Testing Strategies in a Microservice Architecture

ThoughtWorks®

There has been a shift in service based architectures over the last few years towards smaller, more focussed "micro" services. There are many benefits with this approach such as the ability to independently deploy, scale and maintain each component and parallelize development across multiple teams. However, once these additional network partitions have been introduced, the testing strategies that applied for monolithic in process applications need to be reconsidered.

Here, we plan to discuss a number of approaches for managing the additional testing complexity of multiple independently deployable components as well as how to have tests and the application remain correct despite having multiple teams each acting as guardians for different services.

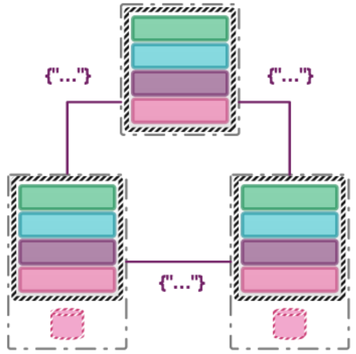
18 November 2014

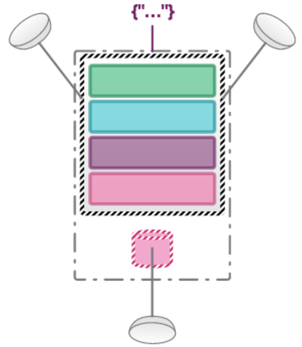
Toby Clemson is a developer at ThoughtWorks with a passion for building large scale distributed business systems. He has worked on projects in four continents and is currently based in New York.

My thanks to **Martin Fowler** for his continued support in compiling this infodeck. Thanks also to **Danilo Sato**, **Dan Coffman**, **Steven Lowe**, **Chris Ford**, **Mark Taylor**, **Praful Todkar**, **Sam Newman** and **Marcos Matos** for their feedback and contributions.

Hints for using this deck

Our agenda





First some Definitions…

- *What is a microservice?*
- *Anatomy*: a look inside a microservice
- *Architecture*: choreographing services

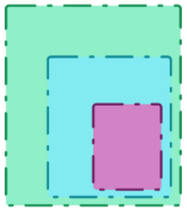
…then the Testing Strategies…

- *Unit*: mockist vs. classic
- *Integration*: datastores and external services
- *Component*: in or out of process?
- *Contract*: ensuring consistency across boundaries
- *End-to-end*: tips and tricks

…then some Conclusions

- *Options*: testing benefits of microservices
- *Test Pyramid*: how many tests?
- *Summary*: wrapping things up

A microservice architecture builds software as suites of collaborating services.



A microservice architecture is the natural consequence of applying the single responsibility principle at the architectural level. This results in a number of benefits over a traditional monolithic architecture such as independent deployability, language, platform and technology independence for different components, distinct axes of scalability and increased architectural flexibility.

In terms of size, there are no hard and fast rules. Commonly, microservices are of the order of hundreds of lines but can be tens or thousands depending on the responsibility they encapsulate. A good, albeit non-specific, rule of thumb is *as small as possible but as big as necessary* to represent the domain concept they own. "How big should a micro-service be?" has more details.

Microservices are **often integrated using REST over HTTP.** In this way, business domain concepts are modelled as resources with one or more of these managed by each service. In the most mature RESTful systems, resources are linked using hypermedia controls such that the location of each resource is opaque to consumers of the services. See the Richardson Maturity Model for more details.

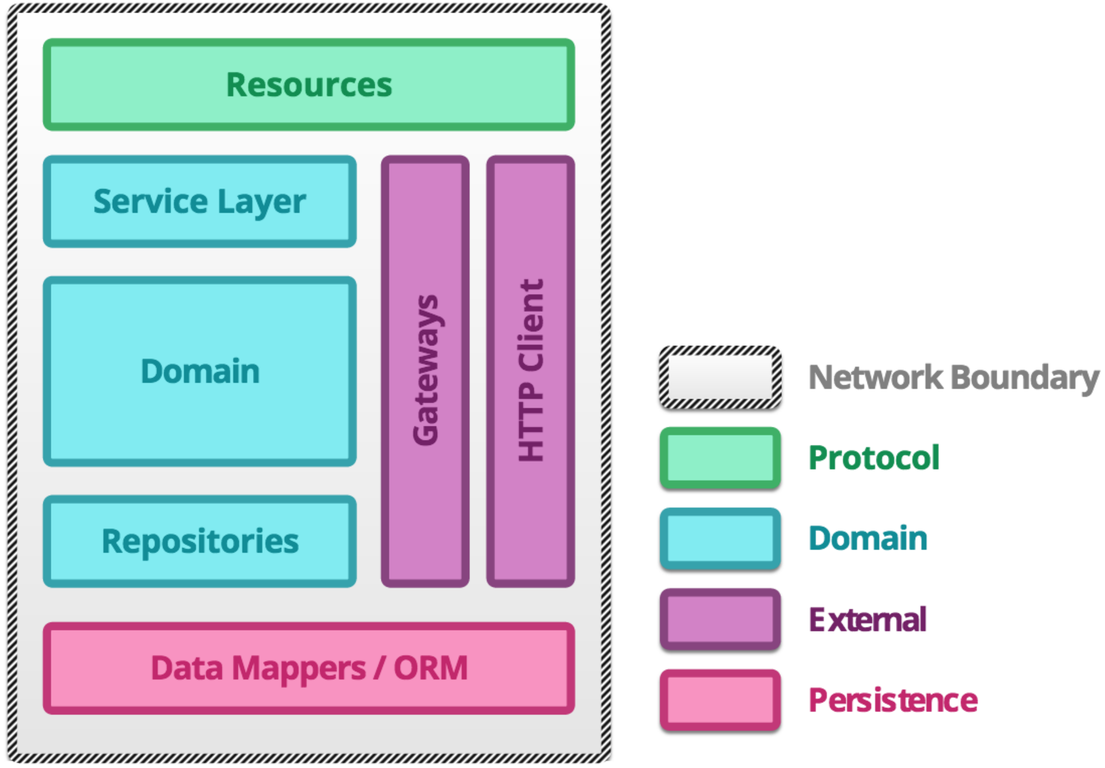
Alternative integration mechanisms are sometimes used such as lightweight messaging protocols, publish-subscribe models or alternative transports such as Protobuf or Thrift.

Each microservice may or may not provide some form of user interface.

Microservices can usually be split into similar kinds of modules

Often, microservices display similar internal structure consisting of some or all of the displayed layers.

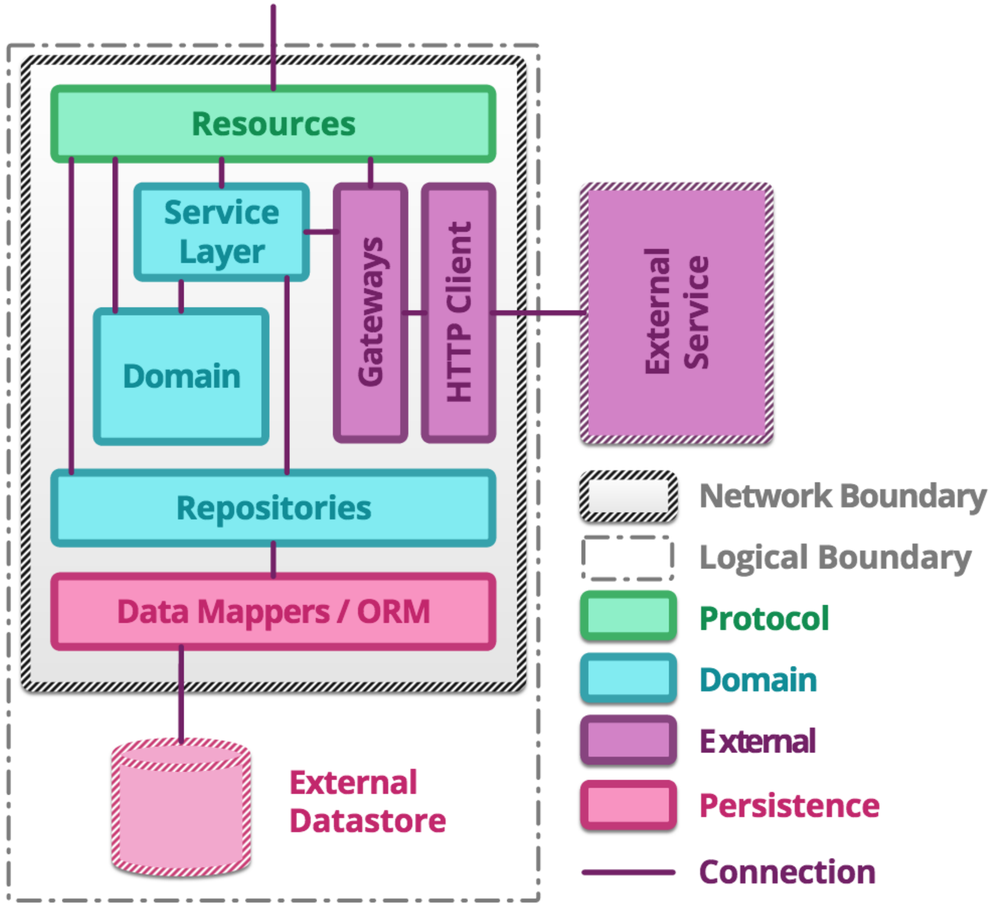
Any testing strategy employed should aim to provide coverage to each layer and between layers of the service whilst remaining lightweight.



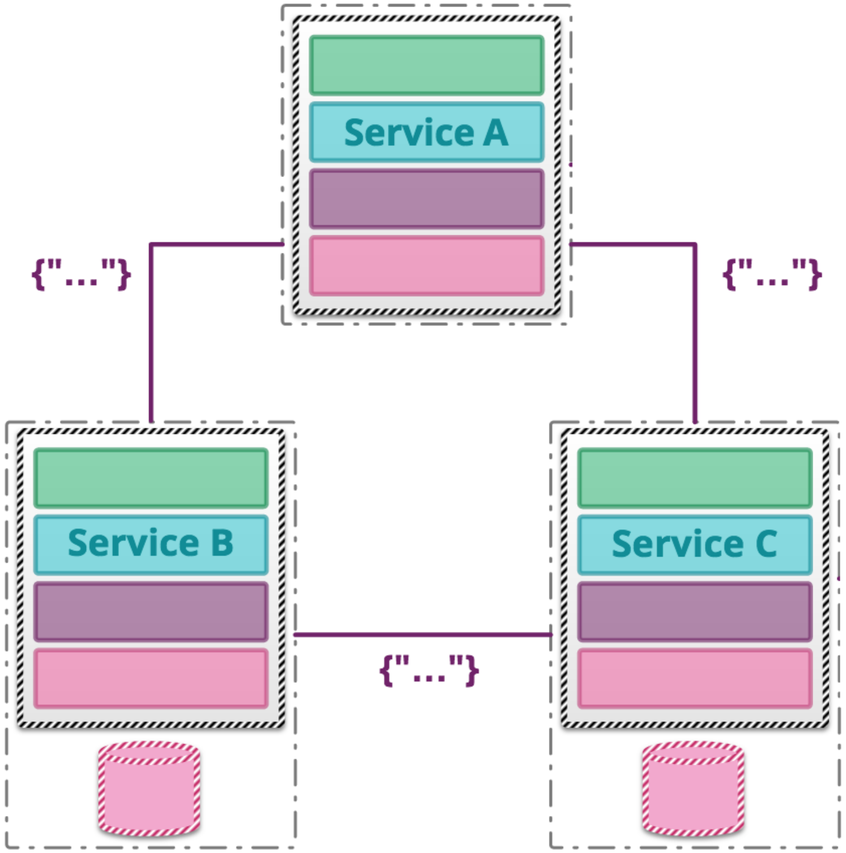
Microservices connect with each other over networks… …and make use of "external" datastores

Microservices handle requests by passing messages between each of the relevant modules to form a response. A particular request may require interaction with services, gateways or repositories and so the connections between modules are loosely defined.

Automated tests should provide coverage for each of these communications at the finest granularity possible. Thus, each test provides a focussed and fast feedback cycle.



Multiple services work together as a system...
...to provide business valuable features

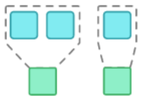


Typically, a team will act as guardians to one or more microservices. These services exchange messages in order to process larger business requests. In terms of interchange format, JSON is currently most popular although there are a number of alternatives with XML being the most common.

In some cases, an asynchronous publish-subscribe communication mechanism suits the use case better than a synchronous point-to-point mechanism. The Atom syndication format is becoming increasingly popular as a lightweight means of implementing pub-sub between microservices.

Since a business request spans multiple components separated by network partitions, it is important to consider the possible failure modes in the system. Techniques such as timeouts, circuit breakers and bulkheads can help to maintain overall system uptime in spite of a component outage.

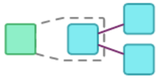
A unit test exercises the smallest piece of testable software in the application to determine whether it behaves as expected. [1][2][3]



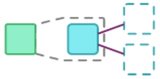
The size of the unit under test is not strictly defined, however unit tests are typically written at the class level or around a small group of related classes. The smaller the unit under test the easier it is to express the behaviour using a unit test since the branch complexity of the unit is lower.

Often, difficulty in writing a unit test can highlight when a module should be broken down into independent more coherent pieces and tested individually. Thus, **alongside being a useful testing strategy, unit testing is also a powerful design tool**, especially when combined with test driven development.

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.

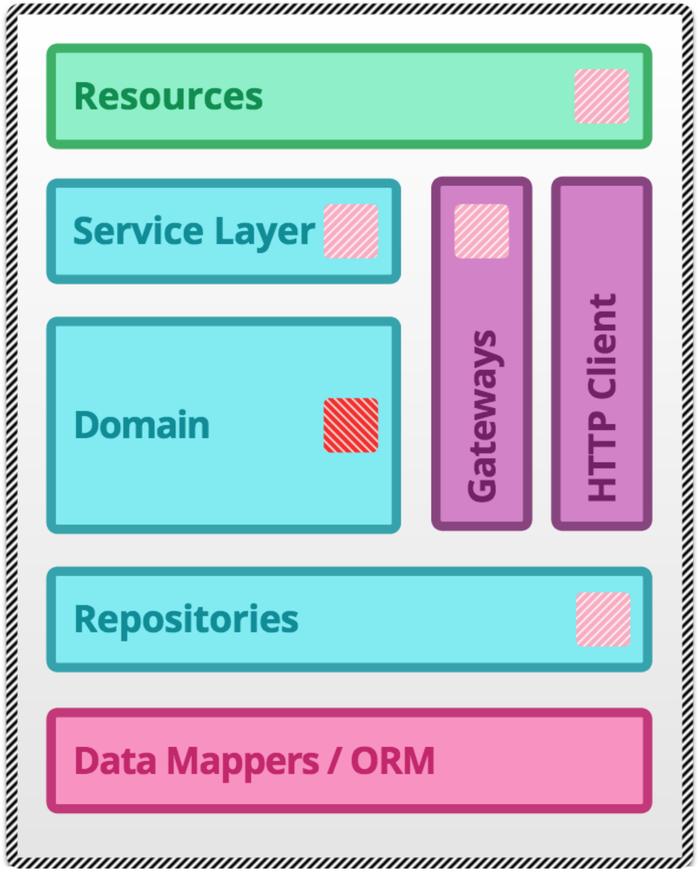


Solitary unit testing looks at the interactions and collaborations between an object and its dependencies, which are replaced by test doubles.

These styles are not competing and are frequently used in the same codebase to solve different testing problems.

UNIT TESTING

Both styles of unit testing play an important role inside a microservice



Services are commonly a rich domain surrounded by plumbing and coordination code.

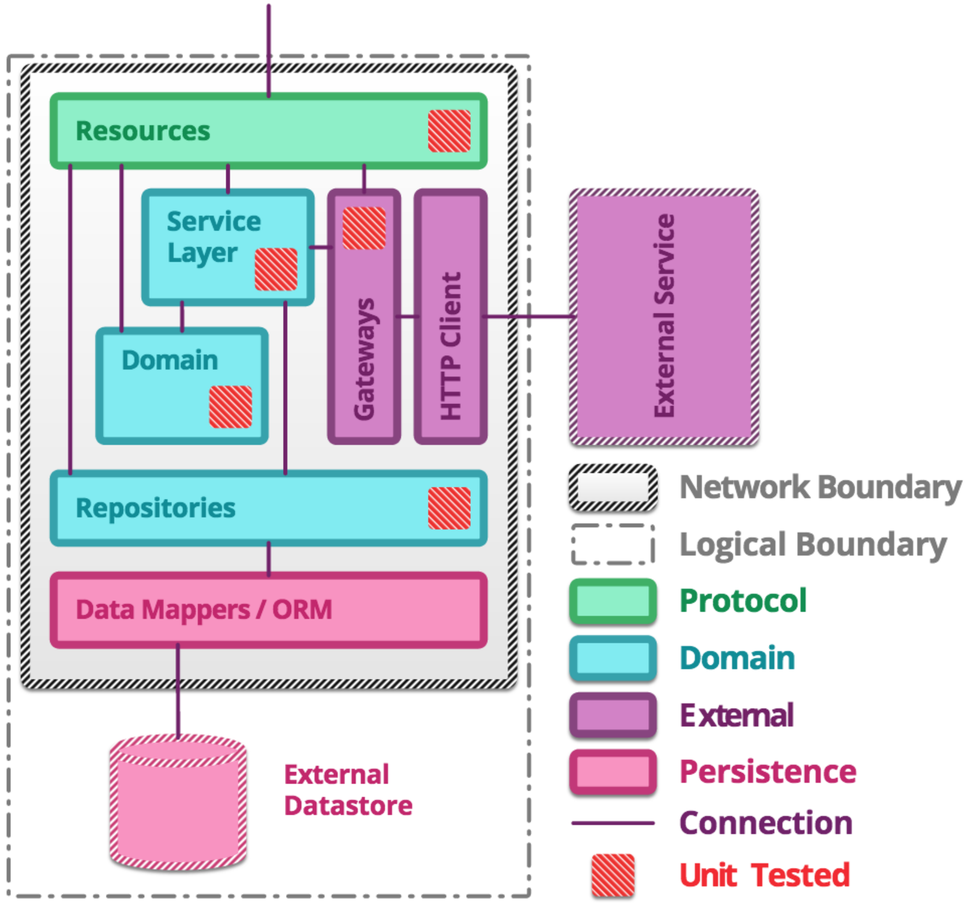
Unit - Solitary

Unit - Sociable

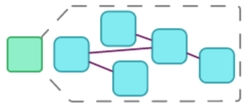
Unit testing alone doesn't provide guarantees about the behaviour of the system

So far we have good coverage of each of the core modules of the system in isolation. However, there is no coverage of those modules when they work together to form a complete service or of the interactions they have with remote dependencies.

To verify that each module correctly interacts with its collaborators, more coarse grained testing is required.

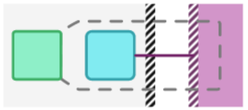


An integration test verifies the communication paths and interactions between components to detect interface defects. [1][2][3]



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.

This is in contrast to unit tests where, even if using real collaborators, the goal is to closely test the behaviour of the unit under test, not the entire subsystem.

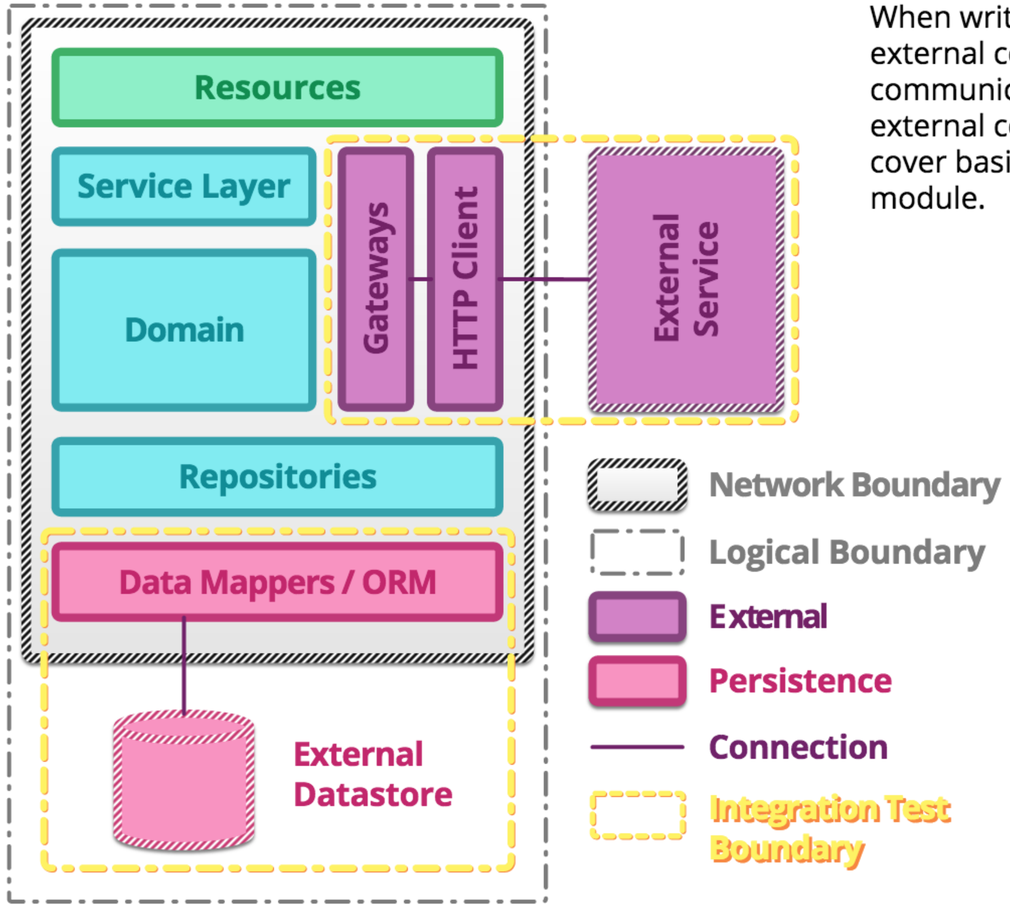


Whilst tests that integrate components or modules can be written at any granularity, in microservice architectures they are typically used to verify interactions between layers of integration code and the external components to which they are integrating.

Examples of the kinds of external components against which such integration tests can be useful include other microservices, data stores and caches.

INTEGRATION TESTING

Integrations with data stores and external components...
...benefit from the fast feedback of integration tests



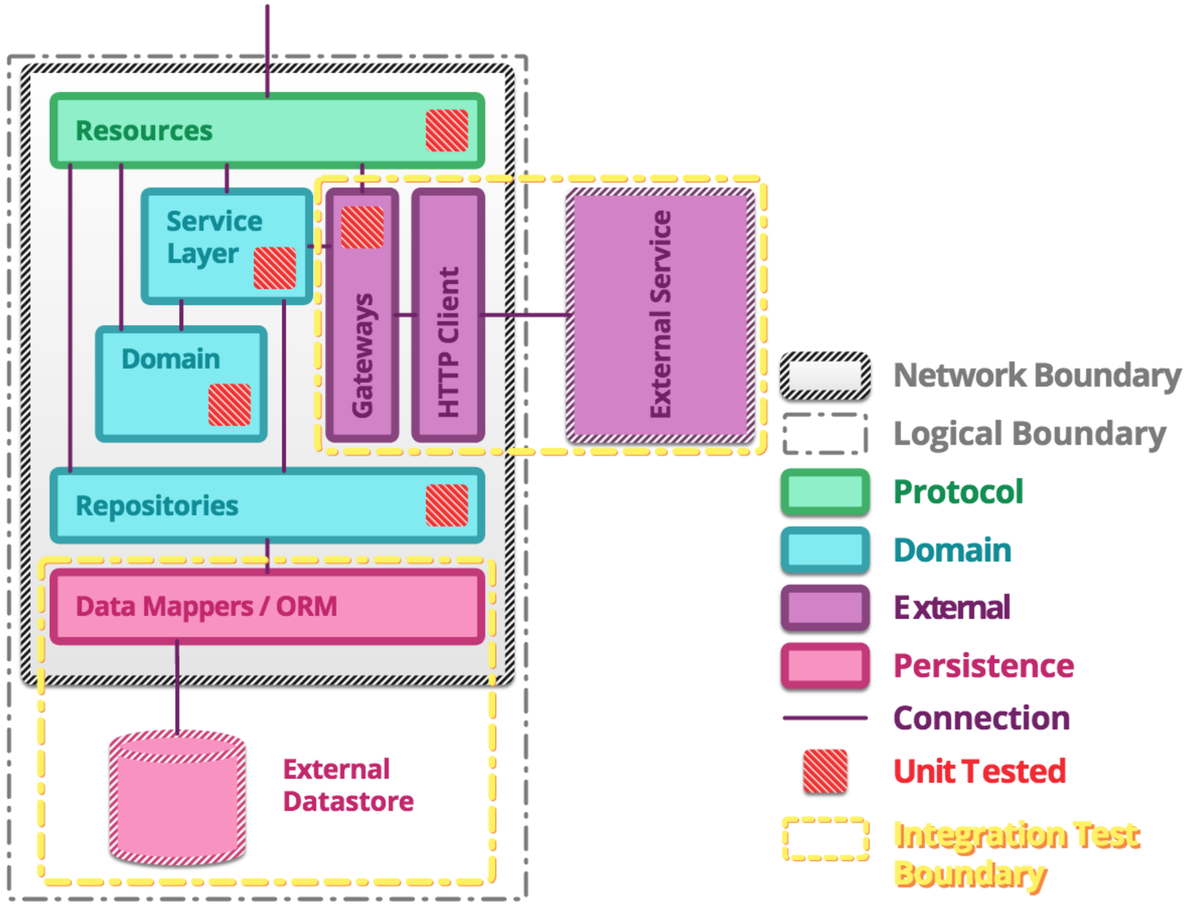
When writing automated tests of the modules which interact with external components, the goal is to verify that the module can communicate sufficiently rather than to acceptance test the external component. As such, tests of this type should aim to cover basic success and error paths through the integration module.

Without more coarse grained tests of the microservice...
...we have no confidence the business requirements are satisfied

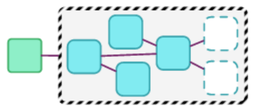
Through unit and integration testing, we can have confidence in the correctness of the logic contained in the individual modules that make up the microservice.

However, without a more coarse grained suite of tests, we cannot be sure that the microservice works together as a whole to satisfy business requirements.

Whilst this can be achieved with fully integrated end-to-end tests, more accurate test feedback and smaller test runtimes can be obtained by testing the microservice isolated from its external dependencies.



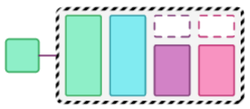
> *A component test limits the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components.* [1]



A component is any well-encapsulated, coherent and independently replaceable part of a larger system.

Testing such components in isolation provides a number of benefits. By limiting the scope to a single component, it is possible to thoroughly acceptance test the behaviour encapsulated by that component whilst maintaining tests that execute more quickly than broad stack equivalents.

Isolating the component from its peers using test doubles avoids any complex behaviour they may have. It also helps to provide a controlled testing environment for the component, triggering any applicable error cases in a repeatable manner.

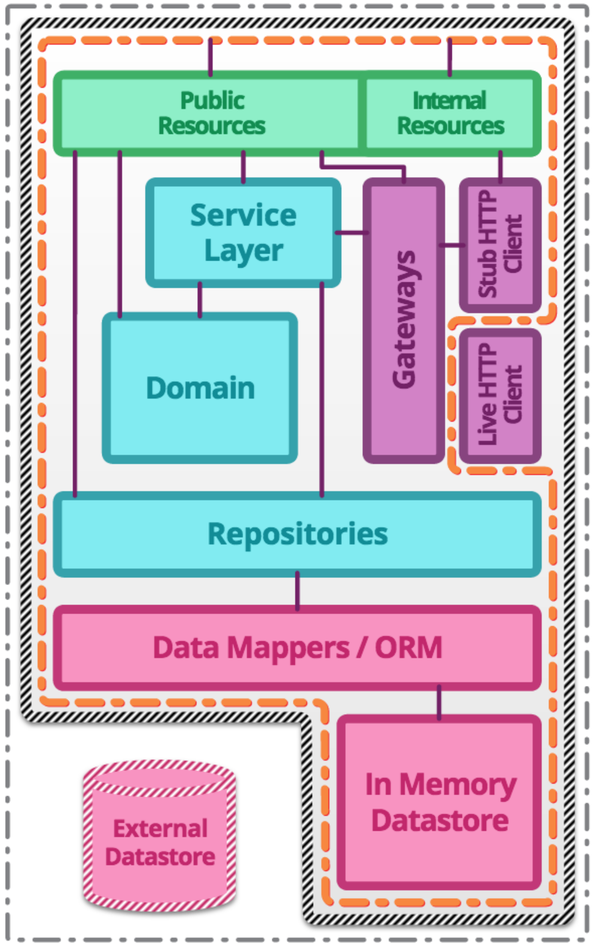


In a microservice architecture, the components are the services themselves. By writing tests at this granularity, the contract of the API is driven through tests from the perspective of a consumer. Isolation of the service is achieved by replacing external collaborators with test doubles and by using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

COMPONENT TESTING

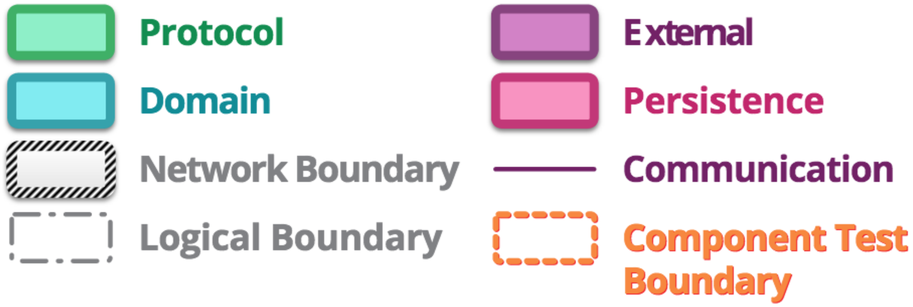
In-process component tests allow comprehensive testing... ...whilst minimising moving parts



By instantiating the full microservice in-memory using in-memory test doubles and datastores it is possible to write component tests that do not touch the network whatsoever.

This can lead to faster test execution times and minimises the number of moving parts reducing build complexity.

However, it also means that the artifact being tested has to be altered for testing purposes to allow it to start up in a 'test' mode. Dependency injection frameworks can help to achieve this by wiring the application differently based on configuration provided at start-up time.

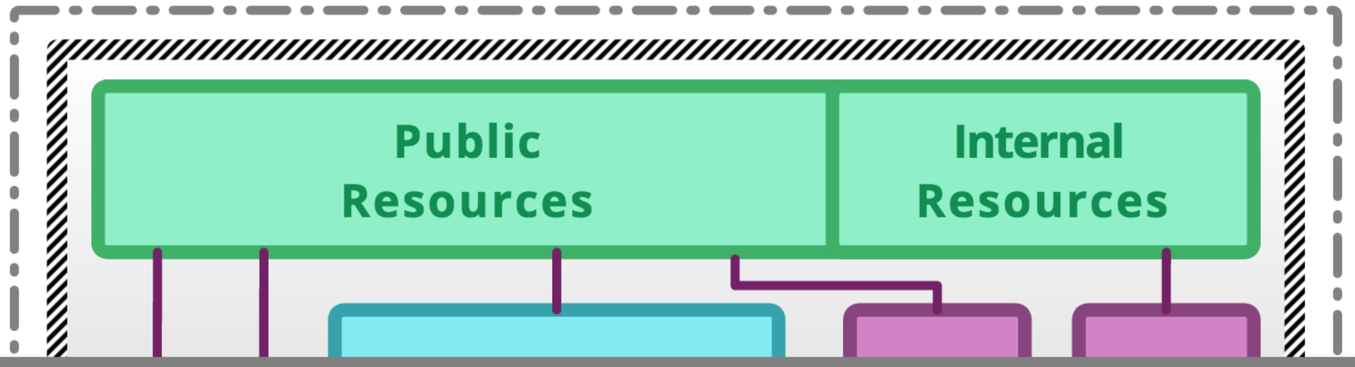


Internal resources are useful for more than just testing...

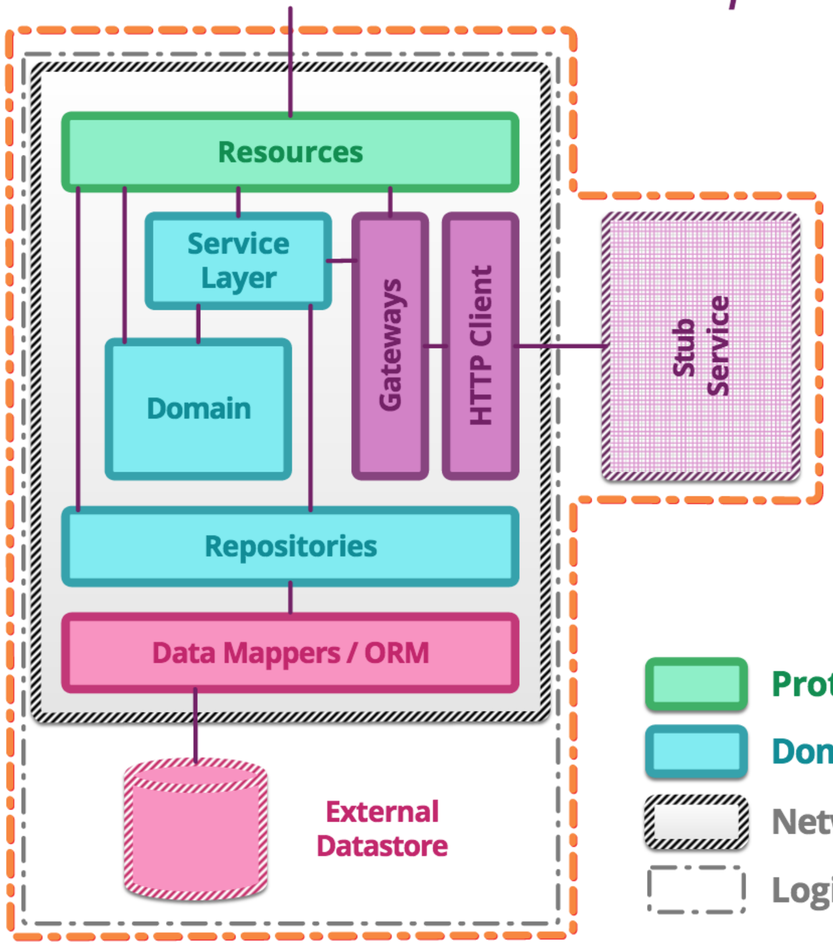
Though it may seem strange, **exposing internal controls as resources can prove useful in a number of cases besides testing such as monitoring, maintenance and debugging**. The uniformity of a RESTful API means that many tools already exist for interacting with such resources which can help reduce overall operational complexity.

The kinds of internal resources that are typically exposed include logs, feature flags, database commands and system metrics. Many microservices also include health check resources which provide information about the health of the service and its dependencies, timings for key transactions and details of configuration parameters. A simple ping resource can also be useful to aid in load balancing.

Since these resources are more privileged in terms of the control they have or the information they expose, they often require their own authentication or to be locked down at the network level. By namespacing those parts of the API that form the internal controls using URL naming conventions or by exposing those resources on a different network port, access can be restricted at the firewall level.



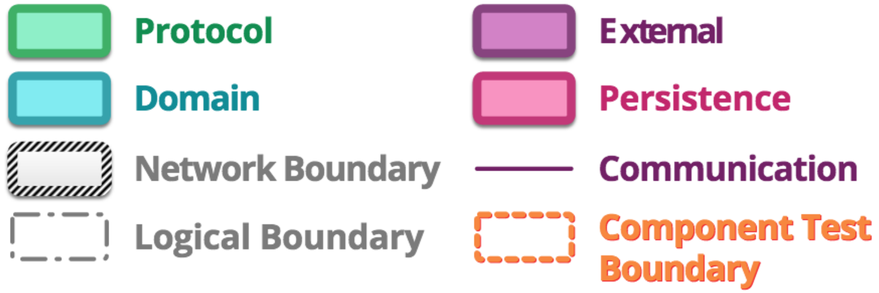
Out of process component tests exercise the fully deployed artifact... ...pushing stubbing complexity into... ...the test harness



Executing component tests against a microservice deployed as a separate process allows more layers and integration points to be exercised. Since all interactions make use of real network calls, the deployment artifact can remain unchanged with no need for any test specific logic.

With this approach, the complexity is pushed into the test harness which is responsible for starting and stopping external stubs and coordinating network ports and configuration.

As a result of the network interactions and use of a real datastore, test execution time is likely to increase. However, if the microservice has complex integration, persistence or startup logic, the out-of-process approach may be more appropriate.



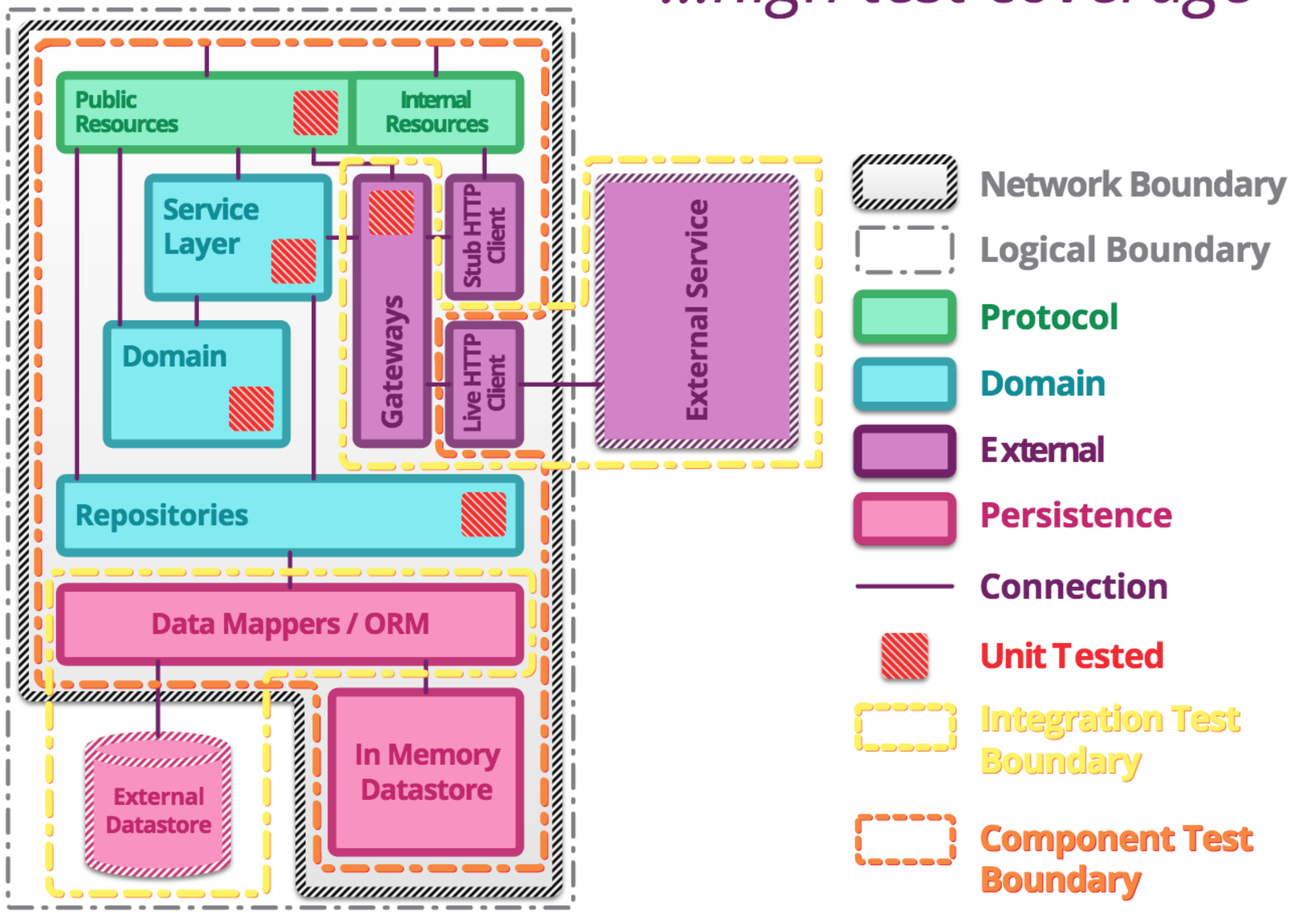
A combination of testing strategies leads to...

...high test coverage

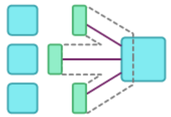
By combining unit, integration and component testing, we are able to achieve high coverage of the modules that make up a microservice and can be sure that the microservice correctly implements the required business logic.

Yet, in all but the simplest use cases, business value is not achieved unless many microservices work together to fulfil larger business processes. Within this testing scheme, there are still no tests that ensure external dependencies meet the contract expected of them or that our collection of microservices collaborate correctly to provide end-to-end business flows.

Contract testing of external dependencies and more coarse grained end-to-end testing of the whole system help to provide this.



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]



Whenever some consumer couples to the interface of a component to make use of its behaviour, a contract is formed between them. This contract consists of expectations of input and output data structures, side effects and performance and concurrency characteristics.

Each consumer of the component forms a different contract based on its requirements. If the component is subject to change over time, it is important that the contracts of each of the consumers continue to be satisfied.

Integration contract tests provide a mechanism to explicitly verify that a component meets a contract.

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.

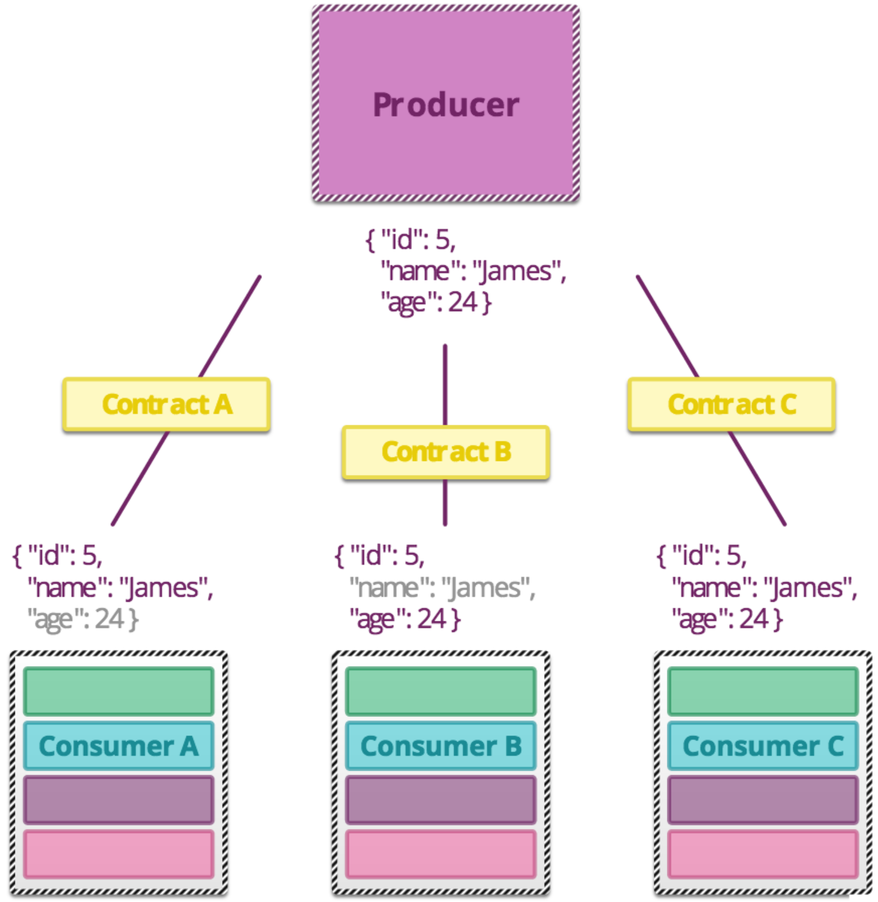
These tests are not component tests. They do not test the behaviour of the service deeply but that the inputs and outputs of service calls contain required attributes and that response latency and throughput are within acceptable limits.

Ideally, the contract test suites written by each consuming team are packaged and runnable in the build pipelines for the producing services. In this way, the maintainers of the producing service know the impact of their changes on their consumers.

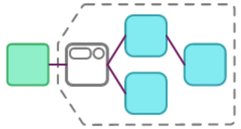
CONTRACT TESTING

The sum of all consumer contract tests...
...defines the overall service contract

Whilst contract tests provide confidence for consumers of external services, they are even more valuable to the maintainers of those services. By receiving contract test suites from all consumers of a service, it is possible to make changes to that service safe in the knowledge that consumers won't be impacted.



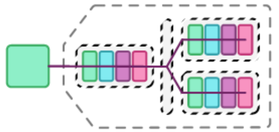
An end-to-end test verifies that a system meets external requirements and achieves its goals, testing the entire system, from end to end. [1][2]



In contrast to other types of test, the intention with end-to-end tests is to verify that the system as a whole meets business goals irrespective of the component architecture in use.

In order to achieve this, the system is treated as a black box and the tests exercise as much of the fully deployed system as possible, manipulating it through public interfaces such as GUIs and service APIs.

Since end-to-end tests are more business facing, they often utilise business readable DSLs which express test cases in the language of the domain.



As a microservice architecture includes more moving parts for the same behaviour, end-to-end tests provide value by adding coverage of the gaps between the services. This gives additional confidence in the correctness of messages passing between the services but also ensures any extra network infrastructure such as firewalls, proxies or load-balancers is correctly configured.

End-to-end tests also allow a microservice architecture to evolve over time. As more is learnt about the problem domain, services are likely to split or merge and end-to-end tests give confidence that the business functions provided by the system remain intact during such large scale architectural refactorings.

END-TO-END TESTING

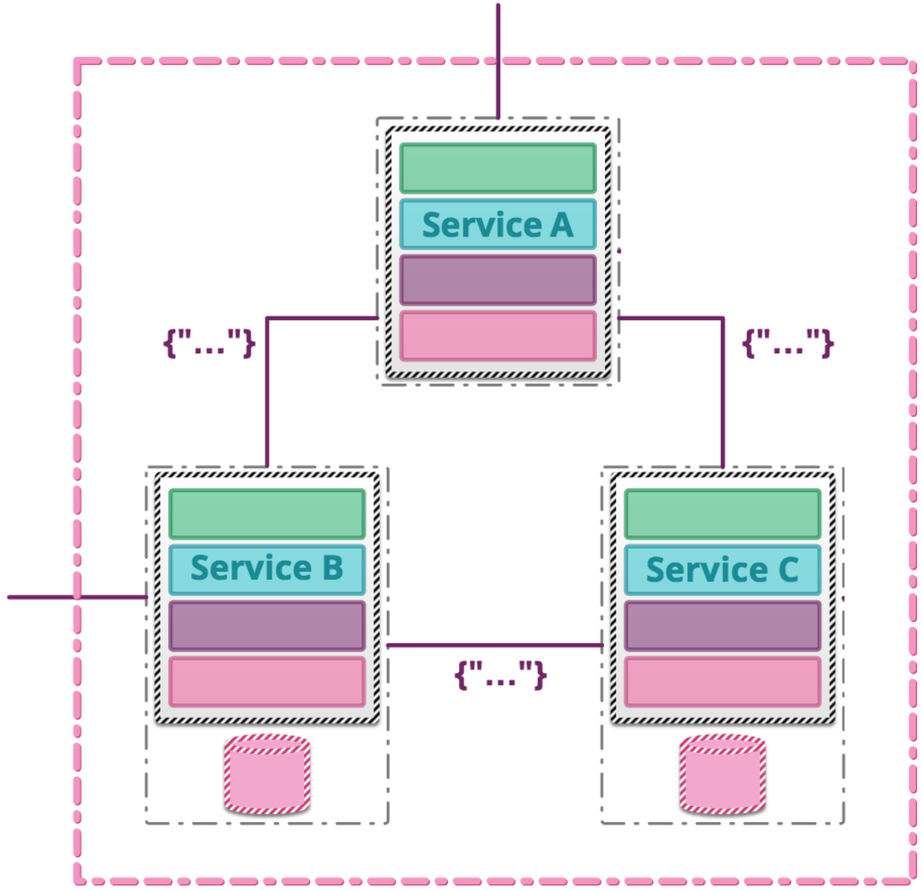
The test boundary for end-to-end tests...
...is much larger than for other types of test

Since the goal is to test the behaviour of the fully integrated system, end-to-end tests interact at the most coarse granularity possible.

If the system requires direct user manipulation, this interaction may be through GUIs exposed by one or more of the microservices. In this case, tools such as Selenium WebDriver can help to drive the GUI to trigger particular use cases within the system.

For headless systems, end-to-end tests directly manipulate the microservices through their public APIs using an HTTP client.

In this way, the correctness of the system is ascertained by observing state changes or events at the perimeter formed by the test boundary.



Writing and maintaining end-to-end tests can be very difficult

Since end-to-end tests involve many more moving parts than the other strategies discussed so far, they in turn have more reasons to fail. End-to-end tests may also have to account for asynchrony in the system, whether in the GUI or due to asynchronous backend processes between the services. These factors can result in flakiness, excessive test runtime and additional cost of maintenance of the test suite.

The following guidelines can help to manage the additional complexity of end-to-end tests:

① Write as few end-to-end tests as possible

② Focus on personas and user journeys

③ Choose your ends wisely

④ Rely on infrastructure-as-code for repeatability

⑤ Make tests data-independent

Due to the difficulty inherent in writing tests in this style, some teams opt to avoid end-to-end testing completely, in favour of thorough production monitoring and testing directly against the production environment.

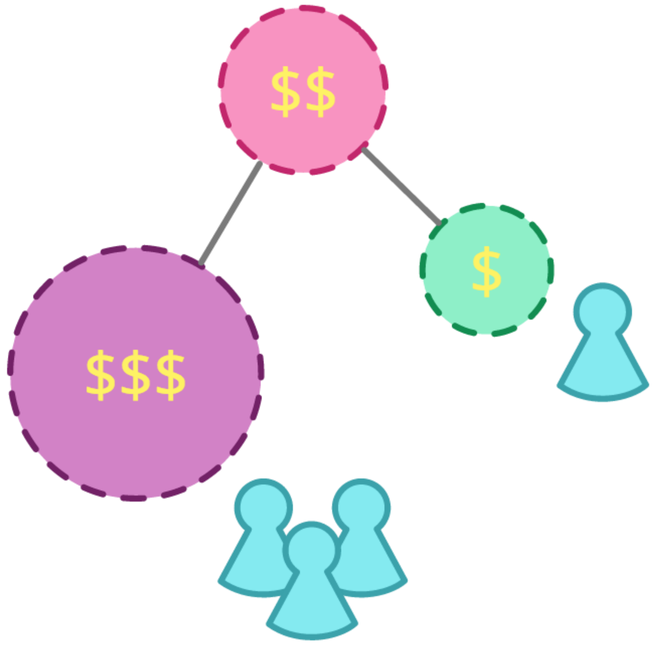
Synthetic transactions – fake users exercising real transactions against the production system – can supplement typical monitoring techniques to provide more insight into production health. Additionally, alerting when key business metrics fall outside of acceptable norms can help to identify production issues fast.

Microservice architectures provide more options for where and how to test.

By breaking a system up into small well defined services, additional boundaries are exposed that were previously hidden. These boundaries provide opportunities and flexibility in terms of the type and level of testing that can be employed.

In some cases, a microservice may encapsulate a central business process with complex requirements. The criticality of this process may necessitate very comprehensive testing of the service such as the full range of test strategies discussed here. In other cases, a microservice may be experimental, less crucial from a business standpoint or may have a short lifespan. The level of testing required may be lower such that only a couple of the strategies make sense.

While this decision making process is still possible in a monolithic architecture, the addition of clear, well defined boundaries makes it easier to see the components of your system and treat them in isolation.



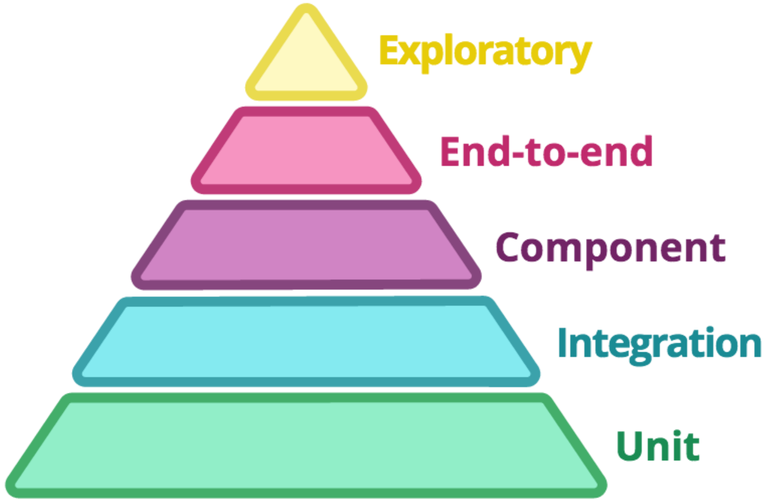
The test pyramid helps us to maintain a balance between the different types of test

In general, the more coarse grained a test is, the more brittle, time consuming to execute and difficult to write and maintain it becomes. This additional expense stems from the fact that such tests naturally involve more moving parts than more fine grained focussed ones.

The concept of the test pyramid is a simple way to think about the relative number of tests that should be written at each granularity. Moving up through the tiers of the pyramid, the scope of the tests increases and the number of tests that should be written decreases.

At the top of the pyramid sits exploratory testing, manually exploring the system in ways that haven't been considered as part of the scripted tests. Exploratory testing allows the team to learn about the system and to educate and improve their automated tests.

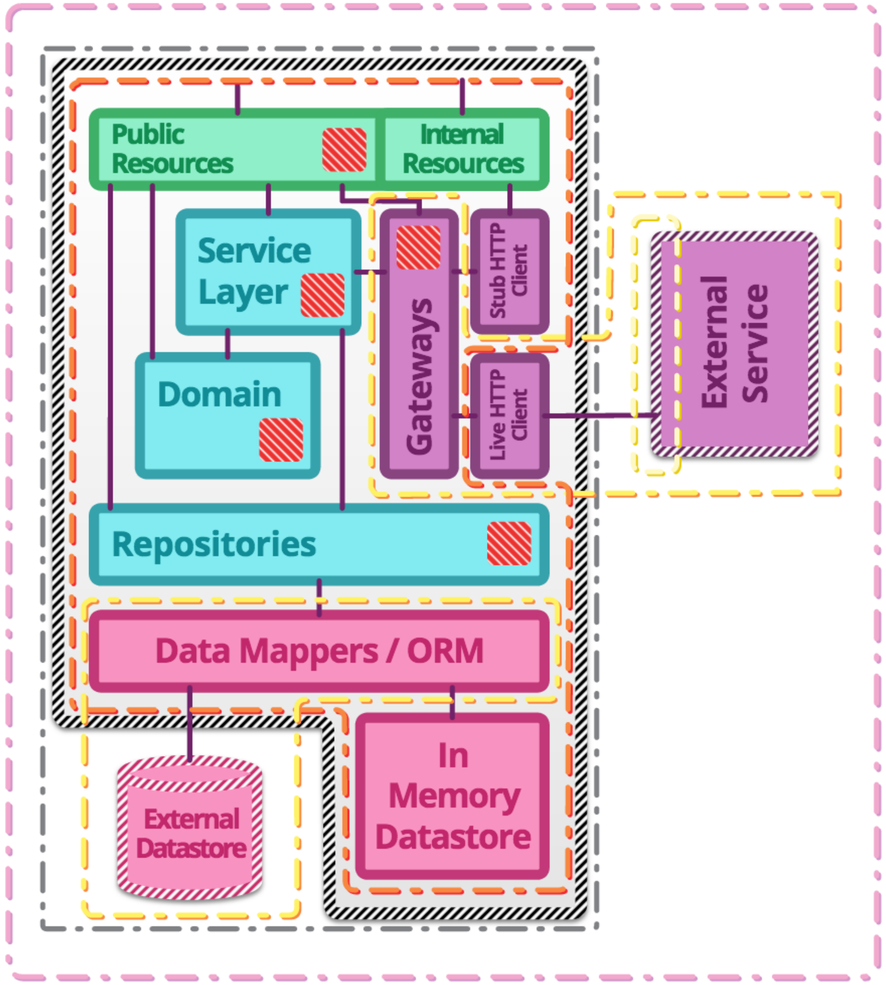
By following the guidelines of the test pyramid, we can avoid decreasing the value of the tests through large boated test suites that are expensive to maintain and execute.



In summary...

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

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



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.

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

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