# DockerSpawner

*Release 14.1.0.dev0*

**Jupyter Contributors**

**Apr 01, 2026**

# CONTENTS

The **dockerspawner** (also known as JupyterHub Docker Spawner), enables JupyterHub to spawn single user notebook servers in Docker containers.

There are three basic types of spawners available for dockerspawner:

- DockerSpawner: takes an authenticated user and spawns a notebook server in a Docker container for the user.
- SwarmSpawner: launches single user notebook servers as Docker Swarm mode services.
- SystemUserSpawner: spawns single user notebook servers that correspond to system users.

CHAPTER

# ONE

# CONTENTS

## 1.1 Installation Guide

### 1.1.1 Installation

#### DockerSpawner installation

1. JupyterHub 0.7 or above is required, which also means Python 3.3 or above.

```
$ python3 -m pip install jupyterhub
```

2. Install dockerspawner to the system:

```
$ python3 -m pip install dockerspawner
```

## 1.2 Choosing a spawner

### 1.2.1 Choosing a spawner

Three basic types of spawners are available for dockerspawner:

- *DockerSpawner*: useful if you would like to spawn single user notebook servers on the fly. It will take an authenticated user and spawn a notebook server in a Docker container for the user.
- *SwarmSpawner*: same behavior as DockerSpawner, but launches single user notebook servers as Docker Swarm mode services instead of as individual containers. This allows for running JupyerHub in a swarm so that notebook containers can be run on any of multiple servers.
- *SystemUserSpawner*: useful if you would like to spawn single user notebook servers that correspond to the system's users.

In most cases, we recommend using DockerSpawner. Use cases where you may wish to use SystemUserSpawner are:

- You are using docker just for environment management, but are running on a system where the users already have accounts and files they should be able to access from within the container. For example, you wish to use the system users and user home directories that already exist on a system.
- You are using an external service, such as nbgrader, that relies on distinct unix user ownership and permissions.

> **Note**
>
> If neither of those cases applies, DockerSpawner is probably the right choice.

### DockerSpawner

Tell JupyterHub to use DockerSpawner by adding the following line to your `jupyterhub_config.py`:

```
c.JupyterHub.spawner_class = 'dockerspawner.DockerSpawner'
```

There is a complete example in examples/oauth for using GitHub OAuth to authenticate users, and spawn containers with docker.

### SwarmSpawner

Tell JupyterHub to use SwarmSpawner by adding the following line to your `jupyterhub_config.py`:

```
c.JupyterHub.spawner_class = 'dockerspawner.SwarmSpawner'
```

You need to make sure that the JupyterHub process is launched on a Swarm manager node, since its node needs to have permission to launch new Swarm services. It also needs to have the docker socket mounted (like DockerSpawner) to communicate out of its own container with the host's docker server. You can accomplish this in your `docker-compose.yml` with the following settings:

```
jupyterhub:
  image: jupyterhub/jupyterhub
  # This is necessary to prevent the singleton hub from using its service number as its␣
↪hostname
  hostname: jupyterhub
  # Permit communication with the host's docker server
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
  # Ensure Hub and Notebook servers are on the same network
  networks:
    - jupyterhub_network
  environment:
    DOCKER_NETWORK_NAME: jupyterhub_network
  # Ensure that we execute on a Swarm manager
  deploy:
    replicas: 1
    placement:
      constraints:
        - node.role == manager
```

You'll also need to ensure that the JupyterHub service and the launched single-user services all run on the same Swarm overlay network. You can create one easily using:

```
$ docker network create --driver overlay jupyterhub_network
```

Then use this network in your jupyterhub_config.py like the following example:

```
network_name = os.environ['DOCKER_NETWORK_NAME']
c.SwarmSpawner.network_name = network_name
c.SwarmSpawner.extra_host_config = {'network_mode': network_name}
```

Unless otherwise noted, SwarmSpawner supports the same configuration options as DockerSpawner.

#### SystemUserSpawner

If you want to spawn notebook servers for users that correspond to system users, you can use the SystemUserSpawner instead. Add the following to your `jupyterhub_config.py`:

```
c.JupyterHub.spawner_class = 'dockerspawner.SystemUserSpawner'
```

The SystemUserSpawner will also need to know where the user home directories are on the host. By default, it expects them to be in `/home/<username>`, but if you want to change this, you'll need to further modify the `jupyterhub_config.py`. For example, the following will look for a user's home directory on the host system at `/volumes/user/<username>`:

```
c.SystemUserSpawner.host_homedir_format_string = '/volumes/user/{username}'
```

For a full example of how `SystemUserSpawner` is used, see the compmodels-jupyterhub repository (this additionally runs the JupyterHub server within a docker container, and authenticates users using GitHub OAuth).

#### Using Docker Swarm (not swarm mode!)

> **Note**
>
> This is the older Docker Swarm, which makes a swarm look like a single docker instance. For the newer Docker Swarm Mode, see *SwarmSpawner*. This used to be supported by cassinyio, but this repository has been deprecated.

Both `DockerSpawner` and `SystemUserSpawner` are compatible with Docker Swarm when multiple system nodes will be used in a cluster for JupyterHub. Simply add `0.0.0.0` to your `jupyterhub_config.py` file as the `host_ip`:

```
c.DockerSpawner.host_ip = "0.0.0.0"
```

This will configure DockerSpawner and SystemUserSpawner to get the container IP address and port number using the `docker port` command.

#### Using Podman

Podman is an alternative to Docker for running containers, and supports running containers without root access. It is not officially supported by JupyterHub, but it can be used with DockerSpawner by running a podman service:

```
podman system service --time=0 &
export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/podman/podman.sock
# Run jupyterhub as normal
jupyterhub --config=jupyterhub_config.py
```

There are several other ways of running the Podman service.

Not all Docker features are supported by Podman.

## 1.3 Data persistence

### 1.3.1 Data persistence

With `DockerSpawner`, the user's home directory is **not** persistent by default, so some configuration is required to do so unless the directory is to be used with temporary or demonstration JupyterHub deployments.

The simplest version of persistence to the host filesystem is to isolate users in the filesystem, but leave everything owned by the same 'actual' user with DockerSpawner. That is, using **docker mounts** to isolate user files, not ownership or permissions on the host.

### Volume mapping

Volume mapping for DockerSpawner in `jupyterhub_config.py` is required configuration for persistence. To map volumes from the host file/directory to the container (referred to as guest) file/directory mount point, set the `c.DockerSpawner.volumes` to specify the guest mount point (bind) for the volume.

If you use `{username}` in either the host or guest file/directory path, username substitution will be done and `{username}` will be replaced with the current user's name.

> **Note**
>
> The jupyter/docker-stacks notebook images run the Notebook server as user `jovyan` and set the user's notebook directory to `/home/jovyan/work`.

Example:

```
# Explicitly set notebook directory because we'll be mounting a host volume to
# it.  Most jupyter/docker-stacks *-notebook images run the Notebook server as
# user `jovyan`, and set the notebook directory to `/home/jovyan/work`.
# We follow the same convention.
notebook_dir = os.environ.get('DOCKER_NOTEBOOK_DIR') or '/home/jovyan/work'
c.DockerSpawner.notebook_dir = notebook_dir

# Mount the real user's Docker volume on the host to the notebook user's
# notebook directory in the container
c.DockerSpawner.volumes = { 'jupyterhub-user-{username}': notebook_dir }

# Mount a directory on the host to the notebook user's notebook directory in the container
c.DockerSpawner.mounts = [
  {'source': '/jupyterhub-user-{username}', 'target': notebook_dir, 'type': 'bind'}
]
```

### Memory limits

If you have `jupyterhub >= 0.7`, you can set a memory limit for each user's container easily.

```
c.Spawner.mem_limit = '2G'
```

The value can either be an integer (bytes) or a string with a 'K', 'M', 'G' or 'T' suffix.

### Resources

The `jupyterhub-deploy-docker` repo contains a reference deployment that persists the notebook directory; see its `jupyterhub_config.py` for an example configuration.

See Docker documentation on data volumes for more information on data persistence.

## 1.4 Picking or building a Docker image

### 1.4.1 Picking or building a Docker image

By default, DockerSpawner uses the `quay.io/jupyterhub/singleuser` image with the appropriate tag that pins the right version of JupyterHub, but you can also build your own image.

#### How to pick a Docker image

Any of the existing Jupyter docker stacks can be used with JupyterHub, provided that the version of JupyterHub in the image matches, and are encouraged as the image of choice. Make sure to pick a tag!

Example:

```
c.DockerSpawner.image = 'quay.io/jupyter/scipy-notebook:2023-10-23'
```

The docker-stacks are moving targets with always changing versions. Since you need to make sure that JupyterHub in the image is compatible with JupyterHub, always include the `:hash` tag part when specifying the image.

#### How to build your own Docker image

You can also build your own image. The only requirements for an image to be used with JupyterHub:

1. it has Python >= 3.6
2. it has JupyterHub version matching your Hub deployment
3. it has the Jupyter `notebook` package
4. CMD launches jupyterhub-singleuser, OR jupyter-labhub, OR the `c.Spawner.cmd` configuration is used to do this.

For just about any starting image, you can make it work with JupyterHub by installing the appropriate JupyterHub version and the Jupyter notebook package.

For instance, from the docker-stacks, pin your JupyterHub version and you are done:

```
FROM quay.io/jupyter/scipy-notebook:67b8fb91f950
ARG JUPYTERHUB_VERSION=4.0.2
RUN pip3 install --no-cache \
    jupyterhub==$JUPYTERHUB_VERSION
```

Or for the absolute minimal JupyterHub user image starting only from the base Python image:

**NOTE: make sure to pick the jupyterhub version you are using!**

```
FROM python:3.11
RUN pip3 install \
    'jupyterhub==4.*' \
    'notebook==7.*'

# create a user, since we don't want to run as root
RUN useradd -m jovyan
ENV HOME=/home/jovyan
WORKDIR $HOME
USER jovyan

CMD ["jupyterhub-singleuser"]
```

This Dockerfile should work with just about any base image in the `FROM` line, provided it has Python 3 installed.

## 1.5 Contributing

### 1.5.1 Contributing

Thank you for thinking about contributing to dockerspawner!

If you would like to contribute to the project, please read our contributor documentation and our Code of Conduct (reporting guidelines) as this helps keep our community welcoming to as many people as possible.

#### Development install

For a **development install**:

1. Clone the repository:

```
$ git clone https://github.com/jupyterhub/dockerspawner
```

2. Install from source:

```
$ cd dockerspawner
$ python3 -m pip install -r dev-requirements.txt -e .
```

## 1.6 API Reference

### 1.6.1 DockerSpawner API

#### Module: *dockerspawner*

*DockerSpawner*

**class** dockerspawner.**DockerSpawner**(***kwargs: Any*)

A Spawner for JupyterHub that runs each user's server in a separate docker container

**allowed_images** **c.DockerSpawner.allowed_images = Union({})**

List or dict of images that users can run.

If specified, users will be presented with a form from which they can select an image to run.

If a dictionary, the keys will be the options presented to users and the values the actual images that will be launched.

If a list, will be cast to a dictionary where keys and values are the same (i.e. a shortcut for presenting the actual images directly to users).

If a callable, will be called with the Spawner instance as its only argument. The user is accessible as spawner.user. The callable should return a dict or list or None as above.

If empty (default), the value from `image` is used and any attempt to specify the image via user_options will result in an error.

Changed in version 13: Empty allowed_images means no user-specified images are allowed. This is the default. Prior to 13, restricting to single image required a length-1 list, e.g. `allowed_images = [image]`.

Added in version 13: To allow any image, specify `allowed_images = "*"`.

Changed in version 12.0: `DockerSpawner.image_whitelist` renamed to `allowed_images`

**apply_user_options c.DockerSpawner.apply_user_options = Union(None)**

Hook to apply inputs from user_options to the Spawner.

Typically takes values in user_options, validates them, and updates Spawner attributes:

```
def apply_user_options(spawner, user_options):
    if "image" in user_options and isinstance(user_options["image"], str):
        spawner.image = user_options["image"]

c.Spawner.apply_user_options = apply_user_options
```

*apply_user_options may* be async.

Default: do nothing.

Typically a callable which takes *(spawner: Spawner, user_options: dict)*, but for simple cases this can be a dict mapping user option fields to Spawner attribute names, e.g.:

```
c.Spawner.apply_user_options = {"image_input": "image"}
c.Spawner.options_from_form = "simple"
```

allows users to specify the image attribute, but not any others. Because *user_options* generally comes in as strings in form data, the dictionary mode uses traitlets *from_string* to coerce strings to values, which allows setting simple values from strings (e.g. numbers) without needing to implement callable hooks.

> **Note**
>
> Because *user_options* is user input and may be set directly via the REST API, no assumptions should be made on its structure or contents. An empty dict should always be supported. Make sure to validate any inputs before applying them, either in this callable, or in whatever is consuming the value if this is a dict.

Added in version 5.3: Prior to 5.3, applying user options must be done in *Spawner.start()* or *Spawner.pre_spawn_hook()*.

**args c.DockerSpawner.args = List()**

Extra arguments to be passed to the single-user server.

Some spawners allow shell-style expansion here, allowing you to use environment variables here. Most, including the default, do not. Consult the documentation for your spawner to verify!

**auth_state_hook c.DockerSpawner.auth_state_hook = Any(None)**

An optional hook function that you can implement to pass *auth_state* to the spawner after it has been initialized but before it starts. The *auth_state* dictionary may be set by the *.authenticate()* method of the authenticator. This hook enables you to pass some or all of that information to your spawner.

Example:

```
def userdata_hook(spawner, auth_state):
    spawner.userdata = auth_state["userdata"]

c.Spawner.auth_state_hook = userdata_hook
```

**certs_volume_name c.DockerSpawner.certs_volume_name = Unicode('{prefix}ssl-{username}')**

Volume name

The same string-templating applies to this as other volume names.

**property client**

single global client instance

**client_kwargs c.DockerSpawner.client_kwargs = Dict()**

Extra keyword arguments to pass to the docker.Client constructor.

**cmd c.DockerSpawner.cmd = Command()**

The command used for starting the single-user server.

Provide either a string or a list containing the path to the startup script command. Extra arguments, other than this path, should be provided via *args*.

This is usually set if you want to start the single-user server in a different python environment (with virtualenv/conda) than JupyterHub itself.

Some spawners allow shell-style expansion here, allowing you to use environment variables. Most, including the default, do not. Consult the documentation for your spawner to verify!

**consecutive_failure_limit c.DockerSpawner.consecutive_failure_limit = Int(0)**

Maximum number of consecutive failures to allow before shutting down JupyterHub.

This helps JupyterHub recover from a certain class of problem preventing launch in contexts where the Hub is automatically restarted (e.g. systemd, docker, kubernetes).

A limit of 0 means no limit and consecutive failures will not be tracked.

**property container_id**

alias for object_id

**container_image c.DockerSpawner.container_image = Unicode('quay.io/jupyterhub/singleuser:5.4')**

Deprecated, use `DockerSpawner.image`.

**container_ip c.DockerSpawner.container_ip = Unicode('127.0.0.1')**

Deprecated, use `DockerSpawner.host_ip`

**property container_name**

alias for object_name

**container_name_template c.DockerSpawner.container_name_template = Unicode('')**

Deprecated, use `DockerSpawner.name_template`.

**container_port c.DockerSpawner.container_port = Int(8888)**

Deprecated, use `DockerSpawner.port`.

**container_prefix c.DockerSpawner.container_prefix = Unicode('')**

Deprecated, use `DockerSpawner.prefix`.

**cpu_guarantee c.DockerSpawner.cpu_guarantee = Float(None)**

Minimum number of cpu-cores a single-user notebook server is guaranteed to have available.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

**This is a configuration setting. Your spawner must implement support for the limit to work.** The default spawner, *LocalProcessSpawner*, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

**cpu_limit c.DockerSpawner.cpu_limit = Union()**

CPU limit for containers

Will set cpu_quota = cpu_limit * cpu_period

The default cpu_period of 100ms will be used, unless overridden in extra_host_config.

Alternatively to a single float, cpu_limit can also be a callable that takes the spawner as the only argument and returns a float:

**def per_user_cpu_limit(spawner):**
username = spawner.user.name cpu_limits = {'alice': 2.5, 'bob': 2} return cpu_limits.get(username, 1)

c.DockerSpawner.cpu_limit = per_user_cpu_limit

**async create_object()**

Create the container/service object

**debug c.DockerSpawner.debug = Bool(False)**

Enable debug-logging of the single-user server

**default_url c.DockerSpawner.default_url = Unicode('')**

The URL the single-user server should start in.

*{username}* will be expanded to the user's username

Example uses:

- You can set *notebook_dir* to */* and *default_url* to */tree/home/{username}* to allow people to navigate the whole filesystem from their notebook server, but still start in their home directory.
- Start with */notebooks* instead of */tree* if *default_url* points to a notebook instead of a directory.
- You can set this to */lab* to have JupyterLab start by default, rather than Jupyter Notebook.

**disable_user_config c.DockerSpawner.disable_user_config = Bool(False)**

Disable per-user configuration of single-user servers.

When starting the user's single-user server, any config file found in the user's $HOME directory will be ignored.

Note: a user could circumvent this if the user modifies their Python environment, such as when they have their own conda environments / virtualenvs / containers.

**docker**(*method*, **args*, ***kwargs*)

Call a docker method in a background thread

returns a Future

**env_keep c.DockerSpawner.env_keep = List()**

List of environment variables for the single-user server to inherit from the JupyterHub process.

This list is used to ensure that sensitive information in the JupyterHub process's environment (such as *CONFIGPROXY_AUTH_TOKEN*) is not passed to the single-user server's process.

**environment c.DockerSpawner.environment = Dict()**

Extra environment variables to set for the single-user server's process.

**Environment variables that end up in the single-user server's process come from 3 sources:**

- This *environment* configurable
- The JupyterHub process' environment variables that are listed in *env_keep*

- Variables to establish contact between the single-user notebook and the hub (such as JUPYTERHUB_API_TOKEN)

The *environment* configurable should be set by JupyterHub administrators to add installation specific environment variables. It is a dict where the key is the name of the environment variable, and the value can be a string or a callable. If it is a callable, it will be called with one parameter (the spawner instance), and should return a string fairly quickly (no blocking operations please!).

Note that the spawner class' interface is not guaranteed to be exactly same across upgrades, so if you are using the callable take care to verify it continues to work after upgrades!

Changed in version 1.2: environment from this configuration has highest priority, allowing override of 'default' env variables, such as JUPYTERHUB_API_URL.

**escape c.DockerSpawner.escape = Any(None)**

Override escaping with any callable of the form escape(str)->str

This is used to ensure docker-safe container names, etc.

The default escaping should ensure safety and validity, but can produce cumbersome strings in cases.

Set c.DockerSpawner.escape = 'legacy' to preserve the earlier, unsafe behavior if it worked for you.

Added in version 12.0.

Changed in version 12.0: Escaping has changed in 12.0 to ensure safety, but existing deployments will get different container and volume names.

**property escaped_name**

Escape the username so it's safe for docker objects

**property executor**

single global executor

**extra_create_kwargs c.DockerSpawner.extra_create_kwargs = Union()**

Additional args to pass for container create

For example, to change the user the container is started as:

```
c.DockerSpawner.extra_create_kwargs = {
    "user": "root" # Can also be an integer UID
}
```

The above is equivalent to `docker run --user root`.

If a callable, will be called with the Spawner as the only argument, must return the same dictionary structure, and may be async.

Changed in version 13: Added callable support.

**extra_host_config c.DockerSpawner.extra_host_config = Union()**

Additional args to create_host_config for container create.

If a callable, will be called with the Spawner as the only argument, must return the same dictionary structure, and may be async.

Changed in version 13: Added callable support.

**format_volume_name c.DockerSpawner.format_volume_name = Any(None)**

Any callable that accepts a string template and a DockerSpawner instance as parameters in that order and returns a string.

Reusable implementations should go in dockerspawner.VolumeNamingStrategy, tests should go in ...

**async get_command()**

Get the command to run (full command + args)

**get_env()**

Return the environment dict to use for the Spawner.

This applies things like *env_keep*, anything defined in *Spawner.environment*, and adds the API token to the env.

When overriding in subclasses, subclasses must call *super().get_env()*, extend the returned dict and return it.

Use this to access the env in Spawner.start to allow extension in subclasses.

**async get_ip_and_port()**

Queries Docker daemon for container's IP and port.

If you are using network_mode=host, you will need to override this method as follows:

```
async def get_ip_and_port(self):
    return self.host_ip, self.port
```

You will need to make sure host_ip and port are correct, which depends on the route to the container and the port it opens.

**get_state()**

Save state of spawner into database.

A black box of extra state for custom spawners. The returned value of this is passed to *load_state*.

Subclasses should call *super().get_state()*, augment the state returned from there, and return that state.

**Returns**
**state** – a JSONable dict of state

**Return type**
dict

**group_overrides c.DockerSpawner.group_overrides = Union()**

Override specific traitlets based on group membership of the user.

This can be a dict, or a callable that returns a dict. The keys of the dict are *only* used for lexicographical sorting, to guarantee consistent ordering of the overrides. If it is a callable, it may be async, and will be passed one parameter - the spawner instance. It should return a dictionary.

The values of the dict are dicts with the following keys:

- *"groups"* - If the user belongs to *any* of these groups, these overrides are applied to their server before spawning.
- *"spawner_override"* - a dictionary with overrides to apply to the Spawner settings. Each value can be either the final value to change or a callable that take the *Spawner* instance as parameter and returns the final value. If the traitlet being overriden is a *dictionary*, the dictionary will be *recursively updated*, rather than overriden. If you want to remove a key, set its value to *None*.

### Example

The following example config will:

1. Add the environment variable "AM_I_GROUP_ALPHA" to everyone in the "group-alpha" group

2. Add the environment variable "AM_I_GROUP_BETA" to everyone in the "group-beta" group. If a user is part of both "group-beta" and "group-alpha", they will get *both* these env vars, due to the dictionary merging functionality.
3. Add a higher memory limit for everyone in the "group-beta" group.

```
c.Spawner.group_overrides = {
    "01-group-alpha-env-add": {
        "groups": ["group-alpha"],
        "spawner_override": {"environment": {"AM_I_GROUP_ALPHA": "yes"}},
    },
    "02-group-beta-env-add": {
        "groups": ["group-beta"],
        "spawner_override": {"environment": {"AM_I_GROUP_BETA": "yes"}},
    },
    "03-group-beta-mem-limit": {
        "groups": ["group-beta"],
        "spawner_override": {"mem_limit": "2G"}
    }
}
```

**host_ip c.DockerSpawner.host_ip = Unicode('127.0.0.1')**

The ip address on the host on which to expose the container's port

Typically 127.0.0.1, but can be public interfaces as well in cases where the Hub and/or proxy are on different machines from the user containers.

Only used when `use_internal_ip = False`.

**http_timeout c.DockerSpawner.http_timeout = Int(30)**

Timeout (in seconds) before giving up on a spawned HTTP server

Once a server has successfully been spawned, this is the amount of time we wait before assuming that the server is unable to accept connections.

**hub_connect_url c.DockerSpawner.hub_connect_url = Unicode(None)**

The URL the single-user server should connect to the Hub.

If the Hub URL set in your JupyterHub config is not reachable from spawned notebooks, you can set differnt URL by this config.

Is None if you don't need to change the URL.

**hub_ip_connect c.DockerSpawner.hub_ip_connect = Unicode('')**

DEPRECATED since JupyterHub 0.8. Use c.JupyterHub.hub_connect_ip.

**image c.DockerSpawner.image = Unicode('quay.io/jupyterhub/singleuser:5.4')**

The image to use for single-user servers.

This image should have the same version of jupyterhub as the Hub itself installed.

If the default command of the image does not launch jupyterhub-singleuser, set `c.Spawner.cmd` to launch jupyterhub-singleuser, e.g.

Any of the jupyter docker-stacks should work without additional config, as long as the version of jupyterhub in the image is compatible.

**image_whitelist c.DockerSpawner.image_whitelist = Union()**

Deprecated, use `DockerSpawner.allowed_images`.

property internal_hostname

Return our hostname

used with internal SSL

ip c.DockerSpawner.ip = Unicode('127.0.0.1')

The IP address (or hostname) the single-user server should listen on.

Usually either '127.0.0.1' (default) or '0.0.0.0'. On IPv6 only networks use '::1' or '::'.

If the spawned singleuser server is running JupyterHub 5.3.0 later You can set this to the empty string '' to indicate both IPv4 and IPv6.

The JupyterHub proxy implementation should be able to send packets to this interface.

Subclasses which launch remotely or in containers should override the default to '0.0.0.0'.

Changed in version 5.3: An empty string '' means all interfaces (IPv4 and IPv6). Prior to this the behaviour of '' was not defined.

Changed in version 2.0: Default changed to '127.0.0.1', from unspecified.

links c.DockerSpawner.links = Dict()

Specify docker link mapping to add to the container, e.g.

links = {'jupyterhub': 'jupyterhub'}

If the Hub is running in a Docker container, this can simplify routing because all traffic will be using docker hostnames.

load_state(*state*)

Restore state of spawner from database.

Called for each user's spawner after the hub process restarts.

*state* is a dict that'll contain the value returned by *get_state* of the spawner, or {} if the spawner hasn't persisted any state yet.

Override in subclasses to restore any extra state that is needed to track the single-user server for that user. Subclasses should call super().

mem_guarantee c.DockerSpawner.mem_guarantee = ByteSpecification(None)

Minimum number of bytes a single-user notebook server is guaranteed to have available.

**Allows the following suffixes:**

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

**This is a configuration setting. Your spawner must implement support for the limit to work.** The default spawner, *LocalProcessSpawner*, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

mem_limit c.DockerSpawner.mem_limit = Union()

Maximum number of bytes a single-user notebook server is allowed to use. Allows the following suffixes:

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes

- T -> Terabytes

If the single user server tries to allocate more memory than this, it will fail. There is no guarantee that the single-user notebook server will be able to allocate this much memory - only that it can not allocate more than this.

Alternatively to a string it can also be a callable that takes the spawner as the only argument and returns a string:

**def per_user_mem_limit(spawner):**
username = spawner.user.name ram_limits = {'alice': '4G', 'bob': '2G'} return ram_limits.get(username, '1G')

c.DockerSpawner.mem_limit = per_user_mem_limit

`mounts c.DockerSpawner.mounts = List()`

List of dict with keys to match docker.types.Mount for more advanced configuration of mouted volumes. As with volumes, if the default format_volume_name is in use, you can use {username} in the source or target paths, and it will be replaced with the current user's name.

`async move_certs`(*paths*)

Takes certificate paths and makes them available to the notebook server

**Parameters**
**paths** (*dict*) – a list of paths for key, cert, and CA. These paths will be resolvable and readable by the Hub process, but not necessarily by the notebook server.

**Returns**

**a list (potentially altered) of paths for key, cert, and CA.**
These paths should be resolvable and readable by the notebook server to be launched.

**Return type**
dict

*.move_certs* is called after certs for the singleuser notebook have been created by create_certs.

By default, certs are created in a standard, central location defined by *internal_certs_location*. For a local, single-host deployment of JupyterHub, this should suffice. If, however, singleuser notebooks are spawned on other hosts, *.move_certs* should be overridden to move these files appropriately. This could mean using *scp* to copy them to another host, moving them to a volume mounted in a docker container, or exporting them as a secret in kubernetes.

`move_certs_image c.DockerSpawner.move_certs_image = Unicode('busybox:1.30.1')`

The image used to stage internal SSL certificates.

Busybox is used because we just need an empty container that waits while we stage files into the volume via .put_archive.

`name_template c.DockerSpawner.name_template = Unicode('')`

Name of the container or service: with {username}, {imagename}, {prefix}, {servername} replacements. {raw_username} can be used for the original, not escaped username (may contain uppercase, special characters). It is important to include {servername} if JupyterHub's "named servers" are enabled (`JupyterHub.allow_named_servers = True`). If the server is named, the default name_template is "{prefix}-{username}–{servername}". If it is unnamed, the default name_template is "{prefix}-{username}".

Note: when using named servers, it is important that the separator between {username} and {servername} is not a character that can occur in an escaped {username}, and also not the single escape character '-'.

**network_name c.DockerSpawner.network_name = Unicode('bridge')**

Run the containers on this docker network. If it is an internal docker network, the Hub should be on the same network, as internal docker IP addresses will be used. For bridge networking, external ports will be bound.

**notebook_dir c.DockerSpawner.notebook_dir = Unicode('')**

Path to the notebook directory for the single-user server.

The user sees a file listing of this directory when the notebook interface is started. The current interface does not easily allow browsing beyond the subdirectories in this directory's tree.

~ will be expanded to the home directory of the user, and {username} will be replaced with the name of the user.

Note that this does *not* prevent users from accessing files outside of this path! They can do so with many other means.

**oauth_client_allowed_scopes c.DockerSpawner.oauth_client_allowed_scopes = Union()**

Allowed scopes for oauth tokens issued by this server's oauth client.

This sets the maximum and default scopes assigned to oauth tokens issued by a single-user server's oauth client (i.e. tokens stored in browsers after authenticating with the server), defining what actions the server can take on behalf of logged-in users.

Access to the current server will always be included in this list. This property contains additional scopes. Default is an empty list, meaning minimal permissions to identify users, no actions can be taken on their behalf.

If callable, will be called with the Spawner as a single argument. Callables may be async.

**oauth_roles c.DockerSpawner.oauth_roles = Union()**

Allowed roles for oauth tokens.

Deprecated in 3.0: use oauth_client_allowed_scopes

**options_form c.DockerSpawner.options_form = Union()**

An HTML form for options a user can specify on launching their server.

The surrounding *<form>* element and the submit button are already provided.

For example:

```
Set your key:
<input name="key" val="default_key"></input>
<br>
Choose a letter:
<select name="letter" multiple="true">
  <option value="A">The letter A</option>
  <option value="B">The letter B</option>
</select>
```

The data from this form submission will be passed on to your spawner in *self.user_options*

Instead of a form snippet string, this could also be a callable that takes as one parameter the current spawner instance and returns a string. The callable will be called asynchronously if it returns a future, rather than a str. Note that the interface of the spawner class is not deemed stable across versions, so using this functionality might cause your JupyterHub upgrades to break.

**options_from_form c.DockerSpawner.options_from_form = Union()**

Interpret HTTP form data

Form data will always arrive as a dict of lists of strings. Override this function to understand single-values, numbers, etc.

This should coerce form data into the structure expected by self.user_options, which must be a dict, and should be JSON-serializeable, though it can contain bytes in addition to standard JSON data types.

This method should not have any side effects. Any handling of *user_options* should be done in *.apply_user_options()* (JupyterHub 5.3) or *.start()* (JupyterHub 5.2 or older) to ensure consistent behavior across servers spawned via the API and form submission page.

Instances will receive this data on self.user_options, after passing through this function, prior to *Spawner.start*.

Changed in version 1.0: user_options are persisted in the JupyterHub database to be reused on subsequent spawns if no options are given. user_options is serialized to JSON as part of this persistence (with additional support for bytes in case of uploaded file data), and any non-bytes non-jsonable values will be replaced with None if the user_options are re-used.

Added in version 5.3: The strings *'simple'* and *'passthrough'* may be specified to select some predefined behavior. These are the only string values accepted.

*'passthrough'* is the longstanding default behavior, where form data is stored in *user_options* without modification. With *'passthrough'*, *user_options* from a form will always be a dict of lists of strings.

*'simple'* applies some minimal processing that works for most simple forms:

- Single-value fields get unpacked from lists. They are still always strings, no attempt is made to parse numbers, etc..
- Multi-value fields are left alone.
- The default checked value of "on" for a checkbox is converted to True. This is the only non-string value that can be produced.

Example for *'simple'*:

```
{
    "image": ["myimage"],
    "checked": ["on"], # checkbox
    "multi-select": ["a", "b"],
}
# becomes
{
    "image": "myimage",
    "checked": True,
    "multi-select": ["a", "b"],
}
```

**async poll()**

Check for my id in `docker ps`

**poll_interval c.DockerSpawner.poll_interval = Int(30)**

Interval (in seconds) on which to poll the spawner for single-user server's status.

At every poll interval, each spawner's *.poll* method is called, which checks if the single-user server is still running. If it isn't running, then JupyterHub modifies its own state accordingly and removes appropriate routes from the configurable proxy.

**poll_jitter c.DockerSpawner.poll_jitter = Float(0.1)**

Jitter fraction for poll_interval.

Avoids alignment of poll calls for many Spawners, e.g. when restarting JupyterHub, which restarts all polls for running Spawners.

*poll_jitter=0* means no jitter, 0.1 means 10%, etc.

**port c.DockerSpawner.port = Int(0)**

The port for single-user servers to listen on.

Defaults to *0*, which uses a randomly allocated port number each time.

If set to a non-zero value, all Spawners will use the same port, which only makes sense if each server is on a different address, e.g. in containers.

New in version 0.7.

**post_start_cmd c.DockerSpawner.post_start_cmd = UnicodeOrFalse(False)**

If specified, the command will be executed inside the container after starting. Similar to using 'docker exec'

**async post_start_exec()**

Execute additional command inside the container after starting it.

e.g. calling 'docker exec'

**post_stop_hook c.DockerSpawner.post_stop_hook = Any(None)**

An optional hook function that you can implement to do work after the spawner stops.

This can be set independent of any concrete spawner implementation.

**pre_spawn_hook c.DockerSpawner.pre_spawn_hook = Any(None)**

An optional hook function that you can implement to do some bootstrapping work before the spawner starts. For example, create a directory for your user or load initial content.

This can be set independent of any concrete spawner implementation.

This maybe a coroutine.

Example:

```
def my_hook(spawner):
    username = spawner.user.name
    spawner.environment["GREETING"] = f"Hello {username}"

c.Spawner.pre_spawn_hook = my_hook
```

**prefix c.DockerSpawner.prefix = Unicode('jupyter')**

Prefix for container names. See name_template for full container name for a particular user's server.

**progress_ready_hook c.DockerSpawner.progress_ready_hook = Any(None)**

An optional hook function that you can implement to modify the ready event, which will be shown to the user on the spawn progress page when their server is ready.

This can be set independent of any concrete spawner implementation.

This maybe a coroutine.

Example:

```
async def my_ready_hook(spawner, ready_event):
    ready_event["html_message"] = f"Server {spawner.name} is ready for {spawner.
↪user.name}"
    return ready_event

c.Spawner.progress_ready_hook = my_ready_hook
```

**async pull_image**(*image*)

Pull the image, if needed

- pulls it unconditionally if pull_policy == 'always'
- skipped entirely if pull_policy == 'skip' (default for swarm)
- otherwise, checks if it exists, and - raises if pull_policy == 'never' - pulls if pull_policy == 'ifnotpresent'

**pull_policy c.DockerSpawner.pull_policy = CaselessStrEnum('ifnotpresent')**

The policy for pulling the user docker image.

Choices:

- ifnotpresent: pull if the image is not already present (default)
- always: always pull the image to check for updates, even if it is present
- never: never perform a pull, raise if image is not present
- skip: never perform a pull, skip the step entirely (like never, but without raising when images are not present; default for swarm)

**read_only_volumes c.DockerSpawner.read_only_volumes = Dict()**

Map from host file/directory to container file/directory. Volumes specified here will be read-only in the container.

If format_volume_name is not set, default_format_volume_name is used for naming volumes. In this case, if you use {username} in either the host or guest file/directory path, it will be replaced with the current user's name.

**remove c.DockerSpawner.remove = Bool(False)**

If `True`, delete containers when servers are stopped.

This will destroy any data in the container not stored in mounted volumes.

**remove_containers c.DockerSpawner.remove_containers = Bool(False)**

Deprecated, use `DockerSpawner.remove`.

**server_token_scopes c.DockerSpawner.server_token_scopes = Union()**

The list of scopes to request for $JUPYTERHUB_API_TOKEN

If not specified, the scopes in the *server* role will be used (unchanged from pre-4.0).

If callable, will be called with the Spawner instance as its sole argument (JupyterHub user available as spawner.user).

JUPYTERHUB_API_TOKEN will be assigned the _subset_ of these scopes that are held by the user (as in oauth_client_allowed_scopes).

Added in version 4.0.

**ssl_alt_names c.DockerSpawner.ssl_alt_names = Union()**

List of SSL alt names (list of strings).

May be set in config if all spawners should have the same value(s), or set at runtime by Spawner that know their names.

Changed in version 5.4.1: May now be a callable. The callable will receive the Spawner as its only argument, and must return a list of strings. It may be async.

**ssl_alt_names_include_local c.DockerSpawner.ssl_alt_names_include_local = Bool(True)**

Whether to include *DNS:localhost*, *IP:127.0.0.1* in alt names

**async start()**

Start the single-user server in a docker container.

If the container exists and `c.DockerSpawner.remove` is `True`, then the container is removed first. Otherwise, the existing containers will be restarted.

**async start_object()**

Actually start the container/service

e.g. calling `docker start`

**start_timeout c.DockerSpawner.start_timeout = Int(60)**

Timeout (in seconds) before giving up on starting of single-user server.

This is the timeout for start to return, not the timeout for the server to respond. Callers of spawner.start will assume that startup has failed if it takes longer than this. start should return when the server process is started and its location is known.

**async stop**(*now=False*)

Stop the container

Will remove the container if `c.DockerSpawner.remove` is `True`.

Consider using pause/unpause when docker-py adds support.

**async stop_object()**

Stop the container/service

e.g. calling `docker stop`. Does not remove the container.

**template_namespace()**

Return the template namespace for format-string formatting.

Currently used on default_url and notebook_dir.

Subclasses may add items to the available namespace.

The default implementation includes:

```
{
  'username': user.name,
  'base_url': users_base_url,
}
```

**Returns**
namespace for string formatting.

**Return type**
ns (dict)

**tls c.DockerSpawner.tls = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_assert_hostname c.DockerSpawner.tls_assert_hostname = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_ca c.DockerSpawner.tls_ca = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_cert c.DockerSpawner.tls_cert = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**property tls_client**

A tuple consisting of the TLS client certificate and key if they have been provided, otherwise None.

**tls_config c.DockerSpawner.tls_config = Dict()**

Arguments to pass to docker TLS configuration.

See docker.client.TLSConfig constructor for options.

**tls_key c.DockerSpawner.tls_key = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_verify c.DockerSpawner.tls_verify = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**use_docker_client_env c.DockerSpawner.use_docker_client_env = Bool(True)**

Deprecated. Docker env variables are always used if present.

**use_internal_hostname c.DockerSpawner.use_internal_hostname = Bool(False)**

Use the docker hostname for connecting.

instead of an IP address. This should work in general when using docker networks, and must be used when internal_ssl is enabled. It is enabled by default if internal_ssl is enabled.

**use_internal_ip c.DockerSpawner.use_internal_ip = Bool(False)**

Enable the usage of the internal docker ip. This is useful if you are running jupyterhub (as a container) and the user containers within the same docker network. E.g. by mounting the docker socket of the host into the jupyterhub container. Default is `True` if using a docker network, `False` if bridge or host networking is used.

**property volume_binds**

The second half of declaring a volume with docker-py happens when you actually call start(). The required format is a dict of dicts that looks like:

```
{
    host_location: {'bind': container_location, 'mode': 'rw'}
}
```

Mode may be 'ro', 'rw', 'z', or 'Z'.

**property volume_mount_points**

Volumes are declared in docker-py in two stages. First, you declare all the locations where you're going to mount volumes when you call create_container.

Returns a sorted list of all the values in self.volumes or self.read_only_volumes.

**volumes c.DockerSpawner.volumes = Dict()**

Map from host file/directory to container (guest) file/directory mount point and (optionally) a mode. When specifying the guest mount point (bind) for the volume, you may use a dict or str. If a str, then the volume will default to a read-write (mode="rw"). With a dict, the bind is identified by "bind" and the "mode" may be one of "rw" (default), "ro" (read-only), "z" (public/shared SELinux volume label), and "Z" (private/unshared SELinux volume label).

If format_volume_name is not set, default_format_volume_name is used for naming volumes. In this case, if you use {username} in either the host or guest file/directory path, it will be replaced with the current user's name.

**property will_resume**

Whether the Spawner will resume on next start

Default is False where each launch of the Spawner will be a new instance. If True, an existing Spawner will resume instead of starting anew (e.g. resuming a Docker container), and API tokens in use when the Spawner stops will not be deleted.

### *SwarmSpawner*

**class** dockerspawner.**SwarmSpawner**(***kwargs: Any*)

A Spawner for JupyterHub that runs each user's server in a separate docker service

**allowed_images c.SwarmSpawner.allowed_images = Union({})**

List or dict of images that users can run.

If specified, users will be presented with a form from which they can select an image to run.

If a dictionary, the keys will be the options presented to users and the values the actual images that will be launched.

If a list, will be cast to a dictionary where keys and values are the same (i.e. a shortcut for presenting the actual images directly to users).

If a callable, will be called with the Spawner instance as its only argument. The user is accessible as spawner.user. The callable should return a dict or list or None as above.

If empty (default), the value from `image` is used and any attempt to specify the image via user_options will result in an error.

Changed in version 13: Empty allowed_images means no user-specified images are allowed. This is the default. Prior to 13, restricting to single image required a length-1 list, e.g. `allowed_images = [image]`.

Added in version 13: To allow any image, specify `allowed_images = "*"`.

Changed in version 12.0: `DockerSpawner.image_whitelist` renamed to `allowed_images`

**apply_user_options c.SwarmSpawner.apply_user_options = Union(None)**

Hook to apply inputs from user_options to the Spawner.

Typically takes values in user_options, validates them, and updates Spawner attributes:

```
def apply_user_options(spawner, user_options):
    if "image" in user_options and isinstance(user_options["image"], str):
        spawner.image = user_options["image"]

c.Spawner.apply_user_options = apply_user_options
```

*apply_user_options may* be async.

Default: do nothing.

Typically a callable which takes *(spawner: Spawner, user_options: dict)*, but for simple cases this can be a dict mapping user option fields to Spawner attribute names, e.g.:

```
c.Spawner.apply_user_options = {"image_input": "image"}
c.Spawner.options_from_form = "simple"
```

allows users to specify the image attribute, but not any others. Because *user_options* generally comes in as strings in form data, the dictionary mode uses traitlets *from_string* to coerce strings to values, which allows setting simple values from strings (e.g. numbers) without needing to implement callable hooks.

**Note**

Because *user_options* is user input and may be set directly via the REST API, no assumptions should be made on its structure or contents. An empty dict should always be supported. Make sure to validate any inputs before applying them, either in this callable, or in whatever is consuming the value if this is a dict.

Added in version 5.3: Prior to 5.3, applying user options must be done in *Spawner.start()* or *Spawner.pre_spawn_hook()*.

**args c.SwarmSpawner.args = List()**

Extra arguments to be passed to the single-user server.

Some spawners allow shell-style expansion here, allowing you to use environment variables here. Most, including the default, do not. Consult the documentation for your spawner to verify!

**auth_state_hook c.SwarmSpawner.auth_state_hook = Any(None)**

An optional hook function that you can implement to pass *auth_state* to the spawner after it has been initialized but before it starts. The *auth_state* dictionary may be set by the *.authenticate()* method of the authenticator. This hook enables you to pass some or all of that information to your spawner.

Example:

```
def userdata_hook(spawner, auth_state):
    spawner.userdata = auth_state["userdata"]

c.Spawner.auth_state_hook = userdata_hook
```

**certs_volume_name c.SwarmSpawner.certs_volume_name = Unicode('{prefix}ssl-{username}')**

Volume name

The same string-templating applies to this as other volume names.

**client_kwargs c.SwarmSpawner.client_kwargs = Dict()**

Extra keyword arguments to pass to the docker.Client constructor.

**cmd c.SwarmSpawner.cmd = Command()**

The command used for starting the single-user server.

Provide either a string or a list containing the path to the startup script command. Extra arguments, other than this path, should be provided via *args*.

This is usually set if you want to start the single-user server in a different python environment (with virtualenv/conda) than JupyterHub itself.

Some spawners allow shell-style expansion here, allowing you to use environment variables. Most, including the default, do not. Consult the documentation for your spawner to verify!

**consecutive_failure_limit c.SwarmSpawner.consecutive_failure_limit = Int(0)**

Maximum number of consecutive failures to allow before shutting down JupyterHub.

This helps JupyterHub recover from a certain class of problem preventing launch in contexts where the Hub is automatically restarted (e.g. systemd, docker, kubernetes).

A limit of 0 means no limit and consecutive failures will not be tracked.

**container_image c.SwarmSpawner.container_image = Unicode('quay.io/jupyterhub/singleuser:5.4')**

Deprecated, use `DockerSpawner.image`.

**container_ip c.SwarmSpawner.container_ip = Unicode('127.0.0.1')**

Deprecated, use `DockerSpawner.host_ip`

**container_name_template c.SwarmSpawner.container_name_template = Unicode('')**

Deprecated, use `DockerSpawner.name_template`.

**container_port c.SwarmSpawner.container_port = Int(8888)**

Deprecated, use `DockerSpawner.port`.

**container_prefix c.SwarmSpawner.container_prefix = Unicode('')**

Deprecated, use `DockerSpawner.prefix`.

**cpu_guarantee c.SwarmSpawner.cpu_guarantee = Float(None)**

Minimum number of cpu-cores a single-user notebook server is guaranteed to have available.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

**This is a configuration setting. Your spawner must implement support for the limit to work.** The default spawner, *LocalProcessSpawner*, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

**cpu_limit c.SwarmSpawner.cpu_limit = Union()**

CPU limit for containers

Will set cpu_quota = cpu_limit * cpu_period

The default cpu_period of 100ms will be used, unless overridden in extra_host_config.

Alternatively to a single float, cpu_limit can also be a callable that takes the spawner as the only argument and returns a float:

**def per_user_cpu_limit(spawner):**
username = spawner.user.name cpu_limits = {'alice': 2.5, 'bob': 2} return cpu_limits.get(username, 1)

c.DockerSpawner.cpu_limit = per_user_cpu_limit

**async create_object()**

Start the single-user server in a docker service.

**debug c.SwarmSpawner.debug = Bool(False)**

Enable debug-logging of the single-user server

**default_url c.SwarmSpawner.default_url = Unicode('')**

The URL the single-user server should start in.

*{username}* will be expanded to the user's username

Example uses:

- You can set *notebook_dir* to `/` and *default_url* to */tree/home/{username}* to allow people to navigate the whole filesystem from their notebook server, but still start in their home directory.
- Start with */notebooks* instead of */tree* if *default_url* points to a notebook instead of a directory.
- You can set this to */lab* to have JupyterLab start by default, rather than Jupyter Notebook.

**disable_user_config c.SwarmSpawner.disable_user_config = Bool(False)**

Disable per-user configuration of single-user servers.

When starting the user's single-user server, any config file found in the user's $HOME directory will be ignored.

Note: a user could circumvent this if the user modifies their Python environment, such as when they have their own conda environments / virtualenvs / containers.

**env_keep c.SwarmSpawner.env_keep = List()**

List of environment variables for the single-user server to inherit from the JupyterHub process.

This list is used to ensure that sensitive information in the JupyterHub process's environment (such as *CONFIGPROXY_AUTH_TOKEN*) is not passed to the single-user server's process.

**environment c.SwarmSpawner.environment = Dict()**

Extra environment variables to set for the single-user server's process.

**Environment variables that end up in the single-user server's process come from 3 sources:**

- This *environment* configurable
- The JupyterHub process' environment variables that are listed in *env_keep*
- Variables to establish contact between the single-user notebook and the hub (such as JUPYTERHUB_API_TOKEN)

The *environment* configurable should be set by JupyterHub administrators to add installation specific environment variables. It is a dict where the key is the name of the environment variable, and the value can be a string or a callable. If it is a callable, it will be called with one parameter (the spawner instance), and should return a string fairly quickly (no blocking operations please!).

Note that the spawner class' interface is not guaranteed to be exactly same across upgrades, so if you are using the callable take care to verify it continues to work after upgrades!

Changed in version 1.2: environment from this configuration has highest priority, allowing override of 'default' env variables, such as JUPYTERHUB_API_URL.

**escape c.SwarmSpawner.escape = Any(None)**

Override escaping with any callable of the form escape(str)->str

This is used to ensure docker-safe container names, etc.

The default escaping should ensure safety and validity, but can produce cumbersome strings in cases.

Set c.DockerSpawner.escape = 'legacy' to preserve the earlier, unsafe behavior if it worked for you.

Added in version 12.0.

Changed in version 12.0: Escaping has changed in 12.0 to ensure safety, but existing deployments will get different container and volume names.

**extra_container_spec c.SwarmSpawner.extra_container_spec = Dict()**

Keyword arguments to pass to the ContainerSpec constructor

**extra_create_kwargs c.SwarmSpawner.extra_create_kwargs = Union()**

Additional args to pass for container create

For example, to change the user the container is started as:

```
c.DockerSpawner.extra_create_kwargs = {
    "user": "root" # Can also be an integer UID
}
```

The above is equivalent to `docker run --user root`.

If a callable, will be called with the Spawner as the only argument, must return the same dictionary structure, and may be async.

Changed in version 13: Added callable support.

**extra_endpoint_spec c.SwarmSpawner.extra_endpoint_spec = Dict()**

Keyword arguments to pass to the Endpoint constructor

**extra_host_config c.SwarmSpawner.extra_host_config = Union()**

Additional args to create_host_config for container create.

If a callable, will be called with the Spawner as the only argument, must return the same dictionary structure, and may be async.

Changed in version 13: Added callable support.

**extra_placement_spec c.SwarmSpawner.extra_placement_spec = Dict()**

Keyword arguments to pass to the Placement constructor

**extra_resources_spec c.SwarmSpawner.extra_resources_spec = Dict()**

Keyword arguments to pass to the Resources spec

**extra_task_spec c.SwarmSpawner.extra_task_spec = Dict()**

Keyword arguments to pass to the TaskTemplate constructor

**format_volume_name c.SwarmSpawner.format_volume_name = Any(None)**

Any callable that accepts a string template and a DockerSpawner instance as parameters in that order and returns a string.

Reusable implementations should go in dockerspawner.VolumeNamingStrategy, tests should go in ...

**async get_ip_and_port()**

Queries Docker daemon for service's IP and port.

If you are using network_mode=host, you will need to override this method as follows:

```
async def get_ip_and_port(self):
    return self.host_ip, self.port
```

You will need to make sure host_ip and port are correct, which depends on the route to the service and the port it opens.

**group_overrides c.SwarmSpawner.group_overrides = Union()**

Override specific traitlets based on group membership of the user.

This can be a dict, or a callable that returns a dict. The keys of the dict are *only* used for lexicographical sorting, to guarantee consistent ordering of the overrides. If it is a callable, it may be async, and will be passed one parameter - the spawner instance. It should return a dictionary.

The values of the dict are dicts with the following keys:

- *"groups"* - If the user belongs to *any* of these groups, these overrides are applied to their server before spawning.
- *"spawner_override"* - a dictionary with overrides to apply to the Spawner settings. Each value can be either the final value to change or a callable that take the *Spawner* instance as parameter and returns the final value. If the traitlet being overriden is a *dictionary*, the dictionary will be *recursively updated*, rather than overriden. If you want to remove a key, set its value to *None*.

### Example

The following example config will:

1. Add the environment variable "AM_I_GROUP_ALPHA" to everyone in the "group-alpha" group
2. Add the environment variable "AM_I_GROUP_BETA" to everyone in the "group-beta" group. If a user is part of both "group-beta" and "group-alpha", they will get *both* these env vars, due to the dictionary merging functionality.
3. Add a higher memory limit for everyone in the "group-beta" group.

```
c.Spawner.group_overrides = {
    "01-group-alpha-env-add": {
        "groups": ["group-alpha"],
        "spawner_override": {"environment": {"AM_I_GROUP_ALPHA": "yes"}},
    },
    "02-group-beta-env-add": {
        "groups": ["group-beta"],
        "spawner_override": {"environment": {"AM_I_GROUP_BETA": "yes"}},
    },
    "03-group-beta-mem-limit": {
        "groups": ["group-beta"],
        "spawner_override": {"mem_limit": "2G"}
    }
}
```

**host_ip c.SwarmSpawner.host_ip = Unicode('127.0.0.1')**

The ip address on the host on which to expose the container's port

Typically 127.0.0.1, but can be public interfaces as well in cases where the Hub and/or proxy are on different machines from the user containers.

Only used when `use_internal_ip = False`.

**http_timeout c.SwarmSpawner.http_timeout = Int(30)**

Timeout (in seconds) before giving up on a spawned HTTP server

Once a server has successfully been spawned, this is the amount of time we wait before assuming that the server is unable to accept connections.

**hub_connect_url c.SwarmSpawner.hub_connect_url = Unicode(None)**

The URL the single-user server should connect to the Hub.

If the Hub URL set in your JupyterHub config is not reachable from spawned notebooks, you can set differnt URL by this config.

Is None if you don't need to change the URL.

**hub_ip_connect c.SwarmSpawner.hub_ip_connect = Unicode('')**

DEPRECATED since JupyterHub 0.8. Use c.JupyterHub.hub_connect_ip.

**image c.SwarmSpawner.image = Unicode('quay.io/jupyterhub/singleuser:5.4')**

The image to use for single-user servers.

This image should have the same version of jupyterhub as the Hub itself installed.

If the default command of the image does not launch jupyterhub-singleuser, set `c.Spawner.cmd` to launch jupyterhub-singleuser, e.g.

Any of the jupyter docker-stacks should work without additional config, as long as the version of jupyterhub in the image is compatible.

**image_whitelist c.SwarmSpawner.image_whitelist = Union()**

Deprecated, use `DockerSpawner.allowed_images`.

**property internal_hostname**

Return our hostname

used with internal SSL

**ip c.SwarmSpawner.ip = Unicode('127.0.0.1')**

The IP address (or hostname) the single-user server should listen on.

Usually either '127.0.0.1' (default) or '0.0.0.0'. On IPv6 only networks use '::1' or '::'.

If the spawned singleuser server is running JupyterHub 5.3.0 later You can set this to the empty string '' to indicate both IPv4 and IPv6.

The JupyterHub proxy implementation should be able to send packets to this interface.

Subclasses which launch remotely or in containers should override the default to '0.0.0.0'.

Changed in version 5.3: An empty string '' means all interfaces (IPv4 and IPv6). Prior to this the behaviour of '' was not defined.

Changed in version 2.0: Default changed to '127.0.0.1', from unspecified.

**links c.SwarmSpawner.links = Dict()**

Specify docker link mapping to add to the container, e.g.

links = {'jupyterhub': 'jupyterhub'}

If the Hub is running in a Docker container, this can simplify routing because all traffic will be using docker hostnames.

**mem_guarantee c.SwarmSpawner.mem_guarantee = ByteSpecification(None)**

Minimum number of bytes a single-user notebook server is guaranteed to have available.

**Allows the following suffixes:**

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

**This is a configuration setting. Your spawner must implement support for the limit to work.** The default spawner, *LocalProcessSpawner*, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

**mem_limit c.SwarmSpawner.mem_limit = Union()**

Maximum number of bytes a single-user notebook server is allowed to use. Allows the following suffixes:

- K -> Kilobytes

- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

If the single user server tries to allocate more memory than this, it will fail. There is no guarantee that the single-user notebook server will be able to allocate this much memory - only that it can not allocate more than this.

Alternatively to a string it can also be a callable that takes the spawner as the only argument and returns a string:

**def per_user_mem_limit(spawner):**
username = spawner.user.name ram_limits = {'alice': '4G', 'bob': '2G'} return ram_limits.get(username, '1G')

c.DockerSpawner.mem_limit = per_user_mem_limit

`property mounts`

List of dict with keys to match docker.types.Mount for more advanced configuration of mouted volumes. As with volumes, if the default format_volume_name is in use, you can use {username} in the source or target paths, and it will be replaced with the current user's name.

`move_certs_image c.SwarmSpawner.move_certs_image = Unicode('busybox:1.30.1')`

The image used to stage internal SSL certificates.

Busybox is used because we just need an empty container that waits while we stage files into the volume via .put_archive.

`name_template c.SwarmSpawner.name_template = Unicode('')`

Name of the container or service: with {username}, {imagename}, {prefix}, {servername} replacements. {raw_username} can be used for the original, not escaped username (may contain uppercase, special characters). It is important to include {servername} if JupyterHub's "named servers" are enabled (`JupyterHub.allow_named_servers = True`). If the server is named, the default name_template is "{prefix}-{username}–{servername}". If it is unnamed, the default name_template is "{prefix}-{username}".

Note: when using named servers, it is important that the separator between {username} and {servername} is not a character that can occur in an escaped {username}, and also not the single escape character '-'.

`network_name c.SwarmSpawner.network_name = Unicode('bridge')`

Run the containers on this docker network. If it is an internal docker network, the Hub should be on the same network, as internal docker IP addresses will be used. For bridge networking, external ports will be bound.

`notebook_dir c.SwarmSpawner.notebook_dir = Unicode('')`

Path to the notebook directory for the single-user server.

The user sees a file listing of this directory when the notebook interface is started. The current interface does not easily allow browsing beyond the subdirectories in this directory's tree.

~ will be expanded to the home directory of the user, and {username} will be replaced with the name of the user.

Note that this does *not* prevent users from accessing files outside of this path! They can do so with many other means.

`oauth_client_allowed_scopes c.SwarmSpawner.oauth_client_allowed_scopes = Union()`

Allowed scopes for oauth tokens issued by this server's oauth client.

This sets the maximum and default scopes assigned to oauth tokens issued by a single-user server's oauth client (i.e. tokens stored in browsers after authenticating with the server), defining what actions the server can take on behalf of logged-in users.

Access to the current server will always be included in this list. This property contains additional scopes. Default is an empty list, meaning minimal permissions to identify users, no actions can be taken on their behalf.

If callable, will be called with the Spawner as a single argument. Callables may be async.

**oauth_roles c.SwarmSpawner.oauth_roles = Union()**

Allowed roles for oauth tokens.

Deprecated in 3.0: use oauth_client_allowed_scopes

**options_form c.SwarmSpawner.options_form = Union()**

An HTML form for options a user can specify on launching their server.

The surrounding *<form>* element and the submit button are already provided.

For example:

```
Set your key:
<input name="key" val="default_key"></input>
<br>
Choose a letter:
<select name="letter" multiple="true">
  <option value="A">The letter A</option>
  <option value="B">The letter B</option>
</select>
```

The data from this form submission will be passed on to your spawner in *self.user_options*

Instead of a form snippet string, this could also be a callable that takes as one parameter the current spawner instance and returns a string. The callable will be called asynchronously if it returns a future, rather than a str. Note that the interface of the spawner class is not deemed stable across versions, so using this functionality might cause your JupyterHub upgrades to break.

**options_from_form c.SwarmSpawner.options_from_form = Union()**

Interpret HTTP form data

Form data will always arrive as a dict of lists of strings. Override this function to understand single-values, numbers, etc.

This should coerce form data into the structure expected by self.user_options, which must be a dict, and should be JSON-serializeable, though it can contain bytes in addition to standard JSON data types.

This method should not have any side effects. Any handling of *user_options* should be done in *.apply_user_options()* (JupyterHub 5.3) or *.start()* (JupyterHub 5.2 or older) to ensure consistent behavior across servers spawned via the API and form submission page.

Instances will receive this data on self.user_options, after passing through this function, prior to *Spawner.start*.

Changed in version 1.0: user_options are persisted in the JupyterHub database to be reused on subsequent spawns if no options are given. user_options is serialized to JSON as part of this persistence (with additional support for bytes in case of uploaded file data), and any non-bytes non-jsonable values will be replaced with None if the user_options are re-used.

Added in version 5.3: The strings *'simple'* and *'passthrough'* may be specified to select some predefined behavior. These are the only string values accepted.

*'passthrough'* is the longstanding default behavior, where form data is stored in *user_options* without modification. With *'passthrough'*, *user_options* from a form will always be a dict of lists of strings.

*'simple'* applies some minimal processing that works for most simple forms:

- Single-value fields get unpacked from lists. They are still always strings, no attempt is made to parse numbers, etc..
- Multi-value fields are left alone.
- The default checked value of "on" for a checkbox is converted to True. This is the only non-string value that can be produced.

Example for *'simple'*:

```
{
    "image": ["myimage"],
    "checked": ["on"], # checkbox
    "multi-select": ["a", "b"],
}
# becomes
{
    "image": "myimage",
    "checked": True,
    "multi-select": ["a", "b"],
}
```

**async poll()**

Check for my id in *docker ps*

**poll_interval c.SwarmSpawner.poll_interval = Int(30)**

Interval (in seconds) on which to poll the spawner for single-user server's status.

At every poll interval, each spawner's *.poll* method is called, which checks if the single-user server is still running. If it isn't running, then JupyterHub modifies its own state accordingly and removes appropriate routes from the configurable proxy.

**poll_jitter c.SwarmSpawner.poll_jitter = Float(0.1)**

Jitter fraction for poll_interval.

Avoids alignment of poll calls for many Spawners, e.g. when restarting JupyterHub, which restarts all polls for running Spawners.

*poll_jitter=0* means no jitter, 0.1 means 10%, etc.

**port c.SwarmSpawner.port = Int(0)**

The port for single-user servers to listen on.

Defaults to *0*, which uses a randomly allocated port number each time.

If set to a non-zero value, all Spawners will use the same port, which only makes sense if each server is on a different address, e.g. in containers.

New in version 0.7.

**post_start_cmd c.SwarmSpawner.post_start_cmd = UnicodeOrFalse(False)**

If specified, the command will be executed inside the container after starting. Similar to using 'docker exec'

**post_stop_hook c.SwarmSpawner.post_stop_hook = Any(None)**

An optional hook function that you can implement to do work after the spawner stops.

This can be set independent of any concrete spawner implementation.

**pre_spawn_hook c.SwarmSpawner.pre_spawn_hook = Any(None)**

An optional hook function that you can implement to do some bootstrapping work before the spawner starts. For example, create a directory for your user or load initial content.

This can be set independent of any concrete spawner implementation.

This maybe a coroutine.

Example:

```
def my_hook(spawner):
    username = spawner.user.name
    spawner.environment["GREETING"] = f"Hello {username}"

c.Spawner.pre_spawn_hook = my_hook
```

**prefix c.SwarmSpawner.prefix = Unicode('jupyter')**

Prefix for container names. See name_template for full container name for a particular user's server.

**progress_ready_hook c.SwarmSpawner.progress_ready_hook = Any(None)**

An optional hook function that you can implement to modify the ready event, which will be shown to the user on the spawn progress page when their server is ready.

This can be set independent of any concrete spawner implementation.

This maybe a coroutine.

Example:

```
async def my_ready_hook(spawner, ready_event):
    ready_event["html_message"] = f"Server {spawner.name} is ready for {spawner.
↪user.name}"
    return ready_event

c.Spawner.progress_ready_hook = my_ready_hook
```

**pull_policy c.SwarmSpawner.pull_policy = CaselessStrEnum('ifnotpresent')**

The policy for pulling the user docker image.

Choices:

- ifnotpresent: pull if the image is not already present (default)
- always: always pull the image to check for updates, even if it is present
- never: never perform a pull, raise if image is not present
- skip: never perform a pull, skip the step entirely (like never, but without raising when images are not present; default for swarm)

**read_only_volumes c.SwarmSpawner.read_only_volumes = Dict()**

Map from host file/directory to container file/directory. Volumes specified here will be read-only in the container.

If format_volume_name is not set, default_format_volume_name is used for naming volumes. In this case, if you use {username} in either the host or guest file/directory path, it will be replaced with the current user's name.

**remove_containers c.SwarmSpawner.remove_containers = Bool(False)**

Deprecated, use `DockerSpawner.remove`.

**server_token_scopes c.SwarmSpawner.server_token_scopes = Union()**

The list of scopes to request for $JUPYTERHUB_API_TOKEN

If not specified, the scopes in the *server* role will be used (unchanged from pre-4.0).

If callable, will be called with the Spawner instance as its sole argument (JupyterHub user available as spawner.user).

JUPYTERHUB_API_TOKEN will be assigned the _subset_ of these scopes that are held by the user (as in oauth_client_allowed_scopes).

Added in version 4.0.

**property service_id**

alias for object_id

**property service_name**

alias for object_name

**ssl_alt_names c.SwarmSpawner.ssl_alt_names = Union()**

List of SSL alt names (list of strings).

May be set in config if all spawners should have the same value(s), or set at runtime by Spawner that know their names.

Changed in version 5.4.1: May now be a callable. The callable will receive the Spawner as its only argument, and must return a list of strings. It may be async.

**ssl_alt_names_include_local c.SwarmSpawner.ssl_alt_names_include_local = Bool(True)**

Whether to include *DNS:localhost*, *IP:127.0.0.1* in alt names

**async start_object()**

Not actually starting anything

but use this to wait for the container to be running.

Spawner.start shouldn't return until the Spawner believes a server is *running* somewhere, not just requested.

**start_timeout c.SwarmSpawner.start_timeout = Int(60)**

Timeout (in seconds) before giving up on starting of single-user server.

This is the timeout for start to return, not the timeout for the server to respond. Callers of spawner.start will assume that startup has failed if it takes longer than this. start should return when the server process is started and its location is known.

**async stop_object()**

Nothing to do here

There is no separate stop action for services

**tls c.SwarmSpawner.tls = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_assert_hostname c.SwarmSpawner.tls_assert_hostname = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_ca c.SwarmSpawner.tls_ca = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_cert c.SwarmSpawner.tls_cert = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_config c.SwarmSpawner.tls_config = Dict()**

Arguments to pass to docker TLS configuration.

See docker.client.TLSConfig constructor for options.

**tls_key c.SwarmSpawner.tls_key = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_verify c.SwarmSpawner.tls_verify = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**use_docker_client_env c.SwarmSpawner.use_docker_client_env = Bool(True)**

Deprecated. Docker env variables are always used if present.

**use_internal_hostname c.SwarmSpawner.use_internal_hostname = Bool(False)**

Use the docker hostname for connecting.

instead of an IP address. This should work in general when using docker networks, and must be used when internal_ssl is enabled. It is enabled by default if internal_ssl is enabled.

**use_internal_ip c.SwarmSpawner.use_internal_ip = Bool(False)**

Enable the usage of the internal docker ip. This is useful if you are running jupyterhub (as a container) and the user containers within the same docker network. E.g. by mounting the docker socket of the host into the jupyterhub container. Default is `True` if using a docker network, `False` if bridge or host networking is used.

**volume_driver c.SwarmSpawner.volume_driver = Unicode('')**

Use this driver for mounting the notebook volumes. Note that this driver must support multiple hosts in order for it to work across the swarm. For a list of possible drivers, see https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins

**volume_driver_options c.SwarmSpawner.volume_driver_options = Dict()**

Configuration options for the multi-host volume driver.

**volumes c.SwarmSpawner.volumes = Dict()**

Map from host file/directory to container (guest) file/directory mount point and (optionally) a mode. When specifying the guest mount point (bind) for the volume, you may use a dict or str. If a str, then the volume will default to a read-write (mode="rw"). With a dict, the bind is identified by "bind" and the "mode" may be one of "rw" (default), "ro" (read-only), "z" (public/shared SELinux volume label), and "Z" (private/unshared SELinux volume label).

If format_volume_name is not set, default_format_volume_name is used for naming volumes. In this case, if you use {username} in either the host or guest file/directory path, it will be replaced with the current user's name.

### *SystemUserSpawner*

**class** dockerspawner.**SystemUserSpawner**(***kwargs: Any*)

**allowed_images c.SystemUserSpawner.allowed_images = Union({})**

List or dict of images that users can run.

If specified, users will be presented with a form from which they can select an image to run.

If a dictionary, the keys will be the options presented to users and the values the actual images that will be launched.

If a list, will be cast to a dictionary where keys and values are the same (i.e. a shortcut for presenting the actual images directly to users).

If a callable, will be called with the Spawner instance as its only argument. The user is accessible as spawner.user. The callable should return a dict or list or None as above.

If empty (default), the value from `image` is used and any attempt to specify the image via user_options will result in an error.

Changed in version 13: Empty allowed_images means no user-specified images are allowed. This is the default. Prior to 13, restricting to single image required a length-1 list, e.g. `allowed_images = [image]`.

Added in version 13: To allow any image, specify `allowed_images = "*"`.

Changed in version 12.0: `DockerSpawner.image_whitelist` renamed to `allowed_images`

**apply_user_options c.SystemUserSpawner.apply_user_options = Union(None)**

Hook to apply inputs from user_options to the Spawner.

Typically takes values in user_options, validates them, and updates Spawner attributes:

```
def apply_user_options(spawner, user_options):
    if "image" in user_options and isinstance(user_options["image"], str):
        spawner.image = user_options["image"]

c.Spawner.apply_user_options = apply_user_options
```

*apply_user_options may* be async.

Default: do nothing.

Typically a callable which takes *(spawner: Spawner, user_options: dict)*, but for simple cases this can be a dict mapping user option fields to Spawner attribute names, e.g.:

```
c.Spawner.apply_user_options = {"image_input": "image"}
c.Spawner.options_from_form = "simple"
```

allows users to specify the image attribute, but not any others. Because *user_options* generally comes in as strings in form data, the dictionary mode uses traitlets *from_string* to coerce strings to values, which allows setting simple values from strings (e.g. numbers) without needing to implement callable hooks.

> **Note**
>
> Because *user_options* is user input and may be set directly via the REST API, no assumptions should be made on its structure or contents. An empty dict should always be supported. Make sure to validate any inputs before applying them, either in this callable, or in whatever is consuming the value if this is a dict.

Added in version 5.3: Prior to 5.3, applying user options must be done in *Spawner.start()* or *Spawner.pre_spawn_hook()*.

**args c.SystemUserSpawner.args = List()**

Extra arguments to be passed to the single-user server.

Some spawners allow shell-style expansion here, allowing you to use environment variables here. Most, including the default, do not. Consult the documentation for your spawner to verify!

**auth_state_hook c.SystemUserSpawner.auth_state_hook = Any(None)**

An optional hook function that you can implement to pass *auth_state* to the spawner after it has been initialized but before it starts. The *auth_state* dictionary may be set by the *.authenticate()* method of the authenticator. This hook enables you to pass some or all of that information to your spawner.

Example:

```
def userdata_hook(spawner, auth_state):
    spawner.userdata = auth_state["userdata"]

c.Spawner.auth_state_hook = userdata_hook
```

**certs_volume_name c.SystemUserSpawner.certs_volume_name = Unicode('{prefix}ssl-{username}')**

Volume name

The same string-templating applies to this as other volume names.

**client_kwargs c.SystemUserSpawner.client_kwargs = Dict()**

Extra keyword arguments to pass to the docker.Client constructor.

**cmd c.SystemUserSpawner.cmd = Command()**

The command used for starting the single-user server.

Provide either a string or a list containing the path to the startup script command. Extra arguments, other than this path, should be provided via *args*.

This is usually set if you want to start the single-user server in a different python environment (with virtualenv/conda) than JupyterHub itself.

Some spawners allow shell-style expansion here, allowing you to use environment variables. Most, including the default, do not. Consult the documentation for your spawner to verify!

**consecutive_failure_limit c.SystemUserSpawner.consecutive_failure_limit = Int(0)**

Maximum number of consecutive failures to allow before shutting down JupyterHub.

This helps JupyterHub recover from a certain class of problem preventing launch in contexts where the Hub is automatically restarted (e.g. systemd, docker, kubernetes).

A limit of 0 means no limit and consecutive failures will not be tracked.

**container_image c.SystemUserSpawner.container_image = Unicode('quay.io/jupyterhub/singleuser:5.4')**

Deprecated, use `DockerSpawner.image`.

**container_ip c.SystemUserSpawner.container_ip = Unicode('127.0.0.1')**

Deprecated, use `DockerSpawner.host_ip`

**container_name_template c.SystemUserSpawner.container_name_template = Unicode('')**

Deprecated, use `DockerSpawner.name_template`.

**container_port c.SystemUserSpawner.container_port = Int(8888)**

Deprecated, use DockerSpawner.port.

**container_prefix c.SystemUserSpawner.container_prefix = Unicode('')**

Deprecated, use DockerSpawner.prefix.

**cpu_guarantee c.SystemUserSpawner.cpu_guarantee = Float(None)**

Minimum number of cpu-cores a single-user notebook server is guaranteed to have available.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

**This is a configuration setting. Your spawner must implement support for the limit to work.** The default spawner, *LocalProcessSpawner*, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

**cpu_limit c.SystemUserSpawner.cpu_limit = Union()**

CPU limit for containers

Will set cpu_quota = cpu_limit * cpu_period

The default cpu_period of 100ms will be used, unless overridden in extra_host_config.

Alternatively to a single float, cpu_limit can also be a callable that takes the spawner as the only argument and returns a float:

**def per_user_cpu_limit(spawner):**
username = spawner.user.name cpu_limits = {'alice': 2.5, 'bob': 2} return cpu_limits.get(username, 1)

c.DockerSpawner.cpu_limit = per_user_cpu_limit

**debug c.SystemUserSpawner.debug = Bool(False)**

Enable debug-logging of the single-user server

**default_url c.SystemUserSpawner.default_url = Unicode('')**

The URL the single-user server should start in.

*{username}* will be expanded to the user's username

Example uses:

- You can set *notebook_dir* to */* and *default_url* to */tree/home/{username}* to allow people to navigate the whole filesystem from their notebook server, but still start in their home directory.
- Start with */notebooks* instead of */tree* if *default_url* points to a notebook instead of a directory.
- You can set this to */lab* to have JupyterLab start by default, rather than Jupyter Notebook.

**disable_user_config c.SystemUserSpawner.disable_user_config = Bool(False)**

Disable per-user configuration of single-user servers.

When starting the user's single-user server, any config file found in the user's $HOME directory will be ignored.

Note: a user could circumvent this if the user modifies their Python environment, such as when they have their own conda environments / virtualenvs / containers.

**env_keep c.SystemUserSpawner.env_keep = List()**

List of environment variables for the single-user server to inherit from the JupyterHub process.

This list is used to ensure that sensitive information in the JupyterHub process's environment (such as *CONFIGPROXY_AUTH_TOKEN*) is not passed to the single-user server's process.

**environment c.SystemUserSpawner.environment = Dict()**

Extra environment variables to set for the single-user server's process.

**Environment variables that end up in the single-user server's process come from 3 sources:**

- This *environment* configurable
- The JupyterHub process' environment variables that are listed in *env_keep*
- Variables to establish contact between the single-user notebook and the hub (such as JUPYTERHUB_API_TOKEN)

The *environment* configurable should be set by JupyterHub administrators to add installation specific environment variables. It is a dict where the key is the name of the environment variable, and the value can be a string or a callable. If it is a callable, it will be called with one parameter (the spawner instance), and should return a string fairly quickly (no blocking operations please!).

Note that the spawner class' interface is not guaranteed to be exactly same across upgrades, so if you are using the callable take care to verify it continues to work after upgrades!

Changed in version 1.2: environment from this configuration has highest priority, allowing override of 'default' env variables, such as JUPYTERHUB_API_URL.

**escape c.SystemUserSpawner.escape = Any(None)**

Override escaping with any callable of the form escape(str)->str

This is used to ensure docker-safe container names, etc.

The default escaping should ensure safety and validity, but can produce cumbersome strings in cases.

Set c.DockerSpawner.escape = 'legacy' to preserve the earlier, unsafe behavior if it worked for you.

Added in version 12.0.

Changed in version 12.0: Escaping has changed in 12.0 to ensure safety, but existing deployments will get different container and volume names.

**extra_create_kwargs c.SystemUserSpawner.extra_create_kwargs = Union()**

Additional args to pass for container create

For example, to change the user the container is started as:

```
c.DockerSpawner.extra_create_kwargs = {
    "user": "root" # Can also be an integer UID
}
```

The above is equivalent to `docker run --user root`.

If a callable, will be called with the Spawner as the only argument, must return the same dictionary structure, and may be async.

Changed in version 13: Added callable support.

**extra_host_config c.SystemUserSpawner.extra_host_config = Union()**

Additional args to create_host_config for container create.

If a callable, will be called with the Spawner as the only argument, must return the same dictionary structure, and may be async.

Changed in version 13: Added callable support.

`format_volume_name c.SystemUserSpawner.format_volume_name = Any(None)`

Any callable that accepts a string template and a DockerSpawner instance as parameters in that order and returns a string.

Reusable implementations should go in dockerspawner.VolumeNamingStrategy, tests should go in ...

`get_env()`

Return the environment dict to use for the Spawner.

This applies things like *env_keep*, anything defined in *Spawner.environment*, and adds the API token to the env.

When overriding in subclasses, subclasses must call *super().get_env()*, extend the returned dict and return it.

Use this to access the env in Spawner.start to allow extension in subclasses.

`get_state()`

Save state of spawner into database.

A black box of extra state for custom spawners. The returned value of this is passed to *load_state*.

Subclasses should call *super().get_state()*, augment the state returned from there, and return that state.

**Returns**
**state** – a JSONable dict of state

**Return type**
dict

`group_id Int(-1)`

If system users are being used, then we need to know their group id in order to mount the home directory with correct group permissions.

Group IDs are looked up in two ways:

1. stored in the state dict (authenticator can write here)
2. lookup via pwd

`group_overrides c.SystemUserSpawner.group_overrides = Union()`

Override specific traitlets based on group membership of the user.

This can be a dict, or a callable that returns a dict. The keys of the dict are *only* used for lexicographical sorting, to guarantee consistent ordering of the overrides. If it is a callable, it may be async, and will be passed one parameter - the spawner instance. It should return a dictionary.

The values of the dict are dicts with the following keys:

- *"groups"* - If the user belongs to *any* of these groups, these overrides are applied to their server before spawning.
- *"spawner_override"* - a dictionary with overrides to apply to the Spawner settings. Each value can be either the final value to change or a callable that take the *Spawner* instance as parameter and returns the final value. If the traitlet being overriden is a *dictionary*, the dictionary will be *recursively updated*, rather than overriden. If you want to remove a key, set its value to *None*.

### Example

The following example config will:

1. Add the environment variable "AM_I_GROUP_ALPHA" to everyone in the "group-alpha" group

2. Add the environment variable "AM_I_GROUP_BETA" to everyone in the "group-beta" group. If a user is part of both "group-beta" and "group-alpha", they will get *both* these env vars, due to the dictionary merging functionality.
3. Add a higher memory limit for everyone in the "group-beta" group.

```
c.Spawner.group_overrides = {
    "01-group-alpha-env-add": {
        "groups": ["group-alpha"],
        "spawner_override": {"environment": {"AM_I_GROUP_ALPHA": "yes"}},
    },
    "02-group-beta-env-add": {
        "groups": ["group-beta"],
        "spawner_override": {"environment": {"AM_I_GROUP_BETA": "yes"}},
    },
    "03-group-beta-mem-limit": {
        "groups": ["group-beta"],
        "spawner_override": {"mem_limit": "2G"}
    }
}
```

**property homedir**

Path to the user's home directory in the docker image.

**homedir_bind_propagation c.SystemUserSpawner.homedir_bind_propagation = Unicode('')**

Mode for bind mount propagation for home directory.

Requires docker-py 7.0.

See https://docs.docker.com/engine/storage/bind-mounts/#configure-bind-propagation

Added in version 14.

**property host_homedir**

Path to the volume containing the user's home directory on the host. Looked up from *pwd* if an empty format string or *None* has been specified.

**host_homedir_format_string c.SystemUserSpawner.host_homedir_format_string = Unicode('/home/{username}')**

Format string for the path to the user's home directory on the host. The format string should include a *username* variable, which will be formatted with the user's username.

If the string is empty or *None*, the user's home directory will be looked up via the *pwd* database.

**host_ip c.SystemUserSpawner.host_ip = Unicode('127.0.0.1')**

The ip address on the host on which to expose the container's port

Typically 127.0.0.1, but can be public interfaces as well in cases where the Hub and/or proxy are on different machines from the user containers.

Only used when `use_internal_ip = False`.

**http_timeout c.SystemUserSpawner.http_timeout = Int(30)**

Timeout (in seconds) before giving up on a spawned HTTP server

Once a server has successfully been spawned, this is the amount of time we wait before assuming that the server is unable to accept connections.

**hub_connect_url c.SystemUserSpawner.hub_connect_url = Unicode(None)**

The URL the single-user server should connect to the Hub.

If the Hub URL set in your JupyterHub config is not reachable from spawned notebooks, you can set differnt URL by this config.

Is None if you don't need to change the URL.

**hub_ip_connect c.SystemUserSpawner.hub_ip_connect = Unicode('')**

DEPRECATED since JupyterHub 0.8. Use c.JupyterHub.hub_connect_ip.

**image c.SystemUserSpawner.image = Unicode('quay.io/jupyterhub/singleuser:5.4')**

The image to use for single-user servers.

This image should have the same version of jupyterhub as the Hub itself installed.

If the default command of the image does not launch jupyterhub-singleuser, set `c.Spawner.cmd` to launch jupyterhub-singleuser, e.g.

Any of the jupyter docker-stacks should work without additional config, as long as the version of jupyterhub in the image is compatible.

**image_homedir_format_string c.SystemUserSpawner.image_homedir_format_string = Unicode('/home/{username}')**

Format string for the path to the user's home directory inside the image. The format string should include a *username* variable, which will be formatted with the user's username.

**image_whitelist c.SystemUserSpawner.image_whitelist = Union()**

Deprecated, use `DockerSpawner.allowed_images`.

**ip c.SystemUserSpawner.ip = Unicode('127.0.0.1')**

The IP address (or hostname) the single-user server should listen on.

Usually either '127.0.0.1' (default) or '0.0.0.0'. On IPv6 only networks use '::1' or '::'.

If the spawned singleuser server is running JupyterHub 5.3.0 later You can set this to the empty string '' to indicate both IPv4 and IPv6.

The JupyterHub proxy implementation should be able to send packets to this interface.

Subclasses which launch remotely or in containers should override the default to '0.0.0.0'.

Changed in version 5.3: An empty string '' means all interfaces (IPv4 and IPv6). Prior to this the behaviour of '' was not defined.

Changed in version 2.0: Default changed to '127.0.0.1', from unspecified.

**links c.SystemUserSpawner.links = Dict()**

Specify docker link mapping to add to the container, e.g.

links = {'jupyterhub': 'jupyterhub'}

If the Hub is running in a Docker container, this can simplify routing because all traffic will be using docker hostnames.

**load_state**(*state*)

Restore state of spawner from database.

Called for each user's spawner after the hub process restarts.

*state* is a dict that'll contain the value returned by *get_state* of the spawner, or {} if the spawner hasn't persisted any state yet.

Override in subclasses to restore any extra state that is needed to track the single-user server for that user. Subclasses should call super().

**mem_guarantee c.SystemUserSpawner.mem_guarantee = ByteSpecification(None)**

Minimum number of bytes a single-user notebook server is guaranteed to have available.

**Allows the following suffixes:**

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

**This is a configuration setting. Your spawner must implement support for the limit to work.** The default spawner, *LocalProcessSpawner*, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

**mem_limit c.SystemUserSpawner.mem_limit = Union()**

Maximum number of bytes a single-user notebook server is allowed to use. Allows the following suffixes:

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

If the single user server tries to allocate more memory than this, it will fail. There is no guarantee that the single-user notebook server will be able to allocate this much memory - only that it can not allocate more than this.

Alternatively to a string it can also be a callable that takes the spawner as the only argument and returns a string:

**def per_user_mem_limit(spawner):**
username = spawner.user.name ram_limits = {'alice': '4G', 'bob': '2G'} return ram_limits.get(username, '1G')

c.DockerSpawner.mem_limit = per_user_mem_limit

**mounts c.SystemUserSpawner.mounts = List()**

List of dict with keys to match docker.types.Mount for more advanced configuration of mouted volumes. As with volumes, if the default format_volume_name is in use, you can use {username} in the source or target paths, and it will be replaced with the current user's name.

**move_certs_image c.SystemUserSpawner.move_certs_image = Unicode('busybox:1.30.1')**

The image used to stage internal SSL certificates.

Busybox is used because we just need an empty container that waits while we stage files into the volume via .put_archive.

**name_template c.SystemUserSpawner.name_template = Unicode('')**

Name of the container or service: with {username}, {imagename}, {prefix}, {servername} replacements. {raw_username} can be used for the original, not escaped username (may contain uppercase, special characters). It is important to include {servername} if JupyterHub's "named servers" are enabled (`JupyterHub.allow_named_servers = True`). If the server is named, the default name_template is "{prefix}-{username}–{servername}". If it is unnamed, the default name_template is "{prefix}-{username}".

Note: when using named servers, it is important that the separator between {username} and {servername} is not a character that can occur in an escaped {username}, and also not the single escape character '-'.

**network_name c.SystemUserSpawner.network_name = Unicode('bridge')**

Run the containers on this docker network. If it is an internal docker network, the Hub should be on the same network, as internal docker IP addresses will be used. For bridge networking, external ports will be bound.

**notebook_dir c.SystemUserSpawner.notebook_dir = Unicode('')**

Path to the notebook directory for the single-user server.

The user sees a file listing of this directory when the notebook interface is started. The current interface does not easily allow browsing beyond the subdirectories in this directory's tree.

~ will be expanded to the home directory of the user, and {username} will be replaced with the name of the user.

Note that this does *not* prevent users from accessing files outside of this path! They can do so with many other means.

**oauth_client_allowed_scopes c.SystemUserSpawner.oauth_client_allowed_scopes = Union()**

Allowed scopes for oauth tokens issued by this server's oauth client.

This sets the maximum and default scopes assigned to oauth tokens issued by a single-user server's oauth client (i.e. tokens stored in browsers after authenticating with the server), defining what actions the server can take on behalf of logged-in users.

Access to the current server will always be included in this list. This property contains additional scopes. Default is an empty list, meaning minimal permissions to identify users, no actions can be taken on their behalf.

If callable, will be called with the Spawner as a single argument. Callables may be async.

**oauth_roles c.SystemUserSpawner.oauth_roles = Union()**

Allowed roles for oauth tokens.

Deprecated in 3.0: use oauth_client_allowed_scopes

**options_form c.SystemUserSpawner.options_form = Union()**

An HTML form for options a user can specify on launching their server.

The surrounding *<form>* element and the submit button are already provided.

For example:

```
Set your key:
<input name="key" val="default_key"></input>
<br>
Choose a letter:
<select name="letter" multiple="true">
  <option value="A">The letter A</option>
  <option value="B">The letter B</option>
</select>
```

The data from this form submission will be passed on to your spawner in *self.user_options*

Instead of a form snippet string, this could also be a callable that takes as one parameter the current spawner instance and returns a string. The callable will be called asynchronously if it returns a future, rather than a str. Note that the interface of the spawner class is not deemed stable across versions, so using this functionality might cause your JupyterHub upgrades to break.

**options_from_form** `c.SystemUserSpawner.options_from_form = Union()`

Interpret HTTP form data

Form data will always arrive as a dict of lists of strings. Override this function to understand single-values, numbers, etc.

This should coerce form data into the structure expected by self.user_options, which must be a dict, and should be JSON-serializeable, though it can contain bytes in addition to standard JSON data types.

This method should not have any side effects. Any handling of *user_options* should be done in *.apply_user_options()* (JupyterHub 5.3) or *.start()* (JupyterHub 5.2 or older) to ensure consistent behavior across servers spawned via the API and form submission page.

Instances will receive this data on self.user_options, after passing through this function, prior to *Spawner.start*.

Changed in version 1.0: user_options are persisted in the JupyterHub database to be reused on subsequent spawns if no options are given. user_options is serialized to JSON as part of this persistence (with additional support for bytes in case of uploaded file data), and any non-bytes non-jsonable values will be replaced with None if the user_options are re-used.

Added in version 5.3: The strings *'simple'* and *'passthrough'* may be specified to select some predefined behavior. These are the only string values accepted.

*'passthrough'* is the longstanding default behavior, where form data is stored in *user_options* without modification. With *'passthrough'*, *user_options* from a form will always be a dict of lists of strings.

*'simple'* applies some minimal processing that works for most simple forms:

- Single-value fields get unpacked from lists. They are still always strings, no attempt is made to parse numbers, etc..
- Multi-value fields are left alone.
- The default checked value of "on" for a checkbox is converted to True. This is the only non-string value that can be produced.

Example for *'simple'*:

```
{
    "image": ["myimage"],
    "checked": ["on"], # checkbox
    "multi-select": ["a", "b"],
}
# becomes
{
    "image": "myimage",
    "checked": True,
    "multi-select": ["a", "b"],
}
```

**poll_interval** `c.SystemUserSpawner.poll_interval = Int(30)`

Interval (in seconds) on which to poll the spawner for single-user server's status.

At every poll interval, each spawner's *.poll* method is called, which checks if the single-user server is still running. If it isn't running, then JupyterHub modifies its own state accordingly and removes appropriate routes from the configurable proxy.

**poll_jitter** `c.SystemUserSpawner.poll_jitter = Float(0.1)`

Jitter fraction for poll_interval.

Avoids alignment of poll calls for many Spawners, e.g. when restarting JupyterHub, which restarts all polls for running Spawners.

*poll_jitter=0* means no jitter, 0.1 means 10%, etc.

**port c.SystemUserSpawner.port = Int(0)**

The port for single-user servers to listen on.

Defaults to *0*, which uses a randomly allocated port number each time.

If set to a non-zero value, all Spawners will use the same port, which only makes sense if each server is on a different address, e.g. in containers.

New in version 0.7.

**post_start_cmd c.SystemUserSpawner.post_start_cmd = UnicodeOrFalse(False)**

If specified, the command will be executed inside the container after starting. Similar to using 'docker exec'

**post_stop_hook c.SystemUserSpawner.post_stop_hook = Any(None)**

An optional hook function that you can implement to do work after the spawner stops.

This can be set independent of any concrete spawner implementation.

**pre_spawn_hook c.SystemUserSpawner.pre_spawn_hook = Any(None)**

An optional hook function that you can implement to do some bootstrapping work before the spawner starts. For example, create a directory for your user or load initial content.

This can be set independent of any concrete spawner implementation.

This maybe a coroutine.

Example:

```
def my_hook(spawner):
    username = spawner.user.name
    spawner.environment["GREETING"] = f"Hello {username}"

c.Spawner.pre_spawn_hook = my_hook
```

**prefix c.SystemUserSpawner.prefix = Unicode('jupyter')**

Prefix for container names. See name_template for full container name for a particular user's server.

**progress_ready_hook c.SystemUserSpawner.progress_ready_hook = Any(None)**

An optional hook function that you can implement to modify the ready event, which will be shown to the user on the spawn progress page when their server is ready.

This can be set independent of any concrete spawner implementation.

This maybe a coroutine.

Example:

```
async def my_ready_hook(spawner, ready_event):
    ready_event["html_message"] = f"Server {spawner.name} is ready for {spawner.
↪user.name}"
    return ready_event

c.Spawner.progress_ready_hook = my_ready_hook
```

**pull_policy c.SystemUserSpawner.pull_policy = CaselessStrEnum('ifnotpresent')**

The policy for pulling the user docker image.

Choices:

- ifnotpresent: pull if the image is not already present (default)
- always: always pull the image to check for updates, even if it is present
- never: never perform a pull, raise if image is not present
- skip: never perform a pull, skip the step entirely (like never, but without raising when images are not present; default for swarm)

**read_only_volumes c.SystemUserSpawner.read_only_volumes = Dict()**

Map from host file/directory to container file/directory. Volumes specified here will be read-only in the container.

If format_volume_name is not set, default_format_volume_name is used for naming volumes. In this case, if you use {username} in either the host or guest file/directory path, it will be replaced with the current user's name.

**remove c.SystemUserSpawner.remove = Bool(False)**

If `True`, delete containers when servers are stopped.

This will destroy any data in the container not stored in mounted volumes.

**remove_containers c.SystemUserSpawner.remove_containers = Bool(False)**

Deprecated, use `DockerSpawner.remove`.

**run_as_root c.SystemUserSpawner.run_as_root = Bool(False)**

Run the container as root

Relies on the image itself having handling of $NB_UID and $NB_GID options to switch.

This was the default behavior prior to 0.12, but has become opt-in. This enables images to nicely map usernames to userids inside the container.

Added in version 0.12.

**server_token_scopes c.SystemUserSpawner.server_token_scopes = Union()**

The list of scopes to request for $JUPYTERHUB_API_TOKEN

If not specified, the scopes in the *server* role will be used (unchanged from pre-4.0).

If callable, will be called with the Spawner instance as its sole argument (JupyterHub user available as spawner.user).

JUPYTERHUB_API_TOKEN will be assigned the _subset_ of these scopes that are held by the user (as in oauth_client_allowed_scopes).

Added in version 4.0.

**ssl_alt_names c.SystemUserSpawner.ssl_alt_names = Union()**

List of SSL alt names (list of strings).

May be set in config if all spawners should have the same value(s), or set at runtime by Spawner that know their names.

Changed in version 5.4.1: May now be a callable. The callable will receive the Spawner as its only argument, and must return a list of strings. It may be async.

**ssl_alt_names_include_local c.SystemUserSpawner.ssl_alt_names_include_local = Bool(True)**

Whether to include *DNS:localhost*, *IP:127.0.0.1* in alt names

**start**(** (Keyword-only parameters separator (PEP 3102))*, *image=None*, *extra_create_kwargs=None*, *extra_host_config=None*)

start the single-user server in a docker container

**start_timeout c.SystemUserSpawner.start_timeout = Int(60)**

Timeout (in seconds) before giving up on starting of single-user server.

This is the timeout for start to return, not the timeout for the server to respond. Callers of spawner.start will assume that startup has failed if it takes longer than this. start should return when the server process is started and its location is known.

**tls c.SystemUserSpawner.tls = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_assert_hostname c.SystemUserSpawner.tls_assert_hostname = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_ca c.SystemUserSpawner.tls_ca = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_cert c.SystemUserSpawner.tls_cert = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_config c.SystemUserSpawner.tls_config = Dict()**

Arguments to pass to docker TLS configuration.

See docker.client.TLSConfig constructor for options.

**tls_key c.SystemUserSpawner.tls_key = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**tls_verify c.SystemUserSpawner.tls_verify = Any(None)**

Deprecated, use `DockerSpawner.tls_config` dict to set any TLS options.

**use_docker_client_env c.SystemUserSpawner.use_docker_client_env = Bool(True)**

Deprecated. Docker env variables are always used if present.

**use_internal_hostname c.SystemUserSpawner.use_internal_hostname = Bool(False)**

Use the docker hostname for connecting.

instead of an IP address. This should work in general when using docker networks, and must be used when internal_ssl is enabled. It is enabled by default if internal_ssl is enabled.

**use_internal_ip c.SystemUserSpawner.use_internal_ip = Bool(False)**

Enable the usage of the internal docker ip. This is useful if you are running jupyterhub (as a container) and the user containers within the same docker network. E.g. by mounting the docker socket of the host into the jupyterhub container. Default is `True` if using a docker network, `False` if bridge or host networking is used.

**user_id Int(-1)**

If system users are being used, then we need to know their user id in order to mount the home directory.

User IDs are looked up in two ways:

1. stored in the state dict (authenticator can write here)

2. lookup via pwd

**property volume_binds**

The second half of declaring a volume with docker-py happens when you actually call start(). The required format is a dict of dicts that looks like:

```
{
    host_location: {'bind': container_location, 'ro': True}
}
```

**property volume_mount_points**

Volumes are declared in docker-py in two stages. First, you declare all the locations where you're going to mount volumes when you call create_container.

Returns a list of all the values in self.volumes or self.read_only_volumes.

**volumes c.SystemUserSpawner.volumes = Dict()**

Map from host file/directory to container (guest) file/directory mount point and (optionally) a mode. When specifying the guest mount point (bind) for the volume, you may use a dict or str. If a str, then the volume will default to a read-write (mode="rw"). With a dict, the bind is identified by "bind" and the "mode" may be one of "rw" (default), "ro" (read-only), "z" (public/shared SELinux volume label), and "Z" (private/unshared SELinux volume label).

If format_volume_name is not set, default_format_volume_name is used for naming volumes. In this case, if you use {username} in either the host or guest file/directory path, it will be replaced with the current user's name.

## 1.7 Changes in DockerSpawner

### 1.7.1 Changes in DockerSpawner

For detailed changes from the prior release, click on the version number, and its link will bring up a GitHub listing of changes. Use `git log` on the command line for details.

#### Unreleased

#### 14

#### 14.0.0 2024-02-12

14.0.0 is a small release. It is designated a major revision because it drops support for Python < 3.9 and JupyterHub < 4.

(full changelog)

#### API and Breaking Changes

- require Python 3.9, jupyterhub 4, unpin pytest-asyncio #530 (@minrk)

#### New features added

- support for bind propagation #526 (@jannefleischer, @minrk)

### Enhancements made

- compact debug log into single line #521 (@haobibo, @minrk)

### Bugs fixed

- Support options_from_form config #525 (@akhmerov, @minrk)

### Maintenance and upkeep improvements

- update some test dependencies #531 (@minrk)
- update test versions in ci #529 (@minrk)
- update internal-ssl test for current docker compose #527 (@minrk)

### Documentation improvements

- Document c.DockerSpawner.mounts #510 (@zhangnew, @minrk)

### Contributors to this release

The following people contributed discussions, new ideas, code and documentation contributions, and review. See our definition of contributors.

(GitHub contributors page for this release)

@akhmerov (activity) | @consideRatio (activity) | @haobibo (activity) | @jannefleischer (activity) | @manics (activity) | @minrk (activity) | @yuvipanda (activity) | @zhangnew (activity)

## 13

### 13.0 2023-11-21

(full changelog)

13.0 Fixes security vulnerability GHSA-hfgr-h3vc-p6c2, which allowed authenticated users to spawn arbitrary images unless `DockerSpawner.allowed_images` was specified.

### API and Breaking Changes

- Add and require `DockerSpawner.allowed_images='*'` to allow any image to be spawned via `user_options`. (GHSA-hfgr-h3vc-p6c2)
- Remove deprecated, broken hub_ip_connect #499 (@minrk)
- Require python 3.8+ and jupyterhub 2.3.1+ #488 (@consideRatio, @minrk)

### New features added

- Switch default image to quay.io #504 (@yuvipanda, @minrk, @manics)
- allow extra_host_config and extra_create_kwargs to be callable #500 (@minrk)

### Enhancements made

- Merge host config/create_kwargs #501 (@minrk)

### Bugs fixed

- update object_name with current image #466 (@floriandeboissieu, @minrk)
- Fix imagename not to include letter ':' #464 (@yamaton, @minrk)
- clear object_id when removing object #447 (@minrk, @manics)

### Maintenance and upkeep improvements

- pre-commit: add pyupgrade and autoflake, simplify flake8 config #489 (@consideRatio, @minrk)
- Require python 3.8+ and jupyterhub 2.3.1+ #488 (@consideRatio, @minrk)
- Add dependabot.yaml to bump github actions #487 (@consideRatio, @minrk)
- Update release workflow and RELEASE.md, set version with tbump #486 (@consideRatio, @minrk)
- Refresh test workflow and associated config, accept podmain test failure for now #485 (@consideRatio, @minrk)
- Use python 3.11 on RTD #482 (@minrk)
- Add test strategy for JupyterHub v3.1.1 #479 (@Sheila-nk, @GeorgianaElena, @minrk)
- test options_form and escape #468 (@Sheila-nk, @minrk)
- test callable allowed_images and host_ip #467 (@Sheila-nk, @minrk)
- Test jupyterhub2 #443 (@manics, @minrk)

### Documentation improvements

- Add extra_create_kwargs example, plus docs readability improvements #493 (@matthewwiese, @manics)
- update versions in swarm example #454 (@minrk, @GeorgianaElena)
- add generate-certs service to internal-ssl example #446 (@minrk, @manics)
- Add Podman to docs #444 (@manics, @minrk)

### Contributors to this release

The following people contributed discussions, new ideas, code and documentation contributions, and review. See our definition of contributors.

(GitHub contributors page for this release)

@consideRatio (activity) | @floriandeboissieu (activity) | @gatoniel (activity) | @GeorgianaElena (activity) | @manics (activity) | @matthewwiese (activity) | @minrk (activity) | @Sheila-nk (activity) | @yamaton (activity) | @yuvipanda (activity)

## 12

### 12.1 2021-07-22

(full changelog)

### Enhancements made

- support cpu limit via cpu_quota / cpu_period #435 (@minrk)
- Log post_start exec output #427 (@minrk)
- Allow to specify a callable for `mem_limit` #420 (@zeehio)

### Maintenance and upkeep improvements

- update release steps for main branch #434 (@minrk)
- more debug info from docker when tests fail #433 (@minrk)

### Contributors to this release

(GitHub contributors page for this release)

@1kastner | @manics | @minrk | @welcome | @zeehio

## 12.0 2021-03-26

This is a big release!

Several bugs have been fixed, especially in SwarmSpawner, and more configuration options added.

### New escaping scheme

In particular, the biggest backward-incompatible change to highlight is the container (and volume) name escaping scheme now produces DNS-safe results, which matches the behavior of kubespawner. This is a stricter subset of characters than docker containers strictly require, but many features don't work right without it. The result is for certain user names and/or server names, their container and/or volume names will change. Upgrading existing deployments will result in disconnecting these users from their running containers and volumes, which means:

- if there are running users across the upgrade, some containers will need to be manually stopped
- some volumes may need to be renamed, which docker doesn't support, but can be done:

```
docker volume create --name $new_volume
docker run --rm -it -v $old_volume:/from -v $new_volume:/to alpine ash -c "cd /from␣
↪; cp -av . /to"
docker volume rm $old_volume
```

The main differences are:

- The escape character is `-` instead of `_` which means `-` cannot itself be a safe character and must be escaped to `-2d`
- Uppercase characters are now escaped (normalizing to lowercase at the username level is common)

So affected usernames are those with `-` or uppercase letters, or any that already needed escaping.

You can restore the pre-12.0 behavior with:

```
c.DockerSpawner.escape = "legacy"
```

### SystemUserSpawner.run_as_root

Another security-related change is the addition of `SystemUserSpawner.run_as_root`. Prior to 12.0, SystemUserSpawner always ran as root and relied on the container to use $NB_USER and $NB_UID to "become" the user. This behavior meant that user containers based on images that lacked this behavior would all run as root. To address this, `run_as_root` behavior is now opt-in

All changes are detailed below.

(full changelog)

### New features added

- apply template formatting to all of extra_create_kwargs, extra_host_config #409 (@minrk)
- Add mounts option for more advanced binds #406 (@minrk)
- Add JUPYTER_IMAGE_SPEC to env. #316 (@danielballan)
- Added post_start_cmd #307 (@mohirio)

### Enhancements made

- Use default cmd=None to indicate using the image command #415 (@minrk)
- add 'skip' option for pull_policy #411 (@minrk)
- Add auto_remove to host_config #318 (@jtpio)
- Make default name_template compatible with named servers. #315 (@danielballan)
- SystemUserSpawner: Pass group id to the container #304 (@zeehio)
- Allow lookup of host homedir via `pwd` #302 (@AdrianoKF)

### Bugs fixed

- (PATCH) SwarmSpawner, InvalidArgument: Incompatible options have been provided for the bind type mount. #419 (@cmotadev)
- Make sure that create_object() creates the service task #396 (@girgink)
- avoid name collisions when using named servers #386 (@minrk)
- Fix issue with pulling images from custom repos that contain a port #334 (@raethlein)

### Maintenance and upkeep improvements

- async/await #417 (@minrk)
- stop building docs on circleci #387 (@minrk)
- Test with latest jh #379 (@GeorgianaElena)
- Fix RTD build #378 (@GeorgianaElena)
- Add release instructions and Travis deploy #377 (@GeorgianaElena)
- Fix tests #374 (@GeorgianaElena)
- Add README badges #356 (@GeorgianaElena)

### Documentation improvements

- Update swarm example #418 (@minrk)
- improve robustness of internal-ssl example #416 (@minrk)
- update versions in docker-image docs #410 (@minrk)
- Add GitHub Action readme badge #408 (@consideRatio)
- Switch CI to GitHub actions #407 (@minrk)
- touch up simple example #405 (@minrk)
- add example for selecting arbitrary image via options_form #401 (@minrk)
- Typo fix in the docs #380 (@jtpio)
- Add docs #375 (@GeorgianaElena)
- Fix dead link in doc #350 (@JocelynDelalande)
- Fix project name typo #339 (@kinow)

### API and Breaking Changes

- Make escaping DNS-safe #414 (@minrk)
- add SystemUserSpawner.run_as_root #412 (@minrk)
- Rename DockerSpawner.image_whitelist to allowed_images #381 (@GeorgianaElena)

### Contributors to this release

(GitHub contributors page for this release)

@1kastner | @AdrianoKF | @anmtan | @AnubhavUjjawal | @belfhi | @bellackn | @bjornandre | @blacksailer | @cblomart | @choldgraf | @cmotadev | @cmseal | @co60ca | @consideRatio | @cyliu0204 | @danielballan | @danlester | @efagerberg | @gatoniel | @GeorgianaElena | @girgink | @hugoJuhel | @jameholme | @jamesdbrock | @JocelynDelalande | @jtpio | @kinow | @kkr78 | @ltupin | @manics | @mathematicalmichael | @meeseeksmachine | @minrk | @missingcharacter | @mohirio | @myurasov | @nazeels | @nmvega | @nuraym | @parente | @raethlein | @sabuhish | @sangramga | @support | @TimoRoth | @vlizanae | @welcome | @Wildcarde | @willingc | @wwj718 | @yuvipanda | @z3ky | @zeehio | @zhiyuli

## 0.11

### 0.11.1 - 2019-04-25

- Fix some compatibility issues
- Add more states to be recognized as pending for SwarmSpawner

### 0.11.0 - 2019-03-01

### New features:

- Support selecting docker spawner via JupyterHub 1.0's entrypoints:

```
c.JupyterHub.spawner_class = 'docker' # or 'docker-swarm' or 'docker-system-user'
```

- Support total internal SSL encryption with JupyterHub 1.0

- Add new `DockerSpawner.pull_policy` to configure pulling of images. Values are inspired by Kubernetes, and case-insensitive. Can be any of "IfNotPresent" (new default), "Always", and "Never" (pre-0.11 behavior). Now the image will be pulled by default if it is not present.
- Add `image_whitelist` configuration which, if set, defines a default options form for users to pick the desired image. `image_whitelist` is a dict of `{'descriptive key': 'image:tag'}`.
- Add `SwarmSpawner.extra_placement_spec` configuration for setting service placement

#### Fixes:

- Slow startup in SwarmSpawner could be treated as failures.

### 0.10

#### 0.10.0 - 2018-09-03

- Add `dockerspawner.SwarmSpawner` for spawning with Docker Swarm
- Removed deprecated `extra_start_kwargs`
- `host_ip` is configurable
- Added `container_name_template` configuration for custom container naming

### 0.9

#### 0.9.1 - 2017-08-23

- Fix typo which would cause using the deprecated `.hub_ip_connect` configuration with JupyterHub 0.8 to crash instead of warn in 0.9.0.

#### 0.9.0 - 2017-08-20

0.9 cleans up some configuration and improves support for the transition from JupyterHub 0.8 to 0.9. It also reduces some of the special arguments and env handling, allowing for more consistency with other Spawners, and fewer assumptions about the image that will be used by the Spawner.

The following is a minimal Dockerfile that works with DockerSpawner 0.9 and JupyterHub 0.7.2:

```
FROM python:3.6
RUN pip install \
    jupyterhub==0.8.0 \
    'notebook==5.0.0'

# Don't want to run as root!
RUN useradd -m jovyan
ENV HOME=/home/jovyan
WORKDIR $HOME
USER jovyan

CMD ["jupyterhub-singleuser"]
```

In particular:

- any image with the correct version of JupyterHub installed (it should match JupyterHub) should work with DockerSpawner.

- any image **based on one of the jupyter/docker-stacks** should work with SystemUserSpawner. There is no longer any need for the `jupyterhub/systemuser` docker image.
- The jupyterhub/singleuser image is now built from the JupyterHub repo, not this one.
- `jupyterhub/systemuser` image is deprecated. `jupyterhub/systemuser` launches containers as root and relies on the `NB_UID` and `NB_GID` handling of `jupyter/docker-stacks` to setup the user.
- The default `jupyterhub/singleuser` image has tags for JupyterHub versions, to ensure image compatibility with JupyterHub. The default image is now `jupyterhub/singleuser:x.y`, where `x.y` is the major.minor version of the current JupyterHub instance, so DockerSpawner should work by default with JupyterHub 0.7 or 0.8 without needing to specify the image.
- `Spawner.cmd` config is now supported, which can be used to override the CMD arg. By default, the image's CMD is used.
- `Spawner.get_args()` behavior is now properly inherited, and args are appended to the spawn command as in other Spawners.
- Arguments are now passed via `.get_args()` as in the base Spawner, rather than custom environment variables which user images had to support.
- `DockerSpawner.hub_ip_connect` is deprecated when running with JupyterHub 0.8. Use `JupyterHub.hub_connect_ip` instead, which is used by all Spawners.

Some configuration has been cleaned up to be clearer and more concise:

- `DockerSpawner.container_image` is deprecated in favor of `DockerSpawner.image`.
- `DockerSpawner.container_port` is deprecated in favor of existing `Spawner.port`.
- Inaccurately named `DockerSpawner.container_ip` is deprecated in favor of `DockerSpawner.host_ip` because it configures the host IP forwarded to the container.

### 0.8 - 2017-07-28

- experimental fixes for running on Windows
- added `DockerSpawner.client_kwargs` config to passthrough to the `docker.Client` constructor
- workaround bug where Docker can report ports as strings
- bump docker dependency to new `docker` package from `docker-py`

### 0.7 - 2017-03-14

- Only need to set `DockerSpawner.network_name` to run on a docker network, instead of setting `host_config`, `network_name`, and `use_internal_ip` separately.
- Set `mem_limit` on `host_config` for docker API 1.19
- Match start keyword args on SystemUserSpawner to DockerSpawner

### 0.6 - 2017-01-02

- Add `DockerSpawner.format_volume_name` for custom naming strategies for mounted volumes.
- Support `mem_limit` config introduced in JupyterHub 0.7.
- Support `will_resume` flag necessary for resuming containers with `DockerSpawner.remove_containers = False` and JupyterHub 0.7 (requires JupyterHub 0.7.1).

### 0.5 - 2016-10-05

- return ip, port from `DockerSpawner.start`, for future-compatibility (setting ip, port directly is deprecated in JupyterHub 0.7).
- Support `{username}` in volume_mounts

### 0.4 - 2016-06-07

- get singleuser script from jupyterhub 0.6.1 (0.7 will require jupyterhub package to run singleuser script)
- `get_ip_and_port()` is a tornado coroutine, rather than an asyncio coroutine, for consistency with the rest of the code.
- more configuration for ports and mounts

### 0.3 - 2016-04-22

- Moved to jupyterhub org (`jupyterhub/singleuser`, `jupyterhub/systemuser` on Docker)
- Add `rebase-singleuser` tool for building new single-user images on top of different bases
- Base default docker images on `jupyter/scipy-notebook` from jupyter/docker-stacks
- Fix environment setup to use `get_env` instead of `_env_default` (Needed for JupyterHub 0.5)

### 0.2 - 2016-02-16

- Add `DockerSpawner.links`
- Use HostIp from docker port output
- Make user home string template configurable

### 0.1 - 2016-02-03

First release

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## d

# INDEX

## D



## E



## F



## G

## H



## I



## L



## M



## N

## O



## P



## R

## S



## T



## U

## V



## W