System Testing with pytest and docker-py

By Christie Wilson (@bobcatwilson) & Michael Tom-Wing (@mtomwing)
github.com/keeppythonweird/pytest-dockerpy

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
Hello!

● Christie Wilson
  ○ Senior Developer @ Demonware
  ○ Team Lead: Test Tools

● Michael Tom-Wing
  ○ Software Engineer in Test @ Demonware
  ○ Focus on automation and quality

● We’re from Canada!
demonware

- Video Game Industry
- Online services for games
- Come see us in the vendor area!
COMMANDER McFLUFFLES AND THE QUEST FOR QUALITY

Once upon a system test
Testing at Demonware 2011-2016
What is testing?
Best Practices for System Testing
PYTEST FIXTURES AND DOCKER-PY
TAKEAWAYS FOR DEV AND OPS

@bobcatwilson
@mtomwing
goo.gl/JkzmYJ
Once upon a System test
Testing at Demonware 2011-2016
Demonware - 2011
Démonware - RIP other testing methods

Testing in prod

Local manual testing

Tools to install rpms
Demonware - 2016

- Test Tools team
- Variety of tests
  - Unit tests
  - Integration tests
  - System tests
What is testing?
What is testing? Why do we test?
Why do we test?

- To increase confidence in our software
- Avoid regressions
- Document behaviour
Why don’t we test?

- NOT to find all the bugs in our software
- Will never find all bugs
Testing + Software Quality

- Testing does not increase the quality of our software
- By the time our tests run our software is already buggy
  - Introduce quality through requirements + design!
- But...
Testing + Software Quality

- Tests provide metrics to let us reason about quality
  - E.g. coverage, timing, # of logs
- Untested software is always VIEWED as lower quality
- Less information about the quality
Example
How to test it

- Unit tests
- Integration tests
- System tests
Unit Tests

- Unit tests: ~100% coverage
- Integration tests
- System tests
Integration Tests

- Unit tests: ~100% coverage
- Integration tests: service <-> DB
- System tests
System Tests

Test the entire system

Pros

- Most valuable
- Most likely to find bugs

Cons

- Slowest
- Hardest to maintain
**System Tests**

- Unit tests: ~100% coverage
- Integration tests: service <-> DB
- System tests: happy path, few simple failures
Best Practices for System Testing
BEST PRACTICES

- Use fresh state between tests
- Will help avoid dependencies between tests
  - e.g. test ordering shouldn’t affect their outcomes
- Docker helps make this very easy!
BEST PRACTICES

● Ensure tests can run on build servers and locally
  ○ Ease the burden for writing and running tests
● Restrict the test environments you’ll support
  ○ e.g. Linux, OSX, toaster
BEST PRACTICES

- Tests should clean up after themselves
- Fail fast
- Fail informatively
Glue Code

def glue_code():
    with open('file') as f:
        result = my_module.do_something(f.readlines())
    other_result = other_module.do_things(result)
    do_something_amazing(other_result)
    return good_things(os.getcwd())
PYTEST FIXTURES AND DOCKER-PY
PyTest

- Python testing library
- v.s. unittest = less boilerplate
- More batteries included
  - e.g. fixtures, plugins
PyTest Fixtures

- Provide setup and teardown for tests
- Pytest will ensure that the setup and teardown always happen
  - And in that order!
- System tests generally set up a lot of things!
- Very slick!
PyTest Fixtures

SETUP

TEARDOWN

YOUR TEST
PyTest Fixtures

1. SETUP -> TEST #1 -> TEARDOWN
2. SETUP -> TEST #2 -> TEARDOWN
3. SETUP -> TEST #3 -> TEARDOWN

@bobcatwilson
@mtomwing
goo.gl/JkzmYJ
PyTest Fixtures - Scope

SETUP

TEST #1

TEST #2

TEST #3

TEARDOWN
PyTest Fixtures - Scope

TEST #1

 SETUP

 TEST #1

 TEARDOWN

 SETUP

 TEST #2

 TEARDOWN

 SETUP

 TEARDOWN

Docker

docker docker docker docker docker docker docker docker

docker docker docker docker docker docker docker docker

docker docker docker docker docker docker docker docker

docker docker docker docker docker docker docker docker

docker docker docker docker docker docker docker docker

docker docker docker docker docker docker docker docker

goo.gl/JkzmYJ
Docker

- Challenging service setup = docker
- Setup + teardown: bash scripts
Docker-py

- Python library for using docker
- Interface is 1:1 with the REST interface
  - Can be a bit clunky
**DOCKER-py**

1. Create client
2. Pull image
3. Create container
4. Start container
5. Remove container
CREATE CLIENT

```python
import docker
docker_client = docker.Client('unix:///var/run/docker.sock', version='auto')
```
docker_client.pull('percona:5.6')
response = self._docker_client.pull('busybox:latest')

lines = [line for line in response.split('
') if line]
pull_result = json.loads(lines[-1])

if 'error' in pull_result:
    raise Exception(pull_result['error'])

Thanks Steven Erenst!
Create container

```python
container = docker_client.create_container(
    image='busybox:latest',
    labels=['docker-test-log'])
```
Start container

docker_client.start(container=container['Id'])
Kill and remove container

docker_client.remove_container(
    container=container["Id"],
    force=True,
)

goo.gl/JkzmYj
Pytest Fixtures + Docker-py

CREATE CONTAINER → TEST #1 → DELETE CONTAINER

CREATE CONTAINER → TEST #2 → DELETE CONTAINER

CREATE CONTAINER → TEST #3 → DELETE CONTAINER

goo.gl/JkzmYJ
**Pytest Fixtures + Docker-py**

RUN httpd:2.4

RUN percona:5.6

RUN redis

RUN percona:5.6

RUN redis

RUN percona:5.6

RUN redis

RUN percona:5.6

RUN redis

TEST #1

TEST #2

TEST #3

DELETE percona:5.6

DELETE redis

DELETE percona:5.6

DELETE redis

DELETE percona:5.6

DELETE redis

DELETE percona:5.6

DELETE redis

DELETE httpd:2.4
import docker
import pytest
@pytest.yield_fixture
def example_container():
    docker_client = docker.Client('unix://var/run/docker.sock', version='auto')
    docker_client.pull(IMAGE)
    container = docker_client.create_container(
        image=IMAGE,
        detach=True,
        labels=[labels.CONTAINERS_FOR_TESTING_LABEL]
    )
    docker_client.start(container=container['Id'])
    container_info = docker_client.inspect_container(container.get('Id'))

    yield container_info['NetworkSettings']['IPAddress']

    docker_client.remove_container(
        container=container['Id'],
        force=True
    )
Pytest Hooks

- pytest magic!

```python
def pytest_runtest_logreport(report):
```
def pytest_runtest_logreport(report):
    if report.failed:
        docker_client = _docker_client()
        test_containers = docker_client.containers(
            all=True,
            filters={'label': labels.CONTAINERS_FOR_TESTING_LABEL})
        for container in test_containers:
            log_lines = [
                "docker inspect {!r}:".format(container['Id']),
                pprint.pformat(docker_client.inspect_container(container['Id'])),
                "docker logs {!r}:".format(container['Id']),
                docker_client.logs(container['Id']),
            ]
        report.longrepr.addsection('docker logs', os.linesep.join(log_lines))
Pytest Fixtures - Docker Logs
What about docker-compose?

● Works well when the deployment is static between tests
  ○ Not as well suited when deployment is different for each test
● Integrates with pytest fixtures!
  ○ e.g. Use a fixture to run docker-compose up
● docker-py can help
What about docker-compose?

```python
@pytest.fixture
def docker_client():
    return docker.Client('unix://var/run/docker.sock', version='auto')

@ pytest.fixture
def my_cluster(request):
    def fin():
        subprocess.check_output(
            shlex.split('docker-compose down'))

        request.addfinalizer(fin)
    subprocess.check_output(
        shlex.split('docker-compose up -d'))

@ pytest.fixture
def some_container_ip(my_cluster, docker_client):
    output = docker_client.inspect_container(SOME_CONTAINER)
    return output['NetworkSettings']['Networks'][DOCKER_COMPOSE_NETWORK_NAME]['IPAddress']
```
Gotchas

- Wait for the service to start (backoff)
- Maximize container startup speed!
Takeaways for Dev and Ops
What do I do with this?

- Developers
- Ops
Developers

- When to write tests and when not to
- Try some TDD: start with a system test
Developers - Introduce System Tests

- Add one system test to each piece of software you own
- Make sure tests can run:
  - With as little setup as possible
  - As quickly as possible
- Add the system tests to your CI
Developers - Already Have System Tests

- Do you need all the tests you have?
  - Can you replace with integration or unit tests?
  - How many retest functionality?
  - Can some of the tests be removed?
- Can the tests be faster?
Ops

- One off scripts: *don’t need system tests*
- Scripts and automation that will be used in the future *need system tests*
  - What is one bare minimum system test you can add?
- Use automation to regularly run your tests
  - e.g. Travis CI
Ops - Tests for tools

- Tools that use services you can run:
  - Use something like pytest + docker-py
- Tools that use services you can’t run (e.g. AW$):
  - Can you run a short system test, e.g. once per week?
  - Is it going to cost you a lot?
  - Make sure the tests clean up after themselves
Overview

github.com/keeppythonweird/pytest-dockerpy
@bobcatwilson
@mtomwing