DEVCONF.cz

Dealing with false failures in automated tests

Marian Ganišin

Software Quality Assurance Engineer in Red Hat

Introduction - What and How

Software under test - 3scale

3scale is API Management System - a service to publish API, control access, collect and analyze usage statistics

- It has REST API to control the system
- It has UI (Web interface)
- It has gateway/proxy that provides access to published API

Testing Framework - pytest

The pytest is testing framework written in python. Instead of setup/cleanup it provides a mechanism of fixture(s) definition.

We write tests in python..

Testing Conditions

Integration/System/E2E Testing

5+ different execution environments

Various CI systems

Available resources are not same

Deployment methods are changing

False positives/alarms are unavoidable

The problem with false alarms

- Automatic acceptance pipelines in CI systems blocked
- Overlooking the failures
- Masking the failures
- Wasted time with re-runs
- Nightly testing - lot's of results to analyze

Accept the reality

Embrace the pain

97 % success rate is fine for 10 or 50 tests but pain for hundreds or thousands of tests.

Instability remains in large amount of tests 5 - 15 %.

No silver bullet - except of infinite budget.

Neverending work - one test fixed, another set of fresh potentially flaky tests added meanwhile.

There will be some threshold above which the investment into a work on more stability won't deliver noticeable benefit.

Why the tests fail?

The tests are faulty

Missing or incomplete cleanup

Assumed preconditions

Dependencies between tests

Test Complexity (e.g. UI Tests)

The tests are faulty

Missing or incomplete cleanup

Assumed preconditions

Dependencies between tests

Test Complexity (e.g. UI Tests)

Keep tests simple

Strict test isolation

Setup & Cleanup for the test

Never assume test execution order

Containerization

Test Blockers

Testing may stress corner cases that never affect real-world workload - this is a test blocker.

It can be hard to identify and reproduce.

The test blocker can be present in third-party dependency.

Test Blockers

Testing may stress corner cases that never affect real-world workload - this is a test blocker.

It can be hard to identify and reproduce.

The test blocker can be present in third-party dependency.

Developers should address test blocker with appropriate priority.

Change the test to mitigate the impact.

Infrastructure

No reliable test results in unreliable environment

This is a must!

A chain is **weaker** than the weakest link!

Two parts, one with 99.9 % reliability, the other one 95 %

Overall reliability of the system: 0.999 * 0.95 = 0.949 -> 94.9 %

Replace weak parts. Reduce the complexity. Remove unnecessary elements. Reduce the load generated by tests.

All that effort so far - false alarms persist

Different sources of non-deterministic behavior

- Network latency and connection drops
- Waiting on openshift operations
- Waiting on the application to process the requests
- Waiting on UI

Something we tried and failed

Increase socket timeout

- An attempt to mitigate network latency issue.
- It didn't help, just made the execution of failed test longer.

Use (short) sleep

- Trial and error method to find correct value
- Limited impact - didn't prevent failure under different conditions or different environment
- Made test execution longer

Something that works to certain extent

Quarantine - known technique to isolate unstable tests, run them separately, systematically work on their fixes.

- Split from the rest works well.
- It's quite "easy" to accept it as new standard (these are those red tests).
- Still potentially hides real issues

Pipelining - splitting the test execution into groups

- Primarily helps with re-runs (smaller amount to execute again)
- Potentially smaller numbers of failures to review together
- Hiding problem remains, total number of failed tests remains unchanged.

Last mile to the success rate > 99.5%

Retry

pytest builtin plugin to run only failures

There is an option `pytest --last-failed` that will execute only failed tests from recent run.

- Powerful feature, available out of the box, noticeably effective.
- It can be ran in sequence couple of times.
- Can not cover all the glitches.
- Track of full results bit of inconvenient (has to be archived manually, merging issue)

flaky - pytest plugin to rerun marked tests

Plugin for pytest and nose (https://github.com/box/flaky)

- It does the rerun during test execution, no need to execute it again
- Flaky tests already marked as part of quarantine process
- No problems to get complete test results
- Limited effect in our tests (scope of the fixtures)

```
@pytest.mark.flaky
def test_service_discovery(api_client):
```

Retry in used libraries

Python requests library (https://requests.readthedocs.io/en/latest/) allows retry on failed HTTP requests

- Works pretty well for our case (testing many many HTTP requests)
- Easy to setup default retry for all the tests (disable it in the test setup if necessary)
- Works only for requests

backoff - a module made for retrying

backoff (https://github.com/litl/backoff) is python package that provides function decorators to run decorated function again under certain condition.

- Very flexible, universal solution.
- Provides retry with progressive delay.

```
@backoff.on_exception(backoff.fibo, AssertionError, max_tries=8, jitter=None)
def test_unique_invoice(create_invoice, threescale):


@backoff.on_predicate(backoff.constant, lambda x: x.status_code != 429, max_tries=16)
def make_requests(api_client):
```

Summary

- Test Design
- Test Blockers
- Infrastructure Reliability
- Quarantine
- Splitting the test execution
- Retry: pytest --last-failed
- Retry: flaky
- Retry: requests
- Retry: backoff

From 90 - 95 % success rate (up to approx. 100 false failures) a year ago to 99.7 % success rate (2 - 4 false failures) nowadays.

Thank you for your attention