets system administrators, so if you are a service provider, or an interested user, it is encouraged that you read that document as well.

2.1 Welcome to Singularity!

Singularity is a container solution created by necessity for scientific and application driven workloads. Over the past decade and a half, virtualization has gone from an engineering toy to a global infrastructure necessity and the evolution of enabling technologies has flourished. Most recently, we have seen the introduction of the latest spin on virtualization… "containers". People tend to view containers in light of their virtual machine ancestry and these preconceptions influence feature sets and expected use cases. This is both a good and a bad thing… For industry and enterprise-centric container technologies this is a good thing. Web enabled cloud requirements are very much in alignment with the feature set of virtual machines, and thus the preceding container technologies. But the idea of containers as miniature virtual machines is a bad thing for the scientific world and specifically the high performance computation (HPC) community. While there are many overlapping requirements in these two fields, they differ in ways that make a shared implementation generally incompatible. Some groups have leveraged custom-built resources that can operate on a lower performance scale, but proper integration is difficult and perhaps impossible with today's technology. Many scientists could benefit greatly by using container technology, but they need a feature set that differs somewhat from that available with current container technology. This necessity drives the creation of Singularity and articulated its four primary functions:

2.1.1 Mobility of Compute

Mobility of compute is defined as the ability to define, create and maintain a workflow and be confident that the workflow can be executed on different hosts, operating systems (as long as it is Linux) and service providers. Being able to contain the entire software stack, from data files to library stack, and portably move it from system to system means true mobility. Singularity achieves this by utilizing a distributable image format that contains the entire container and stack into a single file. This file can be copied, shared, archived, and standard UNIX file permissions also apply. Additionally containers are portable (even across different C library versions and implementations) which makes sharing and copying an image as easy as `cp` or `scp` or `ftp`.

2.1.2 Reproducibility

As mentioned above, Singularity containers utilize a single file which is the complete representation of all the files within the container. The same features which facilitate mobility also facilitate reproducibility. Once a contained

workflow has been defined, a snapshot image can be taken of a container. The image can be then archived and locked down such that it can be used later and you can be confident that the code within the container has not changed.

2.1.3 User Freedom

System integrators, administrators, and engineers spend a lot of effort maintaining their systems, and tend to take a cautious approach. As a result, it is common to see hosts installed with production, mission critical operating systems that are "old" and have few installed packages. Users may find software or libraries that are too old or incompatible with the software they must run, or the environment may just lack the software stack they need due to complexities with building, specific software knowledge, incompatibilities or conflicts with other installed programs.

Singularity can give the user the freedom they need to install the applications, versions, and dependencies for their workflows without impacting the system in any way. Users can define their own working environment and literally copy that environment image (single file) to a shared resource, and run their workflow inside that image.

2.1.4 Support on Existing Traditional HPC

Replicating a virtual machine cloud like environment within an existing HPC resource is not a reasonable goal for many administrators. There are a lots of container systems available which are designed for enterprise, as a replacement for virtual machines, are cloud focused, or require unstable or unavailable kernel features. Singularity supports existing and traditional HPC resources as easily as installing a single package onto the host operating system. Custom configurations may be achieved via a single configuration file, and the defaults are tuned to be generally applicable for shared environments. Singularity can run on host Linux distributions from RHEL6 (RHEL5 for versions lower than 2.2) and similar vintages, and the contained images have been tested as far back as Linux 2.2 (approximately 14 years old). Singularity natively supports InfiniBand, Lustre, and works seamlessly with all resource managers (e.g. SLURM, Torque, SGE, etc.) because it works like running any other command on the system. It also has built-in support for MPI and for containers that need to leverage GPU resources.

2.2 A High Level View of Singularity

2.2.1 Security and privilege escalation

A user inside a Singularity container is the same user as outside the container This is one of Singularities defining characteristics. It allows a user (that may already have shell access to a particular host) to simply run a command inside of a container image as themselves. The following example scenario will help illustrate a Singularity container:

%SERVER and %CLUSTER are large expensive systems with resources far exceeding those of my personal workstation. But because they are shared systems, no users have root access. The environments are tightly controlled and managed by a staff of system administrators. To keep these systems secure, only the system administrators are granted root access and they control the state of the operating systems and installed applications. If a user is able to escalate to root (even within a container) on %SERVER or %CLUSTER, they can do bad things to the network, cause denial of service to the host (as well as other hosts on the same network), and may have unrestricted access to file systems reachable by the container.

To mitigate security concerns like this, Singularity limits one's ability to escalate permission inside a container. For example, if I do not have root access on the target system, I should not be able to escalate my privileges within the container to root either. This is semi-antagonistic to Singularity's 3rd tenant; allowing the users to have freedom of their own environments. Because if a user has the freedom to create and manipulate their own container environment, surely they know how to escalate their privileges to root within that container. Possible means to escalation could include setting the root user's password, or enabling themselves to have sudo access. For these reasons, Singularity prevents user context escalation within the container, and thus makes it possible to run user supplied containers on shared infrastructures. This mitigation dictates the *Singularity workflow*. If a user needs to be root in order to make

changes to their containers, then they need to have an endpoint (a local workstation, laptop, or server) where they have root access. Considering almost everybody at least has a laptop, this is not an unreasonable or unmanageable mitigation, but it must be defined and articulated.

2.2.2 The Singularity container image

Singularity makes use of a container image file, which physically includes the container. This file is a physical representation of the container environment itself. If you obtain an interactive shell within a Singularity container, you are literally running within that file. This design simplifies management of files to the element of least surprise, basic file permission. If you either own a container image, or have read access to that container image, you can start a shell inside that image. If you wish to disable or limit access to a shared image, you simply change the permission ACLs to that file. There are numerous benefits for using a single file image for the entire container:

- Copying or branching an entire container is as simple as `cp`
- Permission/access to the container is managed via standard file system permissions
- Large scale performance (especially over parallel file systems) is very efficient
- No caching of the image contents to run (especially nice on clusters)
- Containers are compressed and consume very little disk space
- Images can serve as stand-alone programs, and can be executed like any other program on the host

2.2.2.1 *Copying, sharing, branching, and distributing your image*

A primary goal of Singularity is mobility. The single file image format makes mobility easy. Because Singularity images are single files, they are easily copied and managed. You can copy the image to create a branch, share the image and distribute the image as easily as copying any other file you control!

If you want an automated solution for building and hosting your image, you can use our container registry Singularity Hub. Singularity Hub can automatically build *Singularity recipe files* from a GitHub repository each time that you push. It provides a simple cloud solution for storing and sharing your image. If you want to host your own Registry, then you should check out Singularity Registry. If you have ideas or suggestions for how Singularity can better support reproducible science, please reach out!.

2.2.2.2 *Supported container formats*

- **squashfs**: the default container format is a compressed read-only file system that is widely used for things like live CDs/USBs and cell phone OS's
- **ext3**: (also called `writable`) a writable image file containing an ext3 file system that was the default container format prior to Singularity version 2.4
- **directory**: (also called `sandbox`) standard Unix directory containing a root container image
- **tar.gz**: zlib compressed tar archive
- **tar.bz2**: bzip2 compressed tar archive
- **tar**: uncompressed tar archive

2.2.2.3 *Supported Uniform Resource Identifiers (URI)*

Singularity also supports several different mechanisms for obtaining the images using a standard URI format.

- **shub://** Singularity Hub is our own registry for Singularity containers. If you want to publish a container, or give easy access to others from their command line, or enable automatic builds, you should build it on Singularity Hub.
- **docker://** Singularity can pull Docker images from a Docker registry, and will run them non-persistently (e.g. changes are not persisted as they can not be saved upstream). Note that pulling a Docker image implies assembling layers at runtime, and two subsequent pulls are not guaranteed to produce an identical image.
- **instance://** A Singularity container running as service, called an instance, can be referenced with this URI.

2.2.3 Name-spaces and isolation

When asked, "What namespaces does Singularity virtualize?", the most appropriate response from a Singularity point of view is "As few as possible!". This is because the goals of Singularity are mobility, reproducibility and freedom, not full isolation (as you would expect from industry driven container technologies). Singularity only separates the needed namespaces in order to satisfy our primary goals.

Coupling incomplete isolation with the fact that a user inside a container is the same user outside the container, allows Singularity to blur the lines between a container and the underlying host system. Using Singularity feels like running in a parallel universe, where there are two time lines. In one time line, the system administrators installed their operating system of choice. But on an alternate time line, we bribed the system administrators and they installed our favorite operating system and apps, and gave us full control but configured the rest of the system identically. And Singularity gives us the power to pick between these two time lines. In other words, Singularity allows you to virtually swap out the underlying operating system for one that you've defined without affecting anything else on the system and still having all of the host resources available to you.

The container is similar to logging into another identical host running a different operating system. e.g. One moment you are on Centos-6 and the next minute you are on the latest version of Ubuntu that has Tensorflow installed, or Debian with the latest OpenFoam, or a custom workflow that you installed. But you are still the same user with the same files running the same PIDs. Additionally, the selection of name-space virtualization can be dynamic or conditional. For example, the PID namespace is not separated from the host by default, but if you want to separate it, you can with a command line (or environment variable) setting. You can also decide if you want to contain a process so it can not reach out to the host file system if you don't know if you trust the image. But by default, you are allowed to interface with all of the resources, devices and network inside the container as you are outside the container (given your level of permissions).

2.2.4 Compatibility with standard work-flows, pipes and IO

Singularity abstracts the complications of running an application in an environment that differs from the host. For example, applications or scripts within a Singularity container can easily be part of a pipeline that is being executed on the host. Singularity containers can also be executed from a batch script or other program (e.g. an HPC system's resource manager) natively. Some usage examples of Singularity can be seen as follows:

```
$ singularity exec dummy.img xterm  # run xterm from within the container

$ singularity exec dummy.img python script.py  # run a script on the host system␣
↪using container's python

$ singularity exec dummy.img python < /path/to/python/script.py  # do the same via␣
↪redirection

$ cat /path/to/python/script.py | singularity exec dummy.img python  # do the same␣
↪via a pipe
```

You can even run MPI executables within the container as simply as:

```
$ mpirun -np X singularity exec /path/to/container.img /usr/bin/mpi_program_inside_
↪container (mpi program args)
```

2.2.5 The Singularity Process Flow

When executing container commands, the Singularity process flow can be generalized as follows:

1. Singularity application is invoked
2. Global options are parsed and activated
3. The Singularity command (subcommand) process is activated
4. Subcommand options are parsed
5. The appropriate sanity checks are made
6. Environment variables are set
7. The Singularity Execution binary is called (`sexec`)
8. Sexec determines if it is running privileged and calls the `SUID` code if necessary
9. Namespaces are created depending on configuration and process requirements
10. The Singularity image is checked, parsed, and mounted in the namespace
11. Bind mount points are setup so that files on the host are visible in the `CLONE_NEWNS` container
12. The namespace `CLONE_FS` is used to virtualize a new root file system
13. Singularity calls `execvp()` and Singularity process itself is replaced by the process inside the container
14. When the process inside the container exits, all namespaces collapse with that process, leaving a clean system

All of the above steps are fast and seem instantaneous to the user (i.e. there is little run-time overhead), when measured on a standard server the overhead was approximately 15-25 thousandths of a second).

2.3 The Singularity Usage Workflow

The security model of Singularity (as described above, *"A user inside a Singularity container is the same user as outside the container"*) defines the Singularity workflow. There are generally two groups of actions you must implement on a container; management (building your container) and usage.

In many circumstances building containers require root administrative privileges just like these actions would require on any system, container, or virtual machine. This means that a user must have access to a system on which they have root privileges. This could be a server, workstation, a laptop, virtual machine, or even a cloud instance. If you are using OS X or Windows on your laptop, it is recommended to setup Vagrant, and run Singularity from there (there are recipes for this which can be found at the Singularity website. Once you have Singularity installed on your endpoint of choice, this is where you will do the bulk of your container development. This workflow can be described visually as follows:

On the left side, you have your build environment: a laptop, workstation, or a server that you control. Here you will (optionally):

1. develop and test containers using `--sandbox` (build into a writable directory) or `--writable` (build into a writable ext3 image)
2. build your production containers with a squashfs filesystem.

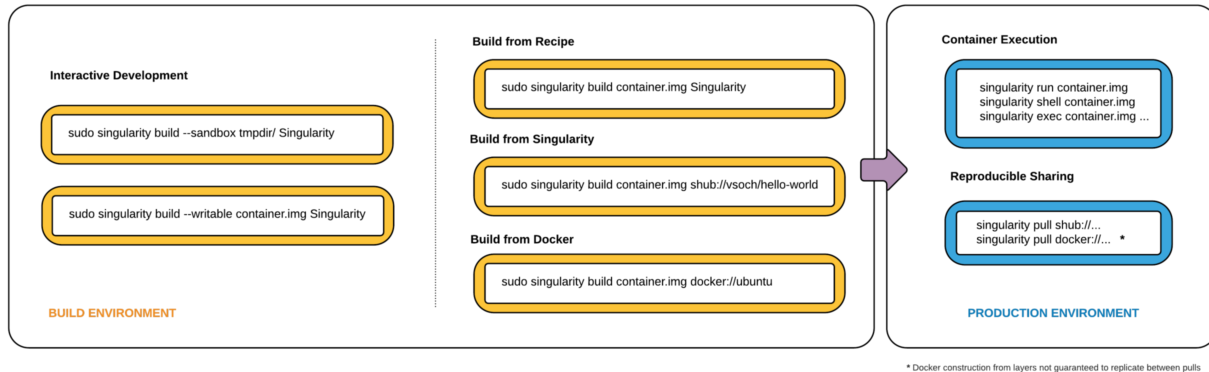

Fig. 1: Singularity workflow

Once you have the container with the necessary applications, libraries and data inside it can be easily shared to other hosts and executed without requiring root access. A production container should be an immutable object, so if you need to make changes to your container you should go back to your build system with root privileges, rebuild the container with the necessary changes, and then re-upload the container to the production system where you wish to run it.

2.3.1 Singularity Commands

How do the commands work?

The following is a list of the Singularity commands, Click on each command for more information.

- *build* : Build a container on your user endpoint or build environment
- *exec* : Execute a command to your container
- *inspect* : See labels, run and test scripts, and environment variables
- *pull* : pull an image from Docker or Singularity Hub
- *run* : Run your image as an executable
- *shell* : Shell into your image

Image Commands

- *image.import* : import layers or other file content to your image
- *image.export* : export the contents of the image to tar or stream
- *image.create* : create a new image, using the old ext3 filesystem
- *image.expand* : increase the size of your image (old ext3)

Instance Commands
Instances were added in 2.4. This list is brief, and likely to expand with further development.

- *instances* : Start, stop, and list container instances

Deprecated Commands The following commands are deprecated in 2.4 and will be removed in future releases.

- *bootstrap* : Bootstrap a container recipe

2.4 Support

Have a question, or need further information? Reach out to us.

2.5 About

2.5.1 Overview

While there are many container solutions being used commonly in this day and age, what makes Singularity different stems from it's primary design features and thus it's architecture:

1. **Reproducible software stacks:** These must be easily verifiable via checksum or cryptographic signature in such a manner that does not change formats (e.g. splatting a tarball out to disk). By default Singularity uses a container image file which can be checksummed, signed, and thus easily verified and/or validated.
2. **Mobility of compute:** Singularity must be able to transfer (and store) containers in a manner that works with standard data mobility tools (rsync, scp, gridftp, http, NFS, etc..) and maintain software and data controls compliancy (e.g. HIPAA, nuclear, export, classified, etc..)
3. **Compatibility with complicated architectures:** The runtime must be immediately compatible with existing HPC, scientific, compute farm and even enterprise architectures any of which maybe running legacy kernel versions (including RHEL6 vintage systems) which do not support advanced namespace features (e.g. the user namespace)
4. **Security model:** Unlike many other container systems designed to support trusted users running trusted containers we must support the opposite model of untrusted users running untrusted containers. This changes the security paradigm considerably and increases the breadth of use cases we can support.

2.5.2 Background

A Unix operating system is broken into two primary components, the kernel space, and the user space. The Kernel supports the user space by interfacing with the hardware, providing core system features and creating the software compatibility layers for the user space. The user space on the other hand is the environment that most people are most familiar with interfacing with. It is where applications, libraries and system services run.

Containers are shifting the emphasis away from the runtime environment by commoditizing the user space into swappable components. This means that the entire user space portion of a Linux operating system, including programs, custom configurations, and environment can be interchanged at runtime. Singularity emphasis and simplifies the distribution vector of containers to be that of a single, verifiable file.

Software developers can now build their stack onto whatever operating system base fits their needs best, and create distributable runtime encapsulated environments and the users never have to worry about dependencies, requirements, or anything else from the user space.

Singularity provides the functionality of a virtual machine, without the heavyweight implementation and performance costs of emulation and redundancy!

2.5.2.1 The Singularity Solution

Singularity has two primary roles:

1. **Container Image Generator:** Singularity supports building different container image formats from scratch using your choice of Linux distribution bases or leveraging other container formats (e.g. Docker Hub). Container formats supported are the default compressed immutable (read only) image files, writable raw file system based images, and sandboxes (chroot style directories).
2. **Container Runtime:** The Singularity runtime is designed to leverage the above mentioned container formats and support the concept of untrusted users running untrusted containers. This counters the typical container runtime practice of trusted users running trusted containers and as a result of that, Singularity utilizes a very different security paradigm. This is a required feature for implementation within any multi-user environment.

The Singularity containers themselves are purpose built and can include a simple application and library stack or a complicated work flow that can interface with the hosts resources directly or run isolated from the host and other containers. You can even launch a contained work flow by executing the image file directly! For example, assuming that `~/bin` is in the user's path as it is normally by default:

```
$ mkdir ~/bin

$ singularity build ~/bin/python-latest docker://python:latest

Docker image path: index.docker.io/library/python:latest

Cache folder set to /home/gmk/.singularity/docker

Importing: base Singularity environment

Importing: /home/gmk/.singularity/docker/
↪sha256:aa18ad1a0d334d80981104c599fa8cef9264552a265b1197af11274beba767cf.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:15a33158a1367c7c4103c89ae66e8f4fdec4ada6a39d4648cf254b32296d6668.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:f67323742a64d3540e24632f6d77dfb02e72301c00d1e9a3c28e0ef15478fff9.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:c4b45e832c38de44fbab83d5fcf9cbf66d069a51e6462d89ccc050051f25926d.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:b71152c33fd217d4408c0e7a2f308e66c0be1a58f4af9069be66b8e97f7534d2.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:c3eac66dc8f6ae3983a7f37e3da84a8acb828faf909be2d6649e9d7c9caffc28.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:494ffdf1660cdec946ae13d6b726debbcec4c393a7eecfabe97caf3961f62c36.tar.gz

Importing: /home/gmk/.singularity/docker/
↪sha256:f5ec737c23de3b1ae2b1ce3dce1fd20e0cb246e4c73584dcd4f9d2f50063324e.tar.gz

Importing: /home/gmk/.singularity/metadata/
↪sha256:5dd22488ce22f06bed1042cc03d3efa5a7d68f2a7b3dcad559df4520154ef9c2.tar.gz

WARNING: Building container as an unprivileged user. If you run this container as root

WARNING: it may be missing some functionality.

Building Singularity image...
```

(continues on next page)

(continued from previous page)

```
Cleaning up...

Singularity container built: /home/gmk/bin/python-latest


$ which python-latest

/home/gmk/bin/python-latest


$ python-latest --version

Python 3.6.3


$ singularity exec ~/bin/python-latest cat /etc/debian_version

8.9

$ singularity shell ~/bin/python-latest

Singularity: Invoking an interactive shell within container...


Singularity python-latest:~>
```

Additionally, Singularity blocks privilege escalation within the container and you are always yourself within a container! If you want to be root inside the container, you first must be root outside the container. This simple usage paradigm mitigates many of the security concerns that exists with containers on multi-user shared resources. You can directly call programs inside the container from outside the container fully incorporating pipes, standard IO, file system access, X11, and MPI. Singularity images can be seamlessly incorporated into your environment.

2.5.2.2 Portability and Reproducibility

Singularity containers are designed to be as portable as possible, spanning many flavors and vintages of Linux. The only known limitation is binary compatibility of the kernel and container. Singularity has been ported to distributions going as far back as RHEL 5 (and compatibles) and works on all currently living versions of RHEL, Debian, Arch, Alpine, Gentoo and Slackware. Within the container, there are almost no limitations aside from basic binary compatibility.

Inside the container, it is also possible to have a very old version of Linux supported. The oldest known version of Linux tested was a Red Hat Linux 8 container, that was converted by hand from a physical computer's hard drive as the 15 year old hardware was failing. The container was transferred to a new installation of Centos7, and is still running in production!

Each Singularity image includes all of the application's necessary run-time libraries and can even include the required data and files for a particular application to run. This encapsulation of the entire user-space environment facilitates not only portability but also reproducibility.

2.5.3 Features

2.5.3.1 Encapsulation of the environment

Mobility of Compute is the encapsulation of an environment in such a manner to make it portable between systems. This operating system environment can contain the necessary applications for a particular work-flow, development tools, and/or raw data. Once this environment has been developed it can be easily copied and run from any other Linux system.

This allows users to BYOE (Bring Their Own Environment) and work within that environment anywhere that Singularity is installed. From a service provider's perspective we can easily allow users the flexibility of "cloud"-like environments enabling custom requirements and workflows.

Additionally there is always a misalignment between development and production environments. The service provider can only offer a stable, secure tuned production environment which in many times will not keep up with the fast paced requirements of developers. With Singularity, you can control your own development environment and simply copy them to the production resources.

2.5.3.2 Containers are image based

Using image files have several key benefits:

First, this image serves as a vector for mobility while retaining permissions of the files within the image. For example, a user may own the image file so they can copy the image to and from system to system. But, files within an image must be owned by the appropriate user. For example, '/etc/passwd' and '/' must be owned by root to achieve appropriate access permission. These permissions are maintained within a user owned image.

There is never a need to build, rebuild, or cache an image! All IO happens on an as needed basis. The overhead in starting a container is in the thousandths of a second because there is never a need to pull, build or cache anything!

On HPC systems a single image file optimizes the benefits of a shared parallel file system! There is a single metadata lookup for the image itself, and the subsequent IO is all directed to the storage servers themselves. Compare this to the massive amount of metadata IO that would be required if the container's root file system was in a directory structure. It is not uncommon for large Python jobs to DDOS (distributed denial of service) a parallel meta-data server for minutes! The Singularity image mitigates this considerably.

2.5.3.3 No user contextual changes or root escalation allowed

When Singularity is executed, the calling user is maintained within the container. For example, if user 'gmk' starts a Singularity container, the same user 'gmk' will end up within the container. If 'root' starts the container, 'root' will be the user inside the container.

Singularity also limits a user's ability to escalate privileges within the container. Even if the user works in their own environment where they configured 'sudo' or even removed root's password, they will not be able to 'sudo' or 'su' to root. If you want to be root inside the container, you must first be root outside the container.

Because of this model, it becomes possible to blur the line of access between what is contained and what is on the host as Singularity does not grant the user any more access than they already have. It also enables the implementation on shared/multi-tenant resources.

2.5.3.4 No root owned daemon processes

Singularity does not utilize a daemon process to manage the containers. While daemon processes do facilitate certain types of workflows and privilege escalation, it breaks all resource controlled environments. This is because a user's

job becomes a subprocess of the daemon (rather than the user's shell) and the daemon process is outside of the reach of a resource manager or batch scheduler.

Additionally, securing a root owned daemon process which is designed to manipulate the host's environment becomes tricky. In currently implemented models, it is possible to grant permissions to users to control the daemon, or not. There is no sense of ACL's or access of what users can and can not do.

While there are some other container implementations that do not leverage a daemon, they lack other features necessary to be considered as reasonable user facing solution without having root access. For example, there has been a standing unimplemented patch to RunC (already daemon-less) which allows for root-less usage (no root). But, user contexts are not maintained, and it will only work with chroot directories (instead of an image) where files must be owned and manipulated by the root user!

2.5.4 Use Cases

2.5.4.1 BYOE: Bring Your Own Environment

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

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

2.5.4.2 Reproducible science

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

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

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

2.5.4.3 Static environments (software appliances)

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

2.5.4.4 Legacy code on old operating systems

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

2.5.4.5 Complicated software stacks that are very host specific

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

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

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

2.5.5 License

Singularity is released under a standard 3 clause BSD license. Please see our LICENSE file for more details).

2.5.6 Getting started

Jump in and *get started*, or find ways to get help.

- Project lead: Gregory M. Kurtzer

CHAPTER

THREE

INSTALLATION

This document will guide you through the process of installing Singularity from source with the version and location of your choice.

3.1 Before you begin

If you have an earlier version of Singularity installed, you should remove it before executing the installation commands.

These instructions will build Singularity from source on your system. So you will need to have some development tools installed. If you run into missing dependencies, try installing them WITH `apt-get` or `yum/rpm` as shown below.

```
$ sudo apt-get update && \

    sudo apt-get install \

    python \

    dh-autoreconf \

    build-essential \

    libarchive-dev
```

```
$ sudo yum update && \

    sudo yum groupinstall 'Development Tools' && \

    sudo yum install libarchive-devel
```

3.2 Install from a tag

The following commands will install a tagged version of the GitHub repo to `/usr/local`. This will work for pre 3.0 tags.

```
$ git clone https://github.com/sylabs/singularity.git

$ cd singularity

$ git fetch --all
```

(continues on next page)

(continued from previous page)

```
$ git tag -l

$ git checkout [TAG]

$ ./autogen.sh

$ ./configure --prefix=/usr/local --sysconfdir=/etc

$ make

$ sudo make install
```

Singularity will be installed in the `/usr/local` directory hierarchy by default. And if you specify a custom directory with the `--prefix` option, all of Singularity's binaries and the configuration file will be installed within that directory. This last option can be useful if you want to install multiple versions of Singularity, install Singularity on a shared system, or if you want to remove Singularity easily after installing it.

If you omit the `--sysconfdir` option , the configuration file will be installed in `/usr/local/etc`. If you omit the `--prefix` option, Singularity will be installed in the `/usr/local` directory hierarchy by default. And if you specify a custom directory with the `--prefix` option, all of Singularity's binaries and the configuration file will be installed within that directory.

3.3 Install a specific release

The following commands will install a specific release from GitHub releases page to `/usr/local`.

```
$ VER=2.5.1

$ wget https://github.com/sylabs/singularity/releases/download/$VER/singularity-$VER.
↪tar.gz

$ tar xvf singularity-$VER.tar.gz

$ cd singularity-$VER

$ ./configure --prefix=/usr/local --sysconfdir=/etc

$ make

$ sudo make install
```

3.4 Install the development branch

The primary development of Singularity now happens on the `master` branch. Please see the `INSTALL.md` file in a copy of the repository.

3.5 Remove an old version

Let's say that we installed Singularity to `/usr/local`. To remove it completely, you need to hit all of the following:

```
$ sudo rm -rf /usr/local/libexec/singularity

$ sudo rm -rf /usr/local/etc/singularity

$ sudo rm -rf /usr/local/include/singularity

$ sudo rm -rf /usr/local/lib/singularity

$ sudo rm -rf /usr/local/var/lib/singularity/

$ sudo rm /usr/local/bin/singularity

$ sudo rm /usr/local/bin/run-singularity

$ sudo rm /usr/local/etc/bash_completion.d/singularity

$ sudo rm /usr/local/man/man1/singularity.1
```

If you modified the system configuration directory, remove the `singularity.conf` file there as well. If you installed Singularity in a custom directory, you need only remove that directory to uninstall Singularity. For instance if you installed singularity with the `--prefix=/some/temp/dir` option argument pair, you can remove Singularity like so:

```
$ sudo rm -rf /some/temp/dir
```

What should you do next? You can check out the *quickstart* guide, or learn how to interact with your container via the *shell* , *exec* , or *run* commands. Or click **next** below to continue reading.

3.6 Install on Windows

3.6.1 Setup

First, install the following software:

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

3.6.2 Singularity Vagrant Box

We are maintaining a set of Vagrant Boxes via Vagrant Cloud, one of Hashicorp many tools that likely you've used and haven't known it. The current stable version of Singularity is available here:

- sylabs/singularity-2.4

For other versions of Singularity see our Vagrant Cloud repository

Run GitBash. The default home directory will be C:Usersyour_username

```
mkdir singularity-2.4

cd singularity-2.4
```

Note that if you had installed a previous version of the vm (and are using the same folder), you must destroy it first. In our example we create a new folder. To destroy a previous vm:

```
vagrant destroy
```

Then issue the following commands to bring up the Virtual Machine:

```
vagrant init singularityware/singularity-2.4

vagrant up

vagrant ssh
```

You are then ready to go with Singularity 2.4!

```
vagrant@vagrant:~$ which singularity

/usr/local/bin/singularity

vagrant@vagrant:~$ singularity --version

2.4-dist


vagrant@vagrant:~$ sudo singularity build growl-llo-world.simg shub://vsoch/hello-
↪world

Cache folder set to /root/.singularity/shub

Progress |===================================| 100.0%

Building from local image: /root/.singularity/shub/vsoch-hello-world-master.simg

Building Singularity image...

Singularity container built: growl-llo-world.simg

Cleaning up...

vagrant@vagrant:~$ ./growl-llo-world.simg

RaawwWWWWWRRRR!!
```

Note that when you do `vagrant up` you can also select the provider, if you use vagrant for multiple providers. For example:

```
vagrant up --provider virtualbox
```

although this isn't entirely necessary if you only have it configured for virtualbox.

3.7 Install on Linux

3.7.1 Installation from Source

You can try the following two options:

3.7.1.1 Option 1: Download latest stable release

You can always download the latest tarball release from GitHub

For example, here is how to download version `2.5.2` and install:

```
VERSION=2.5.2

wget https://github.com/sylabs/singularity/releases/download/$VERSION/singularity-
↪$VERSION.tar.gz

tar xvf singularity-$VERSION.tar.gz

cd singularity-$VERSION

./configure --prefix=/usr/local

make

sudo make install
```

Note that when you configure, `squashfs-tools` is **not** required, however it is required for full functionality. You will see this message after the configuration:

```
mksquashfs from squash-tools is required for full functionality
```

If you choose not to install `squashfs-tools`, you will hit an error when you try a pull from Docker Hub, for example.

3.7.1.2 Option 2: Download the latest development code

To download the most recent development code, you should use Git and do the following:

```
git clone https://github.com/sylabs/singularity.git

cd singularity

./autogen.sh

./configure --prefix=/usr/local

make

sudo make install
```

Note: The 'make install' is required to be run as root to get a properly installed Singularity implementation. If you do not run it as root, you will only be able to launch Singularity as root due to permission limitations.

3.7.1.3 Prefix in special characters

If you build Singularity with a non-standard `--prefix` argument, please be sure to review the admin guide for details regarding the `--localstatedir` variable. This is especially important in environments utilizing shared filesystems.

3.7.1.4 Updating

To update your Singularity version, you might want to first delete the executables for the old version:

```
sudo rm -rf /usr/local/libexec/singularity
```

And then install using one of the methods above.

3.7.2 Debian Ubuntu Package

Singularity is available on Debian (and Ubuntu) systems starting with Debian stretch and the Ubuntu 16.10 yakkety releases. The package is called `singularity-container`. For recent releases of singularity and backports for older Debian and Ubuntu releases, we recommend that you use the NeuroDebian repository.

3.7.2.1 Testing first with Docker

If you want a quick preview of the NeuroDebian mirror, you can do this most easily with the NeuroDebian Docker image (and if you don't, skip to the next section). Obviously you should have Docker installed before you do this.

First we run the `neurodebian` Docker image:

```
$ docker run -it --rm neurodebian
```

Then we update the cache (very quietly), and look at the `singularity-container` policy provided:

```
$ apt-get update -qqq

$ apt-cache policy singularity-container

singularity-container:

  Installed: (none)

  Candidate: 2.3-1~nd80+1

  Version table:

    2.3-1~nd80+1 0

      500 http://neuro.debian.net/debian/ jessie/main amd64 Packages
```

You can continue working in Docker, or go back to your host and install Singularity.

3.7.2.2 Adding the Mirror and installing

You should first enable the NeuroDebian repository following instructions on the NeuroDebian site. This means using the dropdown menus to find the correct mirror for your operating system and location. For example, after selecting Ubuntu 16.04 and selecting a mirror in CA, I am 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
```

and then update

```
sudo apt-get update
```

then singularity can be installed as follows:

```
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
```

And for a user, it's probably well suited to use the defaults. For a cluster admin, we recommend that you read the admin docs to get a better understanding of the configuration file options available to you. Remember that you can always tweak the files at `/etc/singularity/singularity.conf` and `/etc/singularity/init` if you want to make changes.

After this install, you should confirm that `2.3-dist` is the version installed:

```
$ singularity --version

  2.4-dist
```

Note that if you don't add the NeuroDebian lists, the version provided will be old (e.g., 2.2.1). If you need a backport build of the recent release of Singularity on those or older releases of Debian and Ubuntu, you can see all the various builds and other information here.

3.7.3 Build an RPM from source

Like the above, you can build an RPM of Singularity so it can be more easily managed, upgraded and removed. From the base Singularity source directory do the following:

```
./autogen.sh

./configure

make dist

rpmbuild -ta singularity-*.tar.gz

sudo yum install ~/rpmbuild/RPMS/*/singularity-[0-9]*.rpm
```

Note: If you want to have the RPM install the files to an alternative location, you should define the environment variable 'PREFIX' to suit your needs, and use the following command to build:

```
PREFIX=/opt/singularity

rpmbuild -ta --define="_prefix $PREFIX" --define "_sysconfdir $PREFIX/etc" --define "_
↪defaultdocdir $PREFIX/share" singularity-*.tar.gz
```

When using `autogen.sh` If you get an error that you have packages missing, for example on Ubuntu 16.04:

```
./autogen.sh

+libtoolize -c

./autogen.sh: 13: ./autogen.sh: libtoolize: not found

+aclocal

./autogen.sh: 14: ./autogen.sh: aclocal: not found

+autoheader

./autogen.sh: 15: ./autogen.sh: autoheader: not found

+autoconf

./autogen.sh: 16: ./autogen.sh: autoconf: not found

+automake -ca -Wno-portability

./autogen.sh: 17: ./autogen.sh: automake: not found
```

then you need to install dependencies:

```
sudo apt-get install -y build-essential libtool autotools-dev automake autoconf
```

3.7.4 Build an DEB from source

To build a deb package for Debian/Ubuntu/LinuxMint invoke the following commands:

```
$ fakeroot dpkg-buildpackage -b -us -uc # sudo will ask for a password to run the␣
↪tests

$ sudo dpkg -i ../singularity-container_2.3_amd64.deb
```

Note that the tests will fail if singularity is not already installed on your system. This is the case when you run this procedure for the first time. In that case run the following sequence:

```
$ echo "echo SKIPPING TESTS THEYRE BROKEN" > ./test.sh

$ fakeroot dpkg-buildpackage -nc -b -us -uc # this will continue the previous build␣
↪without an initial 'make clean'
```

3.7.5 Install on your Cluster Resource

In the case that you want Singularity installed on a shared resource, you will need to talk to the administrator of the resource. Toward this goal, we've prepared a *helpful guide* that you can send to him or her. If you have unanswered questions, please reach out..

3.8 Install on Mac

This recipe demonstrates how to run Singularity on your Mac via Vagrant and Ubuntu. The recipe requires access to `brew` which is a package installation subsystem for OS X. This recipe may take anywhere from 5-20 minutes to complete.

3.8.1 Setup

First, install brew if you do not have it already.

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

Next, install Vagrant and the necessary bits.

```
brew cask install virtualbox

brew cask install vagrant

brew cask install vagrant-manager
```

3.8.2 Option 1: Singularity Vagrant Box

We are maintaining a set of Vagrant Boxes via Vagrant Cloud, one of Hashicorp many tools that likely you've used and haven't known it. The current stable version of Singularity is available here:

- singularityware/singularity-2.4

For other versions of Singularity see our Vagrant Cloud repository.

```
mkdir singularity-vm

cd singularity-vm
```

Note that if you have installed a previous version of the vm, you can either destroy it first, or create a new directory.

```
vagrant destroy
```

Then issue the following commands to bring up the Virtual Machine:

```
vagrant init singularityware/singularity-2.4

vagrant up

vagrant ssh
```

You are then ready to go with Singularity 2.4!

```
vagrant@vagrant:~$ which singularity

/usr/local/bin/singularity

vagrant@vagrant:~$ singularity --version

2.4-dist


vagrant@vagrant:~$ sudo singularity build growl-llo-world.simg shub://vsoch/hello-
↪world

Cache folder set to /root/.singularity/shub

Progress |===================================| 100.0%

Building from local image: /root/.singularity/shub/vsoch-hello-world-master.simg

Building Singularity image...

Singularity container built: growl-llo-world.simg

Cleaning up...

vagrant@vagrant:~$ ./growl-llo-world.simg

RaawwWWWWWRRRR!!
```

Note that when you do `vagrant up` you can also select the provider, if you use vagrant for multiple providers. For example:

```
vagrant up --provider virtualbox
```

although this isn't entirely necessary if you only have it configured for virtualbox.

3.8.3 Option 2: Vagrant Box from Scratch (more advanced alternative)

If you want to get more familiar with how Vagrant and VirtualBox work, you can instead build your own Vagrant Box from scratch. In this case, we will use the Vagrantfile for `bento/ubuntu-16.04`, however you could also try any of the other bento boxes that are equally delicious. As before, you should first make a separate directory for your Vagrantfile, and then init a base image.

```
mkdir singularity-2.4

cd singularity-2.4

vagrant init bento/ubuntu-16.04
```

Next, build and start the vagrant hosted VM, and you will install Singularity by sending the entire install script as a command (with the `-c` argument). You could just as easily shell into the box first with vagrant ssh, and then run these commands on your own. To each bento, his own.

```
vagrant up --provider virtualbox


# Run the necessary commands within the VM to install Singularity

vagrant ssh -c /bin/sh <<EOF

    sudo apt-get update

    sudo apt-get -y install build-essential curl git sudo man vim autoconf libtool

    git clone https://github.com/sylabs/singularity.git

    cd singularity

    ./autogen.sh

    ./configure --prefix=/usr/local

    make

    sudo make install

EOF
```

At this point, Singularity is installed in your Vagrant Ubuntu VM! Now you can use Singularity as you would normally by logging into the VM directly

```
vagrant ssh
```

Remember that the VM is running in the background because we started it via the command `vagrant up`. You can shut the VM down using the command `vagrant halt` when you no longer need it.

3.9 Requesting an Installation

3.9.1 How do I ask for Singularity on my local resource?

Installation of a new software is no small feat for a shared cluster resource. Whether you are an administrator reading this, or a user that wants a few talking points and background to share with your administrator, this document is for you. Here we provide you with some background and resources to learn about Singularity. We hope that this information will be useful to you in making the decision to build reproducible containers with Singularity

3.9.2 Information Resources

3.9.2.1 Background

- Frequently Asked Questions is a good first place to start for quick question and answer format.
- Singularity Publication: Reviews the history and rationale for development of the Software, along with comparison to other container software available at the time.
- Documentation Background is useful to read about use cases, and goals of the Software.

3.9.2.2 Security

- Administrator Control: The configuration file template is the best source to learn about the configuration options that are under the administrator's control.
- Security Overview discusses common security concerns

3.9.2.3 Presentations

- Contributed Content is a good source of presentations, tutorials, and links.

3.9.3 Installation Request

Putting all of the above together, a request might look like the following:

```
Dear Research Computing,


We are interested in having an installation of the Singularity software (https://
↪sylabs.github.io) installed on our cluster. Singularity containers will allow us to␣
↪build encapsulated environments, meaning that our work is reproducible and we are␣
↪empowered to choose all dependencies including libraries, operating system, and␣
↪custom software. Singularity is already installed on over 50 centers␣
↪internationally (http://singularity.lbl.gov/citation-registration) including TACC,␣
↪NIH,

and several National Labs, Universities, Hospitals. 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:
```

(continues on next page)

(continued from previous page)

```
  - 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/2.5.2/admin-guide/


If you have questions about any of the above, you can email the list (singularity@lbl.
↪gov) or join the slack channel (singularity-container.slack.com) to get a human␣
↪response. 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 letter above, you can always reach out to us for additional questions or support.

CHAPTER

FOUR

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 Singularity Hub and Docker Hub. You can use it to convert containers between the various formats supported by Singularity. And you can use it in conjunction with a *Singularity recipe* file to create a container from scratch and customized it to fit your needs.

4.1 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 **shub://** to build from Singularity Hub
- URI beginning with **docker://** to build from Docker Hub
- path to a **existing container** on your local machine
- path to a **directory** to build from a sandbox
- path to an **archive** in .tar or compressed .tar.gz format
- path to a *Singularity recipe file*

In addition `build` can produce containers in three different formats. Formats types can be specified by passing the following options to build.

- compressed read-only **squashfs** file system suitable for production (default)
- writable **ext3** file system suitable for interactive development (`--writable` option)
- 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 any of these three formats you can convert existing containers from one format to another.

The following diagram illustrates the targets that can be supplied to `build` as inputs and the containers `build` can produce as outputs. Green arrows represent operations that can be carried out without root privileges (though the container may not perform properly when run as root). Red arrows represent operations that must be carried out with root privileges.

4.2 Downloading a existing container from Singularity Hub

You can use the build command to download a container from Singularity Hub.

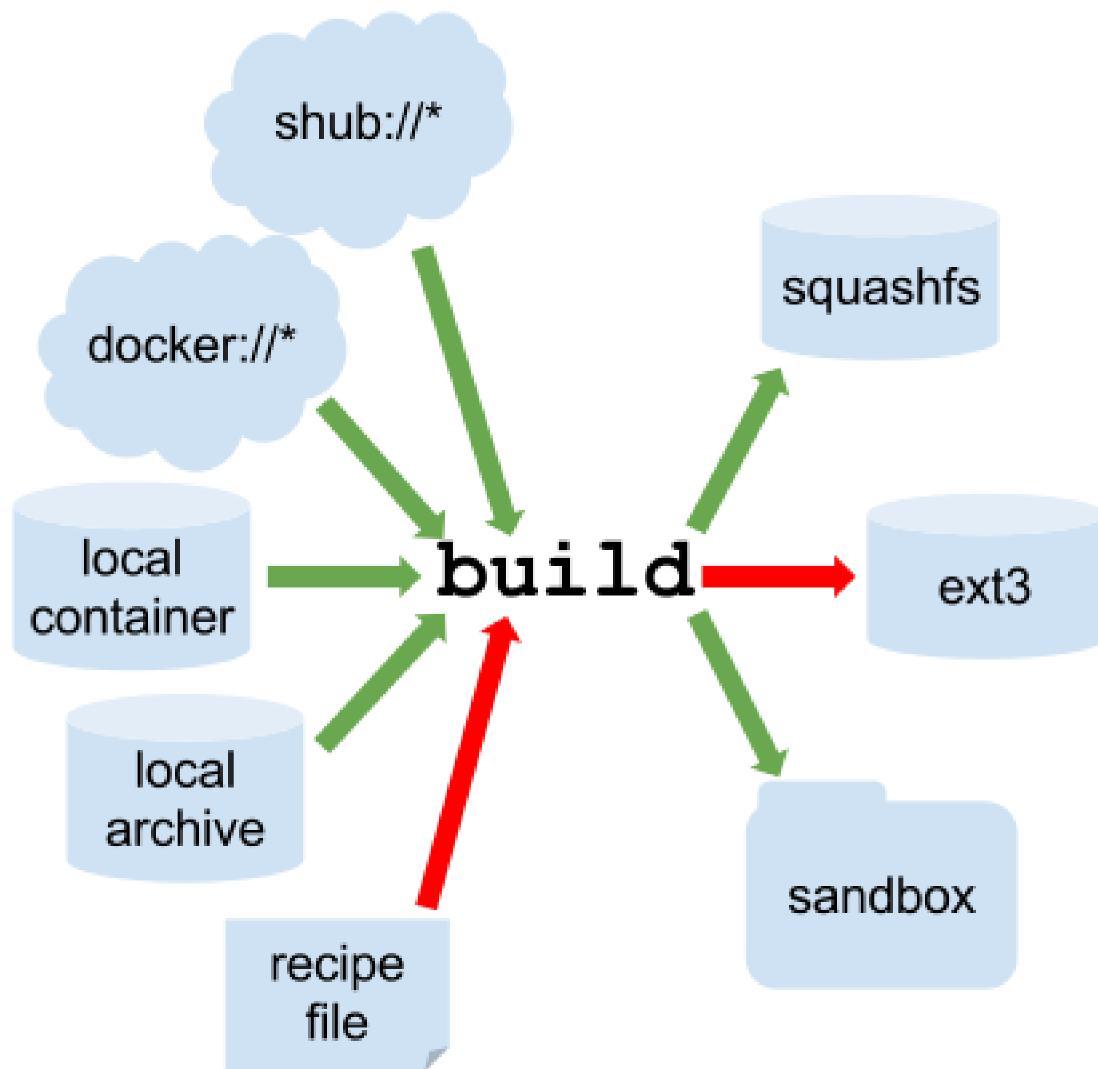

Fig. 1: Singularity build process

```
$ singularity build lolcow.simg shub://GodloveD/lolcow
```

The first argument (`lolvow.simg`) specifies a path and name for your container. The second argument (`shub://GodloveD/lolcow`) gives the Singularity Hub URI from which to download. But default the container will be converted to a compressed, read-only squashfs file. If you want your container in a different format use the `--writable` or `--sandbox` options.

4.3 Downloading a existing container from Docker Hub

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

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

4.4 Creating `--writable` images and `--sandbox` directories

4.4.1 `--writable`

If you wanted to create a writable ext3 image similar to those used by Singularity version < 2.4, you could do so with the `--writable` option. You must create writable containers as root.

Extending the Singularity Hub example from above:

```
$ sudo singularity build --writable lolcow.img shub://GodloveD/lolcow
```

The resulting container is writable, but is still mounted as read-only when executed with commands such as `run`, `exec`, and `shell`. To mount the container as read-write when using these commands add the `--writable` option to them as well.

To ensure that you have the proper permissions to write to the container as you like, it is also a good idea to make changes as root. For example:

```
$ sudo singularity shell --writable lolcow.img
```

4.4.2 `--sandbox`

If you wanted to create a container within a writable directory (called a sandbox) you could 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/ shub://GodloveD/lolcow
```

The resulting directory operates just like a container in an image file. You are permitted to make changes and write files within the directory, but those changes will not persist when you are finished using the container. To make your changes persistent, use the `--writable` flag when you invoke your container. Once again, 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/
```

4.5 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 squashfs container called `production.simg` and wanted to convert it to a writable ext3 container called `development.img` you could:

```
$ sudo singularity build --writable development.img production.simg
```

Similarly, to convert it to a writable directory (a sandbox):

```
$ singularity build --sandbox development/ production.simg
```

If you omit any options you can also convert your sandbox back to a read-only compressed squashfs image suitable for use in a production environment:

```
$ singularity build production2 development/
```

You can convert the three supported container formats using any combination.

Use care when converting writable ext3 images or sandbox directories to the default squashfs file format. If changes were made to the writable container before conversion, there is no record of those changes in the Singularity recipe file rendering your container non-reproducible. It is a best practice to build your immutable production containers directly from a Singularity recipe file instead.

4.6 Building containers from Singularity recipe files

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

```
Bootstrap: docker

From: ubuntu:16.04


%post

    apt-get -y update

    apt-get -y install fortune cowsay lolcat


%environment

    export LC_ALL=C

    export PATH=/usr/games:$PATH


%runscript

    fortune | cowsay | lolcat
```

You can do so with the following command.

```
$ sudo singularity build lolcow.simg Singularity
```

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

4.6.1 --force

You can build into the same 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 container, the `build` command will skip the steps involved in adding a new base. You can override this default with the `--force` option requiring that a new base OS is bootstrapped into the existing container. This behavior does not delete the existing OS, it just adds the new OS on top of the existing one.

Use care with this option: you may get results that you did not expect.

4.6.2 --section

If you only want to build a single section of your Singularity recipe file use the `--section` option. For instance, if you have edited the `%environment` section of a long Singularity recipe and don't want to completely re-build the container, you could re-build only the `%environment` section like so:

```
$ sudo singularity build --section environment image.simg Singularity
```

Under normal build conditions, the Singularity recipe 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.

4.6.3 --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.

```
$ sudo singularity build GPU.simg --notest Singularity
```

4.6.4 --checks

Checks are a new feature (in 2.4) that offer an easy way for an admin to define a security (or any other kind of check) to be run on demand for a Singularity image. They are defined (and run) via different tags.

```
CHECKS OPTIONS:

    -c|--checks    enable checks

    -t|--tag       specify a check tag (not default)

    -l|--low       Specify low threshold (all checks, default)

    -m|--med       Perform medium and high checks

    -h|--high      Perform only checks at level high
```

When you add the `--checks` option along with applicable tags to the `build` command Singularity will run the desired checks on your container at build time. See `singularity check --help` for available tags.

4.7 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 Singularity Hub, (because you don't have root access on a Linux machine or want to host your container on the cloud) check out this guide

CHAPTER

FIVE

BUILD ENVIRONMENT

It's commonly the case that you want to customize your build environment, such as specifying a custom cache directory for layers, or sending your Docker Credentials to the registry endpoint. Here we will discuss those topics.

5.1 Cache Folders

To make download of layers for build and *pull* faster and less redundant, we use a caching strategy. By default, the Singularity software will create a set of folders in your `$HOME` directory for docker layers, Singularity Hub images, and Docker metadata, respectively:

```
$HOME/.singularity

$HOME/.singularity/docker

$HOME/.singularity/shub

$HOME/.singularity/metadata
```

Fear not, you have control to customize this behavior! If you don't want the cache to be created (and a temporary directory will be used), set `SINGULARITY_DISABLE_CACHE` to True/yes, or if you want to move it elsewhere, set `SINGULARITY_CACHEDIR` to the full path where you want to cache. Remember that when you run commands as sudo this will use root's home at `/root` and not your user's home.

5.2 Temporary Folders

Singularity also uses some temporary directories to build the squashfs filesystem, so this temp space needs to be large enough to hold the entire resulting Singularity image. By default this happens in `/tmp` but can be overridden by setting `SINGULARITY_TMPDIR` to the full path where you want the squashfs temp files to be stored. Since images are typically built as root, be sure to set this variable in root's environment.

If you are building an image on the fly, for example

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

by default a temporary runtime directory is created that looks like `/tmp/.singularity-runtime.xxxxxxxx`.

This can be problematic for some `/tmp` directories that are hosted at Jetstream/OpenStack, Azure, and possibly EC2, which are very small. If you need to change the location of this runtime, then **export** the variable `SINGULARITY_LOCALCACHEDIR`.

```
SINGULARITY_LOCALCACHEDIR=/tmp/pancakes

export SINGULARITY_LOCALCACHEDIR

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

The above runtime folder would be created under `/tmp/pancakes/.singularity-runtime.xxxxxxxx`

5.3 Pull Folder

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

5.4 Environment Variables

All environmental variables are parsed by Singularity python helper functions, and specifically the file defaults.py is a gateway between variables defined at runtime, and pre-defined defaults. By way of import from the file, variables set at runtime do not change if re-imported. This was done intentionally to prevent changes during the execution, and could be changed if needed. For all variables, the order of operations works as follows:

1. First preference goes to environment variable set at runtime
2. Second preference goes to default defined in this file
3. Then, if neither is found, null is returned except in the case that `required=True`. A `required=True` variable not found will system exit with an error.
4. Variables that should not be displayed in debug logger are set with `silent=True`, and are only reported to be defined.

For boolean variables, the following are acceptable for True, with any kind of capitalization or not:

```
("yes", "true", "t", "1","y")
```

5.5 Cache

The location and usage of the cache is also determined by environment variables.

SINGULARITY_DISABLE_CACHE If you want to disable the cache, this means is that the layers are written to a temporary directory. Thus, if you want to disable cache and write to a temporary folder, simply set `SINGULARITY_DISABLE_CACHE` to any true/yes value. By default, the cache is not disabled.

SINGULARITY_CACHEDIR Is the base folder for caching layers and singularity hub images. If not defined, it uses default of `$HOME/.singularity`. If defined, the defined location is used instead.

If `SINGULARITY_DISABLE_CACHE` is set to True, this value is ignored in favor of a temporary directory. For specific sub-types of things to cache, subdirectories are created (by python), including `$SINGULARITY_CACHEDIR/docker` for docker layers and `$SINGULARITY_CACHEDIR/shub` for Singularity Hub images. If the cache is not created, the Python script creates it.

SINGULARITY_PULLFOLDER While this isn't relevant for build, since build is close to pull, we will include it here. By default, images are pulled to the present working directory. The user can change this variable to change that.

SINGULARITY_TMPDIR Is the base folder for squashfs image temporary building. If not defined, it uses default of `$TEMPDIR`. If defined, the defined location is used instead.

SINGULARITY_LOCALCACHEDIR Is the temporary folder (default `/tmp`) to generate runtime folders (containers "on the fly") typically a `run`, `exec` , or `shell` or a `docker://` image. This is different from where downloaded layers are cached (`$SINGULARITY_CACHEDIR`) or pulled (`SINGULARITY_PULLFOLDER`) or where a (non on-the-fly build) happens (`$SINGULARITY_TMPDIR`). See *temporary folders* above for an example. You can generally determine the value of this setting by running a command with `--debug` , and seeing the last line "Removing directory:"

```
singularity --debug run docker://busybox echo "pizza!"

...

DEBUG   [U=1000,P=960]     s_rmdir()                                   Removing␣
↪directory: /tmp/.singularity-runtime.oArO0k
```

5.5.1 Defaults

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

5.5.1.1 Docker

DOCKER_API_BASE Set as `index.docker.io`, which is the name of the registry. In the first version of Singularity we parsed the Registry argument from the build spec file, however now this is removed because it can be obtained directly from the image name (eg, `registry/namespace/repo:tag`). If you don't specify a registry name for your image, this default is used. If you have trouble with your registry being detected from the image URI, use this variable.

DOCKER_API_VERSION Is the version of the Docker Registry API currently being used, by default now is `v2`. **DOCKER_OS** This is exposed via the exported environment variable `SINGULARITY_DOCKER_OS` and pertains to images that reveal a version 2 manifest with a manifest list. In the case that the list is present, we must choose an operating system (this variable) and an architecture (below). The default is `linux`.

DOCKER_ARCHITECTURE This is exposed via the exported environment variable `SINGULARITY_DOCKER_ARCHITECTURE` and the same applies as for the `DOCKER_OS` with regards to being used in context of a list of manifests. In the case that the list is present, we must choose an architecture (this variable) and an os (above). The default is `amd64`, and other common ones include `arm`, `arm64`, `ppc64le`, `386`, and `s390x`. **NAMESPACE** Is the default namespace, `library`.

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

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

5.5.1.2 Singularity Hub

SHUB_API_BASE The default base for the Singularity Hub API, which is `https://singularity-hub.org/api`. If you deploy your own registry, you don't need to change this, you can again specify the registry name in the URI.

5.5.2 General

SINGULARITY_PYTHREADS The Python modules use threads (workers) to download layer files for Docker, and change permissions. By default, we will use 9 workers, unless the environment variable `SINGULARITY_PYTHREADS` is defined. **SINGULARITY_COMMAND_ASIS** By default, we want to make sure the container running process gets passed forward as the current process, so we want to prefix whatever the Docker command or entrypoint is with `exec`. We also want to make sure that following arguments get passed, so we append `"$@"`. Thus, some entrypoint or cmd might look like this:

```
/usr/bin/python
```

and we would parse it into the runscript as:

```
exec /usr/bin/python "$@"
```

However, it might be the case that the user does not want this. For this reason, we have the environmental variable `RUNSCRIPT_COMMAND_ASIS`. If defined as yes/y/1/True/true, etc., then the runscript will remain as `/usr/bin/python`.

CHAPTER

SIX

CONTAINER RECIPES

A Singularity Recipe is the driver of a custom build, and the starting point for designing any custom container. It includes specifics about installation software, environment variables, files to add, and container metadata. You can even write a help section, or define modular components in the container called based on the Scientific Filesystem.

6.1 Overview

A Singularity Recipe file is divided into several parts:

1. **Header**: The Header describes the core operating system to build within the container. Here you will configure the base operating system features that you need within your container. Examples of this include, what distribution of Linux, what version, what packages must be part of a core install.
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. 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`

Please see the *examples* directory in the Singularity source code for some ideas on how to get started.

6.1.1 Header

The header is at the top of the file, and tells Singularity the base Operating System that it should use to build the container. It is composed of several keywords. Specifically:

- `Bootstrap`: references the kind of base you want to use (e.g., docker, debootstrap, shub). For example, a shub bootstrap will pull containers for shub as bases. A Docker bootstrap will pull docker layers to start your image. For a full list see *build*
- `From`: is the named container (shub) or reference to layers (Docker) that you want to use (e.g., vsoch/hello-world)

Depending on the value assigned to `Bootstrap`, other keywords may also be valid in the header.
For example, a very minimal Singularity Hub build might look like this:

```
Bootstrap: shub

From: vsoch/hello-world
```

A build that uses a 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 build base requires particular details during build time. You can read about them and see examples at the following links:

- *shub* (images hosted on Singularity Hub)
- *docker* (images hosted on Docker Hub)
- *localimage* (images saved on your machine)
- *yum* (yum based systems such as CentOS and Scientific Linux)
- *debootstrap* (apt based systems such as Debian and Ubuntu)
- *arch* (Arch Linux)
- *busybox* (BusyBox)
- *zypper* (zypper based systems such as Suse and OpenSuse)

6.1.2 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. Let's add each section to our container to see how it works. For each section, we will build the container from the recipe (a file called Singularity) as follows:

```
$ sudo singularity build roar.simg Singularity
```

6.1.2.1 %help

You don't need to do much programming to add a `%help` section to your container. Just write it into a section:

```
Bootstrap: docker

From: ubuntu


%help

Help me. I'm in the container.
```

And it will work when the user asks the container for help.

```
$ singularity help roar.simg

Help me. I'm in the container.
```

6.1.2.2 %setup

Commands in the %setup section are executed on the host system outside of the container after the base OS has been installed. For versions earlier than 2.3 if you need files during %post, you should copy files from your host to `$SINGULARITY_ROOTFS` to move them into the container. For >2.3 you can add files to the container (added before %post) using the %files section.

In the above, we see that copying something to `$SINGULARITY_ROOTFS` during `%setup` was successful to move the file into the container, but copying during `%post` was not. Let's add a setup to our current container, just writing a file to the root of the image:

```
Bootstrap: docker

From: ubuntu


%help

Help me. I'm in the container.


%setup

    touch ${SINGULARITY_ROOTFS}/tacos.txt

    touch avocados.txt
```

Importantly, notice that the avocados file isn't relative to $SINGULARITY_ROOTFS, so we would expect it not to be in the image. Is tacos there?

```
$ singularity exec roar.simg ls /

bin   environment  lib    mnt   root  scif     sys        usr

boot  etc       lib64  opt   run   singularity  **tacos.txt**  var

dev   home      media  proc  sbin  srv      tmp
```

Yes! And avocados.txt isn't inside the image, but in our present working directory:

```
$ ls

avocados.txt   roar.simg   Singularity
```

6.1.2.3 %files

If you want to copy files from your host system into the container, you should do so using the `%files` section. Each line is a pair of `<source>` and `<destination>`, where the source is a path on your host system, and the destination is a path in the container.

The `%files` section uses the traditional `cp` command, so the same conventions apply Files are copied **before** any `%post` or installation procedures for Singularity versions >2.3. If you are using a legacy version, files are copied after `%post` so you must do this via `%setup`. Let's add the avocado.txt into the container, to join tacos.txt.

```
Bootstrap: docker

From: ubuntu


%help

Help me. I'm in the container.


# Both of the below are copied before %post

# 1. This is how to copy files for legacy < 2.3


%setup

    touch ${SINGULARITY_ROOTFS}/tacos.txt

    touch avocados.txt


# 2. This is how to copy files for >= 2.3


%files

    avocados.txt

    avocados.txt /opt
```

Notice that I'm adding the same file to two different places. For the first, I'm adding the single file to the root of the image. For the second, I'm adding it to opt. Does it work?

```
$ singularity exec roar.simg ls /

 singularity exec roar.simg ls /

**avocados.txt**  dev      home   media  proc  sbin     srv        tmp

bin       environment  lib    mnt    root  scif     sys        usr

boot          etc      lib64  opt    run   singularity  **tacos.txt**  var


$ singularity exec roar.simg ls /opt

**avocados.txt**
```

We have avocados!

6.1.2.4 %labels

To store metadata with your container, you can add them to the `%labels` section. They will be stored in the file `/.singularity.d/labels.json` as metadata within your container. The general format is a `LABELNAME` followed by a `LABELVALUE`. Labels from Docker bootstraps will be carried forward here. Let's add to our example:

```
Bootstrap: docker

From: ubuntu


%help

Help me. I'm in the container.


%setup

    touch ${SINGULARITY_ROOTFS}/tacos.txt

    touch avocados.txt


%files

    avocados.txt

    avocados.txt /opt


%labels

    Maintainer Vanessasaurus

    Version v1.0
```

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

```
$ singularity inspect roar.simg

{

    "org.label-schema.usage.singularity.deffile.bootstrap": "docker",

    "MAINTAINER": "Vanessasaurus",

    "org.label-schema.usage.singularity.deffile": "Singularity",

    "org.label-schema.usage": "/.singularity.d/runscript.help",

    "org.label-schema.schema-version": "1.0",

    "VERSION": "v1.0",

    "org.label-schema.usage.singularity.deffile.from": "ubuntu",

    "org.label-schema.build-date": "2017-10-02T17:00:23-07:00",
```

(continues on next page)

(continued from previous page)

```
    "org.label-schema.usage.singularity.runscript.help": "/.singularity.d/runscript.
↪help",

    "org.label-schema.usage.singularity.version": "2.3.9-development.g3dafa39",

    "org.label-schema.build-size": "1760MB"

}
```

You'll notice some other labels that are captured automatically from the build process. You can read more about labels and metadata *here*.

6.1.2.5 %environment

As of Singularity 2.3, you can add environment variables to your Singularity Recipe in a section called `%environment`. Keep in mind that these environment variables are sourced at runtime and not at build time. This means that if you need the same variables during build time, you should also define them in your `%post` section. Specifically:

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

Since the file is ultimately sourced, you should generally use the same conventions that you might use in a bashrc or profile. In the example below, the variables `VADER` and `LUKE` would not be available during build, but when the container is finished and run:

```
Bootstrap: docker

From: ubuntu


%help

Help me. I'm in the container.


%setup

    touch ${SINGULARITY_ROOTFS}/tacos.txt

    touch avocados.txt


%files

    avocados.txt

    avocados.txt /opt


%labels

    Maintainer Vanessasaurus
```

(continues on next page)

(continued from previous page)

```
    Version v1.0

%environment

    VADER=badguy

    LUKE=goodguy

    SOLO=someguy

    export VADER LUKE SOLO
```

For the rationale behind this approach and why we do not source the %environment section at build time, refer to this issue. When the container is finished, you can easily see environment variables also with inspect, and this is done by showing the file produced above:

```
$ singularity inspect -e roar.simg # Custom environment shell code should follow


    VADER=badguy

    LUKE=goodguy

    SOLO=someguy

    export VADER LUKE SOLO
```

or in the case of variables generated at build time, you can add environment variables to your container in the `%post` section (see below) using the following syntax:

```
%post

    echo 'export JAWA_SEZ=wutini' >> $SINGULARITY_ENVIRONMENT
```

When we rebuild, is it added to the environment?

```
singularity exec roar.simg env | grep JAWA

JAWA_SEZ=wutini
```

Where are all these environment variables going? Inside the container is a metadata folder located at `/.singularity.d`, and a subdirectory `env` for environment scripts that are sourced. Text in the `%environment` section is appended to a file called `/.singularity.d/env/90-environment.sh`. Text redirected to the `SINGULARITY_ENVIRONMENT` variable will 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 `$SINGULARITY_ENVIRONMENT` take precedence over those added via `%environment`. Note that you won't see these variables in the inspect output, as inspect only shows the contents added from `%environment`. See *Environment and Metadata* for more information about the `%labels` and `%environment` sections.

6.1.2.6 %post

Commands in the `%post` section are executed within the container after the base OS has been installed at build time. This is where the meat of your setup will live, including making directories, and installing software and libraries. We

will jump from our simple use case to show a more realistic scientific container. Here we are installing yum, openMPI, and other dependencies for a Centos7 bootstrap:

```
%post

    echo "Installing Development Tools YUM group"

    yum -y groupinstall "Development Tools"

    echo "Installing OpenMPI into container..."


    # Here we are at the base, /, of the container

    git clone https://github.com/open-mpi/ompi.git


    # Now at /ompi

    cd ompi

    ./autogen.pl

    ./configure --prefix=/usr/local

    make

    make install


    /usr/local/bin/mpicc examples/ring_c.c -o /usr/bin/mpi_ring
```

You cannot copy files from the host to your container in this section, but you can of course download with commands like `git clone` and `wget` and `curl`.

6.1.2.7 %runscript

The `%runscript` is another scriptlet, but it does not get executed during bootstrapping. Instead it gets persisted within the container to a file (or symlink for later versions) called `singularity` which is the execution driver when the container image is run (either via the `singularity run` command or via executing the container directly). When the `%runscript` is executed, all options are passed along to the executing script at runtime, this means that you can (and should) manage argument processing from within your runscript. Here is an example of how to do that, adding to our work in progress:

```
Bootstrap: docker

From: ubuntu


%help

Help me. I'm in the container.


%setup
```

(continues on next page)

(continued from previous page)

```
    touch ${SINGULARITY_ROOTFS}/tacos.txt

    touch avocados.txt


%files

    avocados.txt

    avocados.txt /opt


%labels

    Maintainer Vanessasaurus

    Version v1.0


%environment

    VADER=badguy

    LUKE=goodguy

    SOLO=someguy

    export VADER LUKE SOLO



%post

    echo 'export JAWA_SEZ=wutini' >> $SINGULARITY_ENVIRONMENT


%runscript

    echo "Rooooar!"

    echo "Arguments received: $*"

    exec echo "$@"
```

In this particular runscript, the arguments 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. Using the `exec` command is like handing off the calling process to the one in the container. The final command (the echo) replaces the current entry in the process table (which originally was the call to Singularity). This makes it so the runscript shell process ceases to exist, and the only process running inside this container is the called echo command. This could easily be another program like python, or an analysis script. Running it, it works as expected:

```
$ singularity run roar.simg

Rooooar!

Arguments received:
```

(continues on next page)

(continued from previous page)

```
$ singularity run roar.simg one two

Rooooar!

Arguments received: one two

one two
```

6.1.2.8 %test

You may choose to add a `%test` section to your definition file. This section will be run at the very end of the build process and will give you a chance to validate the container during the bootstrap process. You can also execute this scriptlet through the container itself, such that you can always test the validity of the container itself as you transport it to different hosts. Extending on the above Open MPI `%post`, consider this real world example:

```
%test

    /usr/local/bin/mpirun --allow-run-as-root /usr/bin/mpi_test
```

This is a simple Open MPI test to ensure that the MPI is build properly and communicates between processes as it should. If you want to build without running tests (for example, if the test needs to be done in a different environment), you can do so with the `--notest` argument:

```
$ sudo singularity build --notest mpirun.simg Singularity
```

This argument is useful in cases where you need hardware that is available during runtime, but is not available on the host that is building the image.

6.2 Apps

What if you want to build a single container with two or three different apps that each have their own runscripts and custom environments? In some circumstances, it may be redundant to build different containers for each app with almost equivalent dependencies.

Starting in Singularity 2.4 all of the above commands can also be used in the context of internal modules called *apps* based on the Standard Container Integration Format. For details on apps, see the *apps* documentation. For a quick rundown of adding an app to your container, here is an example runscript:

```
Bootstrap: docker

From: ubuntu


%environment

    VADER=badguy

    LUKE=goodguy

    SOLO=someguy
```

(continues on next page)

(continued from previous page)

```
    export VADER LUKE SOLO

%labels
   Maintainer Vanessasaur

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

%apprun foo
    exec echo "RUNNING FOO"

%applabels foo
   BESTAPP=FOO
   export BESTAPP

%appinstall foo
   touch foo.exec

%appenv foo
    SOFTWARE=foo
    export SOFTWARE

%apphelp foo
    This is the help for foo.

%appfiles foo
   avocados.txt


##############################
# bar
##############################
```

(continues on next page)

(continued from previous page)

```
%apphelp bar

    This is the help for bar.


%applabels bar

   BESTAPP=BAR

   export BESTAPP


%appinstall bar

    touch bar.exec


%appenv bar

    SOFTWARE=bar

    export SOFTWARE
```

Importantly, note that the apps can exist alongside any and all of the primary sections (e.g. `%post` or `%runscript`), and the new `%appinstall` section is the equivalent of %post but for an app. The title sections (######) aren't necessary or required, they are just comments to show you the different apps. The ordering isn't important either, you can have any mixture of sections anywhere in the file after the header. The primary difference is now the container can perform any of it's primary functions in the context of an app:

What apps are installed in the container?

```
$ singularity apps roar.simg

bar

foo
```

Help me with bar!

```
$ singularity help --app bar roar.simg

This is the help for bar.
```

Run foo

```
singularity run --app foo roar.simg

RUNNING FOO
```

Show me the custom environments

Remember how we defined the same environment variable, SOFTWARE for each of foo and bar? We can execute a command to search the list of active environment variables with grep to see if the variable changes depending on the app we specify:

```
$ singularity exec --app foo roar.simg env | grep SOFTWARE

SOFTWARE=foo

$ singularity exec --app bar roar.simg env | grep SOFTWARE

SOFTWARE=bar
```

6.3 Examples

For more examples, for real world scientific recipes we recommend you look at other containers on Singularity Hub. For examples of different bases, look at the examples folder for the most up-to-date examples. For apps, including snippets and tutorial with more walk throughs, see SCI-F Apps Home.

6.4 Best Practices for Build Recipes

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

1. To make your container internally modular, use *SCI-F apps*. Shared dependencies (between app modules) can go under `%post`.
2. For global installs to `%post`, 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).
3. Make your container speak for itself. If your runscript doesn't spit out help, write a `%help` or `%post` or `%apphelp` section. A good container tells the user how to interact with it.
4. If you require any special environment variables to be defined, add them the `%environment` and `%appenv` sections of the build recipe.
5. Files should never be owned by actual users, they should always be owned by a system account (UID less than 500).
6. Ensure that the container's `/etc/passwd` , `/etc/group` , `/etc/shadow` , and no other sensitive files have anything but the bare essentials within them.
7. It is encouraged to build containers from a recipe instead of a sandbox that has been manually changed. This ensures greatest possibility of reproducibility and mitigates the black box effect.

Are you a recipe pro and now ready to build? Take a look at the *build* documentation.

CHAPTER

SEVEN

SINGULARITY FLOW

This section describes a suggested "best-practices" work-flow for building, running, and managing your containers.

There are generally two ways to get images. You either want to pull an image file as is, or (more likely) build your own custom image. We will start with talking about build and the many different use cases.

7.1 Building Images

If you read the *quick start*, you probably remember that building images from a Docker base does not require a *Singularity recipe*. However, if you do want to build and customize your image, you can create a *Singularity recipe* text file, which is a simple text file that describes how the container should be made.

7.1.1 The Singularity Flow

The diagram below is a visual depiction of how you can use Singularity to build images. The high level idea is that we have two environments:

- a **build** environment (where you have sudo privileges) to test and build your container
- a **production** environment where you run your container

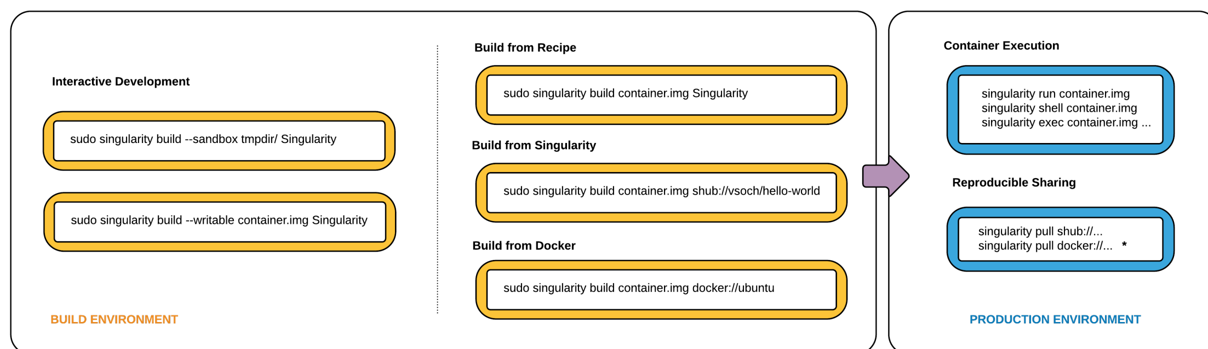

Fig. 1: Singularity Workflow

Singularity production images are immutable. This is a feature added as of Singularity 2.4, and it ensures a higher level of reproducibility and verification of images. To read more about the details, check out the *build* docs. However,

immutability is not so great when you are testing, debugging, or otherwise want to quickly change your image. We will proceed by describing a typical work-flow of developing first, building a final image, and using it in production.

7.1.2 1. Development Commands

If you want a writable image or folder for developing, you have two options:

- build into a directory that has writable permissions using the `--sandbox` option
- build into an ext3 image file, that has writable permissions with the `--writable` option

In both cases you will need to execute your container with the `--writable` option at runtime for your changes to be persistent.

7.1.2.1 Sandbox Folder

To build into a folder (we call this a "sandbox") just ask for it:

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

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /root/.singularity/docker

Importing: base Singularity environment

Importing: /root/.singularity/docker/
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118.tar.gz

Importing: /root/.singularity/docker/
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a.tar.gz

Importing: /root/.singularity/docker/
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2.tar.gz

Importing: /root/.singularity/docker/
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e.tar.gz

Importing: /root/.singularity/docker/
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9.tar.gz

Importing: /root/.singularity/metadata/
↪sha256:22e289880847a9a2f32c62c237d2f7e3f4eae7259bf1d5c7ec7ffa19c1a483c8.tar.gz

Building image from sandbox: ubuntu/

Singularity container built: ubuntu/
```

We now have a folder with the entire ubuntu OS, plus some Singularity metadata, plopped in our present working directory.

```
 $ tree -L 1 ubuntu

ubuntu

├── bin
```

(continues on next page)

(continued from previous page)

```
├── boot
├── dev
├── environment -> .singularity.d/env/90-environment.sh
├── etc
├── home
├── lib
├── lib64
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── singularity -> .singularity.d/runscript
├── srv
├── sys
├── tmp
├── usr
└── var
```

And you can shell into it just like a normal container.

```
$ singularity shell ubuntu

Singularity: Invoking an interactive shell within container...


Singularity ubuntu:~/Desktop> touch /hello.txt

touch: cannot touch '/hello.txt': Permission denied
```

You can make changes to the container (assuming you have the proper permissions to do so) but those changes will disappear as soon as you exit. To make your changes persistent across sessions, use the `--writable` option. It's also a good practice to shell into your container as root to ensure you have permissions to write where you like.

```
$ sudo singularity shell ubuntu

Singularity: Invoking an interactive shell within container...


Singularity ubuntu:/home/vanessa/Desktop> touch /hello.txt
```

7.1.2.2 Writable Image

If you prefer to work with a writable image file rather than a directory, you can perform a similar development build and specify the `--writable` option. This will produce an image that is writable with an ext3 file system. Unlike the sandbox, it is a single image file.

```
$ sudo singularity build --writable ubuntu.img docker://ubuntu

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /root/.singularity/docker

Importing: base Singularity environment

Importing: /root/.singularity/docker/
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118.tar.gz

Importing: /root/.singularity/docker/
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a.tar.gz

Importing: /root/.singularity/docker/
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2.tar.gz

Importing: /root/.singularity/docker/
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e.tar.gz

Importing: /root/.singularity/docker/
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9.tar.gz

Importing: /root/.singularity/metadata/
↪sha256:22e289880847a9a2f32c62c237d2f7e3f4eae7259bf1d5c7ec7ffa19c1a483c8.tar.gz

Building image from sandbox: /tmp/.singularity-build.VCHPpP

Creating empty Singularity writable container 130MB

Creating empty 162MiB image file: ubuntu.img

Formatting image with ext3 file system

Image is done: ubuntu.img

Building Singularity image...


Cleaning up...


Singularity container built: ubuntu.img
```

You can use this image with commands like `shell`, `exec` , `run` , and if you want to change the image you must use the `--writable` flag. As before, it's a good idea to issue these commands as root to ensure you have the proper permissions to write.

```
$ sudo singularity shell --writable ubuntu.img


Development Tip! When building containers, it often is the case that

you will have a lot of testing of installation commands, and if

building a production image, one error will stop the entire build.

If you interactively write the build recipe with one of these

writable containers, you can debug as you go, and then build the

production (squashfs) container without worrying that it will error

and need to be started again.
```

7.1.3 2. Production Commands

Let's set the scene - we just finished building our perfect hello world container. It does a fantastic hello-world analysis, and we have written a paper on it! We now want to build an immutable container - meaning that if someone obtained our container and tried to change it, they could not. They could easily use the same recipe that you used (it is provided as metadata inside the container), or convert your container to one of the writable formats above using `build` . So your work can still be extended.

7.1.3.1 Recommended Production Build

What we want for production is a build into a squashfs image . Squashfs is a read only, and compressed filesystem, and well suited for confident archive and re-use of your hello-world. To build a production image, just remove the extra options:

```
sudo singularity build ubuntu.simg docker://ubuntu

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /root/.singularity/docker

Importing: base Singularity environment

Importing: /root/.singularity/docker/
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118.tar.gz

Importing: /root/.singularity/docker/
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a.tar.gz

Importing: /root/.singularity/docker/
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2.tar.gz

Importing: /root/.singularity/docker/
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e.tar.gz
```

(continues on next page)

(continued from previous page)

```
Importing: /root/.singularity/docker/
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9.tar.gz

Importing: /root/.singularity/metadata/
↪sha256:22e289880847a9a2f32c62c237d2f7e3f4eae7259bf1d5c7ec7ffa19c1a483c8.tar.gz

Building Singularity image...

Cleaning up...

Singularity container built: ubuntu.simg
```

7.1.3.2 Production Build from Sandbox

We understand that it might be wanted to build a Singularity (squashfs) from a previous development image. While we advocate for the first approach, we support this use case. To do this, given our folder called "ubuntu/" we made above:

```
sudo singularity build ubuntu.simg ubuntu/
```

It could be the case that a cluster maintains a "working" base of container folders (with writable) and then builds and provides production containers to its users.

If you want to go through this entire process without having singularity installed locally, or without leaving your cluster, you can build images using Singularity Hub.

CHAPTER

EIGHT

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.

8.1 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 in via two primary methods: system-defined bind points and conditional user-defined bind points.

8.1.1 System-defined bind points

The system administrator has the ability to define what bind points will be included automatically inside each container. The bind paths are locations on the host's root file system which should also be visible within the container. Some of the bind paths are automatically derived (e.g. a user's home directory) and some are statically defined (e.g. bind path in the Singularity configuration file). In the default configuration, the directories `$HOME` , `/tmp` , `/proc` , `/sys` , `/dev` and are among the system-defined bind points.

8.1.2 User-defined bind points

If the system administrator has enabled user control of binds, you will be able to request your own bind points within your container.

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 to bind a directory on the host system. This means that if you want to bind to a point within the container such as `/global`, that directory must already exist within the container.

It is, however, possible that the system administrator has enabled a Singularity feature called overlay in the Singularity configuration file. This will cause the bind points to be created on an as-needed basis in an overlay file system so that the underlying container is not modified. But because the overlay feature is not always enabled or is unavailable in the kernels of some older host systems, it may be necessary for container standards to exist to ensure portability from host to host.

8.1.2.1 Specifying Bind Paths

Many of the Singularity commands such as `run`, `exec` , and `shell` take the `--bind / command-line` option to specify bind paths, in addition to the `SINGULARITY_BINDPATH` environment variable. This option's argument is a comma-delimited string of bind path specifications in the format `src[:dest[:opts]]`, where `src` and `dest`

are outside and inside paths. If `dest` is not given, it is set equal to `src`. Mount options (`opts`) may be specified as `ro` (read-only) or `rw` (read/write, which is the default). The `--bind/-B` option can be specified multiple times, or a comma-delimited string of bind path specifications can be used.

Here's an example of using the `-B` option and binding `/tmp` on the host to `/scratch` in the container (`/scratch` does not need to already exist in the container if file system overlay is enabled):

```
$ singularity shell -B /tmp:/scratch /tmp/Centos7-ompi.img

Singularity: Invoking an interactive shell within container...


Singularity.Centos7-ompi.img> ls /scratch

ssh-7vywtVeOez  systemd-private-cd84c81dda754fe4a7a593647d5a5765-ntpd.service-12nMO4
```

You can bind multiple directories in a single command with this syntax:

```
$ singularity shell -B /opt,/data:/mnt /tmp/Centos7-ompi.img
```

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_BINDPATH="/opt,/data:/mnt"

$ singularity shell /tmp/Centos7-ompi.img
```

Using the environment variable `$SINGULARITY_BINDPATH`, you can bind directories 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.

8.1.2.2 Binding with Overlay

If a bind path is requested and the bind point does not exist within the container, a warning message will be displayed and Singularity will continue trying to start the container. For example:

```
$ singularity shell --bind /global /tmp/Centos7-ompi.img

WARNING: Non existent bind point (directory) in container: '/global'

Singularity: Invoking an interactive shell within container...


Singularity.Centos7-ompi.img>
```

Even though `/global` did not exist inside the container, the shell command printed a warning but continued on. If overlay is available and enabled, you will find that we no longer get the error and `/global` is created and accessible as expected:

```
$ singularity shell --bind /global /tmp/Centos7-ompi.img

Singularity: Invoking an interactive shell within container...


Singularity.Centos7-ompi.img>
```

In this case, Singularity dynamically created the necessary bind point in your container. Without overlay, you would have needed to manually create the `/global` directory inside your container.

CHAPTER

NINE

PERSISTENT OVERLAYS

Persistent overlay images were added in version 2.4. This feature allows you to overlay a writable file system on an immutable read-only container for the illusion of read-write access.

9.1 Overview

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

In Singularity versions 2.4 and later an overlay file system is automatically added to your squashfs or sandbox container when it is mounted. This means you can install new software and create and modify files even though your container is read-only. But your changes will disappear as soon as you exit the container.

If you want your changes to persist in your container across uses, you can create a writable image to use as a persistent overlay. Then you can specify that you want to use the image as an overlay at runtime with the `--overlay` option.

You can use a persistent overlays with the following commands:

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

9.2 Usage

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

```
$ singularity build ubuntu.simg shub://GodloveD/ubuntu
```

Then you must create a writable, ext3 image. We can do so with the `image.create` command:

```
$ singularity image.create my-overlay.img
```

Now you can use this overlay image with your container. Note that it is not necessary to be root to use an overlay partition, but this will ensure that we have write privileges where we want them.

```
$ sudo singularity shell --overlay my-overlay.img ubuntu.simg

Singularity ubuntu.simg:~> touch /foo
```

(continues on next page)

(continued from previous page)

```
Singularity ubuntu.simg:~> apt-get install -y vim

Singularity ubuntu.simg:~> which vim

/usr/bin/vim

Singularity ubuntu.simg:~> exit
```

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

```
$ sudo singularity shell --overlay my-overlay.img ubuntu.simg

Singularity ubuntu.simg:~> ls /foo

/foo

Singularity ubuntu.simg:~> which vim

/usr/bin/vim

Singularity ubuntu.simg:~> exit
```

If you mount your container without the `--overlay` option, your changes will be gone.

```
$ sudo singularity shell ubuntu.simg

Singularity ubuntu.simg:~> ls /foo

ls: cannot access 'foo': No such file or directory

Singularity ubuntu.simg:~> which vim

Singularity ubuntu.simg:~> exit
```

CHAPTER

TEN

RUNNING SERVICES

Singularity 2.4 introduced the ability to run "container instances", allowing you to run services (e.g. Nginx, MySQL, etc...) using Singularity. A container instance, simply put, is a persistent and isolated version of the container image that runs in the background.

10.1 Why container instances?

that is pretty simple, I install nginx and start the service:

```
apt-get update && apt-get install -y nginx

service nginx start
```

With older versions of Singularity, if you were to do something like this, from inside the container you would happily see the service start, and the web server running! But then if you were to log out of the container what would happen? Orphan process within unreachable namespaces! You would lose control of the process. It would still be running, but you couldn't easily kill or interface with it. This is a called an orphan process. Singularity versions less than 2.4 were not designed to handle running services properly.

10.2 Container Instances in Singularity

With Singularity 2.4 and the addition of container instances, the ability to cleanly, reliably, and safely run services in a container is here. First, let's put some commands that we want our instance to execute into a script. Let's call it a `startscript`. This fits into a definition file as follows:

```
%startscript


service nginx start
```

Now let's say we build a container with that startscript into an image called `nginx.img` and we want to run an nginx service. All we need to do is start the instance with the *instance.start* command, and the startscript will run inside the container automatically:

```
              [command]        [image]    [name of instance]

$ singularity instance.start   nginx.img  web
```

When we run that command, Singularity creates an isolated environment for the container instances' processes/services to live inside. We can confirm that this command started an instance by running the instance.list command like so:

```
$ singularity instance.list

INSTANCE NAME    PID      CONTAINER IMAGE

web              790      /home/mibauer/nginx.img
```

If we want to run multiple instances from the same image, it's as simple as running the command multiple times. The instance names are an identifier used to uniquely describe an instance, so they cannot be repeated.

```
$ singularity instance.start   nginx.img  web1

$ singularity instance.start   nginx.img  web2

$ singularity instance.start   nginx.img  web3
```

And again to confirm that the instances are running as we expected:

```
$ singularity instance.list

INSTANCE NAME    PID      CONTAINER IMAGE

web1             790      /home/mibauer/nginx.img

web2             791      /home/mibauer/nginx.img

web3             792      /home/mibauer/nginx.img
```

If the service you want to run in your instance requires a bind mount, then you must pass the `-B` option when calling `instance.start`. For example, if you wish to capture the output of the `web1` container instance which is placed at `/output/` inside the container you could do:

```
$ singularity instance.start -B output/dir/outside/:/output/ nginx.img  web1
```

If you want to poke around inside of your instance, you can do a normal `singularity shell` command, but give it the instance URI:

```
$ singularity shell instance://web1

Singularity: Invoking an interactive shell within container...


Singularity pdf_server.img:~/>
```

Similarly, you can use the `singularity run/exec` commands on instances:

```
$ singularity run instance://web1

$ singularity exec instance://web1 ps -ef
```

When using `run` with an instance URI, the `runscript` will be executed inside of the instance. Similarly with `exec`, it will execute the given command in the instance.

When you are finished with your instance you can clean it up with the *instance.stop* command as follows:

```
$ singularity instance.stop web1
```

If you have multiple instances running and you want to stop all of them, you can do so with a wildcard or the -a flag:

```
$ singularity instance.stop \*

$ singularity instance.stop -a
```

Note: Note that you must escape the wildcard with a backslash like this `*` to pass it properly.

10.3 Nginx "Hello-world" in Singularity

Let's take a look at setting up a sample nginx web server using instances in Singularity. First we will just create a basic definition file:

```
Bootstrap: docker

From: nginx

Includecmd: no


%startscript

    nginx
```

All this does is download the official nginx Docker container, convert it to a Singularity image, and tell it to run nginx when you start the instance. Since we're running a web server, we're going to run the following commands as root.

```
# singularity build nginx.img Singularity

# singularity instance.start nginx.img web1
```

Just like that we've downloaded, built, and ran an nginx Singularity image. And to confirm that it's correctly running:

```
$ curl localhost

127.0.0.1 - - [06/Oct/2017:21:46:43 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0"
↪"-"

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

<style>

    body {

        width: 35em;
```

(continues on next page)

(continued from previous page)

```
        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>
```

10.4 Putting all together

In this section, we will demonstrate an example of packaging a service into a container and running it. The service we will be packaging is an API server that converts a web page into a PDF, and can be found here. The final example can be found here on GitHub. If you wish to just download the final image directly from Singularity Hub, simply run `singularity pull shub://bauerm97/instance-example`.

10.4.1 Building the image

To begin, we need to build the image. When looking at the GitHub page of the `url-to-pdf-api`, we can see that it is a Node 8 server that uses headless Chromium called Puppeteer. Let's first choose a base from which to build our container, in this case I used the docker image `node:8` which comes pre-installed with Node 8:

```
Bootstrap: docker

From: node:8

Includecmd: no
```

Puppeteer also requires a few dependencies to be manually installed in addition to Node 8, so we can add those into the `post` section as well as the installation script for the `url-to-pdf-api`:

```
%post

     apt-get update

     apt-get install -yq gconf-service libasound2 libatk1.0-0 libc6 libcairo2␣
↪libcups2 \

     libdbus-1-3 libexpat1 libfontconfig1 libgcc1 libgconf-2-4 libgdk-pixbuf2.0-0 \

     libglib2.0-0 libgtk-3-0 libnspr4 libpango-1.0-0 libpangocairo-1.0-0 libstdc++6 \

     libx11-6 libx11-xcb1 libxcb1 libxcomposite1 libxcursor1 libxdamage1 libxext6 \

     libxfixes3 libxi6 libxrandr2 libxrender1 libxss1 libxtst6 ca-certificates \

     fonts-liberation libappindicator1 libnss3 lsb-release xdg-utils wget curl

     rm -r /var/lib/apt/lists/*

     cd /

     git clone https://github.com/alvarcarto/url-to-pdf-api.git pdf_server

     cd pdf_server

     npm install

     chmod -R 0755 .
```

And now we need to define what happens when we start an instance of the container. In this situation, we want to run the commands that starts up the url-to-pdf-api server:

```
%startscript

    cd /pdf_server

    # Use nohup and /dev/null to completely detach server process from terminal

    nohup npm start > /dev/null 2>&1 < /dev/null &
```

Also, the `url-to-pdf-api` server requires `environment` some variables be set, which we can do in the environment section:

```
%environment

    NODE_ENV=development

    PORT=8000

    ALLOW_HTTP=true

    URL=localhost
```

(continues on next page)

(continued from previous page)

```
    export NODE_ENV PORT ALLOW_HTTP URL
```

Now we can build the definition file into an image! Simply run `build` and the image will be ready to go:

```
$ sudo singularity build url-to-pdf-api.img Singularity
```

10.4.2 Running the Server

Now that we have an image, we are ready to start an instance and run the server:

```
$ singularity instance.start url-to-pdf-api.img pdf
```

We can confirm it's working by sending the server an http request using curl:

```
$ curl -o google.pdf localhost:8000/api/render?url=http://google.com

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current

                                 Dload  Upload   Total   Spent    Left  Speed

100 51664  100 51664    0     0  12443      0  0:00:04  0:00:04 --:--:-- 12446
```

If you shell into the instance, you can see the running processes:

```
$ singularity shell instance://pdf

Singularity: Invoking an interactive shell within container...


Singularity pdf_server.img:~/bauerm97/instance-example> ps auxf

USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND

node        87  0.2  0.0  20364  3384 pts/0    S    16:16   0:00 /bin/bash --norc

node        88  0.0  0.0  17496  2144 pts/0    R+   16:16   0:00  \_ ps auxf

node         1  0.0  0.0  13968  1904 ?        Ss   16:10   0:00 singularity-
↪instance: mibauer [pdf]

node         3  0.1  0.4 997452 40364 ?        Sl   16:10   0:00 npm

node        13  0.0  0.0   4340   724 ?        S    16:10   0:00  \_ sh -c nodemon --
↪watch ./src -e j

node        14  0.0  0.4 1184492 37008 ?       Sl   16:10   0:00      \_ node /scif/
↪apps/pdf_server/p

node        26  0.0  0.0   4340   804 ?        S    16:10   0:00          \_ sh -c␣
↪node src/index.js

node        27  0.2  0.5 906108 43424 ?        Sl   16:10   0:00              \_ node␣
↪src/index.js

Singularity pdf_server.img:~/bauerm97/instance-example> ls
```

(continues on next page)

(continued from previous page)

```
LICENSE  README.md  Singularity  out  pdf_server.img

Singularity pdf_server.img:~/bauerm97/instance-example> exit
```

10.4.3 Making it Pretty

Now that we have confirmation that the server is working, let's make it a little cleaner. It's difficult to remember the exact curl command and URL syntax each time you want to request a PDF, so let's automate that. To do that, we're going to be using Standard Container Integration Format (SCIF) apps, which are integrated directly into singularity. If you haven't already, check out the *Singularity app documentation* to come up to speed.

First off, we're going to move the installation of the url-to-pdf-api into an app, so that there is a designated spot to place output files. To do that, we want to add a section to our definition file to build the server:

```
%appinstall pdf_server

    git clone https://github.com/alvarcarto/url-to-pdf-api.git pdf_server

    cd pdf_server

    npm install

    chmod -R 0755 .
```

And update our `startscript` to point to the app location:

```
%startscript

    cd "${APPROOT_pdf_server}/pdf_server"

    # Use nohup and /dev/null to completely detach server process from terminal

    nohup npm start > /dev/null 2>&1 < /dev/null &
```

Now we want to define the pdf_client app, which we will run to send the requests to the server:

```
%apprun pdf_client

    if [ -z "${1:-}" ]; then

        echo "Usage: singularity run --app pdf <instance://name> <URL> [output file]"

        exit 1

    fi

    curl -o "${SINGULARITY_APPDATA}/output/${2:-output.pdf}" "${URL}:${PORT}/api/
↪render?url=${1}"
```

As you can see, the `pdf_client` app checks to make sure that the user provides at least one argument. Now that we have an output directory in the container, we need to expose it to the host using a bind mount. Once we've rebuilt the container, make a new directory callout `out` for the generated PDF's to go. Now we simply start the instance like so:

```
$ singularity instance.start -B out/:/scif/data/pdf_client/output/ url-to-pdf-api.img␣
↪pdf
```

And to request a pdf simply do:

```
$ singularity run --app pdf_client instance://pdf http://google.com google.pdf
```

And to confirm that it worked:

```
$ ls out/

google.pdf
```

When you are finished, use the instance.stop command to close all running instances.

```
$ singularity instance.stop \*
```

10.5 Important Notes

Note: The instances are linked with your user. So if you start an instance with sudo, that is going to go under root, and you will need to call `sudo singularity instance.list` in order to see it.

CHAPTER ELEVEN

CONTAINER CHECKS

Singularity 2.4 introduced the ability to run container "checks" on demand. Checks can be anything from a filter for sensitive information, to an analysis of installed binaries. A few default checks are installed with Singularity and others can be added by the administrator. Users can perform checks at build time or on demand: Perform all default checks, these are the same

```
$ singularity check ubuntu.img

$ singularity check --tag default ubuntu.img
```

Perform checks with tag "clean"

```
$ singularity check --tag clean ubuntu.img
```

11.1 Tags and Organization

Currently, checks are organized by tag and security level. If you know a specific tag that you want to use, for example "docker" deploys checks for containers with Docker imported layers, you can specify the tag:

```
USAGE

    -t/--tag       tag to filter checks. default is "default"

                      Available: default, security, docker, clean


EXAMPLE

$ singularity check --tag docker ubuntu.img
```

If you want to run checks associated with a different security level, you can specify with `--low`, `--med`, or `--high` :

```
USAGE: singularity [...] check [exec options...] <container path>

This command will run security checks for an image.

Note that some checks require sudo.


    -l/--low       Specify low threshold (all checks, default)

```

(continues on next page)

(continued from previous page)

```
    -m/--med        Perform medium and high checks

    -h/--high       Perform only checks at level high
```

Note: Note that some checks will require sudo, and you will be alerted if this is the case and you didn't use it. Finally, if you want to run all default checks, just don't specify a tag or level.

11.2 What checks are available?

Currently, you can view all installable checks here, and we anticipate adding an ability to view tags that are available, along with your own custom checks. You should also ask your administration if new checks have been added not supported by Singularity. If you want to request adding a new check, please tell us!.

CHAPTER

TWELVE

ENVIRONMENT AND METADATA

Singularity containers support environment variables and labels that you can add to your container during the build process. This page details general information about defining environments and labels. If you are looking for specific environment variables for build time, see build environment.

12.1 Environment

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

```
Bootstrap: shub

From: vsoch/hello-world


%environment

    VARIABLE_NAME=VARIABLE_VALUE

    export VARIABLE_NAME
```

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

```
%post

    echo 'export VARIABLE_NAME=VARIABLE_VALUE' >>$SINGULARITY_ENVIRONMENT
```

Text in the `%environment` section will be appended to the file `/.singularity.d/env/90-environment.sh` while text redirected to `$SINGULARITY_ENVIRONMENT` will end up in the file /.singularity.d/env/91-environment.sh.

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

If you need to define a variable at runtime, set variables inside the container by prefixing them with `SINGULARITYENV_`. They will be transposed automatically and the prefix will be stripped. For example, let's say we want to set the variable `HELLO` to have value `WORLD`. We can do that as follows:

```
$ SINGULARITYENV_HELLO=WORLD singularity exec --cleanenv centos7.img env
```

(continues on next page)

(continued from previous page)

```
HELLO=WORLD

LD_LIBRARY_PATH=:/usr/local/lib:/usr/local/lib64

SINGULARITY_NAME=test.img

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

PWD=/home/gmk/git/singularity

LANG=en_US.UTF-8

SHLVL=0

SINGULARITY_INIT=1

SINGULARITY_CONTAINER=test.img
```

Notice the `--cleanenv` in the example above? That argument specifies that we want to remove the host environment from the container. If we remove the `--cleanenv`, we will still pass forward `HELLO=WORLD`, and the list shown above, but we will also pass forward all the other environment variables from the host.

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

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

12.2 Labels

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

For containers that are generated with Singularity version 2.4 and later, labels are represented using the rc1 Label Schema. For example:

```
$ singularity inspect dino.img

{

    "org.label-schema.usage.singularity.deffile.bootstrap": "docker",

    "MAINTAINER": "Vanessasaurus",

    "org.label-schema.usage.singularity.deffile": "Singularity.help",

    "org.label-schema.usage": "/.singularity.d/runscript.help",

    "org.label-schema.schema-version": "1.0",

    "org.label-schema.usage.singularity.deffile.from": "ubuntu:latest",
```

(continues on next page)

(continued from previous page)

```
    "org.label-schema.build-date": "2017-07-28T22:59:17-04:00",

    "org.label-schema.usage.singularity.runscript.help": "/.singularity.d/runscript.
↪help",

    "org.label-schema.usage.singularity.version": "2.3.1-add/label-schema.g00f040f",

    "org.label-schema.build-size": "715MB"

}
```

You will notice that the one label doesn't belong to the label schema, `MAINTAINER` . This was a user provided label during bootstrap. Finally, for Singularity versions >= 2.4, the image build size is added as a label, `org.label-schema.build-size`, and the label schema is used throughout. For versions earlier than 2.4, containers did not use the label schema, and looked like this:

```
singularity exec centos7.img cat /.singularity.d/labels.json

{ "name":

      "CentOS Base Image",

       "build-date": "20170315",

       "vendor": "CentOS",

       "license": "GPLv2"

}
```

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

```
Bootstrap: docker

From: ubuntu: latest



%labels

AUTHOR Vanessasaur
```

The `inspect` command is useful for viewing labels and other container meta-data.

12.3 Container Metadata

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

```
/.singularity.d/


├── actions
```

(continues on next page)

(continued from previous page)

```
│   ├── exec
│   ├── run
│   ├── shell
│   ├── start
│   └── test
├── env
│   ├── 01-base.sh
│   ├── 90-environment.sh
│   ├── 95-apps.sh
│   └── 99-base.sh
├── labels.json
├── libs
├── runscript
├── Singularity
└── startscript
```

- **actions**: This directory contains helper scripts to allow the container to carry out the action commands.
- **env**: All *.sh files in this directory are sourced in alpha-numeric order when the container is initiated. For legacy purposes there is a symbolic link called `/environment` that points to `/.singularity.d/env/90-environment.sh`.
- **labels.json**: The json file that stores a containers labels described above.
- **libs**: At runtime the user may request some host-system libraries to be mapped into the container (with the `--nv` option for example). If so, this is their destination.
- **runscript**: The commands in this file will be executed when the container is invoked with the `run` command or called as an executable. For legacy purposes there is a symbolic link called `/singularity` that points to this file
- **Singularity**: This is the Recipe file that was used to generate the container. If more than 1 Recipe file was used to generate the container additional Singularity files will appear in numeric order in a sub-directory called `bootstrap_history`
- **startscript**: The commands in this file will be executed when the container is invoked with the `instance.start` command.

CHAPTER

THIRTEEN

REPRODUCIBLE SCI-F APPS

13.1 Why do we need SCI-F?

The Scientific Filesystem (SCIF) provides a consistent and modular method to create containers. The SCI-F makes some assumptions about how the container will be created. By adhering to with the SCI-F format containers can become more reproducible. For example, installing a set of libraries, defining environment variables, or adding labels that belong to application `foo` makes a strong assertion that those dependencies belong to `foo`. When I run `foo`, I can be confident that the container is running in this context, meaning with `foo's` custom environment, and with `foo's` libraries and executables on the path. This approach is different from serving many executables in a single container, that may have no way to know which application is associated with the container's intended functions. This documentation will walk through some rationale, background, and examples of the SCI-F integration for Singularity containers. For other examples (and a client that works across container technologies) see the the scientific filesystem.

To start, let's take a look at this series of steps to install dependencies for software foo and bar.

```
%post


# install dependencies 1

# install software A (foo)

# install software B (bar)

# install software C (foo)

# install software D (bar)
```

The creator may know that A and C were installed for `foo` and B and D for `bar` , but down the road, when someone discovers the container, if they can find the software at all, the intention of the container creator would be lost. As many are now, containers without any form of internal organization and predictability are black boxes. We don't know if some software installed to `/opt` , or to `/usr/local/bin` , or to their custom favorite folder `/code` . We could assume that the creator added important software to the path and look in these locations, but that approach is still akin to fishing in a swamp. We might only hope that the container's main function, the Singularity runscript, is enough to make the container perform as intended.

13.1.1 Mixed up Modules

If your container truly runs one script, the traditional model of a runscript fits well. Even in the case of having two functions like `foo` and `bar` you probably have something like this.

```
%runscript

if some logic to choose foo:

   check arguments for foo

   run foo

else if some logic to choose bar:

   run bar
```

and maybe your environment looks like this:

```
%environment

    BEST_GUY=foo

    export BEST_GUY
```

but what if you run into this issue, with foo and bar?

```
%environment

    BEST_GUY=foo

    BEST_GUY=bar

    export BEST_GUY
```

You obviously can't have them at separate times. You'd have to source some custom environment file (that you make on your own) and it gets confusing with issues of using shell and sourcing the container. We don't know who the best guy is! You probably get the general idea. Without consistent internal organization and modularity the following may result:

- You have to do a lot of manual work to expose the different software to the user via a custom runscript (and be a generally decent programmer).
- All software must share the same metadata, environment, and labels.

Under these conditions, containers are at best black boxes with unclear delineation between software provided, and only one context of running anything. The container creator shouldn't need to spend inordinate amounts of time writing custom runscripts to support multiple functions and inputs. Each of `foo` and `bar` should be easy to define, and have its own runscript, environment, labels, tests and help section.

13.1.2 Container Transparency

Applications that use the SCI-F make `foo` and `bar` transparent, and solve the problem of mixed up modules. Our simple issue of mixed up modules could be solved if we could do this:

```
Bootstrap:docker

From: ubuntu:16.04
```

(continues on next page)

(continued from previous page)

```
%appenv foo

    BEST_GUY=foo

    export BEST_GUY

%appenv bar

    BEST_GUY=bar

    export BEST_GUY

%apprun foo

    echo The best guy is $BEST_GUY

%apprun bar

    echo The best guy is $BEST_GUY
```

and generate the container as follows:

```
$ sudo singularity build foobar.simg Singularity
```

and finally, run the container with the context of `foo` and then `bar`

```
$ singularity run --app bar foobar.simg

The best guy is bar

$ singularity run --app foo foobar.simg

The best guy is foo
```

Using SCI-F apps, a user can easily discover both `foo` and `bar` without knowing anything about the container:

```
singularity apps foobar.simg

bar

foo
```

Each applications can be inspected individually:

```
singularity inspect --app foo  foobar.simg

{

    "SCIF_APP_NAME": "foo",

    "SCIF_APP_SIZE": "1MB"
```

(continues on next page)

(continued from previous page)

```
}
```

13.1.3 Container Modularity

This behavior is made possible by a simple, clean organization that is tied to a set of sections in the build recipe relevant to each app. For example, I can specify custom install procedures (and they are relevant to each app's specific base defined under `/scif/apps`), labels, tests, and help sections. Before we examine the sections, consider what the organization looks like, for each app:

```
/scif/apps/


     foo/

        bin/

        lib/

        scif/

            runscript.help

            runscript

            env/

                01-base.sh

                90-environment.sh


     bar/

     ....
```

If you are familiar with Singularity, the above should look similar to other environments. It mirrors the Singularity (main container) metadata folder, except instead of `.singularity.d` we have `scif`. The name and base `scif` is chosen intentionally to be something short, and likely to be unique. On the level of organization and metadata, these internal apps are like little containers.

You need to remember all the path details because you can environment variables in your runscripts, etc. Here we are looking at the environment active for lolcat:

```
singularity exec --app foo foobar.simg env | grep foo
```

Consider the output of the above in sections, you will notice some interesting things. First, notice that the app's `bin` has been added to the path, and its `lib` folder is added to the `LD_LIBRARY_PATH` . This means that anything you drop in either will automatically be added. You don't need to make these folders either, they are created for you.

```
LD_LIBRARY_PATH=/scif/apps/foo/lib::/.singularity.d/libs

PATH=/scif/apps/foo/bin:/scif/apps/foo:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
↪bin:/sbin:/bin
```

Next, notice the environment variables relevant to the active app's (foo) data and metadata. They will look like the following:

```
SCIF_APPOUTPUT=/scif/data/foo/output

SCIF_APPDATA=/scif/data/foo

SCIF_APPINPUT=/scif/data/foo/input

SCIF_APPMETA=/scif/apps/foo/scif

SCIF_APPROOT=/scif/apps/foo

SCIF_APPNAME=foo
```

We also have foo's environment variables defined under `%appenv foo` , and importantly, we don't see bar's.

```
BEST_GUY=foo
```

Also provided are more global paths for data and apps:

```
SCIF_APPS=/scif/apps

SCIF_DATA=/scif/data
```

Note, each application has its own modular location. When you do an `%appinstall foo`, the commands are all done in context of that base. The bin and lib are also automatically generated. Consider a simple application:

Just add a script and name it:

```
%appfiles foo

    runfoo.sh   bin/runfoo.sh
```

and then maybe for install I'd make it executable

```
%appinstall foo

    chmod u+x bin/runfoo.sh
```

You don't even need files! You could just do this.

```
%appinstall foo

    echo 'echo "Hello Foo."' >> bin/runfoo.sh

    chmod u+x bin/runfoo.sh
```

We can summarize these observations about using apps:

- the specific environment (`%appenv_foo`) is active because `BEST_APP` is foo
- the lib folder in foo's base is added to the LD_LIBRARY_PATH
- the bin folder is added to the path
- locations for input, output, and general data are exposed. It's up to you how you use these, but you can predictably know that a well made application will look for inputs and outputs in its specific folder.
- environment variables are provided for the app's root, its data, and its name

13.1.4 Sections

Finding the section `%appinstall` , `%apphelp` , or `%apprun` is indication of an application command. The following string is parsed as the name of the application, and this folder is created, in lowercase, under `/scif/apps` if it doesn't exist. A singularity metadata folder, .singularity.d, equivalent to the container's main folder, is generated inside the application. An application thus is like a smaller image inside of its parent. Specifically, SCI-F defines the following new sections for the build recipe, where each is optional for zero or more apps:

%appinstall corresponds to executing commands within the folder to install the application. These commands would previously belong in %post, but are now attributable to the application.

%apphelp is written as a file called runscript.help in the application's metadata folder, where the Singularity software knows where to find it. If no help section is provided, the software simply will alert the user and show the files provided for inspection.

%apprun is also written as a file called runscript.exec in the application's metadata folder, and again looked for when the user asks to run the software. If not found, the container should default to shelling into that location.

%applabels will write a labels.json in the application's metadata folder, allowing for application specific labels.

%appenv will write an environment file in the application's base folder, allowing for definition of application specific environment variables.

%apptest will run tests specific to the application, with present working directory assumed to be the software module's folder

%appfiles will add files to the app's base at `/scif/apps/<app>`

13.1.5 Interaction

The complete output of a `grep` to the environment when running foo in the first example was not shown because the remainder of variables are more germane to a discussion about app interaction. Essentially, when any application is active, we also have named variable that can explicitly reference the environment file, labels file, runscript, `lib` and `bin` folders for all app's in the container. For our above Singularity Recipe, we would also find:

```
SCIF_APPDATA_bar=/scif/data/bar

SCIF_APPRUN_bar=/scif/apps/bar/scif/runscript

SCIF_APPROOT_bar=/scif/apps/bar

SCIF_APPLIB_bar=/scif/apps/bar/lib

SCIF_APPMETA_bar=/scif/apps/bar/scif

SCIF_APPBIN_bar=/scif/apps/bar/bin

SCIF_APPENV_bar=/scif/apps/bar/scif/env/90-environment.sh

SCIF_APPLABELS_bar=/scif/apps/bar/scif/labels.json


SCIF_APPENV_foo=/scif/apps/foo/scif/env/90-environment.sh

SCIF_APPLABELS_foo=/scif/apps/foo/scif/labels.json

SCIF_APPDATA_foo=/scif/data/foo
```

(continues on next page)

(continued from previous page)

```
SCIF_APPRUN_foo=/scif/apps/foo/scif/runscript

SCIF_APPROOT_foo=/scif/apps/foo

SCIF_APPLIB_foo=/scif/apps/foo/lib

SCIF_APPMETA_foo=/scif/apps/foo/scif

SCIF_APPBIN_foo=/scif/apps/foo/bin
```

This design means that we can have apps interact with one another internally. For example, let's modify the recipe a bit:

```
Bootstrap:docker

From: ubuntu:16.04


%appenv cow

    ANIMAL=COW

    NOISE=moo

    export ANIMAL NOISE


%appenv bird

    NOISE=tweet

    ANIMAL=BIRD

    export ANIMAL


%apprun moo

    echo The ${ANIMAL} goes ${NOISE}


%appenv moo

    . ${APPENV_cow}
```

In the above example, we have three apps. One for a cow, one for a bird, and a third that depends on the cow. We can't define global functions or environment variables (in `%post` or `/environment` , respectively) because they would interfere with the third app, bird, that has equivalently named variables. What we do then, is source the environment for "cow" in the environment for "moo" and the result is what we would want:

```
$ singularity run --app moo /tmp/one.simg

The COW goes moo
```

The same is true for each of the labels, environment, runscript, bin, and lib. The following variables are available to you, for each application in the container, whenever any application is run:

- **SCIF_APPBIN_**: the path to the bin folder, if you want to add an application that isn't active to your `PATH`
- **SCIF_APPLIB_**: the path to the lib folder, if you want to add an application that isn't active to your `LD_LIBRARY_PATH`
- **SCIF_APPRUN_**: the application's runscript (so you can call it from elsewhere)
- **SCIF_APPMETA_**: the path to the metadata folder for the application
- **SCIF_APPENV_**: the path to the primary environment file (for sourcing) if it exists
- **SCIF_APPROOT_**: the application's install folder
- **SCIF_APPDATA_**: the application's data folder
- **SCIF_APPLABELS_**: The path to the label.json in the metadata folder, if it exists

Singularity containers are already reproducible in that they package dependencies. The basic format of SCI-F adds to that by making the software inside of containers modular, predictable, and programmatically accessible.

By pre-setting some set of steps, labels, or variables in the runscript is associated with a particular action of the container, users can better encapsulate how dependencies relate to each step in a scientific work-flow.

Making containers can be challenging. When a scientist starts to write a recipe for his set of tools, she probably doesn't know where to put various tags and data. The SCI-F file system makes it easy to build consistent maintainable containers.

13.2 SCI-F Example: Cowsay Container

As an example, we will use the **'cowsay container'_**. `cowsay` is a program that generates ASCII pictures of a cow with a message. It also uses the `fortune``program to produce random "fortunes" and the ``lolcat` applications that add rainbow color to an ASCII string.

> **Warning: Important!** This example has been developed for Singularity 2.4.

Download the recipe, and save it to your present working directory.

```
wget https://raw.githubusercontent.com/sylabs/singularity/master/examples/apps/
↪Singularity.cowsay

sudo singularity build moo.simg Singularity.cowsay
```

Determine what applications are installed using the following command:

```
singularity apps moo.simg

cowsay

fortune

lolcat
```

Ask for help for a specific application as follows:

```
singularity help --app fortune moo.simg

fortune is the best app
```

A simple loop can be used to ask for help from all apps (without asking in advance what they are):

```
for app in $(singularity apps moo.simg)

   do

     singularity help --app $app moo.simg

done

cowsay is the best app

fortune is the best app

lolcat is the best app
```

To run the `fortune` application, enter the following (the actual fortune is random, so your display may differ):

```
singularity run --app fortune moo.simg

    My dear People.

    My dear Bagginses and Boffins, and my dear Tooks and Brandybucks,

and Grubbs, and Chubbs, and Burrowses, and Hornblowers, and Bolgers,

Bracegirdles, Goodbodies, Brockhouses and Proudfoots.  Also my good

Sackville Bagginses that I welcome back at last to Bag End.  Today is my

one hundred and eleventh birthday: I am eleventy-one today!"

        -- J. R. R. Tolkien
```

Next, pipe the output of `fortune` into `lolcat` to add color to the fortune.

```
singularity run --app fortune moo.simg | singularity run --app lolcat moo.simg

You will be surrounded by luxury.
```

Send the output of `fortune```to the ``cowsay` application.

```
singularity run --app fortune moo.simg | singularity run --app cowsay moo.simg

 ________________________________________

/ Executive ability is prominent in your \

\ make-up.                               /

 ----------------------------------------

        \   ^__^

         \  (oo)\_______

            (__)\       )\/\

```

(continues on next page)

(continued from previous page)

```
                ||----w |
                ||     ||
```

Finally use all three applications and demonstrate how to use an environment variable for the command:

```
CMD="singularity run --app"
$CMD fortune moo.simg | $CMD cowsay moo.simg | $CMD lolcat moo.simg
 _________________________________________
/ Ships are safe in harbor, but they were \
\ never meant to stay there.              /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

The application can be inspected with the following command:

```
singularity inspect --app fortune moo.simg
{
    "SCIF_APP_NAME": "fortune",
    "SCIF_APP_SIZE": "1MB"
}
```

If you want to see the full specification or create your own Scientific Filesystem integration (doesn't have to be Singularity, or Docker, or containers!) see the full documentation.

Also, you can follow along with this example by going to: take a look at these examples

CHAPTER

FOURTEEN

SINGULARITY AND DOCKER

Singularity can be used with Docker images. This feature was included because developers use and really like using Docker and scientists have already put much resources into creating Docker images. Thus, one of our early goals was to support Docker. What can you do?

- You don't need Docker installed
- You can shell into a Singularity-ized Docker image
- You can run a Docker image instantly as a Singularity image
- You can pull a Docker image (without sudo)
- You can build images with bases from assembled Docker layers that include environment, guts, and labels

14.1 TLDR (Too Long Didn't Read)

You can shell, import, run, and exec Docker images directly from the Docker Registry.

```
singularity shell docker://ubuntu:latest

singularity run docker://ubuntu:latest

singularity exec docker://ubuntu:latest echo "Hello Dinosaur!"


singularity pull docker://ubuntu:latest

singularity build ubuntu.img docker://ubuntu:latest
```

14.2 Import a Docker image into a Singularity Image

The core of a Docker image is basically a compressed set of files, a set of `.tar.gz` that (if you look in your Docker image folder on your host machine, you will see the files. The Docker Registry, which you probably interact with via Docker Hub, serves these layers. These are the layers that you see downloading when you interact with the docker daemon. We are going to use these same layers for Singularity!

14.3 Quick Start: The Docker Registry

The Docker engine communicates with the Docker Hub via the Docker Remote API, and so can Singularity. The easiest thing to do is create an image, and then pipe a Docker image directly into it from the Docker Registry. You don't need Docker installed on your machine, but you will need a working Internet connection. Let's create an ubuntu operating system, from Docker. We will pull, then build:

```
singularity pull docker://ubuntu

WARNING: pull for Docker Hub is not guaranteed to produce the

WARNING: same image on repeated pull. Use Singularity Registry

WARNING: (shub://) to pull exactly equivalent images.

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /home/vanessa/.singularity/docker

[5/5] |===================================| 100.0%

Importing: base Singularity environment

Importing: /home/vanessa/.singularity/docker/
↪sha256:9fb6c798fa41e509b58bccc5c29654c3ff4648b608f5daa67c1aab6a7d02c118.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:3b61febd4aefe982e0cb9c696d415137384d1a01052b50a85aae46439e15e49a.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:9d99b9777eb02b8943c0e72d7a7baec5c782f8fd976825c9d3fb48b3101aacc2.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:d010c8cf75d7eb5d2504d5ffa0d19696e8d745a457dd8d28ec6dd41d3763617e.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:7fac07fb303e0589b9c23e6f49d5dc1ff9d6f3c8c88cabe768b430bdb47f03a9.tar.gz

Importing: /home/vanessa/.singularity/metadata/
↪sha256:77cece4ce6ef220f66747bb02205a00d9ca5ad0c0a6eea1760d34c744ef7b231.tar.gz

WARNING: Building container as an unprivileged user. If you run this container as root

WARNING: it may be missing some functionality.

Building Singularity image...

Cleaning up...

Singularity container built: ./ubuntu.img
```

The warnings are reminding you that you are creating the image on the fly from layers, and if one of those layers changes, you won't produce the same image next time.

14.4 The Build Specification file, Singularity

Just like Docker has the Dockerfile, Singularity has a file called Singularity that (currently) applications like Singularity Hub know to find. For reproducibility of your containers, our strong recommendation is that you build from these files. Any command that you issue to change a container sandbox (building with `--sandbox`) or to a build with `--writable` is by default not recorded, and your container loses its reproducibility. The following are steps to these files. First, let's look at the absolute minimum requirement:

```
Bootstrap: docker

From: ubuntu
```

We save this content to a file called Singularity and then issue the following commands to bootstrap the image from the file:

```
sudo singularity build ubuntu.img Singularity
```

A particular tag or version can be added to the docker uri:

```
Bootstrap: docker

From: ubuntu:latest
```

Note: Note that the default is `latest` . If you want to customize the Registry or Namespace, just add those to the header:

```
Bootstrap: docker

From: ubuntu

Registry: pancakes.registry.index.io

Namespace: blue/berry/cream
```

The power of build comes with the other things that you can do. This means running specific install commands, specifying your containers runscript (what it does when you execute it), adding files, labels, and customizing the environment. Here is a full Singularity file:

```
Bootstrap: docker

From: tensorflow/tensorflow:latest


%runscript

    exec /usr/bin/python "$@"


%post

    echo "Post install stuffs!"


%files
```

(continues on next page)

(continued from previous page)

```
/home/vanessa/Desktop/analysis.py /tmp/analysis.py

relative_path.py /tmp/analysis2.py


%environment

TOPSECRET=pancakes

HELLO=WORLD

export HELLO TOPSECRET


%labels

AUTHOR Vanessasaur
```

In the example above, I am overriding any Dockerfile `ENTRYPOINT` or `CMD` because I have defined a `%runscript` . If I want the Dockerfile `ENTRYPOINT` to take preference, I would remove the `%runscript` section. If I want to use `CMD` instead of `ENTRYPOINT` , I would again remove the runscript, and add IncludeCmd to the header:

```
Bootstrap: docker

From: tensorflow/tensorflow:latest

IncludeCmd: yes


%post


    echo "Post install stuffs!"
```

You can commit this Singularity file to a GitHub repo and it will automatically build for you when you push to Singularity Hub?. This step will ensure maximum reproducibility of your work.

14.5 How does the runscript work?

Docker has two commands in the `Dockerfile` that have something to do with execution, `CMD` and `ENTRYPOINT`. The differences are subtle, but the a good description is the following:

> A `CMD` is to provide defaults for an executing container.

and

> An `ENTRYPOINT` helps you to configure a container that you can run as an executable.

Given the definition, the `ENTRYPOINT` is most appropriate for the Singularity `%runscript` , and so using the default bootstrap (whether from a `docker://` endpoint or a `Singularity` spec file) will set the `ENTRYPOINT` variable as the runscript. You can change this behavior by specifying `IncludeCmd:  yes` in the Spec file (see below). If you provide any sort of `%runscript` in your Spec file, this overrides anything provided in Docker. In summary, the order of operations is as follows:

1. If a `%runscript` is specified in the Singularity spec file, this takes prevalence over all

2. If no %runscript is specified, or if the import command is used as in the example above, the ENTRYPOINT is used as runscript.
3. If no %runscript is specified, but the user has a Singularity spec with IncludeCmd , then the Docker CMD is used.
4. If no %runscript is specified, and there is no CMD or ENTRYPOINT , the image's default execution action is to run the bash shell.

14.6 How do I specify my Docker image?

In the example above, you probably saw that we referenced the docker image first with the uri docker:// and that is important to tell Singularity that it will be pulling Docker layers. To ask for ubuntu, we asked for docker:// ubuntu . This uri that we give to Singularity is going to be very important to choose the following Docker metadata items:

- registry (e.g., "index.docker.io")
- namespace (e.g., "library")
- repository (e.g., "ubuntu")
- tag (e.g., "latest") OR version (e.g., "@sha256:1234. . .)

When we put those things together, it looks like this:

```
docker://<registry>/<namespace>/<repo_name>:<repo_tag>
```

By default, the minimum requirement is that you specify a repository name (eg, ubuntu) and it will default to the following:

```
docker://index.docker.io/library/ubuntu:latest
```

If you provide a version instead of a tag, that will be used instead:

```
docker://index.docker.io/library/ubuntu@sha256:1235...
```

You can have one or the other, both are considered a "digest" in Docker speak.

If you want to change any of those fields and are having trouble with the uri, you can also just state them explicitly:

```
Bootstrap: docker

From: ubuntu

Registry: index.docker.io

Namespace: library
```

14.7 Custom Authentication

For both import and build using a build spec file, by default we use the Docker Registry index.docker.io . Singularity first tries the call without a token, and then asks for one with pull permissions if the request is defined. However, it may be the case that you want to provide a custom token for a private registry. You have two options. You can either provide a Username and Password in the build specification file (if stored locally and there is no

need to share), or (in the case of doing an import or needing to secure the credentials) you can export these variables to environmental variables. We provide instructions for each of these cases:

14.7.1 Authentication in the Singularity Build File

You can simply specify your additional authentication parameters in the header with the labels `Username` and `Password:`

```
Username: vanessa

Password: [password]
```

Again, this can be in addition to specification of a custom registry with the `Registry` parameter.

14.7.2 Authentication in the Environment

You can export your username, and password for Singularity as follows:

```
export SINGULARITY_DOCKER_USERNAME=vanessasaur

export SINGULARITY_DOCKER_PASSWORD=rawwwwwr
```

14.7.3 Testing Authentication

If you are having trouble, you can test your token by obtaining it on the command line and putting it into an environmental variable, `CREDENTIAL` :

```
CREDENTIAL=$(echo -n vanessa:[password] | base64)

TOKEN=$(http 'https://auth.docker.io/token?service=registry.docker.io&
↪scope=repository:vanessa/code-samples:pull' Authorization:"Basic $CREDENTIAL" | jq -
↪r '.token')
```

This should place the token in the environmental variable `TOKEN` . To test that your token is valid, you can do the following

```
http https://index.docker.io/v2/vanessa/code-samples/tags/list Authorization:"Bearer
↪$TOKEN"
```

The above call should return the tags list as expected. And of course you should change the repository (repo) name to be one that actually exists that you have credentials for.

14.8 Best Practices

While most docker images can import and run without a hitch, there are some special cases for which things can go wrong. Here is a general list of suggested practices, and if you discover a new one in your building ventures please let us know.

14.8.1 1. Installation to Root

When using Docker, you typically run as root, meaning that root's home at `/root` is where things will install given a specification of home. This situation is fine when you stay in Docker, or if the content at `/root` doesn't need any kind of write access, but generally it can lead to a lot of bugs because it is, after all, root's home. This leads us to best practice #1.

Don't install anything to root's home, `/root`.

14.8.2 2. Library Configurations

The command ldconfig is used to update the shared library cache. If you have software that requires symbolic linking of libraries and you do the installation without updating the cache, then the Singularity image (in read only) will likely give you an error that the library is not found. If you look in the image, the library will exist but the symbolic link will not. This leads us to best practice #2:

Update the library cache at the end of your Dockerfile with a call to ldconfig.

14.8.3 3. Don't install to $HOME or $TMP

We can assume that the most common Singularity use case has the $USER home being automatically mounted to `$HOME`, and `$TMP` also mounted. Thus, given the potential for some kind of conflict or missing files, for best practice #3 we suggest the following:

Don't put container valuables in `$TMP` or `$HOME`

Have any more best practices? Please let us know!

14.9 Troubleshooting

Why won't my image build work? If you can't find an answer on this documentation, please send us an issue. If you've found an answer and you'd like to see it on the site for others to benefit from, then post to us here.

CHAPTER

FIFTEEN

TROUBLESHOOTING

A little bit of help.

15.1 No space left on device

Sometimes when you are building an image, Singularity tells you that it runs out of space on the device:

```
sudo singularity build fatty.simg Singularity

IOError: [Errno 28] No space left on device

ABORT: Aborting with RETVAL=255
```

The issue here is that during build of a squashfs image, Singularity is using the `$TMPDIR` . If your `$TMPDIR` is overflowing (or the mount is very small!) then you would see this error. As a test, you can try building a sandbox. If this is the issue, then the sandbox should work.

```
sudo singularity build --sandbox [fatty] Singularity
```

Solution You simply need to set the `$SINGULARITY_CACHEDIR` to a different location that you have more room.

15.2 Segfault on Bootstrap of Centos Image

If you are bootstrapping a centos 6 docker image from a debian host, you might hit a segfault:

```
$ singularity shell docker://centos:6

Docker image path: index.docker.io/library/centos:6

Cache folder set to /home/jbdenis/.singularity/docker

Creating container runtime...

Singularity: Invoking an interactive shell within container...


Segmentation fault
```

The fix is on your host, you need to pass the variable `vsyscall=emulate` to the kernel, meaning in the file `/etc/default/grub` (note, this file is debian specific), add the following:

```
GRUB_CMDLINE_LINUX_DEFAULT="vsyscall=emulate"
```

and then update grub and reboot:

```
update-grub && reboot
```

Please note that this change might have security implications that you should be aware of. For more information, see the original issue.

15.3 How to use Singularity with GRSecurity enabled kernels

To run Singularity on a GRSecurity enabled kernel, you must disable several security features:

```
$ sudo sysctl -w kernel.grsecurity.chroot_caps=0

$ sudo sysctl -w kernel.grsecurity.chroot_deny_mount=0

$ sudo sysctl -w kernel.grsecurity.chroot_deny_chmod=0

$ sudo sysctl -w kernel.grsecurity.chroot_deny_fchdir=0
```

15.4 The container isn't working on a different host!

Singularity by default mounts your home directory. While this is great for seamless communication between your host and the container, it can introduce issues if you have software modules installed at `$HOME`. For example, we had a user run into this issue.

Solution 1: Specify the home to mount A first thing to try is to point to some "sanitized home," which is the purpose of the -H or –home option. For example, here we are creating a home directory under /tmp/homie, and then telling the container to mount it as home:

```
rm -rf /tmp/homie && mkdir -p /tmp/homie && \

singularity exec -H /tmp/homie analysis.img /bin/bash
```

Solution 2: Specify the executable to use It may be the issue that there is an executable in your host environment (e.g. python) that is being called in preference to the containers. To avoid this, in your runscript (the `%runscript` section of the bootstrap file) you should specify the path to the executable exactly. This means:

```
%runscript


# This specifies the python in the container

exec /usr/bin/python "$@"


# This may pick up a different one

exec python "$@"
```

This same idea would be useful if you are issuing the command to the container using `exec`.

15.5 Invalid Argument or Unknown Option

When I try mounting my container with the `-B` or `--bind` option I receive an unknown option or Invalid argument error. Make sure that you are using the most recent Singularity release to mount your container to the host system, and that the `--bind` argument is placed after the execution command. An example might look like this:

```
$ singularity run -B $PWD:/data my_container.img
```

Also, make sure you are using an up-to-date Singularity to bootstrap your container. Some features (such as `--bind`) will not work in earlier versions.

15.6 Error running Singularity with sudo

This fix solves the following error when Singularity is installed into the default compiled prefix of /usr/local:

```
$ sudo singularity instance.start container.img daemon1

sudo: singularity: command not found
```

The cause of the problem is that `sudo` sanitizes the PATH environment variable and does not include `/usr/local/bin` in the default search path. Considering this program path is by default owned by root, it is reasonable to extend the default sudo PATH to include this directory. To add `/usr/local/bin` to the default sudo search path, run the program `visudo` which will edit the sudoers file, and search for the string 'secure_path'. Once found, append `:/usr/local/bin` to that line so it looks like this:

```
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin
```

15.7 How to resolve "Too many levels of symbolic links" error

Running singularity failed with "Too many levels of symbolic links" error

```
$ singularity run -B /apps container.img

ERROR : There was an error binding the path /apps: Too many levels of symbolic links

ABORT : Retval = 255
```

You got this error because `/apps` directory is an autofs mount point. You can fix it by editing singularity.conf and adding the following directive with corresponding path:

```
autofs bug path = /apps
```

CHAPTER

SIXTEEN

APPENDIX

16.1 build-docker-module

16.1.1 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.

16.1.2 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.

16.1.3 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 squashfs or ext3 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 you, consider hosting a base container on Singularity Hub and building from it instead.

For detailed information about setting your build environment see *Build Customization*.

16.2 build-shub

16.2.1 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.

16.2.2 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.

16.2.3 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.

16.3 build-localimage

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.

16.3.1 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.

16.3.2 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.

16.3.3 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.

16.4 build-yum

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

16.4.1 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.

16.4.2 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 URL to use as a mirror to download the OS. If you define the `OSVersion` keyword, than you can use it in the URL 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.

16.4.3 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 `yum` build module, you must have `yum` installed on your system. It may seem counter-intuitive to install YUM on a system that uses a different package manager, but you can do so. For instance, on Ubuntu you can install it like so:

```
$ sudo apt-get update && sudo apt-get install yum
```

16.5 build-debootstrap

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

16.5.1 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.

16.5.2 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 URL 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.

16.5.3 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
```

16.6 build-arch

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

16.6.1 Overview

Use the `arch` module to specify a base for an Arch Linux based container. Arch Linux uses the aptly named the `pacman` package manager (all puns intended).

16.6.2 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.

16.6.3 Notes

Arch Linux is, by design, a very stripped down, light-weight OS. You may need to perform a fair amount of configuration to get a usable OS. Please refer to this README.md and the Arch Linux example for more info.

16.7 build-busybox

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

16.7.1 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.

16.7.2 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 URL to use as a mirror when downloading the OS.

16.7.3 Notes

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

16.8 build-zypper

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

16.8.1 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.

16.8.2 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.

16.9 Singularity Action Flags

For each of `exec`, `run`, and `shell`, there are a few important flags that we want to note for new users that have substantial impact on using your container. While we won't include the complete list of run options (for this complete list see `singularity run --help` or more generally `singularity <action> --help`) we will review some highly useful flags that you can add to these actions.

- **–contain**: Contain suggests that we want to better isolate the container runtime from the host. Adding the `--contain` flag will use minimal `/dev` and empty other directories (e.g., `/tmp`).
- **–containall**: In addition to what is provided with `--contain` (filesystems) also contain PID, IPC, and environment.
- **–cleanenv**: Clean the environment before running the container.
- **–pwd**: Initial working directory for payload process inside the container.

This is **not** a complete list! Please see the `singularity <action> help` for an updated list.

16.9.1 Examples

Here we are cleaning the environment. In the first command, we see that the variable `PEANUTBUTTER` gets passed into the container.

```
PEANUTBUTTER=JELLY singularity exec Centos7.img env | grep PEANUT

PEANUTBUTTER=JELLY
```

And now here we add `--cleanenv` to see that it doesn't.

```
PEANUTBUTTER=JELLY singularity exec --cleanenv Centos7.img env | grep PEANUT
```

Here we will test contain. We can first confirm that there are a lot of files on our host in /tmp, and the same files are found in the container.

```
# On the host

$ ls /tmp | wc -l

17


# And then /tmp is mounted to the container, by default

$ singularity exec Centos7.img  ls /tmp | wc -l


# ..but not if we use --contain

$ singularity exec --contain Centos7.img  ls /tmp | wc -l

0
```

16.10 Commands

16.10.1 Command Usage

16.10.1.1 The Singularity command

Singularity uses a primary command wrapper called `singularity`. When you run `singularity` without any options or arguments it will dump the high level usage syntax.

The general usage form is:

```
$ singularity (opts1) [subcommand] (opts2) ...
```

If you type `singularity` without any arguments, you will see a high level help for all arguments. The main options include: **Container Actions**

- *build* : Build a container on your user endpoint or build environment
- *exec* : Execute a command to your container
- *inspect* : See labels, run and test scripts, and environment variables
- *pull* : pull an image from Docker or Singularity Hub
- *run* : Run your image as an executable
- *shell* : Shell into your image

Image Commands

- *image.import* : import layers or other file content to your image
- *image.export* : export the contents of the image to tar or stream
- *image.create* : create a new image, using the old ext3 filesystem
- *image.expand* : increase the size of your image (old ext3)

Instance Commands

Instances were added in 2.4. This list is brief, and likely to expand with further development.

- *instances* : Start, stop, and list container instances

Deprecated Commands The following commands are deprecated in 2.4 and will be removed in future releases.

- *bootstrap* : Bootstrap a container recipe

For the full usage, *see the bottom of this page*

16.10.1.1.1 Options and argument processing

Because of the nature of how Singularity cascades commands and sub-commands, argument processing is done with a mandatory order. **This means that where you place arguments is important!** In the above usage example, `opts1` are the global Singularity run-time options. These options are always applicable no matter what subcommand you select (e.g. `--verbose` or `--debug`). But subcommand specific options must be passed after the relevant subcommand.

To further clarify this example, the `exec` Singularity subcommand will execute a program within the container and pass the arguments passed to the program. So to mitigate any argument clashes, Singularity must not interpret or interfere with any of the command arguments or options that are not relevant for that particular function.

16.10.1.1.2 Singularity Help

Singularity comes with some internal documentation by using the `help` subcommand followed by the subcommand you want more information about. For example:

```
$ singularity help create

CREATE OPTIONS:

    -s/--size   Specify a size for an operation in MiB, i.e. 1024*1024B

                (default 768MiB)

    -F/--force  Overwrite an image file if it exists


EXAMPLES:


    $ singularity create /tmp/Debian.img

    $ singularity create -s 4096 /tmp/Debian.img


For additional help, please visit our public documentation pages which are

found at:


    https://www.sylabs.io/docs/
```

16.10.1.2 Commands Usage

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


GLOBAL OPTIONS:

    -d|--debug    Print debugging information

    -h|--help     Display usage summary

    -s|--silent   Only print errors

    -q|--quiet    Suppress all normal output

       --version  Show application version

    -v|--verbose  Increase verbosity +1

    -x|--sh-debug Print shell wrapper debugging information


GENERAL COMMANDS:

    help       Show additional help for a command or container
```

(continues on next page)

(continued from previous page)

```
    selftest   Run some self tests for singularity install

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

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

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


CONTAINER USAGE OPTIONS:
    see singularity help <command>

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

16.10.1.3 Support

Have a question, or need further information? Reach out to us.

16.10.2 build

Use `build` to download and assemble existing containers, convert containers from one format to another, or build a container from a *Singularity recipe*.

16.10.2.1 Overview

The `build` command accepts a target as input and produces a container as output. The target can be a Singularity Hub or Docker Hub URI, a path to an existing container, or a path to a Singularity Recipe file. The output container can be in squashfs, ext3, or directory format.

For a complete list of `build` options type `singularity help build`. For more info on building containers see *Build a Container*.

16.10.2.2 Examples

16.10.2.2.1 Download an existing container from Singularity Hub or Docker Hub

```
$ singularity build lolcow.simg shub://GodloveD/lolcow

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

16.10.2.2.2 Create –writable images and –sandbox directories

```
$ sudo singularity build --writable lolcow.img shub://GodloveD/lolcow

$ sudo singularity build --sandbox lolcow/ shub://GodloveD/lolcow
```

16.10.2.2.3 Convert containers from one format to another

You can convert the three supported container formats using any combination.

```
$ sudo singularity build --writable development.img production.simg

$ singularity build --sandbox development/ production.simg

$ singularity build production2 development/
```

16.10.2.2.4 Build a container from a Singularity recipe

Given a Singularity Recipe called `Singularity` :

```
$ sudo singularity build lolcow.simg Singularity
```

16.10.3 exec

The `exec` Singularity sub-command allows you to spawn an arbitrary command within your container image as if it were running directly on the host system. All standard IO, pipes, and file systems are accessible via the command being exec'ed within the container. Note that this exec is different from the Docker exec, as it does not require a container to be "running" before using it.

16.10.3.1 Examples

16.10.3.1.1 Printing the OS release inside the container

```
$ singularity exec container.img cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 8 (jessie)"

NAME="Debian GNU/Linux"

VERSION_ID="8"

VERSION="8 (jessie)"

ID=debian

HOME_URL="http://www.debian.org/"

SUPPORT_URL="http://www.debian.org/support"

BUG_REPORT_URL="https://bugs.debian.org/"

$
```

16.10.3.1.2 Printing the OS release for a running instance

Use the `instance://<instance name>` syntax like so:

```
$ singularity exec instance://my-instance cat /etc/os-release
```

16.10.3.1.3 Runtime Flags

If you are interested in containing an environment or filesystem locations, we highly recommend that you look at the `singularity run help` and our documentation on *flags* to better customize this command.

16.10.3.1.4 Special Characters

And properly passing along special characters to the program within the container.

```
$ singularity exec container.img echo -ne "hello\nworld\n\n"

hello
```

(continues on next page)

(continued from previous page)

```
world

$
```

And a demonstration using pipes:

```
$ cat debian.def | singularity exec container.img grep 'MirrorURL'

MirrorURL "http://ftp.us.debian.org/debian/"

$
```

16.10.3.1.5 A Python example

Starting with the file `hello.py` in the current directory with the contents of:

```
#!/usr/bin/python


import sys

print("Hello World: The Python version is %s.%s.%s" % sys.version_info[:3])
```

Because our home directory is automatically bound into the container, and we are running this from our home directory, we can easily execute that script using the Python within the container:

```
$ singularity exec /tmp/Centos7-ompi.img /usr/bin/python hello.py

Hello World: The Python version is 2.7.5
```

We can also pipe that script through the container and into the Python binary which exists inside the container using the following command:

```
$ cat hello.py | singularity exec /tmp/Centos7-ompi.img /usr/bin/python

Hello World: The Python version is 2.7.5
```

For demonstration purposes, let's also try to use the latest Python container which exists in DockerHub to run this script:

```
$ singularity exec docker://python:latest /usr/local/bin/python hello.py

library/python:latest

Downloading layer: 
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

Downloading layer: 
↪sha256:fbd06356349dd9fb6af91f98c398c0c5d05730a9996bbf88ff2f2067d59c70c4

Downloading layer: 
↪sha256:644eaeceac9ff6195008c1e20dd693346c35b0b65b9a90b3bcba18ea4bcef071

Downloading layer: 
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
```

(continues on next page)

(continued from previous page)

```
Downloading layer:␣
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

Downloading layer:␣
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

Downloading layer:␣
↪sha256:766692404ca72f4e31e248eb82f8eca6b2fcc15b22930ec50e3804cc3efe0aba

Downloading layer:␣
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

Downloading layer:␣
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

Downloading layer:␣
↪sha256:6a3d69edbe90ef916e1ecd8d197f056de873ed08bcfd55a1cd0b43588f3dbb9a

Downloading layer:␣
↪sha256:ff18e19c2db42055e6f34323700737bde3c819b413997cddace2c1b7180d7efd

Downloading layer:␣
↪sha256:7b9457ec39de00bc70af1c9631b9ae6ede5a3ab715e6492c0a2641868ec1deda

Downloading layer:␣
↪sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

Downloading layer:␣
↪sha256:6a5a5368e0c2d3e5909184fa28ddfd56072e7ff3ee9a945876f7eee5896ef5bb

Hello World: The Python version is 3.5.2
```

16.10.3.1.6 A GPU example

If your host system has an NVIDIA GPU card and a driver installed you can leverage the card with the `--nv` option. (This example requires a fairly recent version of the NVIDIA driver on the host system to run the latest version of TensorFlow.

```
$ git clone https://github.com/tensorflow/models.git

$ singularity exec --nv docker://tensorflow/tensorflow:latest-gpu \

    python ./models/tutorials/image/mnist/convolutional.py

Docker image path: index.docker.io/tensorflow/tensorflow:latest-gpu

Cache folder set to /home/david/.singularity/docker

[19/19] |===================================| 100.0%

Creating container runtime...

Extracting data/train-images-idx3-ubyte.gz
```

(continues on next page)

(continued from previous page)

```
Extracting data/train-labels-idx1-ubyte.gz

Extracting data/t10k-images-idx3-ubyte.gz

Extracting data/t10k-labels-idx1-ubyte.gz

2017-08-18 20:33:59.677580: W tensorflow/core/platform/cpu_feature_guard.cc:45] The
↪TensorFlow library wasn't compiled to use SSE4.1 instructions, but these are
↪available on your machine and could speed up CPU computations.

2017-08-18 20:33:59.677620: W tensorflow/core/platform/cpu_feature_guard.cc:45] The
↪TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are
↪available on your machine and could speed up CPU computations.

2017-08-18 20:34:00.148531: I tensorflow/stream_executor/cuda/cuda_gpu_executor.
↪cc:893] successful NUMA node read from SysFS had negative value (-1), but there
↪must be at least one NUMA node, so returning NUMA node zero

2017-08-18 20:34:00.148926: I tensorflow/core/common_runtime/gpu/gpu_device.cc:955]
↪Found device 0 with properties:

name: GeForce GTX 760 (192-bit)

major: 3 minor: 0 memoryClockRate (GHz) 0.8885

pciBusID 0000:03:00.0

Total memory: 2.95GiB

Free memory: 2.92GiB

2017-08-18 20:34:00.148954: I tensorflow/core/common_runtime/gpu/gpu_device.cc:976]
↪DMA: 0

2017-08-18 20:34:00.148965: I tensorflow/core/common_runtime/gpu/gpu_device.cc:986]
↪0:   Y

2017-08-18 20:34:00.148979: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1045]
↪Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX 760 (192-bit),
↪pci bus id: 0000:03:00.0)

Initialized!

Step 0 (epoch 0.00), 21.7 ms

Minibatch loss: 8.334, learning rate: 0.010000

Minibatch error: 85.9%

Validation error: 84.6%

Step 100 (epoch 0.12), 20.9 ms

Minibatch loss: 3.235, learning rate: 0.010000

Minibatch error: 4.7%
```

(continues on next page)

(continued from previous page)

```
Validation error: 7.8%

Step 200 (epoch 0.23), 20.5 ms

Minibatch loss: 3.363, learning rate: 0.010000

Minibatch error: 9.4%

Validation error: 4.2%

[...snip...]

Step 8500 (epoch 9.89), 20.5 ms

Minibatch loss: 1.602, learning rate: 0.006302

Minibatch error: 0.0%

Validation error: 0.9%

Test error: 0.8%
```

16.10.4 inspect

How can you sniff an image? We have provided the inspect command for you to easily see the runscript, test script, environment, help, and metadata labels.

This command is essential for making containers understandable by other tools and applications.

16.10.4.1 JSON Api Standard

For any inspect command, by adding `--json` you can be assured to get a JSON API standardized response, for example:

```
singularity inspect -l --json ubuntu.img

{

    "data": {

        "attributes": {

            "labels": {

                "SINGULARITY_DEFFILE_BOOTSTRAP": "docker",

                "SINGULARITY_DEFFILE": "Singularity",

                "SINGULARITY_BOOTSTRAP_VERSION": "2.2.99",

                "SINGULARITY_DEFFILE_FROM": "ubuntu:latest"

            }
```

(continues on next page)

(continued from previous page)

```
        },

        "type": "container"

    }

}
```

16.10.4.2 Inspect Flags

The default, if run without any arguments, will show you the container labels file

```
$ singularity inspect ubuntu.img

{

    "SINGULARITY_DEFFILE_BOOTSTRAP": "docker",

    "SINGULARITY_DEFFILE": "Singularity",

    "SINGULARITY_BOOTSTRAP_VERSION": "2.2.99",

    "SINGULARITY_DEFFILE_FROM": "ubuntu:latest"

}
```

and as outlined in the usage, you can specify to see any combination of `--labels` , `--environment` , `--runscript` , `--test` , and `--deffile`. The quick command to see everything, in json format, would be:

```
$ singularity inspect -l -r -d -t -e -j -hf ubuntu.img

{

    "data": {

        "attributes": {

            "test": null,

            "help": "This is how you run the image!\n",

            "environment": "# Custom environment shell code should follow\n\n",

            "labels": {

                "SINGULARITY_DEFFILE_BOOTSTRAP": "docker",

                "SINGULARITY_DEFFILE": "Singularity",

                "SINGULARITY_BOOTSTRAP_VERSION": "2.2.99",

                "SINGULARITY_DEFFILE_FROM": "ubuntu:latest"

            },
```

(continues on next page)

(continued from previous page)

```
            "deffile": "Bootstrap:docker\nFrom:ubuntu:latest\n",

            "runscript": "#!/bin/sh\n\nexec /bin/bash \"$@\""

        },

        "type": "container"

    }

}
```

16.10.4.2.1 Labels

The default, if run without any arguments, will show you the container labels file (located at `/.singularity.d/labels.json` in the container. These labels are the ones that you define in the `%labels` section of your bootstrap file, along with any Docker `LABEL` that came with an image that you imported, and other metadata about the bootstrap. For example, here we are inspecting labels for `ubuntu.img`

```
$ singularity inspect ubuntu.img

{

    "SINGULARITY_DEFFILE_BOOTSTRAP": "docker",

    "SINGULARITY_DEFFILE": "Singularity",

    "SINGULARITY_BOOTSTRAP_VERSION": "2.2.99",

    "SINGULARITY_DEFFILE_FROM": "ubuntu:latest"

}
```

This is the equivalent of both of:

```
$ singularity inspect -l ubuntu.img

$ singularity inspect --labels ubuntu.img
```

16.10.4.2.2 Runscript

The commands `--runscript` or `--r` will show you the runscript, which also can be shown in `--json`:

```
$ singularity inspect -r -j ubuntu.img{

    "data": {

        "attributes": {

            "runscript": "#!/bin/sh\n\nexec /bin/bash \"$@\""

        },
```

(continues on next page)

(continued from previous page)

```
        "type": "container"

    }

}
```

or in a human friendly, readable print to the screen:

```
$ singularity inspect -r ubuntu.img


##runscript

#!/bin/sh


exec /bin/bash "$@"
```

16.10.4.2.3 Help

The commands `--helpfile` or `--hf` will show you the runscript helpfile, if it exists. With `--json` you can also see it as such:

```
singularity inspect -hf -j dino.img

{

    "data": {

        "attributes": {

            "help": "\n\n\nHi there! This is my image help section.
↪\n\nUsage:\n\nboobeep doo doo\n\n --arg/a arrrrg I'm a pirate!\n --boo/b eeeeeuzzz␣
↪where is the honey?\n\n\n"

        },

        "type": "container"

    }

}
```

or in a human friendly, readable print to the screen, don't use `-j` or `--json`:

```
$ singularity inspect -hf dino.img



Hi there! This is my image help section.


Usage:

```

(continues on next page)

(continued from previous page)

```
boobeep doo doo


 --arg/a arrrrg I'm a pirate!

 --boo/b eeeeeuzzz where is the honey?
```

16.10.4.2.4 Environment

The commands `--environment` and `-e` will show you the container's environment, again specified by the `%environment` section of a bootstrap file, and other ENV labels that might have come from a Docker import. You can again choose to see `--json` :

```
$ singularity inspect -e --json ubuntu.img

{

    "data": {

        "attributes": {

            "environment": "# Custom environment shell code should follow\n\n"

        },

        "type": "container"

    }

}
```

or human friendly:

```
$ singularity inspect -e ubuntu.img


##environment

# Custom environment shell code should follow
```

The container in the example above did not have any custom environment

variables set.

16.10.4.2.5 Test

The equivalent `--test` or `-t` commands will print any test defined for the container, which comes from the `%test` section of the bootstrap specification Singularity file. Again, we can ask for `--json` or human friendly (default):

```
$ singularity --inspect -t --json ubuntu.img

{
```

(continues on next page)

(continued from previous page)

```
    "data": {

        "attributes": {

            "test": null

        },

        "type": "container"

    }

}


$ singularity inspect -t  ubuntu.img

{

    "status": 404,

    "detail": "This container does not have any tests defined",

    "title": "Tests Undefined"

}
```

16.10.4.2.6 Deffile

Want to know where your container came from? You can see the entire Singularity definition file, if the container was created with a bootstrap, by using `--deffile` or `-d`:

```
$ singularity inspect -d  ubuntu.img


##deffile

Bootstrap:docker

From:ubuntu:latest
```

or with `--json` output.

```
$ singularity inspect -d --json ubuntu.img

{

    "data": {

        "attributes": {

            "deffile": "Bootstrap:docker\nFrom:ubuntu:latest\n"

        },
```

(continues on next page)

(continued from previous page)

```
        "type": "container"

    }

}
```

The goal of these commands is to bring more transparency to containers, and to help better integrate them into common workflows by having them expose their guts to the world! If you have feedback for how we can improve or amend this, please let us know!

16.10.5 pull

Singularity `pull` is the command that you would want to use to communicate with a container registry. The command does exactly as it says - there exists an image external to my host, and I want to pull it here. We currently support pull for both Docker and Singularity Hub images, and will review usage for both.

16.10.5.1 Singularity Hub

Singularity differs from Docker in that we serve entire images, as opposed to layers. This means that pulling a Singularity Hub means downloading the entire (compressed) container file, and then having it extract on your local machine. The basic command is the following:

```
singularity pull shub://vsoch/hello-world

Progress |===================================| 100.0%

Done. Container is at: ./vsoch-hello-world-master.img
```

16.10.5.1.1 How do tags work?

On Singularity Hub, a `tag` coincide with a branch. So if you have a repo called `vsoch/hello-world` , by default the file called `Singularity` (your build recipe file) will be looked for in the base of the master branch. The command that we issued above would be equivalent to doing:

```
singularity pull shub://vsoch/hello-world:master
```

To enable other branches to build, they must be turned on in your collection. If you then put another Singularity file in a branch called development, you would pull it as follows:

```
singularity pull shub://vsoch/hello-world:development
```

The term `latest` in Singularity Hub will pull, across all of your branches, the most recent image. If `development` is more recent than `master`, it would be pulled, for example.

16.10.5.1.2 Image Names

As you can see, since we didn't specify anything special, the default naming convention is to use the username, reponame, and the branch (tag). You have three options for changing this:

```
PULL OPTIONS:

    -n/--name   Specify a custom container name (first priority)

    -C/--commit Name container based on GitHub commit (second priority)

    -H/--hash   Name container based on file hash (second priority)
```

16.10.5.1.3 Custom Name

```
singularity pull --name meatballs.img shub://vsoch/hello-world

Progress |===================================| 100.0%

Done. Container is at: ./meatballs.img
```

16.10.5.1.4 Name by commit

Each container build on Singularity Hub is associated with the GitHub commit of the repo that was used to build it. You can specify to name your container based on the commit with the `--commit` flag, if, for example, you want to match containers to their build files:

```
singularity pull --commit shub://vsoch/hello-world

Progress |===================================| 100.0%

Done. Container is at: ./4187993b8b44cbfa51c7e38e6b527918fcdf0470.img
```

16.10.5.1.5 Name by hash

If you prefer the hash of the file itself, you can do that too.

```
singularity pull --hash shub://vsoch/hello-world

Progress |===================================| 100.0%

Done. Container is at: ./4db5b0723cfd378e332fa4806dd79e31.img
```

16.10.5.1.6 Pull to different folder

For any of the above, if you want to specify a different folder for your image, you can define the variable `SINGULARITY_PULLFOLDER`. By default, we will first check if you have the `SINGULARITY_CACHEDIR` defined, and pull images there. If not, we look for `SINGULARITY_PULLFOLDER`. If neither of these are defined, the image is pulled to the present working directory, as we showed above. Here is an example of pulling to `/tmp` .

```
SINGULARITY_PULLFOLDER=/tmp

singularity pull shub://vsoch/hello-world
```

(continues on next page)

(continued from previous page)

```
Progress |===================================| 100.0%

Done. Container is at: /tmp/vsoch-hello-world-master.img
```

16.10.5.1.7 Pull by commit

You can also pull different versions of your container by using their commit id (`version`).

```
singularity pull shub://vsoch/hello-world@42e1f04ed80217895f8c960bdde6bef4d34fab59

Progress |===================================| 100.0%

Done. Container is at: ./vsoch-hello-world-master.img
```

In this example, the first build of this container will be pulled.

16.10.5.2 Docker

Docker pull is similar (on the surface) to a Singularity Hub pull, and we would do the following:

```
singularity pull docker://ubuntu

Initializing Singularity image subsystem

Opening image file: ubuntu.img

Creating 223MiB image

Binding image to loop

Creating file system within image

Image is done: ubuntu.img

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /home/vanessa/.singularity/docker

Importing: base Singularity environment

Importing: /home/vanessa/.singularity/docker/
↪sha256:b6f892c0043b37bd1834a4a1b7d68fe6421c6acbc7e7e63a4527e1d379f92c1b.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:55010f332b047687e081a9639fac04918552c144bc2da4edb3422ce8efcc1fb1.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:2955fb827c947b782af190a759805d229cfebc75978dba2d01b4a59e6a333845.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:3deef3fcbd3072b45771bd0d192d4e5ff2b7310b99ea92bce062e01097953505.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:cf9722e506aada1109f5c00a9ba542a81c9e109606c01c81f5991b1f93de7b66.tar.gz
```

(continues on next page)

(continued from previous page)

```
Importing: /home/vanessa/.singularity/metadata/
↪sha256:fe44851d529f465f9aa107b32351c8a0a722fc0619a2a7c22b058084fac068a4.tar.gz

Done. Container is at: ubuntu.img
```

If you specify the tag, the image would be named accordingly (eg, `ubuntu-latest.img`). Did you notice that the output looks similar to if we did the following?

```
singularity create ubuntu.img

singularity import ubuntu.img docker://ubuntu
```

this is because the same logic is happening on the back end. Thus, the pull command with a docker uri also supports arguments `--size` and `--name` . Here is how I would pull an ubuntu image, but make it bigger, and name it something else.

```
singularity pull --size 2000 --name jellybelly.img docker://ubuntu

Initializing Singularity image subsystem

Opening image file: jellybelly.img

Creating 2000MiB image

Binding image to loop

Creating file system within image

Image is done: jellybelly.img

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /home/vanessa/.singularity/docker

Importing: base Singularity environment

Importing: /home/vanessa/.singularity/docker/
↪sha256:b6f892c0043b37bd1834a4a1b7d68fe6421c6acbc7e7e63a4527e1d379f92c1b.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:55010f332b047687e081a9639fac04918552c144bc2da4edb3422ce8efcc1fb1.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:2955fb827c947b782af190a759805d229cfebc75978dba2d01b4a59e6a333845.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:3deef3fcbd3072b45771bd0d192d4e5ff2b7310b99ea92bce062e01097953505.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:cf9722e506aada1109f5c00a9ba542a81c9e109606c01c81f5991b1f93de7b66.tar.gz

Importing: /home/vanessa/.singularity/metadata/
↪sha256:fe44851d529f465f9aa107b32351c8a0a722fc0619a2a7c22b058084fac068a4.tar.gz

Done. Container is at: jellybelly.img
```

16.10.6 run

It's common to want your container to "do a thing." Singularity run allows you to define a custom action to be taken when a container is either run or executed directly by file name. Specifically, you might want it to execute a command, or run an executable that gives access to many different functions for the user.

16.10.6.1 Overview

First, how do we run a container? We can do that in one of two ways - the commands below are identical:

```
$ singularity run centos7.img

$ ./centos7.img
```

In both cases, we are executing the container's "runscript" (the executable /singularity at the root of the image) that is either an actual file (version 2.2 and earlier) or a link to one (2.3 and later). For example, looking at a 2.3 image, I can see the runscript via the path to the link:

```
$ singularity exec centos7.img cat /singularity

#!/bin/sh


exec /bin/bash "$@"
```

or to the actual file in the container's metadata folder, /.singularity.d

```
$ singularity exec centos7.img cat /.singularity.d/runscript

#!/bin/sh


exec /bin/bash "$@"
```

Notice how the runscript has bash followed by \$@ ? This is good practice to include in a runscript, as any arguments passed by the user will be given to the container.

16.10.6.2 Runtime Flags

If you are interested in containing an environment or filesystem locations, we highly recommend that you look at the singularity run help and our documentation on *flags* to better customize this command.

16.10.6.3 Examples

In this example the container has a very simple runscript defined.

```
$ singularity exec centos7.img cat /singularity

#!/bin/sh


echo motorbot

```

(continues on next page)

(continued from previous page)

```
$ singularity run centos7.img

motorbot
```

16.10.6.3.1 Defining the Runscript

When you first create a container, the runscript is defined using the following order of operations:

1. A user defined runscript in the `%runscript` section of a bootstrap takes preference over all
2. If the user has not defined a runscript and is importing a Docker container, the Docker `ENTRYPOINT` is used.
3. If a user has not defined a runscript and adds `IncludeCmd: yes` to the bootstrap file, the `CMD` is used over the `ENTRYPOINT`
4. If the user has not defined a runscript and the Docker container doesn't have an `ENTRYPOINT`, we look for `CMD`, even if the user hasn't asked for it.
5. If the user has not defined a runscript, and there is no `ENTRYPOINT` or `CMD` (or we aren't importing Docker at all) then we default to `/bin/bash`

Here is how you would define the runscript section when you *build* an image:

```
Bootstrap: docker

From: ubuntu:latest


%runscript

exec /usr/bin/python "$@"
```

and of course python should be installed as /usr/bin/python. The addition of `$@` ensures that arguments are passed along from the user. If you want your container to run absolutely any command given to it, and you want to use run instead of exec, you could also just do:

```
Bootstrap: docker

From: ubuntu:latest


%runscript

exec "$@"`
```

If you want different entrypoints for your image, we recommend using the %apprun syntax (see *apps*). Here we have two entrypoints for foo and bar:

```
%runscript

exec echo "Try running with --app dog/cat"


%apprun dog

exec echo Hello "$@", this is Dog
```

(continues on next page)

(continued from previous page)

```
%apprun cat

exec echo Meow "$@", this is Cat
```

and then running (after build of a complete recipe) would look like:

```
sudo singularity build catdog.simg Singularity


$ singularity run catdog.simg

Try running with --app dog/cat


$ singularity run --app cat catdog.simg

Meow , this is Cat

$ singularity run --app dog catdog.simg

Hello , this is Dog
```

Generally, it is advised to provide help for your container with `%help` or `%apphelp`. If you find it easier, you can also provide help by way of a runscript that tells your user how to use the container, and gives access to the important executables. Regardless of your strategy. a reproducible container is one that tells the user how to interact with it.

16.10.7 shell

The `shell` Singularity sub-command will automatically spawn an interactive shell within a container. As of v2.3 the default that is spawned via the shell command is `/bin/bash` if it exists otherwise `/bin/sh` is called.

```
$ singularity shell

USAGE: singularity (options) shell [container image] (options)
```

Here we can see the default shell in action:

```
$ singularity shell centos7.img

Singularity: Invoking an interactive shell within container...


Singularity centos7.img:~> echo $SHELL

/bin/bash
```

Additionally any arguments passed to the Singularity command (after the container name) will be passed to the called shell within the container, and shell can be used across image types. Here is a quick example of shelling into a container assembled from Docker layers. We highly recommend that you look at the `singularity shell help` and our documentation on *flags* to better customize this command.

16.10.7.1 Change your shell

The `shell` sub-command allows you to set or change the default shell using the `--shell` argument. As of Singularity version 2.2, you can also use the environment variable `SINGULARITY_SHELL` which will use that as your shell entry point into the container.

16.10.7.1.1 Bash

The correct way to do it:

```
export SINGULARITY_SHELL="/bin/bash --norc"

singularity shell centos7.img Singularity: Invoking an interactive shell within␣
↪container...

Singularity centos7.img:~/Desktop> echo $SHELL

/bin/bash --norc
```

Don't do this, it can be confusing:

```
$ export SINGULARITY_SHELL=/bin/bash

$ singularity shell centos7.img

Singularity: Invoking an interactive shell within container...


# What? We are still on my Desktop? Actually no, but the uri says we are!

vanessa@vanessa-ThinkPad-T460s:~/Desktop$ echo $SHELL

/bin/bash
```

Depending on your shell, you might also want the `--noprofile` flag. How can you learn more about a shell? Ask it for help, of course!

16.10.7.2 Shell Help

```
$ singularity shell centos7.img --help

Singularity: Invoking an interactive shell within container...


GNU bash, version 4.2.46(1)-release-(x86_64-redhat-linux-gnu)

Usage:  /bin/bash [GNU long option] [option] ...

    /bin/bash [GNU long option] [option] script-file ...

GNU long options:

    --debug
```

(continues on next page)

(continued from previous page)

```
    --debugger

    --dump-po-strings

    --dump-strings

    --help

    --init-file

    --login

    --noediting

    --noprofile

    --norc

    --posix

    --protected

    --rcfile

    --rpm-requires

    --restricted

    --verbose

    --version

Shell options:

    -irsD or -c command or -O shopt_option      (invocation only)

    -abefhkmnptuvxBCHP or -o option

Type `/bin/bash -c "help set"' for more information about shell options.

Type `/bin/bash -c help' for more information about shell builtin commands.
```

And thus we should be able to do:

```
$ singularity shell centos7.img -c "echo hello world"

Singularity: Invoking an interactive shell within container...


hello world
```

16.11 Image Command Group

16.11.1 image.export

Export is a way to dump the contents of your container into a `.tar.gz`, or a stream to put into some other place. For example, you could stream this into an in memory tar in python. Importantly, this command was originally intended for Singularity version less than 2.4 in the case of exporting an ext3 filesystem. For Singularity greater than 2.4, the resulting export file is likely to be larger than the original squashfs counterpart. An example with an ext3 image is provided.

Here we export an image into a `.tar` file:

```
singularity image.export container.img > container.tar
```

We can also specify the file with `--file`

```
singularity image.export --file container.tar container.img
```

And here is the recommended way to compress your image:

```
singularity image.export container.img | gzip -9 > container.img.tar.gz
```

16.11.2 image.expand

While the squashfs filesystem means that you typically don't need to worry about the size of your container being built, you might find that if you are building an ext3 image (pre Singularity 2.4) you want to expand it.

16.11.2.1 Increasing the size of an existing image

You can increase the size of an image after it has been instantiated by using the image.expand Singularity sub-command. In the example below, we:

1. create an empty image
2. inspect it's size
3. expand it
4. confirm it's larger

```
$ singularity image.create container.img

Creating empty 768MiB image file: container.imglarity image.create container.im

Formatting image with ext3 file system

Image is done: container.img


$ ls -lh container.img

-rw-rw-r-- 1 vanessa vanessa 768M Oct  2 18:48 container.img


$ singularity image.expand container.img
```

(continues on next page)

(continued from previous page)

```
Expanding image by 768MB

Checking image's file system

e2fsck 1.42.13 (17-May-2015)

Pass 1: Checking inodes, blocks, and sizes

Pass 2: Checking directory structure

Pass 3: Checking directory connectivity

Pass 4: Checking reference counts

Pass 5: Checking group summary information

container.img: 11/49152 files (0.0% non-contiguous), 7387/196608 blocks

Resizing image's file system

resize2fs 1.42.13 (17-May-2015)

Resizing the filesystem on container.img to 393216 (4k) blocks.

The filesystem on container.img is now 393216 (4k) blocks long.

Image is done: container.img


$ ls -lh container.img

-rw-rw-r-- 1 vanessa vanessa 1.5G Oct  2 18:48 container.img
```

Similar to the create sub-command, you can override the default size increase (which is 768MiB) by using the `--size` option.

16.11.3 image.import

Singularity import is essentially taking a dump of files and folders and adding them to your image. This works for local compressed things (e.g., tar.gz) but also for docker image layers that you don't have on your system. As of version 2.3, import of docker layers includes the environment and metadata without needing sudo. It's generally very intuitive.

As an example, here is a common use case: wanting to import a Docker image:

```
singularity image.import container.img docker://ubuntu:latest
```

16.11.4 image.create

A Singularity image, which can be referred to as a "container," is a single file that contains a virtual file system. As of Singularity 2.4, we strongly recommend that you build (create and install) an image using *build*. If you have reason to create an empty image, or use create for any other reason, the original `create` command is replaced with a

more specific `image.create`. After creating an image you can install an operating system, applications, and save meta-data with it.

Whereas Docker assembles images from layers that are stored on your computer (viewed with the `docker history` command), a Singularity image is just one file that can sit on your Desktop, in a folder on your cluster, or anywhere. Having Singularity containers housed within a single image file greatly simplifies management tasks such as sharing, copying, and branching your containers. It also means that standard Linux file system concepts like permissions, ownership, and ACLs apply to the container (e.g. I can give read only access to a colleague, or block access completely with a simple `chmod` command).

16.11.4.1 Creating a new blank Singularity container image

Singularity will create a default container image of 768MiB using the following command:

```
singularity image.create container.img

Creating empty 768MiB image file: container.img

Formatting image with ext3 file system

Image is done: container.img
```

How big is it?

```
$ du -sh container.img

29M     container.img
```

Create will make an `ext3` filesystem. Let's create and import a docker base (the pre-2.4 way with two commands), and then compare to just building (one command) from the same base.

```
singularity create container.img

sudo singularity bootstrap container.img docker://ubuntu


...


$ du -sh container.img

769M
```

Prior to 2.4, you would need to provide a `--size` to change from the default:

```
$ singularity create --size 2048 container2.img

Initializing Singularity image subsystem

Opening image file: container2.img

Creating 2048MiB image

Binding image to loop

Creating file system within image
```

(continues on next page)

(continued from previous page)

```
Image is done: container2.img

$ ls -lh container*.img
-rwxr-xr-x 1 user group 2.1G Apr 15 11:34 container2.img
-rwxr-xr-x 1 user group 769M Apr 15 11:11 container.img
```

Now let's compare to if we just built, without needing to specify a size.

```
sudo singularity build container.simg docker://ubuntu

...

du -sh container.simg
45M container.simg
```

Quite a difference! And one command instead of one.

16.11.4.1.1 Overwriting an image with a new one

For any commands that If you have already created an image and wish to overwrite it, you can do so with the `--force` option.

```
$ singularity image.create container.img
ERROR: Image file exists, not overwriting.


$ singularity image.create --force container.img
Creating empty 768MiB image file: container.img
Formatting image with ext3 file system
Image is done: container.img
```

`@GodLoveD` has provided a nice interactive demonstration of creating an image (pre 2.4).

16.12 Instance Command Group

16.12.1 instance.start

New in Singularity version 2.4 you can use the `instance` command group to run instances of containers in the background. This is useful for running services like databases and web servers. The `instance.start` command lets you initiate a named instance in the background.

16.12.1.1 Overview

To initiate a named instance of a container, you must call the `instance.start` command with 2 arguments: the name of the container that you want to start and a unique name for an instance of that container. Once the new instance is running, you can join the container's namespace using a URI style syntax like so:

```
$ singularity shell instance://<instance_name>
```

You can specify options such as bind mounts, overlays, or custom namespaces when you initiate a new instance of a container with instance.start. These options will persist as long as the container runs.

For a complete list of options see the output of:

```
singularity help instance.start
```

16.12.1.2 Examples

These examples use a container from Singularity Hub, but you can use local containers or containers from Docker Hub as well. For a more detailed look at `instance` usage see *Running Instances*.

16.12.1.2.1 Start an instance called cow1 from a container on Singularity Hub

```
$ singularity instance.start shub://GodloveD/lolcow cow1
```

16.12.1.2.2 Start an interactive shell within the instance that you just started

```
$ singularity shell instance://cow1

Singularity GodloveD-lolcow-master.img:~> ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD

ubuntu       1     0  0 20:03 ?        00:00:00 singularity-instance: ubuntu [cow1]

ubuntu       3     0  0 20:04 pts/0    00:00:00 /bin/bash --norc

ubuntu       4     3  0 20:04 pts/0    00:00:00 ps -ef

Singularity GodloveD-lolcow-master.img:~> exit
```

16.12.1.2.3 Execute the runscript within the instance

```
$ singularity run instance://cow1

 _________________________________________

/ Clothes make the man. Naked people have \

| little or no influence on society.      |
```

(continues on next page)

(continued from previous page)

```
|                                          |
\ -- Mark Twain                            /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

16.12.1.2.4 Run a command within a running instance

```
$ singularity exec instance://cow1 cowsay "I like blending into the background"
 _____________________________________
< I like blending into the background >
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

16.12.2 instance.list

New in Singularity version 2.4 you can use the `instance` command group to run instances of containers in the background. This is useful for running services like databases and web servers. The `instance.list` command lets you keep track of the named instances running in the background.

16.12.2.1 Overview

After initiating one or more named instances to run in the background with the `instance.start` command you can list them with the `instance.list` command.

16.12.2.2 Examples

These examples use a container from Singularity Hub, but you can use local containers or containers from Docker Hub as well. For a more detailed look at `instance` usage see *Running Instances*.

16.12.2.2.1 Start a few named instances from containers on Singularity Hub

```
$ singularity instance.start shub://GodloveD/lolcow cow1

$ singularity instance.start shub://GodloveD/lolcow cow2

$ singularity instance.start shub://vsoch/hello-world hiya
```

16.12.2.2.2 List running instances

```
$ singularity instance.list

DAEMON NAME      PID      CONTAINER IMAGE

cow1             20522    /home/ubuntu/GodloveD-lolcow-master.img

cow2             20558    /home/ubuntu/GodloveD-lolcow-master.img

hiya             20595    /home/ubuntu/vsoch-hello-world-master.img
```

16.12.3 instance.stop

New in Singularity version 2.4 you can use the `instance` command group to run instances of containers in the background. This is useful for running services like databases and web servers. The `instance.stop` command lets you stop instances once you are finished using them

16.12.3.1 Overview

After initiating one or more named instances to run in the background with the `instance.start` command you can stop them with the `instance.stop` command.

16.12.3.2 Examples

These examples use a container from Singularity Hub, but you can use local containers or containers from Docker Hub as well. For a more detailed look at `instance` usage see *Running Instances*.

16.12.3.2.1 Start a few named instances from containers on Singularity Hub

```
$ singularity instance.start shub://GodloveD/lolcow cow1

$ singularity instance.start shub://GodloveD/lolcow cow2

$ singularity instance.start shub://vsoch/hello-world hiya
```

16.12.3.2.2 Stop a single instance

```
$ singularity instance.stop cow1

Stopping cow1 instance of /home/ubuntu/GodloveD-lolcow-master.img (PID=20522)
```

16.12.3.2.3 Stop all running instances

```
$ singularity instance.stop --all

Stopping cow2 instance of /home/ubuntu/GodloveD-lolcow-master.img (PID=20558)

Stopping hiya instance of /home/ubuntu/vsoch-hello-world-master.img (PID=20595)
```

16.13 Deprecated

16.13.1 bootstrap

Bootstrapping was the original way (for Singularity versions prior to 2.4) to install an operating system and then configure it appropriately for a specified need. Bootstrap is very similar to build, except that it by default uses an ext3 filesystem and allows for writability. The images unfortunately are not immutable in this way, and can degrade over time. As of 2.4, bootstrap is still supported for Singularity, however we encourage you to use *build* instead.

16.13.1.1 Quick Start

A bootstrap is done based on a Singularity recipe file (a text file called Singularity) that describes how to specifically build the container. Here we will overview the sections, best practices, and a quick example.

```
$ singularity bootstrap

USAGE: singularity [...] bootstrap <container path> <definition file>
```

The `<container path>` is the path to the Singularity image file, and the `<definition file>` is the location of the definition file (the recipe) we will use to create this container. The process of building a container should always be done by root so that the correct file ownership and permissions are maintained. Also, so installation programs check to ensure they are the root user before proceeding. The bootstrap process may take anywhere from one minute to one hour depending on what needs to be done and how fast your network connection is.

Let's continue with our quick start example. Here is your spec file, `Singularity` ,

```
Bootstrap:docker

From:ubuntu:latest
```

You next create an image:

```
$ singularity image.create ubuntu.img

Initializing Singularity image subsystem

Opening image file: ubuntu.img
```

(continues on next page)

(continued from previous page)

```
Creating 768MiB image

Binding image to loop

Creating file system within image

Image is done: ubuntu.img
```

and finally run the bootstrap command, pointing to your image (`<container path>`) and the file Singularity (`<definition file>`).

```
$ sudo singularity bootstrap ubuntu.img Singularity

Sanitizing environment

Building from bootstrap definition recipe

Adding base Singularity environment to container

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /root/.singularity/docker

[5/5] |===================================| 100.0%

Exploding layer: sha256:b6f892c0043b37bd1834a4a1b7d68fe6421c6acbc7e7e63a4527e1d379f92c1b.tar.gz

Exploding layer: sha256:55010f332b047687e081a9639fac04918552c144bc2da4edb3422ce8efcc1fb1.tar.gz

Exploding layer: sha256:2955fb827c947b782af190a759805d229cfebc75978dba2d01b4a59e6a333845.tar.gz

Exploding layer: sha256:3deef3fcbd3072b45771bd0d192d4e5ff2b7310b99ea92bce062e01097953505.tar.gz

Exploding layer: sha256:cf9722e506aada1109f5c00a9ba542a81c9e109606c01c81f5991b1f93de7b66.tar.gz

Exploding layer: sha256:fe44851d529f465f9aa107b32351c8a0a722fc0619a2a7c22b058084fac068a4.tar.gz

Finalizing Singularity container
```

Notice that bootstrap does require sudo. If you do an import, with a docker uri for example, you would see a similar flow, but the calling user would be you, and the cache your `$HOME`.

```
$ singularity image.create ubuntu.img

singularity import ubuntu.img docker://ubuntu:latest

Docker image path: index.docker.io/library/ubuntu:latest

Cache folder set to /home/vanessa/.singularity/docker
```

(continues on next page)

(continued from previous page)

```
Importing: base Singularity environment

Importing: /home/vanessa/.singularity/docker/
↪sha256:b6f892c0043b37bd1834a4a1b7d68fe6421c6acbc7e7e63a4527e1d379f92c1b.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:55010f332b047687e081a9639fac04918552c144bc2da4edb3422ce8efcc1fb1.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:2955fb827c947b782af190a759805d229cfebc75978dba2d01b4a59e6a333845.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:3deef3fcbd3072b45771bd0d192d4e5ff2b7310b99ea92bce062e01097953505.tar.gz

Importing: /home/vanessa/.singularity/docker/
↪sha256:cf9722e506aada1109f5c00a9ba542a81c9e109606c01c81f5991b1f93de7b66.tar.gz

Importing: /home/vanessa/.singularity/metadata/
↪sha256:fe44851d529f465f9aa107b32351c8a0a722fc0619a2a7c22b058084fac068a4.tar.gz
```

For details and best practices for creating your Singularity recipe, *read about them here*.

CHAPTER

SEVENTEEN

CONTRIBUTING

17.1 Support Singularity

Singularity is an open source project, meaning we have the challenge of limited resources. We are grateful for any support that you might offer to other users in the way of helping with issues, documentation, or code! If you haven't already, check out some of the ways to contribute to code and docs:

- contribute code
- contribute docs

17.1.1 Singularity Google Group

This is a huge endeavor, and it is greatly appreciated! If you have been using Singularity and having good luck with it, join our Google Group and help out other users! Post to online communities about Singularity, and request that your distribution vendor, service provider, and system administrators include Singularity for you!

17.1.2 Singularity on Slack

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

17.2 Contribute to the code

To contribute to the development of Singularity, you must:

- Own the code and/or have the right to contribute it
- Be able to submit software under the 3 clause BSD (or equivalent) license (while other licenses are allowed to be submitted by the license, acceptance of any contribution is up to the project lead)
- Read, understand and agree to the license
- Have a GitHub account (this just makes it easier on me)

We use the traditional GitHub Flow to develop. This means that you fork the repo and checkout a branch to make changes, you submit a pull request (PR) to the development branch with your changes, and the development branch gets merged with master for official releases. We also have an official CONTRIBUTING document, which also includes a code of conduct .

17.2.1 Step 1. Fork the repo

To contribute to the web based documentation, you should obtain a GitHub account and fork the Singularity repository. Once forked, you will want to clone the fork of the repo to your computer. Let's say my GitHub username is vsoch, and I am using ssh:

```
git clone git@github.com:vsoch/singularity.git

cd singularity/
```

17.2.2 Step 2. Set up your config

The GitHub config file, located at .git/config, is the best way to keep track of many different forks of a repository. I usually open it up right after cloning my fork to add the repository that I forked as a remote, so I can easily get updated from it. Let's say my `.git/config` first looks like this, after I clone my own branch:

```
[core]

    repositoryformatversion = 0

    filemode = true

    bare = false

    logallrefupdates = true

[remote "origin"]

    url = git@github.com:vsoch/singularity

    fetch = +refs/heads/*:refs/remotes/origin/*

[branch "master"]

    remote = origin

    merge = refs/heads/master
```

I would want to add the upstream repository, which is where I forked from.

```
[core]

    repositoryformatversion = 0

    filemode = true

    bare = false

    logallrefupdates = true

[remote "origin"]

    url = git@github.com:vsoch/singularity

    fetch = +refs/heads/*:refs/remotes/origin/*
```

(continues on next page)

(continued from previous page)

```
[remote "upstream"]

    url = https://github.com/sylabs/singularity

    fetch = +refs/heads/*:refs/remotes/origin/*

[branch "master"]

    remote = origin

    merge = refs/heads/master
```

I can also add some of my colleagues, if I want to pull from their branches:

```
[core]

    repositoryformatversion = 0

    filemode = true

    bare = false

    logallrefupdates = true

[remote "origin"]

    url = git@github.com:vsoch/singularity

    fetch = +refs/heads/*:refs/remotes/origin/*

[remote "upstream"]

    url = https://github.com/sylabs/singularity

    fetch = +refs/heads/*:refs/remotes/origin/*

[remote "greg"]

    url = https://github.com/gmkurtzer/singularity

    fetch = +refs/heads/*:refs/remotes/origin/*

[branch "master"]

    remote = origin

    merge = refs/heads/master
```

In the GitHub flow, the master branch is the frozen, current version of the software. Your master branch is always in sync with the upstream (our sylabs master), and the sylabs master is always the latest release of Singularity.

This would mean that I can update my master branch as follows:

```
git checkout master

git pull upstream master
```

(continues on next page)

(continued from previous page)

```
git push origin master
```

and then I would return to working on the branch for my feature. How to do that exactly? Read on!

17.2.3 Step 3. Checkout a new branch

Branches are a way of isolating your features. For example, if I am working on several features, I would want to keep them separate, and "submit them" (in what is called a pull request) to be added to the main repository codebase. Each repository, including your fork, has a main branch, which is usually called "master". As mentioned earlier, the master branch of a fork should always be in sync with the repository it is forked from (which I usually refer to as "upstream") and then branches of the fork consistently updated with that master. Given that we've just cloned the repo, we probably want to work off of the current development branch, which has the most up to date "next version" of the software. So we can start by checking out that branch:

```
git checkout -b development

git pull origin development
```

At this point, you can either choose to work on this branch, push to your origin development and pull request to sylabs development, or you can checkout another branch specific to your feature. We recommend always working from, and staying in sync with development. The command below would checkout a branch called `add/my-awesome-new-feature` from development.

```
# Checkout a new branch called add/my-awesome-feature

git checkout -b add/my-awesome-feature development
```

The addition of the `-b` argument tells git that we want to make a new branch. If I want to just change branches (for example back to master) I can do the same command without `-b`:

```
# Change back to master

git checkout master
```

Note that you should commit changes to the branch you are working on before changing branches, otherwise they would be lost. GitHub will give you a warning and prevent you from changing branches if this is the case, so don't worry too much about it.

17.2.4 Step 4. Make your changes

On your new branch, go nuts! Make changes, test them, and when you are happy with a bit of progress, commit the changes to the branch:

```
git commit -a
```

This will open up a little window in your default text editor that you can write a message in the first line. This commit message is important - it should describe exactly the changes that you have made. Bad commit messages are like:

- changed code
- updated files

Good commit messages are like:

- changed function "get_config" in functions.py to output csv to fix #2

- updated docs about shell to close #10

The tags "close #10" and "fix #2" are referencing issues that are posted on the main repo you are going to do a pull request to. Given that your fix is merged into the master branch, these messages will automatically close the issues, and further, it will link your commits directly to the issues they intended to fix. This is very important down the line if someone wants to understand your contribution, or (hopefully not) revert the code back to a previous version.

17.2.5 Step 5. Push your branch to your fork

When you are done with your commits, you should push your branch to your fork (and you can also continuously push commits here as you work):

```
git push origin add/my-awesome-feature
```

Note that you should always check the status of your branches to see what has been pushed (or not):

```
git status
```

17.2.6 Step 6. Submit a Pull Request

Once you have pushed your branch, then you can go to either fork and (in the GUI) submit a Pull Request. Regardless of the name of your branch, your PR should be submit to the sylabs development branch. This will open up a nice conversation interface / forum for the developers of Singularity to discuss your contribution, likely after testing. At this time, any continuous integration that is linked with the code base will also be run. If there is an issue, you can continue to push commits to your branch and it will update the Pull Request.

17.2.7 Support, helping and spreading the word!

This is a huge endeavor, and it is greatly appreciated! If you have been using Singularity and having good luck with it, join our Google Group and help out other users! Post to online communities about Singularity, and request that your distribution vendor, service provider, and system administrators include Singularity for you!

17.3 Contributing to Documentation

We (like almost all open source software providers) have a documentation dilemma… We tend to focus on the code features and functionality before working on documentation. And there is very good reason for this, we want to share the love so nobody feels left out!

You can contribute to the documentation, by sending a pull request on our repository for documentation.

The current documentation is generated with:

- reStructured Text (RST) and ReadTheDocs

Other dependencies include:

- Python 2.7
- Sphinx

More information about contributing to the documentation and the instructions on how to install the dependencies and how to generate the files can be obtained here.

CHAPTER

EIGHTEEN

FAQ

18.1 General Singularity Info

18.1.1 Why the name "Singularity"?

A "Singularity" is an astrophysics phenomenon in which a single point becomes infinitely dense. This type of a singularity can thus contain massive quantities of universe within it and thus encapsulating an infinite amount of data within it.

Additionally, the name "Singularity" for me (Greg) also stems back from my past experience working at a company called Linuxcare where the Linux Bootable Business Card (LNX-BBC) was developed. The BBC, was a Linux rescue disk which paved the way for all live CD bootable distributions using a compressed single image file system called the "singularity".

The name has **NOTHING** to do with Kurzweil's (among others) prediction that artificial intelligence will abruptly have the ability to reprogram itself, surpass that of human intelligence and take control of the planet. If you are interested in this may I suggest the movie Terminator 2: Judgment Day.

18.1.2 What is so special about Singularity?

While Singularity is a container solution (like many others), Singularity differs in it's primary design goals and architecture:

1. **Reproducible software stacks:** These must be easily verifiable via checksum or cryptographic signature in such a manner that does not change formats (e.g. splatting a tarball out to disk). By default Singularity uses a container image file which can be checksummed, signed, and thus easily verified and/or validated.
2. **Mobility of compute:** Singularity must be able to transfer (and store) containers in a manner that works with standard data mobility tools (rsync, scp, gridftp, http, NFS, etc..) and maintain software and data controls compliancy (e.g. HIPPA, nuclear, export, classified, etc..)
3. **Compatibility with complicated architectures:** The runtime must be immediately compatible with existing HPC, scientific, compute farm and even enterprise architectures any of which maybe running legacy kernel versions (including RHEL6 vintage systems) which do not support advanced namespace features (e.g. the user namespace)
4. **Security model:** Unlike many other container systems designed to support trusted users running trusted containers we must support the opposite model of untrusted users running untrusted containers. This changes the security paradigm considerably and increases the breadth of use cases we can support.

18.1.3 Which namespaces are virtualized? Is that select-able?

That is up to you!

While some namespaces, like newns (mount) and fs (file system) must be virtualized, all of the others are conditional depending on what you want to do. For example, if you have a workflow that relies on communication between containers (e.g. MPI), it is best to not isolate any more than absolutely necessary to avoid performance regressions. While other tasks are better suited for isolation (e.g. web and data base services).

Namespaces are selected via command line usage and system administration configuration.

18.1.4 What Linux distributions are you trying to get on-board?

All of them! Help us out by letting them know you want Singularity to be included!

18.1.5 How do I request an installation on my resource?

It's important that your administrator have all of the resources available to him or her to make a decision to install Singularity. We've prepared a *helpful guide* that you can send to him or her to start a conversation. If there are any unanswered questions, we recommend that you reach out.

18.2 Basic Singularity usage

18.2.1 Do you need administrator privileges to use Singularity?

You generally do not need admin/sudo to use Singularity containers but you do however need admin/root access to install Singularity and for some container build functions (for example, building from a recipe, or a writable image).

This then defines the work-flow to some extent. If you have a container (whether Singularity or Docker) ready to go, you can run/shell/import without root access. If you want to build a new Singularity container image from scratch it must be built and configured on a host where you have root access (this can be a physical system or on a VM). And of course once the container image has been configured it can be used on a system where you do not have root access as long as Singularity has been installed there.

18.2.2 What if I don't want to install Singularity on my computer?

If you don't want to build your own images, Singularity Hub will connect to your GitHub repos with build specification files, and build the containers automatically for you. You can then interact with them easily where Singularity is installed (e.g., on your cluster):

```
singularity shell shub://vsoch/hello-world

singularity run shub://vsoch/hello-world

singularity pull shub://vsoch/hello-world

singularity build hello-world.simg shub://vsoch/hello-world # redundant, you would␣
↪already get an image
```

18.2.3 Can you edit/modify a Singularity container once it has been instantiated?

We strongly advocate for reproducibility, so if you build a squashfs container, it is immutable. However, if you build with `--sandbox` or `--writable` you can produce a writable sandbox folder or a writable ext3 image, respectively. From a sandbox you can develop, test, and make changes, and then build or convert it into a standard image.

We recommend to use the default compressed, immutable format for production containers.

18.2.4 Can multiple applications be packaged into one Singularity Container?

Yes! You can even create entire pipe lines and work flows using many applications, binaries, scripts, etc.. The `%runscript` bootstrap section is where you can define what happens when a Singularity container is run, and with the introduction of *modular apps* you can now even define `%apprun` sections for different entrypoints to your container.

18.2.5 How are external file systems and paths handled in a Singularity Container?

Because Singularity is based on container principals, when an application is run from within a Singularity container its default view of the file system is different from how it is on the host system. This is what allows the environment to be portable. This means that root ('/') inside the container is different from the host!

Singularity automatically tries to resolve directory mounts such that things will just work and be portable with whatever environment you are running on. This means that `/tmp` and `/var/tmp` are automatically shared into the container as is `/home`. Additionally, if you are in a current directory that is not a system directory, Singularity will also try to bind that to your container.

There is a caveat in that a directory must already exist within your container to serve as a mount point. If that directory does not exist, Singularity will not create it for you! You must do that. To create custom mounts at runtime, you should use the `-B` or `--bind` argument:

```
singularity run --bind /home/vanessa/Desktop:/data container.img
```

18.2.6 How does Singularity handle networking?

As of 2.4, Singularity can support the network namespace to a limited degree. At present, we just use it for isolation, but it will soon be more featurefull.

18.2.7 Can Singularity support daemon processes?

Singularity has container "instance" support which allows one to start a container process, within its own namespaces, and use that instance like it was a stand alone, isolated system.

At the moment (as above describes), the network (and UTS) namespace is not well supported, so if you spin up a process daemon, it will exist on your host's network. This means you can run a web server, or any other daemon, from within a container and access it directly from your host.

18.2.8 Can a Singularity container be multi-threaded?

Yes. Singularity imposes no limitations on forks, threads or processes in general.

18.2.9 Can a Singularity container be suspended or check-pointed?

Yes and maybe respectively. Any Singularity application can be suspended using standard Linux/Unix signals. Check-pointing requires some preloaded libraries to be automatically loaded with the application but because Singularity escapes the hosts library stack, the checkpoint libraries would not be loaded. If however you wanted to make a Singularity container that can be check-pointed, you would need to install the checkpoint libraries into the Singularity container via the specfile.

On our roadmap is the ability to checkpoint the entire container process thread, and restart it. Keep an eye out for that feature!

18.2.10 Are there any special requirements to use Singularity through an HPC job scheduler?

Singularity containers can be run via any job scheduler without any modifications to the scheduler configuration or architecture. This is because Singularity containers are designed to be run like any application on the system, so within your job script just call Singularity as you would any other application!

18.2.11 Does Singularity work in multi-tenant HPC cluster environments?

Yes! HPC was one of the primary use cases in mind when Singularity was created.

Most people that are currently integrating containers on HPC resources do it by creating virtual clusters within the physical host cluster. This precludes the virtual cluster from having access to the host cluster's high performance fabric, file systems and other investments which make an HPC system high performance.

Singularity on the other hand allows one to keep the high performance in High Performance Computing by containerizing applications and supporting a runtime which seamlessly interfaces with the host system and existing environments.

18.2.12 Can I run X11 apps through Singularity?

Yes. This works exactly as you would expect it to.

18.2.13 Can I containerize my MPI application with Singularity and run it properly on an HPC system?

Yes! HPC was one of the primary use cases in mind when Singularity was created.

While we know for a fact that Singularity can support multiple MPI implementations, we have spent a considerable effort working with Open MPI as well as adding a Singularity module into Open MPI (v2) such that running at extreme scale will be as efficient as possible.

note: We have seen no major performance impact from running a job in a Singularity container.

18.2.14 Why do we call 'mpirun' from outside the container (rather than inside)?

With Singularity, the MPI usage model is to call 'mpirun' from outside the container, and reference the container from your 'mpirun' command. Usage would look like this:

```
$ mpirun -np 20 singularity exec container.img /path/to/contained_mpi_prog
```

By calling 'mpirun' outside the container, we solve several very complicated work-flow aspects. For example, if 'mpirun' is called from within the container it must have a method for spawning processes on remote nodes. Historically ssh is used for this which means that there must be an sshd running within the container on the remote nodes, and this sshd process must not conflict with the sshd running on that host! It is also possible for the resource manager to launch the job and (in Open MPI's case) the Orted processes on the remote system, but that then requires resource manager modification and container awareness.

In the end, we do not gain anything by calling 'mpirun' from within the container except for increasing the complexity levels and possibly losing out on some added performance benefits (e.g. if a container wasn't built with the proper OFED as the host).

See the Singularity on HPC page for more details.

18.2.15 Does Singularity support containers that require GPUs?

Yes. Many users run GPU-dependent code within Singularity containers. The experimental `--nv` option allows you to leverage host GPUs without installing system level drivers into your container. See the *exec* command for an example.

18.3 Container portability

18.3.1 Are Singularity containers kernel-dependent?

No, never. But sometimes yes.

Singularity is using standard container principals and methods so if you are leveraging any kernel version specific or external patches/module functionality (e.g. OFED), then yes there maybe kernel dependencies you will need to consider.

Luckily most people that would hit this are people that are using Singularity to inter-operate with an HPC (High Performance Computing) system where there are highly tuned interconnects and file systems you wish to make efficient use of. In this case, See the documentation of MPI with Singularity.

There is also some level of glibc forward compatibility that must be taken into consideration for any container system. For example, I can take a Centos-5 container and run it on Centos-7, but I can not take a Centos-7 container and run it on Centos-5.

note: If you require kernel-dependent features, a container platform is probably not the right solution for you.

18.3.2 Can a Singularity container resolve GLIBC version mismatches?

Yes. Singularity containers contain their own library stack (including the Glibc version that they require to run).

18.3.3 What is the performance trade off when running an application native or through Singularity?

So far we have not identified any appreciable regressions of performance (even in parallel applications running across nodes with InfiniBand). There is a small start-up cost to create and tear-down the container, which has been measured to be anywhere from 10 - 20 thousandths of a second.

18.4 Misc

The following are miscellaneous questions.

18.4.1 Are there any special security concerns that Singularity introduces?

No and yes.

While Singularity containers always run as the user launching them, there are some aspects of the container execution which requires escalation of privileges. This escalation is achieved via a SUID portion of code. Once the container environment has been instantiated, all escalated privileges are dropped completely, before running any programs within the container.

Additionally, there are precautions within the container context to mitigate any escalation of privileges. This limits a user's ability to gain root control once inside the container.

You can read more about the Singularity *security overview here*.