newt|global
Aiming At Customer Delight

info@newtglobal.com

CONTINUOUS INTEGRATION & CONTINUOUS DELIVERY

TESTING MICRO SERVICES PART I OF II

Testing Micro Services is an area that cannot be avoided or procrastinated to any point of time. Each services' build before it reaches the deployment stage must be ensured that it passes the test criteria defined by the project team. While the Project team / Organization focusses on Designing and Developing Applications using Micro Services, it is also equally important to design Testing Strategies to test those Micro Services. A combination of testing methods along with tools and frameworks that can provide support at every layer of testing is key. We may not be able stick to the traditional model of testing, as in an agile environment changes are very dynamic and with a micro service, we would have to deep dive into each layer to make sure there is not a hit in the production environment.

AN INTRODUCTION

The independent nature of Micro Services possesses significant challenges to the Testing team. Picking the right tools that are inclined to test the Web API, SDK's that are built around SOA becomes a necessity factor in order to automate the Testing process. The Test pyramid gets a little wider and deeper with Integration Testing, Component Testing and Contract Testing coming into the framework above Unit Testing. A bottom-up approach to testing improves stability of the micro services as they are developed. It is also very essential to remain light weight while we are implementing the various tests to test a service, at the same time make sure that there is a coverage to each layer and layers of service.

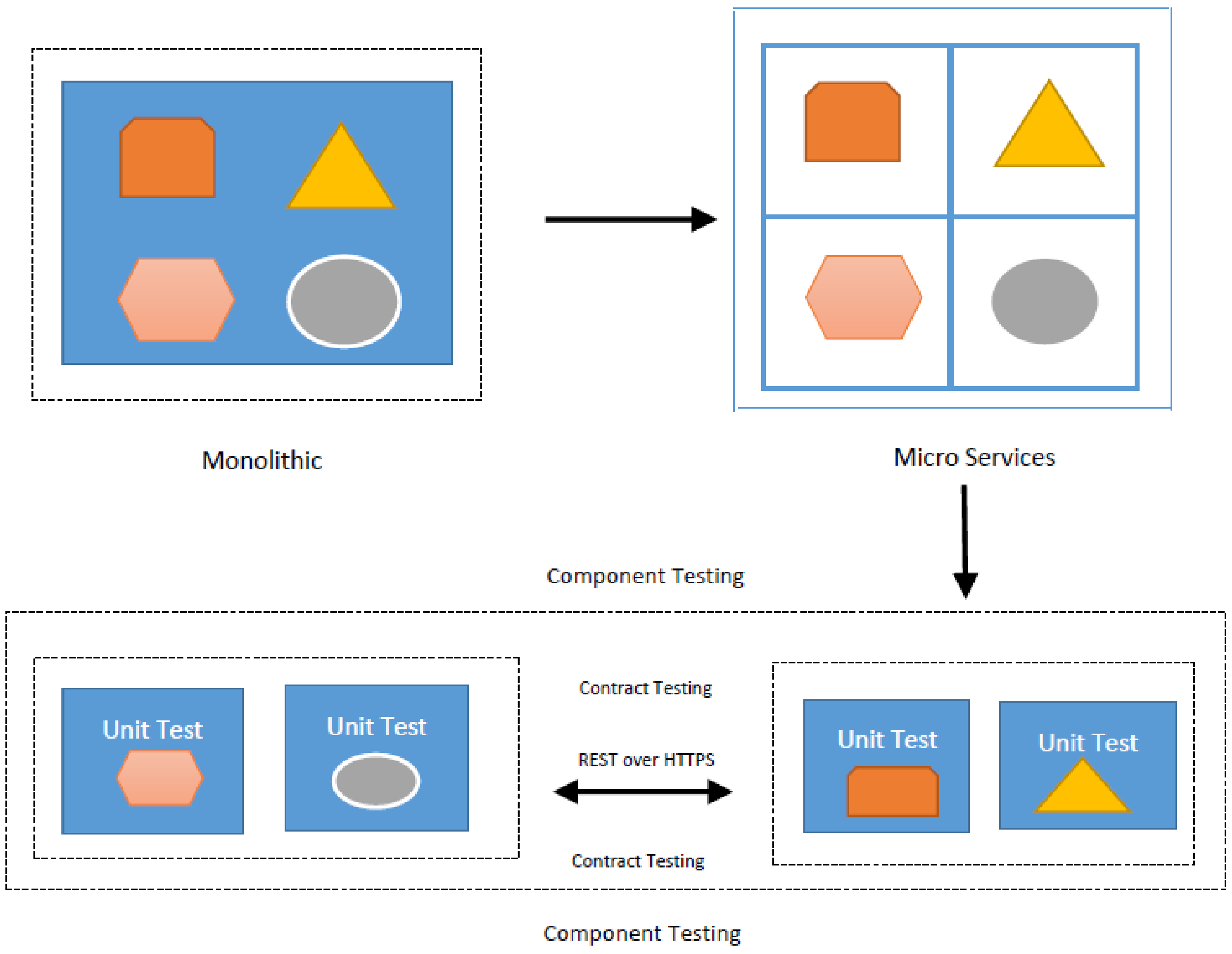


Fig.: Testing Micro Services

UNIT TESTING

Any function has to be tested at the unit level first, by testing the code written by the Developers for a particular service. Unit Testing is done usually at class level by testing a small piece of behavior and make sure the code works correctly at the lowest level. Unit Testing focusses on small test suite by testing the behavior of modules. Test Doubles looks the interactions and collaborations between an object and its dependencies.

In a micro service, unit tests are most useful in the service layer, where they can verify business logic under controlled circumstances against conditions provided by mock collaborators. They are also useful in domain logic, resources, repositories, and adapters for testing exceptional conditions. Unit Tests have to be executed frequently, with each build. To follow an automated process, we can configure Unit Tests on a Continuous Integration server (say Jenkins) that constantly monitors for changes in the code.

COMPONENT TESTING

In micro services, components are services that tests a portion of the system. Tests written at this granularity makes it possible to thoroughly acceptance the test behavior. This is done by creating an internal interface inside the unit for testing before reaching the production. Component Testing is also otherwise called 'Functional Testing', where the services test to verify integration of all components is functionally correct that do not require external dependencies. Testing is done in isolation by using mock components or by creating a Test Service on the same server and test the complexity contained within that micro service, but make sure that that final application code reaches the production and not the test code.

The scope is limited to a single component and Testing is done using the same technology that incoming requests would use, restful api for example. This method also thoroughly tests the network calls for that particular service and the results are quick, so that developers can gain confidence that whatever changes they have made, does not affect any other service or component. Test doubles can be configured to isolate the micro service from external service that allows error conditions tested in a controlled environment and repeatable manner. Mock implementations can be avoided by testing against external devices with recorded responses. Though this practice is more realistic and traditional, there is a complexity involved in order to maintain the recorded responses for each service.

FACTS TO CONSIDER

Test for agreed set of input and output attributes is set to be defined as Contract Testing. In other words, the boundary of an external service is verified that it meets the contract expected by consuming the service. Contract Testing makes sure that any additions shouldn't break the existing functionality of the service, even as the service changes overtime. The production team are well aware of the impact of the changes on their consumers by looking at the test suites written by the testing team that are well packaged and runnable in build pipeline for producing services. In Micro Service there is a contract involved with each of the service that interacts with the Producer.

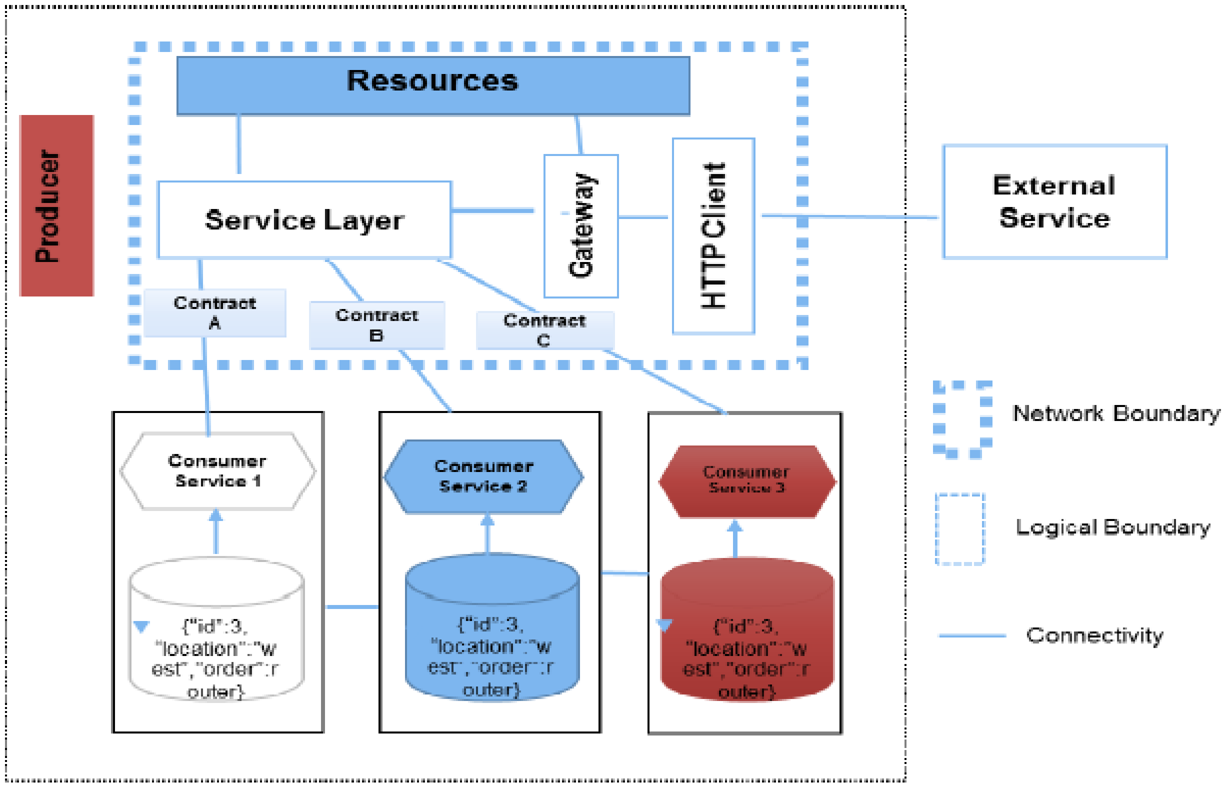
Mock frameworks can be used to fake responses from external services that gives the possibility to test services in an isolated environment without relying on external services. They can also be used as Integration Contract Testing to validate the parameters that are passed to the external service are integrated to receive the correct request. When there are different teams that run tests for different services, which means essentially each team have their own contract testing to be executed. A service contract can be defined by the maintainer by looking at the different contract tests. Service contract can be used to manage changes as in when they are introduced.

Scenario:

Consider a service that exposes a resource with three fields, an identifier, a location and an order. This service is adopted by three different consumers coupled to different parts of the resource. Consumer A couples to only the identifier and location field. This does not make any claim regarding the order field. Whereas Consumer B couples to the identifier and order fields, and does not make assert to the location field. Consumer C requires all three fields and has a contract test suite that affirms they all are present and communicating.

Aiming At Customer Delight

info@newtglobal.com



When a new consumer adopts the API but requires both location and address, the maintainers can choose to deprecate the location field and introduce another field containing an object with location components. The maintainers can check the Service contract, remove the old location field and see which contract test fail. In this case, Consumer A and C would have to be notified on the change. Post migration, the deprecated field can be removed from Consumer A and C. For this process to be effective, the service contract need to be updated on a regular basis with fields that are not important.

This is an example of how an API can be changed over a period of time without breaking the contract of the consumers. To test this efficiency is known to be Contract Testing. These tests can be automated with a variety of contract testing tools.