Build Pipeline with a Python app

Install Docker & Jenkins

1. Create a bridge network in Docker using the following docker network create command:

```
$ docker network create jenkins
470a40abd7f2e06b22886827f162b3f1db7ea2fd535b242326b82482fd42b205
```

2. Create the following volumes to share the Docker client TLS certificates needed to connect to the Docker daemon and persist the Jenkins data using the following docker volume create commands:

```
$ docker volume create jenkins-docker-certs
jenkins-docker-certs
$ docker volume create jenkins-data
jenkins-data
```

3. In order to execute Docker commands inside Jenkins nodes, download and run the **docker:dind** Docker image using the following docker container run command:

```
$ docker container run --name jenkins-docker --rm --detach \
  --privileged --network jenkins --network-alias docker \
  --env DOCKER_TLS_CERTDIR=/certs \
  --volume jenkins-docker-certs:/certs/client \
  --volume jenkins-data:/var/jenkins_home \
  --volume "$HOME":/home docker:dind
```

4. Run the jenkinsci/blueocean image as a container in Docker using the following docker container run command (bearing in mind that this command automatically downloads the image if this hasn't been done):

```
docker container run --name jenkins-tutorial --rm --detach \
  --network jenkins --env DOCKER_HOST=tcp://docker:2376 \
  --env DOCKER_CERT_PATH=/certs/client --env DOCKER_TLS_VERIFY=1 \
  --volume jenkins-data:/var/jenkins_home \
  --volume jenkins-docker-certs:/certs/client:ro \
  --volume "$HOME":/home --publish 8080:8080 jenkinsci/blueocean
```

(1) Maps the /var/jenkins_home directory in the container to the Docker volume with the name jenkins-data. If this volume does not exist, then this docker container run command will automatically create the volume for you.

(2) Maps the $HOME directory on the host (i.e. your local) machine (usually the /Users/ directory) to the /home directory in the container.

Fork and clone the sample repository on GitHub

Obtain the simple "add" Python application from GitHub, by forking the sample repository of the application's source code into your own GitHub account and then cloning this fork locally.

1. Ensure you are signed in to your GitHub account.
2. Fork the simple-python-pyinstaller-app on GitHub into your local GitHub account. If you need help with this process, refer to the Fork A Repo documentation on the GitHub website for more information.
3. Clone your forked `simple-python-pyinstaller-app repository` (on GitHub) locally to your machine. To begin this process, do either of the following (where is the name of your user account on your operating system):

- Open up a terminal/command line prompt and cd to the appropriate directory on:

macOS - `/Users/<your-username>/Documents/GitHub/`

Linux - `/home/<your-username>/GitHub/`
(For ec2 instance: `home/ec2-user/GitHub`)

Windows - `C:\Users\<your-username>\Documents\GitHub\` (although use a Git bash command line window as opposed to the usual Microsoft command prompt)

4. Run the following command to continue/complete cloning your forked repo:

```
git clone https://github.com/YOUR-GITHUB-ACCOUNT-NAME/simple-python-pyinstaller-
```

where YOUR-GITHUB-ACCOUNT-NAME is the name of your GitHub account.

Create your Pipeline project in Jenkins

1. We can access our jenkins in http://localhost:8080 (your ec2 instance IPv4 Public IP:8080)
2. To access the admin password, execute the following commands:

```
$ docker container exec -it jenkins-tutorial bash
$ cat /var/jenkins_home/secrets/initialAdminPassword
```

3. After you log in to Jenkins, click create new jobs under Welcome to Jenkins! Note: If you don't see this, click New Item at the top left.
4. In the Enter an item name field, write a name for your new Pipeline project.
5. Scroll down and click Pipeline, then click OK at the end of the page.
6. (Optional) On the next page, specify a brief description for your Pipeline in the Description field (e.g. An entry-level Pipeline demonstrating how to use Jenkins to build a simple Python application)
7. Click the Pipeline tab at the top of the page to scroll down to the Pipeline section.
8. From the Definition field, choose the Pipeline script from SCM option. This option instructs Jenkins to obtain your Pipeline from Source Control Management (SCM).
9. From the SCM field, choose Git.
10. In the Repository URL field, specify the directory path of your locally cloned repository above, which is from your user account/home directory on your host machine, mapped to the /home directory of the Jenkins container - i.e.

- For macOS - `/home/Documents/GitHub/simple-python-pyinstaller-app`
- For Linux - `/home/GitHub/simple-python-pyinstaller-app`

(For ec2 instance: `/home/GitHub/simple-python-pyinstaller-app)`

- For Windows - `/home/Documents/GitHub/simple-python-pyinstaller-app`

The Jenkinsfile

You're now ready to create your Pipeline that will automate building your Python application. Your Pipeline will be created as a Jenkinsfile, which will be committed to your locally cloned Git repository (simple-python-pyinstaller-app).

This is the foundation of "Pipeline-as-Code", which treats the continuous delivery pipeline a part of the application to be versioned and reviewed like any other code.

First, create an initial Pipeline with a "Build" stage that executes the first part of the entire production process for your application. This "Build" stage downloads a Python Docker image and runs it as a Docker container, which in turn compiles your simple Python application into byte code.

1. Using a text editor, create and save new text file with the name Jenkinsfile at the root of your local simple-python-pyinstaller-app Git repository.
2. Copy the following Declarative Pipeline code and paste it into your empty Jenkinsfile:

The Jenkinsfile in the repository is

```
pipeline {
    agent none       //(1)
    stages {
        stage('Build') {       //(2)
            agent {
                docker {
                    image 'python:2-alpine'       //(3)
                }
            }
            steps {
                sh 'python -m py_compile sources/add2vals.py sources/calc.py'
                stash(name: 'compiled-results', includes: 'sources/*.py*')
            }
        }
    }
}
```

(1) The agent section with the none parameter specified at the top of this Pipeline code block means that no global agent will be allocated for the entire Pipeline's execution and that each stage directive must specify its own agent section.

(2) Defines a stage (directive) called Build that appears on the Jenkins UI. This image parameter (of the agent section's docker parameter) downloads the python:2-alpine Docker image (if it's not already available on your machine) and runs this image as a separate container. This means that:

(3)

- You'll have separate Jenkins and Python containers running locally in Docker.
- The Python container becomes the agent that Jenkins uses to run the Build stage of your Pipeline project. However, this container is short-lived - its lifespan is only that of the duration of your Build stage's execution.

(4) This sh step (of the steps section) runs the Python command to compile your application and its calc library into byte code files (each with .pyc extension), which are placed into the sources workspace directory (within the /var/jenkins_home/workspace/simple-python-pyinstaller-app directory in the Jenkins container).

(5) This stash step (of the basic steps section) saves the Python source code and compiled byte code files (with .pyc extension) from the sources workspace directory for use in later stages.

3. Save your edited Jenkinsfile and commit it to your local simple-python-pyinstaller-app Git repository.

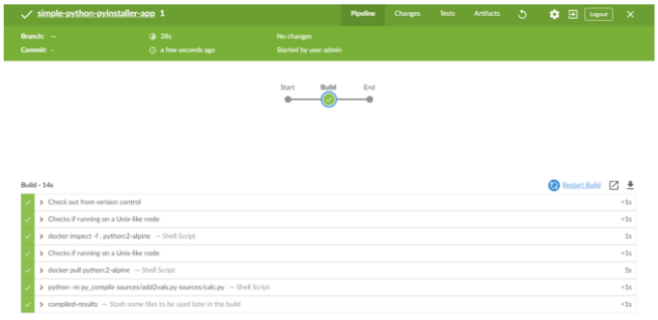
```
git commit -m "Add initial Jenkinsfile"
```

4. Go back to Jenkins again, log in again if necessary and click Open Blue Ocean on the left to access Jenkins's Blue Ocean interface.

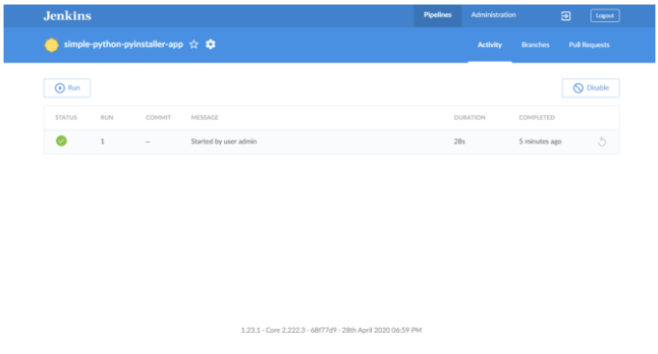
5. In the This job has not been run message box, click Run, then quickly click the OPEN link which appears briefly at the lower-right to see Jenkins running your Pipeline project. If you weren't able to click the OPEN link, click the row on the main Blue Ocean interface to access this feature.

> **Note:**
> - You may need to wait a few minutes for this first run to complete. After making a clone of your local simple-python-pyinstaller-app Git repository itself, Jenkins:
>
> a. Initially queues the project to be run on the agent.
>
> b. Runs the Build stage (defined in the Jenkinsfile) on the Python container. During this time, Python uses the py_compile module to compile the code of your Python application and its calc library into byte code, which are stored in the sources workspace directory (within the Jenkins home directory).

The Blue Ocean interface turns green if Jenkins compiled your Python application successfully.



6. Click the X at the top-right to return to the main Blue Ocean interface.



Add a test stage to your Pipeline

1. Go back to your text editor and open your Jenkinsfile.
2. Copy and paste the following Declarative Pipeline syntax immediately under the Build stage of your Jenkinsfile:

.

```
        stage('Test') {
            agent {
                docker {
                    image 'qnib/pytest'
                }
            }
            steps {
                sh 'py.test --junit-xml test-reports/results.xml sources/test_ca
            }
            post {
                always {
                    junit 'test-reports/results.xml'
                }
            }
        }
```

so that you end up with:

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:2-alpine'
                }
            }
            steps {
                sh 'python -m py_compile sources/add2vals.py sources/calc.py'
                stash(name: 'compiled-results', includes: 'sources/*.py*')
            }
        }
        stage('Test') {       //(1)
            agent {
                docker {
                    image 'qnib/pytest'       //(2)
                }
            }
            steps {
                sh 'py.test --junit-xml test-reports/results.xml sources/test_ca
            }
            post {
                always {
                    junit 'test-reports/results.xml'       //(4)
                }
            }
        }
    }
}
```

(1) Defines a stage (directive) called Test that appears on the Jenkins UI.

(2) This image parameter (of the agent section's docker parameter) downloads the qnib:pytest Docker image (if it's not already available on your machine) and runs this image as a separate container. This means that:

- You'll have separate Jenkins and pytest containers running locally in Docker.
- The pytest container becomes the agent that Jenkins uses to run the Test stage of your Pipeline project. This container's lifespan lasts the duration of your Test stage's execution.

(3) This sh step (of the steps section) executes pytest's py.test command on sources/test_calc.py, which runs a set of unit tests (defined in test_calc.py) on the "calc" library's add2 function (used by your simple Python application add2vals). The:

- --junit-xml test-reports/results.xml option makes py.test generate a JUnit XML report, which is saved to test-reports/results.xml (within the /var/jenkins_home/workspace/simple-python-pyinstaller-app directory in the Jenkins container).

(4) This junit step (provided by the JUnit Plugin) archives the JUnit XML report (generated by the py.test command above) and exposes the results through the Jenkins interface. In Blue Ocean, the results are accessible through the Tests page of a Pipeline run. The post section's always condition that contains this junit step ensures that the step is always executed at the completion of the Test stage, regardless of the stage's outcome.

3. Save your edited Jenkinsfile and commit it to your local simple-python-pyinstaller-app Git repository.
4. Go back to Jenkins again, log in again if necessary and ensure you've accessed Jenkins's Blue Ocean interface.

5. Click Run at the top left, then quickly click the OPEN link which appears briefly at the lower-right to see Jenkins running your amended Pipeline project. If you weren't able to click the OPEN link, click the top row on the Blue Ocean interface to access this feature.

If your amended Pipeline ran successfully, here's what the Blue Ocean interface should look like. Notice the additional "Test" stage. You can click on the previous "Build" stage circle to access the output from that stage.

6. Click the X at the top-right to return to the main Blue Ocean interface.

Add a final deliver stage to your Pipeline

1. Go back to your text editor and open your Jenkinsfile.
2. Copy and paste the following Declarative Pipeline syntax immediately under the Test stage of your Jenkinsfile:

```
        stage('Deliver') {
            agent any
            environment {
                VOLUME = '$(pwd)/sources:/src'
                IMAGE = 'cdrx/pyinstaller-linux:python2'
            }
            steps {
                dir(path: env.BUILD_ID) {
                    unstash(name: 'compiled-results')
                    sh "docker run --rm -v ${VOLUME} ${IMAGE} 'pyinstaller -F ac
                }
            }
            post {
                success {
                    archiveArtifacts "${env.BUILD_ID}/sources/dist/add2vals"
                    sh "docker run --rm -v ${VOLUME} ${IMAGE} 'rm -rf build dist
                }
            }
        }
```

and add a `skipStagesAfterUnstable` option so that you end up with:

```
pipeline {
    agent none
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:2-alpine'
                }
            }
            steps {
                sh 'python -m py_compile sources/add2vals.py sources/calc.py'
                stash(name: 'compiled-results', includes: 'sources/*.py*')
            }
        }
        stage('Test') {
            agent {
                docker {
                    image 'qnib/pytest'
                }
            }
            steps {
                sh 'py.test --junit-xml test-reports/results.xml sources/test_ca
            }
            post {
                always {
                    junit 'test-reports/results.xml'
                }
            }
        }
        stage('Deliver') {       //(1)
            agent any
            environment {        //(2)
                VOLUME = '$(pwd)/sources:/src'
                IMAGE = 'cdrx/pyinstaller-linux:python2'
            }
            steps {
                dir(path: env.BUILD_ID) {        //(3)
                    unstash(name: 'compiled-results')        //(4)
                    sh "docker run --rm -v ${VOLUME} ${IMAGE} 'pyinstaller -F ac
                }
            }
            post {
                success {
                    archiveArtifacts "${env.BUILD_ID}/sources/dist/add2vals"
                    sh "docker run --rm -v ${VOLUME} ${IMAGE} 'rm -rf build dist
                }
            }
        }
    }
```

(1) Defines a stage (directive) called Deliver that appears on the Jenkins UI.

(2) This environment block defines two variables which will be used later in the 'Deliver' stage.

(3) This dir step (of the basic steps section) creates a new subdirectory named by the build number. The final program will be created in that directory by pyinstaller. BUILD_ID is one of the pre-defined Jenkins environment variables and is available in all jobs.

(4) This unstash step (of the basic steps section) restores the Python source code and compiled byte code files (with .pyc extension) from the previously saved stash. image] (if it's not already available on your machine) and runs this image as a separate container. This means that:

- You'll have separate Jenkins and PyInstaller (for Linux) containers running locally in Docker.
- The PyInstaller container becomes the agent that Jenkins uses to run the Deliver stage of your Pipeline project. This container's lifespan lasts the duration of your Deliver stage's execution.

(5) This sh step (of the steps section) executes the pyinstaller command (in the PyInstaller container) on the `pyinstaller` container. But you should notice that the volume option binds the jenkins container's sources directory to the newly created pyinstaller containers src directory. Pyinstaller bundles your add2vals.py Python application into a single standalone executable file (via the -F (or --onefile can be used) option) and outputs this file to the dist directory (under jenkins_home/workspace///sources/dist) working directory under the directory with build the build number). Although this step consists of a single command, as a general principle, it's a good idea to keep your Pipeline code (i.e. the Jenkinsfile) as tidy as possible and place more complex build steps (particularly for stages consisting of 2 or more steps) into separate shell script files like the deliver.sh file. This ultimately makes maintaining your Pipeline code easier, especially if your Pipeline gains more complexity.

(6) This archiveArtifacts step (provided as part of Jenkins core) archives the standalone executable file (generated by the pyinstaller command above at dist/add2vals within the Jenkins home's workspace directory) and exposes this file through the Jenkins interface. In Blue Ocean, archived artifacts like these are accessible through the Artifacts page of a Pipeline run. The post section's success condition that contains this archiveArtifacts step ensures that the step is executed at the completion of the Deliver stage only if this stage completed successfully.

3. Save your edited Jenkinsfile and commit it to your local simple-python-pyinstaller-app Git repository.
4. Go back to Jenkins again, log in again if necessary and ensure you've accessed Jenkins's Blue Ocean interface.
5. Click Run at the top left, then quickly click the OPEN link which appears briefly at the lower-right to see Jenkins running your amended Pipeline project. If you weren't able to click the OPEN link, click the top row on the Blue Ocean interface to access this feature.

If your amended Pipeline ran successfully, here's what the Blue Ocean interface should look like. Notice the additional "Deliver" stage. Click on the previous "Test" and "Build" stage circles to access the outputs from those stages.

6. Click the X at the top-right to return to the main Blue Ocean interface, which lists your previous Pipeline runs in reverse chronological order.

Summary

Well done! You've just used Jenkins to build a simple Python application!

The "Build", "Test" and "Deliver" stages you created above are the basis for building more complex Python applications in Jenkins, as well as Python applications that integrate with other technology stacks.

Because Jenkins is extremely extensible, it can be modified and configured to handle practically any aspect of build orchestration and automation.