Working with Containers

Please open this in your browser to follow along:

`https://goo.gl/hhkKSP`

v.2.1 (2019-09-10)

Agenda

1. The problem we're solving
2. Virtual machines vs containers
3. Docker vs Singularity
4. Installing and testing Singularity
5. Creating and working with containers
6. Writing a Singularity definition file
7. Using host resources
8. Distributing Singularity containers
9. Cloud resources
10. Docker <-> Singularity interoperability
11. Extra credits (if time allows)

The problem we're solving

Problem (for developers)

Suppose you're writing some software.
It works great on your machine.

However, eventually it has to leave your machine: has to run on your colleague's machine, or deployed in its production environment.

It can be a completely different flavour of OS, with a different set of libraries and supporting tools.

It can be difficult to test if you accounted for all those variations on your own development system.
You may have things in your environment you're not even aware of that make a difference.

Your users could also be less technically inclined to deal with dependencies. You may wish to decrease this friction.

Problem (for users)

Suppose you want to run some piece of software.

First off, you really would like some sort of turn-key solution. Of course there's none, there's only the source code.

The build instuctions indicate 5-years-old out of date libraries on top of a similarly old OS distribution.

And no, the original developer is most certainly no longer available.

You also don't trust this software fully not to mess up your OS.

Or, you want to run it on a remote server for which you don't even have the privileges to comfortably install all the dependencies.

Problem (for researchers)

Suppose you have a piece of scientific software you used to obtain some result.

Then someone half across the globe tries to reproduce it, and can't get it to run, or worse - is getting different results for the same inputs. What is to blame?

Or, even simpler: your group tries to use your software a couple of years after you left, and nobody can get it to work.

For a reproducible way to do science with the help of software, packaging just the source code might not be enough; the environment should also be predictable.

Problem (for server administrators)

Suppose you have a hundred of users, each requesting certain software.

Some of it needs to be carefully built from scratch, as there are no prebuilt packages.

Some of the software works with mutually-incompatible library versions. Possibly even known-insecure ones.

Any such software change has to be injected in a scheduled maintenance window, but users want it yesterday.

And finally, *you most certainly don't* trust any of this software not to mess up your OS. From experience.

What would be a solution?

- **A turnkey solution**

  A recipe that can build a working instance of your software, reliably and fast.

- **BYOE: Bring Your Own Environment**

  A way to capture the prerequisites and environment together with the software.

- **Mitigate security risks**

  Provide a measure of isolation between the software running on a system. No security is perfect, but some is better than none.

The Solution(s)

Solution: Virtual Machines?

A virtual machine is an isolated instance of a **whole other "guest" OS** running under your "host" OS.

A **hypervisor** is responsible for handling the situations where this isolation causes issues for the guest.

From the point of view of the guest, it runs under its own, dedicated hardware. Hence, it's called **hardware-level virtualization**.

Most* guest/host OS combinations can run: you can run Windows on Linux, Linux on Windows, etc.

---

* MacOS being a complicated case due to licensing.

Virtual Machines: the good parts

- **The BYOE principle is fully realized**

  Whatever your environment is, you can package it fully, OS and everything.

- **Security risks are truly minimized**

  Very narrow and secured bridge between the guest and the host means little opportunity for a bad actor to break out of isolation

- **Easy to precisely measure out resources**

  The contained application, together with its OS, has restricted access to hardware: you measure out its disk, memory and alotted CPU.

Virtual Machines: the not so good parts

- **Operational overhead**

  For every piece of software, the full underlying OS has to be run, and corresponding resources allocated.

- **Setup overhead**

  Starting and stopping a virtual machine is not very fast, and/or requires saving its state.

  Changing the allocated resources can be hard too.

- **Hardware availability**

  The isolation between the host and the guest can hinder access to specialized hardware on the host system.

Solution: Containers (on Linux)?

If your software expects Linux, there's a more direct and lightweight way to reach similar goals.

Recent kernel advances allow to isolate processes from the rest of the system, presenting them with their own view of the system.

You can package entire other Linux distributions, and with the exception of the host kernel, all the environment can be different for the process.

From the point of view of the application, it's running on the same hardware as the host, hence containers are sometimes called **operating system level virtualization**.

Containers: the good parts

- **Lower operational overhead**

  You don't need to run a whole second OS to run an application.

- **Lower startup overhead**

  Setup and teardown of a container is much less costly.

- **More hardware flexibility**

  You don't have to dedicate a set portion of memory to your VM well in advance, or contain your files in a fixed-size filesystem.

  Also, the level of isolation is up to you. You may present devices on the system directly to containers if needed.

Containers: the not so good parts

- **Kernel compatibility**

  Kernel is shared between the host and the container, so there may be some incompatibilties.

  Plus, container support is (relatively) new, so it needs a recent kernel on the host.

- **Security concerns**

  The isolation is thinner than in VM case, and kernel of the host OS is directly exposed.

- **Linux on Linux**

  Containers are inherently a Linux technology. You need a Linux host (or a Linux VM) to run containers, and only Linux software can run.

History of containers

The idea of running an application in a different environment is not new to UNIX-like systems.

Perhaps the first effort in that direction is the `chroot` command and concept (1982): presenting applications with a different view of the filesystem (a different root directory `/`).

This minimal isolation was improved in in FreeBSD with `jail` (2000), separating other resources (processes, users) and restricting how applications can interact with each other and the kernel.

Linux developed facilities for isolating and controlling access to some processes with namespaces (2002) and cgroups (2007).

Those facilities led to creation of solutions for containerization, notably LXC (2008), Docker (2013) and Singularity (2016).

Docker vs Singularity

Why did another technology emerge?

Docker

- Docker came about in 2013 and since has been on a meteoric rise as the golden standard for containerization technology.
- A huge amount of tools is built around Docker to build, run, orchestrate and integrate Docker containers.
- Many cloud service providers can directly integrate Docker containers. Docker claims x26 resource efficiency improvement at cloud scale.
- Docker encourages splitting software into microservice chunks that can be portably used as needed.

Docker concerns

- Docker uses a pretty complicated model of images/volumes/metadata, orchestrating swarms of those containers to work together, and it not always very transparent with how those are stored.
- Also, isolation features require superuser privileges; Docker has a persistent daemon running with those privileges and many container operations require root as well.

Docker concerns

- Docker uses a pretty complicated model of images/volumes/metadata, orchestrating swarms of those containers to work together, and it not always very transparent with how those are stored.
- Also, isolation features require superuser privileges; Docker has a persistent daemon running with those privileges and many container operations require root as well.

Both of those issues make Docker undesirable in applications where you don't wholly own the computing resource - HPC environments.

Out of those concerns, and out of scientific community, came Singularity.

Singularity

Singularity was created in 2016 as an HPC-friendly alternative to Docker. It is still in rapid development.

Singularity

Singularity was created in 2016 as an HPC-friendly alternative to Docker. It is still in rapid development.

- It's usually straightforward to convert a Docker container to a Singularity image.

  This gives users access to a vast library of containers.

Singularity

Singularity was created in 2016 as an HPC-friendly alternative to Docker. It is still in rapid development.

- It's usually straightforward to convert a Docker container to a Singularity image.

  This gives users access to a vast library of containers.

- Singularity uses a monolithic, image-file based approach. Instead of dynamically overlaid layers.

  You build a single file on one system and simply copy it over or archive it.

  This addresses the "complex storage" issue with Docker.

Singularity and root privileges

The privilege problem was a concern from the ground-up, to make Singularity acceptable for academic clusters.

Singularity and root privileges

The privilege problem was a concern from the ground-up, to make Singularity acceptable for academic clusters.

- Addressed by having a `setuid`-enabled binary that can accomplish container startup and drop privileges ASAP.

Singularity and root privileges

The privilege problem was a concern from the ground-up, to make Singularity acceptable for academic clusters.

- Addressed by having a `setuid`-enabled binary that can accomplish container startup and drop privileges ASAP.
- Privilege elevation inside a container is impossible: `setuid` mechanism is disabled inside the container, so to be root inside, you have to be root outside.

Singularity and root privileges

The privilege problem was a concern from the ground-up, to make Singularity acceptable for academic clusters.

- Addressed by having a `setuid`-enabled binary that can accomplish container startup and drop privileges ASAP.
- Privilege elevation inside a container is impossible: `setuid` mechanism is disabled inside the container, so to be root inside, you have to be root outside.
- Users don't need explicit root access to operate containers (at least after the initial build).

Singularity and HPC

Thanks to the above improvements over Docker, HPC cluster operators are much more welcoming to the idea of Singularity support.

As a result of a joint Pipeline Interoperability project between Swiss Science IT groups, the UniBE Linux cluser UBELIX started to support Singularity.

Once your software is packaged in Singularity, it should work across all Science IT platforms supporting the technology.

Singularity niche

When is Singularity useful over Docker?

- The major use case was and still is **shared systems**: systems where unprivileged users need the ability to run containers.

  However, an admin still needs to install Singularity for it to function.

- Singularity is useful as an alternative to Docker. If you have admin privileges on the host, Singularity can do more than in unprivileged mode.

  It doesn't have the same level of ecosystem around it, but currently gaining features such as OCI runtime interface, native Kubernetes integration and own cloud services.

Singularity "sales pitch"

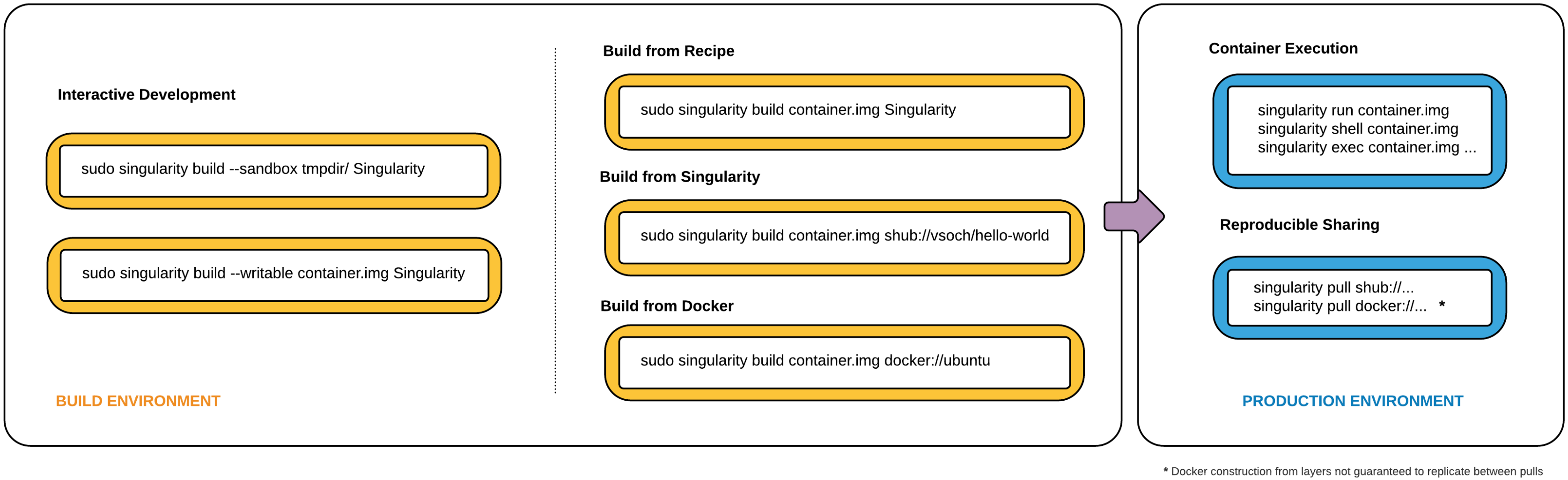
Quoting from Singularity Admin documentation:

> *Untrusted users (those who don't have root access and aren't getting it) can run untrusted containers (those that have not been vetted by admins) safely.*

This won over quite a few academic users; for a sampling:

https://www.sylabs.io/singularity/whos-using-singularity/

Singularity workflow



1. Interactively develop steps to construct a container.
2. Describe the steps in a recipe.
3. Build an immutable container on own machine.
4. Deploy this container in the production environment.

Working with Singularity

Installation and basic use

Singularity versions

There are two major branches of Singularity:

- 2.x branch (currently at 2.6.1): legacy branch with no active development, but still deployed in places.
- 3.x branch (currently at 3.4.0): actively developed branch, with most of the code completely rewritten in Go.

Singularity versions

There are two major branches of Singularity:

- 2.x branch (currently at 2.6.1): legacy branch with no active development, but still deployed in places.
- 3.x branch (currently at 3.4.0): actively developed branch, with most of the code completely rewritten in Go.

Due to freshness of code and new Go dependency, 3.x adoption is slow. This course will cover 3.2.1 release (for UBELIX compatibility).

Singularity versions

There are two major branches of Singularity:

- 2.x branch (currently at 2.6.1): legacy branch with no active development, but still deployed in places.
- 3.x branch (currently at 3.4.0): actively developed branch, with most of the code completely rewritten in Go.

Due to freshness of code and new Go dependency, 3.x adoption is slow. This course will cover 3.2.1 release (for UBELIX compatibility).

Singularity aims to be backwards-compatible: containers built with earlier versions should work with newer ones.

Installing Singularity

Installing Singularity from source is probably preferred, as it's still a relatively new piece of software.

Installing Singularity

Installing Singularity from source is probably preferred, as it's still a relatively new piece of software.

Instructions at: https://sylabs.io/guides/3.2/user-guide/installation.html#install-on-linux

It is required to install Golang compiler >= 1.11.1 as a **build** dependency. It is not required to run the compiled software.

On Ubuntu, Go can be installed with

```
sudo snap install --classic go
```

Exercise:

If you want to try installing Singularity on your Linux system, follow the build instructions.

If you're using the remote training machine, skip this step.

Using Singularity

If you followed build instructions, you should now have `singularity` available from the shell.

```
user@host:~$ singularity --version
singularity version 3.2.1-1
```

Using Singularity

If you followed build instructions, you should now have `singularity` available from the shell.

```
user@host:~$ singularity --version
singularity version 3.2.1-1
```

The general format of Singularity commands is:

```
singularity [<global flags>] <command> [<command flags>] [<arguments>]
```

Singularity is pretty sensitive to the order of those.

Use `singularity help [<command>]` to check built-in help.

You can find the configuration of Singularity under `/usr/local/etc/singularity` if you used the default prefixes.

Container images

A Singularity image is, for practical purposes, a filesystem tree that will be presented to the applications running inside it.

Container images

A Singularity image is, for practical purposes, a filesystem tree that will be presented to the applications running inside it.

A Docker container is built with a series of *layers* that are stacked upon each other to form the filesystem. Layers are collections of updates to files, and must be inspected to find the latest version of the file.

Singularity collapses those into a single, portable file.

Container images

A Singularity image is, for practical purposes, a filesystem tree that will be presented to the applications running inside it.

A Docker container is built with a series of *layers* that are stacked upon each other to form the filesystem. Layers are collections of updates to files, and must be inspected to find the latest version of the file.

Singularity collapses those into a single, portable file.

A container needs to be somehow bootstrapped to contain a base operating system before further modifications can be made.

Pulling Docker images

The simplest way of obtaining a working Singularity image is to pull it from either Docker Hub or Singularity Hub.

Let's try it with CentOS 6:

```
user@host:~$ singularity pull docker://centos:6
```

This will download the layers of the Docker container to your machine and assemble them into an image.

The result will be stored as `centos_6.sif`

Pulling Docker images

```
user@host:~$ singularity pull docker://centos:6
INFO:    Starting build...
Getting image source signatures
Skipping fetch of repeat blob sha256:ff50d722b38227ec8f2bbf0cdbce428b66745077c1
Copying config sha256:5d1ece75fd80b4dd0e4b2d78a1cfebbabad9eb3b5bf48c4e1ba7f9dd2
 1.51 KiB / 1.51 KiB [==================================================] 0
Writing manifest to image destination
Storing signatures
INFO:    Creating SIF file...
INFO:    Build complete: centos_6.sif
```

Note that this **does not require `sudo` or Docker**!

Exercise:

Pull the CentOS 6 image from Dockerhub with the above command

Entering shell in the container

To test our freshly-created container, we can invoke an interactive shell to explore it with **`shell`**:

```
user@host:~$ singularity shell centos_6.sif
Singularity centos_6.sif:~>
```

At this point, you're within the environment of the container.

We can verify we're "running" CentOS:

```
Singularity centos_6.sif:~> cat /etc/centos-release
CentOS release 6.10 (Final)
```

User/group within the container

Inside the container, we are the same user:

```
Singularity centos_6.sif:~> whoami
user
Singularity centos_6.sif:~> exit
user@host:~$ whoami
user
```

We will also have the same groups.

That way, if any host resources are mounted in the container, we'll have the same access privileges.

Root within the container

If we launched `singularity` with `sudo`, we would be `root` inside the container.

```
user@host:~$ sudo singularity shell centos_6.sif
Singularity centos_6.sif:/home/user> whoami
root
```

Root within the container

If we launched `singularity` with `sudo`, we would be `root` inside the container.

```
user@host:~$ sudo singularity shell centos_6.sif
Singularity centos_6.sif:/home/user> whoami
root
```

Most important ly: `setuid` mechanism will not work within the container. Once launched as non-root, no command can elevate your privileges.

Default mounts

In addition to the container filesystem, by default:

- user's home folder,
- `/tmp`,
- `/dev`,
- the folder we've invoked Singularity from

are accessible inside the container.

Default mounts

In addition to the container filesystem, by default:

- user's home folder,
- `/tmp`,
- `/dev`,
- the folder we've invoked Singularity from

are accessible inside the container.

The idea is to provide minimal friction working with software inside the container: no need for extra mounts to access data or store preferences.

It is possible to override this default behavior.

Default mounts

```
user@host:~$ singularity shell centos_6.sif
Singularity centos_6.sif:~> ls ~
[..lists home folder..]
Singularity centos_6.sif:~> touch ~/test_container
Singularity centos_6.sif:~> exit
user@host:~$ ls ~/test_container
/home/user/test_container
```

The current working directory inside the container is the same as outside at launch time.

Running a command directly

Besides the interactive shell, we can execute any command inside the container directly with **exec**:

```
user@host:~$ singularity exec centos_6.sif cat /etc/centos-release
CentOS release 6.10 (Final)
```

Exercise:

Invoke the `python` interpreter with `exec`.

Compare the version with the host system.

Modifying containers

Let's make our own

Modifying the container

Let's try to install some software in the container.

```
user@host:~$ singularity shell centos_6.sif
Singularity centos_6.sif:~> fortune
bash: fortune: command not found
```

`fortune` is not part of the base image. Let's try installing it.

Modifying the container

Let's try to install some software in the container.

```
user@host:~$ singularity shell centos_6.sif
Singularity centos_6.sif:~> fortune
bash: fortune: command not found
```

`fortune` is not part of the base image. Let's try installing it.

```
Singularity centos_6.sif:~> exit
user@host:~$ sudo singularity shell centos_6.sif
Singularity centos_6.sif:~> whoami
root
Singularity centos_6.sif:~> yum -y --enablerepo=extras install epel-release
[...]
[Errno 30] Read-only file system: '/var/lib/rpm/.rpm.lock'
[...]
^C
```

Despite having root, we can't write to the filesystem.

Images and overlays

Singularity image files are read-only squashfs filesystems.

Singularity can use an **overlay**: a layer on top of the image that holds changes to it.

Images and overlays

Singularity image files are read-only squashfs filesystems.

Singularity can use an **overlay**: a layer on top of the image that holds changes to it.

Overlays can be persistent (stored in a folder) or temporary. Singularity 2.x uses a temporary overlay by default.

```
user@host:~$ sudo singularity shell --writable-tmpfs centos_6.sif
Singularity centos_6.sif:~> touch /test
Singularity centos_6.sif:~> ls /test
/test
```

```
user@host:~$ mkdir persistent_overlay
user@host:~$ sudo singularity shell --overlay persistent_overlay centos_6.sif
Singularity centos_6.sif:~> touch /test
Singularity centos_6.sif:~> ls /test
/test
```

Sandbox containers

A more conventional way to write to a container is to use **sandbox** format, which is just a filesystem tree stored in a folder.

```
$ sudo singularity build --sandbox centos-writable docker://centos:6
$ ls centos-writable/
bin  dev  environment  etc  home  lib  lib64  lost+found  media  mnt  opt
proc  root  sbin  selinux  singularity  srv  sys  tmp  usr  var
```

Building sandbox containers requires root.

Sandbox containers

A more conventional way to write to a container is to use **sandbox** format, which is just a filesystem tree stored in a folder.

```
$ sudo singularity build --sandbox centos-writable docker://centos:6
$ ls centos-writable/
bin  dev  environment  etc  home  lib  lib64  lost+found  media  mnt  opt
proc  root  sbin  selinux  singularity  srv  sys  tmp  usr  var
```

Building sandbox containers requires root.

Passing `--writable` to `shell` or `exec` will now enable changes:

```
$ sudo singularity shell --writable centos-writable
Singularity centos-writable:~> touch /test
Singularity centos-writable:~> ls /test
/test
Singularity centos-writable:~> exit
$ ls centos-writable/test
centos-writable/test
```

Writing to a container, finally:

We should now be able to enter it **in writable mode** and install software:

```
user@host:~$ sudo singularity shell --writable centos-writable
Singularity centos-writable:~> yum -y --enablerepo=extras install epel-release
[...]
Singularity centos-writable:~> yum -y install fortune-mod
[...]
Singularity centos-writable:~> exit
user@host:~$ singularity exec centos-writable fortune
[some long-awaited wisdom of a fortune cookie]
```

Default run script

A container can have a "default" command which is run without specifying it.

Inside the container, it's `/singularity`. Let's try modifying it:

```
user@host:~$ sudo nano centos-writable/singularity
```

By default you'll see a sizeable shell script.

```
#!/bin/sh
OCI_ENTRYPOINT=''
OCI_CMD='"/bin/bash"'
CMDLINE_ARGS=""
# [...] #
```

Custom run script

We installed `fortune`, so let's use that instead:

```
#!/bin/sh

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

Now we can invoke it with **`run`**:

```
user@host:~$ singularity run centos-writable
[..some wisdom or humor..]
```

Converting to final container

One way to produce a "final" container is to convert it from the sandbox version:

```
user@host:~$ sudo singularity build fortune.sif centos-writable
[...]
```

Now we can test our container:

```
user@host:~$ singularity run fortune.sif
[..some more wisdom..]
```

Running a container directly

Note that the container file is executable:

```
user@host:~$ ls -lh fortune.sif
-rwxr-xr-x 1 root root 99M Feb 30 13:37 fortune.sif
```

If we run it directly, it's the same as invoking `run`:

```
user@host:~$ ./fortune.sif
[..a cracking joke..]
```

This does require to have `singularity` installed on the host, however, and is just a convenience.

Container definition files

Making the container reproducible

Making the container reproducible

Instead of taking some base image and making changes to it by hand, we want to make this build process reproducible.

This is achieved with definition files called **Definition files**, historically also called "recipes".

Let's try to retrace out steps to obtain a fortune-telling CentOS.

Exercise:

Open a file called `fortune.def` in an editor, and prepare to copy along.

Bootstrapping

The definition file starts with a header section.

The key part of it is the `Bootstrap:` configuration, which defines how we obtain the "base" image.

There are multiple types of bootstrap methods:

- pull an image from a cloud service such as `docker`
- using `yum`/`debootstrap` on the host system to bootstrap a similar one
- `localimage` to base off another image on your computer

We'll be using the Docker method.

```
Bootstrap: docker
From: centos:6
```

Setting up the container

There are 2 sections for setup commands (essentially shell scripts):

1. **`%setup`** for commands to be executed **outside the container**.

   You can use `$SINGULARITY_ROOTFS` to access the container's filesystem, as it is mounted on the host during the build.

2. **`%post`** for commands to be executed **inside** the container.

   This is a good place to set up the OS, such as installing packages.

Setting up the container

Let's save the name of the build host and install `fortune`:

```
Bootstrap: docker
From: centos:6

%setup
  hostname -f > $SINGULARITY_ROOTFS/etc/build_host

%post
  yum -y --enablerepo=extras install epel-release
  yum -y install fortune-mod
  yum clean all
```

Adding files to the container

An additional section, `%files`, allows to copy files or folders to the container.

We won't be using it here, but the format is very similar to `cp`, with sources being outside and the final destination being inside the container:

```
%files
  some/file /some/other/file some/path/
  some/directory some/path/
```

Note that this happens **after** `%post`. If you need the files earlier, copy them manually in `%setup`.

Setting up the environment

You can specify a script to be sourced when something is run in the container.

This goes to the **`%environment`** section. Treat it like `.bash_profile`.

```
%environment
  export HELLO=World
```

Note that by defaut, the host environment variables are passed to the container.

To disable it, use `-e` when running the container.

Setting up the runscript

The runscript (`/singularity`) is specified in the `%runscript` section.

Let's use the file we copied at `%setup` and run `fortune`:

```
%runscript
  read host < /etc/build_host
  echo "Hello, $HELLO! Fortune Teller, built by $host"
  exec /usr/bin/fortune "$@"
```

Testing the built image

You can specify commands to be run at the end of the build process inside the container to perform sanity checks.

Use `%test` section for this:

```
%test
  test -f /etc/build_host
  test -x /usr/bin/fortune
```

All commands must return successfully or the build will fail.

The whole definition file

```
Bootstrap: docker
From: centos:6

%setup
  hostname -f > $SINGULARITY_ROOTFS/etc/build_host
%post
  yum -y --enablerepo=extras install epel-release
  yum -y install fortune-mod
  yum clean all
%environment
  export HELLO="World"
%runscript
  read host < /etc/build_host
  echo "Hello, $HELLO! Fortune Teller, built by $host"
  exec /usr/bin/fortune "$@"
%test
  test -f /etc/build_host
  test -x /usr/bin/fortune
```

Exercise:

Check that your `fortune.def` is the same as above.

Building a container from definition

To fill a container using a definition file, we invoke `build`:

```
user@host:~$ rm fortune.sif
user@host:~$ sudo singularity build fortune.sif fortune.def
[...]
```

Exercise:

1. Bootstrap the image as shown above.
2. Test running it directly.

Inspecting a built container

Container has some metadata you can read:

```
user@host:~$ singularity inspect fortune.sif
==labels==
org.label-schema.build-date: Tuesday_10_September_2019_11:1:10_CEST
org.label-schema.schema-version: 1.0
org.label-schema.usage.singularity.deffile.bootstrap: docker
[...]
```

You can inspect the original definiton file:

```
user@host:~$ singularity inspect -d fortune.sif
Bootstrap: docker
From: centos:6
%setup
  hostname -f > $SINGULARITY_ROOTFS/etc/build_host
[...]
```

See `singularity help inspect` for more options, and `/.singularity.d/` inside the container to see how it's all stored.

Runtime options

Fine-tuning container execution

Host resources

A container can have more host resources exposed.

For providing access to more directories, one can specify bind options at runtime with `-B`:

```
$ singularity run -B source[:destination[:mode]] container.sif
```

where **source** is the path on the host, **destination** is the path in a container (if different) and **mode** is optionally `ro` if you don't want to give write access.

Of course, more than one bind can be specified.
Note that you can't specify this configuration in the container!

System administrators may specify binds that apply to all containers (e.g. `/scratch`).

Host resources

Additionally, devices on the host can be exposed, e.g. the GPU; but you need to make sure that the guest has the appropriate drivers. One solution is to bind the drivers on the container.

For Nvidia CUDA applications specifically, Singularity supports the `--nv` flag, which looks for specific libraries on the host and binds them in the container.

OpenMPI should also work, provided the libraries on the host and in the container are sufficiently close.

If set up correctly, it should work normally with `mpirun`:

```
$ mpirun -np 20 singularity run mpi_job.sif
```

Network

Historically, Singularity defaulted to no network isolation, with an option of full isolation.

With 3.x, Singularity implements in-between options through Container Network Interface:

https://github.com/containernetworking/cni

Network

Historically, Singularity defaulted to no network isolation, with an option of full isolation.

With 3.x, Singularity implements in-between options through Container Network Interface:

https://github.com/containernetworking/cni

Port remapping example:

```
$ sudo singularity instance start --writable-tmpfs \
    --net --network-args "portmap=8080:80/tcp" docker://nginx web2
$ sudo singularity exec instance://web2 nginx
$ curl localhost:8080
[...]
$ sudo singularity instance stop web2
```

This requires root, but it's a common problem with containerization technology at the moment.

Fuller isolation

By default, a container is allowed a lot of "windows" into the host system (dictated by Singularity configuration).

For an untrusted container, you can further restrict this with options like `--contain`, `--containall`.

In this case, you have to manually define where standard binds like the home folder or `/tmp` point.

See `singularity help run` for more information.

Distributing the container

Using the container after creation on another Linux machine is simple: you simply copy the image file there.

Note that you can't just run the image file on a host without Singularity installed!

Exercise:

Test the above, by trying to run `fortune.sif` inside itself.

Distributing the container

Using the container after creation on another Linux machine is simple: you simply copy the image file there.

Note that you can't just run the image file on a host without Singularity installed!

Exercise:

Test the above, by trying to run `fortune.sif` inside itself.

This approach makes it easy to deploy images on clusters with shared network storage.

You can easily integrate Singularity with the usual scheduler scripts (e.g. Slurm).

Cloud services

Current and upcoming ecosystem

Using Singularity Hub

Singularity Hub allows you to cloud-build your containers from Bootstrap files, which you can then simply `pull` on a target host.

https://singularity-hub.org/

Using Singularity Hub

Singularity Hub allows you to cloud-build your containers from Bootstrap files, which you can then simply `pull` on a target host.

https://singularity-hub.org/

This requires a GitHub repository with a `Singularity` definition file. After creating an account and connecting to the GitHub account, you can select a repository and branches to be built.

Afterwards, you can pull the result:

```
user@host:~$ singularity pull shub://kav2k/fortune
[...]
user@host:~$ ./fortune_latest.sif
Hello, World! Fortune Teller, built by shub-builder-1450-kav2k-fortune-[...]
```

Singularity Hub quirks

- Singularity Hub is not like Docker Hub, or similar registry. You can't "push" an image there, it can only be built on their side.
- Singularity Hub is not an official Sylabs project, it's an academic non-profit project by other developers.
- Singularity Hub runs a modified version of Singularity 2.4, making some newer build-time features unavailable (but not runtime features).
- There are no paid plans. Users are allowed a single private project.

Sylabs cloud offering

Starting with Singluarity 3.0, the company behind Singularity aims to provide a range of cloud services to improve Singularity user experience.

- **Container Library** as a counterpart for Docker Hub, serving as an official image repository.
- **Remote Builder** service to allow unprivileged users to build containers in the cloud.
- **KeyStore** service to enable container signature verification.

Sylabs Container Library

Container Libary is the Singularity counterpart to Docker Hub: a cloud registry for both public and private containers.

https://cloud.sylabs.io/library

The Library allows direct upload of pre-built (and signed) containers, unlike Singularity Hub.

```
$ singularity push my.sif library://user/collection/my.sif:latest

$ singularity pull library://user/collection/my.sif:latest
```

As of September 2019, it's still in public beta; eventual plan is a freemium model (pay for private images, pay for builder hours).

Sylabs Remote Builder

Building a container from a recipe requires `sudo`, imposing a need for a separate container creation infrastructure.

Sylabs provides a remote builder service that can build an image from a recipe file, then temporarily host it in Cloud Library to be downloaded.

```
user@host:~$ singularity build --remote output.sif fortune.def
searching for available build agent......INFO:    Starting build...
[...]
user@host:~$ ./output.sif
Hello, World! Fortune Teller, built by ip-10-10-30-146.ec2.internal
[..yet again, a funny quote..]
```

Caveat: all resources for a remote build must be accessible by the build node (i.e. over internet).

Signing containers and Sylabs Keystore

To ensure safety of containers, SIF format allows them to be cryptographically signed.

```
user@host:~$ singularity sign output.sif
user@host:~$ singularity verify output.sif
```

This alone provides assurance of integrity (has not been modified).

For authentication, Sylabs provides a **keyserver** called Keystore, which can be used to check signatures of keys not locally available.

```
user@host:~$ singularity keys push <fingerprint>

user@host2:~$ singularity verify output.sif
```

Sylabs commercial offering

Both the Container Library and Remote Builder are currently in free testing period. However, in future they will have a freemium model.

There will also be on-premise versions of both services (which are not open source).

Besides that, Sylabs offers Singularity PRO: a priority-supported version of Singularity with ready-built packages.

Pricing is "upon request", and is either based on number of hosts or is site-wide.

Running on UBELIX

From a practical standpoint, we want to use the container technology on UBELIX.

Let's try with our toy container:

```
user@host:~$ ssh username@submit.unibe.ch
username@submit01:~$ singularity pull library://kav2k/default/fortune:latest
username@submit01:~$ sbatch -J fortune-test -t 00:00:10 \
--mem-per-cpu 100M --cpus-per-task 1 --wrap "./fortune_latest.sif"
```

Docker and Singularity

Instead of writing a Singularity file, you may write a `Dockerfile`, build a Docker container and convert that.

Pros:

- More portable: for some, using Docker or some other container solution is preferable.
- Easier private hosting: there is no mature private registry tech for Singularity.

Cons:

- Blackbox: Singularity understands less about the build process, in terms of container metadata.
- Complexity: Extra tool to learn if you don't know Docker.

Advice on Docker compatibility: Best Practices

Docker -> Singularity

If you have a Docker image you want to convert to Singularity, you have at least 4 options:

1. Upload the image to a Docker Registry (such as Docker Hub) and `pull`/`Bootstrap` from there.

2. Use a private Docker registry to not rely on external services

3. Directly pull from a local Docker daemon cache

4. Use intermediate format as generated by `docker save`

"Extra credit" topics

Reducing container size

Using traditional Linux distributions, even in minimal configurations, can still be an overkill for running a single application.

One can reduce container size by clearing various artifacts of the build process, such as package manager caches.

Alternatively, one can use minimal Linux distributions, such as Alpine Linux, as a base for containers, though compatibility needs extra testing.

```
$ ll -h
-rwxr-xr-x  1 user group  66M Jun 25 15:04 centos_6.sif*
-rwxr-xr-x  1 user group 2.0M Jun 25 16:08 alpine.sif*
```

Singularity Instances

Running daemon-like persistent services with Singularity (such as a web server) can conveniently be done with the concept of Instances.

A `%startscript` section of the recipe describes what service to launch, which subsequently works with `instance` commands:

```
$ singularity instance start nginx.sif web
$ singularity instance list
INSTANCE NAME    PID      CONTAINER IMAGE
web              790      /home/user/nginx.sif
$ singularity instance stop web
```

While an instance is running, the standard commands like `shell` and `exec` work with an `instance://` namespace.

SCI-F

One of the approaches for building scientific pipelines is bundling several tools in a single "toolset" container.

SCI-F is a proposed standard for discovering and managing tools within such modular containers.

Definition file can have several sections, e.g.:

```
%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
```

SCI-F

You can then discover the apps bundled and run them:

```
$ singularity apps foobar.sif
bar
foo
$ singularity run --app bar foobar.sif
The best guy is bar
```

More sections can be made app-specific, including providing a `help` description:

```
$ singularity help --app fortune moo.sif
fortune is the best app
```

Singularity Checks

A container check is a utility script that can verify a container.

Example uses:

- Making sure no leftover artifacts from the build process remains (e.g. root's bash history)
- Testing for common vulnerabilities
- Custom checks for your specific environment

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

Reproducibility going forward

Pinning a specific version of a base image makes it more probable that in future building the same recipe will be impossible.

Singularity allows for easy storage of resulting containers, and is good at providing backwards compatibility. This provides archival capability (but containers can be large).

Reproducibility going forward

Pinning a specific version of a base image makes it more probable that in future building the same recipe will be impossible.

Singularity allows for easy storage of resulting containers, and is good at providing backwards compatibility. This provides archival capability (but containers can be large).

But a "frozen" container can get other compatibility problems down the line, especially if it needs some host-container interaction.

For example, compiled software in it is no longer optimized for newer hardware architectures.

Reproducibility going forward

Pinning a specific version of a base image makes it more probable that in future building the same recipe will be impossible.

Singularity allows for easy storage of resulting containers, and is good at providing backwards compatibility. This provides archival capability (but containers can be large).

But a "frozen" container can get other compatibility problems down the line, especially if it needs some host-container interaction.

For example, compiled software in it is no longer optimized for newer hardware architectures.

Bottom line: containers are not a silver bullet to solve reproducibility problems, but they help.

Further reading

- Singularity User Guide: https://www.sylabs.io/guides/3.2/user-guide/
- Singularity Admin Guide: https://www.sylabs.io/guides/3.2/admin-guide/
- Singularity White Paper: link
- *Extra credit:* https://rootlesscontaine.rs/

Questions?