# Tests en DevOps

Ryan Lahfa

# Tests automatiques

# Distinction entre un test unitaire et un test d'intégration

Un test unitaire s'applique bien souvent à une fonction, une classe complètement isolé de ses « services » [1] au sens large. Il y a deux façons d'en écrire:

- Soit le code est dit « couplé » au service et il faut recourir à du monkey patching et des mocks.

[1] Une base de donnée, un micro-service, par exemple.

# Distinction entre un test unitaire et un test d'intégration

Un test unitaire s'applique bien souvent à une fonction, une classe complètement isolé de ses « services » [1] au sens large. Il y a deux façons d'en écrire:

- Soit le code est dit « couplé » au service et il faut recourir à du monkey patching et des mocks.
- Soit le code le permet (une bonne architecture & écriture du code suffit en général) et on peut directement tester sans galère.

[1] Une base de donnée, un micro-service, par exemple.

# Tests d'intégration

Par opposition, un test d'intégration teste un système entier, couplé à ses éventuels services.

Dans ce cas, ces tests sont plus coûteux, plus longs, mais aussi plus intéressants. Il faut à la fois en même temps ramener l'environnement de test dans un environnement proche de celui de la production puis lancer des tests, nettoyer l'environnement et recommencer.

# Tests d'intégration

Par opposition, un test d'intégration teste un système entier, couplé à ses éventuels services.

Dans ce cas, ces tests sont plus coûteux, plus longs, mais aussi plus intéressants. Il faut à la fois en même temps ramener l'environnement de test dans un environnement proche de celui de la production puis lancer des tests, nettoyer l'environnement et recommencer.

Ces tests font émerger les soucis d'intégration, e.g. une implémentation "en RAM" d'une base de données ne se comporte pas comme une implémentation "sur le disque", certains paramétrages se comportement bizarrement et peuvent créer de la perte de données et induire des comportements inattendus en production, qu'on ne voit pas en test.

# Méthodologie BDD/TDD

L'idée est de coder vos fonctionnalités en implémentant d'abord des prototypes de votre architecture sans leur implémentation, puis d'écrire une couverture de test naturelle, et de remplir les blancs, tant que les tests ne passent pas, vous implémentez au fur et à mesure.

# Méthodologie BDD/TDD

L'idée est de coder vos fonctionnalités en implémentant d'abord des prototypes de votre architecture sans leur implémentation, puis d'écrire une couverture de test naturelle, et de remplir les blancs, tant que les tests ne passent pas, vous implémentez au fur et à mesure.

Cette méthodologie est intéressante, puisque dans votre processus de conception d'architecture, la partie « concevoir » est intrinsèquement lié à « tester », vous dites: « Je veux que mon système se comporte ainsi dans tel contraintes », dès lors, vous proclamez et pensez à un test dans votre tête, alors au lieu de l'imaginer, vous pouvez l'écrire d'abord, puis résoudre les problèmes.

# Couverture de test

Une fois que vous testez votre code, vous voudriez savoir si toutes les lignes que vous avez dans votre code sont au moins atteintes, que vous n'oubliez pas des cas subtiles qu'il serait intéressant de tester.

# Couverture de test

Une fois que vous testez votre code, vous voudriez savoir si toutes les lignes que vous avez dans votre code sont au moins atteintes, que vous n'oubliez pas des cas subtiles qu'il serait intéressant de tester.

Les couvertures de test remplissent ce rôle, elles vont tracer l'exécution des chemins de code d'un test et reporter ça dans un format utilisable (XML, JUnit, etc.), ensuite vous pourrez les visualiser pour comprendre ce qui se passe.

Vous verrez donc plusieurs projets OSS qui proclament des couvertures à 98, 99, 100 %, i.e. toutes les lignes sont atteintes par un test au moins. Cependant.

# Attention au piège

Avoir 100 % de couverture de test, ça ne veut pas dire que le code est bien testé et qu'il aura zéro bug voire encore moins aucun bug.

# Attention au piège

Avoir 100 % de couverture de test, ça ne veut pas dire que le code est bien testé et qu'il aura zéro bug voire encore moins aucun bug.

C'est très naïf, on peut tester dans son entireté un code sans jamais **bien le tester**, faire émerger les cas ou les entrées qui amènent des bugs, ou des états indéterminées, surtout dans des tests unitaires où l'état du système est remis à zéro.

# Attention au piège

Avoir 100 % de couverture de test, ça ne veut pas dire que le code est bien testé et qu'il aura zéro bug voire encore moins aucun bug.

C'est très naïf, on peut tester dans son entireté un code sans jamais **bien le tester**, faire émerger les cas ou les entrées qui amènent des bugs, ou des états indéterminées, surtout dans des tests unitaires où l'état du système est remis à zéro.

Les tests d'intégration rendent ça moins flagrant, mais les tests ne remplaceront jamais un vrai monitoring et des leçons tirés d'années de production, puisqu'elle n'accumulent pas le même genre d'état et de requêtes.

# Tester, tester quoi ?

Alors, la question qui peut se poser, c'est tester quoi ou comment **bien tester** ? Il y a jamais de bonne réponse ou de vraie réponse.

# Tester, tester quoi ?

Alors, la question qui peut se poser, c'est tester quoi ou comment **bien tester** ? Il y a jamais de bonne réponse ou de vraie réponse.

On testera souvent ce qui est susceptible de nous mordre ou ce qui nous a déjà mordu dans le passé (empêcher les regressions), on peut ajouter un peu de robustesse à nos tests en tapant dans les méthodes formelles avec des outils comme Hypothesis (Python) ou Quickcheck (Haskell).

# Tests exotiques

Bien sûr, on peut encore aller plus loin, mais c'est pas le but de ce cours, on peut avoir des tests:

- D'orchestration de déploiements dans des VMs, par exemple. Tester qu'on puisse reproduire l'infrastructure sur plusieurs clouds, ou des choses comme ça.

# Tests exotiques

Bien sûr, on peut encore aller plus loin, mais c'est pas le but de ce cours, on peut avoir des tests:

- D'orchestration de déploiements dans des VMs, par exemple. Tester qu'on puisse reproduire l'infrastructure sur plusieurs clouds, ou des choses comme ça.
- Des tests visuels et cross-browser testing[2]: https://storybook.js.org/docs/testing/automated-visual-testing/

[2] Notamment, avec Puppeteer ou des implémentations orienté DOM virtuel

# Certification ou vérification

Enfin, ce ne sont pas des tests, mais on peut aller plus loin et **certifier** son implémentation ou la vérifier:

- Cryptographie: https://verifpal.com/

Ces choses là assurent des niveaux de robustesse très proche de ce qu'on sait faire mieux et probablement ce qu'on saura faire de mieux pour les 40 prochaines années, certaines entreprises d'ailleurs en font un cœur de métier:
https://www.provenrun.com/products/provencore/

# Certification ou vérification

Enfin, ce ne sont pas des tests, mais on peut aller plus loin et **certifier** son implémentation ou la vérifier:

- Cryptographie: https://verifpal.com/
- Système d'exploitation (ou micro-noyaux): https://sel4.systems/

Ces choses là assurent des niveaux de robustesse très proche de ce qu'on sait faire mieux et probablement ce qu'on saura faire de mieux pour les 40 prochaines années, certaines entreprises d'ailleurs en font un cœur de métier: https://www.provenrun.com/products/provencore/