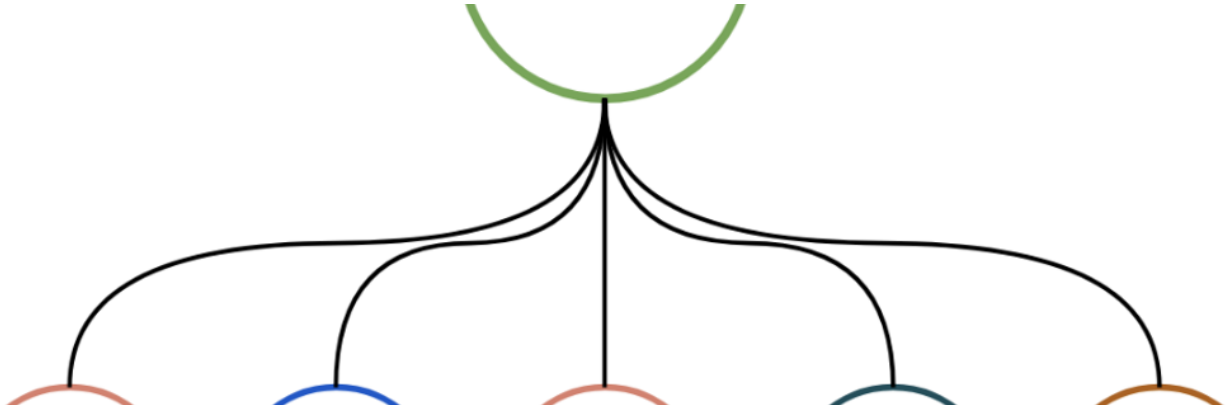
Hemanth Kandukuri FOLLOW
Lead SDET (API & UI Automation, Performance Testing, Automation Framework Develo...

Test Automation Strategy and Philosophy for Micro Services

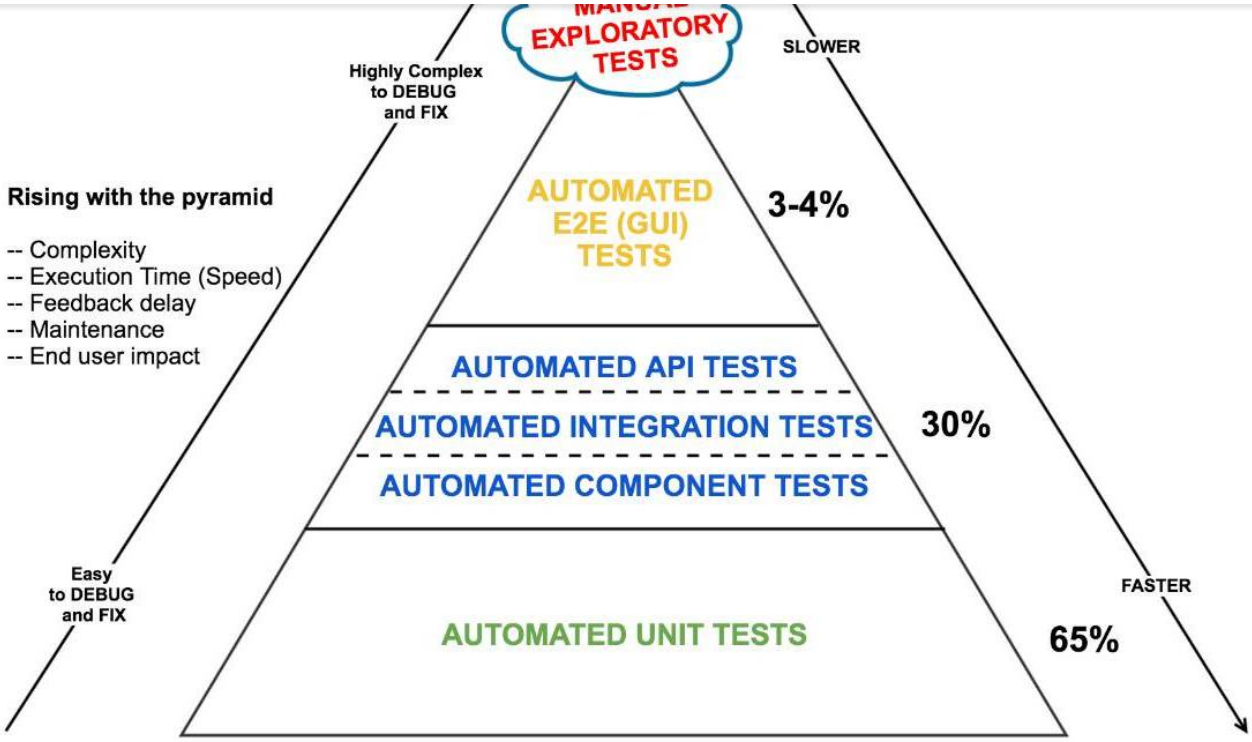
Published Jun 10, 2019



Motivation - Testing of Microservices

Testing of microservices is the first step in making the services reliable for the user or edge devices/clients. For the internal functional dependency of the microservices, the testing is more important for the service to stay strong and holds together for all the assumptions made on them while building.

At a high level, the testing cycle should be quick and give faster feedback to the developers, so that the fixes can be put in or plan other mitigations quickly, such that the overhead of fixing after integrating or losing customers, can be avoided, following the Test Pyramid strategy. 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.



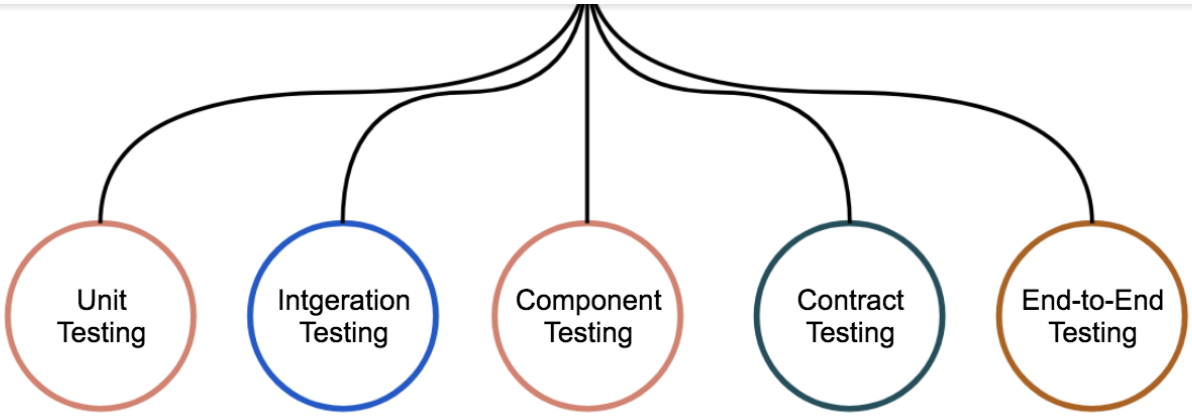
By following the guidelines of the test pyramid, we can avoid decreasing the value of the tests through large test suites that are expensive to maintain and execute.

Microservices Testing - Strategy & Approach

Microservices are to be built with an approach and understanding of the microservice and how it can be tested in different states (isolated, integrated, end-to-end). And also having good code coverage for each microservice helps in improving the quality of the service itself and when services like these are integrated the whole system's quality can be assured with confidence.

Along with these, the deployment of such services should be made faster with quick feedback cycles from the test teams with the help of build pipelines for continues integrations and delivery.

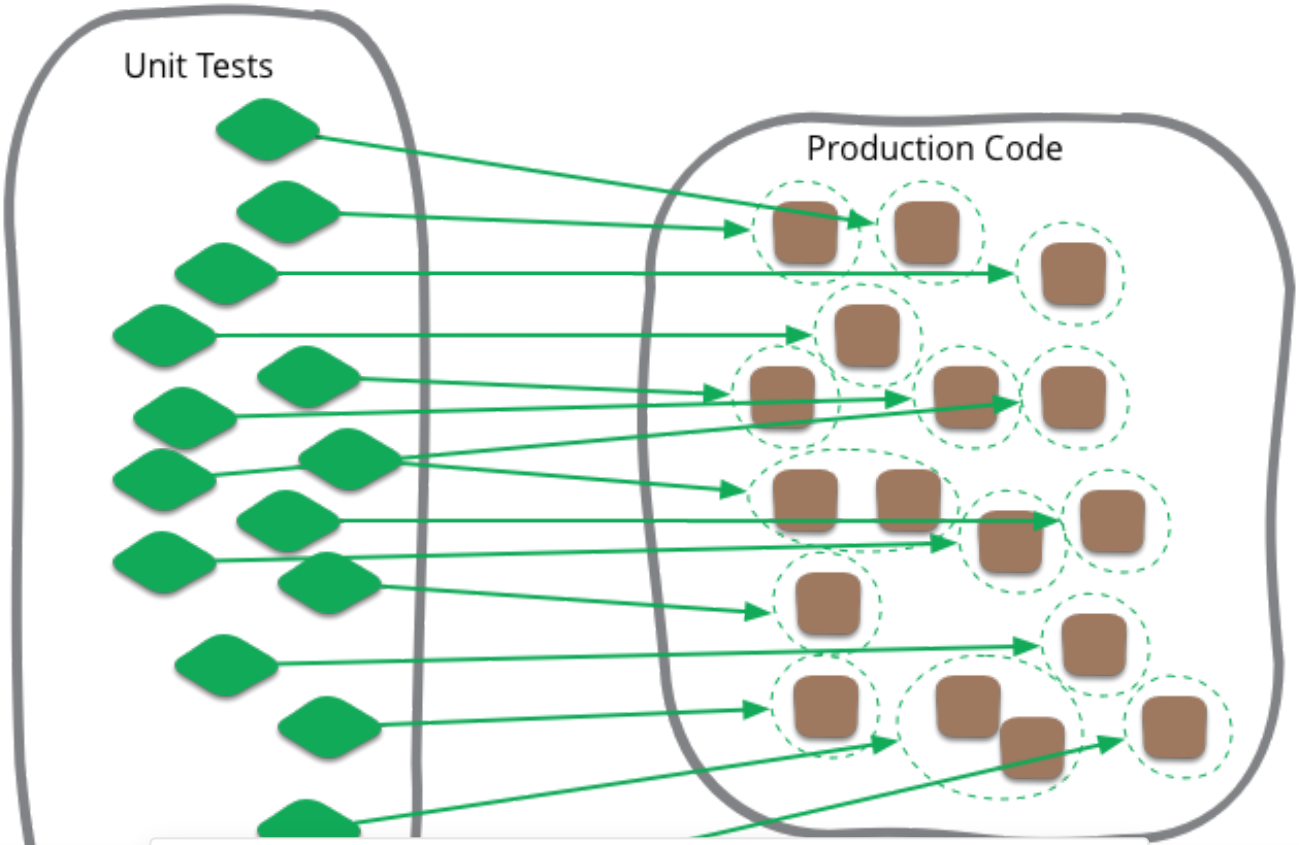
In this approach, automation testing can be divided into different layers of testing.



Unit Testing

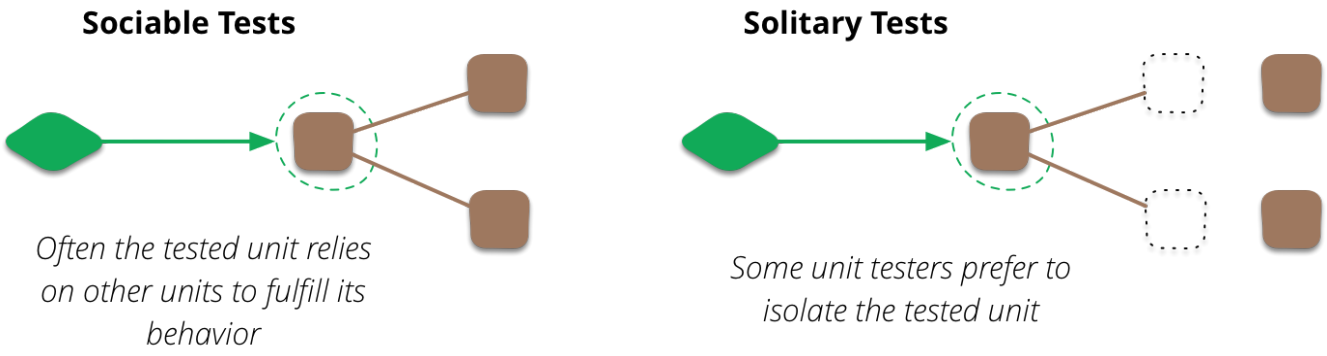
Ownership: Developers

Unit Tests, test a single class or a set of closely coupled classes. These unit tests can either be run using the actual objects that the unit interacts with or by employing the use of test doubles or mocks. A unit can be a class (or) a method under test and also we can consider the notion of a bunch of closely related classes and treat them as a single unit in some scenarios as well.

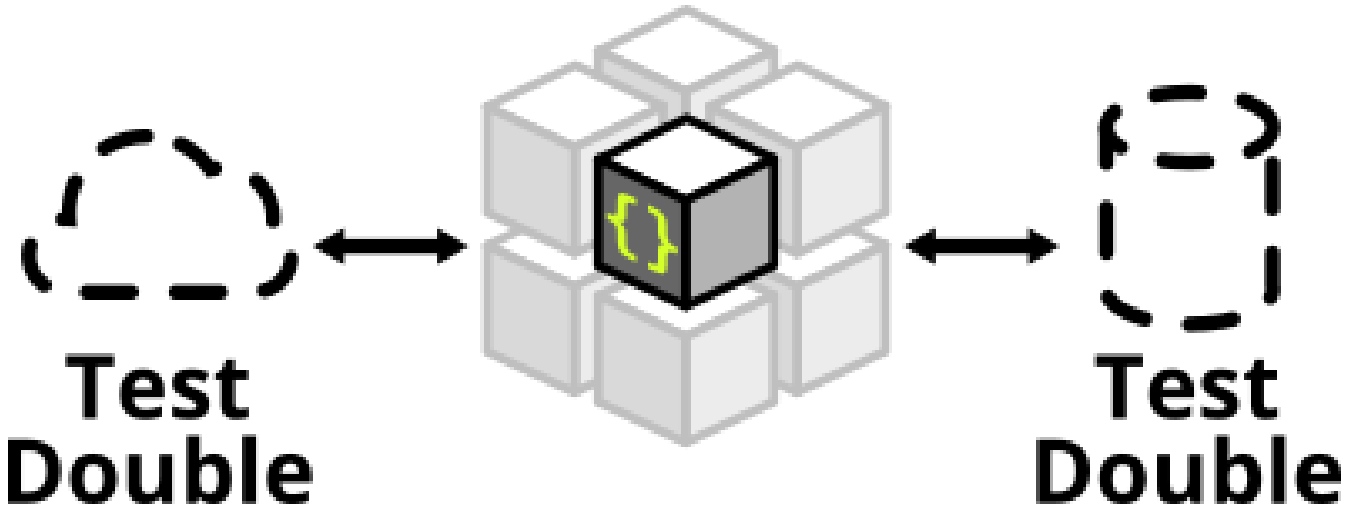


Solitary or Sociable?

When grouping similar classes to test behaviors expected from the system, they are considered to be Sociable Tests, while the similar classes are replaced with mocks and the actual class is isolated these unit tests are considered Solitary Tests.



Unit tests are usually written by the programmers using their regular tools, like JUnit (as Test Framework) coupled with Mockito (or) WireMock for mocking. Mocks or Stubs are referred to as Test Doubles, wherein you replace an entire or parts of a service (or dependent system) which are expected to be up and responding for current service under test to function properly. An ideal way is to, stub out external collaborators, set up some input data, call your subject under test and check that the returned value is what you expected. Using test doubles is not specific to unit testing. More elaborate test doubles can be used to simulate entire parts of our system in a controlled way. However, in unit testing, we're most likely to encounter a lot of mocks and stubs (depending on if the tests are the sociable or solitary kind.)



Speed is a ... e of

helps in identifying the defects occurred with the last change and also need not have to look far for fixing it.

As defined in agile practices, having a compile and commit suites of unit tests is very helpful for pushing quality code to repositories. Compile suite is the one to be executed whenever you compile the code and want to check if it is intact even after changes are made. While the Commit Suite is used when pushing the code to version control with shared code bases and commits across developers.

Test Structure
An ideal structure for Unit Testing would be **("Arrange, Act, Assert")**,

Set up the test data
Call the class/method under test with multiple times with different set(s) of test data
Assert that the expected results are returned

Similarly, a BDD approach is, the **"given", "when", "then"** triad, where given reflects the setup when the method call and then the assertion part.

Integration Testing

Ownership: Developers and Testers

Integration testing is a way of testing software by grouping software components together. In a complex software system, there are numerous interconnections applicable to a particular feature, which makes writing integration tests difficult.

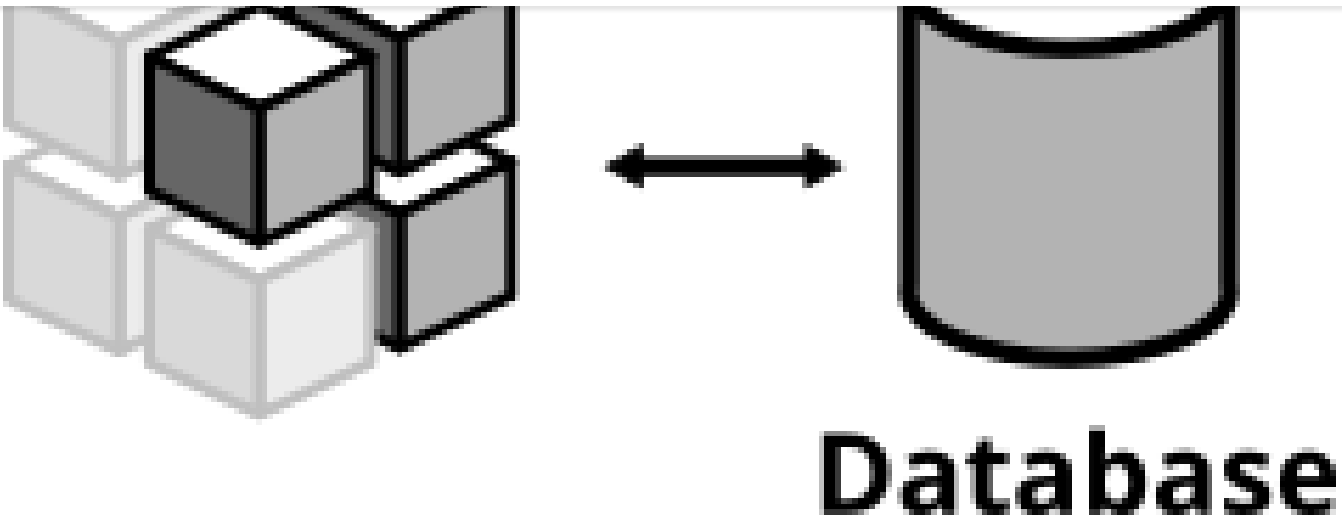
Here, all test modules are integrated together and tested as a subsystem. It checks that the communication paths between the subsystem work correctly while interacting with its peers. In a microservices architecture, they are typically used to verify interactions between layers of integration code and the external components to which they are integrating.

Components of Integration Testing

1. Developing features and writing a test case.
   Engineers start with writing code for features and their test cases. Once development is done, these features will be pushed to remote branches that would trigger the running of integration tests.
2. Creating a pristine integration testing environment
   A dedicated rather than shared environment is needed since it gives testing good resource management, control and logging for the diagnosis and debugging purposes. Hence, each integration test run should start with a cleared environment.
3. Scheduling and running the tests
   For each pull request of a new business feature, there should be a scheduled integration test build and run to validate that feature.
4. Reporting the result of integration tests
   Finally, reporting the test result alerts team members of the status of feature development, and allows them to react quickly if there is an issue. This means that once all test cases have finished running, there should be a reporting procedure that would send a report with regards to the integration test results.

When the automated tests are written for the modules which are interacting with an external component, the basic goal is to verify the modules are interacting sufficiently with the external component. With unit testing and integration testing, we can have confidence in the correctness of the logic contained in the individual modules that make up the microservice.

Conceptually Integration Tests are always about triggering an action that leads to integrating with the outside part (filesystem, database, separate service).
A database integration test would look like this: (A database integration test integrates your code with a real database)



1. start a database
2. connect your application to the database
3. trigger a function within your code that writes data to the database

4. check that the expected data has been written to the database by reading the data from the database.
   Another example, testing that your service integrates with a separate service via a REST API could look like this: (This kind of integration test checks that your application can communicate with a separate service correctly)

3. trigger a function within your code that reads from the separate service's API
4. check that your application can parse the response correctly

Integration tests - like unit tests - can be whitebox testing. Using Mockito or WireMock frameworks we can start our application while still being able to mock some other parts of our application so that we can check that the correct interactions have happened.

We should write integration tests for all pieces of code where we either serialize or deserialize data. For example:

- Calls to our services' REST API
- Reading from and writing to databases
- Calling other application's APIs
- Reading from and writing to queues
- Writing to the filesystem
- Reading from and writing to data streams
- Actions upon WebSocket messages

Writing integration tests around these boundaries ensures that writing data to and reading data from these external collaborators/services works correctly as expected.

Contract Testing

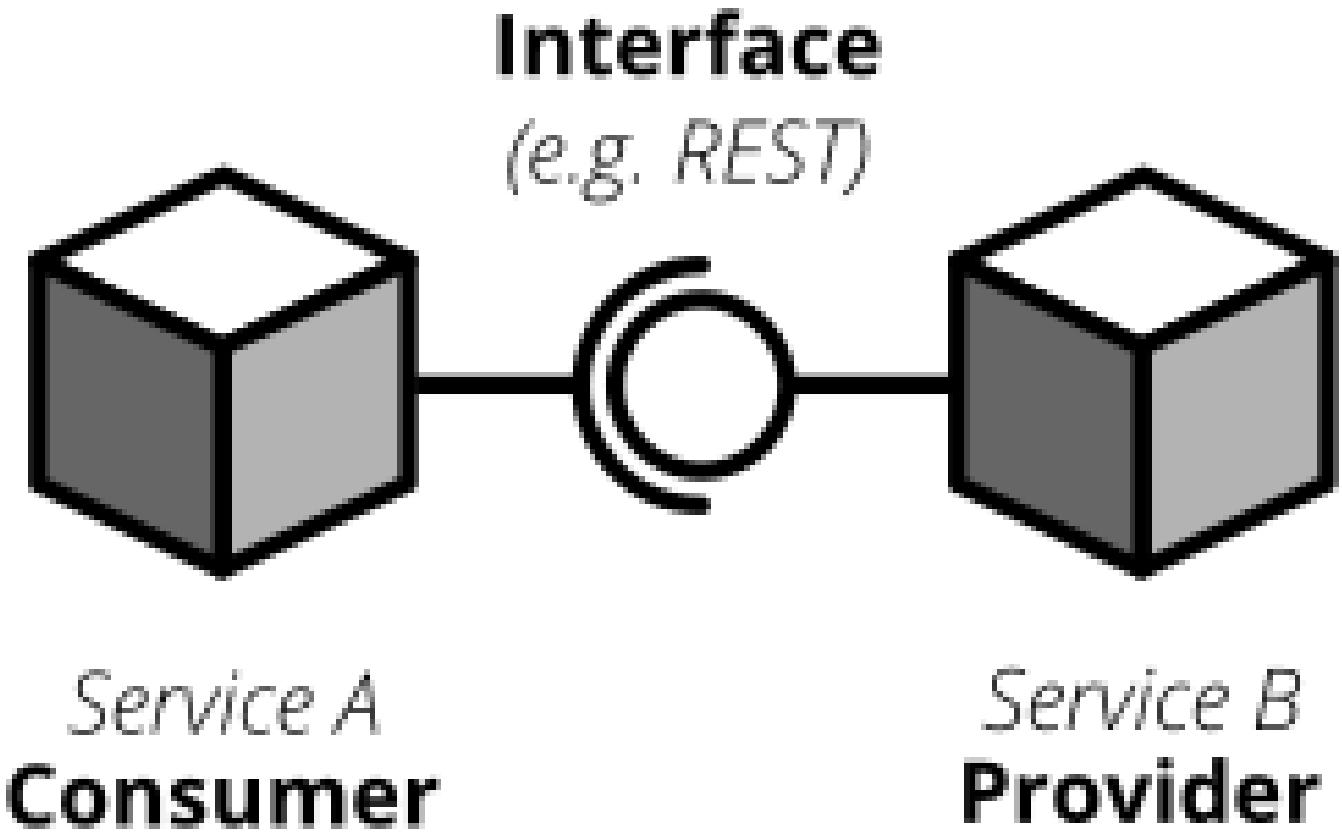
Since the development efforts are spreading the development of a system across different teams. Thus the individual teams build individual, loosely coupled services without stepping on each other's and integrate these services into a big, cohesive system. Contract tests around microservices focus on exactly that.

Splitting the system into many small services often means that these individual services need to communicate with each other via certain well-defined (or dynamic), interfaces. Interfaces between different applications can come in different shapes and technologies. Common ones are:

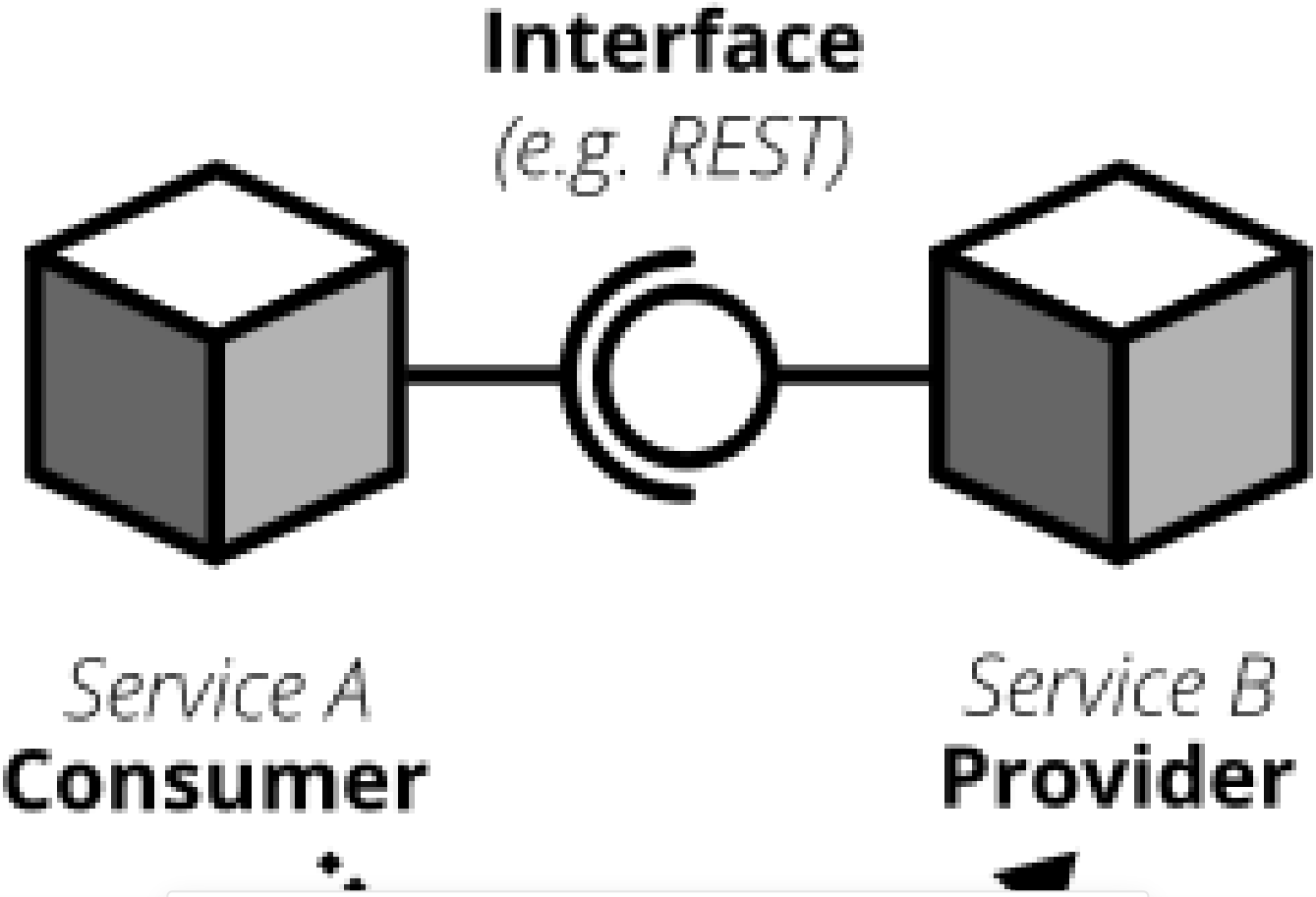
- REST and JSON via HTTPS
- RPC using something like Apache Thrift, JSON RPC, XML-RPC, gRPC, etc.

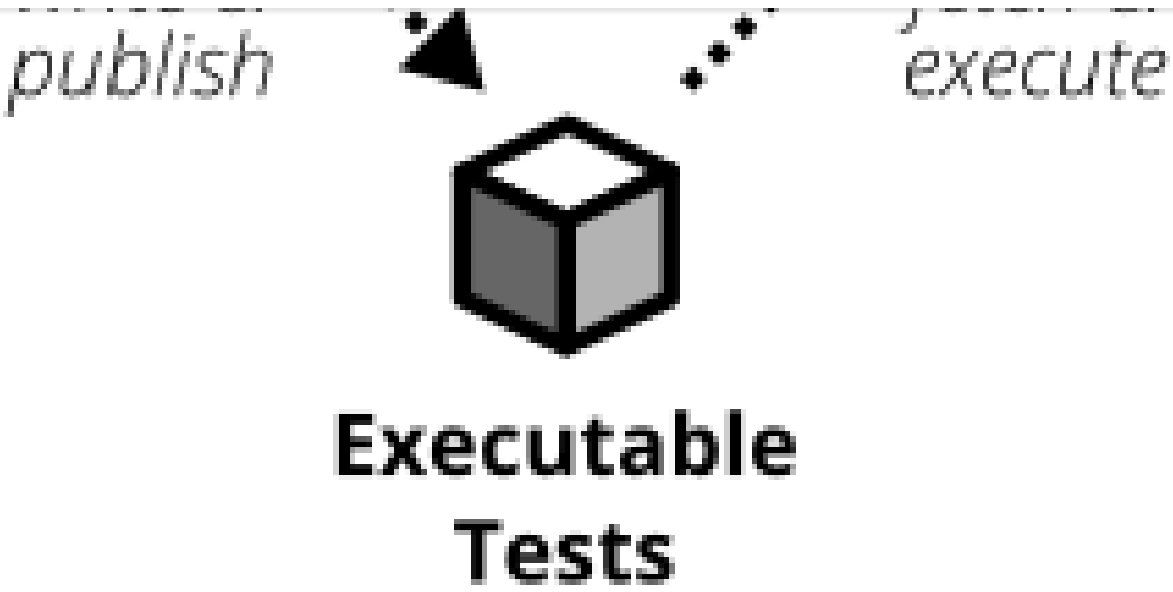
microservice. This test is written by each test consuming team and then packaged. The main aim of this test is to know the impact of the changes made by the providers of the services on the consumers consuming these services.

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. 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 is in use.



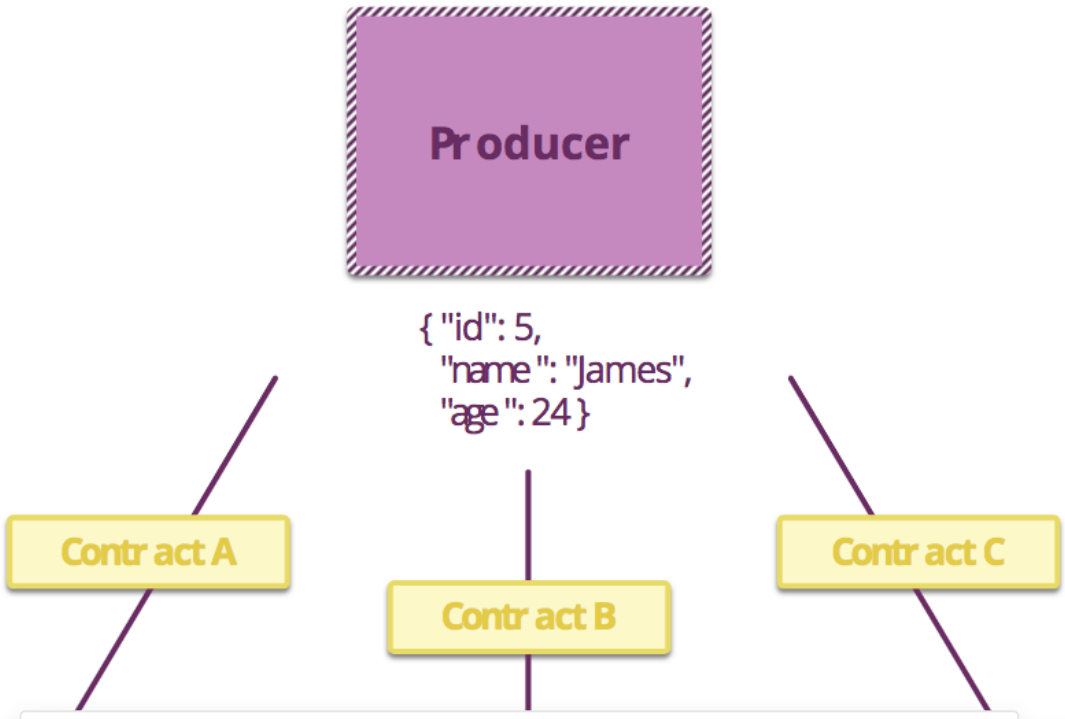
Above is the definition of two services communicating with each other, while the below states the Contract Tests asserting the contract between them is intact.

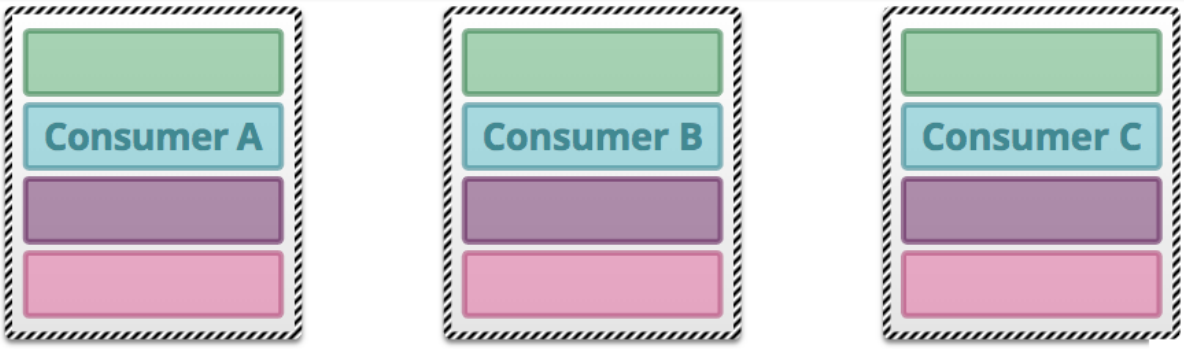




These tests are not component/integration tests. They do not test the behavior of the service deeply but that the inputs and outputs of service calls contain required attributes and that response latency and throughput are within acceptable limits. Ideally, the contract test suites written by each consuming 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.

Considering the below Service dependency, here is an example,



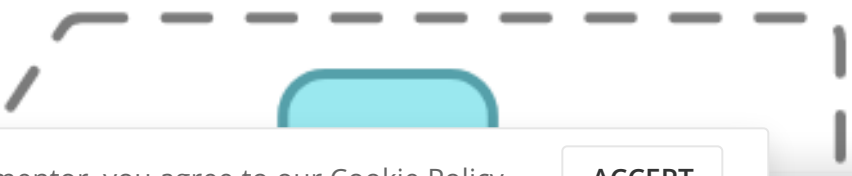


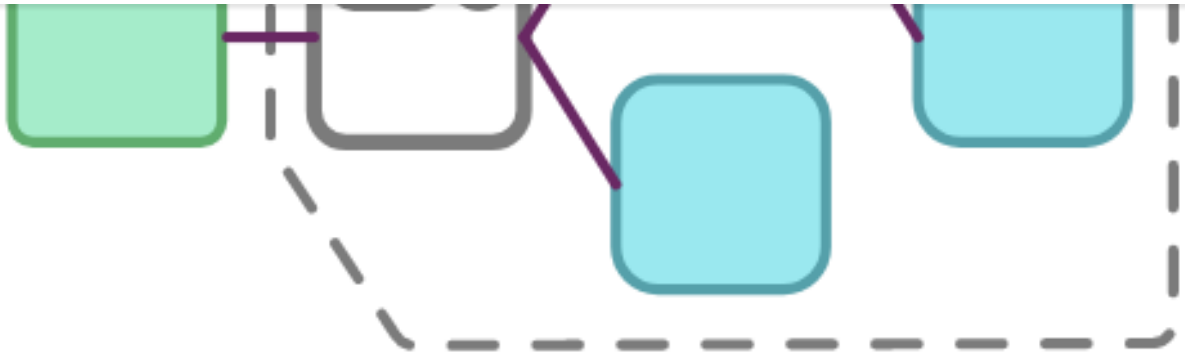
Consumer A: couples to only the **identifier** and **name** fields. As such, the corresponding contract test suite asserts that the resource response contains those fields. It does not make any assertion regarding the **age** field.
Consumer B: couples to the **identifier** and **age** fields so the contract tests assert they are present but make no assertions about the **name** field.
Consumer C: requires all three **(identifier, name, and age)** 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 producers may choose to deprecate the name field and introduce another field containing a composite object with the name components. In order to 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 depreciation. 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 robustness when serializing and deserializing messages ignoring any fields that are not important to them. Hence, contract test suites are 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. The sum of all consumer contract tests defines the overall service contract.

End-To-End Testing (API and UI)



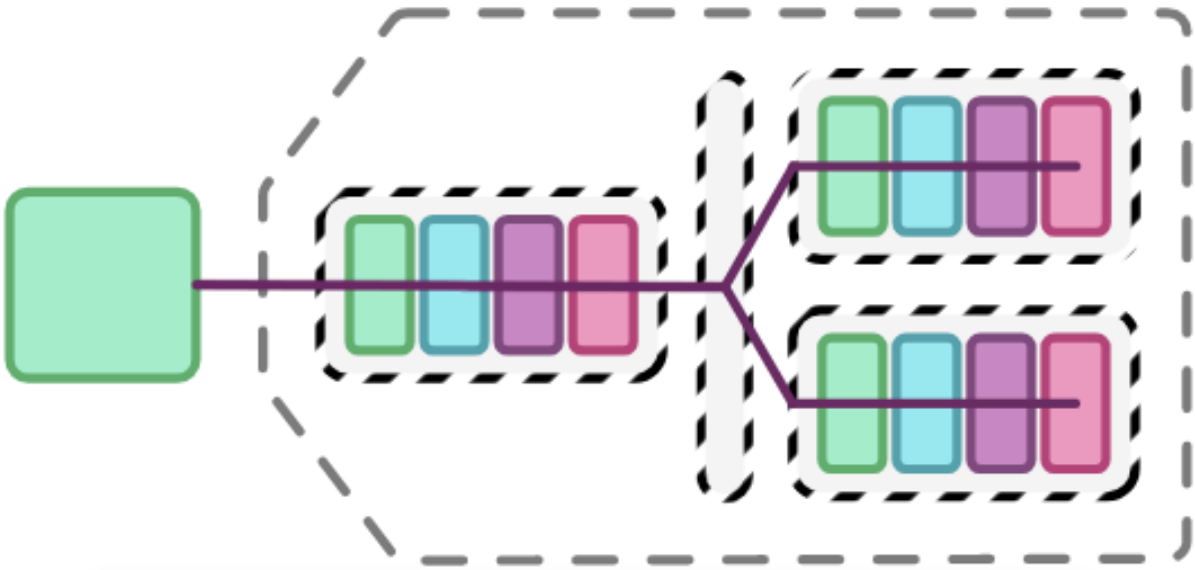


An end-to-end test verifies that a system meets external requirements and achieves its goals, testing the entire system, from end to end. In contrast to other types of test, the intention with end-to-end tests is to verify that the system as a whole meets business goals irrespective of the component architecture in use.

The system is fully deployed and is treated as a black box and the test is exercised. With Public interference through GUIs and API, the system is manipulated. End to End Tests is more business facing. In a microservices architecture, for one behavior, there are

many microservices which interact to respond to that behavior, an end to end testing provides value by adding coverage of gaps between the system.

Thus making, the test boundary for end-to-end tests, is much larger than for other types of test.



state changes or events at the perimeter formed by the tests' boundary.

NOTE: UI End-to-end tests can be flaky and difficult to automate since the ever-changing UI/UX. Thus to have more stability coming one level down the service abstraction layer (APIs) and automating there End-to-end tests should also make your test suites reasonably qualitative and resourceful for automation of end-to-end key application flows/behaviors.

End-to-end Tests Maintenance

Maintenance of E2E tests can be very high because of its dynamic, volatile, ever-changing behaviors and many more moving parts. Thus follow the below guidelines for E2E tests,

1. 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. Always have an upper time threshold for the runtime of E2E tests, thus limiting the maintenance by prioritizing and automating only the required and essential ones.

2. Focus on service call flows, user journeys, and important data flow between sub-systems

To ensure all tests in an end-to-end suite are valuable, model them around clients of the service/system and the journeys those client's data make through the system. This provides confidence in the parts of the system the client services (or users) value the most and leaves coverage of anything else to other types of testing.

3. Make tests data-independent

In E2E tests always the test data should be created and used the tests themselves, rather than depending pre-existing data, which is 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. Automated management of the data relied upon by end-to-end tests reduces the chances of failures.

Microservices Continuous Testing and Integration

an individual isolated service's testing or individual service tested by stubbing the **other required services using test doubles** or **integrated testing of real, deployed services communicating with each other** and **testing an entire echo system with n services acting as one.**

Moving forward, the definition of each and every stage and how the development and test artifacts at each stage are going to be deployed using/defining the pipelines shall be defined.

Unit tests

- Written by developers as part of the code
- Code coverage of up to 60% and no lower than 50%. It's better to deliver code to QA and build integration/end-to-end tests than to reach 80 or 90% coverage with time-consuming unit tests that must be maintained in future
- While running all unit tests should take about 1 minute, all tests for a single class must be launchable in seconds
- 100% of tests must pass before the code is released to the QA process

Integration tests

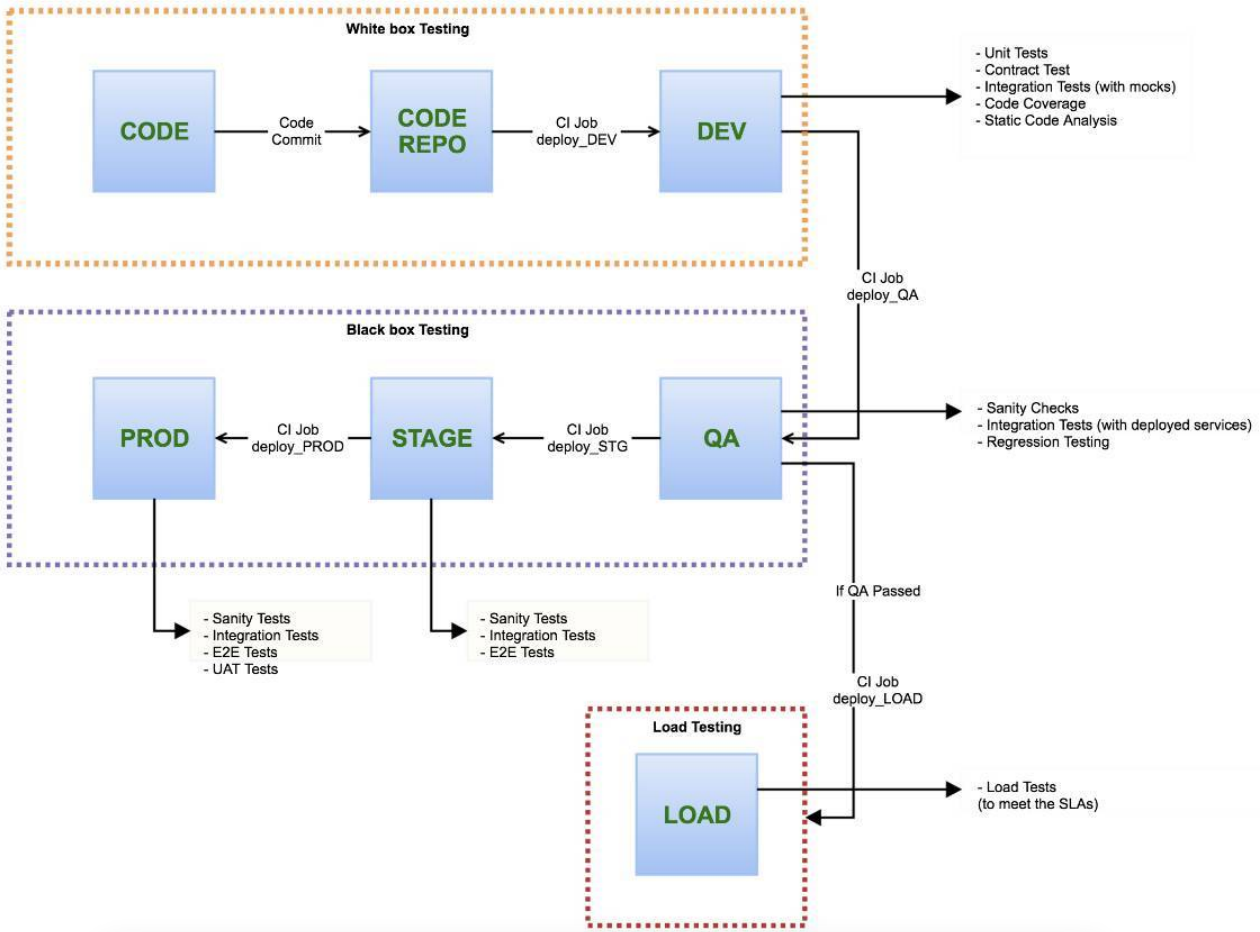
- Written by QA engineers, supported by developers. While QA decides on testing scenarios and the codebase, the developer must expose modules or single dependencies to make mocking easy and effective
- Measure how many features are covered by tests
- Use an appropriate environment (parallel execution of tests), with mocked data (usually hardcoded JSON instead of API calls or hardcoded database model).

End-to-end tests

- Target should be that all E2E tests should take no more than three hours (considering UI tests, but only for API this should be less than 20-30 minutes, even when the test cases are 1000+).
- This limit gives us enough time to test and release in a single day. We also have enough time to do final fixes, rerun failing tests or do manual checks for them.
- Cover all product/service/portal/system features. Our time limit means we have to prioritize

- P2

- 95% of tests must pass before the code is released to production. With such a complex, inter-dependent and distributed services in a system (happy and sad paths, decent combinations of configurations) and hiccups like latency api responses, flaky tests, etc., it is not possible to guarantee 100%.
- That's why the remaining 5% of failing tests are retested automatically (with Retry Implementations) and/or checked manually by QA engineers



Test driven development Unit testing Rest Web Services Test strategy
Test automation

Enjoy this post? Give **Hemanth Kandukuri** a like if it's helpful.

1 SHARE

Hemanth Kandukuri

Lead SDET (API & UI Automation, Performance Testing, Automation Framework Development)

Lead SDET with 7 years of experience in Test Automation, API Testing, UI Automation, Mobile Apps Test Automation, and Performance Testing and Engineering. Core expertise in Test Engineering, Unit Testing, Integration Testing, E2E ...

FOLLOW

Be the first to share your opinion

Leave a reply

Enjoy this post?

1

Joe Masilotti

Better Unit Testing with Swift

I'm a big fan of third party testing frameworks. My first foray into testing was with Cedar and eventually Rspec. Now that more of my work has moved to Swift I've been enjoying Quick and Nimble.

With the release of Swift 2.0 and Xcode 7 I though it was a great time to try writing my unit tests using only Apple's XCTest framework. That's right, no dependencies, no BDD, and no matchers.

Here are some concepts t ...

Join and start **discussions** with fellow developers

START A DISCUSSION

Enjoy this post?