shpc Documentation

Release 0.1.22

Vanessa Sochat

Jun 07, 2023

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 ${RELEASE}
$ git clone -b ${RELEASE} 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 will come from the remote registry shpc-registry by default. 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 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
```

Or set a configuration value on the fly for any command:

```
$ shpc install -c set:views_base:/tmp/views biocontainers/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
```

Also know that by default, we use a remote registry, shpc-registry on GitHub to find recipes for. If you want to use a local filesystem registry (a clone of that registry or your own custom) see the following sections, where we will walk through these steps in more detail.

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 typically 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 inituser
```

When you create a user settings file (or provide a custom settings file one off to the client) the shpc default settings will be read first, and then updated by your file. We do this so that if the default file updates and your user settings is missing a variable, we still use the default. 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 (`$install_dir` and `$root_dir`) 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: Settings

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 remotes (e.g., GitHub addresses) or local directories (each with subfolders with container.yaml recipes)	["https://github.com/singularityhub/shpc-registry"]
sync_registry	A default remote to sync from (is not required to have an API/docs, as it is cloned).	https://github.com/singularityhub/shpc-registry
module_base	The install directory for modules	$root_dir/modules
container_base	Where to install containers. If not defined, they are installed in "containers" in the install root	$root_dir/containers
container_tech	The container technology to use (singularity or podman)	singularity
views_base	The default root for creating custom views. Defaults to `views` in the root directory.	$root_dir/views
default_view	Install to this default view (e.g., meaning you always create a second symlink tree of the same modules)	unset
updated_at	a timestamp to keep track of when you last saved	never
label_separator	When parsing labels, replace newlines with this string	', '
default_version	Should a default version be used?	module_sys
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
wrapper_shell	The shell used for wrapper scripts	/bin/bash
wrapper_scripts:enabled	enable or disable generation of wrapper scripts, instead of module aliases	false
wrapper_scripts:docker	The name of the generic wrapper script template for docker	docker.sh
wrapper_scripts:podman	The name of the generic wrapper script template for podman	docker.sh
wrapper_scripts:singularity	The name of the generic wrapper script template for singularity	singularity.sh
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)	null

Note that any configuration value can be set permanently by using `shpc config` or manually editing the file, but you can also set config values "one off." As an example, here is a "one off" command to install to a different shpc module root:

```
$ shpc install -c set:modules_base:/tmp/modules ghcr.io/autamus/clingo
```

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 containers are stored in a directory alongside it. For example, if you were to add a container with unique resource identifier *python/3.8* you would see:

```
$install_dir/modules/
└── python
```

(continues on next page)

(continued from previous page)

```
    └── 3.9.2
        └── module.lua

$install_dir/containers/
└── python
    └── 3.9.2
        └── python-3.9.2.sif
```

Singularity Registry HPC uses this simple directory structure to ensure a unique namespace.

Container Images Folder

If you don't want your container images (sif files) to live in the root of shpc in a directory called "containers," then you should define the `container_base` to be something different. 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. It's strongly recommended to keep modules separate from containers for faster loading (applies to container technologies like Singularity that pull binary files directly).

Registry

The registry parameter is a list of one or more registry locations (filesystem directories or remote GitHub repositories with the same structure) where shpc will search for `container.yaml` files. The default registry used to be shipped with shpc, but as of version 0.1.0 is provided remotely. This means that by default, you don't need to worry about updating or syncing recipes - they will always be retrieved from the latest, as the remote registry shpc-registry is automatically updated. However, you have several options for managing your own (or updating) recipes.

1. Use the default remote, no additional work needed
2. Clone the default remote to a local filesystem folder and manage manually (e.g., git pull)
3. Create your own local registry in addition (or without) the remote.
4. For any local registry, you can sync (`shpc sync`) from a remote.

If you want to do the first, no further action is needed! Each of these remaining examples will be described here, and for instructions for creating your own registry, see *Developer Guide*.

1. Use the Default Remote

Congratulations, you are done! This is the default and you don't need to make any changes.

2. Clone a Remote Registry

It could be the case that you want to start with a remote registry, but keep it locally with your own changes or secrets. This is essentially turning a remote registry into a filesystem (local) one. The easiest thing to do here is to clone it to your filesyste, and then add to shpc as a filesystem registry.

```
# Clone to a special spot
$ git clone https://github.com/singularityhub/shpc-registry /opt/lmod/my-registry

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

Since add is adding to a list, you might want to open your settings.yaml and ensure that the order is to your liking. The order determines the search path, and you might have preferences about what is searched first.

3. Create A Local Registry

This would correspond to the same set of steps as above, but starting from scratch! For example:

```
$ mkdir -p /opt/lmod/my-registry
$ cd /opt/lmod/my-registry
```

And then you might want to inspect *Add* to see how to use `shpc add` to generate new container.yaml files. See *Creating a FileSystem Registry* for instructions on how to create a registry and getting_started-developer-manual-registry-entries to populate the registry with new entries. After that, you'll still want to ensure your filesystem registry is known to shpc:

```
$ shpc config add registry:/opt/lmod/my-registry
```

4. Sync from a Remote

See getting_started-commands-sync-registry: for instructions of how to sync from a remote. You'll want to ensure you have added a filesystem registry to be known to shpc to sync to.

Want to design your own remote registry? See the *Developer Guide*.

Default Version

The default version setting is there to support you telling shpc how you want module versions to be selected. There are four options:

- `null` do not set any kind of default version, it will be manually controlled by the installer (`false` also supported for backwards compatibility)
- `module_sys`: allow the module software to choose (`true` also supported for backwards compatibility)
- `last_installed`: always set default version to the last version installed
- `first_installed`: only set default version for the first installed

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, exec, shell, inspect, and container 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
python-container
```

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
python-3.9.5-alpine-container
```

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.

Wrapper Scripts

Singularity HPC allows for global definition of wrapper scripts, meaning that instead of writing a module alias to run a container for some given alias, we generate a wrapper script of the same name instead. Since the settings.yml is global, all wrapper scripts defined here are specific to replacing aliases. Container-specific scripts you'll want to include in the container.yaml are described in *Developer Guide*. Let's take a look at the settings:

```
wrapper_scripts:

  # Enable wrapper scripts, period. If enabled, generate scripts for aliases instead␣
↪of commands
  # if enabled, we also allow container-specific wrapper scripts.
  enabled: false

  # use for docker aliases
  docker: docker.sh

  # use for podman aliases
  podman: docker.sh

  # use for singularity aliases
  singularity: singularity.sh
```

Since these are nested values, to get the current value you can use a `:` to separate the fields, e.g.,:

```
$ shpc config get wrapper_scripts:enabled
wrapper_scripts:enabled        False
```

And if you want to change the default, just add another level:

```
$ shpc config set wrapper_scripts:enabled true
Updated wrapper_scripts:enabled to be true
```

And don't forget you can manually update the file in an editor:

```
$ shpc config edit
```

Since different container technologies might expose different environment variables (e.g., `SINGULARITY_OPTS` vs `PODMAN_OPTS`) they are organized above based on the container technology. If you want to customize the wrapper script, simply replace the relative paths above (e.g., `singularity.sh`) with an absolute path to a file that will be used instead. For global alias scripts such as these, Singularity HPC will look for:

1. An absolute path first, if found is used first.
2. Then a script name in the shpc/main/wrappers directory

Here is an example of using wrapper scripts for the "python" container, which doesn't have container specific wrappers. What you see is the one entrypoint, *python*, being placed in a "bin" subdirectory that the module will see instead of defining the alias.

```
modules/python/
└── 3.9.10
    ├── 99-shpc.sh
    ├── bin
    │   └── python
    └── module.lua
```

For container specific scripts, you can add sections to a `container.yaml` to specify the script (and container type) and the scripts must be provided alongside the container.yaml to install.

```
docker_scripts:
  fork: docker_fork.sh
singularity_scripts:
  fork: singularity_fork.sh
```

The above says "given generation of a docker or podman container, write a script named "fork" that uses "docker_fork.sh" as a template" and the same for Singularity. And then I (the developer) would provide the custom scripts alongside container.yaml:

```
registry/vanessa/salad/
├── container.yaml
├── docker_fork.sh
└── singularity_fork.sh
```

And here is what those scripts look like installed. Since we are installing for just one container technology, we are seeing the alias wrapper for salad as "salad" and the container-specific wrapper for fork as "fork."

```
modules/vanessa/salad/
└── latest
    ├── 99-shpc.sh
    ├── bin
    │   ├── fork
    │   └── salad
    └── module.lua
```

We currently don't have a global argument to enable alias wrappers but not container wrappers. If you see a need for this please let us know.

Where are wrapper scripts stored?

Since we don't allow overlap of the name of an alias wrapper script (e.g., `bin/python` as a wrapper to a python entrypoint) from a custom container wrapper script (e.g., a wrapper script with name "python" under a container.yaml) we can keep them both in the modules directory. If you see a need to put them elsewhere please let us know.

Views

A view is a custom splicing of a set of installed modules that are intended to be used together, or loaded with other system modules. The concept is similar to a database in that you can only include in the view what you have in your shpc install, and the views themselves are done via symlinks to not redundantly store containers. If you want to generate a separate, non-symlink view, the suggested approach is to simply use a different shpc install.

Views Base

By default, your modules are installed to your `module_base` described in settings with a complete namespace, meaning the full name of the container registry from where they arise. We do this so that the namespace is consistent and there are no conflicts. However, for views we use a simplified tree to install from, meaning the module full names are _just_ the final container name. As an example, `ghcr.io/autamus/clingo` in a view would simply install to `clingo`.

Views are installed to the `views_base` in your settings, which defaults to `$root_dir/views`. To create a new named view:

Creating a New View

To create a new view, you just need to provide a name to `shpc view create`:

```
$ shpc view create mpi
View mpi was created in /home/vanessa/Desktop/Code/shpc/views/mpi
```

The above would be an example to create a new named "mpi," perhaps for a specific kind of mpi container to be installed there. Since it will be under the same directory, you'll be able to use this custom set of modules together. You can also create a view from an existing view.yaml file, perhaps one of your own existint views or one that has been shared with you!

```
$ shpc view create second-mpi views/mpi/view.yaml
Creating link $module_base/ghcr.io/autamus/clingo/5.5.1/module.lua -> $views_base/
↪second-mpi/clingo/5.5.1.lua
Module ghcr.io/autamus/emacs:27.2 was created.
Creating link $module_base/ghcr.io/autamus/emacs/27.2/module.lua -> $views_base/
↪second-mpi/emacs/27.2.lua
```

Loading a View

When you are ready to use your view, the "get" command returns the path:

```
$ shpc view get mpi
/home/vanessa/Desktop/Code/shpc/views/mpi
```

So you will be able to load as follows:

```
$ module use $(shpc view get mpi)
```

Installing Modules to a View

Installing a module means generating a symlink for a module to your view, and with a shortened name. We do this assuming that views are always smaller versions of the entire module tree, and that we want them to be easier to interact with (e.g., shorter names). To make interactions as easy as possible, if you install a module to your view that does not exist in the main shpc tree, it will be installed there first and linked. When you ask to install a module, always refer to the full name:

```
# install to the mpi view the module "ghcr.io/autamus/clingo"
$ shpc view install mpi ghcr.io/autamus/clingo
Module ghcr.io/autamus/clingo:5.5.1 was created.
Creating link $module_base/ghcr.io/autamus/clingo/5.5.1/module.lua -> $views_base/mpi/
↪clingo/5.5.1.lua
```

This will create symlinks to your previously installed modules in the view:

```
$ tree views
views/
└── mpi
    ├── clingo
    │   └── 5.5.1.lua -> /home/vanessa/Desktop/Code/shpc/modules/ghcr.io/autamus/
↪clingo/5.5.1/module.lua
    └── view.yaml
```

Since we are linking the same file, the same containers will be shared.

Always Install to a View

If you always want to install to an (existing) named view, simply set the `default_view` to a name:

```
$ shpc config set default_view mpi
```

You should obviously create the view first or you'll get an error message that it does not exist! When you have a default view set, any install that you do will install to the module base and also your view.

```
$ shpc install ghcr.io/autamus/emacs
...
Module ghcr.io/autamus/emacs:27.2 was created.
Creating link $module_base/ghcr.io/autamus/emacs/27.2/module.lua -> $views_base/mpi/
↪emacs/27.2.lua
```

And we can confirm it was created!

```
$ tree views/mpi
views/mpi/
├── clingo
│   └── 5.5.1.lua -> /home/vanessa/Desktop/Code/shpc/modules/ghcr.io/autamus/clingo/5.
↪5.1/module.lua
├── emacs
│   └── 27.2.lua -> /home/vanessa/Desktop/Code/shpc/modules/ghcr.io/autamus/emacs/27.
↪2/module.lua
└── view.yaml
```

The above can be useful for a permanent view you want to install everything to, or if you want to enable a view for a short period of time to install to it. If you want to disable this, then just do:

```
$ shpc config set default_view null
```

And note you can also ask to install to a view "one off":

```
$ shpc install --view mpi ghcr.io/autamus/emacs
```

List Views

If you want to list the views, just do:

```
$ shpc view list
               mpi
        second-mpi
```

In the example above you have two views, mpi and second-mpi, and each has it's own tree in views:

```
views/
├── mpi
|   ...
|   └── view.yaml
└── second-mpi
    ...
    └── view.yaml
```

List Modules Installed to a View

Listing modules installed to a view looks like the following:

```
$ shpc view list mpi
    ghcr.io/autamus/emacs:27.2
```

This is read directly from the view.yaml file.

Edit a View

While this isn't yet going to be useful (since we don't have additional modules to load) you can technically edit a view as follows:

```
$ shpc view edit mpi
```

This might be just an easy way to view it for the time being!

Add System Modules to a View

Views have support for customization, such as a system module that you always want loaded. We do this by way of an extra view_module that is generated in the root of the view (and always attempted to be loaded) by the installed modules. For example, let's say that when we load a view module named mpi, we always want to load a system module named "openmpi" and "mymod." We could do:

```
$ shpc view add <view> system_modules <name1> <name2>
$ shpc view add mpi system_modules openmpi mymod
Wrote updated .view_module: /home/vanessa/Desktop/Code/shpc/views/mpi/.view_module
```

The add command always requires a named view attribute (e.g.,``system_modules`` is a list) and then one or more values to add to it. This will write the view module to your view, and the module file symlinked should always attempt to try loading it. Note that if you are using modules version earlier than 4.8 the `try-load` command is not available so you will not have support for view customizations.

Remove System Modules from A View

Of course an "add" command would not be complete without a "remove" command! To remove modules:

```
$ shpc view remove mpi system_modules mymod
Wrote updated .view_module: /home/vanessa/Desktop/Code/shpc/views/mpi/.view_module
```

Note that if you edit the files manually, you would need to edit the view.yaml AND the hidden .view_module that is always updated from it.

Add and Remove Depends On Modules to a View

You can add (or remove) a `depends_on` clause to a view, just like with system modules. The syntax is the same, however you specify a different key to add to:

```
$ shpc view add <view> depends_on <name1> <name2>
$ shpc view add mpi depends_on openmpi
$ shpc view remove mpi depends_on openmpi
```

When you add a `depends_on` or `system_modules` to a view, what we are doing under the hood is adding a `.view_module` that will be loaded with the view, and it includes these extra parameters.

```
views/
└── mpi
    ├── python
    ├── view.yaml
    ├── .view_module
    └── 3.11-rc.lua -> /home/vanessa/Desktop/Code/shpc/modules/python/3.11-rc/module.lua
```

Here are example contents of `.view_module` (this will vary depending on your module software):

```
module load("myextraprogram")
depends_on("openmpi")
```

If you want any extra features added to this custom file (e.g., to support loading in a view) please open an issue for discussion.

Delete a View

If you want to nuke a view, just ask for it to be deleted.

```
$ shpc view delete mpi
```

By default you'll be asked for a confirmation. To force deletion:

```
$ shpc view delete mpi --force
```

Uninstall from a View

Uninstalling from a view is simply removing the symbolic link for a module, and it does not influence your module tree. You can uninstall either a specific symlinked version:

```
$ shpc view uninstall mpi ghcr.io/autamus/emacs:27.2
```

Or the entire tree of symlinks (e.g., all versions of emacs that are symlinked):

```
$ shpc view uninstall mpi ghcr.io/autamus/emacs
```

If you look in the view.yaml, it will be updated with what you install or uninstall. We do this so you can share the file with a collaborator and then can regenerate the view, discussed next.

Using a View

You can easily use a view as follows:

```
$ module use $(shpc view get mpi)
$ module load clingo/5.5.1
```

This is much more efficient compared to the install that uses the full paths:

```
$ module use ./modules
$ module load ghcr.io/autamus/clingo/5.5.1/module
```

Since we install based on the container name *and* version tag, this even gives you the ability to install versions from different container bases in the same root. If there is a conflict, you will be given the option to exit (and abort) or continue.

> **Warning:** Be cautious about creating symlinks in containers or other contexts where a bind could eliminate the symlink or make the path non-existent.

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 will first look at the environment variables `$EDITOR` and `$VISUAL` and will fall back to the `config_editor` in your user settings (vim by default).

Show

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
```

Set a limit of results with *–limit*:

```
$ shpc show --filter bio --limit 5
```

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

```
$ shpc show python
```

Finally, to show recipes in a local filesystem registry (that may not be added to your shpc config) you can specify the path with `--registry`. All of the above should work except with this argument, e.g.,:

```
$ shpc show --registry .
```

Install

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

$ tree containers/
containers/
└── python
    └── 3.9.2
        └── python-3.9.2.sif
```

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.

Install Private Images

What about private containers on Docker Hub? If you have a private image, you can simply use Singularity remote login before attempting the install and everything should work.

Install Local Image

The concept of installing a local image means that you are selecting a container.yaml recipe from an existing registry, however instead of pulling it, you are pairing it was a particular URI of a local image. As an example, let's say we have pulled a local samtools container:

```
$ singularity pull docker://quay.io/biocontainers/samtools:1.10--h2e538c0_3
```

We might then want to install it to the samtools namespace and using the same metadata (e.g., aliases, environment, etc.):

```
$ shpc install quay.io/biocontainers/samtools:1.10--h2e538c0_3 samtools_1.2--0.sif
```

This is similar to an `shpc add`, however instead of needing to write a container.yaml in a local filesystem, you are using an existing one. The use case or assumption here is that you have a local directory of containers that can be matched to existing shpc recipes. Finally to request using the container path "as is" without copying anything into your container folder, add `--keep-path`:

This feature is supported for shpc versions 0.1.15 and up.

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
- remove
- 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
```

Update

As of version 0.0.52, you can request on demand updates of container.yaml recipes, where an update means we ping the registry or resource for the module and find updated tags. An update generally means that:

- We start with the 50 latest tags of the container, as determined by crane.ggcr.dev
- We filter according to any recipe `filters` in the container.yaml
- Given a convention of including a hash, we try to remove it and generate a loose version
- Any versions (including latest) that cannot be sorted based on some semblance to a version are filtered out
- We sort the list, and given duplicates of some major minor (ignoring the last part of): `<major>.<minor>.<ignored>` we take the first seen in the sorted list.
- Then we take the top 5 newest to add.
- We then filter down to not include any versions older than the current oldest in the container.yaml

This action is run automatically on CI for you, however it's just done once a month and you are welcome to run it on your own, and contribute changes to container.yaml files that you think are meaningful. To update one container module recipe in the registry:

```
$ shpc update quay.io/biocontainers/samtools
Looking for updated digests for quay.io/biocontainers/samtools
>> quay.io/biocontainers/samtools
>> Latest
1.15--h3843a85_
 ↪0:sha256:d68e1b5f504dc60eb9f2a02eecbac44a63f144e7d455b3fb1a25323c667ca4c4
>> Tags
+ 1.9--h8571acd_
 ↪11:sha256:3883c91317e7b6b62e31c82e2cef3cc1f3a9862633a13f850a944e828dd165ec
+ 1.8--h46bd0b3_
 ↪5:sha256:e495550231927c4b9b23a9f5920906f608129bf470dc3409ef7c6eecf0fa6d8e
+ 1.7--2:sha256:9b3e923c44aa401e3e2b3bff825d36c9b07e97ba855ca04a368bf7b32f28aa97
+ 1.6--he673b24_
 ↪3:sha256:42031f060cde796279c09e6328d72bbce70d83a8f96e161faee3380ab689246d
+ 1.5--2:sha256:9a2f99c26cee798e3b799447a7cfa0fbb0c1ce27c42eef7a3c1289ba871f55cb
1.12--h9aed4be_
 ↪1:sha256:5fd5f0937adf8a24b5bf7655110e501df78ae51588547c8617f17c3291a723e1
1.15--h3843a85_
 ↪0:sha256:d68e1b5f504dc60eb9f2a02eecbac44a63f144e7d455b3fb1a25323c667ca4c4
1.10--h2e538c0_
 ↪3:sha256:84a8d0c0acec87448a47cefa60c4f4a545887239fcd7984a58b48e7a6ac86390
1.14--hb421002_
 ↪0:sha256:88632c41eba8b94b7a2a1013f422aecf478a0cb278740bcc3a38058c903d61ad
1.13--h8c37831_
 ↪0:sha256:04da5297386dfae2458a93613a8c60216d158ee7cb9f96188dad71c1952f7f72
1.11--h6270b1f_
 ↪0:sha256:141120f19f849b79e05ae2fac981383988445c373b8b5db7f3dd221179af382b
```

or to ask for a dry run, meaning we check for updates but don't perform them.

```
$ shpc update quay.io/biocontainers/samtools --dry-run
```

If you want to look for a specific string or pattern in the tags, just add `--filter`

```
$ shpc update redis --dry-run --filter alpine
```

Since no tags are deleted, this will add the latest set found with the term "alpine." You can also use this strategy to add a specific tag:

```
$ shpc update redis --dry-run --filter 6.0-rc-alpine
```

The current implementation just supports updating from a Docker / oras registry (others can come after if requested). As of version 0.0.58, there is support to ask to update all recipes - just leave out the name!

```
$ shpc update
```

If you are using an earlier release than 0.0.58 you can accomplish the same as follows:

```
$ for name in $(shpc show); do
    shpc update ${name} --dry-run
  done
```

Let us know if there are other features you'd like for update! For specific recipes it could be that a different method of choosing or sorting tags (beyond the defaults mentioned above and filter) is needed.

Sync Registry

A sync is when we take your local filesystem registry, and retrieve updates from the remote defined at `sync_registry` in your settings.yaml. Since sync will be writing recipes to the filesystem it only works if you target a filesystem registry (meaning that the default registry as a remote will not work).

Note: By default, the first filesystem registry found in your settings under the registry list will be used. To provide a one off registry folder (that should exist but does not need to be in your defined list) you can use `--registry`.

As an example, if we do this without changing the defaults:

```
$ shpc sync-registry
This command is only supported for a filesystem registry! Add one or use --registry.
```

We can then make a dummy directory to support sync. You could also make this directory and add to your settings proper under `registry`.

```
$ mkdir -p ./registry
$ shpc sync-registry --registry ./registry
```

Will compare your main registry folder against the main branch and only add new recipes that you do not have. To ask to update from a specific reference (tag or branch):

```
$ shpc sync-registry --registry ./registry --tag 0.0.58
```

You can also ask to add just a specific container:

```
$ shpc sync-registry --registry ./registry quay.io/not-local/container
```

You can also ask to add new containers and completely update container.yaml files.

```
$ shpc sync-registry --registry ./registry --all
```

This means we do a side by side comparison of your filesystem registry and the upstream, and we add new recipes folders that you don't have, and we replace any upstream files into recipes that you do have. Be careful with this

option, as if you've made changes to a container.yaml or associated file in the upstream they will be lost. For this reason, we always recommend that you do a dry run first:

```
$ shpc sync-registry --registry ./registry --dry-run
```

Finally, if you have a more complex configuration that you want to automate, you can provide a yaml file with your specifications:

```
sync_registry:
  "/tmp/github-shpc": "https://github.com/singularityhub/shpc-registry"
  "/tmp/gitlab-shpc": "https://gitlab.com/singularityhub/shpc-registry"
```

The above says to sync each respective local filesystem registry (key) with the remote (value). And then do:

```
$ shpc sync-registry --config-file registries.yaml
```

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"}"
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`. As of version 0.1.18, you can also ask to uninstall all:

```
$ shpc uninstall --all --force
```

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:
- 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 list`. The return value will be 0 is there are no updates, and 1 otherwise. This is how we check for new recipes to test.

```
$ for name in $(shpc list); do
    shpc check $name
 done
 tag 3.1.1 is up to date.
 tag 3.9.10 is up to date.
 tag latest is up to date.
 tag 1.14 is up to date.
 tag 5.5.1 is up to date.
 tag 1.54.0 is up to date.
```

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). You might also have a container on Docker Hub that you want to contribute to the registry! shpc does support the "add" command to perform both of these functions. The steps for adding a container are:

1. Running `shpc add` to create a container.yaml in the registry namespace
2. Customizing the container.yaml to your liking
3. Running `shpc install` to formally install your new container.

In the case of a docker image that is public (that you can share) you are encouraged to contribute your recipe directly to shpc for others to use, and once in the repository tags will also get updated automatically.

Warning: The add command only works for a local filesystem registry. This means it will not work with the default settings that retrieve recipes from a remote registry! To use add and create your own filesystem folder, you can use `--registry` with a newly created directory (that you can then add to your settings.yaml registry list).

Creating a Local Registry

For any of the commands below you can create a local registry very easily - it's just a directory!

```
$ mkdir -p registry
```

And then use it via a one off command to add, e.g.,:

```
$ shpc add --registry ./registry docker://vanessa/pokemon
```

Add a Local Container

As an example, let's start with the container `salad_latest.sif`. We have it on our local machine and cannot pull it from a registry. First, let's run `shpc add` and tell shpc that we want it under the `dinosaur/salad` namespace.

```
$ shpc add salad_latest.sif dinosaur/salad:latest
Registry entry dinosaur/salad:latest was added! Before shpc install, edit:
/home/vanessa/Desktop/Code/shpc/registry/dinosaur/salad/container.yaml
```

At this point, you should open up the container.yaml generated and edit to your liking. This usually means updating the description, maintainer, aliases, and possibly providing a url to find more information or support. Also notice we've provided the tag to be latest. If you update this registry entry in the future with a new version, you'll want to provide a new tag. If you provide an existing tag, you'll be asked to confirm before continuing. When you are happy, it's time to install it, just as you would a regular container!

```
$ shpc install dinosaur/salad:latest
```

And this will generate the expected module and container in your respective directory bases:

```
$ tree modules/dinosaur/salad/
modules/dinosaur/salad/
└── latest
    ├── 99-shpc.sh
    └── module.lua

1 directory, 2 files

$ tree containers/dinosaur/salad/
containers/dinosaur/salad/
└── latest
    └── sha256:77c7326e74d0e8b46d4e50d99e848fc950ed047babd60203e17449f5df8f39d4.sif

1 directory, 1 file
```

Add a Registry Container

Let's say we want to generate a container.yaml recipe for a container on Docker Hub. Let's say we want to add vanessa/pokemon. First, let's run `shpc add`. Note that we provide the `docker://` unique resource identifier to tell shpc it's from a Docker (OCI) registry.

```
$ shpc add docker://vanessa/pokemon
Registry entry vanessa/pokemon:latest was added! Before shpc install, edit:
/home/vanessa/Desktop/Code/shpc/registry/vanessa/pokemon/container.yaml
```

And that's it! The container module will use the same namespace, `vanessa/pokemon` as the Docker image, and we do this purposefully as a design decision. Note that `add` previously would add the container directly to the module directory, and as of version 0.0.49 it's been updated to generate the container.yaml first.

Remove

As of version `0.1.17` you can easily remove a container.yaml entry too! This remove command takes a pattern, and not providing one will remove all entries from the registry (useful if you want to create a new one but preserve the automation). Here is how to remove a specific namespace of container yamls:

```
$ shpc remove quay.io/biocontainers
Searching for container.yaml matching quay.io/biocontainers to remove...
Are you sure you want to remove 8367 container.yaml recipes? (yes/no)?
```

To remove all modules:

```
$ shpc remove
Searching for container yaml to remove...
```

(continues on next page)

(continued from previous page)

```
Are you sure you want to remove 264 container.yaml recipes? (yes/no)? yes
Removal complete!
```

This command can be useful if you want to start with a populated registry as a template for your own registry.

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/containers/vanessa/salad/latest/vanessa-
↪salad-latest-
↪sha256:8794086402ff9ff9f16c6facb93213bf0b01f1e61adf26fa394b78587be5e5a8.sif

$ shpc get tensorflow/tensorflow:2.2.2
/home/vanessa/Desktop/Code/singularity-hpc/containers/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.

Environment

After installing shpc to a local environment, you can use pre-commit to help with linting and formatting. To do that:

```
$ pip install -r .github/dev-requirements.txt
```

Then to run:

```
$ pre-commit run --all-files
```

You can also install as a hook:

```
$ pre-commit install
```

Developer Commands

Singularity Registry HPC has a few "developer specific" commands that likely will only be used in automation, but are provided here for the interested reader.

Docgen

To generate documentation for a registry (e.g., see this registry example we can use docgen. Docgen, by way of needing to interact with the local filesystem, currently only supports generation for a filesystem registry. E.g., here is how to generate a registry module (from a local container.yaml) that ultimately will be found in GitHub pages:

```
$ shpc docgen --registry . --registry-url https://github.com/singularityhub/shpc-
↪registry python
```

And you could easily pipe this to a file. Here is how we generate this programmatically in a loop:

```
for module in $(shpc show --registry ../shpc-registry); do
    flatname=${module#/}
    name=$(echo ${flatname//\//-})
    echo "Generating docs for $module, _library/$name.md"
    shpc docgen --registry ../shpc-registry --registry-url https://github.com/
↪singularityhub/shpc-registry $module > "_library/${name}.md"
done
```

Creating a FileSystem Registry

A filesystem 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'
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.

Creating a Remote Registry

If you want to create your own remote registry (currently supported to be on GitHub or GitLab) the easiest thing to do is start with one of our shpc provided registries as a template:

- **GitHub**
- **GitLab**

This means (for either) you'll want to clone the original repository:

```
$ git clone https://github.com/singularityhub/shpc-registry my-registry
$ cd my-registry
```

Ensure you do a fetch to get the github pages branch, which deploys the web interface!

```
$ git fetch
```

At this point, you can create an empty repository to push to. If you don't mind it being a fork, you can also just fork the original repository (and then pull from it instead). GitLab has a feature to fork and then remove the fork, so that is an option too. Ensure that you push the gh-pages branch too (for GitHub only):

```
$ git checkout gh-pages
$ git push origin gh-pages
```

Once you have your cloned registry repository, it's up to you for how you want to delete / edit / add containers! You'll likely use `shpc add` to generate new configs, and you might want to delete most of the default containers provided. Importantly, you should take note of the workflows in the repository. Generally:

- We have an update workflow (GitHub) that will check for new versions of containers. This still need to be ported to GitLab.
- The docs workflow (on GitHub, this is in the .github-ci.yaml) will deploy docs to GitHub/GitLab pages.

For each of GitLab and GitHub, ensure after you deploy that your pages are enabled. It helps to ensure the website (static) URL is in the description to be easily find-able. Once it's deployed, ensure you see your containers, and clicking the </> (code) icon shows the library.json that shpc will use. Finally, akin to adding a filesystem registry, you can just do the same, but specify your remote URL:

```
$ shpc config add registry https://github.com/singularityhub/shpc-registry
```

And that's it!

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 and containers directories, where each is kept in a separate directory based on version.

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

$ tree containers/
containers/
└── python
    └── 3.9.2
        └── python-3.9.2.sif
```

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
  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.

Overrides

It might be the case that as your containers change over time, the set of any of:

- commands (aliases)
- docker_script
- singularity_script
- environment (env)
- features
- description

does too! Or it be the case that you have hundreds of aliases, and want to better organize them separately from the container.yaml. To support this, shpc (as of version 0.0.56) has support for an `overrides` section in the container.yaml, meaning that you can define pairs of container tags and relative path lookups to external files with any of the stated sections. A simple example might look like this:

```
docker: python
url: https://hub.docker.com/_/python
maintainer: '@vsoch'
description: An interpreted, high-level and general-purpose programming language.
latest:
  3.9.5-alpine:␣
↪sha256:f189f7366b0d381bf6186b2a2c3d37f143c587e0da2e8dcc21a732bddf4e6f7b
tags:
  3.9.2-alpine:␣
↪sha256:f046c06388c0721961fe5c9b6184d2f8aeb7eb01b39601babab06cfd975dae01
overrides:
  3.9.2-alpine: aliases/3.9.2-alpine.yaml
aliases:
    python: /usr/local/bin/python
```

Since this file only has aliases, we chose to use a subdirectory called "aliases" to make that clear, however the file can have any of the fields mentioned above, and can be organized in any relative path to the container directory that you deem appropriate. Here is what this corresponding file with relative path `aliases/3.9.2-alpine.yaml` might look like this:

```
aliases:
  python: /alias/path/to/python
```

Finally, for all fields mentioned above, the format is expected to follow the same convention as above (and it will be validated again on update).

Wrapper Script

Singularity HPC allows exposure of two kinds of wrapper scripts:

1. A global level wrapper intended to replace aliases. E.g., if an alias "samtools" is typically a direct container call, enabling a wrapper will generate an executable script "samtools" in a "bin" directory associated with the container, added to the path, to call instead. This is desired when MPI ("mpirun") or scheduler (e.g. "srun" with Slurm) utilities are needed to run the scripts. This global script is defined in settings.yml and described in the user guide.
2. A container level wrapper that is specific to a container, described here.

For container specific scripts, you can add sections to a `container.yaml` to specify the script (and container type) and the scripts must be provided alongside the container.yaml to install.

```
docker_scripts:
  fork: docker_fork.sh
singularity_scripts:
  fork: singularity_fork.sh
```

The above says "given generation of a docker or podman container, write a script named "fork" that uses "docker_fork.sh" as a template" and the same for Singularity. And then I (the developer) would provide the custom scripts alongside container.yaml:

```
registry/vanessa/salad/
├── container.yaml
├── docker_fork.sh
└── singularity_fork.sh
```

You can look at `registry/vanessa/salad` for an example that includes Singularity and Docker wrapper scripts. For example, when generating for a singularity container with the global wrapped scripts enabled, we get one wrapper script for the alias "salad" and one for the custom container script "fork":

```
$ tree modules/vanessa/salad/
modules/vanessa/salad/
└── latest
    ├── 99-shpc.sh
    ├── bin
    │   ├── fork
    │   └── salad
    └── module.lua
```

If we disable all wrapper scripts, the bin directory would not exist. If we set the default wrapper scripts for singularity and docker in settings.yml and left enable to true, we would only see "fork."

How to write an alias wrapper script

First, decide if you want a global script (to replace or wrap aliases) OR a custom container script. For an alias derived (global) script, you should:

1. Write the new script file into shpc/main/wrappers.
2. Add an entry to shpc/main/wrappers/scripts referencing the script.

For these global scripts, the user can select to use it in their settings.yaml. We will eventually write a command to list global wrappers available, so if you add a new one future users will know about it. For alias wrapper scripts, the following variables are passed for rendering:

Table 3: Wrapper Script Variables

Name	Type	Description	Example
alias	dictionary	The entire alias in question, including subfields name, command, singularity_options or docker_options, singularity_script or docker_script, and args	`{{ alias.name }}`
settings	dictionary	Everything referenced in the user settings	`{{ settings.wrapper_shell }}`
container	dictionary	The container technology	`{{ container.command }}` renders to docker, singularity, or podman
config	dictionary	The entire container config (container.yaml) structured the same	`{{ config.docker }}`
image	string	The name of the container binary (SIF) or unique resource identifier	`{{ image }}`
module_dir	string	The name of the module directory	`{{ module_dir }}`
features	dictionary	A dictionary of parsed features	`{{ features.gpu }}`

How to write an container wrapper script

If you want to write a custom container.yaml script:

1. Add either (or both) of singularity_scripts/docker_scripts in the container.yaml, including an alias command and an associated script.
2. Write the script with the associated name into that folder.

For rendering, the same variables as for alias wrapper scripts are passed, **except** `alias` which is now a *string* (the name of the alias defined under singularity_scripts or docker_scripts) and should be used directly, e.g. `{{ alias }}`.

Templating for both wrapper script types

Note that you are free to use "snippets" and "bases" either as an inclusion or "extends" meaning you can easily re-use code. For example, if we have the following registered directories under `shpc/main/wrappers/templates` for definition of bases and templates:

```
main/wrappers/templates/

# These are intended for use with "extends"
├── bases
│   ├── __init__.py
│   └── shell-script-base.sh

# These are top level template files, as specified in the settings.yml
├── docker.sh
├── singularity.sh
```

(continues on next page)

(continued from previous page)

```
# A mostly empty directory ready for any snippets!
└── snippets
```

For example, a "bases" template to define a shell and some special command that might look like this:

```
#!{{ settings.wrapper_shell }}

script=`realpath $0`
wrapper_bin=`dirname $script`
{% if '/csh' in settings.wrapper_shell %}set moduleDir=`dirname $wrapper_bin`{% else
↪%}export moduleDir=$(dirname $wrapper_bin){% endif %}

{% block content %}{% endblock %}
```

And then to use it for any container- or global- wrapper we would do the following in the wrapper script:

```
{% extends "bases/my-base-shell.sh" %}

# some custom wrapper before stuff here

{% block content %}{% endblock %}

# some custom wrapper after stuff here
```

For snippets, which are intended to be more chunks of code you can throw in one spot on the fly, you can do this:

```
{% include "snippets/export-envars.sh" %}
# some custom wrapper after stuff here
```

Finally, if you want to add your own custom templates directory for which you can refer to templates relatively, define `wrapper_scripts` -> `templates` as a full path in your settings.

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 container. 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
```

Oras

As of version 0.0.39 Singularity Registry HPC has support for oras, meaning we can use the Singularity client to pull an oras endpoint. Instead of using `docker:` in the recipe, the container.yaml might look like this:

```
oras: ghcr.io/singularityhub/github-ci
url: https://github.com/singularityhub/github-ci/pkgs/container/github-ci
maintainer: '@vsoch'
description: An example SIF on GitHub packages to pull with oras
latest:
  latest: sha256:227a917e9ce3a6e1a3727522361865ca92f3147fd202fa1b2e6a7a8220d510b7
tags:
  latest: sha256:227a917e9ce3a6e1a3727522361865ca92f3147fd202fa1b2e6a7a8220d510b7
```

And then given the `container.yaml` file located in `registry/ghcr.io/singularityhub/github-ci/` you would install with shpc and the Singularity container backend as follows:

```
$ shpc install ghcr.io/singularityhub/github-ci
```

Important: You should retrieve the image sha from the container registry and not from the container on your computer, as the two will often be different depending on metadata added.

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 4: Registry YAML Fields

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) as described above	false
overrides	Key value pairs to override container.yaml defaults.	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
singularity_scripts	key value pairs of wrapper names (e.g., executable called by user) and local container script for Singularity	false
docker_scripts	key value pairs of wrapper names (e.g., executable called by user) and local container script for Docker or Singularity	false

A complete table of features is shown here. The

Fields include:

Table 5: Features

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

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

(continues on next page)

(continued from previous page)

```
Container:

 - /home/vanessa/Desktop/Code/singularity-hpc/containers/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 and containers folder, and of course use it.

GitHub Action

As of version `0.1.17` we provide a GitHub action that will allow you to update a registry from an container binary cache. Does any of this not make sense? Don't worry! We have a full tutorial below to walk you through this process. For now, here is how to use the action provided here alongside your remote registry (e.g., running in GitHub actions) to update from a container executable cache of interest. For the example here, we are updating the `singularityhub/shpc-registry` from binaries in the `singularityhub/shpc-registry-cache` that happens to contain over 8K BioContainers.

```yaml
name: Update BioContainers

on:
  pull_request: []
  schedule:
  - cron: 0 0 1 * *

jobs:
  auto-scan:
    runs-on: ubuntu-latest
    steps:
```

(continues on next page)

(continued from previous page)

```
    - name: Checkout
      uses: actions/checkout@v3
      with:
        fetch-depth: '0'

    - name: Create conda environment
      run: conda create --quiet -c conda-forge --name cache spython

    - name: Derive BioContainers List
      run: |
        export PATH="/usr/share/miniconda/bin:$PATH"
        source activate cache
        pip install -r .github/scripts/dev-requirements.txt
        python .github/scripts/get_biocontainers.py /tmp/biocontainers.txt
        head /tmp/biocontainers.txt

      # registry defaults to PWD, branch defaults to main
    - name: Update Biocontainers
      uses: singularityhub/singularity-hpc/actions/cache-update@main
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
        cache: https://github.com/singularityhub/shpc-registry-cache
        min-count-inclusion: 10
        max-count-inclusion: 1000
        additional-count-inclusion: 25
        # Defaults to shpc docs, this gets formatted to include the entry_name
        url_format_string: "https://biocontainers.pro/tools/%s"
        pull_request: "${{ github.event_name != 'pull_request' }}"
        namespace: quay.io/biocontainers
        listing: /tmp/biocontainers.txt
```

The listing we derive in the third step is entirely optional, however providing one will (in addition to updating from the cache) ensure that entries provided there are also added, albeit without aliases. The namespace is provided to supplement the listing. The reason we allow this additional listing is because the cache often misses being able to extract a listing of aliases for some container, and we still wait to add it to the registry (albeit without aliases).

Developer Tutorial

This is currently a small tutorial that will include some of the lessons above and show you how to:

1. Create a new remote registry on GitHub with automated updates
2. Create a new container executable cache
3. Automate updates of the cache to your registry

Preparing a Remote Registry

To start, create a new repository and follow the instructions in *Creating a Remote Registry* to create a remote registry. We will briefly show you the most basic clone and adding a few entries to it here.

```
# Clone the shpc-registry as a template
$ git clone https://github.com/singularityhub/shpc-registry /tmp/my-registry
$ cd /tmp/my-registry
```

The easiest way to delete the entries (to make way for your own) is to use shpc itself! Here is how we can use `shpc show` to remove the entries. First, make sure that shpc is installed (*Installation*) and ensure your registry is the only one in the config registry section. You can use `shpc config edit` to quickly see it. It should look like this:

```
# This is the default line you can comment out / remove
# registry: [https://github.com/singularityhub/shpc-registry]
# This is your new registry path, you'll need to add this.
# Please preserve the flat list format for the yaml loader
registry: [/tmp/my-registry]
```

After making the above change, exit and do a sanity check to make sure your active config is the one you think it is:

```
$ shpc config get registry
registry                        ['/tmp/my-registry']
```

Deleting Entries

If you want to start freshly, you can choose to delete all the existing entries (and this is optional, you can continue the tutorial without doing this!) To do this, use the `shpc remove` command, which will remove all registry entries. We recommend deleting quay.io first since most entries live there and it will speed up the subsequent operation.

```
$ rm -rf quay.io/biocontainers
$ shpc remove # answer yes to confirmation
```

If you do a git status after this, you'll see many entries removed. Save your changes with a commit.

```
$ git commit -a -s -m 'emptying template registry'
```

After this you will have only a skeleton set of files, and most importantly, the .github directory with automation workflows. Feel free to remove or edit files such as the `FUNDING.yml` and `ISSUE_TEMPLATE`.

Fetch GitHub Pages

Next, use "fetch" to get GitHub pages.

```
$ git fetch
```

At this point you can edit the `.git/config` to be your new remote.

```
# Update the remote to be your new repository
vim .git/config
```

As an example, here is a diff where I changed the original registry to a new one I created called *vsoch/test-registry*:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        # url = https://github.com/singularityhub/shpc-registry
        url = git@github.com:vsoch/test-registry
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
```

Note that in the above, we also change "https://" to be "git" to use a different protocol. You should only do this change after you've fetched, as you will no longer be connected to the original remote! Finally, you'll need to change the "baseurl" in *_config.yaml* to be the name of your GitHub repository:

```
`diff - baseurl:  "/shpc-registry" #important:  start with / + baseurl:  "/
my-shpc-registry" #important:  start with / `
```

If you forget this step, the pages will render, but the style sheets will be broken.

Push Branches to your New Remote

Note that we will want to push both main and GitHub pages branches. Now that you've changed the remote and commit, create the repository in GitHub, and push your changes and then push to your main branch. We do this push before gh-pages so "main" becomes the primary branch.

```
$ git push origin main
```

Then you can checkout the gh-pages branch to do the same cleanup and push. Here is the checkout:

```
$ git checkout gh-pages
```

And how to do the cleanup. This cleanup is easier - just delete the markdown files in `_library`.

```
$ rm -rf _library/*.md
```

And then commit and push to gh-pages.

```
$ git commit -a -s -m 'emptying template registry gh-pages'
$ git push origin gh-pages
```

Note that since the main branch will try to checkout gh-pages to generate the docs, the first documentation build might fail. Don't worry about this - the branch will exist the second time when you add recipes.

Manually Adding Registry Entries

Great! Now you have an empty registry on your filesystem that will be pushed to GitHub to serve as a remote. Make sure you are back on the main branch:

```
$ git checkout main
```

Let's now add some containers! There are two ways to go about this:

- Manually add a recipe locally, optionally adding discovered executables
- Use a GitHub action to do the same.

We will start with the manual approach. Here is how to add a container.yaml recipe file, without any customization for executable discovery:

```
$ shpc add docker://vanessa/salad:latest
Registry entry vanessa/salad was added! Before shpc install, edit:
/tmp/my-registry/vanessa/salad/container.yaml
```

You could then edit that file to your liking.

Like for `shpc update` *Update*, tags are automatically populated using crane.ggcr.dev, which only returns the 50 latest tags and obviously can only access public images. If you see a `crane digest` error instead of tags, you'll have to populate the tags yourself.

Executables are by default missing. If you want shpc to discover executables, you'll need to install guts:

```
pip install git+https://github.com/singularityhub/guts@main
```

And then use the provided script to generate the container.yaml (with executables discovered):

```
$ python .github/scripts/add_container.py --maintainer "@vsoch" --description "The␣
↪Vanessa Salad container" --url "https://github.com/vsoch/salad" docker://vanessa/
↪salad:latest
```

That will generate a container.yaml with executables discovered:

```
url: https://github.com/vsoch/salad
maintainer: '@vsoch'
description: The Vanessa Salad container
latest:
  latest: sha256:e8302da47e3200915c1d3a9406d9446f04da7244e4995b7135afd2b79d4f63db
tags:
  latest: sha256:e8302da47e3200915c1d3a9406d9446f04da7244e4995b7135afd2b79d4f63db
docker: vanessa/salad
aliases:
  salad: /code/salad
```

You can then push this to GitHub. If you are curious about how the docs are generated, you can try it locally:

```
$ git checkout gh-pages
$ ./generate.sh
Generating docs for vsoch/salad, _library/vsoch-salad.md
```

There is also an associated workflow to run the same on your behalf. Note that you'll need to:

1. Go to the `repository --> Settings --> Actions --> Workflow Permissions` and enable read and write.

2. Directly under that, check the box to allow actions to open pull requests for this to work.

If you get a message about push being denied to the bot, you forgot to do one of these steps! The workflow is under `Actions --> shpc new recipe manual --> Run Workflow`. Remember that any container, once it goes into the registry, will have tags and digests automatically updated via the "Update Containers" action workflow.

Creating a Cache

This is an advanced part of the developer tutorial! Let's say that you don't want to go through the above to manually run commands. Instead of manually adding entries in this manner, let's create an automated way to populate entries from a cache. You can read more about the algorithm we use to derive aliases in the shpc-registry-cache repository, along with cache generation details. You will primarily need two things:

1. A text listing of containers to add to the cache, ideally automatically generated
2. A workflow that uses it to update your cache.

Both of these files should be in a GitHub repository that you create. E.g.,:

```
containers.txt
.github/
└── workflows
    └── update-cache.yaml
```

For the main shpc registry cache linked above, we derive a list of biocontainers.txt on the fly from the current depot listing. You might do the same for a collection of interest, or just to try it out, create a small listing of your own containers in a `containers.txt` e.g.,:

```
python
rocker/r-ver
julia
```

You can find further dummy examples in the container-executable-discovery repository along with variables that the action accepts. As an example of our small text file above, we might have:

```yaml
name: Update Cache

on:
  workflow_dispatch:
  schedule:
  # Weekly, monday and thursday
  - cron: 0 0 * * 1,4

jobs:
  update-cache:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v3

    - name: Update Cache Action
      uses: singularityhub/container-executable-discovery@main
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
        repo-letter-prefix: true
        listing: ./containers.txt
        dry_run: ${{ github.event_name == 'pull_request' }}
```

And this would use out containers.txt listing to populate the cache in the repository we've created. Keep in mind that caches are useful beyond Singularity Registry HPC - knowing the paths and executables within a container is useful for other applied and research projects too!

Updating a Registry from a Cache

Once you have a cache, it's fairly easy to use another action provided by shpc directly from it. This is the *GitHub Action* mentioned above. The full example provided there does two things:

1. Updates your registry from the cache entries
2. Derives an additional listing to add containers that were missed in the cache.

And you will want to put the workflow alongside your newly created registry. The reason for the second point is that there are reasons we are unable to extract container binaries to the filesystem. In the case of any kind of failure, we might not have an entry in the cache, however we still want to add it to our registry! With the addition of the `listing` variable and the step to derive the listing of BioContainers in the example above, we are still able to add these missing containers, albeit without aliases. Here is an example just updating from the cache (no extra listing):

```yaml
name: Update BioContainers

on:
  pull_request: []
  schedule:
  - cron: 0 0 1 * *

jobs:
  auto-scan:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v3

      # registry defaults to PWD, branch defaults to main
    - name: Update Containers
      uses: singularityhub/singularity-hpc/actions/cache-update@main
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
        # Change this to your cache path
        cache: https://github.com/singularityhub/shpc-registry-cache
        min-count-inclusion: 10
        max-count-inclusion: 1000
        additional-count-inclusion: 25
        # Defaults to shpc docs, this gets formatted to include the entry_name
        url_format_string: "https://biocontainers.pro/tools/%s"
        pull_request: "${{ github.event_name != 'pull_request' }}"
```

The url format string expects a container identifier somewhere, and feel free to link to your registry base if you are unable to do this. You will want to change the `cache` to be your remove cache repository, and then adjust the parameters to your liking:

- **min-count-inclusion**: is the threshold count by which under we include ALL aliases. A rare alias is likely to appear fewer times across all containers.
- **additional-count-inclusion**: an additional number of containers to add after the initial set under `min-count-inclusion` is added (defaults to 25)
- **max-count-inclusion**: don't add counts over this threshold (set to 1000 for biocontainers).

Since the cache will generate a global counts.json and skips.json, this means the size of your cache can influence the aliases chosen. It's recommended to create your entire cache first and then to add it to your registry to update.

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
----------------------------------------------------------------------------------------
 ↪--------------------------------------
  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 /home/shpc-user/singularity-hpc/modules/
↪tensorflow/tensorflow/2.2.2/tensorflow-tensorflow-2.2.2-
↪sha256:e2cde2bb70055511521d995cba58a28561089dfc443895fd5c66e65bbf33bfc0.sif
    - python:
           singularity exec --nv /home/shpc-user/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
----------------------------------------------------------------------------------------
 ↪--------------------------------------
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 /home/shpc-user/singularity-hpc/modules/python/
↪3.9.2-slim/python-3.9.2-slim-
↪sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef.sif
    - python:
       singularity exec /home/shpc-user/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
```

If your workflow requires knowledge of the local path to the sif image, this information can be output by using the "container" suffixed alias:

```
$ python-container
/home/shpc-user/singularity-hpc/modules/python/3.9.2-slim/python-3.9.2-slim-
↪sha256:85ed629e6ff79d0bf796339ea188c863048e9aedbf7f946171266671ee5c04ef.sif
```

or equivalently by accessing the value of the **SINGULARITY_CONTAINER** environment variable (or **PODMAN_CONTAINER** for each of Podman and Docker).

Adding Options

By default, some of the commands will come with singularity options. For example, a container intended for gpu will have a features: gpu set to true, and this will add the `--nv` flag given that the user or cluster settings file has that feature enabled. 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` or `--nv`.

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 /home/shpc-user/singularity-hpc/modules/vanessa/
↪salad/latest/vanessa-salad-latest-
↪sha256:71d1f3e42c1ceee9c02295577c9c6dfba4f011d9b8bce82ebdbb6c187b784b35.sif
     - vanessa-salad-latest: singularity run /home/shpc-user/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.
            /\
           //\\
           // \\
         ^  \\ //  ^
```

(continues on next page)

(continued from previous page)

```
        / \  ) (  / \
        ) (  ) (  ) (
        \ \_/ /\ \_/ /
         \__ _)(_ __/
          \ \ / /
           ) \/ (
           | /\ |
           | )( |
           | )( |
           | \/ |
           )_____(
          /    \
          \_______/
```

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.**LogColors**
Bases: object

BOLD = '\x1b[1m'

ENDC = '\x1b[0m'

OKBLUE = '\x1b[94m'

OKCYAN = '\x1b[96m'

OKGREEN = '\x1b[92m'

PURPLE = '\x1b[95m'

RED = '\x1b[91m'

UNDERLINE = '\x1b[4m'

WARNING = '\x1b[93m'

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 log handler prints the output to the console. :param msg: the log message dictionary :type msg: dict

warning(*msg*)

yellow(*msg*)

shpc.logger.**add_prefix**(*msg*, *char='>>'*)
Add an "OKBLUE" prefix to a message

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

shpc.logger.**underline**(*msg*)
Return an underlined message

shpc.main module

shpc.main.**get_client**(*quiet=False*, ***kwargs*)
Get a singularity HPC client based on the backend (e.g., Lmod) and container technology (currently just Singularity) of interest.

Parameters **quiet** (*if True, suppress most output about the client (e.g. speak)*) –

shpc.main.container module

shpc.main.modules

shpc.main.modules.lmod

class shpc.main.modules.lmod.**Client**(***kwargs*)
Bases: shpc.main.modules.base.ModuleBase

shpc.main.modules.tcl

class shpc.main.modules.tcl.**Client**(***kwargs*)
Bases: shpc.main.modules.base.ModuleBase

PYTHON MODULE INDEX

S

INDEX

S

T

U

W

Y