# developer.skatelescope.org Documentation

***Release 0.10.5***

**Marco Bartolini**

**Aug 18, 2023**

# HOME


**1 Requirements** **3**

**2 Checks** **5**
- 2.1 Configuration . . . 5
- 2.2 How to Add a New Check . . . 5
- 2.3 Testing . . . 7

**3 Current Sub-tasks** **9**
- 3.1 Main task . . . 9
- 3.2 Validation Task . . . 9
- 3.3 Get Metadata Task . . . 10
- 3.4 Quarantine Task . . . 10
- 3.5 Create Merge Request Task . . . 11
- 3.6 Insert on Database Task . . . 11
- 3.7 Scanning Task . . . 11

**4 DB Structure** **13**

**5 Purging Existing Artefacts** **15**

SKA Celery worker server for validations tasks and generic automation.

This repository deploys the celery workers, (mongodb and redis) and hosts artefact validation checks. The checks are used in the webhook plugin to trigger for newly created artefacts.

CHAPTER

ONE

# REQUIREMENTS

Run `make vars` and define the mandatory variables listed in the output the environment variables in your `.env` file and `PrivateRules.mak` file.

*Note:* In this service code, the term component is interchangeble for a single artefact. An artefact/component can consist of multiple assets.

CHAPTER

TWO

# CHECKS

Currently the plugin checks for:

- **Naming Convention:** described in developer portal
- **Tag Convention:** described in developer portal
- **Metadata:** described in developer portal
- **Packaging:** described in developer portal

## 2.1 Configuration

This service is currently only checking for artefacts that are created. The deleted or updated artefacts are not checked. The service is enabled for the Nexus repository that is passed with `NEXUS_URL` variable. To activate the slack reports for input validation alerts, the `SLACK_VALIDATION_WEBHOOK` should be set with an incoming webhook url (currently set in the project variables for the pipelines).

The server that hosts this service should be able to create a valid docker connection with `NEXUS_API_USERNAME` and `NEXUS_API_PASSWORD` with the `NEXUS_URL` Docker Registry and have rights to upload packages to `NEXUS_DOCKER_QUARANTINE_REPO` and `MAIN_DOCKER_REGISTRY_HOST` is used for pulling the artefacts from the main docker registry (This needs to point to the registry hosted in `NEXUS_URL`). For integration testing `TEST_NEXUS_URL`, `TEST_NEXUS_API_USERNAME` and `TEST_NEXUS_API_PASSWORD` are used.

Each check is configured with a feature toggle so that any artefact with its name can be excused from a check. If the artefact name matches the list defined in the feature toggle (ending with *-excluded*), the check won't be performed. For the feature toggles to work, `UNLEASH_API_URL` and `UNLEASH_INSTANCE_ID` should be defined. The feature toggle integration is disabled for unit tests with `UNLEASH_INACTIVE` variable set and `UNLEASH_ENVIRONMENT` variable is used in production to enable it for production environments differentiating it from development environments.

The variable `SKA_TRIVY_IMAGE` corresponds to the trivy image with the current version used in the SKAO project. This variable is used for the image scanning task.

## 2.2 How to Add a New Check

Each new check must use the abstract base class, Check, to ensure to define its `check` action, which performs the actual checking on the artefact and returns a boolean indicating the result.

Base Class:

```
class Check(ABC):

    def __init__(
        self,
        name: str,
        feature_toggle: str,
        quarantine_toggle: str,
        messageType: MessageType,
        mitigationStrategy: str,
        checkVersion: int,
        loggername: str,
    ):
        super().__init__()
        self.feature_toggle = feature_toggle
        self.quarantine_toggle = quarantine_toggle
        self.name = name
        self.message = ""
        self.message_type = messageType
        self.mitigation_strategy = mitigationStrategy
        self.check_version = checkVersion
        self.result: bool = None
        self.logger = logging.getLogger(loggername)
        self.extra_info = {}

    @abstractmethod
    async def check(self, component: Component) -> bool:
        pass

    def toDict(self) -> dict:
        return {
            "name": self.name,
            "check_version": self.check_version,
            "result": self.result,
            "message": self.message,
            "extraInfo": self.extra_info,
            "feature_toggle": self.feature_toggle,
            "quarantine_toggle": self.quarantine_toggle,
        }
```

Example Check:

```
class CheckVulnerabilities(Check):
    def __init__(self, logger_name: str):
        super().__init__(
            self.__class__.__name__,
            "check-vulnerabilities",
            "quarantine-check-vulnerabilities",
            MessageType.FAILURE,
            (
                "Raw artefact is an invalid tarball (tar.gz)! Please refer to "
                "[the developer portal](https://developer.skatelescope.org/en/latest/
↪tools/software-package-release-procedure.html#raw) "  # NOQA: E501
                "to correct this issue."
```

(continues on next page)

(continued from previous page)

```
            ),
            1,
            logger_name,
        )

    async def check(self, component: Component) -> bool:

        if component.format == ComponentFormat.DOCKER:
            result = trivy_scanning_task(component)
        elif component.format == ComponentFormat.PYTHON:
            result = await gemnasium_scanning_task(component)
        else:
            self.result = True
            return self.result

        self.extra_info["vulnerability_scanning"] = result

        if (
            self.extra_info["vulnerability_scanning"]["metrics"]["critical"]
            > 0
        ):
            self.result = False
        else:
            self.result = True

        return self.result
```

After the new check is implemented, the `checks` variable in the repository validator file should be updated to reflect the list of implemented checks.

Then the necessary tests for the added checks should be added in tests folder. These tests should get picked up by the main frameworks testing.

Finally, each check should be initialised and called in the validate_job file to be included into the list of checks that are performed for the Artefacts.

## 2.3 Testing

For unit testing, after setting the *configuration*, run `make unit_test`. This shouldn't make any external API calls so you can (should) set the necessary variables with placeholder values because they are only used to see if they exist or not. Additionally, you can provide a test name to run an individual test with `UNIT_TEST_NAME` variable. i.e. `make unit_test UNIT_TEST_NAME=unit/test_celery.py::test_validation_job_feature_toggle_disabled`

For post-deployment and integration testing, first set up your minikube environment. Next, open a new terminal and run `eval $(minikube docker-env)` so that you use the same docker daemon as minikube. Then, build the latest image with `make docker-build` which should build a dirty tagged OCI image. This image will be used in deploying to your local minikube with `make install-chart-for-testing` or `make install-chart`. For post-deployment tests, `make install-chart-for-testing` uses `TEST_NEXUS_URL`, `TEST_NEXUS_API_USERNAME` and `TEST_NEXUS_API_PASSWORD` variables to override `NEXUS_URL`, `NEXUS_API_USERNAME` and `NEXUS_API_PASSWORD` variables reflectively. So, you either set the correct variables with nexus values or just set the test variables and use the alternative make target. `TEST_NEXUS_URL` should be the service name for the nexus deployed into the cluster, by default it is `nexus3-nexus-repository-manager` and set in the `/post-deployment/resources/nexus-repo/`

`values.yaml` file. Finally, run `make test` for the integration test. This target will also deploy a nexus instance into your minikube and configure it so that it's used in integration tests instead of the production repository.

CHAPTER

THREE

# CURRENT SUB-TASKS

Currently there is a main celery task and 5 subtasks that are being summoned from the main task. All the tasks are inside the tasks folder

All tasks must overwrite the `on_success`, `on_retry` and `on_failure` callbacks with the `log_task_success`, `log_task_retry` and `log_task_failure` functions present in the common file in the task decorator to enable task lifecycle logging. Successes are logged with INFO level, retries with WARNING level and failure with FATAL level. Subtask calling logs are logged with INFO level.

All tasks have a soft time limit meaning that they will raise an expection and restart after a certain ammount of time, and a normal time limit that will shutdown the task without retrying or waiting. This both limits are defined in the celeryconfig.py file.

## 3.1 Main task

The `main_task` is the one responsible to call all the sub-tasks sequentially, gather the information from all of them and create a document that will be inserted on the mongo database.

The tasks are being called from the following sequential order:

- Validation task
- Get Metadata Task
- Quarantine Task
- Create Merge Request Task
- Insert on Database Task

## 3.2 Validation Task

The `validation task` is the first one to be called and is where all the following checks are performed:

- Check Artifact Name
- Check Artifact Version
- Check Artifact Metadata
- Check Raw Artifact Asset
- Check Vulnerabilities

After performing all the checks the validation task returns the information about them in a array format like so:

```
[
    {
        "name": "CheckComponentName",
        "check_version": 1,
        "result": True,
        "message": "",
        "extraInfo": {},
        "feature_toggle": "check-component-name",
        "quarantine_toggle": "quarantine-check-component-name",
    },
    ...
    {
        "name": "CheckVulnerabilities",
        "check_version": 1,
        "result": False,
        "message": "Artifact has critical vulnerabilities",
        "extraInfo": {
            *** Scanning output ***
        },
        "feature_toggle": "check-vulnerabilities",
        "quarantine_toggle": "quarantine-check-vulnerabilities",
    },
]
```

## 3.3 Get Metadata Task

The `get metadata` task will get the metadata from the artifact. In case of a artifact of type pypi it returns the metadata only if the metadata is present both on the .whl file and on the .tar.gz file (considering that both are present). If the MANIFEST.skao.int file is missing on one of this files this task will return None. In case of the artifact being of type docker this task will return the metadata if the metadata is present on the labels of the docker image. The artifacts of type helm are inside .tgz file, so the check will only pass if the file MANIFEST.skao.int is inside and with the right metadata.

## 3.4 Quarantine Task

This task will be called if at least one of the checks returned false. If that is the case the artifact will be downloaded from his original nexus repository, then uploaded to the quarantine repository and finally it will be deleted from his original repository.

## 3.5 Create Merge Request Task

Marvin will create a merge request only if the quarantine task was called before and the get metadata task was able to get the metadata info. If this is the case Marvin will create a merge request where he assigns the GITLAB_USER_ID given on the metadata and creates a description table like the one below to help the developers easily fix the problems and better understand them.

## 3.6 Insert on Database Task

This task is responsible for inserting documents on the database.

## 3.7 Scanning Task

This task recieves the name and version of the image and scans it using **trivy** with the version that is present on the gitlab group variables `SKA_TRIVY_IMAGE`. This Task ouputs a dictionary with the following structure:

```
{
    "timestamp": str(datetime.utcnow()),
    "metrics": {
        "total": 0,
        "critical": 0,
        "high": 0,
        "medium": 0,
        "low": 0,
        "unknown": 0,
    },
    "vulnerabilities": {
        "critical": [
            {
                "Target": "", # Package that has the Vulnerabilities (Example Ubuntu or␣
↪python)
                "Type": "", # The package type being scanned
                "VulnerabilityID": "",
                "Severity": "",
                "PkgName": "",
                "InstalledVersion": "",
                "FixedVersion": ""
                "Description": ""
            }
        ],
        "high": [],
        "medium": [],
        "low": [],
        "unknown": [],
    },
    "rep_command": "", # Bash command to replicate the scanning locally
}
```

CHAPTER

FOUR

# DB STRUCTURE

Every time a task is executed, a DB entry is written in a document DB (MongoDB) which represents the result of a validation performed on an artefact and because that validation is unique, the document is also unique and immutable. According to the MongoDB documentation, the key decision concern whether the related data are stored as embedded document or as reference document. Given that the maximum size for a document is 16 MB (a big number in relation of what we need to store, basically log information about validations), it has been decided to use embedded document.

The essential fields stored in a document are:

- (Component) Name (of the artefact) i.e. ska-tango-images/pytango-runtime (type string)
- (Component) Version (of the artefact) i.e. 9.3.3 (type string)
- Actioner name: usually the taskname but it can change in the future i.e. main_task (type string)
- State: valid(0) or quarantine(1) (type int)
- timestamp (of the mongo document) (type datetime)
- Metadata (from the artefact) (type dictionary) i.e {"metadata": [labels] }
- Nexus information (type dictionary) i.e {"component_id": "cHlwaS1pbnRlcm5hbDpmMjFmZjNiMmE0YzI0YWFhZTMyZWNlN
- Gitlab information (type dictionary) i.e {"MR": "23"}
- Check results (type array) [ { name: CheckComponentManifest result: True or False } ]

Example Document:

```
{
    "name": "tango-example",
    "version": "0.3.8",
    "timestamp": "2021-05-24T00:00:00",
    "actioner": "main_task",
    "metadata": {
        "Additional_metadata": " -- author: Matteo <matteo.dicarlo@inaf.it> --␣
↪description: This image illustrates ...",
        "CI_COMMIT_AUTHOR": "Matteo",
        "CI_COMMIT_REF_NAME": "signals",
        "CI_COMMIT_REF_SLUG": "signals",
        "CI_COMMIT_SHA": "87fe386904c12d47f2e70f3a96f12a0b39d88f53",
        "CI_COMMIT_SHORT_SHA": "87fe3869",
        "CI_COMMIT_TIMESTAMP": "2021-05-21T11: 57: 13+02: 00",
        "CI_JOB_ID": "1282524344",
        "CI_JOB_URL": "https: //gitlab.com/ska-telescope/tango-example/-/jobs/1282524344
↪",
        "CI_PIPELINE_ID": "307198642",
```

(continues on next page)

(continued from previous page)


```
        "CI_PIPELINE_IID": "747",
        "CI_PIPELINE_URL": "https://gitlab.com/ska-telescope/tango-example/-/pipelines/
↪307198642",
        "CI_PROJECT_ID": "9673989",
        "CI_PROJECT_PATH_SLUG": "ska-telescope-tango-example",
        "CI_PROJECT_URL": "https://gitlab.com/ska-telescope/tango-example",
        "CI_RUNNER_ID": "6081833",
        "CI_RUNNER_REVISION": "2ebc4dc4",
        "CI_RUNNER_TAGS": "ska,",
        "GITLAB_USER_EMAIL": "matteo.dicarlo@inaf.it",
        "GITLAB_USER_ID": "3003086",
        "GITLAB_USER_LOGIN": "matteo1981",
        "GITLAB_USER_NAME": "Matteo"
    },
    "validation": [
        {
            "name": "CheckComponentName",
            "check_version": 1,
            "result": True,
            "message": "",
            "extraInfo": {},
            "feature_toggle": "check-component-name",
            "quarantine_toggle": "quarantine-check-component-name",
        },
        ...
        {
            "name": "CheckVulnerabilities",
            "check_version": 1,
            "result": False,
            "message": "Artifact has critical vulnerabilities",
            "extraInfo": {
                *** Scanning output ***
            },
            "feature_toggle": "check-vulnerabilities",
            "quarantine_toggle": "quarantine-check-vulnerabilities",
        },
    ],
    "nexus": { "repository": "quarantine_repository" },
    "MR": {}
    }
```

CHAPTER

FIVE

# PURGING EXISTING ARTEFACTS

A tool to purge existing artefacts from a repository exists which can be used when checks are updated and current artefacts may no longer be compliant.

To use this tool, access to the Kubernetes cluster running the production deployment through `kubectl` is required. From the `ubuntu` account on terminus, the tool can be executed as follows:

```
export KUBECONFIG=/etc/k8s-system-team-production-conf

export POD=$(kubectl get pods -l app.kubernetes.io/name=ska-cicd-artefact-validations -n␣
↪production --no-headers -o=name | head -n1)

kubectl exec $POD -n production -- python3 /ska_cicd_artefact_validations/validation/
↪controllers/repository_validator.py <repository-name>... [--force] [--debug]
```

For example, to purge non-compliant artefacts from the pypi-internal and raw-internal repositories, the above export commands can be used followed by:

```
kubectl exec $POD -n production -- python3 /ska_cicd_artefact_validations/validation/
↪controllers/repository_validator.py pypi-internal raw-internal
```

The `--debug` option provides additional output, the `--force` option forces the validation on every artefact and the `--dry-run` allows to the run the script without sending any artefact for validation. To only see a list of non-compliant artefacts, the output of a `--dry-run` can be piped into a `grep "DRY-RUN"`.