FLEX - Testing

Simon Härer

Created: 2025-12-02 Tue 13:19

Tests

Everything we did targets an independent deployment between teams. In this section we talk about how we can exploit this new-gained independence for testing optimally.

Test Methods

There are different test classes, which here are ordered from fine-grained to coarse:

1. unit tests
2. integration tests
3. UI tests
4. manual tests

Unit Tests

Unit tests are testing preferably small units of the system, mostly methods and functions. Units for unit tests should be **pure**, that is, free of side effects. We often achieve that by replacing dependencies with stubs or by mocking them:

- **Mocks** replace single methods (functions) within the code with another function. A test can e.g. override a function for a specific customer to return a fixed account balance.
- **Stubs** replace a whole dependency or service. This means that for each part of the interface, there must be a replacement that returns data suitable for the test.

While mocks are more comfortable to write, stubs are a more holistic approach. Mocking single methods tends to return inconsistent data among multiple calls. Furthermore, mocks are leaking implementation in the test cases.

Integration Tests

Integration tests are testing the interaction between a system′s components. While we stubbed or mocked dependencies and services in unit tests, we now want them to be explicitly executed. The integration test system is meant to be as close to the production system as possible.

Integration tests test the boundaries of the whole system. In this tests we want to see the behavior of the system as a whole and, thus, have a blackbox view of the system.

UI Tests

UI tests test the application through the user interface. These tests are valuable for *graphical* user interfaces.

While they are hard to implement and maintain, they provide unique test coverage. Beside business functionality, they often implicitly test if a UI can be displayed, if elements can be rendered or if malfunctioning visual logic blocks the user flow. Most of the time UI tests use an integrated testing environment.

UI Tests

Ultimately, UI tests are the non plus ultra of testing, since most requirements are articulated from a user's perspective. With UI tests we can declaretively implement these requirements.

Sadly, existing technologies still make it very hard to create and maintain these tests in a proper amount of time. Often user acceptance tests are covered by UI tests.

@ctive group

Manual Tests

Manual tests are often called *click tests*. A person follows a test protocol to manually test the application. This is often necessary for UI applications where UI tests are not applicable, due to technology or economic reasons.

Manual tests should be explorative and fully protocolled. They must be planned and executed carefully. Opposed to the other tests, which are executed automatically, steps can be missed or protocolled wrongly.

Test Distribution

The distribution of test classes should be as follows, to get a high payoff from testing:

1. Unit tests should dominate. They are fast to execute and easy to implement. They should cover most functionality including all edge cases. A lot of logical errors can be found using unit tests
2. Integration tests are next. They cover complex processes and even chains of multiple requests. They are more difficult to implement and because of the included services and dependencies often slower than unit tests

Test Distribution

3. UI tests are more holistic than integration tests. They test what a user is actually experiencing. They are hard to implement and it can be exhausting to maintain them. They also tend to be slow.
4. Manual tests are time consuming and error prone. They cannot be part of the automatic test pipeline and should only be done if there's no way to cover the tested functionality differently.

Automated Test Pipelines

We want to avoid manual testing since we cannot automate it. We ultimately want to have a high rate of deployments. To do so, we create a test pipeline that gets automatically executed upon commits. This pipeline usually executes tests in the order we just described them:

1. Unit tests
2. Integration tests
3. UI tests

We want to avoid the execution of expensive UI tests before we evaluated the fast and cheap unit tests. In case of an early error, we can react quickly and fix the failed tests, without losing to much time testing.

Why Code Reviews

- when everything is automated mistakes can go into production without noticing quickly
- reviews include technical as well as domain specific errors
- the reviewer also checks if the code is tested properly
- four-eyes principle: Two persons have more knowledge and may complement each other
- when productivity is high and should stay high in the future, it is important to maintain high quality code

Code Review Process

- the development is done in (feature-) branches
- when a feature or ticket is finished, it is marked as "ready for review"
- another developer reviews the commits and gives feedback
- the ticket owner has to apply the feedback and fix the code
- this process is repeated until the reviewer is happy. Then the code can be merged into master and the ticket is marked as "done"

Code reviews not only result in higher code quality, but team members learn from each other very efficiently.

@ctive group

Testing the Entire System

Until now we assumed testing our microservice standalone, even in UI tests and integration tests. We could also test the whole microservice architecture in tests, to check the interplay of the services.

To do so, we need another test pipeline that executes integration and UI tests upon commits of **each** microservice.

Testing the Entire System - Drawbacks

- **Very expensive:** We divided our system into microservices to get fast tests.
- **Dependencies**: What if the test fails? Is deployment for all microservices stopped now? This is not what we wanted in the first place.
- **Pipeline stalling**: If a microservice is in the integration test stage and there are changes in another microservice, the second needs to wait until the first has finished to keep integrity in tests. Testing can get awfully slow.

If the entire system needs to be tested, these tests must be very lean and fast, to not lose all gained advantages.

Testing with Contracts

A microservice offers an interface for other services. This can be seen as a contract. By making these contracts explicit, we can test them.

Consumer-Driven Contract

The contract about the interface of a microservice is formulated from consumer perspective, thus it is called a **consumer-driven contract**. Elemental specifications are:

- the data formats of the exchanged information
- the interface definition, that defines available operations
- meta information for operations, e.g. about authorization
- sequences of operations, e.g. create session -> authenticate -> request -> close session
- non-functional aspects as latency or throughput

I. Robinson: Consumer-Driven Contracts: A Service Evolution Pattern, 2006.

Consumer-Driven Contract

A consumer-driven contract contains two types of sub-contracts:

- Provider Contract
- Consumer Contract

Provider Contract

- This contract describes what the service actually offers
- There is just one provider contract per service

Consumer Contract

- There is one consumer-driven contract per consumer
- Each contract describes what part the consumer is actually *using* from the provider contract
- This contract can change, if the consumer request additional resources or removes certain calls

Consumer-Driven Contract

- A consumer-driven contract contains exactly one provider contract and one or more consumer contracts
- There is only one consumer-driven contract per service
- The contract makes the actual usage of the interface transparent

Consumer-Driven Contract Tests

Tests can be implemented based on these contracts:

- Based on the consumer-requested functionality, the provider can implement specific tests
- Usually the consumer herself formulates tests for the providing service
- Consumer-driven contract tests are a substitute for integration testing the entire system
- The tradeoff is again a certain degree of dependencies between teams

@ctive group

Consumer-Driven Contract Tests - Tooling

- Tests can be generic tests in the used language
- Tooling to integrate consumer-driven contracts in the code exists:
    - Pacto (Ruby): Generates contracts from HTTP/REST interfaces and automates some tests based on these
    - Pact (Ruby): Allows recording interaction with a stub and generates a contract based on the interaction. Also available for jvm based language as pact-jvm
    - Spring Cloud Contract (Java): Generates an interface based on a contract, also generates a stub for testing

Testing Legacy Applications

- legacy application are often slowly testable deployment monoliths
- adding them to the continuous integration and test pipeline would slow down everything considerably
- since legacy applications are often deployed in fix but rather long intervals, we can test integration automatically at these limited releases

Testing Technical Parts of the Entire System

- technical requirements for the system can be a specification of the behavior in case of a crash or performance metrics
- these requirements are mostly well defined, quantifiable and easy to test
- thus, testing them automatically should be easy
- The tradeoff is again a certain degree of dependencies between teams, since changes in interfaces or technologies must be adapted

Run-Time Tests: Chaos Monkey

Another class of tests take place in the production system. One example is **Chaos Monkey** by Netflix. It randomly terminates virtual machines in the production system. The system should be able to handle it by redundancy and very fast restart of unhealthy services.

We will later see how monitoring is a key factor in keeping a system healthy and somehow a class of tests on its own.

Chaos Monkey, Netflix

Exercise: Testing Erlbank

How would the participants design the testing of Erlbank?

Infrastructure as Code

Everything ops-related, should be expressed in code. That makes

- processes repeatable
- processes automated

Tests through Infrastructure as Code in Gitlab

```
test:
  image: docker:stable
  services:
    - docker:dind
    before_script:
      - docker info
      script:
        - docker build -t test-image .
        - docker run test-image test
```

Discussion

- What is the alternative? Configuring by hand?
- What would be the drawbacks?
- Where else is Infrastructure as Code a must? Deployment!