Building a Multi-Container App

Objective: This tutorial uses Docker Compose to demonstrate the automation of CI workflows.

Preparation: Open two terminal windows, one web browser window and navigate to the appropriate lab folder.

Outcome: We will create a Dockerized “Hello world” type Python application and a Bash test script. The Python application will require two containers to run: one for the app itself, and a Redis container for storage that’s required as a dependency for the app.

Data Files: app.py, requirements.txt, Dockerfile, docker-compose.yml, test.sh, Dockerfile.test,

Create the “Hello World” Python Application

In this step we will create a simple Python application as an example of the type of application you can test with this setup.

1. Create a fresh folder for our application by executing:

```
cd ~
mkdir hello_world
cd hello_world
```

2. Edit a new file app.py with nano:

```
nano app.py
```

3. Add the following content:

app.py

```
from flask import Flask
from redis import Redis

app = Flask(__name__)
redis = Redis(host="redis")

@app.route("/")
def hello():
    visits = redis.incr('counter')
    html = "<h3>Hello World!</h3>" \
           "<b>Visits:</b> {visits}" \
           "<br/>"
    return html.format(visits=visits)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80)
```

`app.py` is a web application based on Flask that connects to a Redis data service. The line `visits=redis.incr('counter')` increases the number of visits and persists this value in Redis. Finally, a `Hello World` message with the number of visits is returned in HTML.

4. Our application has two dependencies, `Flask` and `Redis`, which you can see in the first two lines. These dependencies must be defined before we can execute the application. Edit a new file:

```
nano requirements.txt
```

5. Add the contents:

requirements.txt

```
Flask
Redis
```

Dockerize the “Hello World” Application

1. Docker uses a file called Dockerfile to indicate the required steps to build a Docker image for a given application. Edit a new file:

```
nano Dockerfile
```

2. Add the following contents:

Dockerfile

```
FROM python:2.7

WORKDIR /app

ADD requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt

ADD app.py /app/app.py

EXPOSE 80
```

```
CMD ["python", "app.py"]
```

Let’s analyze the meaning of each line:

> *`FROM python:2.7`: indicates that our “Hello World” application image is built from the official python:2.7 Docker image*
> *`WORKDIR /app`: sets the working directory inside of the Docker image to /app*
> *`ADD requirements.txt /app/requirements.txt`: adds the file requirements.txt to our Docker image*
> *`RUN pip install -r requirements.txt`: installs the application pip dependencies*
> *`ADD app.py /app/app.py`: adds our application source code to the Docker image*
> *`EXPOSE 80`: indicates that our application can be reached at port 80 (the standard public web port)*
> *`CMD ["python", "app.py"]`: the command that starts our application*

This Dockerfile file has all the information needed to build the main component of our “Hello World” application.

> ***The Dependency*** *Now we get to the more sophisticated part of the example. Our application requires Redis as an external service. This is the type of dependency that could be difficult to set up in an identical way every time in a traditional Linux environment, but with Docker Compose we can set it up in a repeatable way every time.*

1. Let’s create a `docker-compose.yml` file to start using Docker Compose. Edit a new file:

```
nano docker-compose.yml
```

2. Add the following contents:

docker-compose.yml

```
web:
  build: .
  dockerfile: Dockerfile
  links:
    - redis
  ports:
    - "80:80"
redis:
  image: redis
```

This Docker Compose file indicates how to spin up the “Hello World” application locally in two Docker containers.

It defines two containers, `web` and `redis`, like this:

- `web` uses the current folder for the `build` context, and builds our Python application from the `Dockerfile` file we just created. This is a local Docker image we made just for our Python application. It defines a link to the `redis` container in order to have access to the `redis` container IP. It also makes port 80 publicly accessible from

the Internet using your Ubuntu server's public IP
- `redis` is executed from a standard public Docker image, named `redis`

Deploy the "Hello World" Application

In this step, we'll deploy the application, and by the end it will be accessible over the Internet. For the purposes of your deployment workflow, you could consider this to be either a dev, staging, or production environment, since you could deploy the application the same way numerous times.

1. The `docker-compose.yml` and `Dockerfile` files allow you to automate the deployment of local environments by executing:

```
docker-compose -f ~/hello_world/docker-compose.yml build
docker-compose -f ~/hello_world/docker-compose.yml up -d
```

The first line builds our local application image from the `Dockerfile` file. The second line runs the `web` and `redis` containers in daemon mode (`-d`), as specified in the `docker-compose.yml` file.

2. Check that the application containers have been created by executing:

```
docker ps
```

This should show two running containers, named `helloworld_web_1` and `helloworld_redis_1`.

3. Let’s check to see that the application is up. We can get the IP of the `helloworld_web_1` container by executing:

```
WEB_APP_IP=$(docker inspect --format '{{ .NetworkSettings.IPA
ddress }}' helloworld_web_1)
```

```
echo $WEB_APP_IP
```

4. Check that the web application is returning the proper message:

```
curl http://${WEB_APP_IP}:80
```

Output:

```
<h3>Hello World!</h3><b>Visits:</b> 1<br/>
```

The number of visits is incremented every time you hit this endpoint. You can also access the “Hello World” application from your browser by visiting the public IP address of your Ubuntu server.

> ***How to Customize for Your Own Application:*** *The key to setting up your own application is to put your app in its own Docker container, and to run each dependency from its own container. Then, you can define the relationships between the containers with Docker Compose, as demonstrated in the example. Docker Compose is covered in greater detail in this Docker Compose article.*

Create the Test Script

Now we'll create a test script for our Python application. This will be a simple script that checks the application's `HTTP` output. The script is an example of the type of test that you might want to run as part of your continuous integration deployment process.

1. Edit a new file:

```
nano test.sh
```

2. Add the following contents:

test.sh

```
sleep 5
if curl web | grep -q '<b>Visits:</b> '; then
  echo "Tests passed!"
  exit 0
else
  echo "Tests failed!"
  exit 1
fi
```

`test.sh` tests for basic web connectivity of our "Hello World" application. It uses `CURL` to retrieve the number of visits and reports on whether the test was passed or not.

Create the Testing Environment

In order to test our application, we need to deploy a testing environment. And, we want to make sure it’s identical to the live application environment we created in Step 5.

1. First, we need to Dockerize our testing script by creating a new Dockerfile file. Edit a new file:

```
nano Dockerfile.test
```

2. Add the following contents:

Dockerfile.test

```
FROM ubuntu:trusty

RUN apt-get update && apt-get install -yq curl && apt-get cle
an

WORKDIR /app

ADD test.sh /app/test.sh

CMD ["bash", "test.sh"]
```

`Dockerfile.test` extends the official `ubuntu:trusty` image to install the curl dependency, adds `tests.sh` to the image filesystem, and indicates the `CMD` command that executes the test script with Bash.

Once our tests are Dockerized, they can be executed in a replicable and

agnostic way.

3. The next step is to link our testing container to our “Hello World” application. Here is where Docker Compose comes to the rescue again. Edit a new file:

```
nano docker-compose.test.yml
```

4. Add the following contents:

```
sut:
  build: .
  dockerfile: Dockerfile.test
  links:
    - web
web:
  build: .
  dockerfile: Dockerfile
  links:
    - redis
redis:
  image: redis
```

The second half of the Docker Compose file deploys the main web application and its redis dependency in the same way as the previous docker-compose.yml file. This is the part of the file that specifies the web and redis containers. The only difference is that the web container no longer exposes port 80, so the application won’t be available over the

public Internet during the tests. So, you can see that we're building the application and its dependencies exactly the same way as they are in the live deployment.

The docker-compose.test.yml file also defines a sut container (named for system under tests) that is responsible for executing our integration tests. The sut container specifies the current directory as our build directory and specifies our Dockerfile.test file. It links to the web container so the application container's IP address is accessible to our test.sh script.

How to Customize for Your Own Application

> *NOTE:* `docker-compose.test.yml` *might include dozens of external services and multiple test containers. Docker will be able to run all these dependencies on a single host because every container shares the underlying OS.*

If you have more tests to run on your application, you can create additional Dockerfiles for them, similar to the `Dockerfile.test` file shown above.

Then, you can add additional containers below the sut container in the `docker-compose.test.yml` file, referencing the additional Dockerfiles.

Test the "Hello World" Application

1. Finally, extending the Docker ideas from local environments to testing environments, we have an automated way of testing our application using Docker by executing:

```
docker-compose -f ~/hello_world/docker-compose.test.yml -p ci
 build
```

This command builds the local images needed by `docker-compose.test.yml`.

> *NOTE: We are using `-f` to point to `docker-compose.test.yml` and `-p` to indicate a specific project name.*

2. Now spin up your fresh testing environment by executing:

```
docker-compose -f ~/hello_world/docker-compose.test.yml -p ci
 up -d
```

3. Check the output of the sut container by executing:

```
docker logs -f ci_sut_1
```

Output:

```
  % Total    % Received % Xferd  Average Speed   Time    Time
     Time  Current
                                 Dload  Upload   Total   Spen
t    Left  Speed
100    42  100    42    0     0   3902      0 --:--:-- --:--:
-- --:--:--  4200
Tests passed!
```

4. And finally, check the exit code of the sut container to verify if your tests have passed:

```
docker wait ci_sut_1
```

Output:

```
0
```

After the execution of this command, the value of $? will be 0 if the tests passed. Otherwise, our application tests failed.

> *NOTE: Other CI tools can clone our code repository and execute these few commands to verify if tests are passing with the latest bits of your application without worrying about runtime dependencies or external service configurations.*

That's it! We've successfully run our test in a freshly built environment identical to our production environment.

Conclusion

Thanks to Docker and Docker Compose, we have been able to automate how to build an application (Dockerfile), how to deploy a local environment (`docker-compose.yml`), how to build a testing image (`Dockerfile.test`), and how to execute (integration) tests (`docker-compose.test.yml`) for any application.

> *In particular, the advantages of using the `docker-compose.test.yml` file for testing are that the testing process is:*

- **Automatable:** the way a tool executes the docker-compose.test.yml is independent of the application under test
- **Light-weight:** hundreds of external services can be deployed on a single host, simulating complex (integration) test environments
- **Agnostic:** avoid CI provider lock-in, and your tests can run in any infrastructure and on any OS which supports Docker
- **Immutable:** tests passing on your local machine will pass in your CI tool