Singularity

Giacomo Baruzzo, PhD

Department of Information Engineering

University of Padova

Acknowledgments: Marco Cappellato, Giulia Cesaro, Mikele Milia

Summary

- Introduction to Singularity
  - Characteristics
  - General workflow
- Working with Singularity
  - Run an existing image
  - Build an image
  - Write a definition file
- Working with Singularity in CAPRI
  - Slurm job

Singularity(CE)

SingularityCE (formerly Singularity) is a **free** and **open-source HPC** container engine for **Linux-base OS**

Originally developed at Lawrence Berkeley National Laboratory

Currently developer by Sylabs company

Commercial version from Sylabs: SingularityPRO

Official website: https://sylabs.io/singularity/

How to install: https://sylabs.io/guides/latest/user-guide/quick_start.html#quick-installation-steps

Version installed in CAPRI: SingularityCE v3.8.3

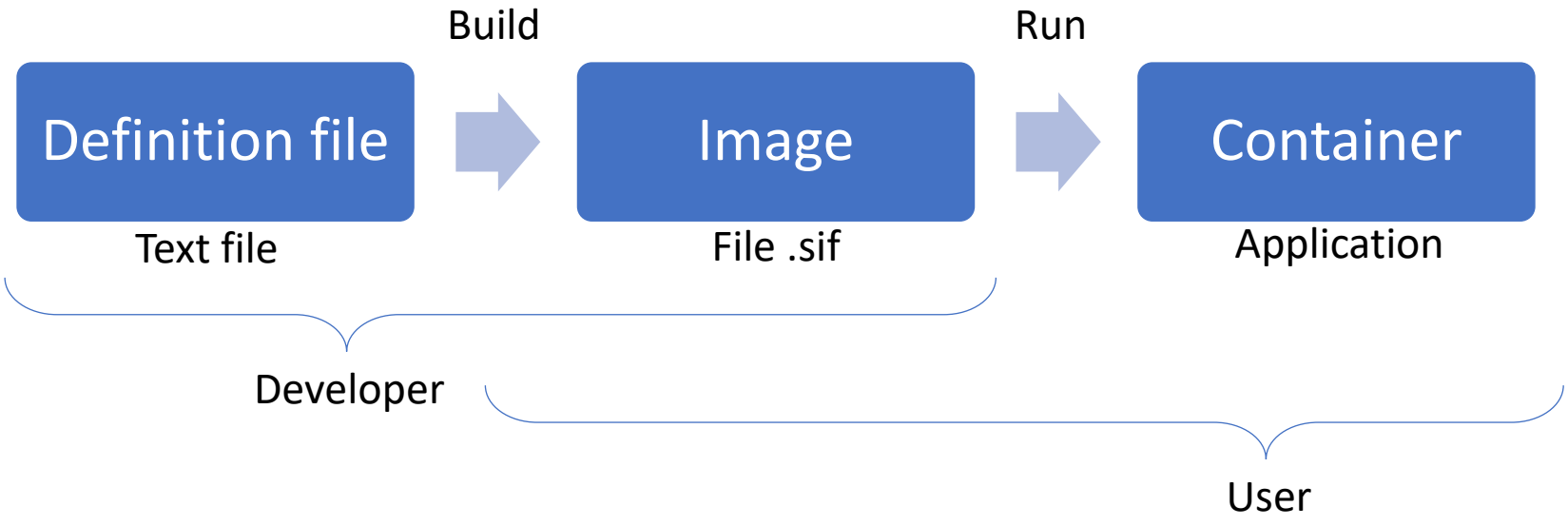
Singularity -> Apptainer

- Original Singularity project (i.e. the one before Sylabs fork) is officially moving into the Linux Foundation, becoming **Apptainer**
- **Sylabs's SingularityCE** and **Linux Foundation's Singularity (i.e. Apptainer)** will co-exist
- SingularityCE = Apptainer (for now)

For us: SingularityCE = Apptainer = Singularity

Singularity container workflow

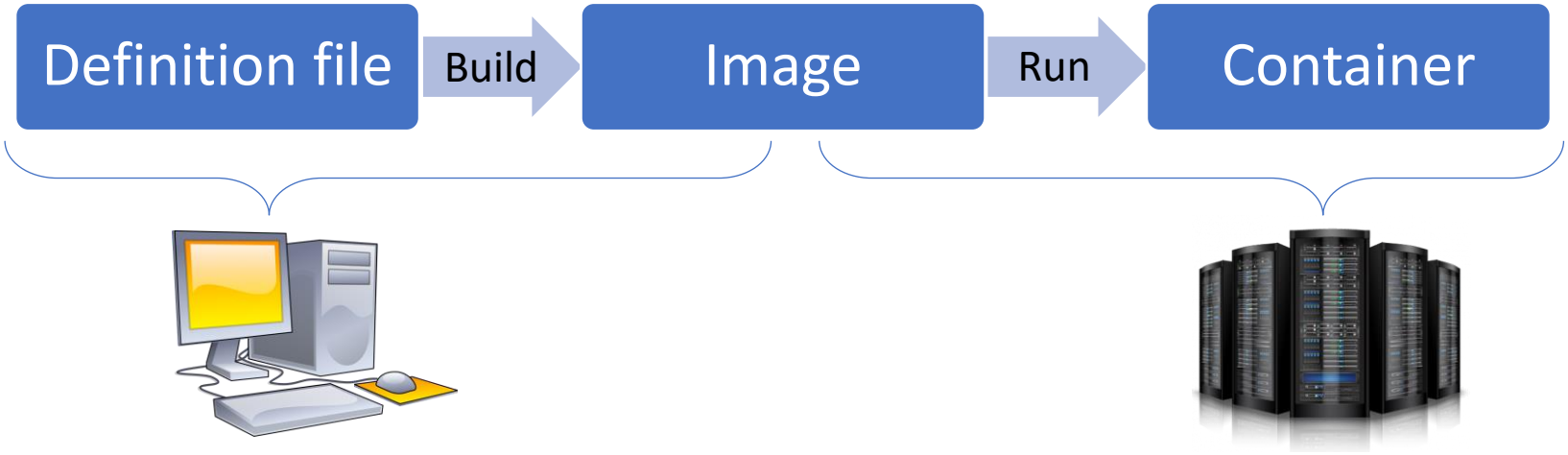
- ***Singularity definition file***: Text file containing Singularity commands about how to create the image
- ***Singularity image***: Immutable file containing a description of the computing environment (libraries, applications, etc.)
- ***Singularity container***: Runtime instantiation of a Singularity image



Using container in HPC

Workflow

1. Build container on your own computer
2. Move the image from your computer to the HPC infrastructure
3. Run container on the HPC infrastructure



Why build container in your computer?

- The build process may require privileges (e.g. sudo)
- Building is not a computation intensive process
- Test on your computer before running on the HPC infrastructure

Singularity image

Singularity image are contained in Singularity Image Format files (file .sif)

A SIF file describe an **immutable** container image: any command run within the container can not alter the container image (e.g. install/remove software in the container)

SIF files can be cryptographically signed to guarantee immutability and provide accountability as to who signed it

SIF files can be encrypted, making everything inside the container inaccessible without the correct key or passphrase

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Run a Singularity image - exec

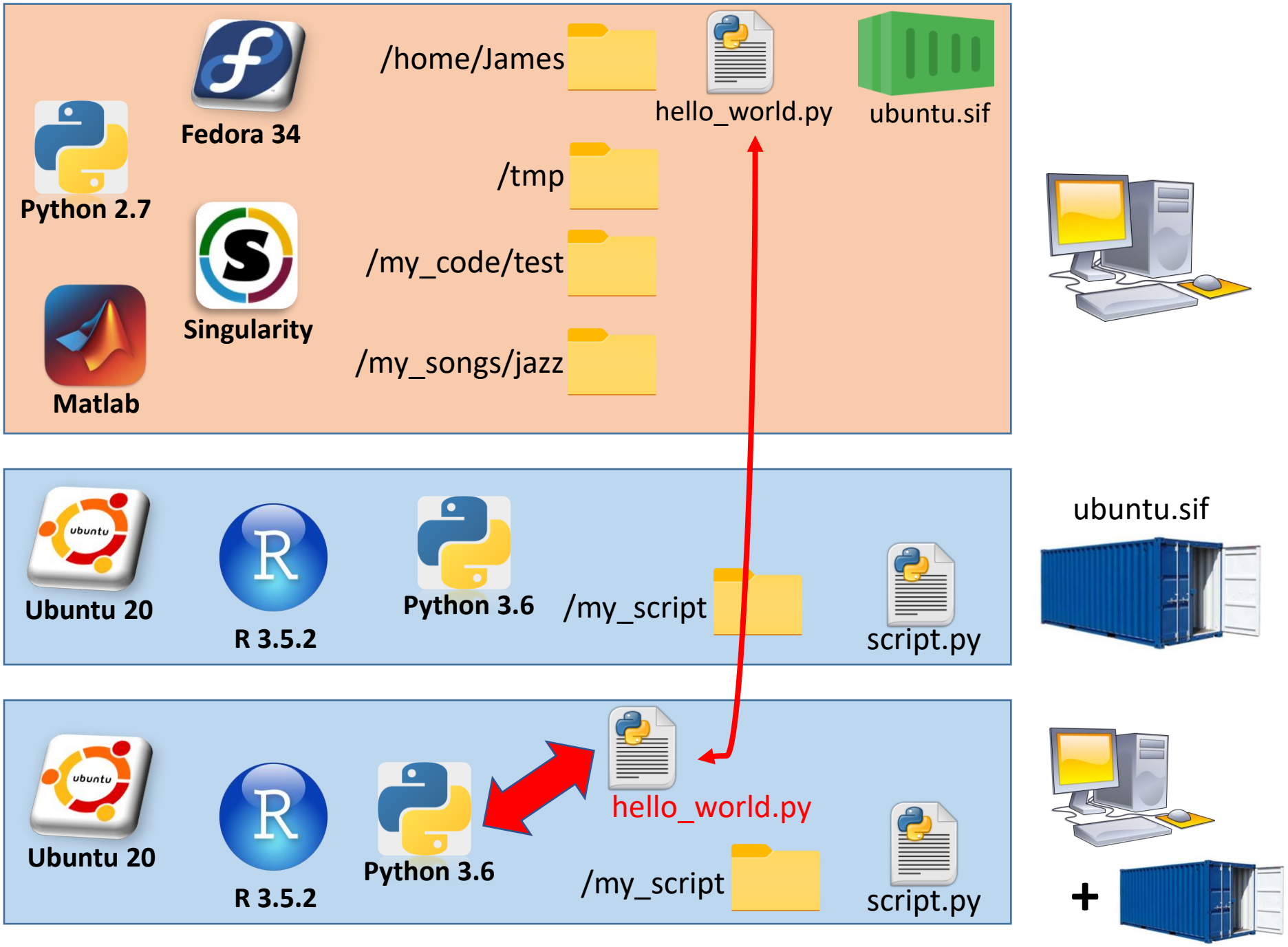
The **exec** commands allows to execute command within the container, so using the software, libraries, dependencies, etc. installed within the container.

Example: run the python script ***hello_world.py*** within the container ***ubuntu.sif*** (i.e. using the python installation within the container). Both files are in the **home directory**.

```
$ singularity exec ubuntu.sif python3 hello_world.py
Hello world!
```

How it works:

1. the script ***hello_world.py*** is in the **host file system** (i.e. home directory)
2. even if **it is outside the container**, it is **visible by the container** (see next slides)
3. it is **executed** by the **python** installed **within the container**
4. the **output** “Hello world!” is then **redirected to the host** shell


Fedora 34
Python 2.7
Singularity
Matlab
/home/James
hello_world.py
ubuntu.sif
/tmp
/my_code/test
/my_songs/jazz
Ubuntu 20
R 3.5.2
Python 3.6
/my_script
script.py
ubuntu.sif
Ubuntu 20
R 3.5.2
Python 3.6
hello_world.py
/my_script
script.py

Run a Singularity image - exec

Why the python script is on the host filesystem but is still visible within the container?

By default, Singularity links the following folders in the host to the corresponding folders in the container

- User home directory (/home/$USER)
- Temp directory (/tmp)
- Other system directories (/sys, /proc, /var/tmp, /etc/resolv.conf, /etc/passwd)

The content of those folder are “visible” (read/write by default) within the container and accessible with the same path used in the host.

If the container is runned by any other folder (or subfolder) within the user home, then the container starting will be that folder.

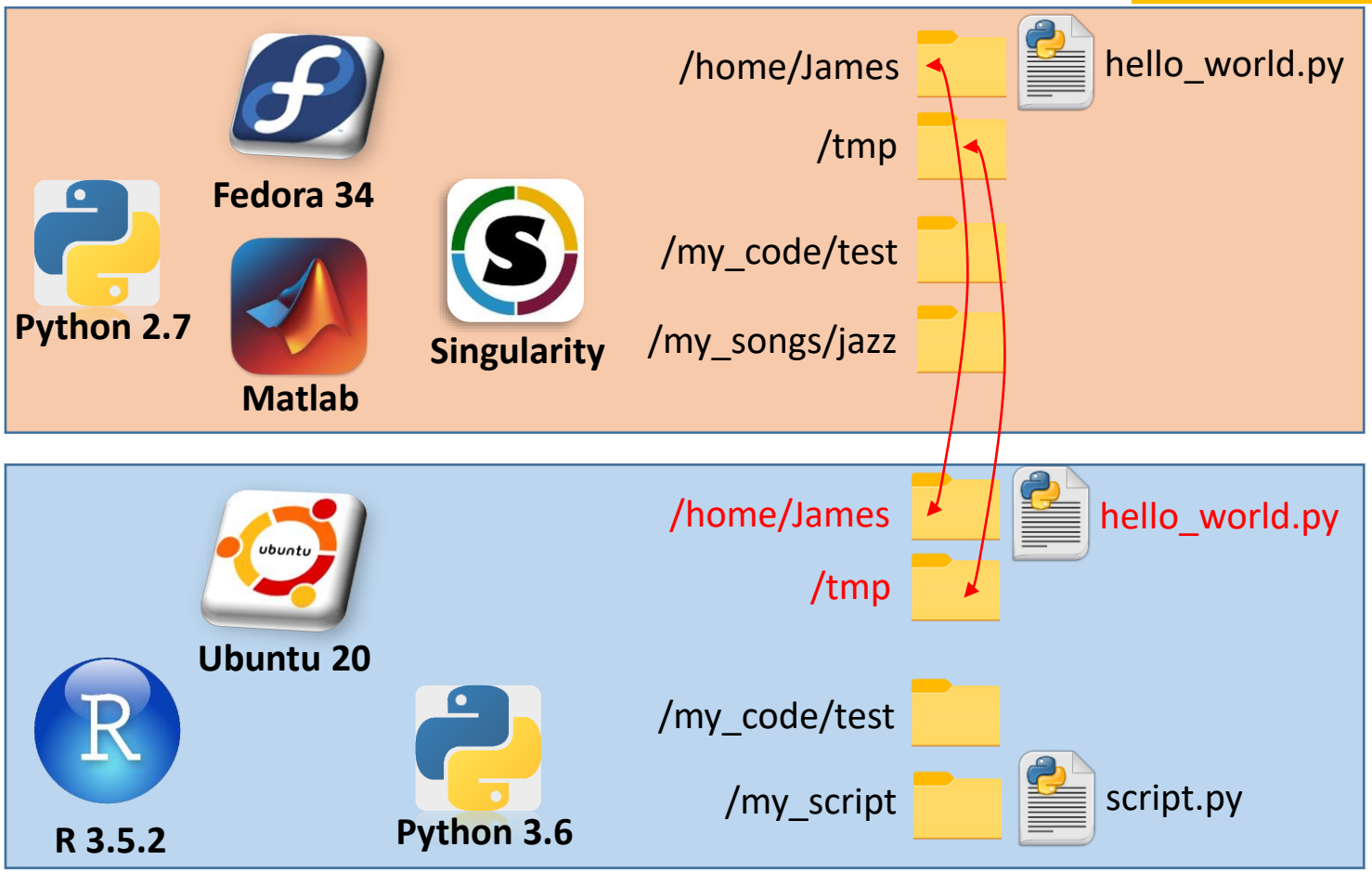
Example: the container is runned from /home/James/Desktop -> the container will start from /home/James/Desktop

Run a Singularity image - exec

```
$ cd /home/James
$ singularity exec my_cont.sif python3 hello_world.py
Hello world!
```

By default, Singularity links some host's folders into the corresponding ones in the container

- User home directory **/home/$USER**
- Temp directory **/tmp**
- …



Run a Singularity image - exec

In addition to the (default) folders, it is possible to bind one (or more) directory in the host to one (or more) directory in the container.

What if my python script **hello_world.py** should read the text file **example.txt** available in the host directory **/data/input/** ?

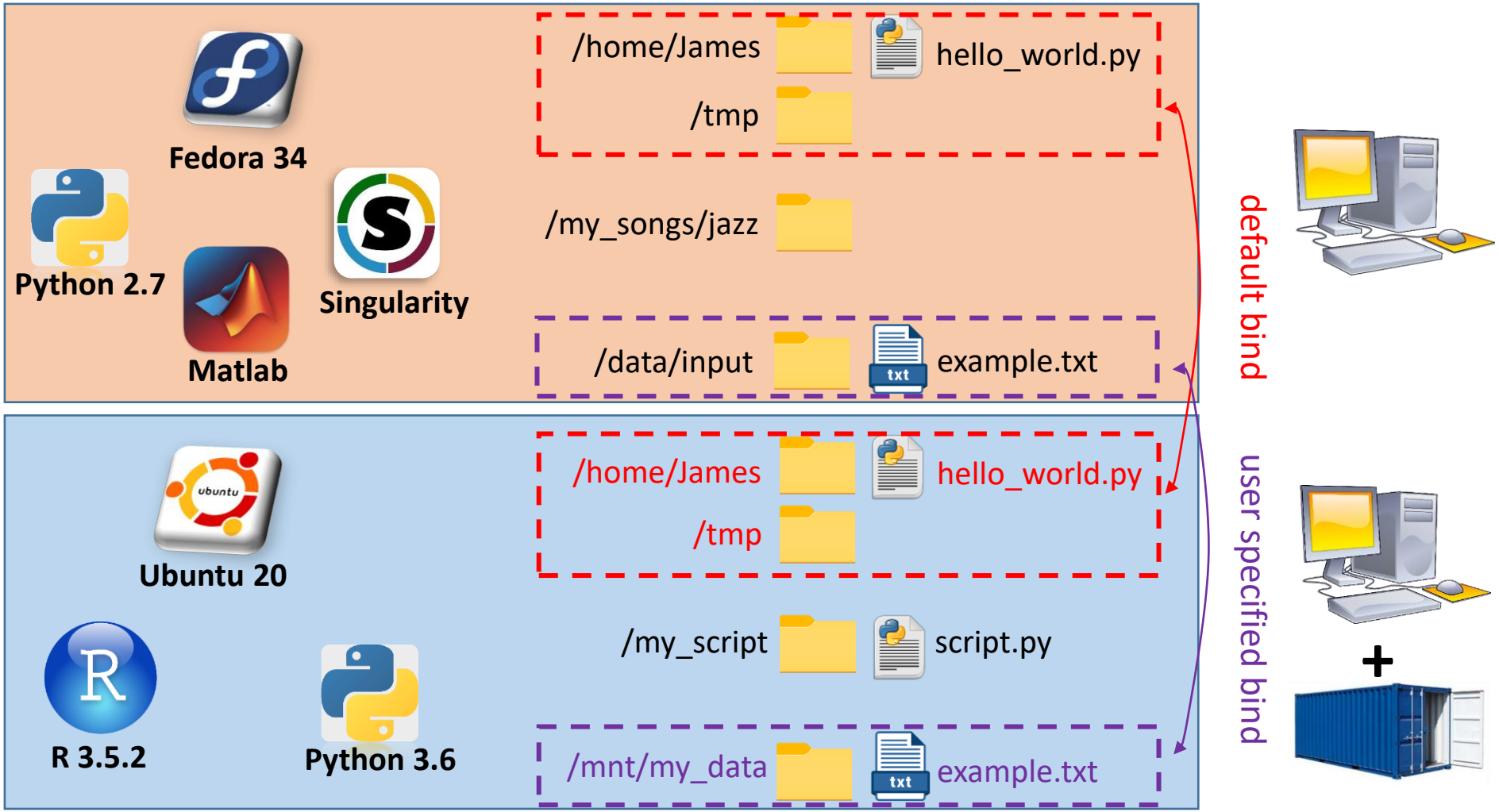
Use the **--bind** option!

As example, let bind the host folder **/data/input/** to the container folder **/mnt/my_data/** using the **--bind** option

```
$ singularity exec \
  --bind /data/input:/mnt/my_data \
  ubuntu.sif \
  python3 hello_world.py /mnt/my_data/example.txt
```

Run a Singularity image - exec

```
$ cd /home/James
$ singularity exec --bind /data/input:/mnt/my_data ubuntu.sif \
    python3 hello_world.py /mnt/my_data/example.txt
Hello world!
```



Run a Singularity image - exec

In the host system, we would have run `$ python3 hello_world.py /data/input/example.txt`

However, the **command to be run within the container must use the container paths**, not the host paths

```
$ singularity exec --bind /data/input:/mnt/my_data ubuntu.sif
python3 hello_world.py /mnt/my_data/example.txt
```

The container and the host system are two separated systems

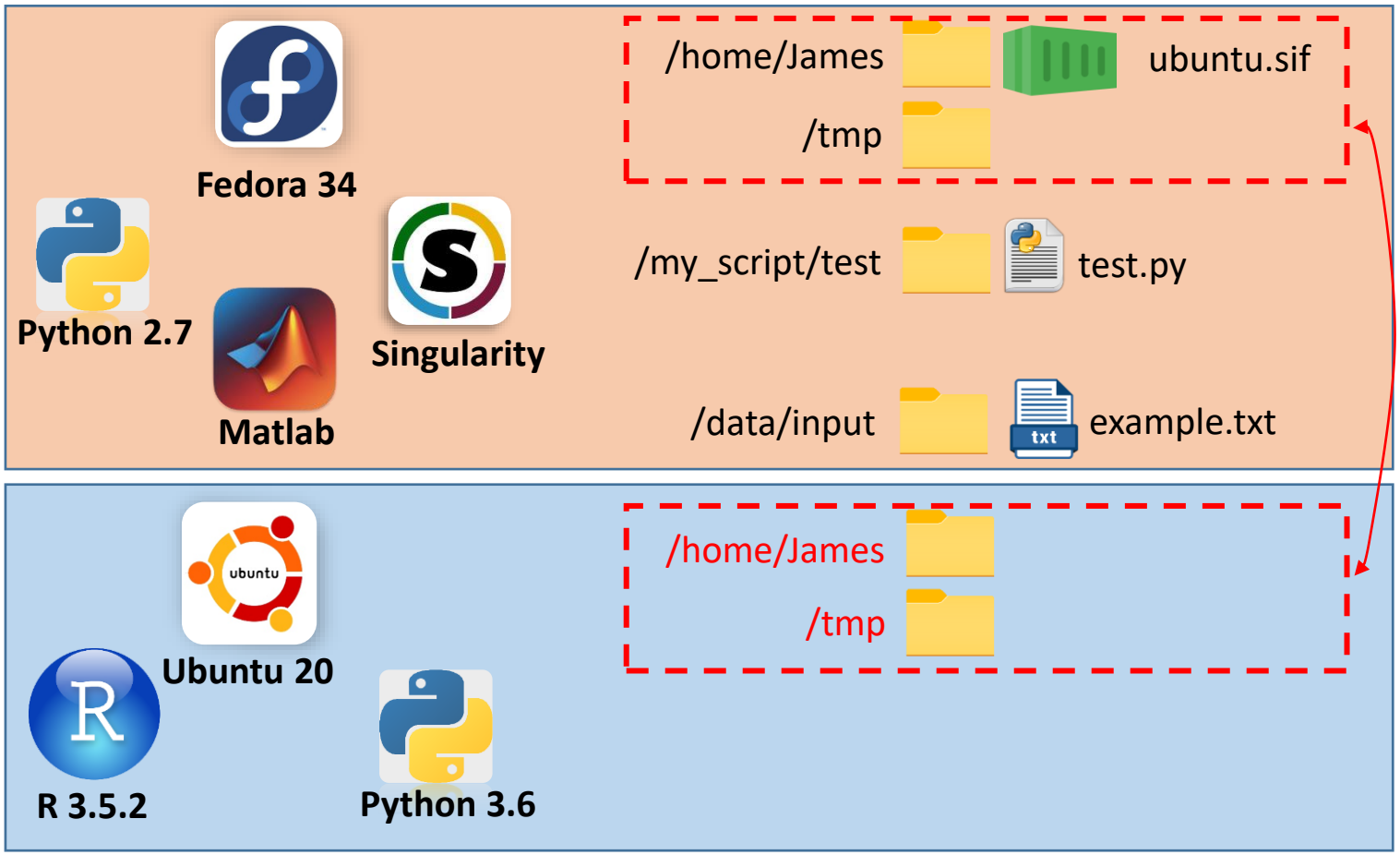
- **`/data/input/`** (and so **`/data/input/example.txt`**) does not exists in the container, it exists only in the host
- **`/mnt/my_data`** exists in the container, and it is a link to the host folder **`/data/input`**, so the right way to identify **`example.txt`** in the container is **`/mnt/my_data/example.txt`**

Indeed, the following command will not work

```
$ singularity exec ubuntu.sif python3 hello_world.py /data/input/example.txt
```

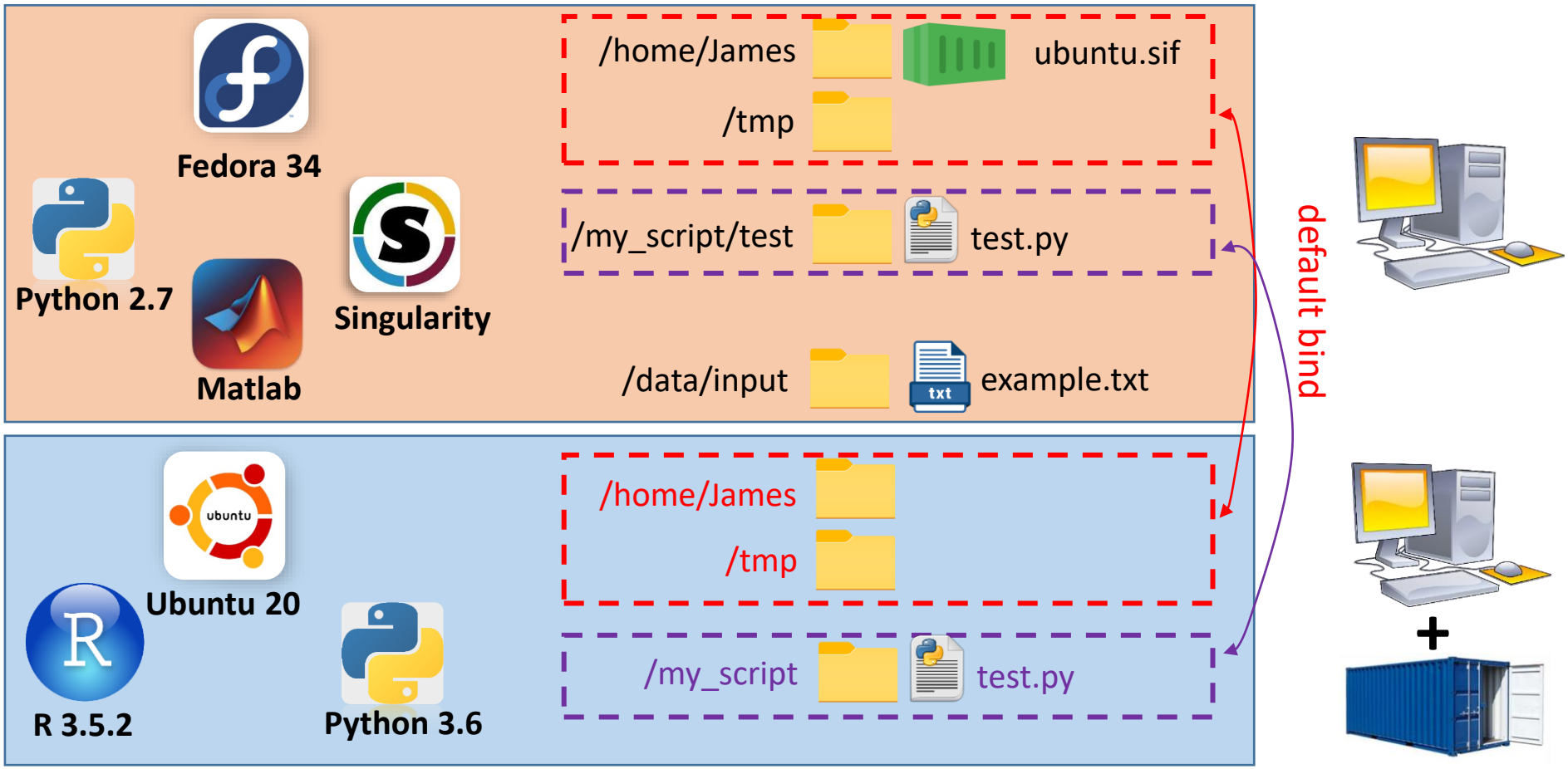
Run a Singularity image - exec

What about running the script **test.py** within the container?
/my_script/test is not one of the default binded folder (e.g. /home/James) …
…so we need to bind the folder **/my_script/test** into a folder within the container and specify the container staring point (**--pwd <path>**).



Run a Singularity image - exec

```
$ singularity exec --bind /my_script/test:/my_script \
   --pwd /my_script \
   debian_run.sif \
   python3 test.py
```



Run a Singularity image - exec

```
$ singularity exec [options] <image> <command>
```

```
$ singularity exec --bind <host_dir>:<cont_dir> <image> <command>
```

```
$ singularity exec --pwd <cont_start_directory> <image> <command>
```

Singularity exec documentation: https://sylabs.io/guides/3.9/user-guide/cli/singularity_exec.html

Run a Singularity image - shell

The **shell** command allows open a shell within the container and interact with it.

```
$ singularity shell ubuntu.sif
Singularity ubuntu.sif :~> cat /etc/os-release | head -n 2
NAME="Ubuntu"
VERSION="16.04.6 LTS (Xenial Xerus)"
Singularity ubuntu.sif :~> exit
```

Flag **`--bind`** can be used even with the **shell** command

Note: The **shell** command is useful during the test of the container, but due to its interactivity it should not be used in a batch job (Slurm)

Run a Singularity image - run

Container may contain the so called “runscript”, i.e. a script that is automatically executed when the container is started, using the **run** command.

The runscript can be any set of shell commands.

When the container is invoked, arguments following the container name are passed to the runscript.

Flag **`--bind`** can be used even with the **run** command.

Example: let’s consider the image **ubuntu_lc.sif** having a runscript that count the number of lines in text files passed as input parameter

```
$ singularity run ubuntu_lc.sif my_note.txt
37
```

Singularity image – exec/shell/run

Singularity exec: execute a command inside the container

```
$ singularity exec <image> <command>
```

Example:

```
$ singularity exec debian_run.sif python3 hello.py
```

Singularity shell: spawn a shell inside the container (interactive)

```
$ singularity shell <image>
```

Singularity run: execute the runscript of the container

```
$ singularity run <image>
```

Example:

```
$ singularity run debian_run.sif
```

Singularity image – exec/shell/run

Useful flags to be used with shell/run/exec (not exhaustive list)

- **`--bind`**: user-bind path specification
- **`--pwd`**: set starting directory
- **`--env`**: pass environment variable to contained process
- **`--no-home`**: do NOT mount users home directory
- **`--nv`**: enable Nvidia GPU
- **`--writable`**: file system accessible as read/write

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Download pre-built image from Singularity Library

Singularity Library is a collection of existing Singularity pre-built images where the user can

- download existing pre-built image
- upload his own pre-built image

Singularity Library contains

- base pre-built images (e.g. ubuntu, centOS)
- application pre-built images (e.g. Golang, octave)

Download pre-built image from Singularity Library

Find an image

- Using web interface https://cloud.sylabs.io/home
- Using command line interface (`$ singularity search <keyword>`)

Example: search octave image

Download pre-built image from Singularity Library

Example: search octave image


```
$ sudo singularity build my_octave.sif library://sylabs/examples/octave
```

Download pre-built image from Docker Hub

Find an image in https://hub.docker.com/

To download a Docker image from Docker Hub and convert it to a Singularity image, use the following command

```
$ sudo singularity build octave_img.sif \
docker://mtmiller/octave
```

NOTE: the above command requires the root privileges, i.e. it can not be run in a HPC system

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Build an image from a Singularity definition file

Given a singularity definition file, the corresponding image can be built using the **build** command

```
$ sudo singularity build <image_name> <def_file>
```

As an example, let's build the image **my_img.sif** using the definition file **my_def_file.def**

```
$ sudo singularity build my_img.sif my_def_file.def
```

Build command

- Create/download an image from Singularity Library

```
$ sudo singularity build <image_name> \
library://<image_path_on_Singularity_library>
```

- Create/download an image from Docker Hub

```
$ sudo singularity build <image_name> \
docker://<image_path_on_docker_hub>
```

- Create an image from a Singularity definition file

```
$ sudo singularity build <image_name> <def_file>
```

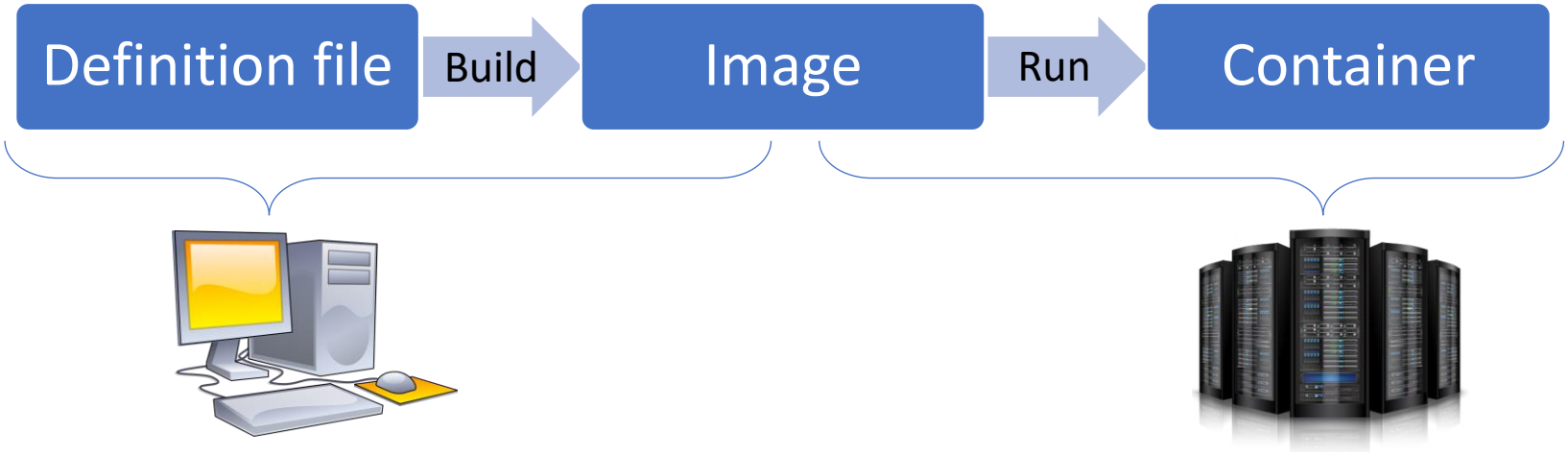
https://sylabs.io/guides/3.7/user-guide/build_a_container.html#

NOTE: the above commands requires the root privileges, so they can not be run in a HPC system

More on Build command

Workflow to use container

1. **Build container on your own computer**
2. Move the image from your computer to the HPC infrastructure
3. Run container on the HPC infrastructure



Why build container in your computer?

- **The build process may require privileges (e.g. sudo)**
- Building is not a computation intensive process
- Test on your computer before running on the HPC infrastructure

More on Build command

Please remember:

- **The build command requires root privileges**
- **Singularity works only in Linux-based OS**

Therefore, to build an image you need **singularity installed** in a **Linux-based OS** where you have the **root privileges** (e.g. you PC).

Please check course webpage to learn how to get a Linux-based OS (with root privileges) on your machine

Alternative*: if you have access to a Linux-based machine where singularity is already installed but you don't have root privileges (e.g. CAPRI) you can use the ***remote build option***

Singularity remote build

Singularity offers the option to run a build remotely, meaning that

- you run the singularity build command in a machine (e.g. CAPRI) using some ad-hoc options (see link below) that do not require the root permissions (i.e. without sudo)
- singularity upload your definition file in a remote server owned by sylabs
- the singularity build command (with the root permissions) is executed in that remote server and so the resulting .sif file is hosted in the remote server
- singularity download the .sif file from the remote server to the machine where you run the singularity build command (e.g. CAPRI)

All the above steps are done automatically by singularity, you only need to follow the instructions here https://cloud.sylabs.io/builder

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Possible scenarios

A) A Singularity image file is already available

B) A Singularity image file is available on an online image repository

C) A Singularity definition file is available

D) Write/modify a Singularity definition file

Singularity definition file

A Singularity definition file (def file) is a text file that contains the “receipt” of how to build the container.

Singularity def file has two main sections

- ***Header***: it describes the core operating system to bootstrap within the container, or an existing image
- ***Body***: it describes the libraries, software and data to put inside the container, together with additional configurations

Singularity definition file

IDEA:

- *If you are able to **install** the **software** in **your PC** using a **terminal***
- *… then you should be **able to install it in a container**.*
- *The definition files contains commands very similar to the ones you would use to install the software in your PC.*

Key idea for developing containers!!!

As an example of a simple def file, let's develop a container with

- *OS: Debian 10*
- *Libraries/software: Python 3.7, R 3.5.2, software from git*

Singularity definition file – example

Header

```
Bootstrap: library
From: debian:10
```

Start from an image available in the Singularity library

An image containing Debian 10

Body

```
%post
  apt-get update
  apt-get install -y r-base r-base-dev
  apt-get install -y python3 python3-dev
  apt-get install -y git
  mkdir $SINGULARITY_ROOTFS/my_software
  cd $SINGULARITY_ROOTFS/my_software
  git clone https://github.com/user/example_repo.git
```

Install R, python and git inside the container

Create a directory and put the code from git

Singularity definition file - header

The header is the first part of any Singularity def file

The first keyword is **`Bootstrap`**, followed by the name of the Bootstrap agent. Bootstrap agents define the source of the "base" image that will be used to build a container, usually an image containing only the OS or some software of interest.

Most used Bootstrap agents:

- **`library`** (Singularity image repository)
- **`docker`** (docker image repository aka Docker Hub)
- **`shub`** (other Singularity image repository aka Singularity hub)
- **`localimage`** (image saved in the file system)

Singularity provides specific keywords for each Bootstrap agents to specify additional information about the chosen "base" image.

The complete list of Bootstrap agent and keywords is available at:
https://sylabs.io/guides/3.7/user-guide/definition_files.html#header

Singularity definition file - header

- "Base" image from Singularity library

```
Bootstrap: library
From: Debian:10
```

- "Base" image from Docker Hub

```
Bootstrap: docker
From: nvidia/cuda
```

- "Base" image from a local image file

```
Bootstrap: localimage
From: /home/james/Documents/my_tensorflow_image.sif
```

Singularity definition file - body

The body is the second part of any Singularity def file.

It defines how the container is build and what it will be inside it.

The body section may contain several sub-sections, all of them are optional. The most common sub-section are:

- **%files**: where to copy files into the container
- **%environment**: where to set to define environment variables that will be set at runtime
- **%post**: where to download file from internet, install software/libraries, create directory/file
- **%runscript**: where to put the commands to execute using **singularity run**
- **%labels**: where to add metadata to the container
- **%help**: where to put the help information available using **singularity run-help**

A complete list of sub-sections is available at https://sylabs.io/guides/3.7/user-guide/definition_files.html#sections

Singularity definition file - body

%files

%file section allows to copy file from the host to the container.

Each line in the %file section is a pair <source> <destination>, where

- <source> is a path on the host filesystem
- <destination> is a path in the container

```
%files
   /home/james/Documents/model_parameter.csv    /mnt/model_parameter.csv
   /home/james/Documents/note.txt
```

%environment

%environment section allow to define environment variables that will be set at runtime. Note that environmental variable that should be used at build time should be specified in the %post section.

```
%environment
   export LC_ALL=C
```

Singularity definition file - body

%post

%post section allows to download file from internet (e.g. using git and wget), install software/libraries, create directory/file, etc.

Commands in the %post section will be interpreted as shell command to the container OS.

```
%post
  apt-get update && apt-get install -y gcc git
  mkdir $SINGULARITY_ROOTFS/my_software
  cd $SINGULARITY_ROOTFS/my_software
  git clone https://github.com/user/example_C_repo.git
  make
  NOW_VAR=`date`
```

Any environmental variable defined in the %post section (e.g. NOW_VAR) is set only at build time. To make environmental variables defined in the %post section available also at run time, they must be included in the $SINGULARITY_ENVIRONMENT variable by text redirection. In the example above, this correspond to add the following command at the end of the post section:

```
echo "export NOW_VAR=\"${NOW_VAR}\"" >> $SINGULARITY_ENVIRONMENT
```

Singularity definition file - body

%runscript

%runscript sections allows to specify the command to execute when the container is invoked using **singularity run**. Any arguments following the container name will be passed to the runscript.

```
%runscript
  echo "Container was created $NOW_VAR"
  echo "Arguments received: $*"
  exec /my_software/my_c_app.exe "$@" /mnt/model_parameter.csv
```

Singularity definition file - body

%labels

%labels section allows to add metadata to the container using a name-value pair. Labels within a container can be inspected using the **singularity inspect** command.

```
%labels
  Author James
  Version 1.0
```

%help

%help section allows to specify help information available using **singularity run-help**

```
%help
  This is a simple example of Singularity container. The container
is based on Debian 10 and it contains a C program available in the
/my_software/ folder. When executed with singularity run, the
container will execute the C program installed in the %post
section, providing as command line input parameters the path to the
file provided in the command line invocation of the container and
the path to the file /mnt/model_parameter.csv
```

Singularity definition file - body

```
%files
  /home/james/Documents/model_parameter.csv /mnt/model_parameter.csv

%environment
  export LC_ALL=C

%post
  apt-get update && apt-get install -y gcc git
  mkdir $SINGULARITY_ROOTFS/my_software && cd $SINGULARITY_ROOTFS/my_software
  git clone https://github.com/user/example_C_repo.git && make
  NOW_VAR=`date`
  echo "export NOW_VAR=\"${NOW_VAR}\"" >> $SINGULARITY_ENVIRONMENT

%runscript
  echo "Container was created $NOW_VAR"
  exec /my_software/my_c_app.exe "$@" /mnt/model_parameter.csv

%labels
  Author James
  Version 1.0

%help
  This is a simple example of Singularity container. The container is based on Debian 10
and it contains a C program available in the /my_software/ folder. When executed with
singularity run, the container will execute the C program installed in the %post section,
providing as command line input parameters the path to the file provided in the command
line invocation of the container and the path to the file /mnt/model_parameter.csv
```

More on singularity definition file

Singularity suggest the following **best practices** for container definition:

- Always install packages, programs, data, and files **into operating system locations** (e.g. not /home, /tmp , or any other directories that might get commonly binded on).
- **Document your container**: write a %help or %apphelp section.
- If you require any **special environment variables** to be defined, add them to the %environment and %appenv sections of the build recipe.
- **Files** should always be **owned by a system account** (UID less than 500).
- Ensure that **sensitive files** like /etc/passwd, /etc/group, and /etc/shadow **do not contain secrets**.
- **Build production containers from a definition file instead of a sandbox**.

A complete list of best practices and additional information about definition files are available here: https://sylabs.io/guides/3.9/user-guide/definition_files.html#best-practices-for-build-recipes

Do you want to see the definition file used to build an image? Use

```
$ singularity inspect -d <image_name>
```

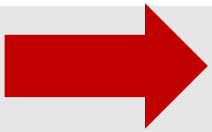
Singularity – Advanced topics

- Sandboxes (mutable containers)
- Container encryption
- Advanced build options
- Work with the Syslab Cloud Library
- Remote build
- Advanced binding/mounting options
- Singularity & MPI
- Singularity & GPU

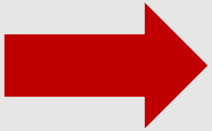
Reproducibility

- In any container engine, the **full reproducibility** is guaranteed by **using/sharing a specific container image**, not building an image from a specific definition file
- The same definition file may result in different images, depending on when/where the build is done

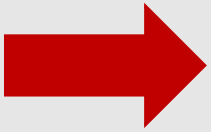
```
Bootstrap: docker
From: Debian:10

%files
  /home/james/param.csv /mnt/model_parameter.csv


%post
  apt-get update && apt-get install -y gcc git


  git clone https://github.com/alexdobin/STAR.git
```



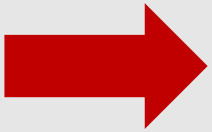
Debian 10 image on DockerHub is regularly updated



File param.csv may be different



Default gcc and git may be updated



Git repository may be updated

- **Reproducibility = image**

Very important!!!

Singularity job in CAPRI

SingularityCE (version 3.8.3) is installed in CAPRI.

Execution of Singularity containers, typically through a **`singularity exec` command**, must be **scheduled using Slurm** within the **allgroups partition**.

In case you need to do a quick check within the container (e.g. check the python version inside the container), the **`singularity shell` command** could be run in a the Slurm **interactive partition**.

IMPORTANT: **no computation should be done** using the `singularity shell` command **in the interactive partition**.

If the Singularity containerized application can use MPI, OpenMP or GPU, then the related Slurm options must be specified.

Additional information about using containerized application in Slurm is available here: https://slurm.schedmd.com/containers.html

Example of Singularity job (serial)

```
#!/bin/sh
#SBATCH --job-name opencv
#SBATCH --error opencv.%j.err
#SBATCH --output opencv.%j.out
#SBATCH --mail-user james@gmail.com
#SBATCH --mail-type END,FAIL
#SBATCH --partition allgroups
#SBATCH --ntasks 1
#SBATCH --mem 32G
#SBATCH --time 02:25:00

cd $SLURM_SUBMIT_DIR

srun singularity exec  tflow_opencv.sif  python script.py
```

Example of Singularity job (OpenMP)

```
#!/bin/bash

#SBATCH --job-name hello_mpi_openmp
#SBATCH --error errors_%j.txt
#SBATCH --output output_%j.txt
#SBATCH --partition allgroups
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 32
#SBATCH --mem 180G
#SBATCH --time 04:15:00

cd $SLURM_SUBMIT_DIR

export OMP_NUM_THREADS=32

srun singularity exec  read_aligner.sif  aligner.exe
```

Request 32 OpenMP threads

Set the required environmental variable OMP_NUM_THREADS
This env variable is defined also inside the container.

singularity exec command

Container image

Command to execute within the container

Example of Singularity job (MPI)

```
#!/bin/sh
#SBATCH --job-name mpi_singularity
#SBATCH --error mpi_singularity.%j.err
#SBATCH --output mpi_singularity.%j.out
#SBATCH --mail-user james@gmail.com
#SBATCH --mail-type END,FAIL
#SBATCH --partition allgroups
#SBATCH --ntasks 8
#SBATCH --mem 1G
#SBATCH --time 08:05:00

cd $SLURM_SUBMIT_DIR

spack load intel-parallel-studio@professional.2019.4

srun singularity exec mpi_ode.sif ./mpi_ode_solver
```

Request 8 MPI tasks

Load MPI (e.g. Intel MPI)

singularity exec command

Container image

MPI application to execute within the container

Example of Singularity job (GPU)

```
#!/bin/sh
#SBATCH --job-name gpu_singularity
#SBATCH --error gpu_singularity.%j.err
#SBATCH --output gpu_singularity.%j.out
#SBATCH --mail-user james@gmail.com
#SBATCH --mail-type END,FAIL
#SBATCH --partition allgroups
#SBATCH --ntasks 1
#SBATCH --gres=gpu:1
#SBATCH --mem 8G
#SBATCH --time 04:45:00

cd $SLURM_SUBMIT_DIR

srun singularity exec --nv  nvidia_nn.sif  ./gpu_nn_train
```

Request 1 GPU

Singularity option to enable GPU

Container image

GPU application to execute within the container