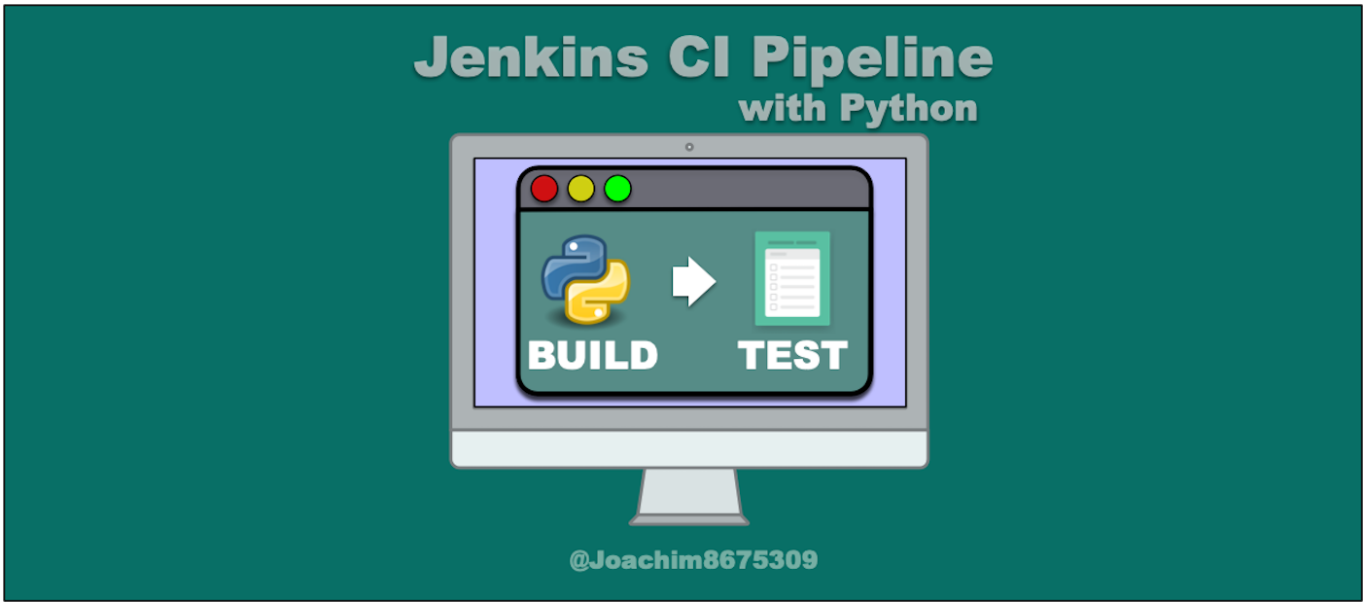


Jenkins CI Pipeline with Python

Building a Jenkins CI Pipeline with Flask + unittest

Joaquín Menchaca (智裕) Follow
Mar 1, 2019 · 6 min read

This is an introductory article about building and testing **Python** web api service with **Jenkins** CI (continuous integration) pipeline (using `Jenkinsfile`). For this process, I'll demonstrate how to:

- build a small HelloWorld API with **Flask**, a popular web microframework originally released in 2010
- create some **xUnit** style unit tests for the service
- how to integrate this into **Jenkins** with JUnit test reporting support

This is a *minimal* material to get started, so we will only have a *build* and *test* stage. In professional setting, we would actually want to have a third stage called *push*, to conditionally push an artifact, the output of the CI, to an artifact repository, such as the **CheeseShop** (**PyPI**) site for a pip modules, or a **Docker** image to a **Docker** registry.

Part 1: The Web Application

The web application is a simple Hello World web application with essentially three routes: `/` , `/hello/` , and `/hello/<name>` , where name is any name you desire.

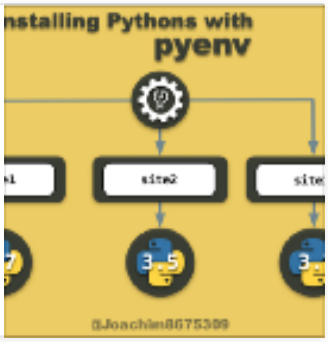
Get Python

First we need to get a Python 3 (Python 3.7.2 is the current version). I highly recommend using a python version manager like **pyenv** to install Python. I wrote a previous article on this topic:

Installing Pythons with PyEnv

Using pyenv to manage Python 2 and Python 3 environments

medium.com

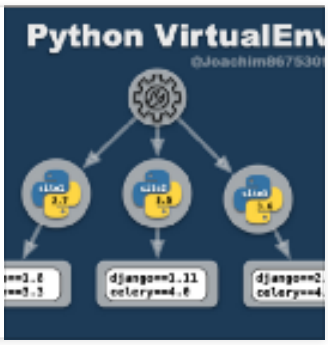


I also recommend using something like **VirtualEnv** to keep pip packages installed for this project separate from system packages:

Python VirtualEnv

Got to Keep 'em Separated

medium.com



After installing **Python** and optionally initializing a virtual environment, we will need to install the **Flask** web-microframework as well as the **WSGI** (**Web Service Gateway Interface**) server **Werkzeug**. We can do this by creating a package manifest called `requirements.txt` and then installing the packages with pip using these bash commands:

```
cat <<-'PACKAGE_MANIFEST' > requirements.txt
Click==7.0
Flask==1.0.2
itsdangerous==1.1.0
```

```
Jinja2==2.10
MarkupSafe==1.1.0
Werkzeug==0.14.1
xmlrunner==1.7.7
PACKAGE_MANIFEST

# install packages
pip install -r requirements.txt
```

The Application

For the application create a file called `app.py` with the following contents:

```
#!/usr/bin/env python
from flask import Flask
app = Flask(__name__)

@app.route('/')
@app.route('/hello/')
def hello_world():
    return 'Hello World!\n'

@app.route('/hello/<username>') # dynamic route
def hello_user(username):
    return 'Why Hello %s!\n' % username

if __name__ == '__main__':
    app.run(host='0.0.0.0')     # open for everyone
```

You can try the server out with `python app.py` or with:

```
# make script executable & run service
chmod +x app.py
./app.py &

# test the server
curl -i localhost:5000/
curl -i localhost:5000/hello/
curl -i localhost:5000/hello/Simon
```

Part 2: The Unit Tests

Before we tested the application with three routes: `/` , `/hello/`, and `/hello/Simon`. Now we can write some tests to test these routes.

Create the Tests

Run this in bash to create our test cases:

```
cat <<-'TEST_CASES' > test.py
#!/usr/bin/env python
import unittest
import app

class TestHello(unittest.TestCase):

    def setUp(self):
        app.app.testing = True
        self.app = app.app.test_client()

    def test_hello(self):
        rv = self.app.get('/')
        self.assertEqual(rv.status, '200 OK')
        self.assertEqual(rv.data, b'Hello World!\n')

    def test_hello_hello(self):
        rv = self.app.get('/hello/')
        self.assertEqual(rv.status, '200 OK')
        self.assertEqual(rv.data, b'Hello World!\n')

    def test_hello_name(self):
        name = 'Simon'
        rv = self.app.get(f'/hello/{name}')
        self.assertEqual(rv.status, '200 OK')
        self.assertIn(bytearray(f"{name}", 'utf-8'), rv.data)

if __name__ == '__main__':
    unittest.main()
TEST_CASES
chmod +x test.py
```

Code Details

These tests will use a **xUnit** style of tests with the `unittest` library that comes bundled with the install of Python. To get started with `unittest`, you want to create a class that is inherited from the `unittest.TestCase` class, and then create methods that begin with `test_` prefix for each of your tests.

In this example, we need to create a `setUp()` method that uses your instances of the `Flask` class, and call the `instance_name.app.test_client()`. As our instance is called `app` (from `app.py`), we will then use `app.app.test_client()`. This way when calling the `self.app.get()` method, it will utilize your instance of the `Flask` class from your code logic.

The **test_client** (`app.test_client()`) is a method provided the **Flask application object**, which creates a test client for the application. This is what we use in conjunction with `unittest` and asserts.

When calling the `get()` method, the data returned is in the `bytearray`, so we must use b-string or `b'string'` for comparisons. In one of the tests, `test_hello_name()`, we use an f-string (`f'string'`) with the mock data of `Simon`, which we coerce to a `bytearray` for the final comparison.

Running the Tests

To run the tests, we simply run something like:

```
./test.py
```

We'll get some output like this:

```
Running tests...
----------------------------------------------------------------------
...
----------------------------------------------------------------------
Ran 3 tests in 0.009s

OK
```

Part 3: The Jenkins Pipeline

Now that we have our web application and unit tests, we can create a **Jenkins** CI Pipeline by creating a `Jenkinsfile`. The `Jenkinsfile` is a **Groovy** script, and can use a

DSL-like syntax to define our stages and shell instructions.

The Jenkinsfile

We'll have two stages: *build* and *test* for our current pipeline. Use this bash command to create the `Jenkinsfile`:

```
cat <<-'JENKINSFILE' > Jenkinsfile
pipeline {
  agent { docker { image 'python:3.7.2' } }
  stages {
    stage('build') {
      steps {
        sh 'pip install -r requirements.txt'
      }
    }
    stage('test') {
      steps {
        sh 'python test.py'
      }
    }
  }
}
JENKINSFILE
```

When this is used by a **Jenkins** agent, it will download a **Docker** image with **Python** environment installed. For *build* and *test* stages, the pipeline will run a shell command, similar to have we have already ran in our previous steps, in the **Python** container.

Running a Jenkins Server Locally

Jenkins has docker image that contains everything we need for this project. We can run this container for all of our **Jenkins** needs. I have a tutorial on running this locally in your development system, as long as you have **Docker** installed.

Import the Project

After logging in to your **Jenkins** server, you'll want to import a pipeline. This code will have to be checked into a **Git** repository (or other Source Code Manager), and then configured to fetch the `Jenkinsfile` from that repository.

I have a small project you can use with the code for this repository:

Test Report Integration

Jenkins has the ability present test results in a graphical visual way, as long as you can output the results in a **JUnit** format. **JUnit** is a popular **xUnit** type of test framework, and **JUnit** output format (an **XML** file) is ubiquitous test reporting. Essentially, any CI (Continuous Integration) solution will support this format, including **Jenkins**.

For this integration, we can use the **XMLRunner** library, and pass this as our test runner to the `unittest.main()` method.

Update these few lines at the bottom of the script `test.py` so that it looks like this:

```
if __name__ == '__main__':
    ############# Add these lines #############
    import xmlrunner
    runner = xmlrunner.XMLTestRunner(output='test-reports')
    unittest.main(testRunner=runner)
    ###########################################
    unittest.main()
```

This will import a library called `xmlrunner` and do a `unittest.main()` run with `XMLTestRunner`. After, will do another run to show output to the standard output. This will generate test reports in the `test-reports` directory.

We need to update the `Jenkinsfile` to have a final *post* step in the *test* stage, that tells **Jenkins** where to find the **JUnit** test report. Update `Jenkinsfile` to look like this:

```
pipeline {
  agent { docker { image 'python:3.7.2' } }
  stages {
    stage('build') {
      steps {
        sh 'pip install -r requirements.txt'
      }
    }
    stage('test') {
      steps {
        sh 'python test.py'
      }
      post {
        always {
          junit 'test-reports/*.xml'
        }
      }
    }
  }
}
```

In Jenkins, run this pipeline again, and you'll see results under the Test link in the Blue Ocean interface.

Final Thoughts

There you have it, we did the following with this tutorial:

- Created a **Flask** web api application
- Created Unit Tests with **JUnit** reporting integration
- Created a pipeline (`Jenkinsfile`) that will run the tests and provide visual feedback in Jenkins.

In a professional setting for **Flask** or **Python** applications, we would want to also add a few things, which were not covered in this tutorial:

- *Code commit to main or release branch*: when tests pass, push an artifact, such as pip package or docker image, or an artifact repository
- *Submission of pull/merge request*: run tests and provide feedback to git server, such as **GitHub** or **GitLab**, and block submission approval if tests fail.

For visual feed back to the a git server, like **GitHub** or **GitLab**, you'll need to use a combination:

- webhooks on those services
- token generated with authorization to access the repository (or an account with access)
- Jenkins plug-in (**GitLab plugin** and **GitHub plugin**)

I hope this was useful. Happy hacking.

Docker Jenkins Python Flask Unit Testing