ci-testing-python Documentation

Release 0.1.0

Anirban Roy Das

Sep 27, 2017

Contents

Sample Microservice App in Python for CI-CD and Testing Purpose using flask, pytest, pytest-flask, uber's test doubles package, tox with Dockerized environment on CI servers like Jenkins and Travis CI both application in docker and jenkins in docker.

CHAPTER 1

Project Home Page

Link : https://github.com/anirbanroydas/ci-testing-python

CHAPTER 2

Details

Author Anirban Roy Das

Email anirban.nick@gmail.com

Contents:

Overview

It is a Sample **Microservice** Application written in **Python** Language for **CI-CD** and **Testing Purpose** using flask, pytest, pytest-flask, uber's test doubles package, tox with Dockerized environment and can used to learn, experiment with docker, testing, pytest and have beginner's introduction/hands-on with CI servers like Jenkins and Travis-CI.

Features

Technical Specs

python 3.6 Python Language (Cpython)

Flask Micro Python Web Framework, good for microservice development and python WSGI apps.

pytest Python testing library and test runner with awesome test discobery

pytest-flask Pytest plugin for flask apps, to test fask apps using pytest library.

Uber's Test-Double Test Double library for python, a good alternative to the mock library

Jenkins (Optional) A Self-hosted CI server

Travis-CI (Optional) A hosted CI server free for open-source projecs

Docker A containerization tool for better devops

Feature Specs

- Web App
- Microservice
- Testing using Docker and Docker Compose
- CI servers like Jenkins, Travis-CI

Installation

Prerequisites (Optional)

To safegurad secret and confidential data leakage via your git commits to public github repo, check `git-secrets`.

This git secrets project helps in preventing secrete leakage by mistake.

Dependencies

1. Docker
2. Make (Makefile)

See, there are so many technologies used mentioned in the tech specs and yet the dependencies are just two. This is the power of Docker.

Install

- **Step 1 - Install Docker**

  Follow my another github project, where everything related to DevOps and scripts are mentioned along with setting up a development environemt to use Docker is mentioned.

  – Project: https://github.com/anirbanroydas/DevOps

  – Go to setup directory and follow the setup instructions for your own platform, linux/macos

- **Step 2 - Install Make**

```
# (Mac Os)
$ brew install automake

# (Ubuntu)
$ sudo apt-get update
$ sudo apt-get install make
```

- **Step 3 - Install Dependencies**

  Install the following dependencies on your local development machine which will be used in various scripts.

  1. openssl
  2. ssh-keygen

3. openssh

Travis Setup

These steps will encrypt your environment variables to secure your confidential data like api keys, docker based keys, deploy specific keys.

```
$ make travis-setup
```

Jenkins Setup

These steps will encrypt your environment variables to secure your confidential data like api keys, docker based keys, deploy specific keys.

```
$ make jenkins-setup
```

CI Setup

If you are using the project in a CI setup (like travis, jenkins), then, on every push to github, you can set up your travis build or jenkins pipeline. Travis will use the `.travis.yml` file and Jenknis will use the `Jenkinsfile` to do their jobs. Now, in case you are using Travis, then run the Travis specific setup commands and for Jenkins run the Jenkins specific setup commands first. You can also use both to compare between there performance.

The setup keys read the values from a `.env` file which has all the environment variables exported. But you will notice an example `env` file and not a `.env` file. Make sure to copy the `env` file to `.env` and **change/modify** the actual variables with your real values.

The `.env` files are not commited to git since they are mentioned in the `.gitignore` file to prevent any leakage of confidential data.

After you run the setup commands, you will be presented with a number of secure keys. Copy those to your config files before proceeding.

NOTE: This is a one time setup. **NOTE:** Check the setup scripts inside the `scripts/` directory to understand what are the environment variables whose encrypted keys are provided. **NOTE:** Don't forget to **Copy** the secure keys to your `.travis.yml` or `Jenkinsfile`

NOTE: If you don't want to do the copy of `env` to `.env` file and change the variable values in `.env` with your real values then you can just edit the `travis-setup.sh` or `jenknis-setup.sh` script and update the values their directly. The scripts are in the `scripts/` project level directory.

IMPORTANT: You have to run the `travis-setup.sh` script or the `jenkins-setup.sh` script in your local machine before deploying to remote server.

Usage

After having installed the above dependencies, and ran the **Optional** (If not using any CI Server) or **Required** (If using any CI Server) **CI Setup** Step, then just run the following commands to use it:

You can run and test the app in your local development machine or you can run and test directly in a remote machine. You can also run and test in a production environment.

Run

The below commands will start everythin in development environment. To start in a production environment, suffix `-prod` to every **make** command.

For example, if the normal command is `make start`, then for production environment, use `make start-prod`. Do this modification to each command you want to run in production environment.

Exceptions: You cannot use the above method for test commands, test commands are same for every environment. Also the `make system-prune` command is standalone with no production specific variation (Remains same in all environments).

- **Start Applcation**

```
$ make clean
$ make build
$ make start

# OR

$ docker-compose up -d
```

- **Stop Application**

```
$ make stop

# OR

$ docker-compose stop
```

- **Remove and Clean Application**

```
$ make clean

# OR

$ docker-compose rm --force -v
$ echo "y" | docker system prune
```

- **Clean System**

```
$ make system-prune

# OR

$ echo "y" | docker system prune
```

Logging

- To check the whole application Logs

```
$ make check-logs

# OR

$ docker-compose logs --follow --tail=10
```

- To check just the python app's logs

```
$ make check-logs-app

# OR

$ docker-compose logs --follow --tail=10 identidock
```

Testing

Now, testing is the main deal of the project. You can test in many ways, namely, using `make` commands as mentioned in the below commands, which automates everything and you don't have to know anything else, like what test library or framework is being used, how the tests are happening, either directly or via `docker` containers, or may be different virtual environments using `tox`. Nothing is required to be known.

On the other hand if you want fine control over the tests, then you can run them directly, either by using `pytest` commands, or via `tox` commands to run them in different python environments or by using `docker-compose` commands to run differetn tests.

But running the make commands is lawasy the go to strategy and reccomended approach for this project.

NOTE: Tox can be used directly, where `docker` containers will not be used. Although we can try to run `tox` inside our test contianers that we are using for running the tests using the `make` commands, but then we would have to change the `Dockerfile` and install all the `python` dependencies like `python2.7`, `python3.x` and then run `tox` commands from inside the `docker` containers which then run the `pytest` commands which we run now to perform our tests inside the current test containers.

CAVEAT: The only caveat of using the make commands directly and not using `tox` is we are only testing the project in a single `python` environment, nameley `python 3.6`.

- To Test everything

  ```
  $ make test
  ```

  Any Other method without using make will involve writing a lot of commands. So use the make command preferrably

- To perform Unit Tests

  ```
  $ make test-unit
  ```

- To perform Component Tests

  ```
  $ make test-component
  ```

- To perform Contract Tests

  ```
  $ make test-contract
  ```

- To perform Integration Tests

  ```
  $ make test-integration
  ```

- To perform End To End (e2e) or System or UI Acceptance or Functional Tests

```
$ make test-e2e

# OR

$ make test-system

# OR

$ make test-ui-acceptance

# OR

$ make test-functional
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search