Testing Microservices, Challenges, Types and NodeJS Solutions

Chrysoula Dikonimaki, Abdullah Abdullah

May 2022

1 Introduction

In software application development, microservices are a style of service-oriented architecture (SOA) where the app is structured around interconnected services. The microservices architecture is gainiing popularity and is now one of the most widely used architectural styles nowadays. Many big companies, such as Netflix, Amazon and LinkedIn, have migrated from the Monolithic Architecture to the microservices Architecture [1]. DevOps practices goes hand in hand with microservices, providing huge benefits in terms of greater flexibility, scalability, continuous development and reliability. However, adopting a new architecture always comes with challenges and testing microservices efficiently is one of them as it involves different types of testings. For each type of test there are a lot of tools and solutions available to choose from and that is what makes this process challenging. Similar is the case for NodeJS platform where different solutions are developed to carry out the testing for microservices.

2 Monolithic to Microservices

2.1 Monolithic vs Microservices

Monolithic Architecture is a traditional software architecture which has a single code base the includes multiple services. [2] This means that practically all functionalities that we want to have in our system exist in a single code base. Microservices split those functionalities in different code bases so the result is having a single system which consists of smaller ones. What we actually do when we create a system like this is that we apply the single-responsibility principle [3] at the architectural level.

2.2 Advantages

Compared to its predecessor, the monolithic architecture, microservices are hands down more beneficial. You don't need to stuff all software components

and services in one big container and pack them tightly. Some of the benefits that comes with microservices are maintainability, reusability, scalability, availability and automated deployment [4]. Managing and maintaining microservices is easier as compared to monolithic as they compose of smaller and independent components in a system and hence making any changes to individual components does not leads to deployment of the whole system again, but only that individual component can be deployed independently. This way, all different microservices in a system can be implemented using different technologies and programming languages by different teams and the integration part is then handled by the communication practices in microservices architecture.

3 Microservices Testing Challenges

3.1 Introduction

Microservices architecture with its number of advantages discussed also comes with some number of challenges that adds the complexity to designing your system with microservices as compared to designing it with the monolithic architecture. How to decompose a monolithic application into smaller is one of them [4]. Determining each microserivce size, framework for integration and testing your microservices are some of the very important challenges which arises from changes on the interactions among services [5].

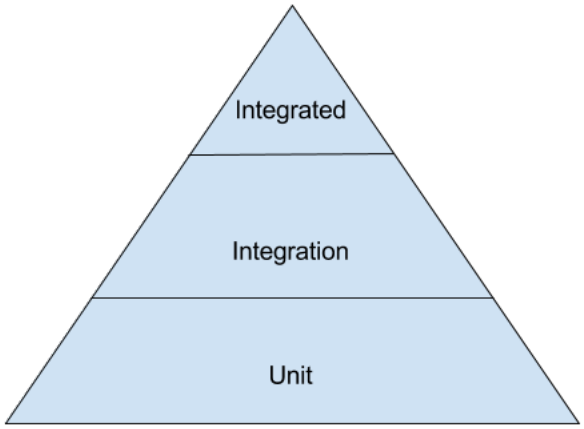
3.2 Pyramid to Honeycomb

According to the article on microservices testing by Spotify [6], the traditional testing pyramid changes when we move from traditional architectures, like monolithic, to microservices. *Figure 1* shows the traditional testing pyramid. The traditional pyramid shows two important characteristics, the first one is that tests are written with different granularity and the second one is that the more high-level you get the fewer tests you should have.

This way of testing was extremely efficient way to organize tests for long time but Spotify explains why this idea no longer works in the microservices world. Writing tests for microservices using the testing pyramid can be harmful because the complexity for a microservice does not lies within itself but the complexity is how they interact with each other which is an important part of microservices design. Having many small unit tests is not necessary for each service as it restricts how the code changes and the end result is changing the tests which does not allow to make changes quickly and has a negative impact on the overall testing process.

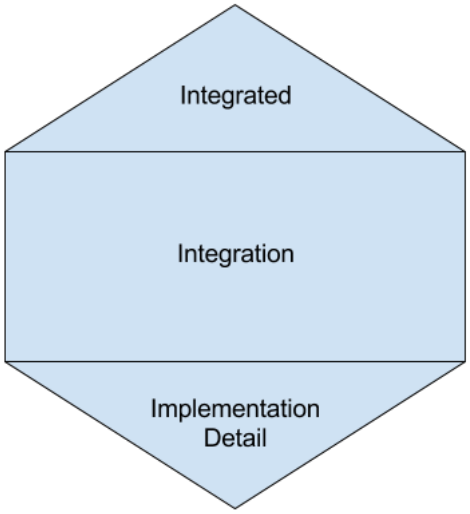
Spotify introduces the Honeycomb solution for testing their microservices (*Figure 2*) where it focuses on testing the integration part of the microservices which verifies the correctness of the communication among different microservices within an application. The focus is shifted towards integration tests and some implementation detail tests and the ideal situation is to have very few

Figure 1: Tradition Testing Pyramid from Spotify [6]



or none integrated tests. Each microservice in this case is considered as an isloated component tested and so for that reason implementation details tests are actually renamed to unit tests.

Figure 2: Microservices Testing Honeycomb from Spotify [6]



4 Testing Techniques

4.1 TDD vs BDD

Test-driven development (TDD) is a software development practice that focuses on creating unit test cases before developing the actual code. It focuses on testing small pieces of code work as expected.

On the other hand, behavior-driven development (BDD) is an agile software development methodology in which an application is designed around the behavior a user expects to experience when interacting with it. So BDD is designed to test an application's behavior from the end user's standpoint.

4.2 Stubs vs Mocks and Test Doubles

Both stubs and mocks are categories of test doubles. A test double is an object that stands in for a real object in a test. A test double allows us to decouple the application from the dependencies when testing the system under test. The term mock is commonly used for both stubs and mocks but in reality they are very different. Stubs are simpler as they hold data defined when it is created and uses them to answer the calls it gets. The main reason we are using them is because we do not want to answer with real data or the call may cause side effects. On the other hand, mocks simulate the behavior of other services that are not easy to be tested and we can verify that all the expected actions have been performed. Developing mocks and stubs are very important for testing microservice because they will represent the other microservices that are in development phase.

5 Testing Types

5.1 Unit Testing

Unit testing is a type of testing that we use to test the smallest testable parts of an software application which are called Units. We test that the code of the Units works as expected so we verify the correctness of the code. A unit can be almost anything but usually it is a method or a class.

The structure of a unit test is usually the following: Data set up, Call, Assert. Other people describe it also as: 3A (Arrange-Act-Assert) [7] or Given-When-Then [8].

When testing Microservices, you should use both Sociable and Solitary tests depending on the case. Solitary unit tests use mocks or stubs to simulate the other services while Sociable unit tests the services that are actually collaborating with the other services. However, it is good to have Sociable tests only to test the Domain logic. [9]

It is important to remember that when it comes to microservices, Unit testing does not provide guarantee about the behaviour of the system and we cannot be sure that the system works properly because our system is a set of microservices.

Even if all microservices work well independently, the combination of them defines the whole system so we need to test that they work all together as expected.

5.2 Integration Testing

In Integration testing different modules, in our case microservices are tested as a combined entity. These types of tests verify the interactions and their purpose is to detect interface defects.

We decide on subset of microservices to verify that they operate as expected together to achieve a business logic goal and also check that the way a microservice interacts with another one is correct and are not executing another service in some other way.

It is clear, that even if Integration tests and Sociable Unit tests sound similar their focus is very different, while the first ones test subsystems, the second ones test each microservice at a time.

5.3 Contract Testing

Contract testing is done at specific integration points by checking each application or microservice in this case in isolation to ensure that the messages it sends or receives are equal to a shared understanding that is documented in a "contract".

For each communication, there are two microservices involved: the provider and the consumer. We have to clearly specify how they will call one another. We do it by defining an interface. When one service calls another a contract is created between them. [10] We test that both parties obey this contract.

One way to write these tests is the Consumer-Driven Contract tests (CDC tests). The team that develops the microservice will act as the consumer of the interface and write tests that are related with the functionality that they need from the provider microservice. The team that develops the provider gets these tests and implement the provider microservice based on these tests. When they pass, they are sure that they have implemented everything the consumers need [11].

If we don't do contract testing, the other way to test that the microservices work together as expected is by implementing expensive and brittle integration tests [12].

5.4 End-to-end Testing

End-to-end testing is a testing type that we use to that test the entire software product from beginning to end to ensure the application flow behaves as expected [13]. This means that we test the system as a whole. The system is treated as a black box and we test it through the GUI and the service API.

End-to-end Tests gives us more confidence that our system works as expected and they are closing the gaps that moving microservices creates. However, they

are not easy to be maintained so we should aim to reduce them to cover only the very basic functionality.

6 Solutions for NodeJS Microservices Testing

Figure 3: NodeJS logo [14]

6.1 NodeJS Microservcies

NodeJS is a back-end JavaScript runtime environment [14] used to built servers and is widely used in the industry. Developing microservices with NodeJS has some advantages and is used by companies like Microsoft, PayPal, Uber, eBay etc [15]. Some advantages are having a fast and scalable code, event-driven architecture which makes it easy to add real-time capabilities to your application and it is asyncrhonous and non-blocking.

6.2 Testing each type of Test

In this section we will analyze and compare some tools and frameworks that can be used for each type of test.

6.2.1 Unit testing: Jest, Mocha etc

There are multiple tools and frameworks that can be used to write Unit tests for your NodeJS microservices. Some of them are the following [16]:

- **Jest** Jest is a framework developed by Facebook [17]. The main advantages of using it is that it is very simple and well documented. Moreover, it is fast since it supports parallel execution of the tests. Last but not least, it has a lot interesting features to be explored like test watching, coverage, and snapshots.

- **Mocha** Since Mocha is the oldest framework it has been used widely by many companies [18]. The main advantage is that it can be extended and customized easily. Some good features that it provides are the following: callbacks, promises, and async/await.
- **Chai** Chai is a Behavior Driven Development (BDD) / Test Driven Development (TDD) assertion library that can be combined with any Javacript testing framework [19]. It's main characteristic is that you write assertions similar to sentences.
- **Jasmine** Jasmine is a Behavior Driven Development (BDD) framework. It is flexible and compatible. [20].

6.2.2 Integration Testing: Mocha

Mocha can also be used for Integration Tests. The reason why it is useful for these types of tests is that we can make asynchronous calls to the microservices and take advantage of the callbacks and promises features that it provides.

6.2.3 Contract Testing: PactJS

Pact is undeniably the most popular framework for contract testing nowadays in the javascript world. One of the main reason is that it is Consumer-Driven contract testing tool.

A Contract is a collection of interactions. We can have 2 types of interactions: HTTP and messages [21]. Each HTTP interaction describes: the expected request and the minimal expected response. Each message interaction describes the minimal expected message.

The logic that Pact follows is: "assuming the provider returns the expected response for this request, does the consumer code correctly generate the request and handle the expected response?" [21].

6.2.4 End-to-end Testing: Cucumber, Selenium

The main difference between Cucumber and Selenium is that Selenium can be used for web-based applications, which means that it automates browser, while Cucumber can be used for any software. That's because Cucumber is not a testing framework but a tool for Behavior-Driven Development that is used to develop test cases for the behavior of software's functionality. So its role is mostly supportive.

7 Conclusion

The fundamentals of testing in microservices are not something new as compared to the traditional service oriented architecture testing but the importance of doing such tests has only become more critical and required in modern systems as building microservices architecture based applications is immensely supported

by using software testing fundamentals. With the increasing interest in the development of microservices, it is important to systematically identify, analyze, and classify the approaches, tools, and challenges in the process. When it comes to building JavaScript microservices, NodeJS platform is widely used and so there are many tools available to test and analyze the Javascript Microservices effectively.

References

[1] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, 2015.

[2] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154, 2018.

[3] Single-responsibility principle. `https://en.wikipedia.org/wiki/Single-responsibility_principle`. Accessed: 2022-05-02.

[4] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475. IEEE, 2017.

[5] Lianping Chen. Microservices: Architecting for continuous delivery and devops. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 39–397, 2018.

[6] Testing of microservices. `https://engineering.atspotify.com/2018/01/testing-of-microservices/?fbclid=IwAR0-Hra9Z3ZyGRmCuxihRrGJmAufWbu5pZM1HMalaqu8Tk7NbflcHLL6PG0`. Accessed: 2022-04-29.

[7] Arrange-act-assert: A pattern for writing good tests. `https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/`. Accessed: 2022-05-02.

[8] Givenwhenthen. `https://martinfowler.com/bliki/GivenWhenThen.html`. Accessed: 2022-05-02.

[9] Testing strategies in a microservice architecture. `https://martinfowler.com/articles/microservice-testing/`. Accessed: 2022-05-02.

[10] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. Consumer-driven contract tests for microservices: A case study. In Xavier Franch, Tomi

Männistö, and Silverio Martínez-Fernández, editors, *Product-Focused Software Process Improvement*, pages 497–512, Cham, 2019. Springer International Publishing.

[11] The practical test pyramid. https://martinfowler.com/articles/practical-test-pyramid.html. Accessed: 2022-05-02.

[12] Introduction - pact docs. https://docs.pact.io. Accessed: 2022-05-09.

[13] What is end-to-end (e2e) testing? all you need to know. https://katalon.com/resources-center/blog/end-to-end-e2e-testing. Accessed: 2022-05-02.

[14] Node.js - wikipedia. https://en.wikipedia.org/wiki/Node.js. Accessed: 2022-05-09.

[15] How to build a microservices architecture with node.js to achieve scale? https://www.cuelogic.com/blog/microservices-with-node-js. Accessed: 2022-05-09.

[16] Unit testing of node.js application. https://www.geeksforgeeks.org/unit-testing-of-node-js-application/#:~:text=Unit%20Testing%20is%20a%20software,In%20Node. Accessed: 2022-05-09.

[17] Jest: Delightful javascript testing. https://jestjs.io. Accessed: 2022-05-12.

[18] Mocha - the fun, simple, flexible javascript test framework. https://mochajs.org. Accessed: 2022-05-12.

[19] Chai. https://www.chaijs.com. Accessed: 2022-05-12.

[20] Jasmine documentation. https://jasmine.github.io. Accessed: 2022-05-12.

[21] How pact works - pact docs. https://docs.pact.io/getting_started/how_pact_works. Accessed: 2022-05-09.