developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Aug 21, 2023

HOME

1 Structure 3

2 Local Development 5
- 2.1 General Workflow . . . 5
- 2.2 Linting and code-style . . . 6
- 2.3 Building and Running . . . 6
- 2.4 Adding a new cicd automation service . . . 7
- 2.5 Testing With FastAPI and Pytest-BDD . . . 8
- 2.6 Publishing/Releasing . . . 9

3 Merge Request Checks 11

This project is the template projects that is intended for use by SKA CICD Services API Applications (*mainly bots*). The project and generated projects from this template should follow the SKA Developer Portal closely. Please check the latest guidelines, standards and practices before moving forward!

CHAPTER

ONE

STRUCTURE

Project structure is following a basic python file structure for FastAPI as below:

```
.
├── Dockerfile
├── LICENSE
├── Makefile
├── README.md
├── app
    ├── ...
├── build
    ├── ...
├── charts
    ├── ...
├── conftest.py
├── docs
├── pyproject.toml
└── tests
    ├── ...
```

Basically, this project uses the following technologies:

- Docker: To build a docker image that exposes port *80* for API endpoints.
- Kubernetes and Helm: Project also includes a helm chart to deploy the above image in a loadbalanced kubernetes cluster.
- Python Package: Project also includes a python package so that it can be downloaded as such. (**The usability of this capability highly depends on the actual implementation!**)

CHAPTER TWO

LOCAL DEVELOPMENT

2.1 General Workflow

Install Poetry for Python package and environment management. This project is structured to use this tool for dependency management and publishing, either install it from the website or using below command:

```
curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py |␣
↪python -
```

However if you want to use other tools (`Pipenv` or any other tool that can work with `requirements.file`) run the following script to generate a requirements file.

```
pip install Poetry
make exportlock
```

This will generate `requirements.txt` and `requirements-dev.txt` files for runtime and development environment.

2.1.1 *What is Poetry and Why are we using it*

Poetry is a python dependency management and packaging tool. It is similar to pipenv for dependency management but it is more actively developed. It organizes dependencies in separate sections of the same file using `pyproject.toml`, described in PEP 518 so it can specify publishing information and configure installed tools (like black, isort, tox etc.) which makes it easy to both configure and manage dependencies and publishing.

Then, you can install all the dependencies with:

```
make requirements
```

Next, you need to define the variables that are used in each plugins (see plugins respected README files for the all variables) in your `PrivateRules.mak` (needed for makefile targets) and, `.env` file (needed tp tests from vscode and interactive docker development). i.e. for GitLab MR and JIRA Support services:

```
PRIVATE_TOKEN=...
REQUESTER=...
GITLAB_TOKEN=...
GITLAB_HEADER=...
JIRA_URL=...
JIRA_USERNAME=...
JIRA_PASSWORD=...
SLACK_BOT_TOKEN=...
UNLEASH_API_URL=...
```

(continues on next page)

(continued from previous page)

```
UNLEASH_INSTANCE_ID=...
UNLEASH_ENVIRONMENT=...
RTD_TOKEN=...
GOOGLE_API_KEY=...
GOOGLE_SPREADSHEET_ID=...
NEXUS_HMAC_SIGNATURE_SECRET=...
OPENID_SECRET=...
GITLAB_CLIENT_ID=...
GITLAB_CLIENT_SECRET=...
```

Now, the project is ready for local development.

Note: depending on the IDE (*vscode is suggested*), `PYTHONPATH` may need to be adjusted so that IDE picks up imports and tests correctly. Please refrain from changing the main structure (*top level folders*) as it may break the CI/CD pipeline, make targets and the very fabric of the universe may be at stake.

2.2 Linting and code-style

The project follows PEP8 standard closely.

The linting step uses `black, flake8, isort and pylint` tools to check to code. It maybe useful to adjust your local environment as such as well.

Run `make lint` to check your code to see any linting errors. `make apply-formatting` could also be used to auto adjust the code style with `black` and `isort`. Note this also includes tests as well.

2.3 Building and Running

k8s note: if you are deploying or testing locally using Minikube, you should first run `eval $(minikube docker-env)` before you create your images, so that Minikube will pull the images directly from your environment.

To build the project for local development (and releasing in later stages), run `make build`. This will build a docker image (if a tag is present it will also tag it accordingly, or a *dev* tag will be used) and will build the python package as well.

To run/deploy the project, you can use Docker and Kubernetes as described below.

2.3.1 With Docker

Testing with Docker only is also possible: `make development` starts the latest built docker image (`make docker-build`) with app folder mounted into it and the server is set to `--reload` flag. This enables local development by reflecting any change in your app/ folder to loaded into the api server as well.

2.3.2 With Kubernetes

An example minikube installation with loadbalancer enabled could be found here - this is the suggested starting point for testing locally with Minikube.

You want to install charts using the docker image created with `make docker-build`. If you ran `eval $(minikube docker-env)` before building, the image will be pulled from your local cache.

Next, you want to deploy your charts. `make install-chart` deploys the helm chart into a k8s environment using the default configuration with the following ingress controllers:

- NGINX
- Traefik

By default, it uses `nginx` for local development and testing. You can override this by providing `INGRESS` variable like `make install-chart INGRESS=traefik`. In deployment correct ingress is automatically selected.

Using `make template-chart` it is possible to inspect the actual deployment that will happen with `make install-chart`.

2.3.3 With VSCode

To run the app directly from VSCode for debugging purposes. Create a `launch.json` under your workspace configuration folder (.vscode by default):

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Python: FastAPI",
      "type": "python",
      "request": "launch",
      "module": "uvicorn",
      "args": ["app.main:app"],
      "jinja": true,
      "envFile": "${workspaceFolder}/.env"
    }
  ]
}
```

2.4 Adding a new cicd automation service

In order to add a new plugin to the main host, it is required to develop it according to the following structure:

```
.
├── app
│   ├── plugins
│   │   ├── thenewplugin
│   │   │   ├── mainmodule
```

(continues on next page)

(continued from previous page)

```
│   │   │   │   ├── mainmodule.py
│   │   │   ├── tests
│   │   │   │   ├── unit
│   │   │   │   │   ├── unit_test_1.py
│   │   │   │   │   ├── unit_test_2.py
```

The plugin must have the following features:

- there must be one module containing a router object created according to the FastAPI documentation;
- a plugin configuration must be added into this folder;
- the above plugin configuration must be also added in the default configuration file;

The configuration item corresponds to the input parameter prefix, tags and a name of the include router operation of the FastAPI framework. There's another parameter in the configuration file which specify the package and module file name where to find the router object to be added in the host.

The plugin should also have a README file on its root folder.

2.4.1 Authentication

Every plugin should have his type of authentication, at the moment only 2 types of authentication can be used:

- Token Authentication via HTTP header (i.e. GitLab and Slack)
- Username and Password

The authentication configuration should be on plguin's configuration file, example: this file. Where it must be added 3 variables, being 2 of them dependent of the type of authentication:

- `auth_type` which represents the type of authentication that the plugin will have. It can be equal to `token` or `password`.
- `Token`:
  - `token_env` name of the environment variable that contains the token of the plugin .
  - `token_header_env` name of the environment variable that corresponds to the type of token header of the service to be authenticated, for example for gitlab the token header would be `X-Gitlab-Token`.
- `Username` and `Password`:
  - `username_env` name of the environment value that contains the username for the plugin authentication.
  - `password_env` name of the environment value that contains the password for the plugin authentication.

2.5 Testing With FastAPI and Pytest-BDD

The tests are done with pytest-bdd style, and are located on the testing folder on a server.feature and a test_server.py. To add new tests edit only server.feature file.

BDD style tests are created that test the correct functioning of:

- Check if Main route gives the Right response
- Post using a .json file (Get cannot be done with body)
- Get using a target (Post can be done only with target as well)

Because FastAPI is being used, tests are done by using the FastAPI TestClient - you can read more about it here.

To run all tests:

```
$ make unit_test
                                     .
                                     .
                                     .
platform linux -- Python 3.6.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/clean/ska-cicd-automation/testing, configfile: pytest.ini
plugins: bdd-4.0.1
collected 3 items

test_server.py::test_check_server PASSED                                                 ␣
↪     [ 33%]
test_server.py::test_add_member_with_json PASSED                                         ␣
↪     [ 66%]
test_server.py::test_get_member_id PASSED                                                ␣
↪     [100%]
```

To run tests for an individual plugin, pass the `PLUGIN` name:

```
make unit_test PLUGIN=gitlab_mr
```

2.5.1 Manual Testing

To debug manually using an actual MR. Change the `project:id`, MR `object_attributes:iid` and MR `object_attributes:source_branch` and any other fields you would like in `tests/unit/gitlab_mr/files/event.json`. Then, using your IDE of choice, implement breakpoints to debug.

2.6 Publishing/Releasing

All the publishing should happen from the pipeline.

TL;DR: run `make release` *to learn what you have to do!*

When you are ready to publish a new version, you need to run `make update-x-release` where `x` is either `patch`, `minor` or `major`. So if you want to update the patch version, you just run `make update-patch-release`.

This will update the necessary version labels in `.release` (for docker image) , `pyproject.toml` (for python package) files and will make a commit and tag it accordingly. At this stage, you can use `make push` to manually push the docker image to your configured registry although it is not encouraged.

Finally, run `make release`. Once the CI job has completed in the pipeline, make sure you trigger the manual step on the tag ref for publishing either for docker/python or deploying the helm chart.

CHAPTER

THREE

MERGE REQUEST CHECKS

These are all the packages, functions and scripts that form part of the project.

- GITLAB_README
- JIRA_README