devops-utils Documentation

Release 0.1.0

gimoh

January 07, 2016

Contents

Docker image containing a set of utilities handy in a devops style environment.

- built in utilities:
  - Ansible
  - Fabric
  - docker-machine
  - IPython, ptpython and konch
- external runner which wraps `docker run` to make it look like the utilities are installed on the host
- extensible startup process allowing derived images to customise options and the runtime environment

CHAPTER 1

Contents:

1.1 Installation

The image can be run directly, but also contains an external runner program that wraps the `docker run` invocation to expose the utilities directly. Running the image with `install` parameter and a host directory mounted on `/target` will install the runner and appropriate links:

```
docker run -v $HOME/.local/bin:/target --rm gimoh/devops-utils install
```

Replace `$HOME/.local/bin` with a directory where you want to place the runner. The result will be along those lines:

```
devops-utils
ansible-galaxy -> devops-utils
ansible-doc -> devops-utils
ansible-vault -> devops-utils
ansible-playbook -> devops-utils
fab -> devops-utils
ansible -> devops-utils
```

By default the runner will invoke `docker run` with the `gimoh/devops-utils` image, if you want to use another name, e.g. when working with a derived image, you can override it:

```
docker run -v $HOME/.local/bin:/target --rm gimoh/devops-utils \
  install --image-name=$USER/devops-utils
```

You can also install the runner without the symlinks by passing `--no-link` option, and you can override the target runner name by passing `--runner-name=NAME` option. To see all available install options, run:

```
docker run --rm gimoh/devops-utils install --help
```

1.2 Usage

1.2.1 Derived Image

First usage scenario is when you build a derived image containing your source (e.g. Ansible playbooks, etc.). An example `Dockerfile`:

```
FROM gimoh/devops-utils
```

```
ADD . /opt/app
WORKDIR /opt/app
```

You may also want to add Ansible roles, or python modules, e.g.:

```
FROM gimoh/devops-utils

ADD reqs-*[lt] /opt/app/
RUN ansible-galaxy install --role-file /opt/app/reqs-ansible.yml
RUN pip install --requirement /opt/app/reqs-py.txt
ADD . /opt/app
WORKDIR /opt/app
```

Then to use:

```
ansible-playbook -i hosts.ini your-playbook.yml
# or
fab -l
```

See also *Extending* which may be useful for derived images.

1.2.2 Development / Mounted Source

The second usage scenario is when you want to use your development tree as source. This may be done with either the original or derived image:

```
devops-utils ++dev ansible-playbook -i hosts.ini your-playbook.yml
# or
ansible-playbook ++dev -i hosts.ini your-playbook.yml
```

This will mount current working directory as `/opt/app` and set `WORKDIR` appropriately.

Notice that parameters to the runner itself start with + instead of the usual −, this is to make them easier to differentiate from parameters to the program being run.

1.2.3 Running

You can use the image without installing the runner, but some features will be unavailable (e.g. SSH agent socket, SSH key, SSH config).

The runner uses + as option prefix character to make it easier to distinguish between options for the runner and options for the program being run. You can see usage help with:

```
devops-utils ++help
# likewise:
fab ++help
# will print the runner help message, vs
fab --help
# which will print Fabric's help message
```

The default options to `docker run` are: `-i -t --rm`.

You can pass any docker option to `docker run` using the `+O / ++docker-opt` option:

```
devops-utils +O privileged ++docker-opt net=host bash
```

As hinted above, `++dev` can be used to mount source in current working directory in the container instead of using the one baked into the image (required if not using a derived image).

When starting the container, the SSH agent socket will be passed in if available, to enable SSH authentication using own keys.

Alternatively, you can use `++key FILE` option to pass a specific key and it will be injected into the container as `/root/.ssh/id_rsa` at runtime.

SSH config file `~/.ssh/config` is also injected into the container if it exists so that any special configuration for particular hosts is respected.

Finally, you can pass `++debug` option to see how options are processed and how arguments to the programs are manipulated.

docker-machine

It is possible to run `docker-machine` commands within the image. When using `docker-machine`, it is important to pass the `++dev` option, otherwise any changes (like adding new machines) are lost.

`docker-machine` saves configuration, like keys for servers, and in fact the names of what servers are managed in a configuration directory, in our image this defaults to `/opt/app/.docker/machine`.

An example of using `docker-machine` is:

```
devops-utils ++dev docker-machine upgrade fred
```

which would execute the `docker-machine upgrade` command on host `fred`, with /opt/app mounted from current working directory. Running the above command in your home directory would pick up any previous `docker-machine` configuration, and would save anything that you change for use at a later date.

You can list machine configurations using:

```
devops-utils ++dev docker-machine ls
```

If you build a derived image containing the stored configurations, you can of course drop the `++dev` option.

docker tools

Command line docker client, docker-compose and python code using docker-py can be used against machines managed using `docker-machine`. To run a command against a specific machine, use e.g.:

```
devops-utils ++dm=NAME docker info
```

This will activate machine `NAME` (using `docker-machine env NAME`) before running the command (`docker` CLI in this case).

Similarly to deploy containers defined in docker compose file (in current directory) on machine `NAME`:

```
devops-utils ++dev ++dm=NAME docker-compose up -d
```

Python Shell

When developing, running or debugging in Python, e.g. Ansible modules or Fabric tasks, it's often useful to have a REPL. That's why the image includes IPython, ptpython and konch. You can just drop a `.konchrc` python in your source tree, e.g.:

```
# -*- coding: utf-8 -*-
# vi: set ft=python :

import konch
```

```
import fabfile

from fabric import api as fa


konch.config({
    'context': {
        # fabric
        'fa': fa,
        'ft': fabfile,
        'env': fa.env,
        'run': fa.run,
        'sudo': fa.sudo,
    }
})
```

and then run:

```
devops-utils ++dev konch
```

and you get a Python REPL with syntax highlighting, completion and quick access to some Fabric operations and tasks.

1.2.4 Extending

Both the external runner and the init (startup) script can be extended with plugins to support additional options and to modify the environment and arguments of the utilities being run.

The plugins are simple Python files that will be executed in a context containing mainly the decorators: *devops_utils.init.initfunc()* for init plugins, and *external_runner.argparse_builder()* and *external_runner.docker_run_builder()* for external runner. They are used to mark functions to be executed at specific stages in the startup process.

They should define functions decorated with the above with signatures matching the ones described in API docs for each decorator.

See *Modules* for details.

Once you have a plugin, in your derived image drop the files into */etc/devops-utils/init_plugins/* or */etc/devops-utils/runner_plugins/* directory for init or runner respectively.

1.3 Modules

1.3.1 devops_utils package

Subpackages

devops_utils.test package

Submodules

devops_utils.test.conftest module

devops_utils.test.init_module_test module

devops_utils.test.test_docker_machine module

devops_utils.test.test_install module

devops_utils.test.test_plugin module

Module contents

Submodules

devops_utils.builders module

Implements an extension mechanism for devops-utils image.

Basically a list of callables (functions defined by plugins), with a simple interface to execute them with given parameters.

NOTE

this module should be kept minimal, specifically without importing anything else from the devops_utils or any other external package; this is because this module is included in the runner, which doesn't have access to the package when installed.

class devops_utils.builders.**Builders**
Bases: list

A list of callables.

devops_utils.init module

Implements initialization process for a devops-utils container.

The *main()* here is the entrypoint of the devops-utils image. It parses the arguments and delegates execution to appropriate handler (either *run()* or *devops_utils.install.install()*).

Function *run()* handles initializing the environment, correspondingly to what the external runner has set up by passing appropriate options to docker.

initfunc() is a decorator that registers a function to be executed during init (from *run()*).

devops_utils.init.**initfunc**()
Register decorated function as initializer.

An initializer is executed on startup and can contribute to environment setup within the container. The function signature should be:

devops_utils.init.**func**(*prog : string*, *args : list*) → None

Parameters

- **prog** (*str*) – name/path to the program that will be executed by init

- **args** (*list*) – arguments it will be executed with; may be mutated to affect the final arguments

devops_utils.init.**install_file**(*src*, *dst*, *owner*, *group*, *mode*)
Install a file, set permissions and ownership.

Parameters

- **src** (*str*) – path to the source
- **dst** (*str*) – path to the destination
- **owner** (*str*) – owner username
- **group** (*str*) – group name
- **mode** (*int*) – mode to set on the destination

devops_utils.init.**install_file_if_exists**(*src*, *dst*, *owner*, *group*, *mode*)
Install a file like *install_file()* if source exists.

devops_utils.init.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
Run a program in devops-utils container.

devops_utils.init.**run**(*prog*, *args*)
Run the specified program.

devops_utils.install module

Implements installation process for the devops-utils image.

The main function here is *install()* which implements the installation process.

The *Replacer*, used from within *install()*, implements a lightweight preprocessing/templating process, thanks to which the external runner can be used as-is when installed as well as directly from source.

devops_utils.install.**install**(*args*)
Install a runner and shortcuts to all supported programs.

The runner will be installed as `devops-utils` script in a directory from host mounted at `/target`. The links to all included programs will be created in the same directory, pointing to `devops-utils`.

The runner will execute the command it's run as (or passed as first parameter if executed as `devops-utils`) via docker run.

Parameters **args** (*list*) – command line arguments

class devops_utils.install.**Replacer**(*input*, *context*)
Bases: `object`

Used to insert/replace chunks of code in a stream of lines.

This is used to embed some values into the runner script (as it doesn't have access to them from outside a container) when installing it on the host system.

Iterating through the object will yield lines from the input with lines containing a special marker replaced. The marker format is `##INIT:OPERATOR[:PARAM]##` and should be followed by a newline.

The OPERATOR can be:

- `MODULE`: then `PARAM` specifies a python module whose contents should be inserted instead of the original line
- `PLUGINS`: then `PARAM` specifies type of plugins to be included instead of the original line

- SUPPRESS: supresses the line from output (no parameter)
- VAR: then PARAM specifies name of variable to look up and place it's definition in output instead of the original line

Typical usage:

```
src = StringIO('FOO = 1  ##INIT:VAR:FOO##\n')
for line in Replacer(src, {'FOO': 2}):
    assert line == 'FOO = 2\n'
```

Parameters

- **input** (*iter*) – iterator for input lines
- **context** (*dict*) – look up variables to be replaced in this dict

RE_MARKER = <_sre.SRE_Pattern object>

handle_module(*mod*)

handle_plugins(*type_*)

handle_suppress()

handle_var(*var*)

devops_utils.plugin module

Implements plugin mechanism for the devops-utils image.

The location of the root plugin directory is defined in main package: devops_utils.PLUGIN_DIR.

The function *load_plugins()* is used to load all plugins of a given type, whereas *get_plugins()* can be used to just get a list of paths to the plugin files.

devops_utils.plugin.**get_plugins**(*type_*, *basedir='/etc/devops-utils'*, *pattern='*'*)

Return a tuple of filenames of plugins of a given type.

Parameters

- **type** (*str*) – type of plugins, i.e. init or runner
- **basedir** (*str*) – base directory to look up plugins in
- **pattern** (*str*) – glob pattern to match against plugin names

devops_utils.plugin.**load_plugins**(*type_*, *globals*, *basedir='/etc/devops-utils'*, *pattern='*'*)

Load plugins of given type.

The plugin files are looked up in ${type}_plugins directory under PLUGIN_DIR. They are execfile()d in the global namespace of the module.

The reason the plugins are exec'ed and not imported is so that it is easier for derived images to add plugins, as they can just drop files into a known directory.

Parameters

- **type** (*str*) – type of plugins, i.e. init or runner
- **globals** (*dict*) – as for execfile()

Module contents

devops-utils - devops-utils image helper tools

1.3.2 external_runner module

The external runner for the devops-utils image.

Essentially wraps `docker run -it --rm devops-utils`, providing options to make the usage more convenient.

The *main()* here is the entrypoint of the devops-utils image runner program. It parses the arguments and executes `docker run` with appropriate arguments.

argparse_builder() is a decorator that registers a function to be executed before argument parsing and can be used to add/modify options, change default values, etc.

docker_run_builder() is a decorator that registers a function to be executed after argument parsing to modify the final command that will be run.

DockerRunCommand is a helper object encapsulating various arguments which can be modified by the functions decorated with *docker_run_builder()*.

class external_runner.**DockerRunCommand**(*prog*, *prog_args*, *docker_args=None*)

Bases: object

Encapsulates components of a docker run command.

The components (exposed as corresponding instance attributes) are:

docker_args
(list) arguments for ``docker run``

prog
(str) program to run inside the container

prog_args
(list) arguments to the above program

All of the above are also accepted as constructor parameters. All of them can also be modified directly to affect the final command.

Exposes a property *cmd()* which returns a fully assembled list of docker command and arguments.

cmd

Return a fully assembled list of docker run command and arguments.

Returns docker run command and arguments; list suitable for passing to `subprocess.Popen`

Return type list

external_runner.**argparse_builder**()

Register decorated function as `ArgumentParser` instance builder.

These builders are executed before argument parsing and can be used to add/modify options, change default values, etc.

The function signature should be:

external_runner.**func**(*parser : argparse.ArgumentParser*) → None

Parameters **parser** (*argparse.ArgumentParser*) – parser to modify

external_runner.**docker_run_builder**()

Register decorated function as `docker run` command builder.

These builders are executed after argument parsing to modify the final command that will be run (either `docker run` or the program inside the container).

The function signature should be:

external_runner.**func**(*args : argparse.Namespace*, *docker_run : DockerRunCommand*) → None

The function can modify `docker_run` object directly to affect the final command that will be executed.

Parameters

- **args** (*argparse.Namespace*) – arguments and options passed to the runner itself
- **docker_run** (DockerRunCommand) – object encapsulating arguments to `docker_run` and the command to run inside the container

external_runner.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Run a program in a devops-utils container.

To see install options run *%(prog)s install –help*.

CHAPTER 2

Feedback

If you have any suggestions or questions or encounter any errors or problems with **devops-utils**, please let me know! Open an Issue at the GitHub http://github.com/gimoh/devops-utils main repository.

d

e