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.

**Definition**

**What is a microservice**

**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?"](http://bovon.org/2013/07/09/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](https://martinfowler.com/articles/richardsonMaturityModel.html)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.

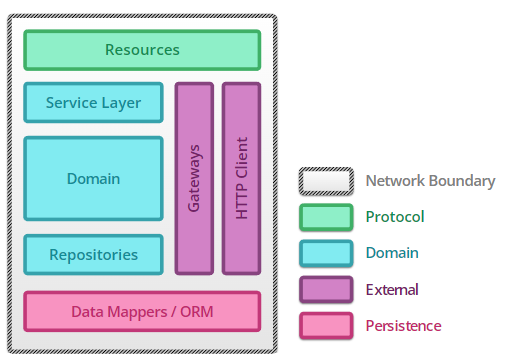
Each microservice may or may not provide some form of user interface.

**Anatomy**

**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.



* **Resources** act as mappers between the application protocol exposed by the service and messages to objects representing the domain. Typically, they are thin, with responsibility for sanity checking the request and providing a protocol specific response according to the outcome of the business transaction.
* **Service Layer / Domain/ Repositories** - Almost all of the service logic resides in a domain model representing the business domain. Of these objects, services coordinate across multiple domain activities, whilst repositories act on collections of domain entities and are often persistence backed.
* **Gateways / HTTP Client** - If one service has another service as a collaborator, some logic is needed to communicate with the external service. A gateway encapsulates message passing with a remote service, marshalling requests and responses from and to domain objects. It will likely use a client that understands the underlying protocol to handle the request-response cycle.
* **Data Mappers / ORM** - Except in the most trivial cases or when a service acts as an aggregator across resources owned by other services, a micro-service will need to be able to persist objects from the domain between requests. Usually this is achieved using object relation mapping or more lightweight data mappers depending on the complexity of the persistence requirements.

Often, this logic is encapsulated in a set of dedicated objects utilised by repositories from the domain.

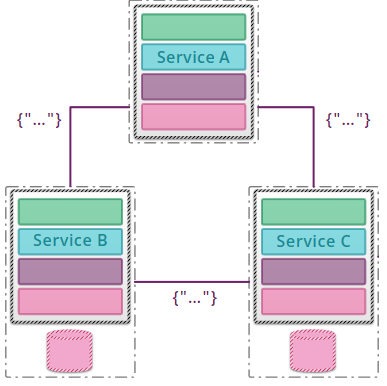
**Architecture**

**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](https://martinfowler.com/bliki/CircuitBreaker.html) and bulkheads can help to maintain overall system uptime in spite of a component outage.



**Testing Strategies**

**Unit: mocklist vs classic**

A unit test exercises the smallest piece of testable software in the application to determine whether it behaves as expected

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](https://leanpub.com/wewut) 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.



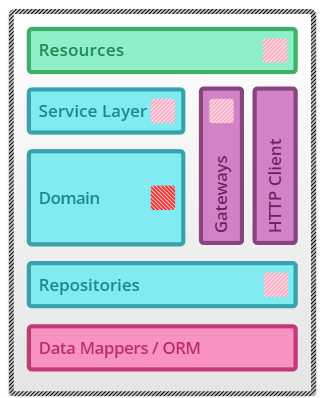
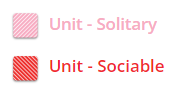
[Solitary unit testing](https://leanpub.com/wewut) looks at the interactions and collaborations between an object and its dependencies, which are replaced by [test doubles](https://martinfowler.com/bliki/TestDouble.html).



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

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

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

* **Domain** - Domain logic often manifests as complex calculations and a collection of state transitions. Since these types of logic are highly state-based there is little value in trying to isolate the units. This means that as far as possible, real domain objects should be used for all collaborators of the unit under test.
* **Gateways / Repositories** - With plumbing code, it is difficult to both isolate the unit under test from external modules and test against state changes. As such, using test doubles is more effective.

The purpose of unit tests at this level is to verify any logic used to produce requests or map responses from external dependencies rather than to verify communication in an integrated way. As such, using test doubles for the collaborators provides a way to control the request-response cycle in a reliable and repeatable manner.

Unit tests at this level provide faster feedback than integration tests and can force error conditions by having doubles respond as an external dependency would in exceptional circumstances.

* **Resources / Service Layer** - Coordination logic cares more about the messages passed between modules than any complex logic within those modules. Using test doubles allows the details of the messages passed to be verified and responses stubbed such that flow of communications within the module can be specified from the test.

If a piece of coordination logic requires too many doubles, it is usually a good indicator that some concept should be extracted and tested in isolation.

As the size of a service decreases the ratio of plumbing and coordination logic to complex domain logic increases. Similarly, some services will contain entirely plumbing and coordination logic such as adapters to different technologies or aggregators over other services.

In such cases comprehensive unit testing may not pay off. Other levels of testing such as [component testing](https://martinfowler.com/articles/microservice-testing/#testing-component-introduction) can provide more value.

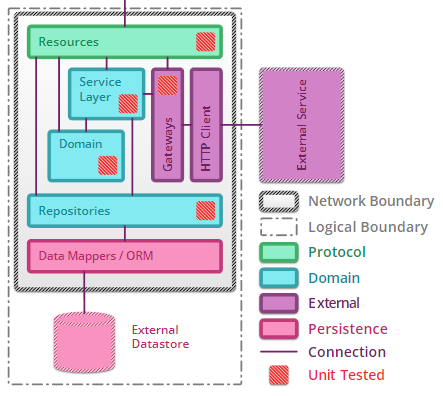
The intention of unit tests and testing in general is to constrain the behaviour of the unit under test. An unfortunate side effect is that sometimes, tests also constrain the implementation. This often manifests with over reliance on mock based approaches.

It is important to constantly question the value a unit test provides versus the cost it has in maintenance or the amount it constrains your implementation. By doing this, it is possible to keep the test suite small, focussed and high value.

**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.



**Integration: datastores and external services**

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

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.

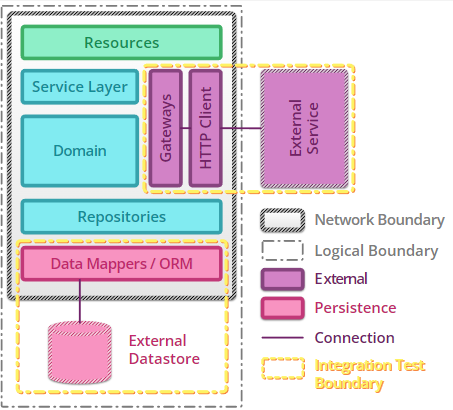
Whilst tests that integrate components or modules can be written at any granularity, in microservice architectures they are typically used to verify interactions between layers of integration code and the external components to which they are integrating.



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

**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.



* **Gateways / HTTP Client / External Service** - ***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.

State management can be difficult when testing against external components since the tests will rely on certain data being available. One way to mitigate this problem is to agree on a fixed set of representative but harmless data that is guaranteed to be available in every environment.

* **Data Mappers ORM / External Datastore** - ***Persistence integration tests*** provide assurances that the schema assumed by the code matches that available in the data store.

In the case that an ORM is in use, these tests also give confidence that any mappings configured in the tool are compatible with the result sets coming back.

Modern ORMs are very sophisticated in terms of caching and only flushing when necessary. It is important to structure tests such that transactions close in between preconditions, actions and assertions to be sure data makes a full round trip.

Since most data stores exist across a network partition, they are also subject to timeouts and network failures. Integration tests should attempt to verify that the integration modules handle these failures gracefully.

Tests in this style provide fast feedback when refactoring or extending the logic contained within integration modules. However they also have more than one reason to fail — if the logic in the integration module regresses *or* if the external component becomes unavailable or breaks its contract.

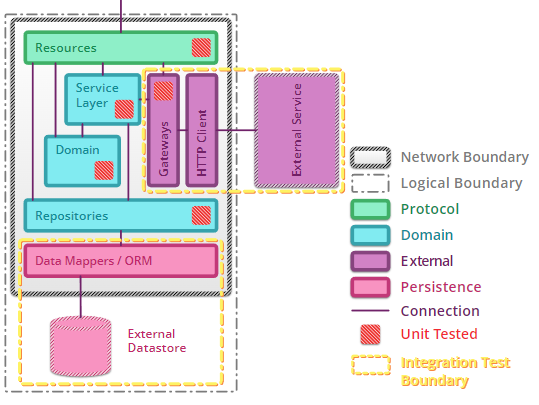
To mitigate this problem, write only a handful of integration tests to provide fast feedback when needed and provide additional coverage with unit tests and [contract tests](https://martinfowler.com/articles/microservice-testing/#testing-contract-introduction) to comprehensively validate each side of the integration boundary. It may also make sense to separate integration tests in the CI build pipeline so that external outages don't block developmen

**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](https://martinfowler.com/articles/microservice-testing/#testing-end-to-end-introduction), more accurate test feedback and smaller test runtimes can be obtained by testing the microservice isolated from its external dependencies.



**Component: in or out of process?**

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

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](https://martinfowler.com/bliki/BroadStackTest.html)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



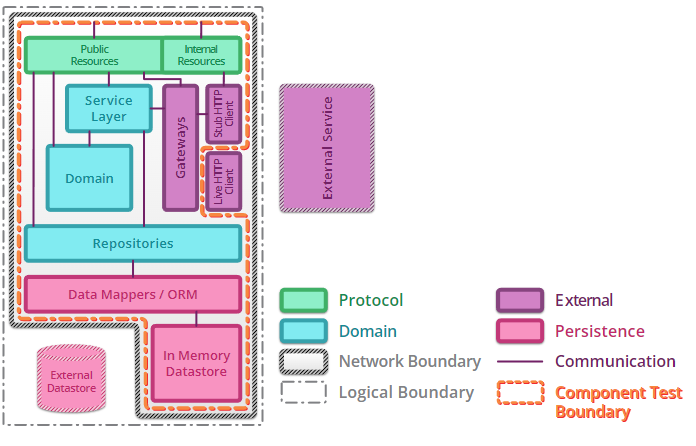
The implementation of such tests includes a number of options. Should the test execute in the same process as the service or out of process over the network? Should test doubles lie inside the service or externally, reached over the network? Should a real datastore be used or replaced with an in-memory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

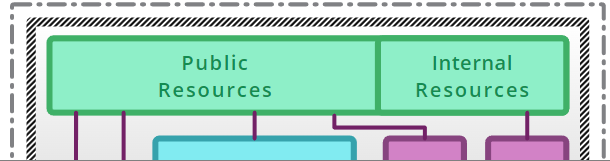
In-process component tests allow comprehensive testing whilst minimising moving parts

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.

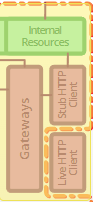
However, it also means that the artifact being tested has to be altered for testing purposes to allow it to start up in a 'test' mode. Dependency injection frameworks can help to achieve this by wiring the application differently based on configuration provided at start-up time



* 

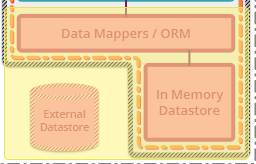
Tests communicate with the microservice through an **internal interface** allowing requests to be dispatched and responses to be retrieved. Often a custom [shim](http://en.wikipedia.org/wiki/Shim_(computing)) is used to achieve this although a number of pre-built libraries exist such as [inproctester](https://github.com/aharin/inproctester" \t "_blank) for JVM based microservices and [plasma](https://github.com/jennifersmith/plasma) for .NET based microservices.

In this way, an in-process component test can get as close as possible to executing real HTTP requests against the service without incurring the additional overhead of real network interactions.

* 

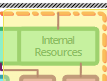
In order to isolate the microservice from external services, gateways can be configured to use test doubles instead of real protocol level clients. Using [internal resources](https://martinfowler.com/articles/microservice-testing/#testing-component-internal-resources), these test doubles can be programmed to return predefined responses when certain requests are matched.

These test doubles can also be used to emulate unhappy paths through the component such as when external collaborators are offline or are responding slowly or with malformed responses. This allows error conditions to be tested in a controlled and repeatable manner.

* 

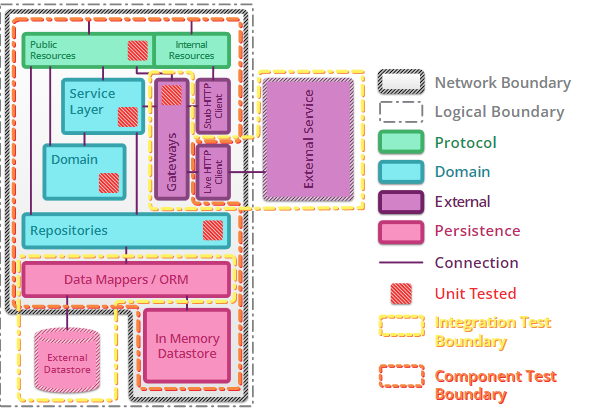
Replacing an external datastore with an in-memory implementation can provide significant test performance improvements. Whilst this excludes the real datastore from the test boundary, any persistence integration tests will provide sufficient coverage.

In some cases, the persistence mechanism employed is simple enough that a lightweight custom implementation can be used. Alternatively, some datastores such as cassandra and elasticsearch provide embedded implementations. There are also tools available that emulate external datastores in-memory, such as the H2 database engine.

* 

Whilst it is possible to configure test doubles and setup data directly when writing in-process acceptance tests, routing all requests through privileged internal resources allows the service to be tested as more of a black box. This allows changes in persistence technology or external service communications to take place without impacting the component test suite.

**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](https://martinfowler.com/articles/microservice-testing/#testing-contract-introduction) of external dependencies and more coarse grained [end-to-end testing](https://martinfowler.com/articles/microservice-testing/#testing-end-to-end-introduction) of the whole system help to provide this

**Contract: ensuring consistency across boundaries**

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

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.

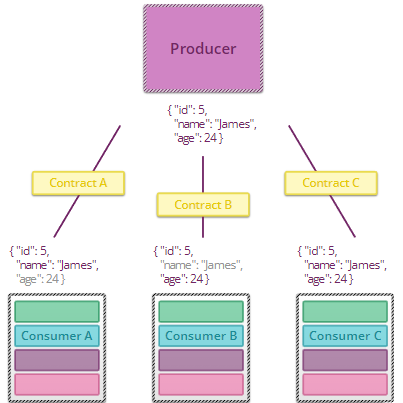
When the components involved are microservices, the interface is the public API exposed by each service. The maintainers of each consuming service write an independent test suite that verifies only those aspects of the producing service that are in use.



These tests are not [component tests](https://martinfowler.com/articles/microservice-testing/#testing-component-introduction). 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.

**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.

* Consider a service that exposes a resource with three fields, an identifier, a name and an age. This service has been adopted by three different consumers who have each coupled to different parts of the resource.

**Consumer A** couples to only the identifier and name fields. As such, the corresponding contract test suite asserts that the resource response contains those fields. It does not make any assertion regarding the age field.

**Consumer B** couples to the identifier and age fields so the contract tests assert they are present but make no assertions about the name field.

**Consumer C** requires all three fields and has a contract test suite that asserts they are all present.

If a new consumer adopts the API but requires both first name and last name, the maintainers may choose to deprecate the name field and introduce another field containing a composite object with the name components.

In order see what would be needed to remove the old name field, the maintainers could delete it from the response and see which contract tests fail. In this case, they would see that consumers A and C would need to be notified of the deprecation. After consumers A and C have migrated to the new field, the deprecated field can be removed.

For this to be effective, both producers and consumers should follow [Postel's Law](https://en.wikipedia.org/wiki/Robustness_principle" \t "_blank) when serializing and deserializing messages ignoring any fields that are not important to them.

This approach is an example of a [Parallel Change](https://martinfowler.com/bliki/ParallelChange.html) wherein the API can be changed over a period of time without breaking the contract of any consumers.

Contract test suites are also valuable when a new service is being defined. Consumers can drive the API design by building a suite of tests that express what they need from the service.

These [consumer driven contracts](https://martinfowler.com/articles/consumerDrivenContracts.html) form a discussion point with the team responsible for building the service as well as being automated tests which give an indication of readiness of the API.

A number of tools exist to aid in writing contract tests such as [Pact](https://github.com/realestate-com-au/pact), [Pacto](https://github.com/thoughtworks/pacto" \t "_blank) and [Janus](https://github.com/gga/janus).

**End to End: tips and tricks**

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

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](https://martinfowler.com/bliki/BusinessFacingTest.html), they often utilise business readable [DSLs](https://martinfowler.com/bliki/DomainSpecificLanguage.html) 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.

**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](http://docs.seleniumhq.org/projects/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.

Whilst some systems are small enough that a single team has ownership of all of the composite components, in many cases, systems grow to have dependencies on one or more externally managed microservices.

Usually, these external services are included within the end-to-end test boundary. However in rare cases, you may choose to exclude them.

If an external service is managed by a third party, it may not be possible to write end-to-end tests in a repeatable and side effect free manner. Similarly, some services may suffer from reliability problems that cause end-to-end tests to fail for reasons outside of the team's control.

In cases such as these, it can be beneficial to stub the external services, losing some end-to-end confidence but gaining stability in the test suite.

**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:

* **Write as few end-to-end tests as possible** (Given that a high level of confidence can be achieved through lower levels of testing, the role of end-to-end tests is to make sure everything ties together and there are no high level disagreements between the microservices. As such, comprehensively testing business requirements at this level is wasteful, especially given the expense of end-to-end tests in time and maintenance.)

One strategy that works well in keeping an end-to-end test suite small is to apply a time budget, an amount of time the team is happy to wait for the test suite to run. As the suite grows, if the run time begins to exceed the time budget, the least valuable tests are deleted to keep within the allotted time. The time budget should be of the order of minutes, not hours.

* **Focus on personas and user journeys**. To ensure all tests in an end-to-end suite are valuable, model them around personas of users of the system and the journeys those users make through the system. This provides confidence in the parts of the system the users value the most and leaves coverage of anything else to other types of testing.

Tools, such as Gauge and Concordion, exist to help in expressing journeys via business readable DSLs.

* **Choose your ends wisely**. If a particular external service or GUI is a major cause of flakiness in the test suite, it can help to redefine the test boundary to exclude the component. In this case, total end-to-end coverage is traded in favour of reliability in the suite. This is acceptable as long as other forms of testing verify the flaky component using different means.
* **Rely on infrastructure-as-code for repeatability**. If you have embraced infrastructure-as-code, which can help greatly in managing the additional deployment complexity of a microservice architecture, it is possible to build environments on the fly in a reproducible manner.

By building a fresh environment for every end-to-end test suite execution, reliability can be improved whilst also acting as a test of the deployment logic.

* **Make tests data-independent**. A common source of difficulty in end-to-end testing is data management. Relying on pre-existing data introduces the possibility of failure as data is changed and accumulated in the environment. I call these false-failures, in that the failure isn’t an indication of a fault in the software. Automated management of the data relied upon by end-to-end tests reduces the chances of false-failures. If services support the construction of the entities they own, via public or internal APIs, end-to-end tests can define their world before execution. For those services that do not allow external construction, canned data can be imported at the database level.

**Conclusion**

**Options: testing benefits of microservices**

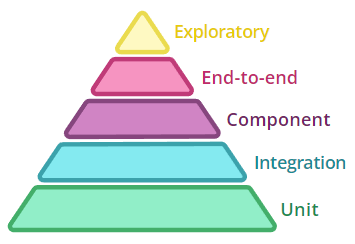
Microservice architectures provide more options for where and how to test.

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.

**Test Pyramid: how many tests**



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](https://martinfowler.com/bliki/TestPyramid.html) 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.

