The Journal of Open Source Software

Singularity Compose: Orchestration for Singularity Instances

Vanessa Sochat[1]

1 Stanford University Research Computing, Stanford University, Stanford, CA 94305

DOI: 10.21105/joss.01578

Software

- Review
- Repository
- Archive

Submitted: 24 June 2019
Published: 26 August 2019

Summary

Singularity Compose is an orchestration tool for Singularity container instances.

The Singularity container technology started to become popular in 2016, as it offered a more secure option to run encapsulated environments (Kurtzer, Sochat, & Bauer, 2017). Traditionally, this meant that Singularity users could run a script built into the container (called a runscript), execute a custom command, or shell into a container. Unlike Docker (Merkel, 2014), these basic interactions simply interacted with processes in the foreground (e.g., running a script and exiting) and were not appropriate to run background services. This was a task for container instances (Singularity contributors, 2019).

A container instance (Singularity contributors, 2019) equates to running a container in a detached or daemon mode. Instances allow for running persistent services in the background, and then interaction with these services from the host and other containers. Examples of services include databases, web servers, and associated applications that interact with them. While a container technology can provide command line and other programmatic interfaces for interaction with instances, what is also needed is a configuration file for orchestration and customization of several instances. For sibling container technology Docker, Docker Compose (Docker, Inc., 2019) was developed for this purpose. For local and production usage, the user could create a `docker-compose.yml` file to define services, volumes, ports exposed, and other customizations to networking and environment (Docker, Inc., 2019). Notably, there was strong incentive for the development of such a tool, because Docker Compose existed before Kubernetes was available in the middle of 2015 (Wikipedia contributors, 2019).

No equivalent orchestration tool was created for Singularity container instances. While Singularity has empowered enterprise users to run services via platforms such as Kubernetes (Meyer, 2019), these platforms come with privilege. It is often the case that a production Kubernetes cluster is not readily available to a user via his or her institution, or that the user cannot pay a cloud provider to deploy one. However, this does not imply that a non enterprise user (e.g., an open source developer or academic) would not benefit from such an orchestration tool. Unfortunately, since the current trend and strongest potential for making profits is centered around encouraging usage of enterprise tools like Kubernetes (Wikipedia contributors, 2019), there is not any urgent incentive on part of the provider companies to invest in a non-enterprise orchestration tool. It is logical, rational, and understandable that companies exist to make profit, and must make profit to exist. As the need is unfulfilled, it is the responsibility of the open source community to step up.

Singularity Compose

The solution for orchestration of container instances from the open source community is Singularity Compose (Sochat, 2019a). Singularity Compose is software for non enterprise

Figure 1: Singularity Compose

users to easily create a configuration file to control creation and interaction of Singularity container instances. It allows for the creation of a `singularity-compose.yml` file, in which the user can define one or more container services, optionally with exposed ports and volumes on the host. The user can easily define a container binary to build or pull from a remote resource, along with custom scripts to run after creation of the instances. Singularity Compose handles designation of addresses on a local bridge network for each container, and creation of resource files to bind to the containers to "see" one another. Importantly, by way of adding a Singularity Compose to a repository, a user is ensuring not just reproducibility of a container recipe, but also reproducibility of it's build and creation of services. For example, a simplified version of a sequence of steps to build two containers and bring them up as instances might look like this:

```
$ sudo singularity build app/app.sif app/Singularity
$ sudo singularity build nginx/nginx.sif nginx/Singularity.nginx

$ singularity instance start \
   --bind nginx.conf:/etc/nginx/conf.d/default.conf \
   --bind nginx/uwsgi_params.par:/etc/nginx/uwsgi_params.par \
   --bind nginx/cache:/var/cache/nginx \
   --bind nginx/run:/var/run \
   --bind app:/code \
   --bind static:/var/www/static \
   --bind images:/var/www/images \
   --bind etc.hosts:/etc/hosts \
   --net --network-args "portmap=80:80/tcp" --network-args "IP=10.22.0.2" \
   --hostname nginx --writable-tmpfs nginx/nginx.sif nginx

$ singularity instance start \
   --bind app:/code \
   --bind static:/var/www/static \
   --bind images:/var/www/images \
   --bind etc.hosts:/etc/hosts \
```

```
    --net --network-args "portmap=8000:8000/tcp" --network-args "IP=10.22.0.3" \
    --hostname app --writable-tmpfs app/app.sif app

$ singularity instance list
```

This is a complicated set of commands. In the above, we first build the two containers. There are no checks here if the recipes exist, or if the containers themselves already exist. We then start instances for them. If we save these commands in a file, we need to consistently hard code the paths to the container binaries, along with the ip addresses, hostnames, and volumes. There are no checks done before attempting the creation if the volumes meant to be bound actually exist. We also take for granted that we've already generated an `etc.hosts` file to bind to the container at `/etc/hosts`, which will define the container instances to have the same names supplied with `--hostname`. For the networking, we have to be mindful of the default bridge provided by Singularity, along with how to specify networking arguments under different conditions. This entire practice is clearly tedious. For a user to constantly need to generate and then re-issue these commands, it's not a comfortable workflow. However, with Singularity Compose, the user writes a `singularity-compose.yml` file once:

```yaml
version: "1.0"
instances:

  nginx:
    build:
      context: ./nginx
      recipe: Singularity.nginx
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf
      - ./uwsgi_params.par:/etc/nginx/uwsgi_params.par
      - ./nginx/cache:/var/cache/nginx
      - ./nginx/run:/var/run
    ports:
      - 80:80
    depends_on:
      - app
    volumes_from:
      - app

  app:
    build:
      context: ./app
    volumes:
      - ./app:/code
      - ./static:/var/www/static
      - ./images:/var/www/images
    ports:
      - 8000:8000
```

And then can much more readily see and reproduce generation of the same services. The user can easily build all non-existing containers, and bring up all services with one command:

```
$ singularity-compose up
```

And then easily bring services down, restart, shell into a container, execute a command to a container, or run a container's internal runscript.

```
$ singularity-compose down                  # stop services
$ singularity-compose restart               # stop and start services
$ singularity-compose shell app             # shell into an instance
$ singularity-compose exec app "Hello!"     # execute a command
$ singularity-compose run app               # run internal runscript
```

These interactions greatly improve both reproducibility and running of any development workflow that is not appropriate for an enterprise cluster but relies on orchestration of container instances.

For the interested reader, the complete documentation for Singularity Compose (Sochat, 2019a) is provided, along with the code on GitHub (Sochat, 2019b). For additional walkthroughs and complete examples, we direct the reader to the examples repository, also on GitHub (Sochat, 2019c). Contribution by way of additional examples, questions, or requests for development of a new example are appreciated and welcome.

References

Docker, Inc. (2019). *Docker Compose*. https://docs.docker.com/compose/.

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLoS One*, *12*(5), e0177459. doi:10.1371/journal.pone.0177459

Merkel, D. (2014, March). Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* Houston, TX: Belltown Media.

Meyer, D. (2019, April). Sylabs slides singularity updates into its enterprise pro package. https://www.sdxcentral.com/articles/news/sylabs-slides-singularity-updates-into-its-enterprise-pro-package/2019/04/.

Singularity contributors. (2019). Running services — Singularity container 3.2 documentation. https://sylabs.io/guides/3.2/user-guide/running_services.html?highlight=instances.

Sochat, V. (2019a). *Singularity-compose*. https://singularityhub.github.io/singularity-compose.

Sochat, V. (2019b). *Singularity compose Github*. https://github.com/singularityhub/singularity-compose.

Sochat, V. (2019c). *Singularity compose examples*. https://github.com/singularityhub/singularity-compose-examples.

Wikipedia contributors. (2019, June). Kubernetes. https://en.wikipedia.org/w/index.php?title=Kubernetes&oldid=903021989.