developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Feb 22, 2022

Home

This project is an example of how a Tango device coded in Python can be structured as an SKA project and integrated with the continuous integration server. The tango-example project builds upon the ska-python-skeleton example project, replacing the pure Python application in ska-python-skeleton with the 'power supply' Tango device presented as an example in the PyTango online documentation. The project uses the SKA Docker images, and shows how source code located in a local workspace can be integrated with these images.

CHAPTER 1

Quickstart

This project is structured to use Docker containers for development and testing so that the build environment, test environment and test results are all completely reproducible and are independent of host environment. It uses `make` to provide a consistent UI (see *Makefile targets*).

Build a new Docker image for the example power supply device with:

```
make build
```

Test the device using:

```
make test
```

Launch an interactive shell inside a container, with your workspace visible inside the container:

```
make interactive
```

CHAPTER 2

Adapting this project

This example project can be used as a starting point for a new project. The general workflow after forking/cloning is to:

1. modify the name of the Docker image so that it refers to your new project by editing Makefile and .release (see *Docker tag management*);
2. if required, modify docker-compose.yml to specify which other Tango devices and Docker containers should be available in the test/interactive environment for your project (see *Test environment / interactive environment*);
3. add any new build and/or test dependencies by editing Pipfile, refreshing Pipfile.lock afterwards (see *Python packaging and dependency management*);
4. introduce new source code and unit tests (see *Device development* and *Unit tests*);
5. modify setuptools configuration so 'python setup.py build' build your new device (see *Python packaging and dependency management*);
6. if required, fix up the test-harness directory by adding any required test data and/or customising the test Makefile (see *Unit tests*);
7. Run tests;
8. Commit.

2.1 Docker tag management

The name/tag of the Docker image created for the project is defined in the project Makefile. In this example project, the images are tagged as nexus.engageska-portugal.pt/tango-example/powersupply:latest. This tag should be changed to refer to your device.

1. Modify Makefile, changing `DOCKER_REGISTRY_USER` and `PROJECT` to give an appropriate Docker image name for your project.
2. edit .release, modifying the 'release' definition to match the name of your project.

The Docker image that results contains a Python environment with all project dependencies installed, plus a copy of the project source code and unit tests. The Python environment is located at /venv in the image, while the project source code is located in the /app folder.

2.2 Test environment / interactive environment

The docker-compose.yml file for each project defines which Docker containers should be created when starting an interactive development session or a test session. Note that the interactive session and tests do not execute in any of the containers defined in docker-compose.yml, but in an separate container created alongside those defined in the file.

In this example project, docker-compose.yml defines a test environment comprising three containers: a Tango database; a databaseds server, and a container for the example power supply device. This project illustrates two execution scenarios for unit tests: one, where unit tests and device server run in the same container, and a second scenario where unit tests and the device server execute in separate containers. The power supply device specified in docker-compose.yml allow this second scenario.

The docker-compose.yml file should be customised for your project. Note that the same container definitions are used for 'make test' and 'make interactive'.

(Optional) Docker entry point management:

1. Modify Dockerfile, redefining CMD to give the name of the Python file that should be executed if the Docker image is run without arguments.

2.3 Python packaging and dependency management

Just as for the ska-python-skeleton project, Python runtime dependencies and test dependencies for the device are defined in the project Pipfile. For the example Tango device in this project, the dependencies are pytango, plus pytango's dependency: numpy. The versions are pinned to use the same version of pytango and numpy contained in the ska-python-builder Docker image, which allows the cached versions of pytango and numpy to be reused, thus saving time when the image is rebuilt.

The steps to modify the project dependencies as required for your project are:

1. edit the project dependencies for your device (pytango, numpy, etc.) in the Pipfile;
2. whenever Pipfile is edited, execute `make build && make piplock` to generate an updated Pipfile.lock inside a container and copy it into your local workspace;
3. update the project dependencies in setup.py to match those in the Pipfile.

The Python packages and modules provided by the project are defined in setup.cfg and setup.py. For this example project, these files refer to the PowerSupply device contained in the powersupply package. These references should be modified as required for your project:

1. update setup.py to refer to your project, updating references to 'powersupply' (the package for the example device provided in this example project) to point to the package(s) for your device;
2. modify setup.cfg, replacing any references to powersupply with references to the Python package(s) for your device.

2.4 Device development

The code for your device exists on disk in your local filesystem and can be edited with the text editor or IDE of your choice. Changes made locally are seen immediately in the Docker container when using `make interactive` (see

section on *Makefile targets*), or on the next image rebuild when an interactive session is not used.

The source code for the power supply device in this example project can be found in the /powersupply directory. SKA convention is that device classes should be contained in a Python module contained in a Python package. In this project, the power supply device is defined in the powersupply.py module contained in the powersupply package. When adding your device, it should be structured similarly.

2.5 Unit tests

Source code for the power supply device's unit tests can be found in the /tests folder. In this example project, tests are not executed in the native host. Instead, the device source code is prepared into a Docker image, a container is launched using this image, and the tests run inside this container.

Files in the 'test-harness' directory are made available to the container during the test procedure. Files in this direcotry are not permanently included in the Docker image. Hence, the 'test-harness' directory includes files that are required for unit testing (data files, makefiles, etc.) that are not required / should not be included for production. The test-harness directory in the example project includes a Makefile; the Makefile defines a `test` make target, which is launched inside a container at runtime and defines the entry point for the test procedure for the device. This `make test` target can also be executed inside a container during a `make interactive` session. For a Python project such as this, the 'test' target calls the standard 'python setup.py test' procedure. The 'test' target for non-Python projects will call different procedures.

test-harness/Makefile should be customised for your project. If testing using a tango client, the 'tango_admin' line should be modified to wait for your device to become available rather than the example device. If your tests do not use a tango client, the 'tango_admin' line can be deleted.

CHAPTER 3

Makefile targets

This project contains a Makefile which acts as a UI for building Docker images, testing images, and for launching interactive developer environments. The following make targets are defined:

Makefile target	Description
build	Build a new application image
test	Test the application image
interactive	Launch a minimal Tango system (including the device under development), mounting the source directory from the host machine inside the container
piplock	Overwrite the Pipfile.lock in the source with the generated version from the application image
push	Push the application image to the Docker registry
up	launch the development/test container service on which this application depends
down	stop all containers launched by 'make up' and 'make interactive'
dsconfigcheck	check a json file (environment variable DSCONFIG_JSON_FILE) according to the project lib-maxiv-dsconfig json schema(link below)
dsconfigdump	dump the entire configuration to the file dsconfig.json
dsconfigadd	Add a configuration json file (environment variable DSCONFIG_JSON_FILE) to the database
help	show a summary of the makefile targets above

3.1 Json Configuration

The tool for configure the database is lib-maxiv-dsconfig. To execute it, please use the docker image at the following link: tango-dsconfig. Note that the environment variable DSCONFIG_JSON_FILE use a volume called 'tango-example' to access the project folder.

3.2 Creating a new application image

`make build` target creates a new Docker image for the application based on the 'ska-python-runtime' image. To optimise final image size and to support the inclusion of C extension Python libraries such as pytango, the application is built inside an intermediate Docker image which includes compilers and cached eggs and wheels for commonly-used Python libraries ('ska-python-builder'). The resulting Python environment from this intermediate stage is copied into a final image which extends a minimal SKA Python runtime environment ('ska-python-runtime'), to give the final Docker image for this application.

3.3 Interactive development using containers

`make interactive` launches an interactive session using the application image, mounting the project source directory at /app inside the container. This allows the container to run code from the local workspace. Any changes made to the project source code will immediately be seen inside the container.

3.4 Test execution

`make test` runs the application test procedure defined in test-harness/Makefile in a temporary container. The Makefile example for this project runs 'python setup.py test' and copies the resulting output and test artefacts out of the container and into a 'build' directory, ready for inclusion in the CI server's downloadable artefacts.

CHAPTER 4

Kubernetes integration

The Tango Example has been enhanced to include an example of deploying the application suite on Kubernetes. This has been done as a working example of the Orchestration Guidelines. Included is a Helm Chart and a set of `Makefile` directives that encapsulate the process of deploying and testing the powersupply example on a target cluster.

4.1 Extended Makefile targets

There are an extended set of make targets are defined that cover Kubernetes based testing and deployment:

Makefile target	Description
delete	delete the helm chart release (without Tiller)
delete_namespace	delete the kubernetes namespace
deploy	deploy the helm chart (without Tiller)
helm_delete	delete the helm chart release (with Tiller)
helm_tests	run Helm chart tests (with Tiller)
describe	describe Pods executed from Helm chart
ingress_check	curl test Tango REST API
install	install the helm chart (with Tiller)
k8s_test	test the application on K8s
k8s	Which kubernetes are we connected to
kubeconfig	export current KUBECONFIG as base64 ready for KUBE_CONFIG_BASE64
lint	lint check the helm chart
localip	set local Minikube IP in /etc/hosts file for Ingress $(INGRESS_HOST)
logs	show Helm chart POD logs
mkcerts	Make dummy certificates for $(INGRESS_HOST) and Ingress
namespace	create the kubernetes namespace
rk8s_test	run k8s_test on K8s using gitlab-runner
rlint	run lint check on Helm Chart with gitlab-runner
show	show the helm chart

4.2 Test execution on Kubernetes

The test execution has been configured to run by default on Minikube so that testing can be carried out locally. For remote execution, further configuration would be required to handle PersistentVolume storage correctly.

The Deployment framework is based on Helm, and has examples with and without Tiller. This is because the Helm community is in the process of deprecating the Tiller component, so ensuring that charts work without Tiller is future proofing. For the examples with Tiller, in order to simplify the installation and keep it secure, the setup uses a Helm plugin for a local only Tiller. This still enables the standard features of Helm such as `install, history, upgrade, rollback` but does not install Tiller in the Kubernetes cluster.

4.2.1 Setting variables

Variables can be set to influence the deployment, and should be placed in a `PrivateRules.mak` file in the root of the project directory. Variables in this file and imported to override `Makefile` defaults:

Makefile variable	Default	Description
KUBE_NAMESPACE	default	the Kubernetes Namespace for deployment
HELM_CHART	tango-example	the Helm chart name for deployment (tango-base or tango-example)
HELM_RELEASE	test	the Helm release name for deployment
KUBECONFIG	/etc/deploy/config	KUBECONFIG location for `kubectl`
KUBE_CONFIG_BASE64	<empty>	base64 encoded contents of KUBECONFIG file to use for connection to Kubernetes

When working with `Minikube`, set `KUBECONFIG` in `PrivateRules.mak` as follows:

- `KUBECONFIG = $(HOME)/.kube/config`

And then the correct `KUBE_CONFIG_BASE64` value can be automatically generated into `PrivateRules.mak` by running `make kubeconfig`.

4.2.2 Running the tests

The following assumes that the available test environment is a local `Minikube` based Kubernetes cluster. To run the tests, follow either of the two workflows on `Minikube`:

Without Tiller:

- run `make deploy` and check that the processes settle with `watch kubectl get all`
- execute the tests with `make k8s_test` - this will run the powersupply test suite
- teardown the test environment with `make delete`

With Tiller:

- run `make install` and check that the processes settle with `watch kubectl get all` - Tiller will block until the deployment is complete
- run the Helm tests with `make helm_tests`. This will run any test `Pod``s specified in the ``charts/tango-example/templates/tests` directory.
- execute the tests with `make k8s_test` - this will run the powersupply test suite
- teardown the test environment with `make helm_delete`

Alternatively the entire process can be executed using gitlab-runner locally with `make rk8s_test`. This will launch the entire suite in a `Namespace` named after the current branch with the following steps:

- Set Namespace
- Install dependencies for Helm and kubectl
- Deploy Helm release into Namespace
- Run Helm tests
- Run test in run to completion Pod
- Extract Pod logs
- Set test return code
- Delete Helm release
- Delete namespace
- Return job step result

Test output is piped out of the test `Pod` and unpacked in the `./build` directory.