Working with Containers

Please open this in your browser to follow along:

`https://goo.gl/hhkKSP`

Agenda

1. The problem we're solving
2. Virtual machines vs containers
3. History of contaners
4. Docker vs Singularity
5. Singularity workflow
6. Installing and testing Singularity
7. Creating and working with containers
8. Writing a Singularity definition file
9. Using host resources
10. Distributing Singularity containers
11. Docker <-> Singularity interoperability

The Problem

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The build instuctions indicate 5-years-old out of date libraries on top of a similarly old OS distribution.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 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. You've been there before.

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 stinker here, of course, with their license.

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.

- **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 host OS is Linux and 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 you so desire.

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

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 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.

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 is quite similar in principles to Docker. In fact, it's pretty straightforward to convert a Docker container to a Singularity image.

Singularity uses a monolithic, image-file based approach. Instead of dynamically overlaid "layers" of Docker, you have a single file you can build once and simply copy over to the target system.

Singularity and root privileges

The privilege problem was a concern from the ground-up, and solved by having a `setuid`-enabled binary that can accomplish container startup - and drop privileges completely as soon as practical.

Privilege elevation inside a container is impossible: to be root inside, you have to be root outside. And users don't need explicit root access to operate containers (at least after the initial build).

Singularity and HPC

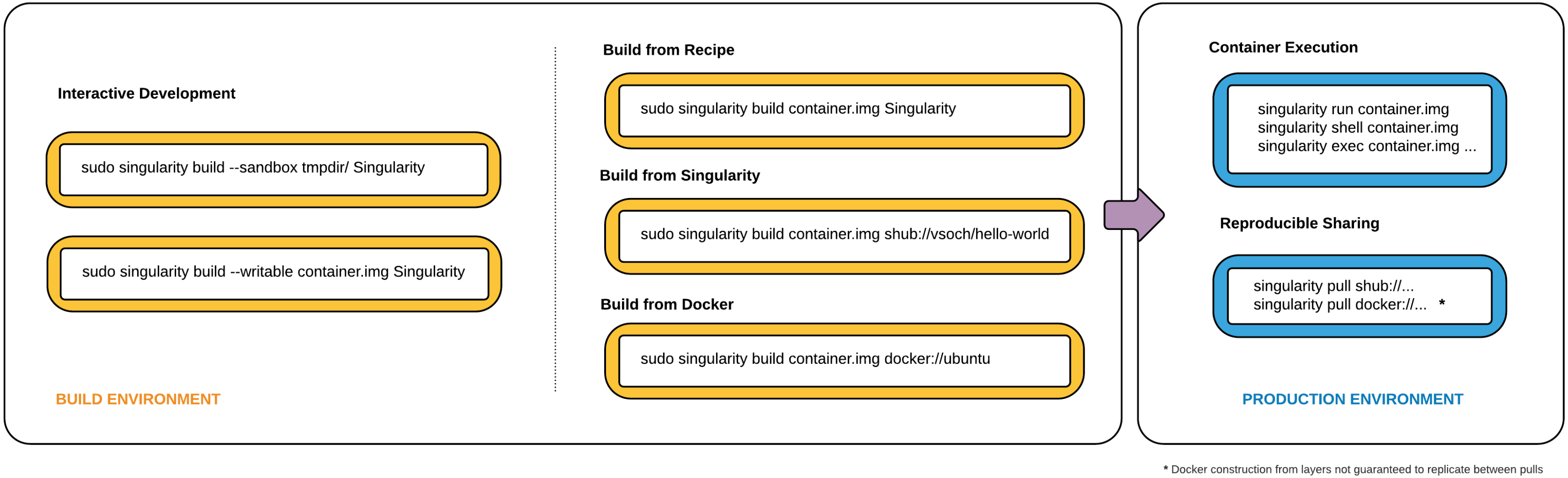
Thanks to the above improvements over Docker, HPC cluster operators are much more welcoming to the idea of Singularity support.

As a result of a joint Pipeline Interoperability project between Swiss Science IT groups, we have a set of guidelines for Singularity deployment and use by scientific community.

Also as part of this effort, UBELIX will support Singularity in the near future (sadly not yet).

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

Singularity workflow



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

Working with Singularity

Installing Singularity

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

Instructions at: http://singularity.lbl.gov/install-linux
(7 lines of shell commands)

On Ubuntu, the `build-essential` metapackage provides enough prerequisites for compilation.

> *Exercise:*
>
> Follow build instructions on your test system.
>
> You will need root access!
>
> Specifically for UBELIX as of Jan 2018, **use `VERSION=2.4`**, as 2.4.1 and 2.4.2 cause issues.

Using Singularity

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

```
user@host:~$ singularity --version
2.4-dist
```

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 prefix.

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. 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 Dockerhub 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.

Pulling Docker images

```
user@host:~$ singularity pull docker://centos:6
WARNING: pull for Docker Hub is not guaranteed to produce the
WARNING: same image on repeated pull. Use Singularity Registry
WARNING: (shub://) to pull exactly equivalent images.
Docker image path: index.docker.io/library/centos:6
Cache folder set to /home/ubuntu/.singularity/docker
Importing: base Singularity environment
Importing: /home/ubuntu/.singularity/docker/sha256:ca9499a209fd9abe7f919a5c9998
Importing: /home/ubuntu/.singularity/metadata/sha256:b4d98049dd466efa5cfbf4b07a
WARNING: Building container as an unprivileged user. If you run this container
WARNING: it may be missing some functionality.
Building Singularity image...
Singularity container built: ./centos-6.img
Cleaning up...
```

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.img
Singularity: Invoking an interactive shell within container...

Singularity centos-6.img:~>
```

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

We can verify we're "running" CentOS:

```
Singularity centos-6.img:~> cat /etc/centos-release
CentOS release 6.9 (Final)
```

User/group within the container

Inside the container, we are the same user:

```
Singularity centos-6.img:~> whoami
user
Singularity centos-6.img:~> 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.img
Singularity: Invoking an interactive shell within container...

Singularity centos-6.img:~> whoami
root
```

Note that the `pull` command complained that building without root does not guarantee that `root` inside will work as expected.

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

Default mounts

Additionally, by default:

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

are accessible inside the container:

```
Singularity centos-6.img:~> ls ~
[..lists home folder..]
Singularity centos-6.img:~> touch ~/test_container
Singularity centos-6.img:~> 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.img cat /etc/centos-release
CentOS release 6.9 (Final)
```

Exercise:

Invoke the `python` interpreter with `exec`.

Compare the version with the host system.

STDIO with container processes

Standard input/output are processed as normal by Singularity. You can redirect them:

```
ubuntu@host:~$ singularity exec centos-6.img echo Boo! > ~/test_container
ubuntu@host:~$ singularity exec centos-6.img cat < ~/test_container
Boo!
```

You can use containers in pipelines:

```
$ singularity exec centos-6.img echo Boo! | singularity exec centos-6.img cat
Boo!
```

Exercise:

Count the number of words in host's `ls /etc`'s output using container's copy of `wc`, then the other way around. Hint:

```
ls /etc | wc -w
```

Modifying the container

Let's try to install some software in the container. We will need root:

```
user@host:~$ sudo singularity shell centos-6.img
Singularity centos-6.img:~> whoami
root
Singularity centos-6.img:~> fortune
bash: fortune: command not found
```

We cannot appeal to the wisdom of `fortune`, and that will not do! Let's install it.

```
Singularity centos-6.img:~> yum -y --enablerepo=extras install epel-release
[...]
Singularity centos-6.img:~> yum -y install fortune-mod
[...]
Singularity centos-6.img:~> fortune
[a random fortune cookie text]
```

Modifying the container

Perfect! Let's exit the container.

```
Singularity centos-6.img:~> exit
exit
user@host:~$
```

We should now be able to use fortune at will:

```
user@host:~$ singularity exec centos-6.img fortune
```

Exercise:

Try the above!

Modifying the container?..

Perfect! Let's exit the container.

```
Singularity centos-6.img:~> exit
exit
user@host:~$
```

We should now be able to use fortune at will:

```
user@host:~$ singularity exec centos-6.img fortune
/.singularity.d/actions/exec: line 9: exec: fortune: not found
```

Wait, what?

```
user@host:~$ singularity shell centos-6.img
Singularity centos-6.img:~> fortune
bash: fortune: command not found
```

What happened to our modifications?

Images and overlays

The image Singularity creates is normally read-only.

While any changes that happen to bind-mounted folders from the host will persist, any changes to the container's filesystem itself are not saved.

Before 2.4, Singularity would just produce write errors; now, it **temporarily** applies the changes (as an "overlay"), which are then discarded when your session ends.

Writing to the image

`shell` and `exec` have a `--writable` flag.

Looks like what we need! Let's try again:

```
user@host:~$ sudo singularity shell --writable centos-6.img
```

Writing to the image?..

`shell` and `exec` have a `--writable` flag.

Looks like what we need! Let's try again:

```
user@host:~$ sudo singularity shell --writable centos-6.img
ERROR  : Unable to open squashfs image in read-write mode: Read-only file syst
em
ABORT  : Retval = 255
```

Well, that didn't work either.

Why? And how *do* we write to an image?

Mutable vs immutable containers

Since 2.4, Singularity has 2 types of containers:

1. Mutable containers. Those are either loose collections of files in a directory, or ext3-formatted images (deprecated).

2. Immutable containers. Those are compressed, read-only squashfs images.

The purpose of mutable containers is to interactively develop a "recipe" for a container, that will be then converted to an immutable "final" form.

`pull` will create immutable containers; for mutable ones, we need `build`.

Creating a writable container

Let's remove our immutable container and build a sandbox container:

```
$ rm centos-6.img
$ sudo singularity build --sandbox centos-writable docker://centos:6
```

This is just a collection of files in `./centos-writable` folder that represents the root of the filesystem:

```
user@host:~$ 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
```

Writing to a container, finally:

We should now be able to enter it **in writable mode** and install software (fortune, and text editors for the next step):

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

Giving container purpose

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 singularity exec -w centos-writable vim /singularity
```

By default you'll see the following:

```
#!/bin/sh

exec /bin/bash "$@"
```

This is a script that will pass all arguments to `/bin/bash`.

Giving container purpose

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

```
#!/bin/sh

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

Exercise:

Make the same modification to your container.

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.img centos-writable
[...]
```

Now we can test our container:

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

Running a container directly

Note that the container file is executable:

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

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

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

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

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 **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 3 currently types of bootstrap methods:

- using `yum/apt/pacman` etc. on the host system to bootstrap a similar one
- pull an image: `docker` (from Dockerhub) or `shub` (from Singularity-Hub)
- `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
```

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 beiong 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 -f /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
%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 -f /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.img
user@host:~$ sudo singularity build fortune.img fortune.def
[...]
```

Exercise:

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

Inspecting a container

If a container was built from a Recipe, it has some metadata you can read:

```
user@host:~$ singularity inspect fortune.img
{
    "org.label-schema.usage.singularity.deffile.bootstrap": "docker",
    "vendor": "CentOS",
    "name": "CentOS Base Image",
[...]
```

You can inspect the original Recipe:

```
user@host:~$ singularity inspect -d fortune.img
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.

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.img
```

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.img
```

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.img` inside itself.

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

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.

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
[...]
Done. Container is at: /home/user/kav2k-fortune-master.simg
user@host:~$ ./kav2k-fortune-master.simg
Hello, World! Fortune Teller, built by shub-builder-1450-kav2k-fortune-[...]
```

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:~$ ssh submit03
username@submit03:~$ singularity pull shub://kav2k/fortune
username@submit03:~$ sbatch -J fortune-test -t 00:00:10 -p el7test
--mem-per-cpu 100M --cpus 1 --wrap "./kav2k-fortune-master.simg"
```

As of Jan 2018, only `submit03` node, `el7test` and `gpu` parititions (running RHEL7) have Singularity 2.4 installed.

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: private version of Singularity Hub, `sregistry`, is still not mature.

Cons:

- Blackbox: Singularity understands less about the build process, in terms of container metadata.
- Source of bugs: Sadly, Docker pull is not perfect and may result in bugs.
- Complexity: Extra tool to learn if you don't know Docker.

Docker -> Singularity

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

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

2. Convert locally with Docker and `docker2singularity` https://github.com/singularityware/docker2singularity

Further topics

- SCI-F: A container format for packing several applications in a single "toolbox" container.
- Minimal operating systems: smaller base images for simple tasks (e.g. Alpine Linux).
- Singularity checks: utility scripts for verifying Singularity images (e.g. scans for vulnerabilities).
- Persistent overlays: a way to store runtime modifications to immutable images.
- Instances: running background services with Singularity.
- Your questions?