# deck-chores

***Release 1.1.5***

**Aug 30, 2020**

# Contents

Contents:

CHAPTER 1

# Usage

## 1.1 Invocation

### 1.1.1 On a single host

Usually you would run `deck-chores` in a container:

```
$ docker run --rm -v /var/run/docker.sock:/var/run/docker.sock funkyfuture/deck-
↪chores:1
```

**Note:** There's a manifest on the Docker Hub that maps images to builds targeting `amd64`, `amd64` and `armv7l` architectures. Thus you don't need to specify any platform indicator, the Docker client will figure out which one is the proper image to pull.

Likewise, docker-compose can be used with such configuration:

```
version: "3.7"

services:
  officer:
    image: funkyfuture/deck-chores:1
    restart: unless-stopped
    environment:
      TIMEZONE: Asia/Tel Aviv
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

You could also install `deck-chores` from the Python Package Index with `pip` or `pipx` (recommended):

```
$ pipx install deck-chores
```

and then run it:

```
$ deck-chores
```

Now one instance of `deck-chores` is running and will handle all job definitions that it discovers on containers that run on the Docker host.

### 1.1.2 In a Docker Swarm

`deck-chores` can be run in a Docker Swarm cluster, but it must be deployed on all nodes and it cannot restrict jobs to be run in only one of the containers that manifest a service. This would be a suitable stack definition:

```
version: "3.7"

services:
  officer:
    image: funkyfuture/deck-chores:1
    deploy:
      mode: global
    environment:
      TIMEZONE: Europe/Berlin
      # it isn't guaranteed that service or job options don't override this:
      DEFAULT_FLAGS: noservice
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

It can be deployed with:

```
$ docker stack deploy --compose-file docker-compose.yml deck-chores
```

Now one instance of `deck-chores` is running on each Swarm node and each will handle all job definitions that it discovers on containers that run on the same Swarm node. No instance is aware of the events and containers on other nodes.

## 1.2 Caveats & Tips

**Caution:** There's yet no way to distinguish container events that happen during an **image build** from others (#6 and #15211). Thus when an image is built, *deck-chores* will register and remove jobs on all intermediate containers following labels that define jobs. It would possibly trigger these jobs, which might lead to a corrupted build. You can avoid this risk by building images on a host that is not observed by *deck-chores* or by pausing it during image builds. Another alternative could be using Podman to build images.

### 1.2.1 Containers without an enduring main process

If the container is supposed to only run the scheduled commands and not a main process, use a non-stopping no-op command as main process like in this snippet of a `docker-compose.yml` file:

```
services:
  neverending:
    # ...
    command: tail -f /dev/null
    labels:
```

(continues on next page)

(continued from previous page)

```
      deck-chores.daily_job.command: daily_command ...
      deck-chores.daily_job.interval: daily
```

### 1.2.2 Making jobs' output available to `docker logs` of the executing container

Docker captures the output of the first process in a container as logged data. In order to capture the output of a job's command as well, its output needs to be redirected to the main process' `stdout` or `stderr`, e.g. with by redirecting a command's output with a shell:

```
deck-chores.a_job.command: sh -c "/usr/local/bin/job_script.sh &> /proc/1/fd/1"
```

### 1.2.3 Listing all registered jobs

Information, including the next scheduled execution, about the registered jobs of a deck-chores instance can be logged at once by sending `SIGUSR1` signal to the process, e.g. to one that runs in a container:

```
docker kill --signal USR1 <CONTAINER>
```

## 1.3 Job definitions

Job definitions are parsed from a container's metadata aka labels. A label's key must be in the namespace defined by *`LABEL_NAMESPACE`* (default: `deck-chores`) to be considered. A job has its own namespace that holds all its attributes. Thus an attribute's key has usually this schema:

```
$LABEL_NAMESPACE.<job name>.<job attribute>
```

An exception is a job's `env` namespace that is structured like this:

```
$LABEL_NAMESPACE.<job name>.env.<variable name>
```

The *job name* `options` cannot be used as it is reserved for setting *Container-scoped configuration*.

A job name can consist of lower-case letters, digits and dashes.

The following attributes are available:

| Attribute | Description |
|---|---|
| command | the command to run |
| cron | a *cron* definition |
| date | a *date* definition |
| env | this namespace holds environment variables that are set on the command's execution context |
| interval | an *interval* definition |
| jitter | the maximum length of a random delay before each job's execution (in conjunction with a *cron* or *interval* trigger); can be either a number that define seconds or a number with a subsequent time unit indicator like the *interval* trigger |
| max | the maximum of simultaneously running command instances, defaults to *DEFAULT_MAX* |
| timezone | the timezone that the trigger relates to, defaults to *TIMEZONE* |
| user | the user to run the command; see *the user option* for details regarding the defaults |
| workdir | the working directory when the command is executed |

The attribute `command` and one of `cron`, `date` or `interval` are *required* for each job.

Example snippet from a `docker-compose.yml` file:

```
services:
  web:
    # ...
    labels:
      deck-chores.clear-caches.command: drush cc all
      deck-chores.clear-caches.interval: daily
      deck-chores.clear-caches.user: www-data
      deck-chores.clear-caches.env.ENVIRONMENT: production
```

Or baked into an image:

```
LABEL deck-chores.clear-caches.command="drush cc all" \
      deck-chores.clear-caches.interval="daily" \
      deck-chores.clear-caches.user="www-data" \
      deck-chores.clear-caches.env.ENVIRONMENT="production"
```

## 1.4 Job triggers

### 1.4.1 cron

cron triggers allow definitions for repeated run times like for the well-known *cron* daemon. In contrast to the classic, the sequence of fields is flipped, starting with the greatest unit on the left. The fields are separated by spaces, missing fields are filled up with `*` on the left.

The fields from left to right define:

- `year`
- `month`
- `day` (of month)

- `week` (of year)
- `day_of_week`
- `hour`
- `minute`
- `second`

See APScheduler's documentation for details on its versatile expressions.

#### Examples

```
* * * * * */3 0 0  # run on all hours dividable by 3
*/3 0 0            # as shortened expression
* * * * 6 1 0 0    # run every Sunday at 1:00
6 1 0 0            # as shortened expression
sun 1 0 0          # as 'speaking' variant
* * * * * 1-4 0 0  # run daily at 1:00, 2:00, 3:00 and 4:00
1-4 0 0            # as shortened expression
```

### 1.4.2 date

A one-time trigger that is formatted as `YYYY-MM-DD [HH:MM:SS]`.

An omitted time is interpreted as `0:00:00`. Note that times must include a seconds field.

### 1.4.3 interval

This trigger defines a repetition by a fixed interval. It can either be a string where time units follow numbers or a sequence of numbers that qualify time units by order.

In the first form the numbers can be decimal fractions and the time units are determined by the first letter of a token as **w**eek, **d**ay, **h**our, **m**inute or **s**econd.

In the anonymous form the interval is added up by the fields *weeks*, *days*, *hours*, *minutes* and *seconds* in that order. Possible field separators are `.`, `:`, `/` and spaces. Missing fields are filled up with `0` on the left.

#### Examples

```
28 Days       # run every 4 weeks
4 wookies     # run every 4 weeks
42s 0.5d      # run every twelve hours and 42 seconds
42:00:00      # run every fourty-two hours
100/00:00:00  # run every one hundred days
```

There are also the convenience shortcuts `weekly`, `daily`, `hourly`, `every minute` and `every second`.

---

**Note:** Though it uses the same units of measurement, an interval is different from a recurring point in time of a specific calendar system, it describes the time *between* two events. Hence you should expect a job that is defined with this type of trigger to run the defined time *after* the job has been registered. To define a recurring point in time, see the *cron* trigger.

---

**Caution:** Mind that `deck-chores` doesn't track jobs' status when they are removed from the scheduler and doesn't persist any data between its invocations. Any such event would therefore reset the next scheduled run time of a job. Depending on a system's usage this is more or less likely to happen. For longer intervals, a *cron* trigger would therefore be preferable.

## 1.5 Container-scoped configuration

### 1.5.1 user

A user that shall run *all* jobs for a container can be set with a label name of this form:

```
$LABEL_NAMESPACE.options.user
```

The option can also be defined for an image and is considered when the `image` *flag* is set. If this option is not set, Docker uses the user that was specified with the `--user` option on container creation or falls back to the one defined in the underlying image.

### 1.5.2 flags

Option flags control *deck-chores*'s behaviour with regard to the labeled container and override the setting of *DEFAULT_FLAGS*. The schema for a flags label name is:

```
$LABEL_NAMESPACE.options.flags
```

Options are set as comma-separated list of flags. An option set by *DEFAULT_FLAGS* can be unset by prefixing with `no`.

These options are available:

**image**
: Job definitions in the container's basing image labels are also parsed while container label keys override these.

**service**
: Restricts jobs to one container of those that are identified with the same service.

  See *SERVICE_ID_LABELS* regarding service identity.

## 1.6 Environment variables

deck-chore's behaviour is defined by these environment variables:

**CLIENT_TIMEOUT**
: The timeout for responses from the Docker daemon in seconds without unit indicator. The default is imported from *docker-py*.

**CONTAINER_CACHE_SIZE**
: default: `128`

  The size of caches that save immutable container properties, like the parsed and possibly absent job definitions. Since memory is cheap and so are the stored objects, increase this when you have a lot of containers floating around to reduce latency.

**DOCKER_HOST**
default: `unix://var/run/docker.sock`

The URL of the Docker daemon to connect to.

**DEBUG**
default: `no`

Log debugging messages, enabled by `on`, `true` or `yes`.

**DEFAULT_FLAGS**
default: `image,service`

The default for a job option's *flags* attribute.

**DEFAULT_MAX**
default: `1`

The default for a job's `max` attribute.

**JOB_POOL_SIZE**
default: `10`

The pool size of job executors defines the maximum number of jobs that can run at the same time.

**LABEL_NAMESPACE**
default: `deck-chores`

The label namespace to look for job definitions and container options.

**LOG_FORMAT**
default: `{asctime}|{levelname:8}|{message}`

Pattern that formats log record attributes.

**SERVICE_ID_LABELS**
default: `com.docker.compose.project,com.docker.compose.service`

A comma-separated list of container labels that identify a unique service with possibly multiple container instances. This has an impact on how the *`service`* option behaves.

**TIMEZONE**

default: `UTC`

The job scheduler's timezone and the default for a job's `timezone` attribute.

### 1.6.1 TLS options

**ASSERT_HOSTNAME**
default: `no`

Enabled by `on`, `true` or `yes`.

**SSL_VERSION**
default: `TLS` (selects the highest version supported by the client and the daemon)

For other options see the names provided by Python's ssl library prefixed with `PROTOCOL_`.

Authentication related files are expected to be available at `/config/ca.pem`, `/config/cert.pem` respectively `/config/key.pem`.

CHAPTER 2

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 2.1 Types of Contributions

### 2.1.1 Report Bugs

If you run into problems, make sure you are running the latest image and run it with *DEBUG* set to `true`.

Report bugs at https://github.com/funkyfuture/deck-chores/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Your used Docker version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 2.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 2.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 2.1.4 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/funkyfuture/deck-chores/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 2.2 Get Started!

Ready to contribute? Here's how to set up *deck-chores* for local development.

1. Fork the *deck-chores* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/deck-chores.git
```

3. Install your local copy and developemnt tools into a virtualenv. Assuming you have pew installed, this is how you set up your fork for local development:

```
$ cd deck-chores
$ pew new -p $(which python) -a $(pwd) deck-chores
$ pip install -r requirements-dev.txt
$ python setup.py develop
```

4. Create a branch for the scope of your issue or feature:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

**5a. When you're done making changes, reformat the code with black and check** that your changes pass flake8 and the tests:

```
$ make black
$ make test
```

5b. If you want to run a container for testing purposes:

```
$ make run-dev
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push
```

7. Submit a pull request through the GitHub website.

## 2.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. The code must be formatted with black (see 6. above).
3. If the pull request adds functionality, the docs should be updated.

# CHAPTER 3

# History

Maintenance releases are not mentioned here, they update all dependencies and trigger complete rebuilds of the container images.

## 3.1 1.0 (2020-03-27)

- *new*: maintenance release automation

## 3.2 1.0-rc1 (2020-02-16)

This release candidate for the final version brings improved documentation, logging, a lot of code cleanup and these notable changes:

- *new*: jobs' container assignments and states are properly adjusted with regards to other instances of a service's state
- *new*: `deck-chores`' cache sizes for container properties can be controlled with `CONTAINER_CACHE_SIZE`
- *new*: the environment variable `JOB_POOL_SIZE` can be used to adapt the job executors pool size
- *new*: images are build for `arm64` (aka `aarch64`) architectures

All previously deprecated options have been removed.

## 3.3 0.3.1 (2019-03-02)

- *fix*: relax interpreter constraint for installations on rtfd.io

## 3.4 0.3 (2019-01-06)

- *fix*: log the version at startup, not its variable name

## 3.5 0.3-rc1 (2018-12-18)

- *new*: the container configuration `options.user` allows to set an executing user for all jobs that don't define one, can also be set on an image (#5)
- *new*: environment variables for a job can be set in a job's `env` namespace (#41)
- *new*: a job's `workdir` attribute can be used to set the working directory (#42)
- *new*: cron and interval triggers can be configured to delay randomly with the `jitter` option (#43)
- *new*: interval triggers and the jitter option can be defined with strings containing time units
- *removed*: the `DEFAULT_USER` environment variable is removed (#17)
- *removed*: parsing of environment variables `ASSERT_FINGERPRINT` and `DOCKER_DAEMON`
- *changed*: the container configuration `options` is moved to `options.flags`
- *changed*: the environment variable `DEFAULT_OPTIONS` is renamed to `DEFAULT_FLAGS`
- *changed*: upgraded base image
- *changed*: upgraded used Cerberus version
- *changed*: requires Python 3.7
- *fix*: includes the `tzdata` package in the image (#33)
- *fix*: add jobs as paused for paused containers on startup
- *refactoring*: uses the Python Docker SDK 3.5 (#31)

## 3.6 0.2 (2018-02-23)

- *new*: documentation how to run scheduled jobs only (#25 by @binnisb)
- *fix*: documentation on cron triggers (#27 by @alpine-digger)

## 3.7 0.2-rc3 (2017-12-23)

- *changed*: arm builds base on python:3.6-alpine that are executed on an ARMv7l architecture
- *changed*: Updated dependencies *APScheduler* and *docker-py*

## 3.8 0.2-rc2 (2017-08-05)

- *changed*: arm builds base on arm32v6/python
- *changed*: therefore `arm32v6` replaces the `arm`-suffix in image tags
- *changed*: there are no more images that get tagged with `latest-$architecture`

## 3.9 0.2-rc1 (2017-07-01)

- *refactoring*: uses the Python Docker SDK 2 (#14)
- *removed*: `ASSERT_FINGERPRINT` environment variable
- *renamed*: `DOCKER_DAEMON` to `DOCKER_HOST` to comply with the SDK
- *fix*: check on fixed labels (#18 by @aeri4list)
- documentation updates

## 3.10 0.1 (2017-03-02)

- *fix*: docker-py returns `None` for labels of images that were created with older Docker versions (#7)

## 3.11 0.1.beta3 (2017-01-22)

- *new*: there's now a build for arm architectures
- *new*: an architecture agnostic manifest is pushed to the image registry for release images

## 3.12 0.1.beta2 (2016-12-08)

- *new:* set log format per :envvar:LOG_FORMAT
- *new:* an options label to set behavioural flags
- *new:* containers can be identified as a service by configurable labels
- *new:* job definitions for further containers of a service are ignored (default, opt-out can be configured)
- *new:* image labels can also be parsed for job definitions (default, opt-out can be configured)

## 3.13 0.1.beta1 (2016-12-04)

- First release with full documentation

CHAPTER 4

# deck-chores

**A job scheduler for Docker containers, configured via container labels.**

- Documentation: https://deck-chores.readthedocs.io
- Image repository: https://hub.docker.com/r/funkyfuture/deck-chores
- Code repository: https://github.com/funkyfuture/deck-chores
- Issue tracker: https://github.com/funkyfuture/deck-chores/issues
- Free software: ISC license

## 4.1 Features

- define regular jobs to run within a container context with container and optionally with image labels
- use date, interval and cron-like triggers
- set a maximum of simultaneously running instances per job
- restrict job scheduling to one container per service
- multi-architecture image supports `amd64`, `arm64` and `armv7l` platforms, no emulator involved

## 4.2 Example

Let's say you want to dump the database of a Wordpress once a day. Here's a `docker-compose.yml` that defines a job that will be handled by *deck-chores*:

```
version: "3.7"

services:
  wordpress:
    image: wordpress
  mysql:
    image: mariadb
    volumes:
      - ./database_dumps:/dumps
    labels:
      deck-chores.dump.command: sh -c "mysqldump --all-databases > /dumps/dump-$
↪$(date -Idate)"
      deck-chores.dump.interval: daily
```

It is however recommended to use scripts with a proper shebang for such actions. Their outputs to `stdout` and `stderr` as well as their exit code will be logged by *deck-chores*.

## 4.3 Maintenance

The final release is supposed to receive monthly updates that includes updates of all updateable dependencies. If one is skipped, don't worry. When a second maintenance release is skipped, feel free to open an issue to ask what the status is.

You can always build images upon an up-to-date base image with:

```
make build
```

## 4.4 Limitations

When running on a cluster of Docker Swarm nodes, each `deck-chores` instance can only observe the containers on the node it's running on, and hence only restrict to run one job per service within the node's context.

## 4.5 Acknowledgements

It wouldn't be as charming to write this piece of software without these projects:

- APScheduler for managing jobs
- cerberus for processing metadata
- docker-py for Docker interaction
- flake8, mypy, pytest and tox for testing
- Python

## 4.6 Authors

- Frank Sachsenheim (maintaining)
- aeri4list

- alpine-digger
- Brynjar Smári Bjarnason

# Index