

# Microservices and DevOps

DevOps and Container Technology

Microservice Tests

Henrik Bærbak Christensen

- Note on the 'infodeck' [Clemson, 2014]
  - <https://martinfowler.com/articles/microservice-testing/>
- *Use arrows for navigation, not page down as you miss stuff!*

# The Classic Take on Testing

- Old-time 'programs' had three levels of testing:

## Definition: **Unit test**

Unit testing is the process of executing a software unit in isolation in order to find defects in the unit itself.

The Algorithms

## Definition: **Integration test**

Integration testing is the process of executing a software unit in collaboration with other units in order to find defects in their interactions.

The Collaboration

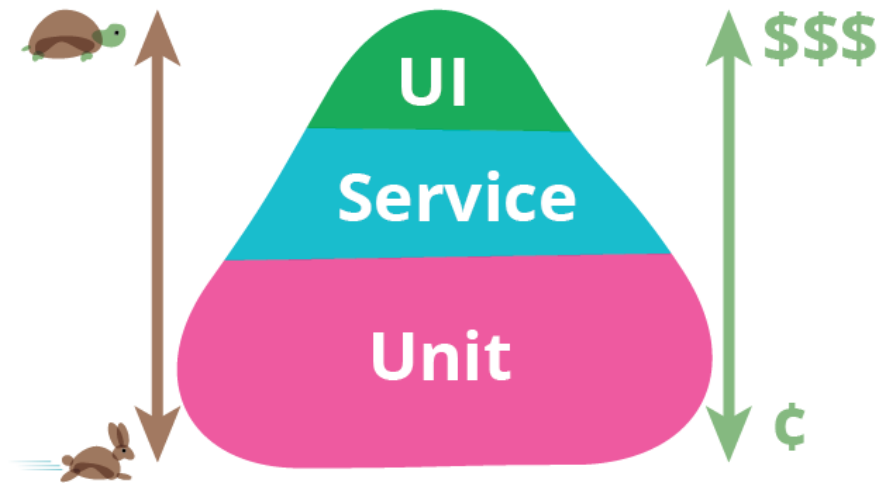
## Definition: **System test**

System testing is the process of executing the whole software system in order to find deviations from the specified requirements.

The User  
Expectation

# Test Pyramid

- Unit tests
  - Tests individual methods, classes
- Service tests
  - Tests service with doubled collaborators
- End-to-End tests
  - Functionality, user expectations

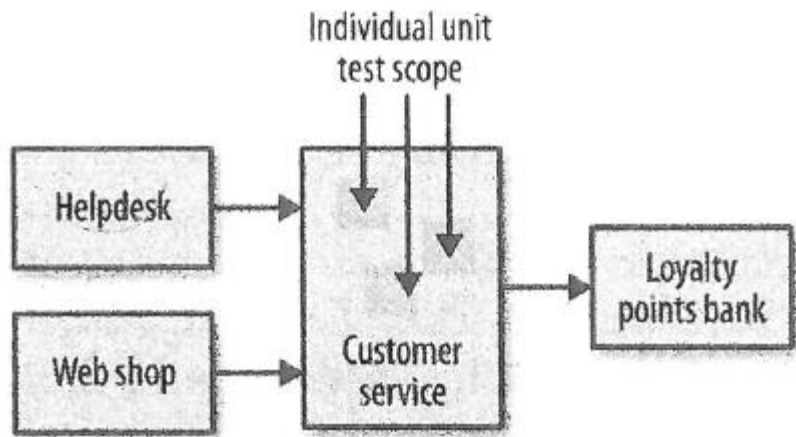


# Unit Tests

In detail the picture becomes a bit blurred...

# Unit Tests

- Testing the “algorithms”, the computation
  - Methods, the classes as a unit...
- Fast!
- We all know what unit tests are, right?
  - I would say ‘no’...



# TDD and Unit Test

- Test-Driven Development

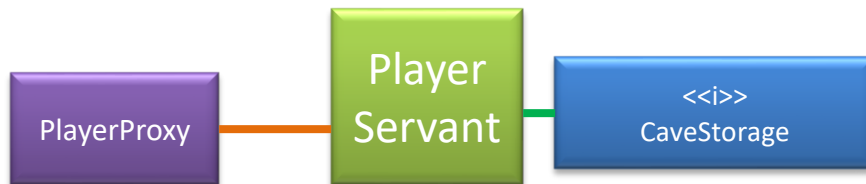
- *Clean (production) code that works* as a result of a systematic process driven by **tests...**
- Dogmatic: "No production code is ever written expect when forced to do so because otherwise a test will fail!"
- Central principle: *Fake it till you make it*
- *Need-driven development*
- *Outside-in development*

Thus it's no surprise that the mockists particularly talk about the effect of mockist testing on a design. In particular they advocate a style called need-driven development. With this style you begin developing a user story by writing your first test for the outside of your system, making some interface object your SUT. By thinking through the expectations upon the collaborators, you explore the interaction between the SUT and its neighbors - effectively designing the outbound interface of the SUT.

- *Design API before implementing it*

# Example:

- When I started the SkyCave development I had this architecture in mind:



- Very first test case in my TDD: query player's room

```

(HenrikBaerbak 2015-07-28 12:33:37 +0200 54) // TDD room description
(HenrikBaerbak 2015-07-28 12:33:37 +0200 55) @Test
(HenrikBaerbak 2015-07-28 12:33:37 +0200 56) public void shouldHaveInitialLocation() {
(HenrikBaerbak 2015-07-28 12:33:37 +0200 57)     description = player.getShortRoomDescription();
  
```

- Lead to 'playerServant' updating from storage:

```

(HenrikBaerbak 2015-07-28 12:33:37 +0200 427) private void refreshFromStorage() {
(HenrikBaerbak 2015-07-28 12:33:37 +0200 428)     PlayerRecord pr = storage.getPlayerByID(ID);
(HenrikBaerbak 2015-07-28 12:33:37 +0200 429)     name = pr.getPlayerName();
(HenrikBaerbak 2015-07-28 12:33:37 +0200 430)     groupName = pr.getGroupName();
  
```



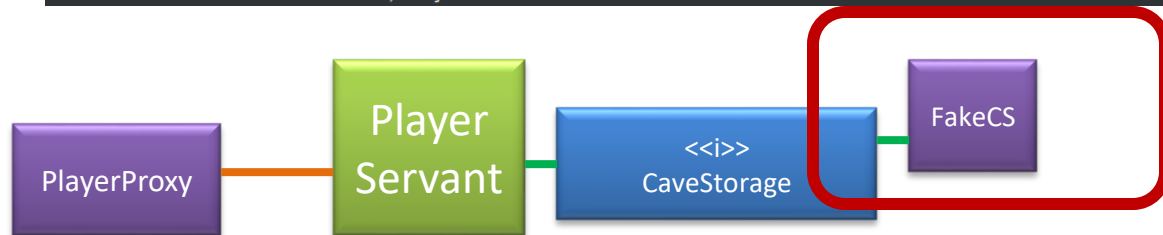
# Example:

```
(HenrikBaerbak 2015-07-28 12:33:37 +0200 54) // TDD room description
(HenrikBaerbak 2015-07-28 12:33:37 +0200 55) @Test
(HenrikBaerbak 2015-07-28 12:33:37 +0200 56) public void shouldHaveInitialLocation() {
(HenrikBaerbak 2015-07-28 12:33:37 +0200 57)     description = player.getShortRoomDescription();

(HenrikBaerbak 2015-07-28 12:33:37 +0200 427) private void refreshFromStorage() {
(HenrikBaerbak 2015-07-28 12:33:37 +0200 428)     PlayerRecord pr = storage.getPlayerByID(ID);
(HenrikBaerbak 2015-07-28 12:33:37 +0200 429)     name = pr.getPlayerName();
(HenrikBaerbak 2015-07-28 12:33:37 +0200 430)     groupName = pr.getGroupName();
```

- As there was no CaveStorage, I *faked-it*, introducing a double

```
2015-07-28 12:33:37 +0200 145) // == The table with primary key playerID whose columns are the
2015-07-28 12:33:37 +0200 146) // specifications of a given player. The private datastructure PlayerSpecs
2015-07-28 12:33:37 +0200 147) // represents the
2015-07-28 12:33:37 +0200 148) // remaining tuple values.
2015-07-28 12:33:37 +0200 149)
2015-07-28 12:33:37 +0200 150) Map<String,PlayerRecord> playerId2PlayerSpecs;
2015-07-28 12:33:37 +0200 151)
2015-07-28 12:33:37 +0200 152) @Override
2015-07-28 12:33:37 +0200 153) public PlayerRecord getPlayerByID(String playerId) {
2015-07-28 12:33:37 +0200 154)     PlayerRecord ps = playerId2PlayerSpecs.get(playerID);
2015-07-28 12:33:37 +0200 155)     return ps;
2015-07-28 12:33:37 +0200 156) }
```



# Example:

- So – what happened? What did I do???
  - Unit testing of 'player.getShortDescription()' algorithm?
  - Integration testing the player – storage collaboration?

- Bottom line

- Both 😊
- ... thus blurring the borderline!

With unit testing, you see an important distinction based on whether or not the unit under test is isolated from its collaborators.



**Sociable unit testing** focusses on testing the behaviour of modules by observing changes in their state. This treats the unit under test as a black box tested entirely through its interface.



**Solitary unit testing** looks at the interactions and collaborations between an object and its dependencies, which are replaced by **test doubles**.

In essence classic xunit tests are not just unit tests, but also mini-integration tests.

## Classical and Mockist Testing

Now I'm at the point where I can explore the second dichotomy: that between classical and mockist TDD. The big issue here is *when* to use a mock (or other double).

The **classical TDD** style is to use real objects if possible and a double if it's awkward to use the real thing. So a classical TDDer would use a real warehouse and a double for the mail service. The kind of double doesn't really matter that much.

A **mockist TDD** practitioner, however, will always use a mock for any object with interesting behavior. In this case for both the warehouse and the mail service.

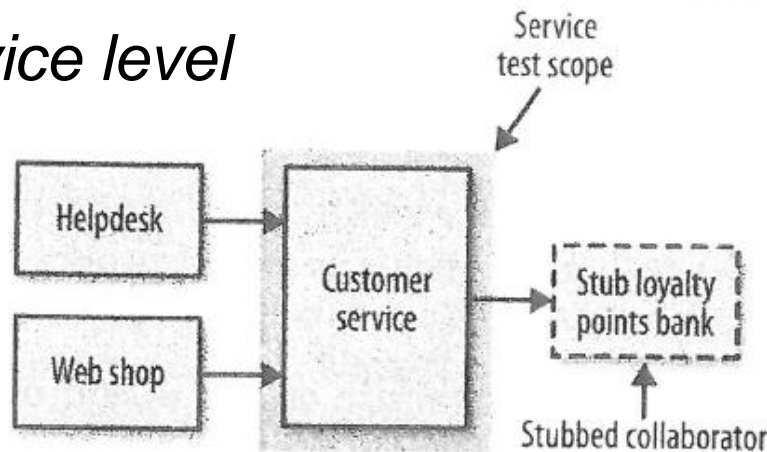
- I am a classical TDD'er. I just use doubles a lot.
- A Mockist would code the CaveStorage mock to ensure a single 'getPlayerByID()' call was made. Tend to make tests very white-box and thus more fragile - IMO!

# Service Tests

In detail the picture becomes a bit blurred again 😊...

# Service Tests

- Akin to *system tests at the service level*
  - *Bypasses the user interface and test services directly*
  - *Tests an individual service's capabilities*
  - *Need to stub out **all external collaborators** so only the service itself is in scope*



- Critique: Newman is imprecise in what ‘stubbing’ is
  - “If you decide to ... go over network to stubbed downstream collaborators”

# Fowler: Component Test

- Fowler instead calls them *component tests*
  - *limits the scope to a portion of the system under test (the microservice itself), manipulating through **internal** code interfaces, using test doubles to isolate.*
- And explicitly talks about
  - **In-process** and **out-of-process** approach for doubles

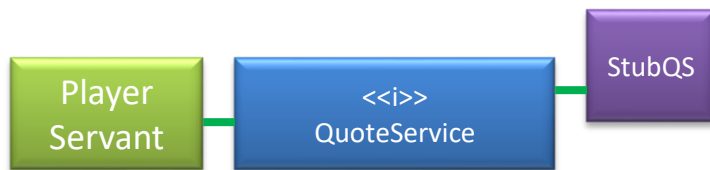


In a microservice architecture, the components are the services themselves. By writing tests at this granularity, the contract of the API is driven through tests from the perspective of a consumer. Isolation of the service is achieved by replacing external collaborators with test doubles and by using internal API endpoints to probe or configure the service.

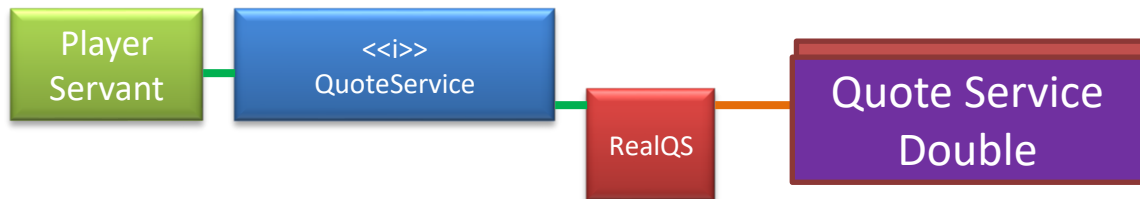
The implementation of such tests includes a number of options. Should the test execute in the same process as the service or out of process over the network? Should test doubles lie inside the service or externally, reached over the network? Should a real datastore be used or replaced with an in-memory alternative? The following section discusses this further.

# Service Test Example

- Service test 'daemon' service's quote retrieval feature
  - *In-process approach*: inject 'StubQuoteService' into PlayerServ.

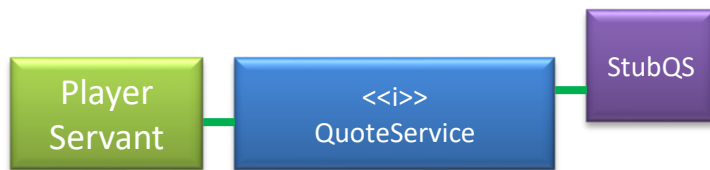


- *Out-of-process approach*: inject 'RealQuoteService' but make it call a (Mountebank) test double service

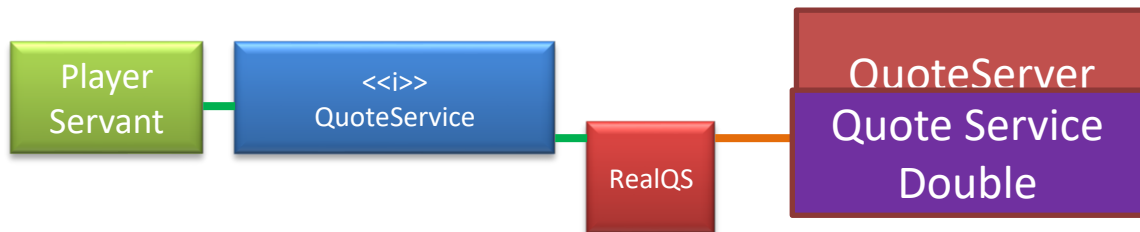


# Exercise:

- What are the benefits and liabilities of each approach?
  - In-process approach:* inject 'StubQuoteService' into PlayerServ.



- Out-of-process approach:* inject 'RealQuoteService' but make it call a Mountebank test double service





# Exercise:

- And the really tricky question:

```
@Test
public void shouldReturnCorrectlyFormattedQuotes() {
    String quote = player.getQuote(7);
    assertThat(quote, is("The true sign of intelligence is not knowledge but imagination. - Albert Einstein"));
}
```

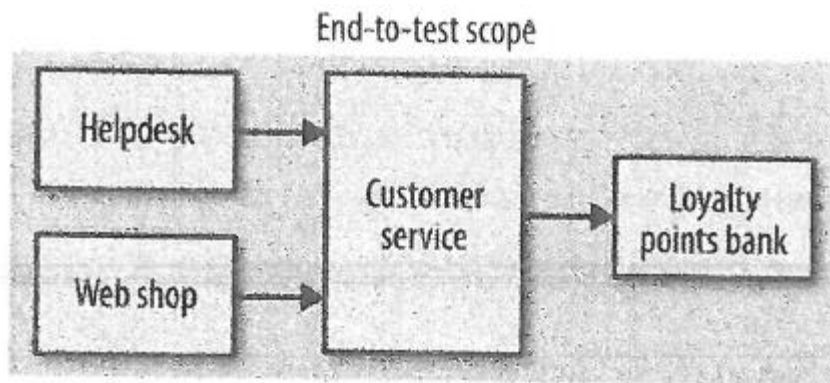
- Is the test above a...
  - Unit test of 'getQuote()'?
  - Service test of PlayerServant?

# End-to-End Tests

Perhaps not that blurred...

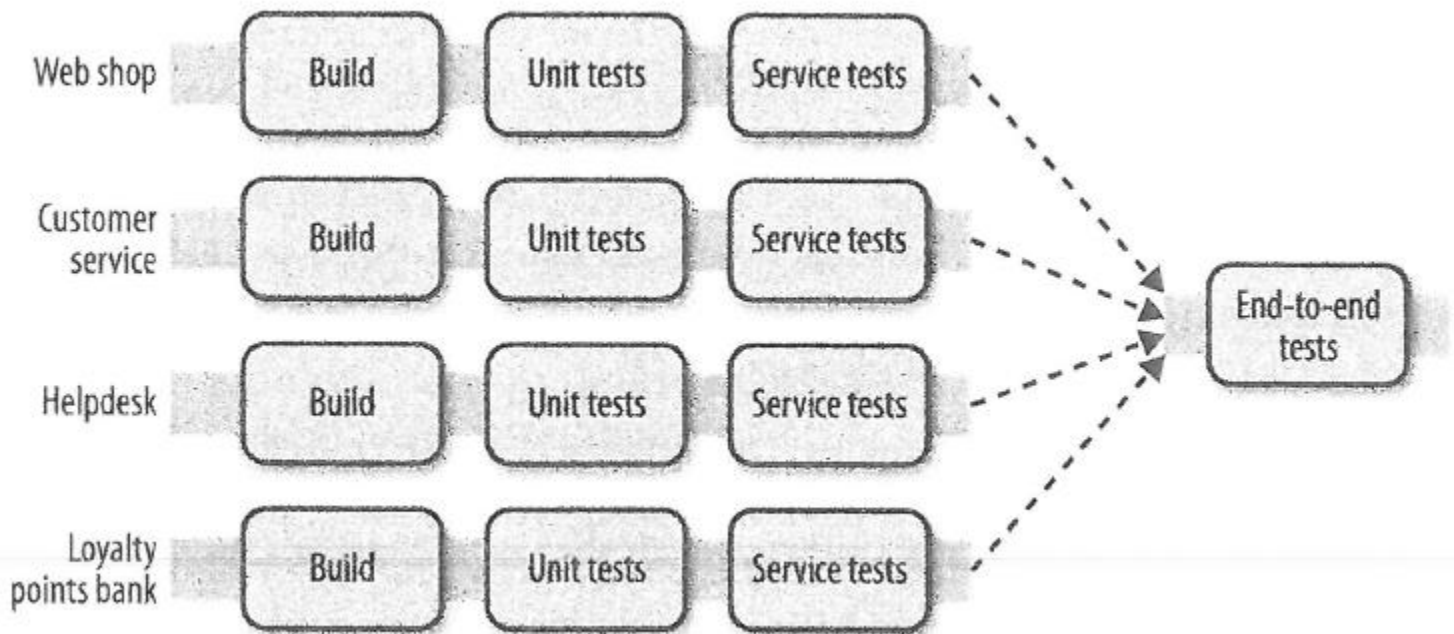
# End-to-End Tests

- User value focus: functionality
- As much as possible of system is under test
  - Real database, etc.



# In the Pipeline

- Trigger End-to-End tests on every service release
  - Ideally...



# Flaky, Brittle Tests

- *Flaky Test*: Tests that fail randomly due to non-determinism in the environment (timing, race conditions, threading, ...)
- *Normalization of deviance*: We get so used to failing tests occasionally that we start thinking that this is the norm!
  - *I.e. we get blind when failing is due to a real issue that needs fixing; and thus the issue persists!*
- Similar:
  - Getting used to 236 compiler warnings => you miss important ones!

# Test Ownerships

- Who owns the end-to-end tests?
  - Newman mentions several anti-patterns
    - Free for all ☹️.
      - Number explodes => End-to-End tests takes too long
    - Dedicated team ☹️.
      - Gets isolated from development teams, becomes bottleneck
- Best balance
  - Shared codebase, joint ownership

# Long Running Tests

- Slow running tests that are flaky are poison!
  - Slow to run, slow to diagnose, painful and slow to remedy
  - **Pile-up:** Lots of work (commits) pile up while failed E2E tests are diagnosed, corrected, rerun...
- Tests are difficult to *remove* due to human nature
  - Who gets rewarded for removing a test?
    - Especially, if that missing test would have caught a defect???

# Journeys, Not Stories

- Recommendation:
  - Have a *small* test of test cases that *tests journeys*; not test cases for every user story/use case
  - *Very low level double digits for even complex systems*
    - *5-20 journey tests?*
  - *Ordering a product, create customer, returning a product – and not much else!*



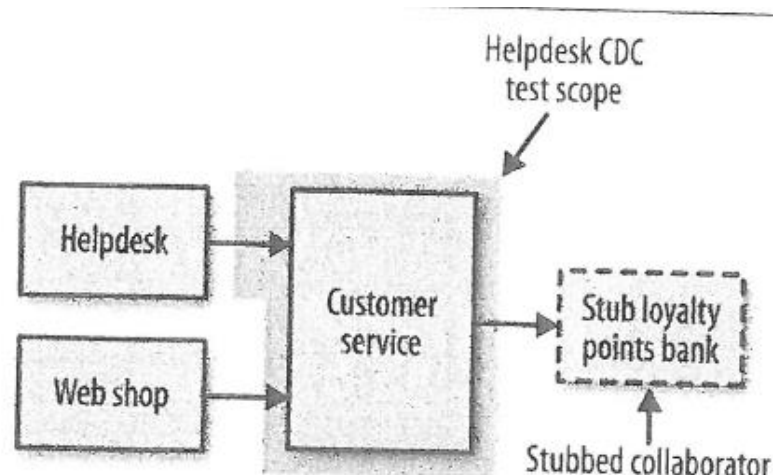
# Consumer-Driven Tests

- CDT: *Create tests that capture expectations of the consumer of the service*

- Thus ‘helpdesk’ service runs CDT’s on the Customer service
- Similar ‘web shop’ runs their own CDT’s against the CS
- Note: Test double for the loyalty point service

- CDT ownership?


- “It’s about conversations” – provide a ‘contract’ between services and thus teams. So – collaboration...



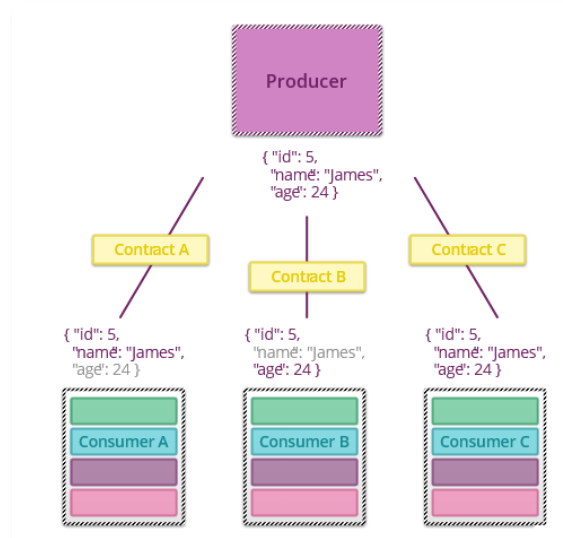
# Fowler: Contract Tests

- Fowler identifies the exact same need.

*An integration contract test is a test at the boundary of an external service verifying that it meets the contract expected by a consuming service. 1 2*

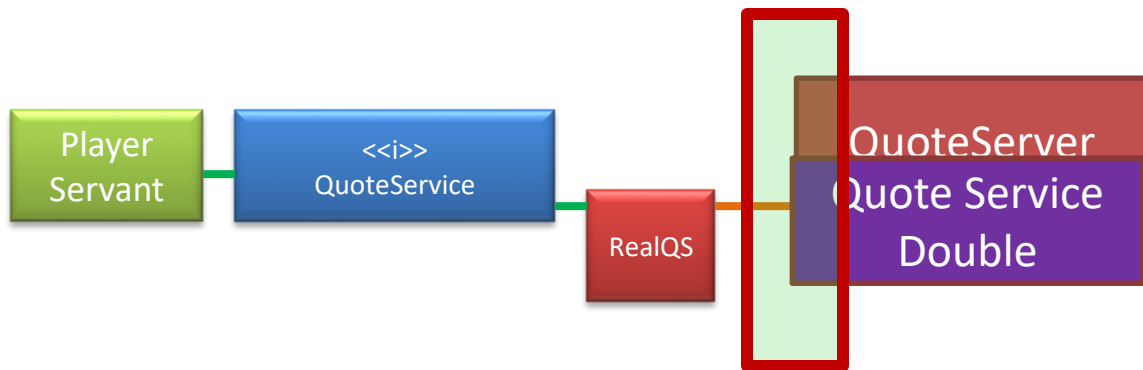


When the components involved are microservices, the interface is the public API exposed by each service. The maintainers of each consuming service write an independent test suite that verifies only those aspects of the producing service that are in use.



# What is the API Then?

- Which API does CDTs communicate by?
  - The protocol level (typically REST)



- Which of course is a **out-of-process** test
  - Slow and tedious, not part of the 'gradle test' cycle



AARHUS UNIVERSITET

# Fowler Integration Tests

Missing in Newman...

Bærbak: *Connector Tests*

# Integration Testing

*An integration test verifies the communication paths and interactions between components to detect interface defects. ①②③*



Integration tests collect modules together and test them as a subsystem in order to verify that they collaborate as intended to achieve some larger piece of

behaviour. They exercise communication paths through the subsystem to check for any incorrect assumptions each module has about how to interact with its peers.

**Gateway integration tests** allow any protocol level errors such as missing HTTP headers, incorrect SSL handling or request/response body mismatches to be flushed out at the finest testing granularity possible.

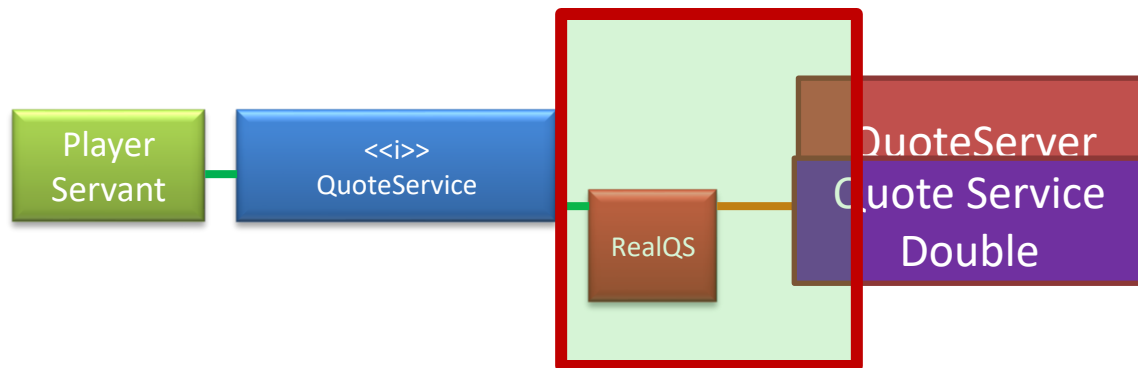
Any special case error handling should also be tested to ensure the service and protocol client employed respond as expected in exceptional circumstances.

At times it is difficult to trigger abnormal behaviours such as timeouts or slow responses from the external component. In this case it can be beneficial to use a stub version of the external component as a test harness which can be configured to fail in predetermined ways.

- Focus is more on *design for failure...*

# Integration Test

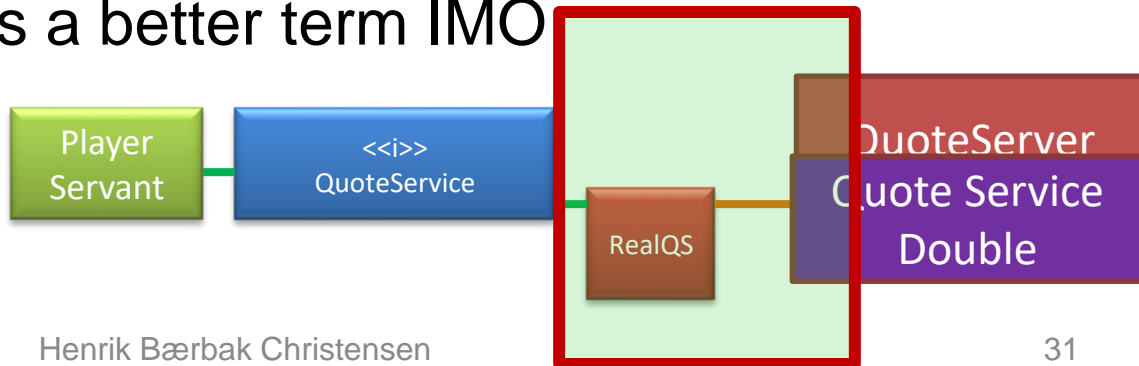
- IMO the integration tests embody testing the *connector/driver* that the consumer uses to interact with the service
  - And the connectors dealing with *failure modes*



- Which of course is a **out-of-process** test
  - Slow and tedious, not part of the 'gradle test' cycle

# Connector Test

- In the Software Architecture lingo, the dynamic view of an executing system is called the ***component connector view***
  - Component: Executing process 'doing stuff'
  - Connector: The flow of control/data between components
- Fowlers integration tests are thus (in my mind) a test of the ***connector***. Here the RealQuoteService impl.
- Connector Test** is a better term IMO



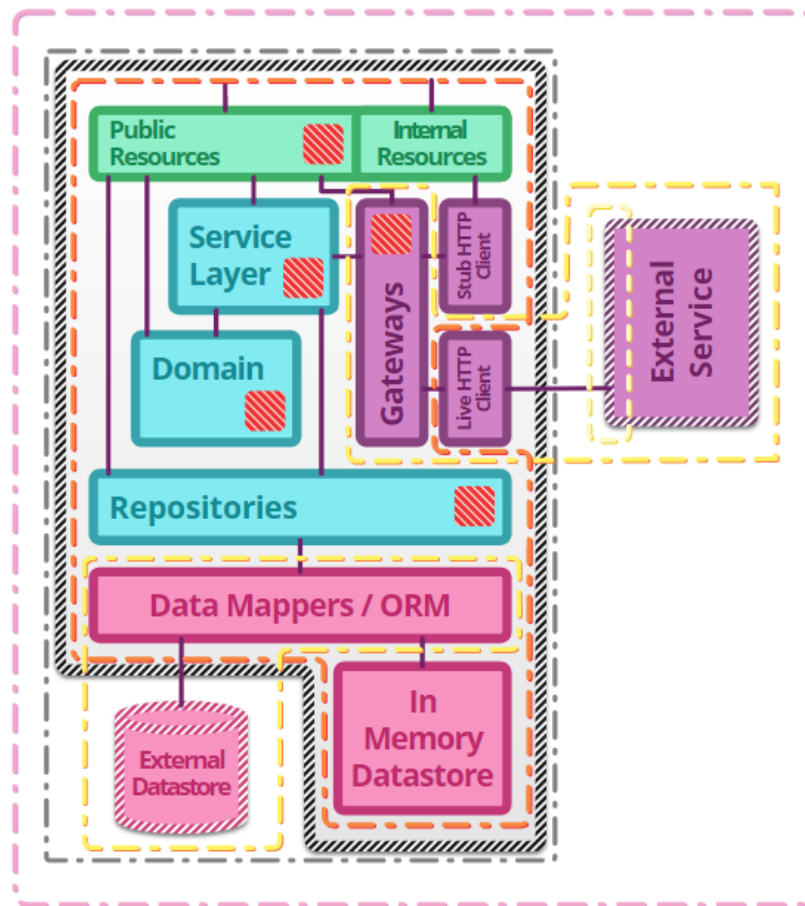
# In summary...

**Unit tests** : exercise the smallest pieces of testable software in the application to determine whether they behave as expected.

No Newman  
Equivalent

**Integration tests** : verify the communication paths and interactions between components to detect interface defects.

Bærbak:  
Connector Test



## Service tests

**Component tests** : limit the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components.

## Consumer Driven Tests

**Contract tests** : verify interactions at the boundary of an external service asserting that it meets the contract expected by a consuming service.

## Test Journeys

**End-to-end tests** : verify that a system meets external requirements and achieves its goals, testing the entire system, from end to end.



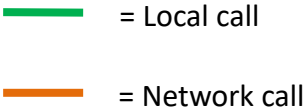


AARHUS UNIVERSITET

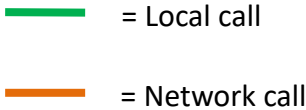
# In Practice

Relation to SkyCave

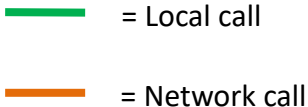
# Rich Picture Architecture



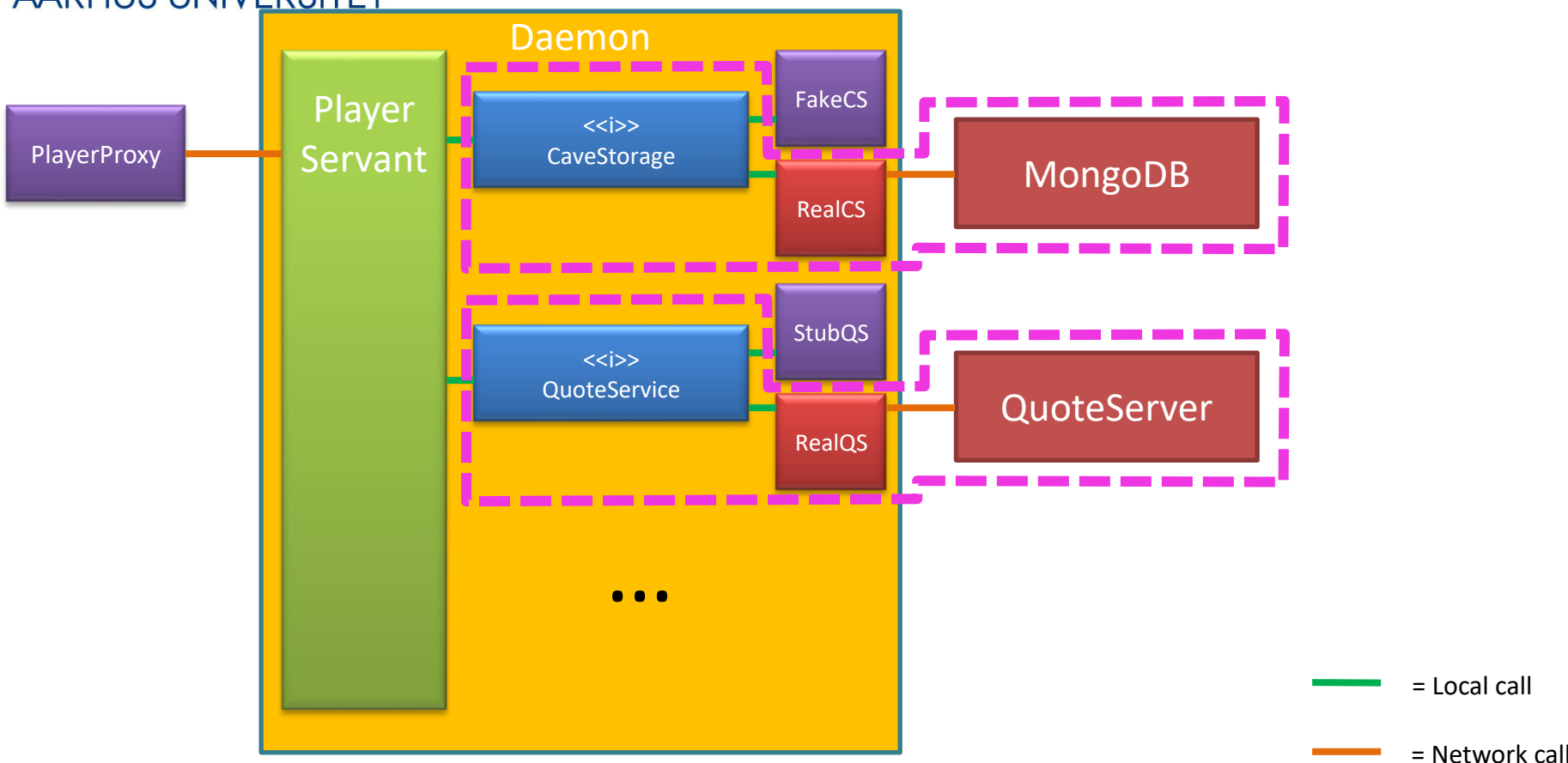
# Service Test (in-process)



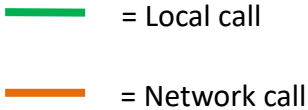
# Service Test (out-of-process)



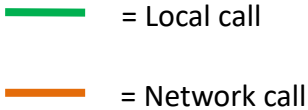
# Integration/Connector Test



# Contract Test / ConsumerDriven T.



# Test Journeys





AARHUS UNIVERSITET

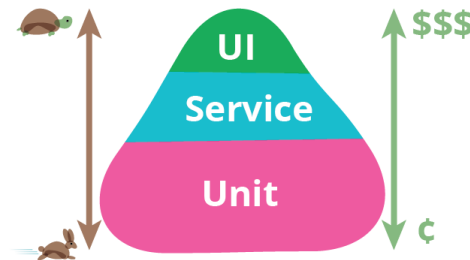
# Gradle and SkyCave

Handling Out-of-process Tests



# Precursor to ContInt

- The ‘tip of the test pyramid’ tests are slow and ‘expensive’ to run
- Conclusion:
  - Not part of normal development rhythm
    - TDD Step 4: *Run all tests and see them pass...*
    - Should be in the seconds timeframe
- Gradle ‘test’ target
  - Runs all JUnit tests ☹



# The Solution

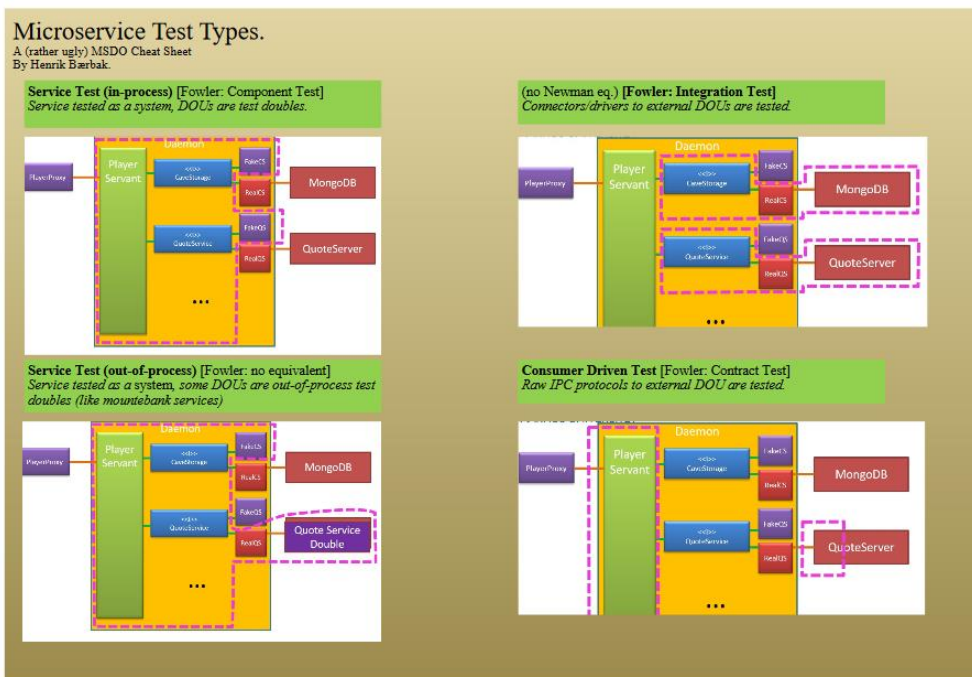
- In SkyCave I like to keep your solution code in a single Gradle project
  - Not really the microservice doctrine, but makes it possible for me to review your code!
- Subproject 'integration'
  - Task 'itest'
- Put your 'out-of-process' tests here...

```
csdev@m1:~/proj/cave$ gradle itest
> Task :common:compileJava UP-TO-DATE
> Task :server:compileJava UP-TO-DATE
```

```
cloud.cave.TestRealQuoteService > shouldTestQuoteIdAPI PASSED
cloud.cave.TestRealQuoteService > shouldTestInvalidQuoteIdAPI PASSED
cloud.cave.TestRealQuoteService > shouldTestQuoteHeaderAPI PASSED
-----
| Results: SUCCESS (25 tests, 24 successes, 0 failures, 1 skipped) |
-----
```

# Lots of Terms

- Find the 'test type' cheat sheet on Weekplan 4
  - (Yes, I am not a graphical designer 😊)



- Microservice architecture introduces a lot of new test types
  - Because integration tests have a decision to make
    - In-process integration – replace service with a local double
    - out-of-process integration – replace service with remote double
  - When out-of-process you also have two choices
    - CDT/Contract test
      - E.g. Call using raw connector, like POST, GET etc.
    - Integration test
      - E.g. Call using using 'driver' class, like 'RealQuoteService'
    - *And you often need both*
      - *CDT to facilitate suppliers development*
      - *IntTest to test your own driver code base*