Jenkins CI Pipeline with Python

Building a Jenkins CI Pipeline with Flask + unittest

 [**Python**](https://www.python.org/) web api service with [**Jenkins**](https://jenkins.io/) CI (continuous integration) pipeline (using Jenkinsfile). For this process, I’ll demonstrate how to:

* build a small HelloWorld API with [**Flask**](http://flask.pocoo.org/), a popular web microframework originally released in 2010
* create some **[xUnit](https://en.wikipedia.org/wiki/XUnit" \t "_blank)** style unit tests for the service
* how to integrate this into [**Jenkins**](https://jenkins.io/)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](https://pypi.org/" \t "_blank)** (**[PyPI](https://pypi.org/" \t "_blank)**) site for a pip modules, or a [**Docker**](https://www.docker.com/) image to a [**Docker**](https://www.docker.com/) 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](https://github.com/pyenv/pyenv" \t "_blank)**to install Python

I also recommend using something like **[VirtualEnv](https://virtualenv.pypa.io/en/latest/" \t "_blank)** to keep pip packages installed for this project separate from system packages:

After installing [**Python**](https://www.python.org/) and optionally initializing a virtual environment, we will need to install the [**Flask**](http://flask.pocoo.org/) web-microframework as well as the [**WSGI**](https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface) ([**Web Service Gateway Interface**](https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface)) server[**Werkzeug**](http://werkzeug.pocoo.org/). 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](https://en.wikipedia.org/wiki/XUnit" \t "_blank)** 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](http://flask.pocoo.org/docs/1.0/api/" \l "flask.Flask.test_client" \t "_blank)** (app.test\_client()) is a method provided the [**Flask application object**](http://flask.pocoo.org/docs/1.0/api/), 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**](https://jenkins.io/) CI Pipeline by creating a Jenkinsfile. The Jenkinsfile is a [**Groovy**](http://groovy-lang.org/) script, and can use a [**DSL**](https://en.wikipedia.org/wiki/Domain-specific_language)-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**](https://jenkins.io/) agent, it will download a [**Docker**](https://www.docker.com/) image with [**Python**](https://www.python.org/) 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**](https://www.python.org/) container.

## Running a Jenkins Server Locally

[**Jenkins**](https://jenkins.io/) has docker image that contains everything we need for this project. We can run this container for all of our [**Jenkins**](https://jenkins.io/) needs. I have a tutorial on running this locally in your development system, as long as you have [**Docker**](https://www.docker.com/) installed.

## Import the Project

After logging in to your [**Jenkins**](https://jenkins.io/) server, you’ll want to import a pipeline. This code will have to be checked into a **[Git](https://git-scm.com/" \t "_blank)** 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**](https://jenkins.io/) has the ability present test results in a graphical visual way, as long as you can output the results in a [**JUnit**](https://junit.org/) format. [**JUnit**](https://junit.org/) is a popular **[xUnit](https://en.wikipedia.org/wiki/XUnit" \t "_blank)** type of test framework, and [**JUnit**](https://junit.org/) output format (an [**XML**](https://www.w3.org/XML/) file) is ubiquitous test reporting. Essentially, any CI (Continuous Integration) solution will support this format, including [**Jenkins**](https://jenkins.io/).

For this integration, we can use the **[XMLRunner](https://github.com/pycontribs/xmlrunner" \t "_blank)** library, and pass this as our test runner to the unittest.main() method.

Insert these few lines at the bottom of the script app.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**](https://jenkins.io/) where to find the [**JUnit**](https://junit.org/) 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**](http://flask.pocoo.org/) web api application
* Created Unit Tests with [**JUnit**](https://junit.org/) reporting integration
* Created a pipeline (Jenkinsfile) that will run the tests and provide visual feedback in Jenkins.

In a professional setting for [**Flask**](http://flask.pocoo.org/) or [**Python**](https://www.python.org/) 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**](https://github.com/) or **[GitLab](https://gitlab.com/" \t "_blank)**, and block submission approval if tests fail.

For visual feed back to the a git server, like [**GitHub**](https://github.com/) or **[GitLab](https://gitlab.com/" \t "_blank)**[,](https://gitlab.com/" \t "_blank) 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](https://github.com/jenkinsci/gitlab-plugin" \t "_blank)** and [**GitHub plugin**](https://github.com/KostyaSha/github-integration-plugin/))