shpc Documentation

Release 0.0.37

Vanessa Sochat

Dec 02, 2021

GETTING STARTED

Singularity Registry HPC (shpc) allows you to install containers as modules. Currently, this includes:

- Lmod
- Environment Modules

And container technologies:

- Singularity
- Podman
- Docker

And coming soon:

- Shifter
- Sarus

You can use shpc if you are:

1. a linux administrator wanting to manage containers as modules for your cluster
2. a cluster user that wants to maintain your own folder of custom modules
3. a cluster user that simply wants to pull Singularity images as GitHub packages.

The library contains a collection of module recipes that will install containers, so you can easily use them or write your own. To see the code, head over to the repository. To browse modules available as containers, see the library.

CHAPTER

ONE

GETTING STARTED WITH SINGULARITY REGISTRY (HPC)

Singularity Registry HPC (shpc) can be installed from pypi or directly from the repository. See *Installation* for installation, and then the *Getting Started* section for using the client. You can browse modules available at the Singularity HPC Library.

CHAPTER

TWO

SUPPORT

- For **bugs and feature requests**, please use the issue tracker.
- For **contributions**, visit Caliper on Github.

CHAPTER

THREE

RESOURCES

GitHub Repository The code for shpc on GitHub.

Singularity HPC Library Shows modules available to install as containers.

Autamus Registry Provides many of the shpc container modules, built directly from spack.

3.1 Getting Started

Singularity Registry (HPC) is a tool that makes it easy to install containers as Lmod modules. You can create your own registry entries (e.g., a specification to pull a particular container and expose some number of entrypoints) or the library also provides you with a community set.

If you have any questions or issues, please let us know

3.1.1 Installation

Singularity Registry HPC (shpc) can be installed from pypi, or from source. In all cases, a module technology is required such as lmod (install intstructions) or environment modules (install instructions). Having module software installed means that the `module` command should be on your path. Once you are ready to install shpc along your module software, it's recommended that you create a virtual environment, if you have not already done so.

Virtual Environment

The recommended approach is to install from the repository directly, whether you use pip or another setup approach, and to install a known release. Here is how to clone the repository and do a local install.

```
# Install release 0.0.24
$ git clone -b 0.0.24 git@github.com:singularityhub/singularity-hpc
$ cd singularity-hpc
$ pip install -e .[all]
```

or (an example with python setuptools and installing from the main branch)

```
$ git clone git@github.com:singularityhub/singularity-hpc
$ cd singularity-hpc
$ python setup.py develop
```

if you install to a system python, meaning either of these commands:

```
$ python setup.py install
$ pip install .
```

You will need to put the registry files elsewhere (update the `registry` config argument to the path), as they will not be installed alongside the package. The same is the case for modules - if you install to system python it's recommended to define `module_base` as something else, unless you can write to your install location. Installing locally ensures that you can easily store your module files along with the install (the default until you change it). Installation of singularity-hpc adds an executable, *shpc* to your path.

```
$ which shpc
/opt/conda/bin/shpc
```

This executable should be accessible by an administrator, or anyone that you want to be able to manage containers. Your user base will be interacting with your containers via Lmod, so they do not need access to *shpc*. If you are a user creating your own folder of modules, you can add them to your module path.

Once it's installed, you should be able to inspect the client!

```
$ shpc --help
```

You'll next want to configure and create your registry, discussed next in *Getting Started*. Generally, remember that your modules will be installed in the `modules` folder, and container recipes are nested in `registry`. If you don't want your container images (sif files) installed alongside your module recipes, then you can define `container_base` to be somewhere else. You can change these easily with `shpc config`, as they are defined via these variables in the config:

```
$ shpc config set registry:/<DIR>
$ shpc config set module_base:/<DIR>
$ shpc config set container_base:/<DIR>
```

Also importantly, if you are using environment modules (Tcl) and not LMOD, you need to tell shpc about this (as it defaults to LMOD):

```
$ shpc config set module_sys:tcl
```

You can also easily (manually) update any settings in the `shpc/settings.yaml` file:

```
$ shpc config edit
```

Take a look at this file for other configuration settings, and see the *Getting Started* pages for next steps for setup and configuration, and interacting with your modules.

Warning: You must have your container technology of choice installed and on your $PATH to install container modules.

Environment Modules

If you are using Environment Modules (tcl) and you find that your aliases do not expand, you can use shopt to fix this issue:

```
$ shopt expand_aliases || true
$ shopt -s expand_aliases
```

Pypi

The module is available in pypi as singularity-hpc, and this is primarily to have a consistent means for release, and an interface to show the package. Since the registry files will not install and you would need to change the registry path and module base (making it hard to update from the git remote) we do not encourage you to install from pip unless you know exactly what you are doing.

3.1.2 User Guide

Singularity Registry HPC (shpc) will allow you to install Singularity containers as modules. This means that you can install them as a cluster admin, or as a cluster user. This getting started guide will walk you through setting up a local registry, either for yourself or your user base. If you haven't read *Installation* you should do that first.

Why shpc?

Singularity Registry HPC is created to be modular, meaning that we support a distinct set of container technologies and module systems. The name of the library "Singularity Registry HPC" does not refer specifically to the container technology "Singularity," but more generally implies the same spirit – a single entity that is "one library to rule them all!"

What is a registry?

A registry consists of a database of local containers configuration files, `container.yaml` files organized in the root of the shpc install in one of the `registry` folders. The namespace is organized by Docker unique resources identifiers. When you install an identifier as we saw above, the container binaries and customized module files are added to the `module_dir` defined in your settings, which defaults to `modules` in the root of the install. You should see the *Developer Guide* for more information about contributing containers to this registry.

Really Quick Start

Once you have shpc installed, make sure you tell shpc what your module software is (note that you only need to run this command if you aren't using Lmod, which is the default).

```
$ shpc config set module_sys:tcl
$ shpc config set module_sys:lmod  # default
```

You can then easily install, load, and use modules:

```
$ shpc install biocontainers/samtools
$ module load biocontainers/samtools
$ samtools
```

The above assumes that you've installed the software, and have already added the modules folder to be seen by your module software. If your module software doesn't see the module, remember that you need to have done:

```
$ module use $(pwd)/modules
```

We walk through these steps in more detail in the next section.

Quick Start

After *Installation*, and let's say shpc is installed at `~/singularity-hpc` you can edit your settings in `settings.yaml`. Importantly, make sure your shpc install is configured to use the right module software, which is typicall lmod or tcl. Here is how to change from the default "lmod" to "tcl" and then back:

```
$ shpc config set module_sys:tcl
$ shpc config set module_sys:lmod # this is the default, which we change back to!
```

Once you have the correct module software indicated, try installing a container:

```
$ shpc install python
```

Make sure that the local ./modules folder can be seen by your module software (you can run this in a bash profile or manually, and note that if you want to use Environment Modules, you need to add `--module-sys tcl`).

```
$ module use ~/singularity-hpc/modules
```

And then load the module!

```
$ module load python/3.9.2-slim
```

If the module executable has a conflict with something already loaded, it will tell you, and it's up to you to unload the conflicting modules before you try loading again. If you want to quickly see commands that are supported, use module help:

```
$ module help python/3.9.2-slim
```

If you want to add the modules folder to your modules path more permanently, you can add it to `MODULEPATH` in your bash profile.

```
export MODULEPATH=$HOME/singularity-hpc/modules:$MODULEPATH
```

For more detailed tutorials, you should continue reading, and see *Use Cases*. Also see the *Config* for how to update configuration values with `shpc config`.

Setup

Setup includes, after installation, editing any configuration values to customize your install. The configuration file will default to `shpc/settings.yml` in the installed module, however you can create your own user settings file to take preference over this one as follows:

```
$ shpc config userinit
```

The defaults in either file are likely suitable for most. For any configuration value that you might set, the following variables are available to you:

- `$install_dir`: the shpc folder

- `$root_dir`: the parent directory of shpc (where this README.md is located)

Additionally, the variables `module_base`, `container_base`, and `registry` can be set with environment variables that will be expanded at runtime. You cannot use the protected set of substitution variables (`$install_dir` and `$install_root`) as environment variables, as they will be subbed in by shpc before environment variable replacement. A summary table of variables is included below, and then further discussed in detail.

Table 1: Title

Name	Description	Default
module_sys	Set a default module system. Currently lmod and tcl are supported	lmod
registry	A list of full paths to one or more registry folders (with subfolders with container.yaml recipes)	[$root_dir/registry]
module_base	The install directory for modules	$root_dir/modules
container_base	Where to install containers. If not defined, they are installed alongside modules.	null
container_tech	The container technology to use (singularity or podman)	singularity
updated_at	a timestamp to keep track of when you last saved	never
default_version	A boolean to indicate generating a .version file (LMOD or lua modules only)	true
singularity_module	if defined, add to module script to load this Singularity module first	null
module_name	Format string for module commands exec,shell,run (not aliases) can include `{{ registry }}`, `{{ repository }}`, `{{ tool }}` and `{{ version }}`	`'{{ tool }}'`
bindpaths	string with comma separated list of paths to binds. If set, expored to SINGULARITY_BINDPATH	null
singularity_shell	exported to SINGULARITY_SHELL	/bin/sh
podman_shell	The shell used for podman	/bin/sh
docker_shell	The shell used for docker	/bin/sh
test_shell	The shell used for the test.sh file	/bin/bash
namespace	Set a default module namespace that you want to install from.	null
environment_file	The name of the environment file to generate and bind to the container.	99-shpc.sh
enable_tty	For container technologies that require -t for tty, enable (add) or disable (do not add)	true
config_editor	The editor to use for your config editing	vim
features	A key, value paired set of features to add to the container (see table below)	All features default to null

These settings will be discussed in more detail in the following sections.

Features

Features are key value pairs that you can set to a determined set of values to influence how your module files are written. For example, if you set the gpu feature to "nvidia" in your settings file:

```
container_features:
  gpu: "nvidia"
```

and a container.yaml recipe has a gpu:true container feature to say "this container supports gpu":

```
features:
  gpu: true
```

Given that you are installing a module for a Singularity container, the `--nv` option will be added. Currently, the following features are supported:

Table 2: Title

Name	Description	Default	Options
gpu	If the container technology supports it, add flags to indicate using gpu.	null	nvidia, amd, null
x11	Bind mount ~/.Xauthority or a custom path	null	true (uses default path ~/.Xauthority), false/null (do not enable) or a custom path to an x11 file
home	Specify and bind mount a custom home path	null	custom path for the home, or false/null

Modules Folder

The first thing you want to do is configure your module location, if you want it different from the default. The path can be absolute or relative to `$install_dir` (the shpc directory) or `$root_dir` (one above that) in your configuration file at `shpc/settings.yml`. If you are happy with module files being stored in a `modules` folder in the present working directory, you don't need to do any configuration. Otherwise, you can customize your install:

```
# an absolute path
$ shpc config set module_base:/opt/lmod/modules

# or a path relative to a variable location remember to escape the "$"
$ shpc config set module_base:\$root_dir/modules
```

This directory will be the base where lua files are added, and container are stored. For example, if you were to add a container with unique resource identifier *python/3.8* you would see:

```
$install_dir/modules/
└── python
    └── 3.9.2
        ├── module.lua
        └── python-3.9.2.sif
```

Although your module path might have multiple locations, Singularity Registry HPC assumes this one location to install container modules to in order to ensure a unique namespace.

Container Images Folder

If you don't want your container images (sif files) to live alongside your module files, then you should define the `container_base` to be something non-null (a path that exists). For example:

```
$ mkdir -p /tmp/containers
$ shpc config set container_base:/tmp/containers
```

The same hierarchy will be preserved as to not put all containers in the same directory.

Registry

The registry parameter is a list of one or more registry locations (filesystem directories) where shpc will search for `container.yaml` files. The default registry shipped with shpc is the folder in the root of the repository, but you can add or remove entries via the config variable `registry`

```
# change to your own registry of container yaml configs
$ shpc config add registry:/opt/lmod/registry
```

Note that "add" is used for lists of things (e.g., the registry config variable is a list) and "set" is used to set a key value pair.

Module Names

The setting `module_name` is a format string in Jinja2 that is used to generate your module command names. For each module, in addition to aliases that are custom to the module, a set of commands for run, inspect, exec, and shell are generated. These commands will use the `module_name` format string to determine their names. For example, for a python container with the default `module_name` of "{{ tool }}" we will derive the following aliases for a Singularity module:

```
python-shell
python-run
python-exec
python-inspect-deffile
python-inspect-runscript
```

A container identifier is parsed as follows:

```
# quay.io   /biocontainers/samtools:latest
# <registry>/ <repository>/  <tool>:<version>
```

So by default, we use tool because it's likely closest to the command that is wanted. But let's say you had two versions of samtools - the namespaces would conflict! You would want to change your format string to `{{ repository }}-{{ tool }}` to be perhaps "biocontainers-samtools-exec" and "another-samtools-exec." If you change the format string to `{{ tool }}-{{ version }}` you would see:

```
python-3.9.5-alpine-shell
python-3.9.5-alpine-run
python-3.9.5-alpine-exec
python-3.9.5-alpine-deffile
python-3.9.5-alpine-runscript
```

And of course you are free to add any string that you wish, e.g., `plab-{{ tool }}`

```
plab-python-shell
```

These prefixes are currently only provided to the automatically generated commands. Aliases that are custom to the container are not modified.

Module Software

The default module software is currently Lmod, and there is also support for environment modules that only use tcl (tcl). If you are interested in adding another module type, please open an issue and provide description and links to what you have in mind. You can either specify the module software on the command line:

```
$ shpc install --module-sys tcl python
```

or you can set the global variable to what you want to use (it defaults to lmod):

```
$ shpc config set module_sys:tcl
```

The command line argument, if provided, always over-rides the default.

Container Technology

The default container technology to pull and then provide to users is Singularity, and we have also recently added Podman and Docker, and will add support for Shifter and Sarus soon. Akin to module software, you can specify the container technology to use on a global setting, or via a one-off command:

```
$ shpc install --container-tech podman python
```

or for a global setting:

```
$ shpc config set container_tech:podman
```

If you would like support for a different container technology that has not been mentioned, please also open an issue and provide description and links to what you have in mind.

Commands

The following commands are available! For any command, the default module system is lmod, and you can change this to tcl by way of adding the `--module-sys` argument after your command of interest.

```
$ shpc <command> --module-sys tcl <args>
```

Config

If you want to edit a configuration value, you can either edit the `shpc/settings.yml` file directly, or you can use `shpc config`, which will accept:

- set to set a parameter and value
- get to get a parameter by name
- add to add a value to a parameter that is a list (e.g., registry)
- remove to remove a value from a parameter that is a list

The following example shows changing the default module_base path from the install directory modules folder.

```
# an absolute path
$ shpc config set module_base:/opt/lmod/modules

# or a path relative to the install directory, remember to escape the "$"
$ shpc config set module_base:\$install_dir/modules
```

And then to get values:

```
$ shpc config get module_base
```

And to add and remove a value to a list:

```
$ shpc config add registry:/tmp/registry
$ shpc config remove registry:/tmp/registry
```

You can also open the config in the editor defined in settings at `config_editor`

```
$ shpc config edit
```

which defaults to vim.

Show and Install

The most basic thing you might want to do is install an already existing recipe in the registry. You might first want to show the known registry entries first. To show all entries, you can run:

```
$ shpc show
tensorflow/tensorflow
python
singularityhub/singularity-deploy
```

The default will not show versions available. To flatten out this list and include versions for each, you can do:

```
$ shpc show --versions
tensorflow/tensorflow:2.2.2
python:3.9.2-slim
python:3.9.2-alpine
singularityhub/singularity-deploy:salad
```

To filter down the result set, use `--filter`:

```
$ shpc show --filter bio
biocontainers/bcftools
biocontainers/vcftools
biocontainers/bedtools
biocontainers/tpp
```

To get details about a package, you would then add it's name to show:

```
$ shpc show python
```

And then you can install a version that you like (or don't specify to default to the latest, which in this case is 3.9.2-slim). You will see the container pulled, and then a message to indicate that the module was created.

```
$ shpc install python
...
Module python/3.9.2 is created.
```

```
$ tree modules/
modules/
└── python
    └── 3.9.2
        ├── module.lua
        └── python-3.9.2.sif

2 directories, 2 files
```

You can also install a specific tag (as shown in list).

```
$ shpc install python:3.9.2-alpine
```

Note that Lmod is the default for the module system, and Singularity for the container technology. If you don't have any module software on your system, you can now test interacting with the module via the *Development or Testing* instructions.

Namespace

Let's say that you are exclusively using continers in the namespace ghcr.io/autamus.

```
registry/ghcr.io/
└── autamus
    ├── abi-dumper
    ├── abyss
    ├── accumulo
    ├── addrwatch
    ...
    ├── xrootd
    ├── xz
    └── zlib
```

It can become arduous to type the entire namespace every time! For this purpose, you can set a namespace:

```
$ shpc namespace use ghcr.io/autamus
```

And then instead of asking to install clingo as follows:

```
$ shpc install ghcr.io/autamus/clingo
```

You can simply ask for:

```
$ shpc install clingo
```

And when you are done, unset the namespace.

```
$ shpc namespace unset
```

Note that you can also set the namespace as any other setting:

```
$ shpc config set namespace:ghcr.io/autamus
```

Namespaces currently work with:

- install
- uninstall
- show
- add
- check

List

Once a module is installed, you can use list to show installed modules (and versions). The default list will flatten out module names and tags into a single list to make it easy to copy paste:

```
$ shpc list
   biocontainers/samtools:v1.9-4-deb_cv1
                  python:3.9.2-alpine
                  python:3.9.5-alpine
                  python:3.9.2-slim
                dinosaur:fork
            vanessa/salad:latest
                   salad:latest
  ghcr.io/autamus/prodigal:latest
  ghcr.io/autamus/samtools:latest
    ghcr.io/autamus/clingo:5.5.0
```

However, if you want a shorter version that shows multiple tags alongside each unique module name, just add `--short`:

```
$ shpc list --short

   biocontainers/samtools: v1.9-4-deb_cv1
                  python: 3.9.5-alpine, 3.9.2-alpine, 3.9.2-slim
                dinosaur: fork
            vanessa/salad: latest
                   salad: latest
  ghcr.io/autamus/prodigal: latest
  ghcr.io/autamus/samtools: latest
    ghcr.io/autamus/clingo: 5.5.0
```

Inspect

Once you install a module, you might want to inspect the associated container! You can do that as follows:

```
$ shpc inspect python:3.9.2-slim
 ENVIRONMENT
/.singularity.d/env/10-docker2singularity.sh : #!/bin/sh
export PATH="/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
↪bin"
export LANG="${LANG:-"C.UTF-8"}"
export GPG_KEY="${GPG_KEY:-"E3FF2839C048B25C084DEBE9B26995E310250568"}"
export PYTHON_VERSION="${PYTHON_VERSION:-"3.9.2"}"
export PYTHON_PIP_VERSION="${PYTHON_PIP_VERSION:-"21.0.1"}"
```

(continues on next page)

(continued from previous page)

```
export PYTHON_GET_PIP_URL="${PYTHON_GET_PIP_URL:-"https://github.com/pypa/get-pip/raw/
↪b60e2320d9e8d02348525bd74e871e466afdf77c/get-pip.py"}"
export PYTHON_GET_PIP_SHA256="${PYTHON_GET_PIP_SHA256:-
↪"c3b81e5d06371e135fb3156dc7d8fd6270735088428c4a9a5ec1f342e2024565"}"
/.singularity.d/env/90-environment.sh : #!/bin/sh
# Custom environment shell code should follow

 LABELS
org.label-schema.build-arch : amd64
org.label-schema.build-date : Sunday_4_April_2021_20:51:45_MDT
org.label-schema.schema-version : 1.0
org.label-schema.usage.singularity.deffile.bootstrap : docker
org.label-schema.usage.singularity.deffile.from : ␣
↪python@sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
org.label-schema.usage.singularity.version : 3.6.0-rc.4+501-g42a030f8f

 DEFFILE
bootstrap: docker
from: python@sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
```

We currently don't show the runscript, as they can be very large. However, if you want to see it:

$ shpc inspect –runscript python:3.9.2-slim

Or to get the entire metadata entry dumped as json to the terminal:

```
$ shpc inspect --json python:3.9.2-slim
```

Test

Singularity HPC makes it easy to test the full flow of installing and interacting with modules. This functionality requires a module system (e.g., Lmod) to be installed, and the assumption is that the test is being run in a shell environment where any supporting modules (e.g., loading Singularity or Podman) would be found if needed. This is done by way of extending the exported `$MODULEPATH`. To run a test, you can do:

```
shpc test python
```

If you don't have it, you can run tests in the provided docker container.

```
docker build -t singularity-hpc .
docker run --rm -it singularity-hpc shpc test python
```

Note that the `Dockerfile.tcl` builds an equivalent container with tcl modules.

```
$ docker build -f Dockerfile.tcl -t singularity-hpc .
```

If you want to stage a module install (e.g., install to a temporary directory and not remove it) do:

```
shpc test --stage python
```

To do this with Docker you would do:

```
$ docker run --rm -it singularity-hpc bash
[root@1dfd9fe90443 code]# shpc test --stage python
...
/tmp/shpc-test.fr1ehcrg
```

And then the last line printed is the directory where the stage exists, which is normally cleaned up. You can also choose to skip testing the module (e.g., lmod):

```
shpc test --skip-module python
```

Along with testing the container itself (the commands are defined in the `tests` section of a `container.yaml`.

```
shpc test --skip-module --commands python
```

Uninstall

To uninstall a module, since we are targeting a module folder, instead of providing a container unique resource identifier like *python:3.9.2-alpine*, we provide the module path relative to your module directory. E.g.,

```
$ shpc uninstall python:3.9.2-alpine
```

You can also uninstall an entire family of modules:

```
$ shpc uninstall python
```

The uninstall will go up to the top level module folder but not remove it in the case that you've added it to your `MODULEPATH`.

Pull

Singularity Registry HPC tries to support researchers that cannot afford to pay for a special Singularity registry, and perhaps don't want to pull from a Docker URI. For this purpose, you can use the Singularity Deploy template to create containers as releases associated with the same GitHub repository, and then pull them down directly with the shpc client with the `gh://` unique resource identifier as follows:

```
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:latest
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:salad
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:pokemon
```

In the example above, our repository is called `singularityhub/singularity-deploy`, and in the root we have three recipes:

- Singularity (builds to latest)
- Singularity.salad
- Singularity.pokemon

And in the `VERSION` file in the root, we have `0.0.1` which corresponds with the GitHub release. This will pull to a container. For example:

```
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:latest
singularity pull --name /home/vanessa/Desktop/Code/singularity-hpc/singularityhub-
↪singularity-deploy.latest.sif https://github.com/singularityhub/singularity-deploy/
↪releases/download/0.0.1/singularityhub-singularity-deploy.latest.sif
/home/vanessa/Desktop/Code/singularity-hpc/singularityhub-singularity-deploy.latest.
↪sif
```

And then you are ready to go!

```
$ singularity shell singularityhub-singularity-deploy.latest.sif
Singularity>
```

See the Singularity Deploy repository for complete details for how to set up your container! Note that this uri (`gh://`) can also be used in a registry entry.

Shell

If you want a quick way to shell into an installed module's container (perhaps to look around or debug without the module software being available) you can use `shell`. For example:

```
shpc shell vanessa/salad:latest
Singularity> /code/salad fork

 My life purpose: I cut butter.

                   _________  .====
                  [_________>< :===
                              '====
```

If you want to interact with the shpc Python client directly, you can do shell without a module identifier. This will give you a python terminal, which defaults to ipython, and then python and bypython (per what is available on your system). To start a shell:

```
$ shpc shell
```

or with a specific interpreter:

```
$ shpc shell -i python
```

And then you can interact with the client, which will be loaded.

```
client
[shpc-client]

client.list()
python

client.install('python')
```

Show

As shown above, show is a general command to show the metadata file for a registry entry:

```
$ shpc show python
docker: python
latest:
  3.9.2-slim: sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
tags:
  3.9.2-slim: sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
  3.9.2-alpine:␣
↪sha256:23e717dcd01e31caa4a8c6a6f2d5a222210f63085d87a903e024dd92cb9312fd
filter:
```

(continues on next page)

(continued from previous page)

```
- 3.9.*
maintainer: '@vsoch'
url: https://hub.docker.com/_/python
aliases:
  python: /usr/local/bin/python
```

Or without any arguments, it will show a list of all registry entries available:

```
$ shpc show
python
```

Check

How do you know if there is a newer version of a package to install? In the future, if you pull updates from the main repository, we will have a bot running that updates container versions (digests) as well as tags. Here is how to check if a module (the tag) is up to date.

```
$ shpc check tensorflow/tensorflow
 latest tag 2.2.2 is up to date.
```

And if you want to check a specific digest for tag (e.g., if you use "latest" it is subject to change!)

```
$ shpc check tensorflow/tensorflow:2.2.2
 tag 2.2.2 is up to date.
```

As a trick, you can loop through registry entries with `shpc show`. The return value will be 0 is there are no updates, and 1 otherwise. This is a trick we use to check for new recipes to test.

```

```

Add

It might be the case that you have a container locally, and you want to make it available as a module (without pulling it from a registry). Although this is discouraged because it means you will need to manually maintain versions, shpc does support the "add" command to do this. You can simply provide the container path and the unique resource identifier:

```
$ shpc add salad_latest.sif vanessa/salad:latest
```

If the unique resource identifier corresponds with a registry entry, you will not be allowed to create it, as this would create a namespace conflict. Since we don't have a configuration file to define custom aliases, the container will just be exposed as it's command to run it.

Get

If you want to quickly get the path to a container binary, you can use get.

```
$ shpc get vanessa/salad:latest
/home/vanessa/Desktop/Code/singularity-hpc/modules/vanessa/salad/latest/vanessa-salad-
↪latest-sha256:8794086402ff9ff9f16c6facb93213bf0b01f1e61adf26fa394b78587be5e5a8.sif

$ shpc get tensorflow/tensorflow:2.2.2
/home/vanessa/Desktop/Code/singularity-hpc/modules/tensorflow/tensorflow/2.2.2/
↪tensorflow-tensorflow-2.2.2-
↪sha256:e2cde2bb70055511521d995cba58a28561089dfc443895fd5c66e65bbf33bfc0.sif
```

If you select a higher level module directory or there is no sif, you'll see:

```
$ shpc get tensorflow/tensorflow
tensorflow/tensorflow is not a module tag folder, or does not have a sif binary.
```

You can add `-e` to get the environment file:

```
$ shpc get -e tensorflow/tensorflow
```

We could update this command to allow for listing all sif files within a top level module folder (for different versions). Please open an issue if this would be useful for you.

3.1.3 Developer Guide

This developer guide includes more complex interactions like contributing registry entries and building containers. If you haven't read *Installation* you should do that first.

Creating a Registry

A registry consists of a database of local containers files, which are added to the module system as executables for your user base. This typically means that you are a linux administrator of your cluster, and shpc should be installed for you to use (but your users will not be interacting with it).

The Registry Folder

Although you likely will add custom containers, it's very likely that you want to provide a set of core containers that are fairly standard, like Python and other scientific packages. For this reason, Singularity Registry HPC comes with a registry folder, or a folder with different containers and versions that you can easily install. For example, here is a recipe for a Python 3.9.2 container that would be installed to your modules as we showed above:

```
docker: python
latest:
  3.9.2: sha256:7d241b7a6c97ffc47c72664165de7c5892c99930fb59b362dd7d0c441addc5ed
tags:
  3.9.2: sha256:7d241b7a6c97ffc47c72664165de7c5892c99930fb59b362dd7d0c441addc5ed
  3.9.2-alpine:␣
↪sha256:23e717dcd01e31caa4a8c6a6f2d5a222210f63085d87a903e024dd92cb9312fd
filter:
- 3.9.*
maintainer: '@vsoch'
```

(continues on next page)

(continued from previous page)

```
url: https://hub.docker.com/_/python
aliases:
  python: python
```

And then you would install the module file and container as follows:

```
$ shpc install python:3.9.2
```

But since latest is already 3.9.2, you could leave out the tag:

```
$ shpc install python
```

The module folder will be generated, with the structure discussed in the USer Guide. Currently, any new install will re-pull the container only if the hash is different, and only re-create the module otherwise.

Contributing Registry Recipes

If you want to add a new registry file, you are encouraged to contribute it here for others to use. You should:

1. Add the recipe to the `registry` folder in its logical namespace, either a docker or GitHub uri
2. The name of the recipe should be `container.yaml`. You can use another recipe as a template, or see details in *Writing Registry Entries*
3. You are encouraged to add tests and then test with `shpc test`. See *Test* for details about testing.
4. You should generally choose smaller images (if possible) and define aliases (entrypoints) for the commands that you think would be useful.

A shell entrypoint for the container will be generated automatically. When you open a pull request, a maintainer can apply the `container-recipe` label and it will test your new or updated recipes accordingly. Once your recipe is added to the repository, the versions will be automatically updated with a nightly run. This means that you can pull the repository to get updated recipes, and then check for updates (the bot to do this is not developed yet):

```
$ shpc check python
==> You have python 3.7 installed, but the latest is 3.8. Would you like to install?
yes/no : yes
```

It's reasonable that you can store your recipes alongside these files, in the `registry` folder. If you see a conflict and want to request allowing for a custom install path for recipes, please open an issue.

Writing Registry Entries

An entry in the registry is a container.yaml file that lives in the `registry` folder. You should create subfolders based on a package name. Multiple versions will be represented in the same file, and will install to the admin user's module folder with version subfolders. E.g., two registry entries, one for python (a single level name) and for tensorflow (a more nested name) would look like this:

```
registry/
├── python
│       └── container.yaml
└── tensorflow
    └── tensorflow
        └── container.yaml
```

And this is what gets installed to the modules folder, where each is kept in a separate directory based on version.

```
$ tree modules/
modules/
└── python
    └── 3.9.2
        ├── module.lua
        └── python-3.9.2.sif

2 directories, 2 files
```

So different versions could exist alongside one another.

Registry Yaml Files

Docker Hub

The typical registry yaml file will reference a container from a registry, one or more versions, and a maintainer GitHub alias that can be pinged for any issues:

```
docker: python
latest:
  3.9.2-slim: "sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
↪"
tags:
  3.9.2-slim: "sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
↪"
  3.9.2-alpine:
↪"sha256:23e717dcd01e31caa4a8c6a6f2d5a222210f63085d87a903e024dd92cb9312fd"
filter:
  - "3.9.*"
maintainer: "@vsoch"
url: https://hub.docker.com/_/python
aliases:
  python: /usr/local/bin/python
```

The above shows the simplest form of representing an alias, where each is a key (python) and value (/usr/local/bin/python) set.

Aliases

Each recipe has an optional section for defining aliases in the modulefile; there are two ways of defining them. In the python sample recipe above the simple form is used, using key value pairs:

```
aliases:
  python: /usr/local/bin/python
```

This format is container technology agnostic, because the command (`python`) and executable it targets (`/usr/local/bin/python`) would be consistent between Podman and Singularity, for example. A second form is allowed, using dicts, in those cases where the command requires to specify custom options for the container runtime. For instance, suppose the python interpreter above requires an isolated shell environment (`--cleanenv` in Singularity):

```
aliases:
- name: python
```

(continues on next page)

(continued from previous page)

```
  command: /usr/local/bin/python
  singularity_options: --cleanenv
```

Or perhaps the container required the docker options `-it` because it was an interactive, terminal session:

```
aliases:
- name: python
  command: /usr/local/bin/python
  docker_options: -it
```

For each of the above, depending on the prefix of options that you choose, it will write them into the module files for Singularity and Docker, respectively. This means that if you design a new registry recipe, you should consider how to run it for both kinds of technology. Also note that `docker_options` are those that will also be used for Podman.

Environment Variables

Finally, each recipe has an optional section for environment variables. For example, the container `vanessa/salad` shows definition of one environment variable:

```
docker: vanessa/salad
url: https://hub.docker.com/r/vanessa/salad
maintainer: '@vsoch'
description: A container all about fork and spoon puns.
latest:
  latest: sha256:e8302da47e3200915c1d3a9406d9446f04da7244e4995b7135afd2b79d4f63db
tags:
  latest: sha256:e8302da47e3200915c1d3a9406d9446f04da7244e4995b7135afd2b79d4f63db
aliases:
  salad: /code/salad
env:
  maintainer: vsoch
```

And then during build, this variable is written to a `99-shpc.sh` file that is mounted into the countainer. For the above, the following will be written:

```
export maintainer=vsoch
```

If a recipe does not have environment variables in the container.yaml, you have two options for adding a variable after install. For a more permanent solution, you can update the container.yaml file and install again. The container won't be re-pulled, but the environment file will be re-generated. If you want to manually add them to the container, each module folder will have an environment file added regardless of having this section or not, so you can export them there. When you shell, exec, or run the container (all but inspect) you should be able to see your environment variables:

```
$ echo $maintainer
vsoch
```

Singularity Deploy

Using Singularity Deploy you can easily deploy a container as a GitHub release! See the repository for details. The registry entry should look like:

```
gh: singularityhub/singularity-deploy
latest:
  salad: "0.0.1"
tags:
  salad: "0.0.1"
maintainer: "@vsoch"
url: https://github.com/singularityhub/singularity-deploy
aliases:
  salad: /code/salad
```

Where `gh` corresponds to the GitHub repository, the tags are the extensions of your Singularity recipes in the root, and the "versions" (e.g., 0.0.1) are the release numbers. There are examples in the registry (as shown above) for details.

Choosing Containers to Contribute

How should you choose container bases to contribute? You might consider using smaller images, when possible (take advantage of multi-stage builds) and for aliases, make sure (if possible) that you use full paths. If there is a directive that you need for creating the module file that isn't there, please open an issue so it can be added. Finally, if you don't have time to contribute directly, suggesting an idea via an issue or Slack to a maintainer (@vsoch).

Registry Yaml Fields

Fields include:

Table 3: Title

Name	Description	Required
docker	A Docker uri, which should include the registry but not tag	true
tags	A list of available tags	true
latest	The latest tag, along with the digest that will be updated by a bot in the repository (e.g., tag: digest)	true
maintainer	The GitHub alias of a maintainer to ping in case of trouble	true
filter	A list of patterns to use for adding new tags. If not defined, all are added	false
aliases	Named entrypoints for container (dict)	false
url	Documentation or other url for the container uri	false
description	Additional information for the registry entry	false
env	A list of environment variables to be defined in the container (key value pairs, e.g. var: value)	false
features	Optional key, value paired set of features to enable for the container. Currently allowed keys: *gpu* *home* and *x11*.	varies

A complete table of features is shown here. The

Fields include:

Table 4: Title

Name	Description	Container.yaml Values	Settings.yaml Values	Default	Supported
gpu	Add flags to the container to enable GPU support (typically amd or nvidia)	true or false	null, amd, or nvidia	null	Singularity
x11	Indicate to bind an Xauthority file to allow x11	true or false	null, true (uses default ~/.Xauthority) or bind path	null	Singularity
home	Indicate a custom home to bind	true or false	null, or path to a custom home	null	Singularity, Docker

For bind paths (e.g., home and x11) you can do a single path to indicate the same source and destination (e.g., /my/path) or a double for customization of that (e,g., /src:/dest). Other supported (but not yet developed) fields could include different unique resource identifiers to pull/obtain other kinds of containers. For this current version, since we are assuming HPC and Singularity, we will typically pull a Docker unique resource identifier with singularity, e.g.,:

```
$ singularity pull docker://python:3.9.2
```

Updating Registry Yaml Files

We will be developing a GitHub action that automatically parses new versions for a container, and then updates the registry packages. The algorithm we will use is the following:

- If docker, retrieve all tags for the image
- Update tags: - if one or more filters ("filter") are defined, add new tags that match - otherwise, add all new tags
- If latest is defined and a version string can be parsed, update latest
- For each of latest and tags, add new version information

Development or Testing

If you first want to test singularity-hpc (shpc) with an Lmod installed in a container, a `Dockerfile` is provided for Lmod, and `Dockerfile.tcl` for tcl modules. The assumption is that you have a module system installed on your cluster or in the container. If not, you can find instructions here for lmod or here for tcl.

```
$ docker build -t singularity-hpc .
```

If you are developing the library and need the module software, you can easily bind your code as follows:

```
$ docker run -it --rm -v $PWD/:/code singularity-hpc
```

Once you are in the container, you can direct the module software to use your module files:

```
$ module use /code/modules
```

Then you can use spider to see the modules:

```
# module spider python

-----------------------------------------------------------------------------------------
 ↪-------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
  python/3.9.2: python/3.9.2/module
----------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------

    This module can be loaded directly: module load python/3.9.2/module
```

or ask for help directly!

```
# module help python/3.9.2-slim

---------------------------------------------------- Module Specific Help for
↪"python/3.9.2-slim/module" -----------------------------------------------------
This module is a singularity container wrapper for python v3.9.2-slim


Container:

 - /home/vanessa/Desktop/Code/singularity-hpc/modules/python/3.9.2-slim/python-3.9.2-
↪slim-sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef.sif

Commands include:

 - python-run:
       singularity run <container>
 - python-shell:
       singularity shell -s /bin/bash <container>
 - python-exec:
       singularity exec -s /bin/bash <container> "$@"
 - python-inspect-runscript:
       singularity inspect -r <container>
 - python-inspect-deffile:
       singularity inspect -d <container>

 - python:
       singularity exec <container> /usr/local/bin/python"


For each of the above, you can export:

 - SINGULARITY_OPTS: to define custom options for singularity (e.g., --debug)
 - SINGULARITY_COMMAND_OPTS: to define custom options for the command (e.g., -b)
```

Note that you typically can't run or execute containers within another container, but you can interact with the module system. Also notice that for every container, we expose easy commands to shell, run, exec, and inspect. The custom commands (e.g., Python) are then provided below that.

Make sure to write to files outside of the container so you don't muck with permissions. Since we are using module use, this means that you can create module files as a user or an admin - it all comes down to who has permission to write to the modules folder, and of course use it. Note that I have not tested this on an HPC system but plan to shortly.

3.1.4 Use Cases

Linux Administrator

If you are a linux administrator, you likely want to clone the repository directly (or use a release when they are available). Then you can install modules for your users from the local `registry` folder, create your own module files (and contribute them to the repository if they are useful!) and update the `module_base` to be where you install modules.

```
# an absolute path
$ shpc config module_base:/opt/lmod/shpc
```

If you pull or otherwise update the install of shpc, the module files will update as well. For example, if you start first by seeing what modules are available to install:

```
$ shpc show
```

And then install a module to your shpc modules directory:

```
$ shpc install tensorflow/tensorflow
Module tensorflow/tensorflow:2.2.2 was created.
```

Make sure that lmod knows about the folder

```
$ module use /opt/lmod/shpc
```

(And likely if you administer an Lmod install you have your preferred way of doing this). And then you can use your modules just as you would that are provided on your cluster.

```
$ module load tensorflow/tensorflow/2.2.2
```

You should then be able to use any of the commands that the tensorflow container provides, e.g., python and python-shell:

```
$ python
Python 3.6.9 (default, Oct 8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()

$ tensorflow-tensorflow-shell
_________                  _________________
___ __/_____________________________________ ____/__ /_________   __
__ / _ _ \_ __ \_ ___/ __ \_ ___/_ /_  __ /_ __ \_ | /| / /
_ /  / __/ / / /(__ )/ /_/ / /  _ __/  _ / / /_/ /_ |/ |/ /
/_/  \___//_/ /_//____/ \____//_/  /_/   /_/ \____/____/|__/
You are running this container as user with ID 34633 and group 34633,
which should map to the ID and group for your user on the Docker host. Great!
Singularity> quit()
```

If you want to inspect aliases available or singularity commands to debug:

```
$ module spider tensorflow/tensorflow/2.2.2/module
----------------------------------------------------------------------------------------------------
 ↪----------------------------------------
 tensorflow/tensorflow/2.2.2: tensorflow/tensorflow/2.2.2/module
----------------------------------------------------------------------------------------------------
 ↪----------------------------------------
```

(continues on next page)

(continued from previous page)

```
  This module can be loaded directly: module load tensorflow/tensorflow/2.2.2/module
  Help:
   This module is a singularity container wrapper for tensorflow/tensorflow v2.2.2
   Commands include:
    - tensorflow-tensorflow-shell:
          singularity shell -s /bin/bash /usr/WS2/sochat1/singularity-hpc/modules/
↪tensorflow/tensorflow/2.2.2/tensorflow-tensorflow-2.2.2-
↪sha256:e2cde2bb70055511521d995cba58a28561089dfc443895fd5c66e65bbf33bfc0.sif
    - python:
          singularity exec --nv /usr/WS2/sochat1/singularity-hpc/modules/tensorflow/
↪tensorflow/2.2.2/tensorflow-tensorflow-2.2.2-
↪sha256:e2cde2bb70055511521d995cba58a28561089dfc443895fd5c66e65bbf33bfc0.sif /usr/
↪local/bin/python")
```

Cluster User

If you are a cluster user, you can easily install shpc to your own space (e.g., in `$HOME` or `$SCRATCH` where you keep software) and then use the defaults for the lmod base (the modules folder that is created alongside the install) and the registry. You can also pull the repository to get updated registry entries. If you haven't yet, clone the repository:

```
$ git clone git@github.com:singularityhub/singularity-hpc.git
$ cd singularity-hpc
```

You can then see modules available for install:

```
$ shpc show
```

And install a module to your local modules folder.

```
$ shpc install python
Module python/3.9.2-slim was created.
```

Finally, you can add the module folder to those that lmod knows about:

```
$ module use $HOME/singularity-hpc/modules
```

And then you can use your modules just as you would that are provided on your cluster.

```
$ module load python/3.9.2-slim
```

An error will typically be printed if there is a conflict with another module name, and it's up to you to unload the conflicting module(s) and try again. For this module, since we didn't use a prefix the container python will be exposed as "python" - an easier one to see is "python-shell" - each container exposes a shell command so you can quickly get an interactive shell. Every installed entry will have it's named suffixed with "shell" if you quickly want an interactive session. For example:

```
$ python-shell
Singularity>
```

And of course running just "python" gives you the Python interpreter. If you don't know the command that you need, or want to see help for the module you loaded, just do:

```
$ module spider python/3.9.2-slim/module
------------------------------------------------------------------------------------------
↪----------------------------------------
```

(continues on next page)

(continued from previous page)

```
python/3.9.2-slim: python/3.9.2-slim/module
------------------------------------------------------------------------------------
↪-------------------------------------
  This module can be loaded directly: module load python/3.9.2-slim/module
  Help:
   This module is a singularity container wrapper for python v3.9.2-slim
   Commands include:
    - python-shell:
        singularity shell -s /bin/bash /usr/WS2/sochat1/singularity-hpc/modules/python/
↪3.9.2-slim/python-3.9.2-slim-    ␣
↪sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef.sif
    - python:
        singularity exec /usr/WS2/sochat1/singularity-hpc/modules/python/3.9.2-slim/
↪python-3.9.2-slim-
↪sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef.sif /usr/
↪local/bin/python")
```

The above not only shows you the description, but also the commands if you need to debug. If you want to see metadata about the container (e.g., labels, singularity recipe) then you can do:

```
$ module whatis python/3.9.2-slim
python/3.9.2-slim/module              : Name    : python/3.9.2-slim
python/3.9.2-slim/module              : Version   : module
python/3.9.2-slim/module              : URL     : https://hub.docker.com/_/python
python/3.9.2-slim/module              : Singularity Recipe  : bootstrap: docker
from: python@sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
python/3.9.2-slim/module              : org.label-schema.build-arch  : amd64
python/3.9.2-slim/module              : org.label-schema.build-date  : Sunday_4_April_
↪2021_19:56:56_PDT
python/3.9.2-slim/module              : org.label-schema.schema-version  : 1.0
python/3.9.2-slim/module              : org.label-schema.usage.singularity.deffile.
↪bootstrap  : docker
python/3.9.2-slim/module              : org.label-schema.usage.singularity.deffile.
↪from  :␣
↪python@sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef
python/3.9.2-slim/module              : org.label-schema.usage.singularity.version  :␣
↪3.7.1-1.el7
```

Adding Options

By default, some of the commands will come with singularity options. For example, a container intended for gpu is always going to give you the `--nv` flag. However, it could be the case that you want to define custom options at the time of use. In this case, you can export the following custom environment variables to add them:

SINGULARITY_OPTS: will provide additional options to the base Singularity command, such as `--debug`
SINGULARITY_COMMAND_OPTS: will provide additional options to the command (e.g., exec), such as `--cleanenv`.

Custom Images that are Added

If you add a custom image, the interaction is similar, whether you are a cluster user or administrator. First, let's say we pull a container:

```
$ singularity pull docker://vanessa/salad
```

And we add it to our unique namespace in the modules folder:

```
$ shpc add salad_latest.sif vanessa/salad:latest
```

We can again load the custom module:

```
$ module load vanessa/salad/latest
```

Since we didn't define any aliases via a registry entry, the defaults provided are to run the container (the squashed unique resource identifier, `vanessa-salad-latest` or the same shell, `vanessa-salad-latest-shell`. Of course you can check this if you don't know:

```
$ module spider vanessa/salad/latest/module
----------------------------------------------------------------------------------------
 ↪------------------------------------------------------------------
 vanessa/salad/latest: vanessa/salad/latest/module
----------------------------------------------------------------------------------------
 ↪------------------------------------------------------------------
  This module can be loaded directly: module load vanessa/salad/latest/module
  Help:
   This module is a singularity container wrapper for vanessa-salad-latest vNone
   Commands include:
    - vanessa-salad-latest-shell:
       singularity shell -s /bin/bash /usr/WS2/sochat1/singularity-hpc/modules/
↪vanessa/salad/latest/vanessa-salad-latest-
↪sha256:71d1f3e42c1ceee9c02295577c9c6dfba4f011d9b8bce82ebdbb6c187b784b35.sif
    - vanessa-salad-latest: singularity run /usr/WS2/sochat1/singularity-hpc/modules/
↪vanessa/salad/latest/vanessa-salad-latest-
↪sha256:71d1f3e42c1ceee9c02295577c9c6dfba4f011d9b8bce82ebdbb6c187b784b35.sif
```

And then use them! For example, the command without `-shell` just runs the container:

```
$ vanessa-salad-latest
 You think you have problems? I'm a fork.
            /\
           //\\
           // \\
         ^  \\ //  ^
        / \  ) (  / \
        ) (  ) (  ) (
        \ \_/ /\ \_/ /
         \__ _)(_ __/
           \ \ / /
            ) \/ (
            | /\ |
            | )( |
            | )( |
            | \/ |
            )____(
           /    \
           \_____/
```

And the command with shell does exactly that.

```
$ vanessa-salad-latest-shell
Singularity> exit
```

If you need more robust commands than that, it's recommended to define your own registry entry. If you think it might be useful to others, please contribute it to the repository!

Pull Singularity Images

Singularity Registry HPC tries to support researchers that cannot afford to pay for a special Singularity registry, and perhaps don't want to pull from a Docker URI. For this purpose, you can use the Singularity Deploy template to create containers as releases associated with the same GitHub repository, and then pull them down directly with the shpc client with the `gh://` unique resource identifier as follows:

```
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:latest
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:salad
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:pokemon
```

In the example above, our repository is called `singularityhub/singularity-deploy`, and in the root we have three recipes:

- Singularity (builds to latest)
- Singularity.salad
- Singularity.pokemon

And in the `VERSION` file in the root, we have `0.0.1` which corresponds with the GitHub release. This will pull to a container. For example:

```
$ shpc pull gh://singularityhub/singularity-deploy/0.0.1:latest
singularity pull --name /home/vanessa/Desktop/Code/singularity-hpc/singularityhub-
↪singularity-deploy.latest.sif https://github.com/singularityhub/singularity-deploy/
↪releases/download/0.0.1/singularityhub-singularity-deploy.latest.sif
/home/vanessa/Desktop/Code/singularity-hpc/singularityhub-singularity-deploy.latest.
↪sif
```

And then you are ready to go!

```
$ singularity shell singularityhub-singularity-deploy.latest.sif
Singularity>
```

See the Singularity Deploy repository for complete details for how to set up your container! Note that this uri (`gh://`) can also be used in a registry entry.

3.2 Singularity Registry HPC

These sections detail the internal functions for shpc.

3.3 Internal API

These pages document the entire internal API of SHPC.

3.3.1 shpc package

Submodules

shpc.client module

shpc.client.**get_parser**()

shpc.client.**run_shpc**()
run_shpc is the entrypoint to the singularity-hpc client.

shpc.logger module

class shpc.logger.**ColorizingStreamHandler**(*nocolor=False*, *stream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *use_threads=False*)

Bases: logging.StreamHandler

BLACK = 0

BLUE = 4

BOLD_SEQ = '\x1b[1m'

COLOR_SEQ = '\x1b[%dm'

CYAN = 6

GREEN = 2

MAGENTA = 5

RED = 1

RESET_SEQ = '\x1b[0m'

WHITE = 7

YELLOW = 3

can_color_tty()

colors = {'CRITICAL': 1, 'DEBUG': 4, 'ERROR': 1, 'INFO': 2, 'WARNING': 3}

decorate(*record*)

emit(*record*)
Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

property is_tty

class shpc.logger.**Logger**

Bases: object

cleanup()

debug(*msg*)

error(*msg*)

exit(*msg*, *return_code=1*)

handler(*msg*)

info(*msg*)

location(*msg*)

progress(*done=None*, *total=None*)

set_level(*level*)

set_stream_handler(*stream_handler*)

shellcmd(*msg*)

text_handler(*msg*)

The default snakemake log handler. Prints the output to the console. :param msg: the log message dictionary :type msg: dict

warning(*msg*)

shpc.logger.**setup_logger**(*quiet=False*, *printshellcmds=False*, *nocolor=False*, *stdout=False*, *debug=False*, *use_threads=False*, *wms_monitor=None*)

shpc.main module

shpc.main.container module

shpc.main.lmod

PYTHON MODULE INDEX

S

INDEX

Y