WHITE PAPER

AN INSIGHT INTO MICROSERVICES TESTING STRATEGIES

Arvind Sundar, Technical Test Lead

Abstract

The ever-changing business needs of the industry necessitate that technologies adopt and align themselves to meet demands and, in the process of doing so, give rise to newer techniques and fundamental methods of architecture in software design. In the context of software design, the evolution of "microservices" is the result of such an activity and its impact percolates down to the teams working on building and testing software in the newer schemes of architecture. This white paper illustrates the challenges that the testing world has to deal with and the effective strategies that can be envisaged to overcome them while testing for applications designed with a microservices architecture. The paper can serve as a guide to anyone who wants an insight into microservices and would like to know more about testing methodologies that can be developed and successfully applied while working within such a landscape.

Infosys® Navigate your next

Introduction

Microservices attempt to streamline the software architecture of an application by breaking it down into smaller units surrounding the business needs of the application. The benefits that are expected out of doing so include creating systems that are more resilient, easily scalable, flexible, and can be quickly and independently developed by individual sets of smaller teams.

Formulating an effective testing strategy for such a system is a daunting task. A combination of testing methods along with tools and frameworks that can provide support at every layer of testing is key; as is a good knowledge of how to go about testing at each stage of the test life cycle. More often than not, the traditional methods of testing have proven to be ineffective in an agile world where changes are dynamic. The inclusion of independent micro-units that have to be thoroughly tested before their integration into the larger application only increases the complexity in testing. The risk of failure and the cost of correction, post the integration of the services, is immense. Hence, there is a compelling need to have a successful test strategy in place for testing applications designed with such an architecture.

Microservices architecture

The definition of what qualifies as a microservice is quite varied and debatable with some SOA (service-oriented architecture) purists arguing that the principles of microservices are the same as that of SOA and hence, fundamentally, they are one and the same. However, there are others who disagree and view microservices as being a new addition to software architectural styles, although there are similarities with SOA in the concepts of design. Thus, a simpler and easier approach to understand what microservices architecture is about, would be to understand its key features:

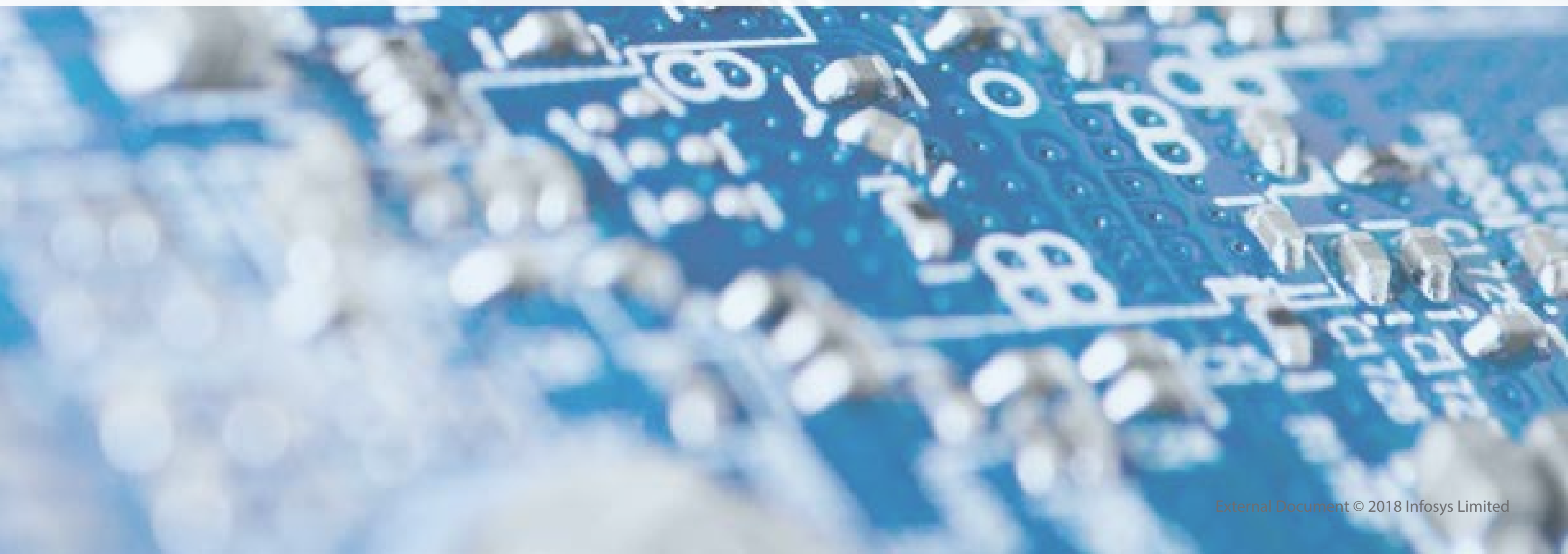
- Self-contained and componentized
- Decentralized data management
- Resilient to failures
- Built around a single business need
- Reasonably small (micro)

The points above are not essentially the must-haves for a service to be called a microservice, but rather are 'good-to-have.' The list is not a closed one either, as it can also include other features that are common among implementations of a microservices architecture. However, the points provide a perspective of what can be termed as a microservice. Now that we know what defines a microservice, let us look at the challenges it poses to testers.

Challenges in testing microservices

The distributed and independent nature of microservices development poses a plethora of challenges to the testing team. Since microservices are typically developed by small teams working on multiple technologies and frameworks, and are integrated over light-weight protocols (usually ReST over HTTPs, though this is not mandatory), the testing teams would be inclined to use the Web API testing tools that are built around SOA testing. This, however, could prove to be a costly mistake as the timely availability of all services for testing is not guaranteed, given that they are developed by different teams. Furthermore, the individual services are expected to be independent of each other although they are interconnected with one another. In such an environment, a key factor in defining a good test strategy would be to understand the right amount of testing required at each point in the test life cycle.

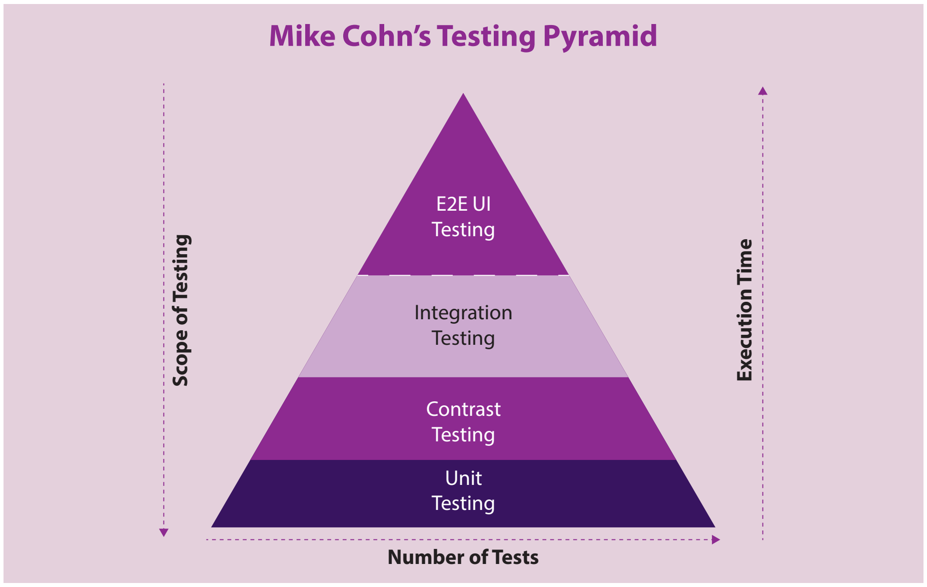
Additionally, if these services integrate with another service or API that is exposed externally or is built to be exposed to the outside world, as a service to consumers, then a simple API testing tool would prove to be ineffective. With microservices, unlike SOA, there is no need to have a service level aggregator like ESB (enterprise service bus) and data storage is expected to be managed by the individual unit. This complicates the extraction of logs during testing and data verification, which is extremely important in ensuring there are no surprises during integration. The availability of a dedicated test environment is also not guaranteed as the development would be agile and not integrated.



Approach to testing microservices and testing phases

In order to overcome the challenges outlined above, it is imperative that the test manager or lead in charge of defining the test strategy appreciates the importance of Mike Cohn's Test Pyramid[i] and is able to draw an inference of the amount of testing required.

The pictorial view emphasizes the need to have a bottom-up approach to testing. It also draws attention to the number of tests and in turn, the automation effort that needs to be factored in at each stage. The representation of the pyramid has been slightly altered for the various phases in microservice testing. These are:



i. Unit testing

The scope of unit testing is internal to the service and in terms of volume of tests, they are the largest in number. Unit tests should ideally be automated, depending on the development language and the framework used within the service.

ii. Contract testing

Contract testing is integral to microservices testing and can be of two types, as explained below. The right method can be decided based on the end purpose that the microservice would cater to and how the interfaces with the consumers would be defined.

a) Integration contract testing:
Testing is carried out using a test double (mock or stub) that replicates a service that is to be consumed. The testing with the test double is documented and this set needs to be periodically verified with the real service to ensure that there are no changes to the service that is exposed by the provider.

b) Consumer-driven contract testing:
In this case, consumers define the way in which they would consume the service via consumer contracts that can be in a mutually agreed schema and language. Here, the provider of the service is entrusted with copies of the individual contracts from all the consumers. The provider can then test the service against these contracts to ensure that there is no confusion in the expectations, in case changes are made to the service.

iii. Integration testing

Integration testing is possible in case there is an available test or staging environment where the individual microservices can be integrated before they are deployed. Another type of integration testing can be envisaged if there is an interface to an externally exposed service and the developer of the service provides a testing or sandbox version. The reliance on integration tests for verification is generally low in case a consumer-driven contract approach is followed.

iv. End-to-end testing

It is usually advised that the top layer of testing be a minimal set, since a failure is not expected at this point. Locating a point of failure from an end-to-end testing of a microservices architecture can be very difficult and expensive to debug.

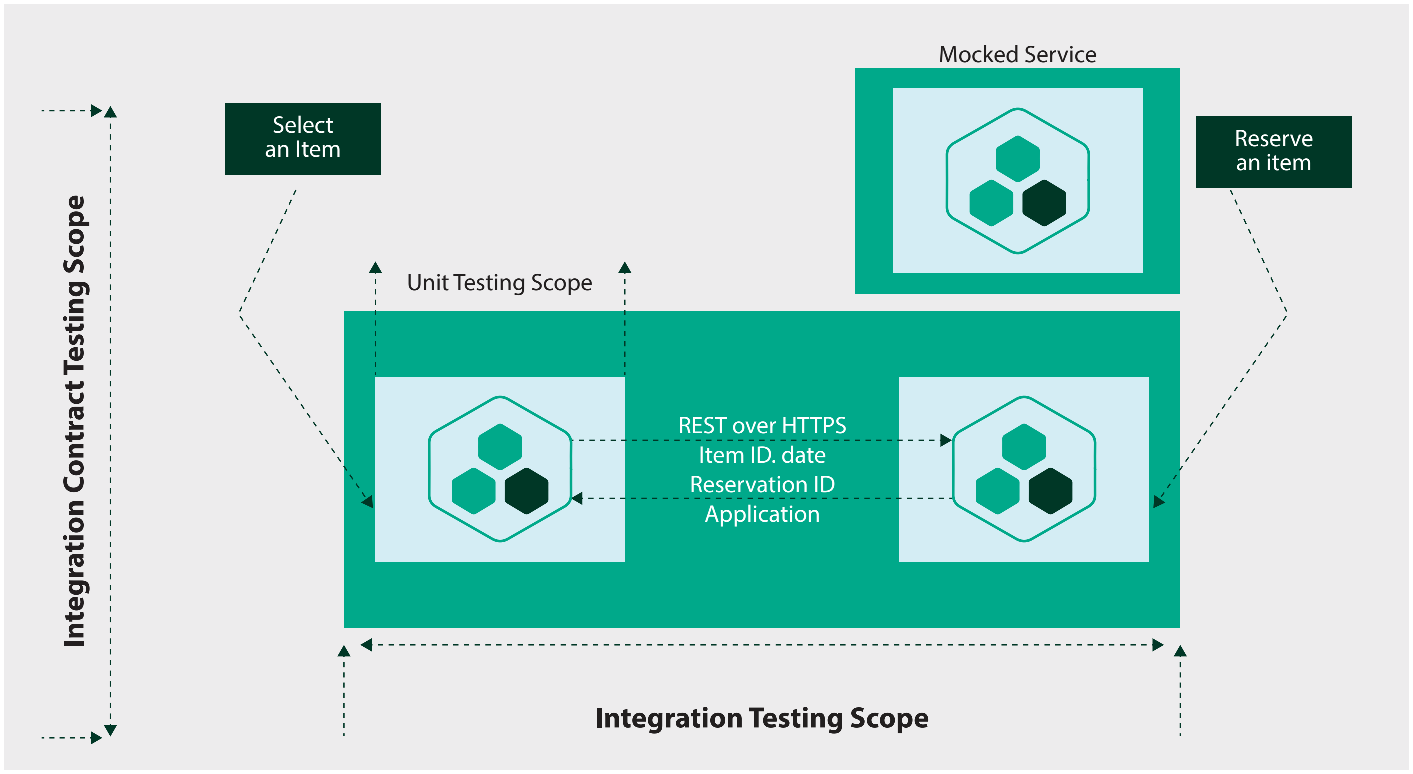
Testing scenarios and test strategy

In order to get a clear understanding of how testing can be carried out in different scenarios, let us look at a few examples that can help elucidate the context of testing and provide a deeper insight into the test strategies used in these cases.

• Scenario 1:

Testing between microservices internal to an application or residing within the same application

This would be the most commonly encountered scenario, where there are small sets of teams working on redesigning an application by breaking it down into microservices from a monolithic architecture.
In this example, we can consider an e-commerce application that has two services **a) selecting an item** and **b) reserving an item**, which are modelled as individual services. We also assume there is a close interaction between these two services and the parameters are defined using agreed schemas and standards.



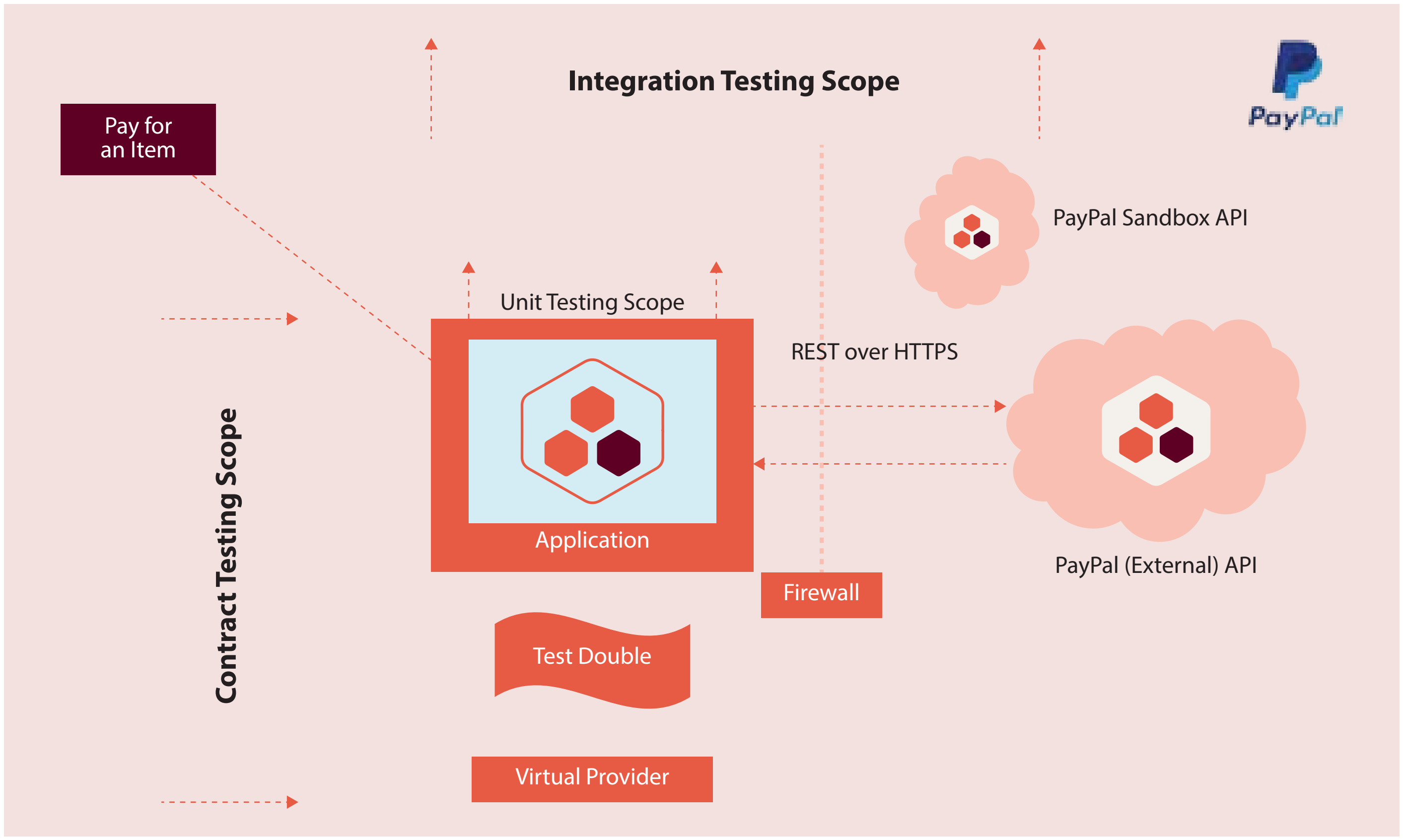
- For **unit testing,** it would be ideal to use a framework like xUnit (NUnit or JUnit). The change in data internal to the application needs to be verified, apart from checking the functional logic. For example, if reserving an item provides a reservation ID on success in the response to a REST call, the same needs to be verified within the service for persistence during unit testing.
- The next phase of testing in **contract testing**. In case there are several dissimilar consumers of the service within the application, it is recommended to use a tool that can enable consumer-driven contract testing. Open source tools like Pact, Pacto, or Janus can be used. This has been discussed in further detail in the last example and hence, in the context of this example, we will assume that there is only a single consumer of the service. For such a condition, a test stub or a mock can be used for testing by way of **integration contract testing**. Data being passed between the services needs to be verified and validated using tools like SOAPUI. For example, an item number being passed between the services that selects it to the one that reserves it.
- **E2E tests** should ensure that dependency between microservices is tested at least in one flow, though extensive testing is not necessary. For example, an item being purchased should trigger both the 'select' and 'reserve' microservices.

• Scenario 2:

Testing between internal microservices and a third-party service

Here, we look at a scenario where a service with an application consumes or interacts with an external API. In this example, we have considered a retail application where **paying for an item** is modelled as a microservices and interacts with the **PayPal API** that is exposed for authenticating the purchase.

Let us look at the testing strategy in each phase of the test cycle in this case:

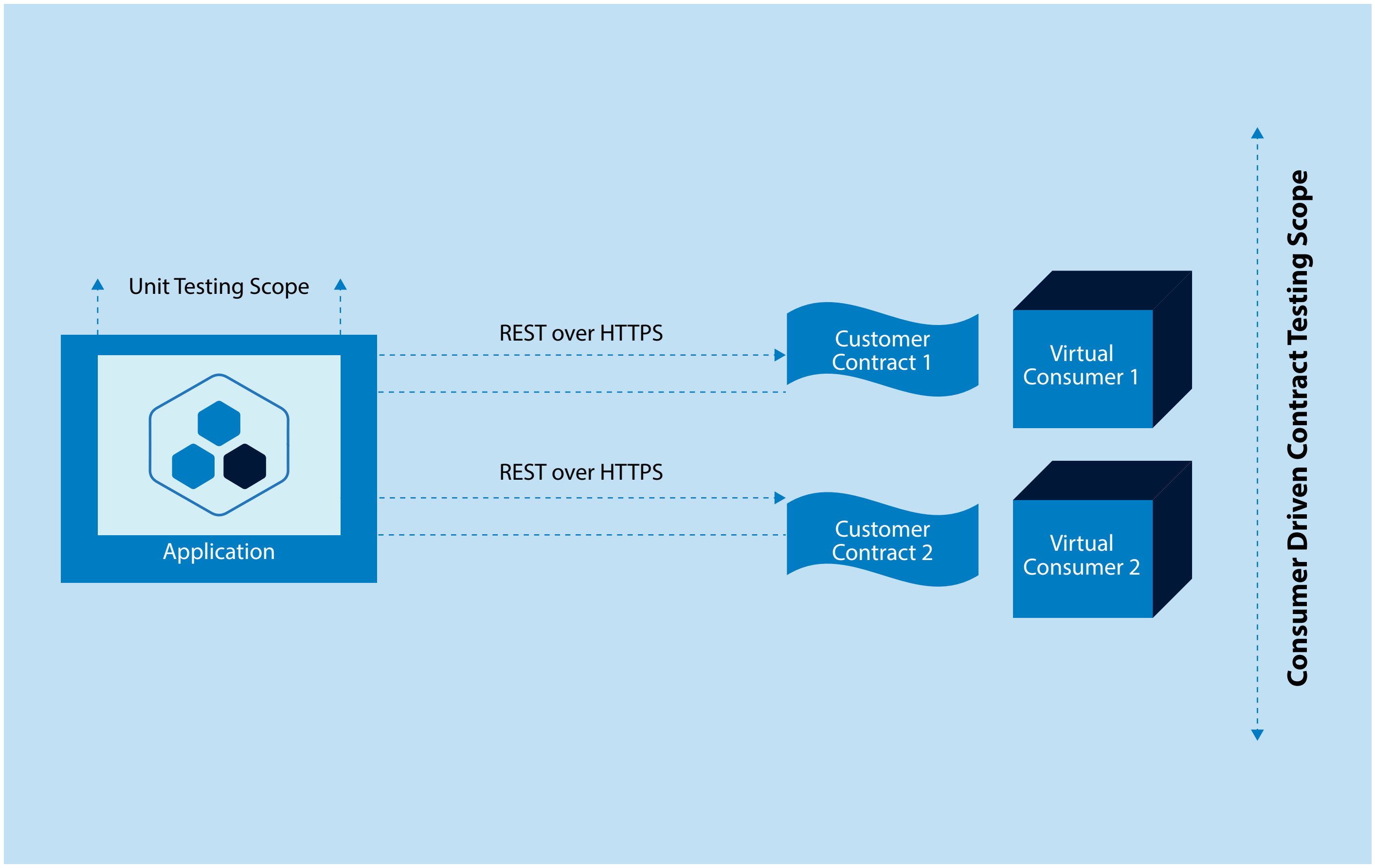


- **Unit tests** should ensure that the service model is catering to the requirements defined for interacting with the external service, while also ensuring that internal logic is maintained. Since there is an external dependency, there exists a need to ensure that requirements are clearly defined and hence, documenting them remains key. TDD approach is suggested where possible and any of the popular frameworks discussed in the previous example can be chosen for this.
- **Contract testing** can be used in this case to test the expectations from consumer microservices, that is, the applications internal service, decoupling it from the dependency on the external web service to be available. In this context, test doubles, created using tools like Mockito or Mountebank, can be used to define the PayPal API's implementation and tested. This is essentially integration contract testing and again needs to be verified with a live instance of the external service periodically, to ensure that there is no change to the external service that has been published and consumed by the consumer.
- **Integration tests** can be executed if the third-party application developer provides a sandbox (e.g. PayPal's Sandbox API[ii]) for testing. Live testing for integration is not recommended. If there is no availability of a sandbox, integration contract testing needs to be exercised thoroughly for verification of integration.
- **E2E tests** should ensure that there are no failures in other workflows that might integrate with the internal service. Also, a few monitoring tests can be set up to ensure that there are no surprises. In this example, selecting and purchasing an item (including payment) can be considered an E2E test that can run at regular and pre-defined intervals to spot any changes or breaks.

• Scenario 3:

Testing for a microservice that is to be exposed to public domain

Consider an e-commerce application where retailers can check for availability of an item by invoking a Web API.



- **Unit tests** should cover testing for the various functions that the service defines. Including a TDD development can help here to ensure that the requirements are clearly validated during unit testing. Unit test should also ensure that data persistence within the service is taken care of and passed on to other services that it might interact with.
- **Contract testing** – In this example, consumers need to be set up by using tools that help define contracts. Also, the expectations from a consumer's perspective need to be understood. The consumer should be well-defined and in line with the expectations in the live situation and contracts should be collated and agreed upon.

  Once the consumer contracts are validated, a consumer-driven contract approach to testing can be followed. It is assumed that in this scenario, there would be multiple consumers and hence, individual consumer contracts for each of them. For example, in the above context, a local retailer and an international retailer can have different methods and parameters of invocation. Both need to be tested by setting up contracts accordingly. It is also assumed that consumers subscribe to the contract method of notifying the provider on the way they would consume the service and the expectations they have from it via consumer contracts.
- **E2E tests** – minimal set of E2E tests would be expected in this case, since interactions with external third parties are key here

In conclusion

Improvements in software architecture has led to fundamental changes in the way applications are designed and tested. Teams working on testing applications that are developed in the microservices architecture need to educate themselves on the behavior of such services, as well as stay informed of the latest tools and strategies that can help deal with the challenges they could potentially encounter. Furthermore, there should be a clear consensus on the test strategy and approach to testing. A consumer-driven contract approach is suggested as it is a better way to mitigate risk when services are exposed to an assorted and disparate set of consumers and as it further helps the provider in dealing with changes without impacting the consumer. Ensuring that the required amount of testing is focused at the correct time, with the most suitable tools, would ensure that organizations are able to deal with testing in such an environment and meet the demands of the customer.

References : [i] https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[ii] https://www.sandbox.paypal.com