***Lost all previously stored data………***

***16-06-2023***

***Testing Integrations with Consumer-Driven Contract Tests***

Consumer-Driven Contract testing is a way to test integrations between services. The main idea is that when an application or a service (consumer) consumes an API provided by another service (provider) a contract is formed between them. The contract contains information about how the consumer calls the provider and what the consumer needs from the responses. The contract can then be used to test both sides of the integration separately.

The testing method is said to be useful when testing integration-heavy systems such as systems based on microservice architecture. Therefore the research question of the thesis is: "with a focus on integrations, is Consumer-Driven Contract testing a viable addition to a testing strategy used to test a system based on microservice architecture, and if so, why?" **RQ**

The research question is first approached by taking a look at the most recent literature. The goal is to learn about different testing methods and create a basic understanding of a general testing strategy for microservices. The next step is to figure out how the Consumer-Driven Contract testing fits that picture. The Consumer-Driven Contract testing is introduced thoroughly to gain a good understanding of its core concepts, advantages, disadvantages, and tooling.

After the literature check, the research question is approached by introducing a case study based on a microservice architecture. Its testing strategy is described in detail, and Consumer-Driven Contract tests are implemented for it. The testing methods are compared by systematically implementing defects to the integrations and seeing how the testing methods catch them. Finally, the results and experiences are shared and analysed, and the research question gets answered.

The results based on literature and experiences from the case study proved that the Consumer-Driven Contract testing is a viable way to test integrations. The tests implemented in the case study caught every defect from the integrations, and the case study was able to verify the advantages mentioned in the literature. It was shown that the Consumer-Driven Contract tests could replace the more traditional integration tests completely. That results to more deterministic testing strategy as the integrations are tested in isolation.

It should be emphasized that the teams have to be able to communicate with each other to implement and achieve the benefits of Consumer-Driven Contract testing. The level of communication between the teams has to be mature enough to share the contracts and to coordinate the implementation. Communication is the foundation that enables or disables the testing method. Because of that improving the ways of communication should be a major focus for the teams who want to implement Consumer-Driven Contract tests.

***(26) Contract testing - part one - docker containers | LinkedIn – (For design and implementation)***

In a distributed microservices environment with tens of services interacting with each other, it make sense to start building the services - and create test cases incorporating the dependent services - at the time when these dependent services are not built yet but we know how their APIs looks like.

Right at the time when the architecture team has completed defining the API's schema for a service by providing a Swagger file, this schema - that is a contract the service will have to respect at runtime - can be used to create a "mock" service that will run in a local "dockerized" environment where test cases can be written and executed against this service.

Why it "make sense" to start building the services in this way? The quick answer is: because it defines a process that has a series of benefits hence becoming one of the "best practices" a development team needs to follow.

In software development, it is better to include the testing from the beginning phase of the development process:

* it is better to involve the QA early into the process to get familiarized with the requirements and create the test cases,
* it is better to define a process that can benefit from running test cases that are created based on architecture specifications - contracts - and verifies them constantly during the local development or during the execution of the promotion pipeline.

At the same time, the test cases can be done in isolation, without having to spawn the whole framework: one service can test its dependency on the immediate service and only on this one. Another advantage is the fact that an end-to-end integration test case it is most likely easier to get executed and passed when all the individual integration test cases have been performed successfully for a while. One last advantage I want to mention here is: its simplicity. If we can setup an environment where we don't have to write any line of code that will give us the ability to create these test cases, wouldn't be this a great benefit?

(refer practical + other parts II,III… )

***Containers vs Microservices: What’s The Difference***?

A post from Ev Kontsevoy summarized the comparison of these two terminologies in an interesting way:

“A container is a useful resource allocation and sharing technology. It’s something DevOps people get excited about. A microservice is a software design pattern. It’s something developers get excited about.”

In other words, we can sum this up as:

* Microservices are about the design of software.
* Containers are about packaging software for deployment.

So, we can choose whether to use a container for hosting a microservice. But to get full value from both, it is significantly better to run microservices within containers.

Deploying an entire application to a single VM introduces a single point of failure risk, whether or not a microservice architecture has been used. But spreading the application through microservices across multiple containers results in fully exploiting the value of both by providing resilience as well as agility through scaling and improvements targeting specific services without negatively impacting the entire application.

Flexibility is also introduced in that developers can write applications in the language of their choice since the container will allow them to deploy across whatever environment is provided. Efficiency comes from containers using less resources compared to VMs.

An added benefit comes in the form of security through isolation and a broader attack surface that limits the impact should a single microservice or container be subject to a security breach such as a hacking attack.

***DevOps for Microservices - Creating Change Together***

***Previous***

***Next***

***DevOps and Microservices – Creating change together***

The cloud-native software architectures are not complete without microservices. Enterprises use modern practices like DevOps and microservices architectures to design software applications. This enables organizations to deploy these applications as independent services. And since DevOps and microservices are connected, they offer greater agility and operational efficiency.

DevOps helps developers and IT operations to work in close coordination with each other to deliver superior quality software quickly. It also allows developers to gain insights into production settings to improve the quality of their software.

Organizations follow DevOps practices to the cloud for replicating the production environment with better accuracy. DevOps facilitates easy collaboration amid development and IT ops teams and minimizes the resistance between them.

Microservice architecture on the other hand is a type of Service-Oriented Architecture (SOA), a software development technique. Microservices are tiny pieces of functionalities that are used as a service. These are independent building blocks, with the help of which it becomes easier to create, test and understand an application.

Since they offer greater modularity, several DevOps teams can construct and set up microservices side by side. Furthermore, it is easy to deploy microservices in any cloud. Hence, developers do not have to write specific codes for Microsoft Azure and AWS platforms separately.

DevOps practices have a simple principle. It focuses on breaking down huge problems into smaller ones and handling them in order of preference as a team. Likewise, Microservices also operate through small teams and make one functional change at a time. Besides, microservices can easily adjust the scaling without impacting the resource allocations for the remaining system. This simply indicates that the blend of DevOps and microservices in the development process is the fastest way to enhance the output of your teams.

The blend of DevOps and microservices enable organizations to develop faster at a lesser cost. This makes them more agile and productive. Microservices bring added productivity to DevOps as they can work on the same toolset for development and operations. This marks similar terminology and processes for requirements, reliance as well as obstacles. This way Devs and Ops can easily work together. When organizations don’t supplement DevOps with microservices, it can lead to competitive disadvantage and impact them negatively. To know more about how DevOps and microservices can improve your business processes and help in your business growth, talk to our experts

***Effective Test Automation Approaches for Modern CI/CD Pipelines – (Read the whole article – restructure your Dissertation)***

The rise of CI/CD has had a massive impact on the software testing world. With developersrequiring pipelines to provide quick feedback on whether their software update has been successful or not, it has forced many testing teams to revisit their existing test automation approaches and find ways of being able to speed up their delivery without compromising on quality. These two factors often contradict each other in the testing world as time is often the biggest enemy in a tester’s quest to be as thorough as possible in achieving their desired testing coverage.

So, how do teams deal with this significant change to ensure they are able to deliver high-quality automated tests while delivering on the expectation that the CI pipeline returns feedback quickly? Well, there are many different ways of looking at this, but what is important to understand is that the solutions are less technical and more cultural ones - with the approach to testing needing to shift rather than big technical enhancements to the testing frameworks.

**Use mocks and stubs wherever possible:** A lot of testers might frown on this, as the thought of using lots of mocks and stubs can be seen as avoiding the true integrated behaviour of an application. This is true for end-to-end testing which you still want to automate, but not ideal for pipeline execution. Not only does it slow down pipeline execution, but creates flakiness in your test results as external functions are not operational or out of sync with your changes. The best way to ensure that your test results are more reliable, along with allowing you to take greater control of your testing effort and improve coverage is to build mocking into your test framework and rely on stubs to intercept complex data patterns that an external function to do it for you.

A good DevOps testing strategy requires a solid base of unit tests to provide most of the coverage with mocking to help drive the rest of the automation effort up, leaving only the need for a few end-to-end automated tests to ensure everything works in order and allow your team to take confidence that the pipeline tests will successfully deliver on their quality needs.

***Microservices and Containers 101 - Learn all About Microservices***

Microservices is an architectural design for building a distributed application. Microservices break an application into independent, loosely-coupled, individually deployable services.

Containers are a lightweight, efficient and standard way for applications to move between environments and run independently. Everything needed (except for the shared operating system on the server) to run the application is packaged inside the container object: code, run time, system tools, libraries and dependencies.

Testing microservices needs a strategy that takes both service dependencies and the isolated nature of microservices into account. In order for the process of microservices testing to work, each microservice that’s in isolation needs to be verified it’s working properly and then proceed with testing the microservices together. There are a variety of tests that can be conducted in order to validate your application: unit testing, contract testing, integration testing, and end-to-end testing

***Testing Microservices - Contract Tests (read full article)***

Testing monolithic applications is relatively easy. We implement unit, integration, and probably end-to-end tests, and try to keep the testing pyramid as perfectly shaped as possible. But when it comes to testing microservices, things complicate a little. The main difference stands for communication which is essential in a microservice architecture. Luckily, for every challenge, we have a pattern and, in this case, it’s contract testing.

This article started by pointing out the challenges of testing communication in a microservice architecture. It also reviewed two common approaches to testing it - integration and e2e testing. Then, contract testing was introduced as a solution for the aforementioned problems.

Contract testing is a concept that allows testing communication (both synchronous and asynchronous) between the services in isolation. The main idea behind it is a contract. This is a place where communication rules are agreed upon and written down by the sides. The contract is a single point where the contract is defined. The parties don’t have to synchronize when the contract changes. Both producer’s and consumer’s tests are based on this contract. It ensures that if the tests pass, the services get along in production. Next, I explained the difference between Consumer Driven and Producer Driven approaches and when to use them. Finally, I gave you a tip on how not to fall into the trap of overusing contract testing.

***The Role of Containers in Your Microservice Architecture (full article)***

Technology is evolving faster than ever. People depend heavily on the internet for all kinds of regular tasks, from shopping to banking and healthcare. That’s made it critical for service providers to fulfill this ever-increasing consumer demand. This forces service providers to abandon monolith software development methods and adopt Agile and DevOps approaches that help them quickly adapt to changing requirements. Another trend is microservices-based architectures, where applications are built as multiple loosely coupled services. In this article, we will discuss the role of microservices in the DevOps process.

DevOps is a paradigm shift in the way organizations approach software development, deployment, and maintenance. DevOps shifts the whole software development lifecycle (SDLC) to a more collaborative process.

DevOps offers a multitude of benefits for an organization. However, to gain those benefits you must properly implement DevOps with standardized concepts and proper tools such as:

* Continuous integration
* Continuous delivery platforms
* Automated testing

The next consideration is the deployment strategy. Most DevOps deployment stages are targeted at highly available and scalable cloud infrastructure with the popularity of cloud-based applications and all the advantages of cloud-based deployments.

Containerized applications are one of the factors that power this shift to cloud-based deployments. Containers allow users to create isolated and portable application environments that can be deployed anywhere with all the required dependencies. Platforms like Kubernetes and Rancher provide robust orchestration capabilities for containerized deployments. Here, automation also plays a major role by automating software packaging and deployments.

A version-controlled and automated DevOps pipeline with a proper deployment strategy allows organizations to create a pipeline that encompasses all the stages of SDLC. Now that we understand the primary considerations of a DevOps pipeline, let’s see how microservices affect all these factors.

Microservices is an architectural approach to development that contrast with traditional, monolithic applications (where the entire application is considered and developed as a single entity). The microservice architecture breaks the application into different loosely coupled services.

Containerization is another factor that extends and complements microservices-based architectures. Packaging each service as a container image further reduces the complexity while streamlining the continuous delivery pipeline. Services can act as fully independent entities with all the dependencies and requirements bundled within the container. This makes the services system-agnostic and reusable while allowing them to interact with any other system.

In a microservices architecture, most testing, packaging, and deployment tasks can be automated for each service. As each service resides in an independent DevOps pipeline, any issues in a single automated task do not affect the other services.

Microservices architecture is tailor-made for DevOps with its services-based approach that allows organizations to break down the application into smaller services. This enables delivery teams to tackle individual services as separate entities—ultimately simplifying the development, testing, and deployment. (This doesn’t mean microservices should be used for every application, however. They do come with certain challenges.) The role microservices plays in DevOps includes streamlining the DevOps process and increasing productivity and quality of the application while moving developments to a flexible architecture. This leads to the development of cloud-native applications that are capable of fulfilling any user demand.

***What Are Containerized Microservices? – DreamFactory Software- Blog***

DevOps is the merger of traditional development and IT operations teams. DevOps practices place the focus on collaboration across the entire IT organization. Key components of a DevOps organization are automated and standardized deployments, freeing staff from repetitive tasks. The speed of standard deployments that containers offer makes DevOps and containerized microservices a perfect match.

***For LR – Literature Review…*** ***(refer whole papers’ sections)***

***Analysis of Consumer-driven contract tests with asynchronous communication between microservices (refer full structure)***

In the context of microservice architecture the speed of the delivery of microservices is one of the most important parts in the release cycle. To be able to deliver microservices fast and reliably the continuous integration (CI) and continuous deployment (CD) pipeline have to be efficient, of good quality and optimized. This way we can ensure the maximum speed of reliable updates. To get the maximum out of the most beneficial aspect of microservices, the fast release cycle, we need to have fast testing. With the introduction of microservices in an architecture the amount of interfaces that can be reached is significantly higher. Most, if not all, of the communication between microservices rely on these interfaces. This proves to be a new problem in testing. These interfaces have to be tested reliably and quick. Consumer-driven contract (CDC) tests apply exactly at these interfaces between microservices without needing as much time as end-to-end tests. One of the reasons consumer-driven contract testing is deployed in testing pipelines in the context of microservice structure is their fast execution with reliable enough results to catch many errors. In the best case, before the end-to-end tests are even run. This allows the testing department to run less end-to-end tests and reduce the intensity and necessity of end-to-end tests. CDC testing is well worked out in synchronous messaging architectures even if it still lacks popularity. It is however not as seemingly integrated in asynchronous communication as it is in its synchronous counterpart.

**(Introduction - structure, RQs)\*\*\*, 2.3, 2.4, 3\*\*\*,7**

***An Empirical Analysis of Microservices Systems Using Consumer-Driven Contract Testing (refer full study)***

Testing has a prominent role in revealing faults in systems that are based on Microservices-Based Architectures (MSAs). A central discussion point in MSAs is the granularity of services, that are often in different levels of abstraction [1]. Similarly, the granularity of tests in MSAs is reflected in different test types [2]. The individuality of microservices [3], [4] means that unit tests are part of the software development lifecycle [5], but the broader architectural scope of MSAs includes also other types of tests (e.g., for integration) [6]. To become part of the overall system, the interactions of individual microservices need to be validated and thus, integration-level, component-level and system-level testing are also crucial [7]. For instance, writing good integration test cases is ranked as the most important skill to sufficiently test microservices [8]. An emerging approach of integration testing in MSAs is Consumer-Driven-Contract testing (CDC).

***Consumer-Driven Contract Tests for Microservices: A Case Study***

Design by contract is a paradigm that aims at capturing the interactions of different software components, and formalizing them so that they can be relied upon in other phases of the design. Such a characteristic is especially helpful in the context of microservice architecture, where each service is an independent entity that can be individually (re)deployed. With contracts, testing of microservice based systems can be improved so that also the integration of different microservices can be tested in isolation by the developers working on the system. In this paper, we study how systems based on microservice architecture and their integrations can be tested more effectively by extending the testing approach with consumer-driven contract tests. Furthermore, we study how the responsibilities and purposes of each testing method are affected when introducing the consumer-driven contract tests to the system.

Consumer-Driven Contract testing [9] is a way to test integrations between services and ensure that all the integrations are still working after new changes have been introduced to the system. The main idea is that when an application or a service (consumer) consumes an API provided by another service (provider), a contract is formed between them. The contract contains information about how the consumer calls the provider and what is being used from the responses. As long as both of the parties obey the contract, they can both use it as a basis to verify their sides of the integration. The consumer can use it to mock the provider in its tests. The provider, on the other hand, can use it to replay the consumer requests against its API. This way the provider can verify that the generated responses match the expectations set by the consumer. With consumer-driven contracts, the provider is always aware of all of its consumers. This comes as a side product when all the consumers deliver their contracts to the provider instead of consumers accepting the contracts offered by the provider. In this paper our objective is to study how systems based on the microservice architecture [1,12] and their integrations [8] can be tested more effectively by extending the testing approach with consumer-driven contract tests. In particular, we are interested in how the responsibilities and purposes of each testing method are affected when introducing the consumer-driven contract tests to the system. The rest of this paper is structured as follows. Section 2 provides the background for the paper, and Section 3 introduces the case study. Section 4 presents the results of the case study. Section 5 provides an extended discussion regarding our observations. Finally, Section 6 draws some final conclusions.

In this paper, we have studied consumer-driven contract testing in the light of a case study based on an industrial system. Our experiences gained from the case study confirmed the benefits commonly associated with such tests: (i) integrations are tested in isolation by decoupling the consumer and the provider using a contract, contributing to fast and stable tests; (ii) the provider knows who are consuming its API and how; (iii) the provider can evolve based on real business needs from its consumers; (iv) the consumer can feel safe as the provider tests always catch breaking changes to the API; and (v) contracts can work as a tool to improve communication between different development teams. Furthermore, our experiences suggest that the consumer-driven contract tests can replace integration tests as they caught all the defects from the integrations that were implemented in the case study. In that light, it can be safely said that consumer-driven contract testing is a viable addition to testing strategies used to test integration-heavy systems, especially those based on microservices.

***Contract-Based Testing for Web Services - (refer whole papers’ sections)***

***Creating a Framework for Consumer-Driven Contract Testing of Java APIs - (refer whole papers’ sections)***

***Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models – whole research important***

Microservice architecture has become increasingly popular due to its good maintainability, scalability, fault tolerance, and extensibility. In addition to the REST style of microservices that has been widely used, the event-driven style of microservices is also gaining more and more attention. However, the current software testing methods have little support for event-driven architecture, and the technical complexity of event-driven microservices has further increased the difficulty of testing. In this regard, we propose a software testing tool for event-driven microservice systems called CCTS (Composite Contract Testing Service). By combining consumer-driven contract testing and the event-driven state model, CCTS records the state transitions of event exchange between services, and automatically retrieves the possible transition paths among services. Simultaneously, CCTS analyzes the event logs from the target system to determine whether the event logs conformed with the specified transitions of states and retrieved paths. Besides, CCTS checks the validity of contract testing to ensure that the communication through services. To evaluate CCTS, we conducted functional testing for CCTS using a real-world microservice system. The results show that CCTS can effectively detect potential defects in the event-driven microservice system, such as isolated states, cyclic states, incomplete contract tests, and unqualified event sequences.

