**Singularity**

**Singularity**

Software is sometimes containerized for ease of installation and use on the clusters, typically when there are numerous dependencies needed for the software. A container is a self-sufficient package containing an entire Linux operating system with the required software and dependencies installed inside of it. On the campus clusters, the software used to make these containers is called Singularity.

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

**Why use containers?**

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

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

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

## Use Cases

**Reproducible Research**

Scientists and researchers can package their entire computing environment, including the software, libraries, and dependencies, into a Singularity container.

**Portability**

Singularity containers are designed to be highly portable. You can create a Singularity container on one system and run it on another without worrying about compatibility issues. This is especially important in HPC environments where different clusters may have varying software configurations.

**Isolation**

Singularity containers provide process and filesystem isolation. This means that processes running within the container are isolated from the host system, reducing the risk of conflicts or interference with other processes. It also allows multiple containers with different software stacks to run concurrently on the same system.

**Security**

Singularity containers are designed with security in mind. They do not require elevated privileges to run, which reduces the attack surface compared to some other containerization technologies. This can be important for shared HPC clusters.

**Legacy Software**

Singularity can be used to run older or legacy software that may not be compatible with the host system's libraries or operating systems.

**HPC and Supercomputing**

Singularity is commonly used in high-performance computing and supercomputing environments to package and distribute scientific applications and simulations. It allows researchers to utilize the full power of these systems while maintaining a high degree of flexibility and control.

**Cloud and Cluster Computing**

Singularity containers can also be used in cloud computing environments and on computing clusters. They enable users to create consistent environments for their applications across various cloud providers or cluster nodes.

**Continuous Integration/Continuous Deployment (CI/CD)**

Singularity containers can be integrated into CI/CD pipelines to ensure that software tests and builds are conducted in a consistent environment, improving the reliability of the software development process.

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

## Download pre-built images

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

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

$ singularity pull docker://godlovedc/lolcow

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

$ singularity build ubuntu.sif library://ubuntu

~ **The**[**exec**](https://www.sylabs.io/guides/3.5/user-guide/cli/singularity_exec.html) command allows you to execute a custom command within a container by specifying the image file. For instance, to execute the cowsay program within the lolcow\_latest.sif container:

$ singularity exec lolcow\_latest.sif cowsay moo

\_\_\_\_\_

< moo >

-----

\ ^\_\_^

\ (oo)\\_\_\_\_\_\_\_

(\_\_)\ )\/\

||----w |

|| ||

## Build images from scratch

Singularity v3.0 and above produces immutable images in the Singularity Image File (SIF) format.

* **the sandbox**

Singularity also supports **the sandbox** format (which is really just a directory). To build into a sandbox (container in a directory) use the build **--sandbox** command and option:

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

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

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

NOTE: The **build** command accepts a target as input and produces a container as output.

### **Singularity Definition Files**

For a reproducible, verifiable, and production-quality container, you should build a SIF file using a Singularity definition file.

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

Here is an example of a definition file:

**BootStrap**: library

**From**: ubuntu:16.04

**%post**

apt-get -y update

apt-get -y install fortune cowsay lolcat

**%environment**

export LC\_ALL=C

export PATH=/usr/games:$PATH

**%runscript**

fortune | cowsay | lolcat

**%labels**

Author GodloveD

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

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

This is a very small example of the things that you can do with a [definition file](https://docs.sylabs.io/guides/3.5/user-guide/definition_files.html#definitionfiles).

## Cache Folders

You may wish to customize your build environment by doing things such as specifying a custom cache directory for images or sending your Docker Credentials to the registry endpoint.

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

$HOME/.singularity/cache/library

$HOME/.singularity/cache/oci

$HOME/.singularity/cache/oci-tmp

ache location as /root/.singularity/cache

For example, running the following command with sudo privilege (considering the sudo privilege location for cache /root/.singularity/cache):

$ sudo singularity cache list

NAME DATE CREATED SIZE TYPE

ubuntu\_latest.sif 2019-01-31 14:59:32 28.11 Mb library

ubuntu\_18.04.sif 2019-01-31 14:58:44 27.98 Mb library

**Temporary Folders**

Singularity uses a temporary directory to build the squashfs filesystem, and this temp space needs to be large enough to hold the entire resulting Singularity image. By default this happens in /tmp but the location can be configured by setting SINGULARITY\_TMPDIR to the full path where you want the sandbox and squashfs temp files to be stored.

## Pull Folder

To customize your pull default location you can do so by specifying Singularity in which folder to pull the image, assuming you own a folder called mycontainers inside your $HOME folder , you would need to do something like the following:

$ singularity pull $HOME/mycontainers library://library/default/alpine

## Making use of public images from Docker Hub

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

$ singularity pull docker://godlovedc/lolcow

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

$ file lolcow\_latest.sif

lolcow\_latest.sif: a /usr/bin/env run-singularity script executable (binary data)

In converting to SIF, individual layers of the Docker image have been combined into a single, native file for use by Singularity

there is no need to subsequently build the image for Singularity. For example, you can now exec, run or shell into the SIF version via Singularity, [as described above](https://docs.sylabs.io/guides/3.5/user-guide/singularity_and_docker.html#sec-action-commands-prebuilt-public-docker-images).

inspect reveals metadata for the container encapsulated via SIF:

$ singularity inspect lolcow\_latest.sif

**Note**: we can also pull the private images from the docker hub by giving credential of the docker hub.

## Building images for Singularity from Docker Registries

The build command is used to **create** Singularity containers.

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

$ singularity build mylolcow\_latest.sif docker://godlovedc/lolcow

#### Building Containers Remotely

By making use of the [Sylabs Cloud Remote Builder](https://cloud.sylabs.io/builder), it is possible to build SIF containers remotely from images hosted at Docker Hub

The Sylabs Cloud Remote Builder is a service that can be used from the Singularity command line or via its Web interface.

## Singularity Definition file vs. Dockerfile

| **Singularity Definition file** | | **Dockerfile** | |
| --- | --- | --- | --- |
| **Section** | **Description** | **Section** | **Description** |
| Bootstrap | Defines from which  library to build  your container from.  You are free to choose  between library  (Our cloud library)  , docker , shub  and oras. | - | Can only bootstrap  from Docker Hub. |
| From: | To specify the provider  from which to build the  container. | FROM | Creates a layer from  the described docker image.  For example, if you got a  Dockerfile with the FROM  section set like:  FROM:ubuntu:18.04,  this means that a layer  will be created from the  ubuntu:18.04  **Docker** image.  (You cannot choose any  other bootstrap provider) |
| %setup | Commands that run  outside the  container (in the host  system) after the base  OS has been installed. | - | Not supported. |
| %files | To copy files from  your local  to the host. | COPY | To copy files from your  Docker’s client current  directory. |
| %environment | To declare and set  your environment  variables. | ENV | ENV will take the name  of the variable and the  value and set it. |
| %help | To provide a help  section to your  container image. | - | Not supported on the  Dockerfile. |
| %post | Commands that will  be run at  build-time. | RUN | Commands to build your  application image  with make |
| %runscript` | Commands that will  be run at  running your  container image. | CMD | Commands that run  within the Docker  container. |
| %startscript | Commands that will  be run when  an instance is started.  This is useful for  container images  using services. | - | Not supported. |
| %test | Commands that run  at the very end  of the build process  to validate the  container using  a method of your  choice. (to verify  distribution or  software versions  installed inside  the container) | HEALTHCHECK | Commands that verify  the health status of  the container. |
| %apps | Allows you to install  internal modules  based on the concept  of SCIF-apps. | - | Not supported. |
| %labels | Section to add and  define metadata  within your container. | LABEL | Section to declare  metadata as a  key-value pair. |

# Fakeroot feature

The fakeroot feature (commonly referred as rootless mode) allows an unprivileged user to run a container as a **“fake root”** user by leveraging [user namespace UID/GID mapping](http://man7.org/linux/man-pages/man7/user_namespaces.7.html).

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

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

### Filesystem

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

Additionally, all files or directories created by the **“fake root”** user are owned by root:root inside container but as user:group outside of the container.

| **UID inside container** | **UID outside container** |
| --- | --- |
| 0 (root) | 1000 (user) |

### Network

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

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

Usage

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

* shell
* exec
* run
* build

# Cloud Library

The Sylabs Cloud Library is the place to [push](https://docs.sylabs.io/guides/3.5/user-guide/cloud_library.html#push) your containers to the cloud so other users can [pull](https://docs.sylabs.io/guides/3.5/user-guide/cloud_library.html#pull), [verify](https://docs.sylabs.io/guides/3.5/user-guide/signNverify.html#signnverify), and use them.

The Sylabs Cloud also provides a [Remote Builder](https://docs.sylabs.io/guides/3.5/user-guide/cloud_library.html#remote-builder), allowing you to build containers on a secure remote service. This is convenient so that you can build containers on systems where you do not have root privileges.

## Creating a Access token

Access tokens for pushing a container, and remote builder.

To generate a access token, do the following steps:

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

Now that you have your token, you are ready to push your container!

## Pushing a Container

The singularity push command will push a container to the container library with the given URL. Here’s an example of a typical push command:

$ singularity push my-container.sif library://your-name/project-dir/my-container:latest

# Environment and Metadata

Singularity containers support environment variables and labels that you can add to your container during the build process.

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

/.singularity.d/

├── actions

│ ├── exec

│ ├── run

│ ├── shell

│ ├── start

│ └── test

├── env

│ ├── 01-base.sh

| ├── 10-docker2singularity.sh

│ ├── 90-environment.sh

│ ├── 91-environment.sh

| ├── 94-appsbase.sh

│ ├── 95-apps.sh

│ └── 99-base.sh

├── labels.json

├── libs

├── runscript

├── runscript.help

├── Singularity

└── startscript

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

# OCI Runtime Support

OCI is an acronym for the [Open Containers Initiative](https://www.opencontainers.org/) - an independent organization whose mandate is to develop open standards relating to containerization. To date, standardization efforts have focused on container formats and runtimes.

Owing to this restricted focus, a subset of the Singularity oci command group receives attention here; specifically:

* Mounting and unmounting OCI filesystem bundles
* Creating OCI compliant container instances

### Mounting an OCI Filesystem Bundle

[BusyBox](https://busybox.net/about.html) is used here for the purpose of illustration.

$ sudo singularity oci mount ./busybox\_latest.sif /var/tmp/busybox

## Creating OCI Compliant Container Instances

SIF files encapsulate the OCI runtime. By ‘OCI mounting’ a SIF file (see above), this encapsulated runtime is revealed

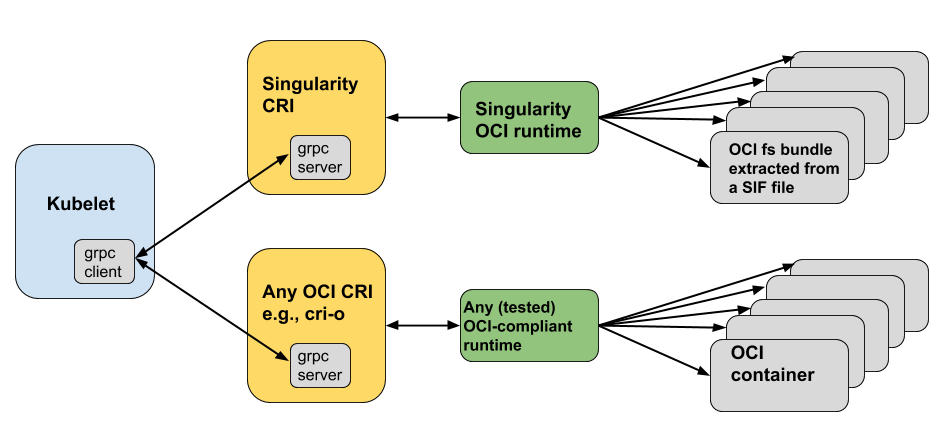
Once revealed, the filesystem bundle can be used to bootstrap the creation of an OCI compliant container instance as follows:

$ sudo singularity oci create -b /var/tmp/busybox busybox1

## Kubernetes Integration

the impetus is initially derived from the requirement to integrate with [Kubernetes](https://kubernetes.io/).

Simply stated, Kubernetes is an open-source system for orchestrating containers; developed originally at Google, Kubernetes was contributed as seed technology to the [Cloud Native Compute Foundation](https://www.cncf.io/) (CNCF).



From this schematic it is evident that integrating Singularity containers with Kubernetes requires the following efforts:

1. Implementation of a CRI for Singularity
2. Implementation of an OCI runtime in Singularity

# Plugins

Singularity plugin is a package that can be dynamically loaded by the Singularity runtime, augmenting Singularity with experimental, non-standard and/or vendor-specific functionality. Currently, plugins are able to add commands and flags to Singularity. In the future, plugins will also be able to interface with more complex subsystems of the Singularity runtime.

$ singularity plugin list

We can also write our own plugins.

# Limiting container resources with cgroups

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

Singularity cgroups support can be configured and utilized via a TOML file. An example file is typically installed at /usr/local/etc/singularity/cgroups/cgroups.toml (but may also be installed in other locations such as /etc/singularity/cgroups/cgroups.toml depending on your installation method). You can copy and edit this file to suit your needs. Then when you need to limit your container resources, apply the settings in the TOML file by using the path as an argument to the --apply-cgroups option like so:

$ sudo singularity shell --apply-cgroups /path/to/cgroups.toml my\_container.sif

The --apply-cgroups option can only be used with root privileges.

### Limiting memory

### Limiting CPU

### Limiting IO

#### Limiting device access

# GPU Support (NVIDIA CUDA & AMD ROCm)

Singularity natively supports running application containers that use NVIDIA’s CUDA GPU compute framework, or AMD’s ROCm solution. This allows easy access to users of GPU-enabled machine learning frameworks such as tensorflow, regardless of the host operating system. As long as the host has a driver and library installation for CUDA/ROCm then it’s possible to e.g. run tensorflow in an up-to-date Ubuntu 18.04 container, from an older RHEL 6 host.

Applications that support OpenCL for compute acceleration can also be used easily, with an additional bind option.

### **NVIDIA GPUs & CUDA Example - tensorflow-gpu**

## Library Search Options