developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Aug 25, 2023

HOME

A set of `Makefile`s with common targets to be included in all SKA software repositories as a `git` submodule.

For any help requests, suggestions, improvements or any issues you may have with the content of this repository please drop us a message on #team-system-support channel at SKAO slack.

CHAPTER

ONE

PURPOSE

These `Makefile`s are designed to be used both standalone - within an SKA project - and in conjunction with the standard GitLab CI pipeline templates, which can be found in ska-telescope/templates-repository

The `Makefile`s contain a set of common targets that standardise the processing and handling of:

- linting
- building
- testing
- packaging
- releasing
- publishing

Artefacts for Python, OCI Images, Helm Charts, Raw files, RPM packages, Sphinx docs, Conan, BDD tests and Ansible Collections.

It also contains a simple framework for deploying Helm Charts, and running PyTest suites against Kubernetes both in cluster and on Minikube.

The general philosophy is to provide the same experience on the desktop for developers, that they will see in the pipelines when it comes to basic actions like lint, build, test etc.

The templates are designed to be imported into the `.gitlab-ci.yml` pipeline definition using the `include:` syntax, eg:

```
---
# See: https://docs.gitlab.com/ee/ci/yaml/includes.html
image: $SKA_K8S_TOOLS_DOCKER_BUILDER_IMAGE

variables:
  GIT_SUBMODULE_STRATEGY: recursive

stages:
  - lint
  - build
  - test
  - scan
  - publish
  - pages

# Include CI templates
include:
```

(continues on next page)

(continued from previous page)

```
  # OCI Images
  # do a custom set of build and publish stages
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/oci-image-lint.gitlab-ci.yml"

  # Helm Charts
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/helm-chart.gitlab-ci.yml"

  # Raw
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/raw.gitlab-ci.yml"

  # RPM
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/rpm.gitlab-ci.yml"

  # Conan
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/conan.gitlab-ci.yml"

  # Docs pages
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/docs-pages.gitlab-ci.yml"

  # .post step finalisers eg: badges
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/finaliser.gitlab-ci.yml"

  # k8s steps
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/k8s.gitlab-ci.yml"

  # Terraform
  - project: "ska-telescope/templates-repository"
    file: "gitlab-ci/includes/terraform.gitlab-ci.yml"

  # XRAY BBD test uploads
  - project: 'ska-telescope/templates-repository'
    file: 'gitlab-ci/includes/xray-publish.gitlab-ci.yml'

  # Configuration capture and publish
  - project: 'ska-telescope/templates-repository'
    file: 'gitlab-ci/includes/config-capture.gitlab-ci.yml'
```

For more details, see https://gitlab.com/ska-telescope/templates-repository/

CHAPTER

TWO

HOW TO INCLUDE MAKEFILES IN A PROJECT

Add this repository as a submodule in the root directory of your project as the `.make` directory:

```
git submodule add https://gitlab.com/ska-telescope/sdi/ska-cicd-makefile.git .make
```

Make sure that you add and commit the `.gitmodules` file and `.make` directory:

```
git add .gitmodules .make
git commit -s
```

Now include the required(i.e. `base.mk`) files in your `Makefile`:

```
...
include .make/base.mk
...
```

Or, you can include specific ones:

```
...
# include makefile targets for Kubernetes management
-include .make/k8s.mk

## The following should be standard includes
# include core makefile targets for release management
-include .make/base.mk

# include your own private variables for custom deployment configuration
-include PrivateRules.mak
...
```

CHAPTER

THREE

PIPELINE INTEGRATION

To ensure that your GitLab CI pipeline automatically clones submodules, add the following to `.gitlab-ci.yml`:

```
variables:
  GIT_SUBMODULE_STRATEGY: recursive
```

CHAPTER

FOUR

KEEPING UP-TO-DATE

:warn: ensure that the repository instructions are updated to remind developers to clone/update submodules. Ensure that `--recurse-submodules` is used on the clone:

```
git clone --recurse-submodules ...
```

Ensure that updates are pulled:

```
git pull --recurse-submodules
```

or just simply run:

```
make make
```

CHAPTER

FIVE

INTEGRATING MAKE WITH YOUR WORKFLOW

The common set of `make` targets provided by this repository are designed to both be integrated with the GitLab CI templates provided here and as part of the developer workflow to manage lifecycle activities from linting, building to tagging and releasing, and to be able to first check whether CI related steps will run in the pipelines as expected.

They are broadly broken up into groups (in `.mk` files) aligned with different artefact types, and the common release process:

- ansible.mk - lint, build, publish
- base.mk - convenience include for release.mk, docs.mk, make.mk, help.mk
- bats.mk - perform BATS (shell) based testing - test
- cpp.mk `#TODO`
- docs.mk - build Sphinx docs (mostly published to Read the Docs)
- helm.mk - Helm Chart lint, build, publish
- infra.mk - reusable targets for `orch` and `playbooks` for inclusion of ska-ser-orchestration and ska-ser-ansible-collections submodules
- k8s.mk - Testing Helm Charts against Kubernetes
- make.mk - update .make submodule
- oci.mk - OCI Images lint, build, publish
- python.mk - format (code), lint, build, test, publish
- raw.mk - build, publish
- rpm.mk - build, publish
- conan.mk - build, publish
- terraform.mk - format (code), lint
- release.mk - management of the `.release` files and language/framework specific metadata and version files (eg: `pyproject.toml`, `Chart.yaml`, `galaxy.yml`), and git tag management
- help.mk - help, long-help
- tmdata.mk - build, publish telescope model data
- xray.mk - publish BDD test results
- configcapture.mk - capture k8s software configuration, such as dsconfig, pod image versions etc

Each of the publish make targets also handles the decoration of the specific artefact with the necessary SKAO metadata. For OCI Images, this is adding `LABEL`s, and for other artefacts it is inserting `MANIFEST.skao.int` file - in both cases,

adding GitLab CI pipeline variables that describe the artefact, where it came from and how it was built. You can read more about this in developer portal here

CHAPTER

SIX

HOOKS FOR DEVELOPERS

The main make targets for each artefact type eg: lint, build, test, publish - have hooks that can be overridden by the Developer to enable pre and post processing. For example, taking the `helm-lint` target, there are dummy(empty) targets for `helm-lint-pre` and `helm-lint-post`. If a target for `helm-lint-pre` is defined in the root(local) `Makefile` of the project that includes the `.make` files, it will be guaranteed to be executed before the `helm-lint` is carried out, eg:

```
...
# include Helm Chart support
include .make/helm.mk
...

helm-pre-lint: ## make sure auto-generate values.yaml happens prior to linting chart
  envsubst < charts/widget/values.yaml.in >charts/widget/values.yaml
...
```

It is also possible to overide common variables on a case by case basis, for example `PYTHON_VARS_AFTER_PYTEST` is a variable that is used both in the context of `python-test` and `k8s-test`. It could be set globally and then specialised to a particular call as follows (example from `ska-tango-examples`):

```
PYTHON_VARS_AFTER_PYTEST = -m 'not post_deployment' --forked \
      --disable-pytest-warnings

# set different switches for in cluster: --true-context
k8s-test: PYTHON_VARS_AFTER_PYTEST := \
   --disable-pytest-warnings --count=1 --timeout=300 --forked --true-context
```

CHAPTER

SEVEN

CREATING A RELEASE

Steps:

- branch for work `git checkout -b xx-999-marvelous-thing`
- edit, hackedy hack hack hack files
- commit and push work `git commit -a -s && git push -u origin xx-999-marvelous-thing`
- create Merge Request from the branch, and rinse and repeat until happy work is done and tested (optional: get early feedback on the MR)
- happy work is done? Then get code review, get approval, and merge the Merge Request
- checkout the default branch and pull `git checkout <default> && git pull`
- create a Issue in the Release Management project
- bump the `.release` file version with `make bump-patch-release`, `make bump-minor-release`, or `make bump-major-release`
- create the git tag with `make create-git-tag` - this will also commit the changes to the `.release` file. Make sure that the commit message starts with the Jira Issue that you have just created, REL-(Ticket number that was just created)
- push up the last commit (if any for the `.release` file) and the git tag: `make push-git-tag`, this will trigger the following events:
  - a release page has been added to the tag you have just created in Gitlab
  - a link to the release notes has been added to your REL jira ticket, under the release notes field and also in the Issue Links
  - a message to slack has been sent with the new artefacts that were published to CAR
- check your CI pipeline to make sure all artefacts have been built and published on the tag pipeline run

CHAPTER

EIGHT

CHANGELOG MANAGEMENT AND RELEASE NOTES PUBLISHING

The changelog regenation process relies on the `generate-changelog` make target present in the `release.mk` makefile. It is meant to be used in a GitLab tag pipeline job as it depends on the following variables to publish the release notes to a newly created tagged commit:

- `CI_COMMIT_TAG`
- `CI_JOB_TOKEN`
- `CI_PROJECT_ID`
- `CI_PROJECT_URL`
- `CI_SERVER_URL`

The process can also be customized using the following variables:

- `CHANGELOG_FILE` - Used to specify the changelog file that is meant to keep the release notes for every release. Defaults to `CHANGELOG.md`.
- `CHANGELOG_VERSION` - Used to change the default `git-chglog` version used. Defaults to `0.15.0`.
- `CHANGELOG_CONFIG` - Used to overwrite the `git-chglog` config file. Defaults to .make/.chglog/config.yml.
- `CHANGELOG_TEMPLATE` - Used to overwrite the `git-chglog` template used to generate the changelog output. Defaults to .make/.chglog/CHANGELOG.tpl.md.

The make target `generate-release-notes` can also be used to generate the release notes (`CHANGELOG.md`) only, without publishing them.

8.1 Author notes

Custom author notes can be added to the release notes for any release. In `CHANGELOG.md`, simply add a level 3 heading called `Author Notes` under the heading for any release. Subsequently running the make targets `generate-changelog` or `generate-release-notes` will re-generate the changelog but keep all author notes for releases that have them. Author notes can be written into a fully populated or not fully populated `CHANGELOG.md` file, however they must appear below a release heading for them to not be overwritten by the `generate-changelog` and `generate-release-notes` make targets. Below is an example for a release 0.4.24:

```
...

## [0.4.24]

### Author Notes
```

(continues on next page)

(continued from previous page)

```
Write notes here

...
```

CHAPTER

NINE

GET HELP

As soon as you include the `.make/help.mk` file in your project `Makefile`, you will have access to the built in help features. These will describe the targets, their hooks and variables.

To get the short help, run `make` or `make help` to see all the documented targets and variables. You can also provide a **section** name as an argument to get a subset eg: `make help oci`, to get all of the OCI Image supporting short help. The short help describes the public make **targets**, and **variables** available.

It is also possible to get more detailed help using `make long-help` which will give a more in-depth description of the **targets** supported and their intended use. As before, you can get a subset by providing a section as an argument eg: `make long-help oci`. You could also just run `make long-help` and follow the prompts.

These integrations are supported by the SKA Systems Team, and assistance can also be gained through #team-system-support Slack channel

CHAPTER

TEN

CONTRIBUTING TO THE REPOSITORY

You are welcome to create a MR and assign it to the any of CODEOWNERS (see `./CODEOWNERS` file).

10.1 How to test the repository

The repository is using BATS libraries as a bash unit testing framework.

- Testing is done from the `./tests` directory and all the flags and commands are called based on that directory. Switch to the testing directory: `cd tests`
- Install the dependencies with `make bats-install`
- Run All Tests with: `make test`
- You can customise which tests to run with `UNIT_TESTS`*(Default: `$(CURDIR)/../tests/unit`)* variable by setting the test file. Note that you need to set the path relative to `tests/` folder. For example, to run oci tests only, simply run `make test UNIT_TESTS=../tests/unit/05_oci.bats`
- Further BATS flags can be used with `BATS_ARGS` variable. For example if you only want to run metadata tests using `--filter` flag for oci tests only, simply run `make test UNIT_TESTS=../tests/unit/05_oci.bats BATS_ARGS="--show-output-of-passing-tests --filter 'check metadata'"`. `--show-output-of-passing-tests` is helpful in debugging to see the output for passing tests as well.