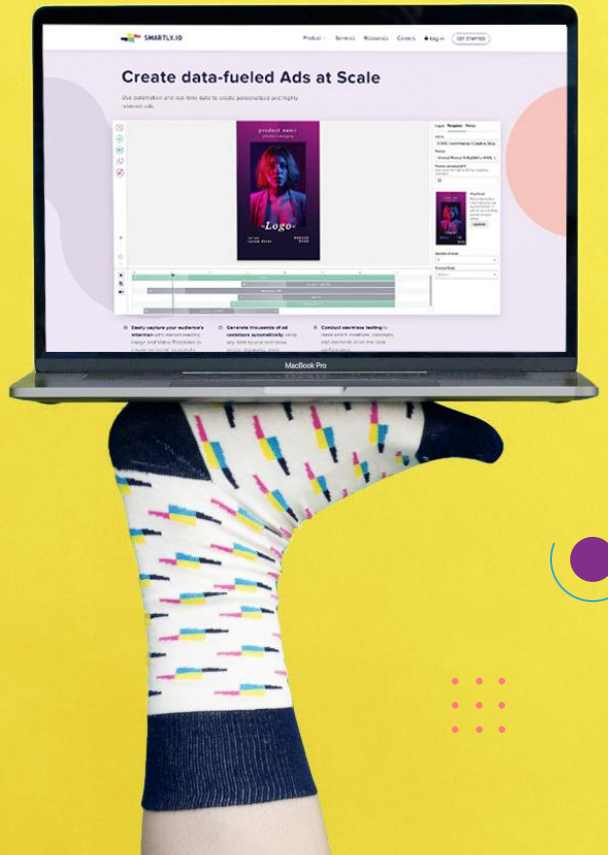
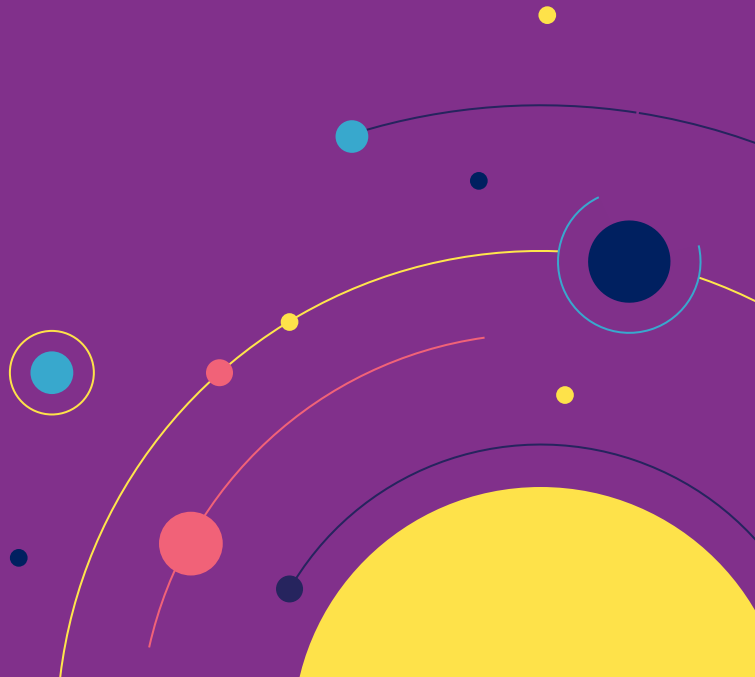
SMARTLY.IO

Software Testing Why, How and What



Agenda of today

1. Why test?
2. How to test?
3. What to test?
4. Testing as a part of the CI/CD pipeline
5. Q&A



Juha Viljanen

- Full Stack Developer @ Smartly.io since 2017
- MSc (Aalto University) 2007-2015

linkedin.com/in/juhaviljanen

What does Smartly.io do?

World's leading brands trust Smartly.io
with $2 billion annual ad spend
700 Customers and 17 Offices

Marketing Partners

Instagram PARTNERS

marketing partners

What does my team do?

Senators:

We own the Facebook ads part of the Smartly tool

Learn more: https://www.smartly.io/blog/meet-our-dev-teams-senators

Why test?

We have an existing product that has no automated tests.

It works and is used by customers.

Why should I spend time to write tests for the existing code?

A customer (of your consulting business) says you're not allowed to write automated tests, because we need to move fast.

Your response?

Why test?

Because it’s the only way to know that the product works as intended.

Or more accurately

Testing is often a cost effective way to have confidence that things we care about work.

How to test?

Manual testing

Pros

- Good for finding bugs in existing code
  - Exploratory testing

Cons

- Bad for ensuring software still works as intended
  - Labor intensive
  - Every time code is changed you'd have to test *everything* again

Automated testing

Pros

- Good for ensuring software still works as defined in tests
  - Cheap to execute
  - Enable to move fast and with confidence

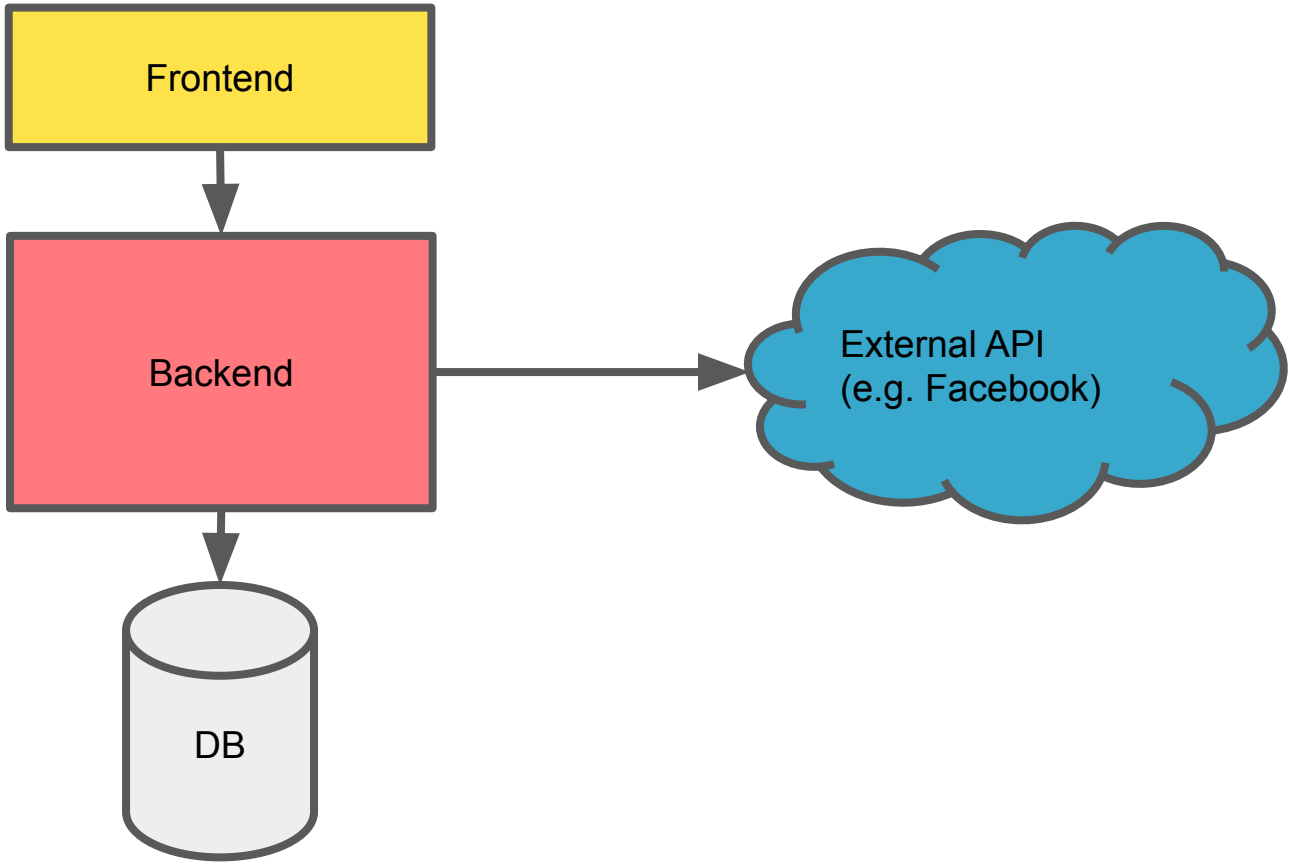
Cons

- Typically bad for finding bugs in code

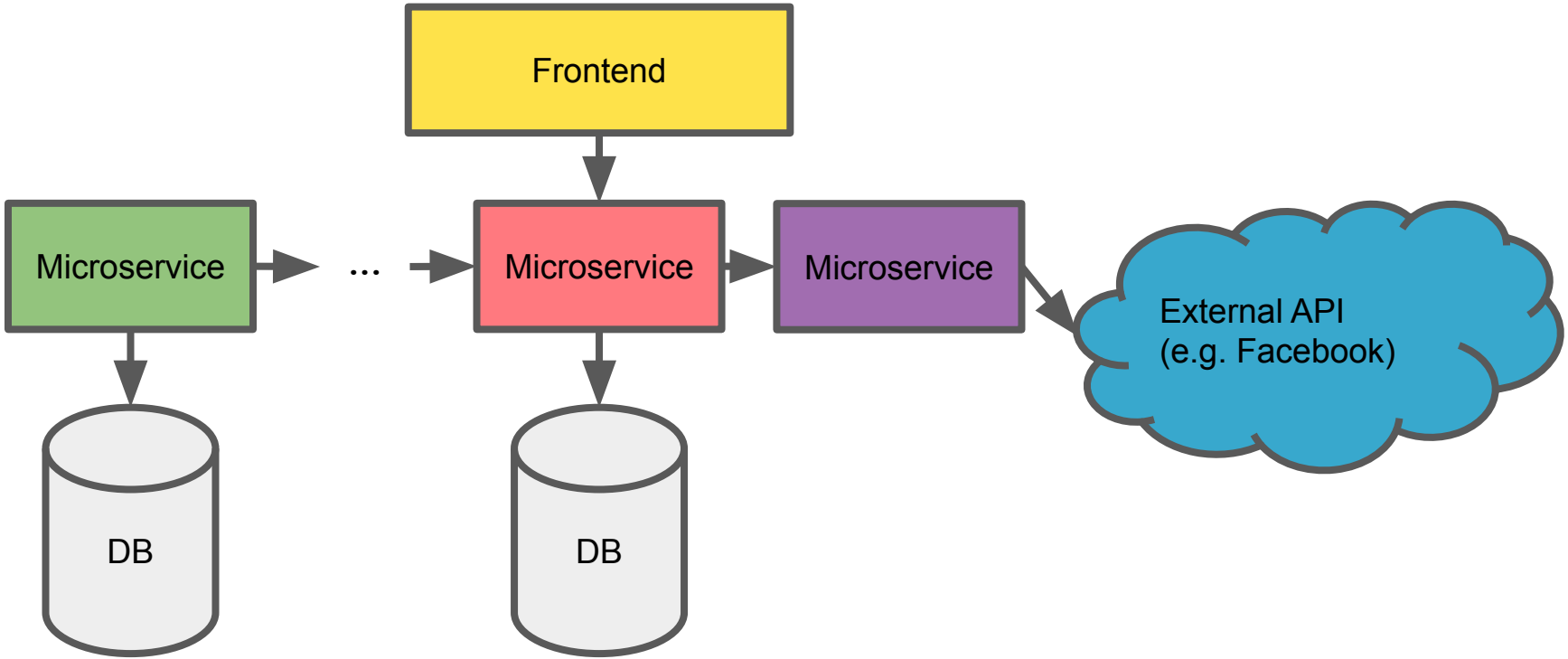
Different types of automated tests

We can test on different abstraction levels

Monolithic software architecture

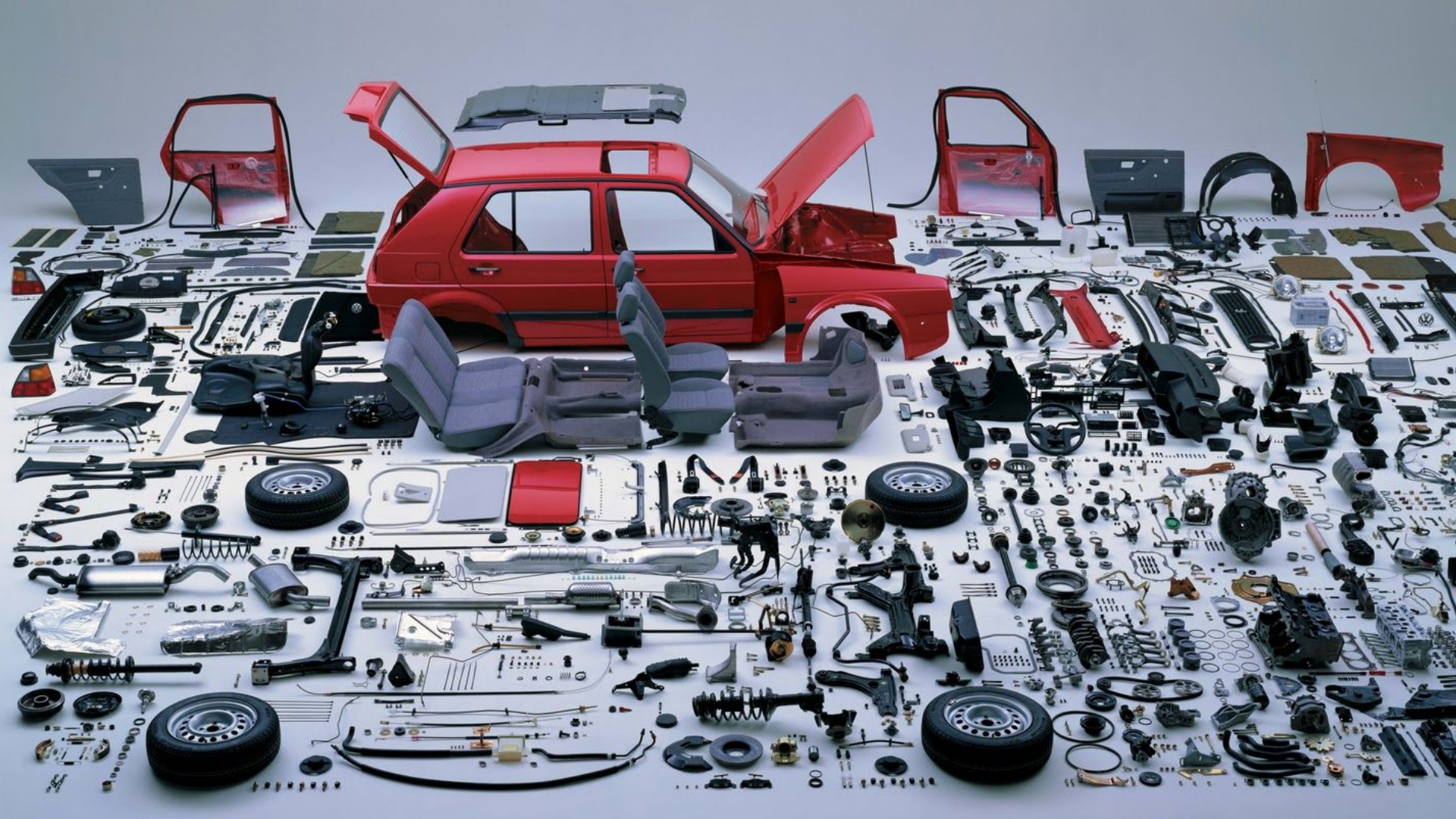


Microservice architecture



Testing on different abstraction levels

- Higher abstraction level tests gives better confidence that system *actually* works
- Lower abstraction level tests are faster to run, typically faster to write and more helpful in pinpointing where the problem lies if a test fails



Most common test types

- Unit tests, Integration tests, API tests, End-to-End tests
  - No universal definitions for what these exactly mean
- Amazing blog post about this 🦄

  https://martinfowler.com/articles/practical-test-pyramid.html

Unit tests

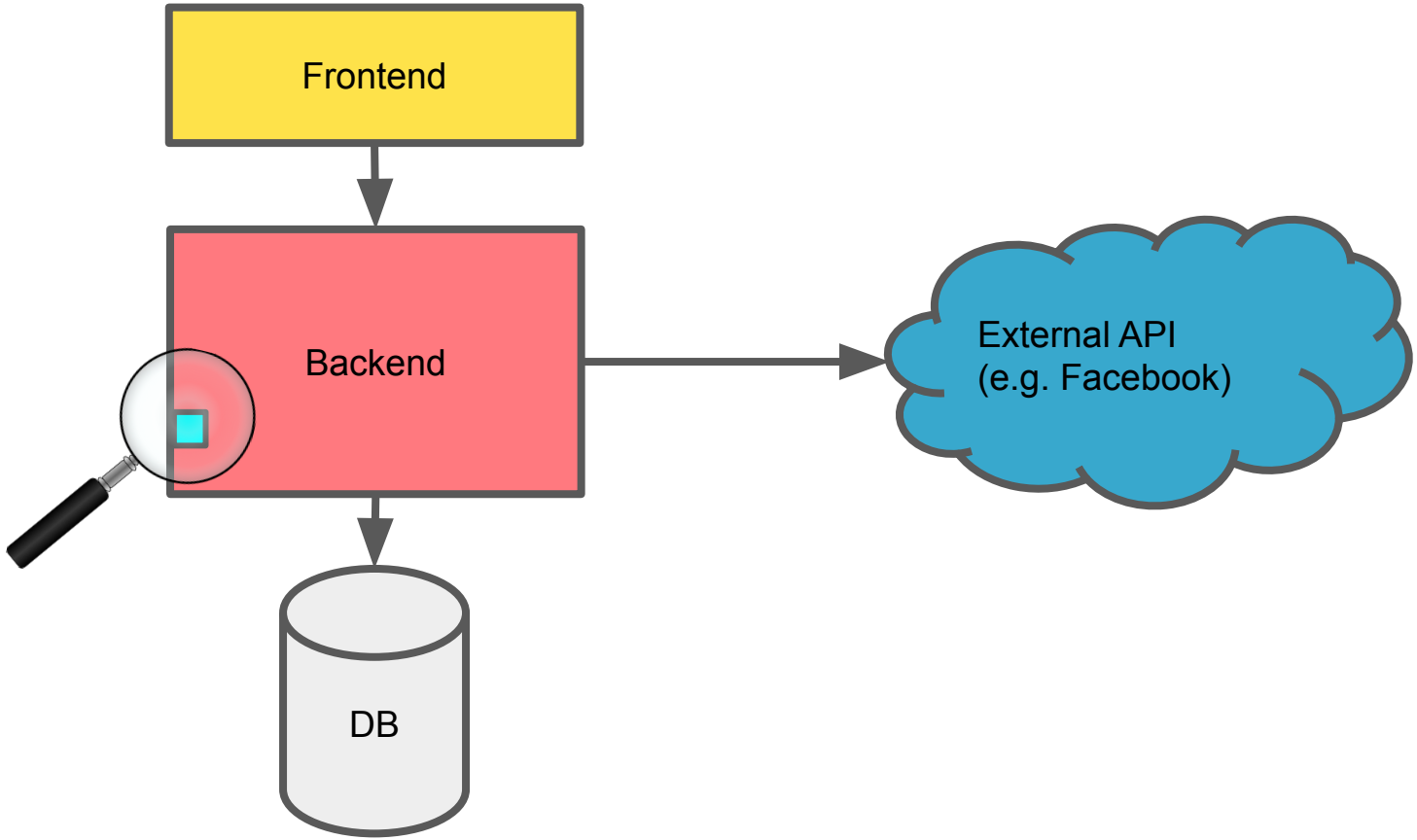
Unit tests ensure that the unit under test works as expected

What's a unit?

- There's no definitive answer to this.
- Typically a unit for functional style code is a public function
    - With input A the function returns result B
- Typically a unit for object oriented code is a class
- Strictest definition only allows testing e.g. the class itself in isolation, other classes called by it should be mocked
    - Other definitions are not that strict
    - At least mock external dependencies, http requests, databases, the filesystem

Scope of a unit test

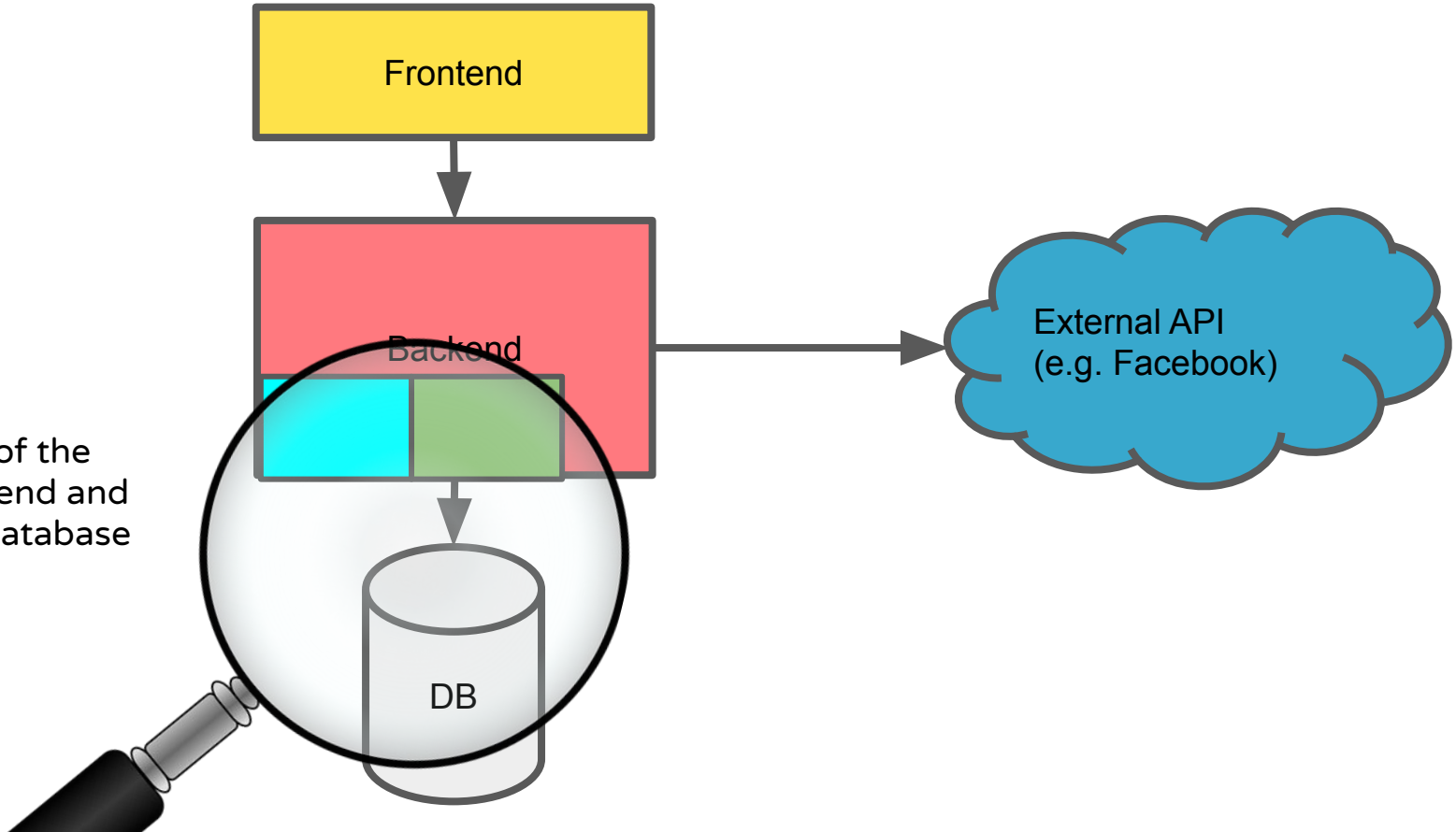
- A public function
- A class and its public methods



Integration tests

- Test functionality that integrates with e.g. databases, other services, etc
- Still would not include the whole system, those tests we would typically call end-to-end (E2E) tests
- Run database, service, etc locally on the same test machine with the application code

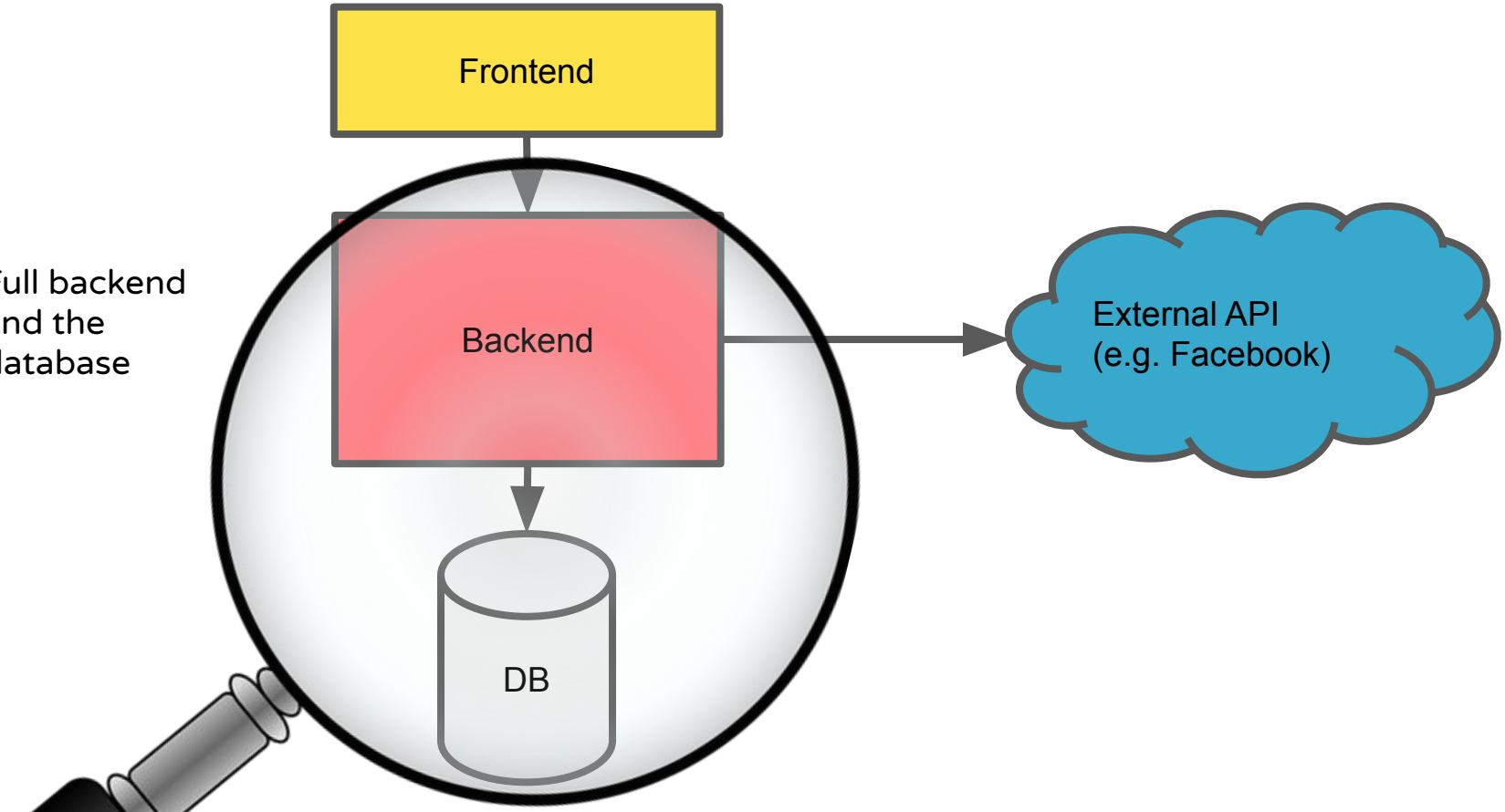
Scope of an integration test



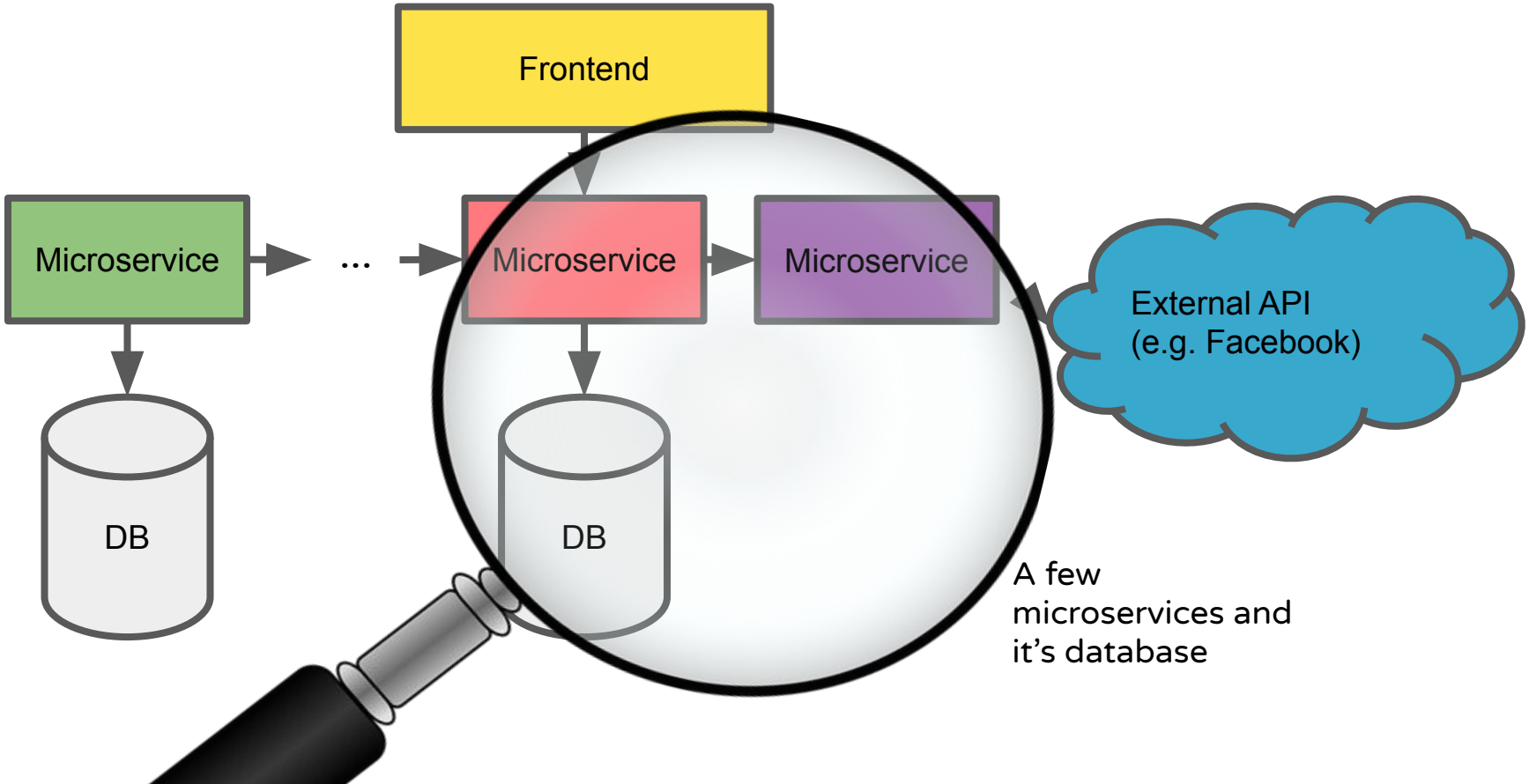
- Part of the backend and the database

Scope of an integration test

- Full backend and the database



Scope of an integration test



API tests

API tests ensure that the API contract of a service is held

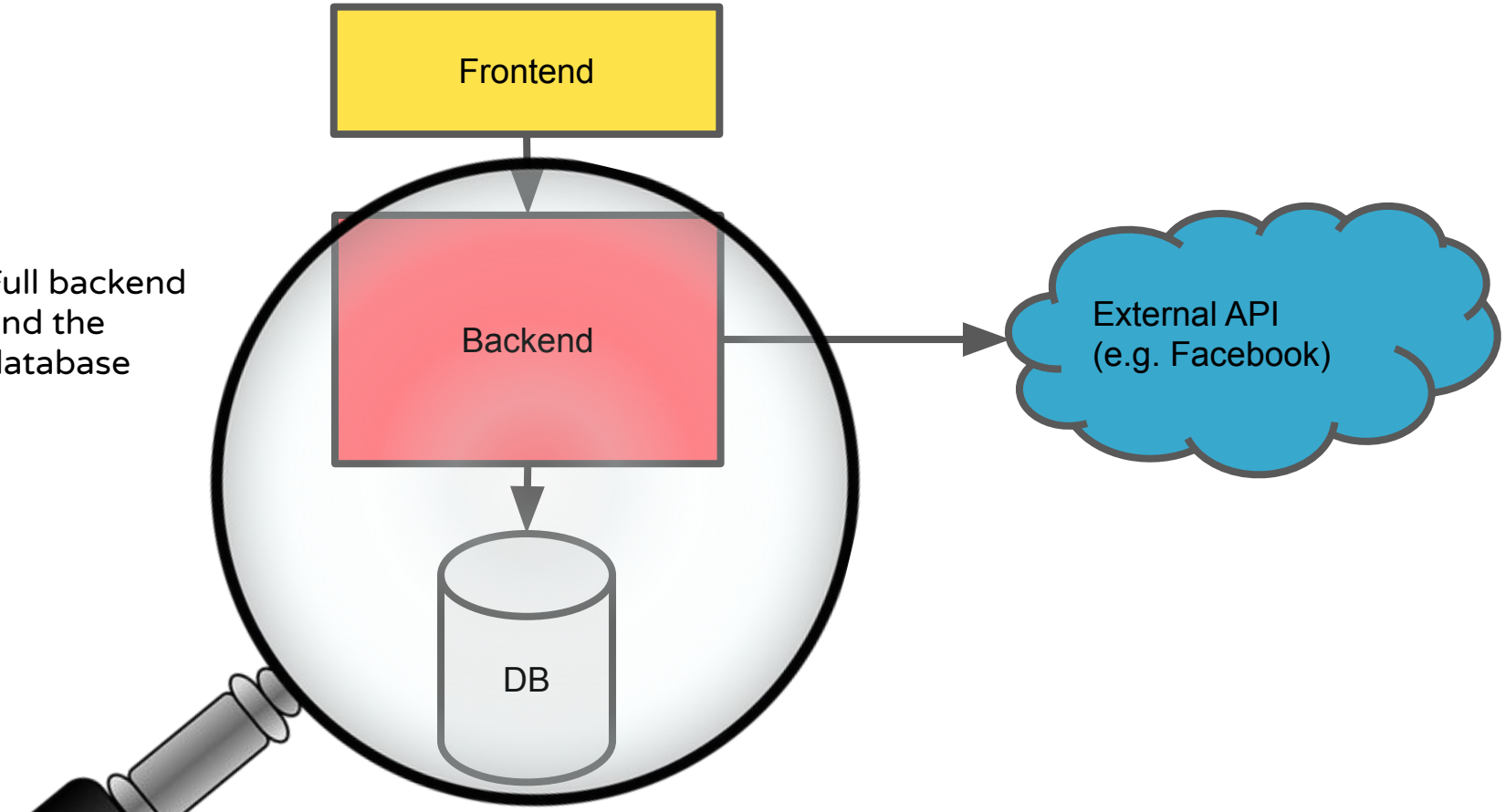
- With given input A (HTTP request to the service) to the service with state B (e.g. what's in the database), we get output C (HTTP response)
- Think of it as a unit test where the unit under test is the whole service
- Run everything the service needs (databases, etc)
- Still mock external services

Can be tested with e.g. snapshot tests

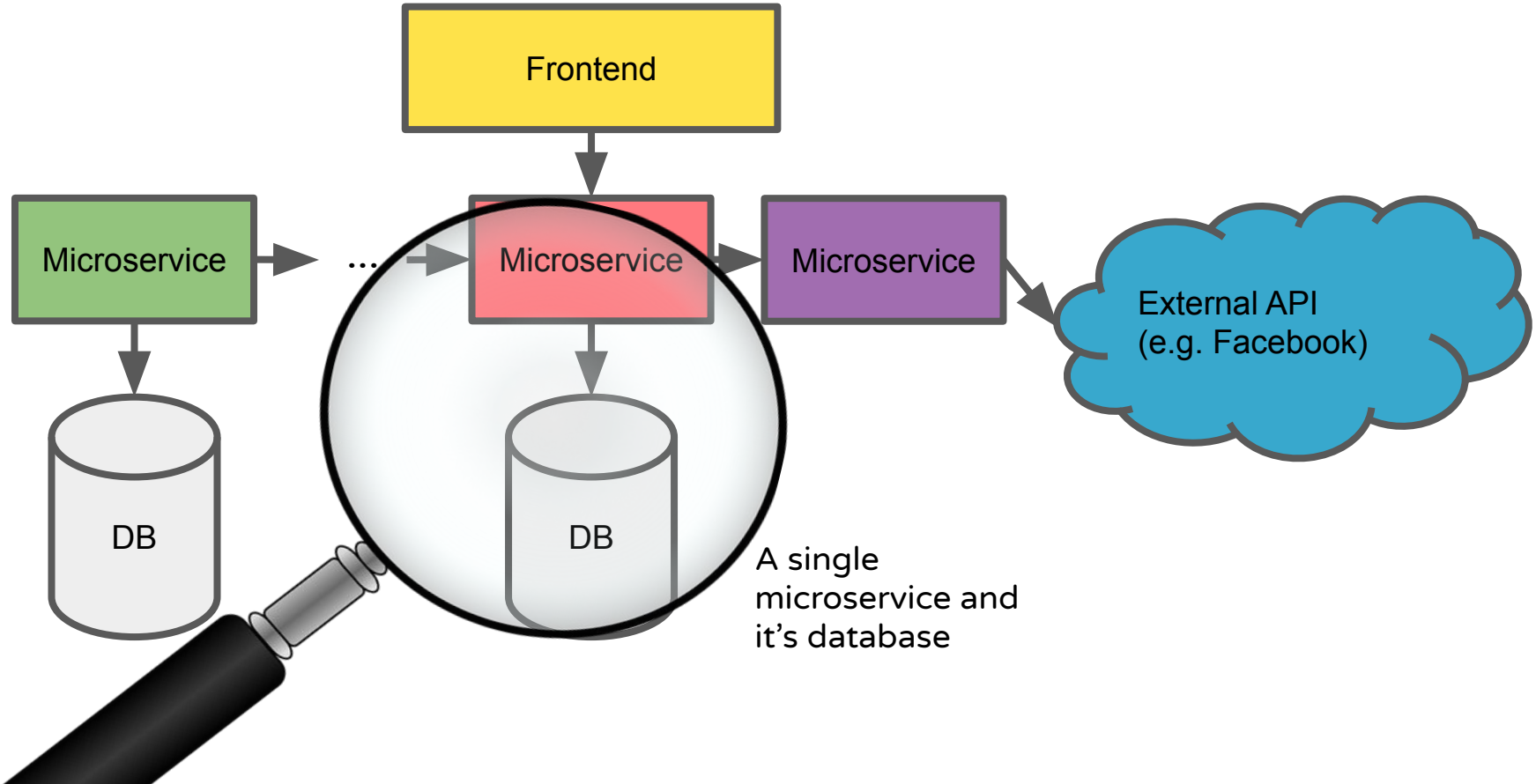
- E.g. our Pinterest team is using these to get notice when API output for given input changes

Scope of an API test

- Full backend and the database



Scope of an API test



UI tests

End-to-End (E2E) tests

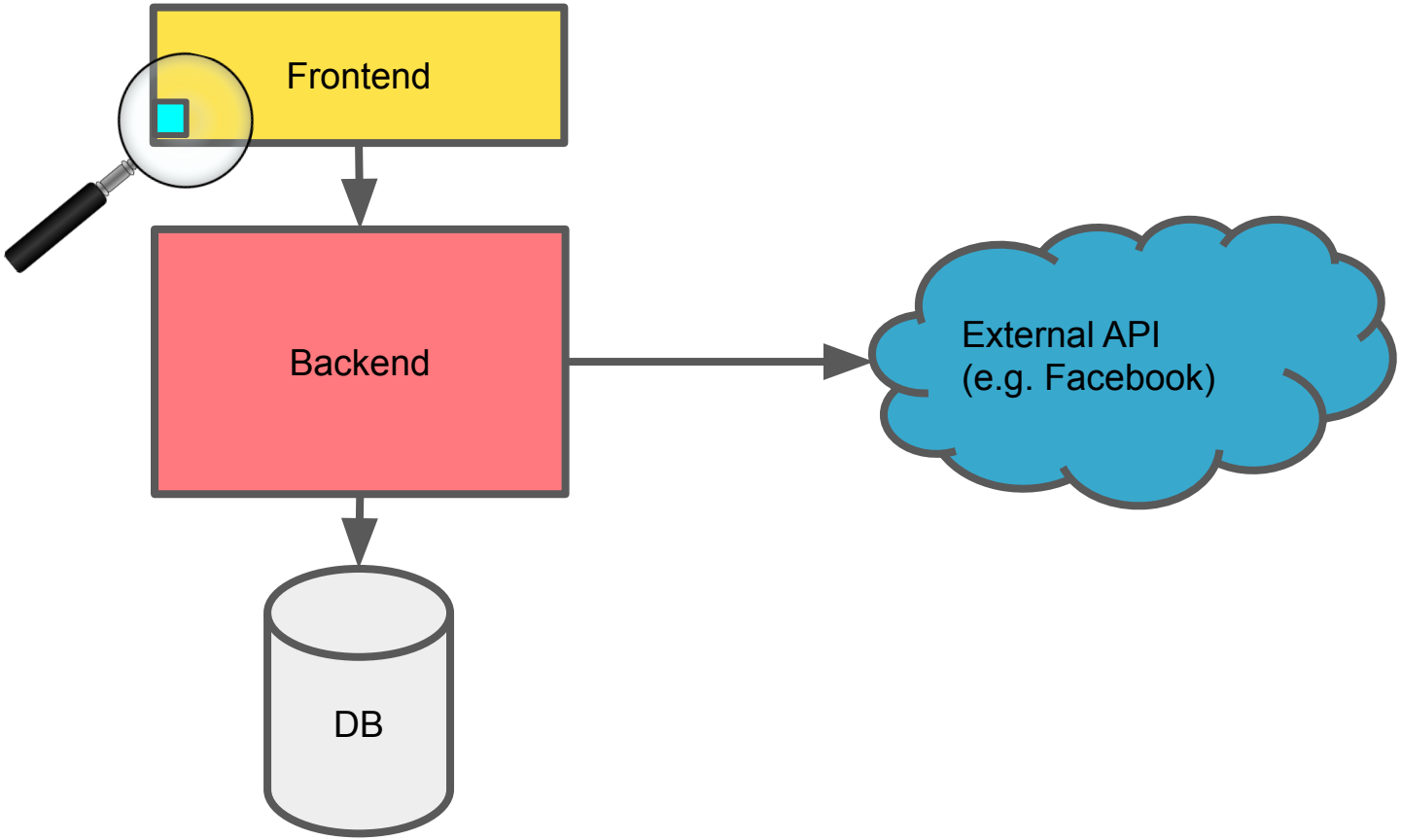
- Run the full system with all required parts, all microservices, databases, UI
- Typically test by automated tests clicking the UI and asserting expected outcomes in the UI
- Typically brittle as UIs change often

Front end unit tests

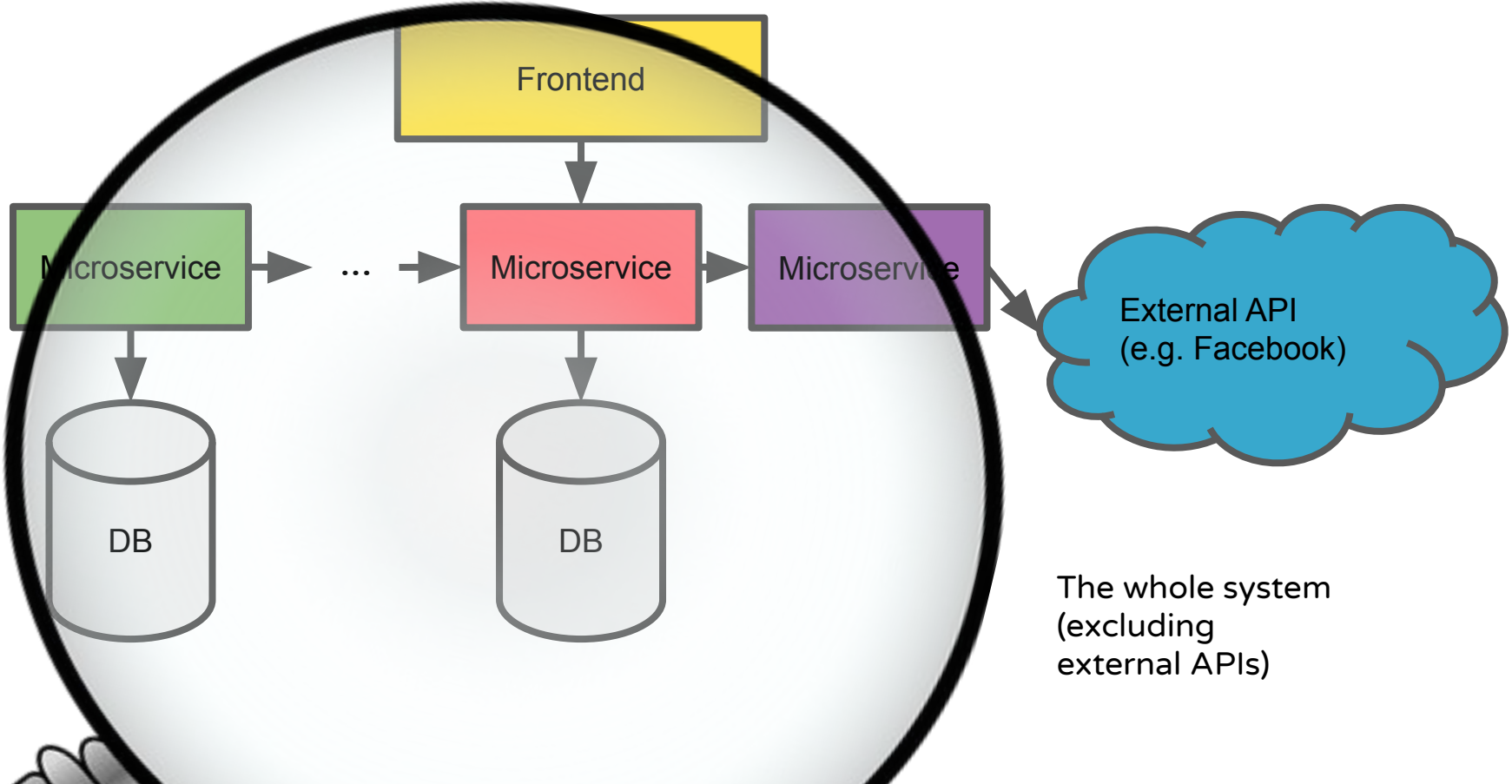
- Unit tests where the unit under test is e.g. a React component
- Mock any API calls done from the UI
- Possibility to affect isolation level by doing a “shallow render”
- Snapshot tests to see if UI changes with code changes
    - Diff of HTML is not as readable though as with JSON output of APIs

Scope of a frontend unit test

- E.g. a React component



Scope of an End-to-End test

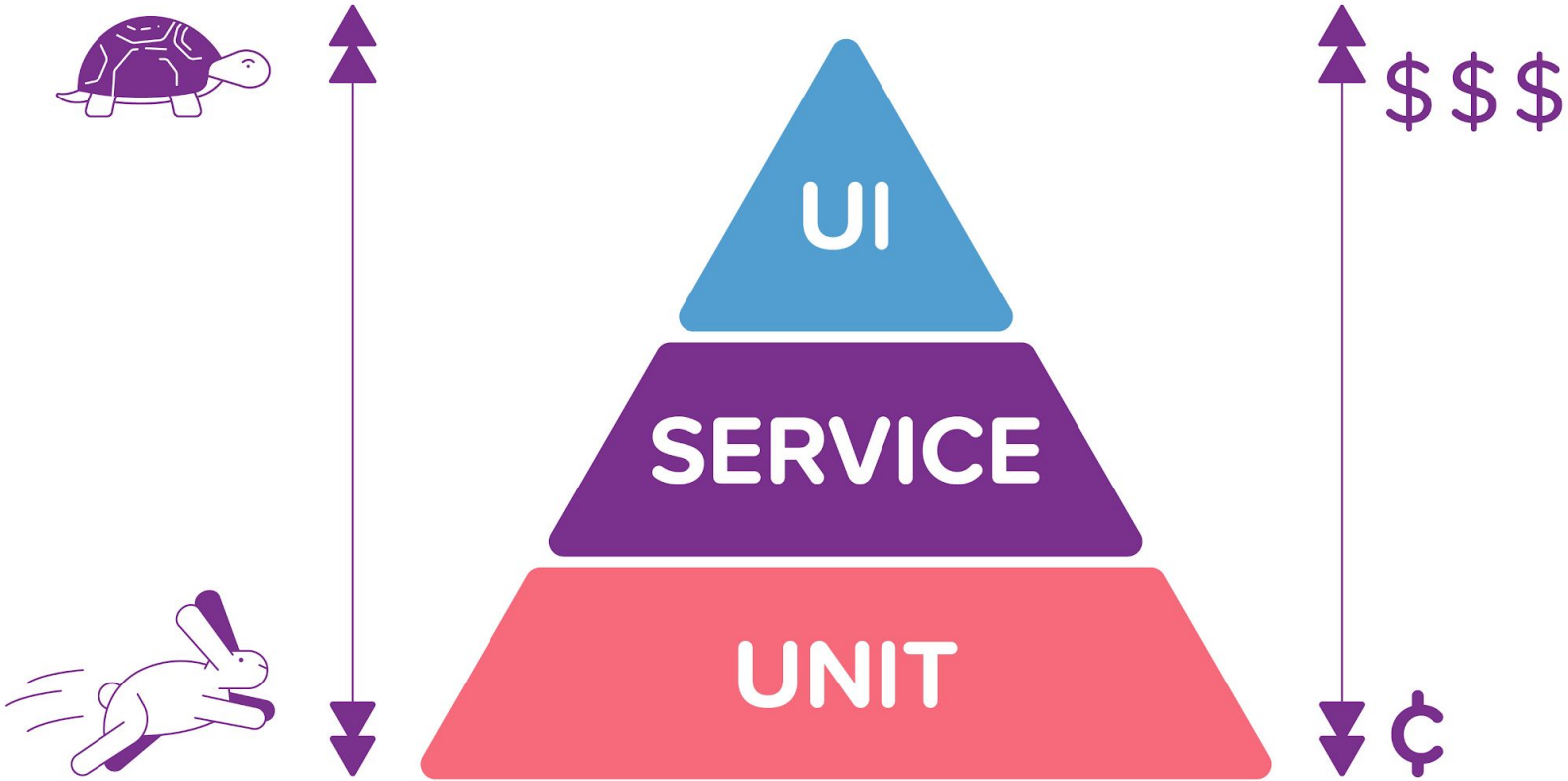


The whole system (excluding external APIs)

So what kind of tests should I write?

It depends

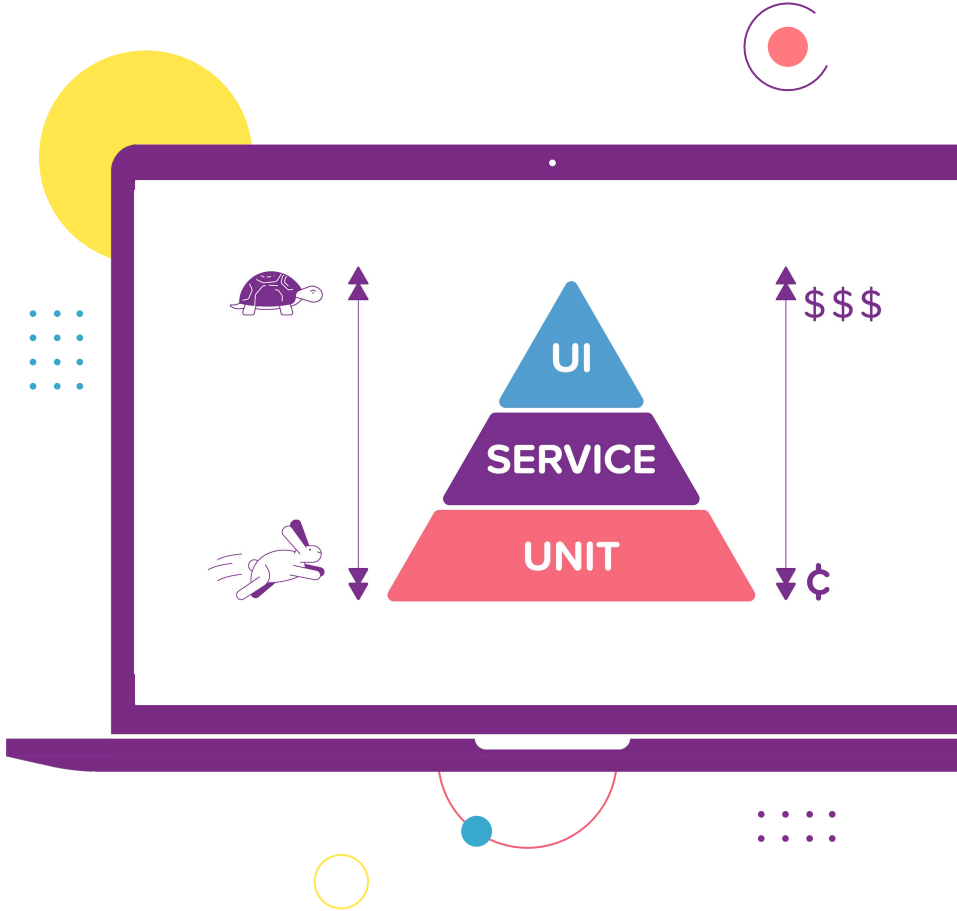
The Test Pyramid


UI
SERVICE
UNIT
$$$
¢

The Test Pyramid

Popularized by Mike Cohn in his book "Succeeding with Agile" as the "Test Automation Pyramid"

- One should have more tests with lower abstraction level (higher granularity)
  - Fast to run
- Less tests with higher abstraction level (more coarse granularity)
  - Slower to run
- Higher level tests should focus on testing the functionality that's not possible to test by lower level tests.



We have an existing product that has no automated tests.
It works and is used by customers.

Your team is assigned to develop new features into the product.

How would you start adding tests?

Legacy code dilemma

Term coined by Michael Feathers in his book "Working Effectively With Legacy Code".

- Writing unit tests for code that was not designed to be unit tested is hard
- It often it requires the code to be refactored.
- Yet to refactor the code safely, you need tests to ensure that you don't introduce regressions.
- API tests are a good means of adding test coverage without the need to change the code under test.

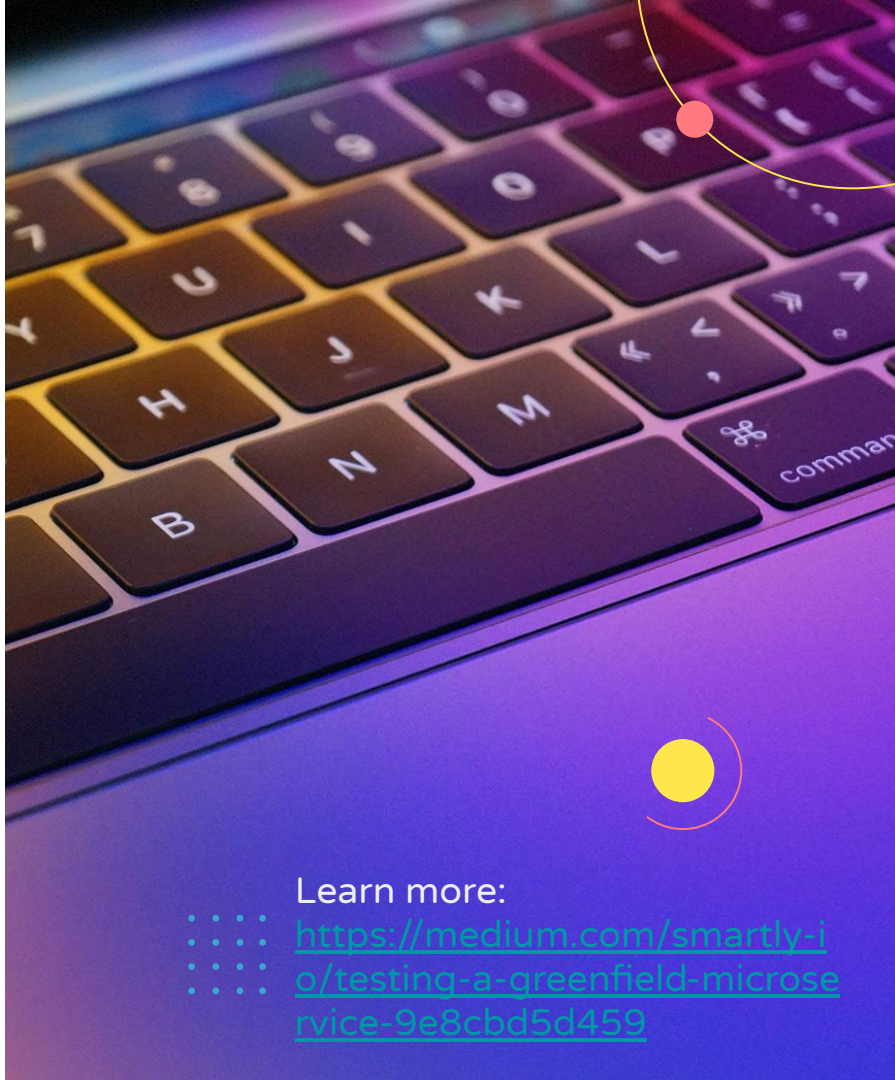
Testing at Smartly.io - How it all started

- Prototyping & finding product- market-fit
- Adding the first API tests
- Broadening the test stack

Case: Our teams recent microservice

Our team chose to only rely on API tests for the time being

- Optimize for easy rewrites
- API tests ensure API contract is held, no need to change if rewriting functionality
- If rewriting larger parts of code unit tests would be painful to keep up to date
- Eventually we'll likely start to rely more on unit tests as the code stabilizes



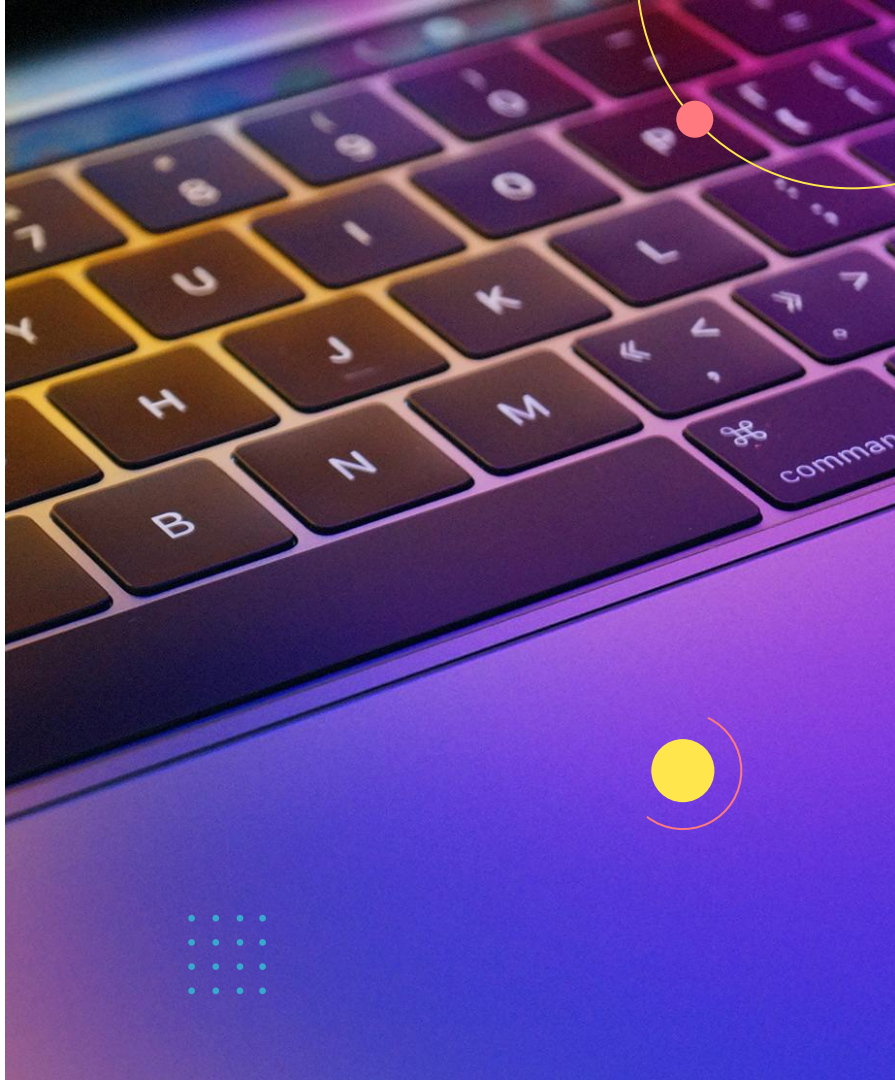
Learn more:
https://medium.com/smartly-io/testing-a-greenfield-microservice-9e8cbd5d459

Case: Our teams recent microservice

No integration tests

We have strict Openapi specifications for the API to verify that we only receive and send correct form data

- Both in production and tests
- Incorrect data will fail the test or throw an error in production (and we can quickly fix the issue)
- Learn more in our upcoming blog post!



API tests	Unit tests
Ensure API contract is held -> give good confidence that code works as intended	With only unit tests the test suite may pass while the feature is broken. This can happen if a failure occurs in the interface between modules. A classic example of this is that of doing “testing” for a human body. Unit tests for hands, legs, head and torso are passing, while the body parts are not attached to each other at all, hence not forming a full human being.
No need to change API tests when refactoring or rewriting functionality. This makes them a good safety harness for these cases.	Since unit tests are more tied to the implementation, the tests might need updating when refactoring code. Tests may even result useless when rewriting some functionality. Due to this unit tests don’t provide as good of a safety harness for refactoring as API tests.
Slower to run, as setting up the full service with databases and other dependencies takes time.	Fast to run, hence provide a faster feedback cycle for development.
If a test is failing it typically takes more time to pinpoint what actually is broken and where to start fixing.	Usually easy to pinpoint what is causing a unit test to fail.
Does not really affect code design, since it’s treating the system as a black box.	Writing code that is easily unit testable can help in designing cleaner code, e.g. with less side effects and cleaner interfaces.
Since treating the system as a black box, API tests should not require a lot of mocking, other than some external services. Yet, testing corner cases will be lead to more setup code than when testing the same corner case in isolation in a unit test.	Depending on how strict one wants to be about the level of isolation of unit tests, they will require a lot of mocking, which in the worst case will make them slow to write.

What should be tested?

What should be tested

- All non-trivial code
  - It makes sense to test any logic in the code
    - e.g. conditionals, data transformations, …
  - E.g. don’t test setters and getters of a class :D
- Happy path
- Edge cases
  - Also every time you fix a bug, start by writing a test for it, making sure it never happens again

When tests become a burden

War stories of annoying tests @ Smartly.io

API tests in our “php monolith”

- Once small change in a commonly used data model can break 500 tests, in worst case needs to be fixed individually in each file
    - In new tests we’re using factories to generate input data (only one place needs to be changed in future if data model changes)
- API tests are very slow to run
    - We’re replacing API tests with lower level integration tests that are faster to run and require less setup code

Old frontend unit tests for Angular 1 code

- Slow to run since setting up Angular takes time
    - Replace with fast React component unit tests as we’re replacing Angular with React
    - Extract non-Angular dependent code to helpers and write faster unit tests for those

War stories of annoying tests @ Smartly.io

End-to-end UI tests

- They were flaky (seemingly randomly failing)

  Risk of losing confidence in all tests: *“My code is not broken, it’s just the flaky tests”*

  - We ended up deleting them

How we test at Smartly.io

Teams divided by features

- Teams own the code building the features
- Each team decides what kind of testing makes sense for them

Typically teams do these kind of tests

- Unit testing / Snapshot testing UI components (React or Angular)
- Unit testing backend functionality
- Integration / API tests testing single microservices
- No tests testing integration of multiple microservices

Testing as a part of the CI/CD pipeline

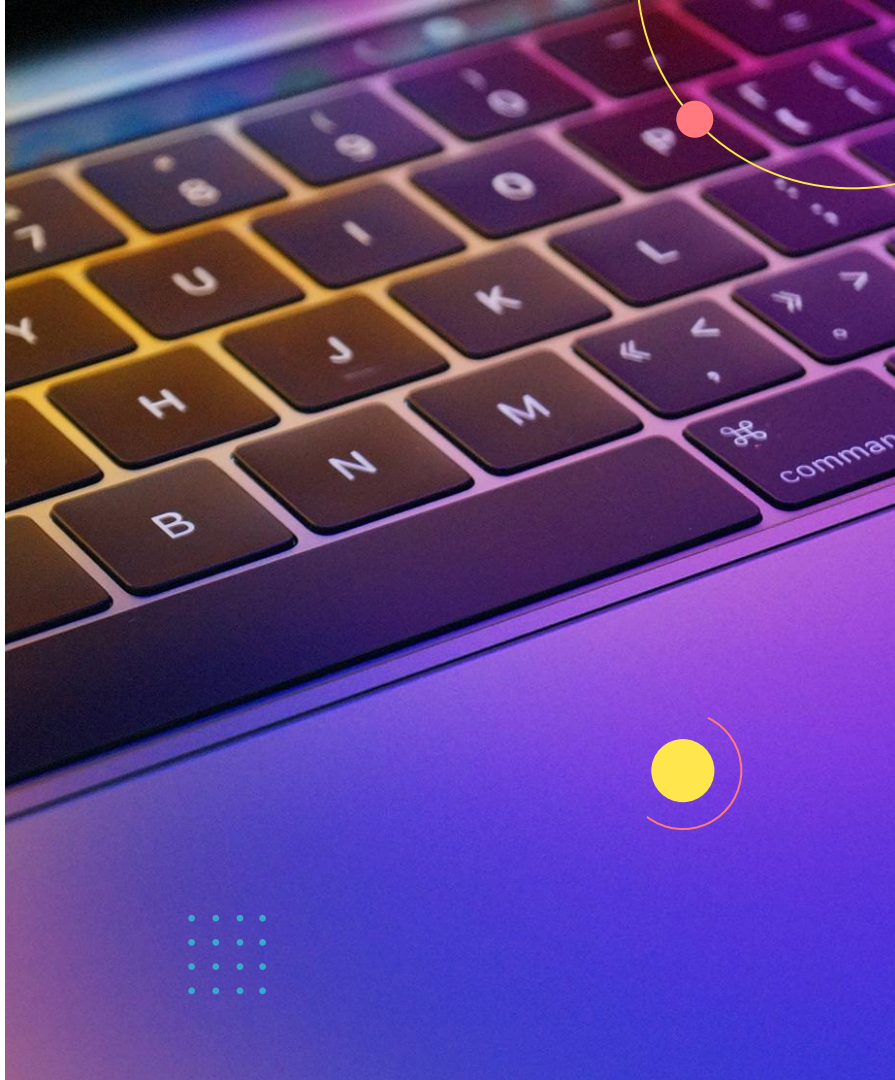
Testing as a part of the CI/CD pipeline

- Don’t rely on people running all automated tests on their computers locally

  -> Run tests as part of the build in the CI/CD pipeline
- Ensure your changes don’t break the code:
  - Run tests before merging your changes to master
- Ensure that after merging your changes to master they didn’t break
  - Run tests after merging to master, but before deploying the changes to production

General workflow at Smartly.io

- Code -> PR -> merge to master -> deploy to production
- No staging environment
  - Quick fix or revert / rollback in case of problems
- Testing in production
  - Limiting the blast radius by gradually rolling out (using “feature toggles”)
  - Monitoring to catch issues early



Other types of testing

- Property based testing
  - Generating different input data for each test run
  - Useful for testing that your code works not only with one certain hard coded input, but with any input of a specific type
  - Can be useful for finding bugs in your code
  - Also gives you a better understanding of / documenting how your code should work
    - You have to define the properties / conditions your code should satisfy

      E.g. *“When I add two numbers, the result should not depend on parameter order”*
  - We use jsverify library for this
- Mutation testing
  - Does random code changes and sees if tests still pass
  - Finding places in code that are not covered by tests

Q&A

- Why test?
- What should be tested?
- Different types of tests and when to use them
- The Test Pyramid and when to apply it
- When tests can become a burden
- Who we test at Smartly.io
- TDD
- Testing as a part of the CI/CD pipeline
- Q&A