Transition from monolith to microservices: a dream or a tester's nightmare?

Natalja Pletneva

nspletneva@gmail.com

Abstract

The changing and developing business needs of the IT industry require that technologies adopt and adjust themselves to meet their demands and, at the same time, allow advancement of new techniques and fundamental methods of architecture in software design. The result of such activity is a transition from monolith to microservices and building a definitely new architecture.

Today microservice architecture is widely implemented in various areas of software development. However, the changes in the testing process often lack the attention they deserve.

Microservice Testing is a new idea which changed the testing approaches and prioritization of testing levels. Moreover, it changed the working culture and the way teams work together.

The paper illustrates effective methods that can be applied to overcome challenges while testing applications designed with microservices architecture, as well as the main opportunities that arise when testing microservices.

Biography

I am a QA Engineer at Okko, a premium VOD service in Russia. I have been testing the service on different platforms such as Smart TV, Android, iOS, and PlayStation. Lately, I have been focusing mainly on the backend testing, functional automated testing and web applications. Along with that, I have got a Master's degree and wrote a thesis on Operation Support System/Business Support System in IoT.

1 Introduction

Every more or less successful product comes to a state where adding new features to an existing code base becomes so complicated that the cost of new functionality exceeds all possible benefits from its use. The most popular approach to solving this problem at the moment involves splitting one large piece of code into many small projects, each of which is responsible for its specific functions. When creating such a system, it is necessary to think ahead and develop a type of interaction between these separately working pieces so that future changes would require minimal development and testing costs.

There are a lot of expert articles on the web that show us the importance of such an approach and talk about which architecture can be called good and which not so good. Of course, every article or method like those is logical and consistent, and especially if the described method is proved out in practice. Nevertheless, the issue of testing when switching to microservices and the changes in the test process often do not receive the attention they deserve.

2 Monolith architecture. Challenges in testing monolith

2.1 Monolithic applications and their problems

In the classic approach to software development, monolithic application architecture is used as a single unit. When developing applications, as a rule, a three-tier client-server architecture is implemented. In this case, the application consists of three main parts (Fig. 1): the client part (User Interface), a database (Database) and the server part (Business Logic Layer and Data Access Layer). The server part of the application processes HTTP requests, performs business logic, receives and modifies data in the database, and also selects the necessary HTML pages and data for display and sends them as an HTTP response to the client (browser). This server-side application is a monolithic structure, and any changes in it require deployment of a new version of this application.

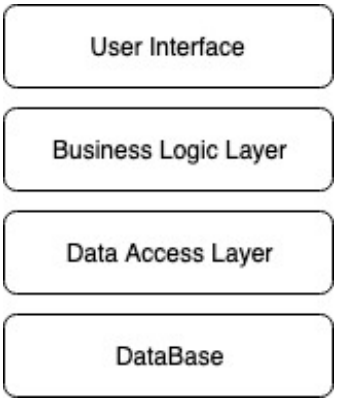

Figure 1. Monolithic application architecture.

We all know that, by making changes in one place of the monolithic program, you can cause damage in other parts of it.

Monolithic architecture means that your application is a large module where all components are designed to work together with each other, shared memory and resources.

The fact is that the components in the monolith can have very complex and non-obvious relationships. Classes calling methods, receiving other classes as inputs, generating new classes that need to be packaged into functions.

This architecture works fine for small and simple applications. But let us imagine you want to improve the application by adding new services and logic to it. Perhaps you even have another application that works with the same data (for example, a mobile client). But monolithic application can only be changed by scaling out horizontally (Fig. 2) by running several instances of it using load balancers. The application architecture will change a bit:

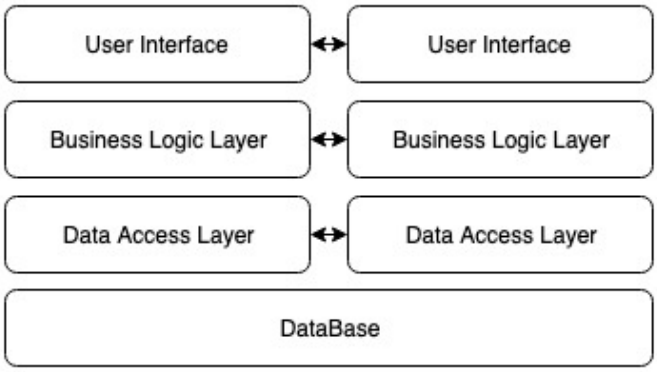

Figure 2. Horizontal scaling of a monolithic application.

In any case, as the application grows and develops, you encounter problems with monolithic architectures:

- Support is more and more difficult.
- In such an architecture, it is always tempting to reuse a class from another module, rather than take data from the database. Thus, an extra connection arises, which should not be the case in the original architecture.
- It is difficult to understand monolith, especially if the system passed from generation to generation, the logic was forgotten, people left and came, but there were no comments and tests.
- If any of the components stops working for any reason, then the entire application will collapse as well.
- Monolithic systems just require more complex environment (when the whole system has to be compiled, built and ran).
- It is expensive to make any changes.
- Since you have to mount everything in one place, this leads to the fact that switching to another programming language or switching to other technologies is problematic.
- And one of major problems in monolithic architectures is that they are hard to scale. Normally you have an option to increase the computer power, but not scale horizontally. For instance, you cannot scale just a particular module. This is one of the reasons why microservices became so useful with highly scalable applications.

Sooner or later you realize that you can no longer do anything with your monolithic system. The customer, of course, is disappointed: he does not understand why adding the simplest function requires several weeks of development, and then stabilization, testing, etc. Surely, many faced these problems.

2.1.1 Example of monolith testing problems

Imagine that our monolithic application is an IVoD (Interactive Video on Demand - a video content delivery system that allows viewers to select content (video) and view it at a convenient time (upon request) on any device designed to play video (web, tablets, smartphones, game consoles, etc.). It is an application in which the user is offered a huge catalogue of movies for watching in different qualities for different rights: EST (Electronic Sell Through), SVoD (Subscription Video on Demand) and TVoD (Transactional Video on Demand).

This project is characterized by a huge number of overlapping business scenarios, movies can be watched on different platforms (TV, mob), and gifts can also be offered to users (movies for free, trial periods).

In this case, we meet the following obstacles:

- Data. Obtaining test data is a separate scenario due to a large number of combinations of input parameters. To create a crystal-clear user for a script, you first have to run it through the entire system, which requires more time and more expertise. Of course, there are workarounds (changes through the database), but from a testing point of view, this is not legitimate.
- Performance Testing. Maintaining a separate environment for stress tests is quite expensive, and the load of the test environment can negatively affect the work of colleagues.
- Bug Analysis. Analysis of the bugs is difficult because we have one monolith, one end point, and it is quite difficult to understand which module the failure occurred in. Thus, the life cycle of the bug is increased.
- Test Automation. The monolith cannot be dismantled and not tested, therefore the first question that arises at the beginning of the autotest path on the monolith is: where to start? Due to the strong interconnection in the monolith, writing “clean” unit tests will not work since you still have to use additional data from other blocks, make stubs and resort to other tricks. Thus, the cost of tests grows and the time to maintain them increases. And what if a new specialist comes to the project? Not only will he have to figure out the monolith, but he will also not be able to understand the whole value of the autotest code. It seems to be very expensive.

These inconveniences led to the development of architectural style of microservices: building applications as a set of services. In addition to the ability to independently deploy and scale, each service also receives a clear physical boundary that allows different services to be written in different programming languages and tested independently. They can also be developed by different teams.

3 Microservice madness

It is not surprising that the video service application presented above also encountered the main difficulties of the monolith. Business ideas coming up constantly and introduction of new features resulted in grow of monolith, and adding a new feature started to seem not worth the potential benefits of the feature itself (Fig 3).

In view of all these difficulties, it was decided to create a set of new services grouped in several servers, in which there will be separate APIs for working with all business functions - the so-called transition to microservices, which solves the main problems of the monolith:

- Less code for one service.
- Implementation independence.

- No single point of failure.
- Independent deployment.
- Independent testing.

3.1 Architecture “before”

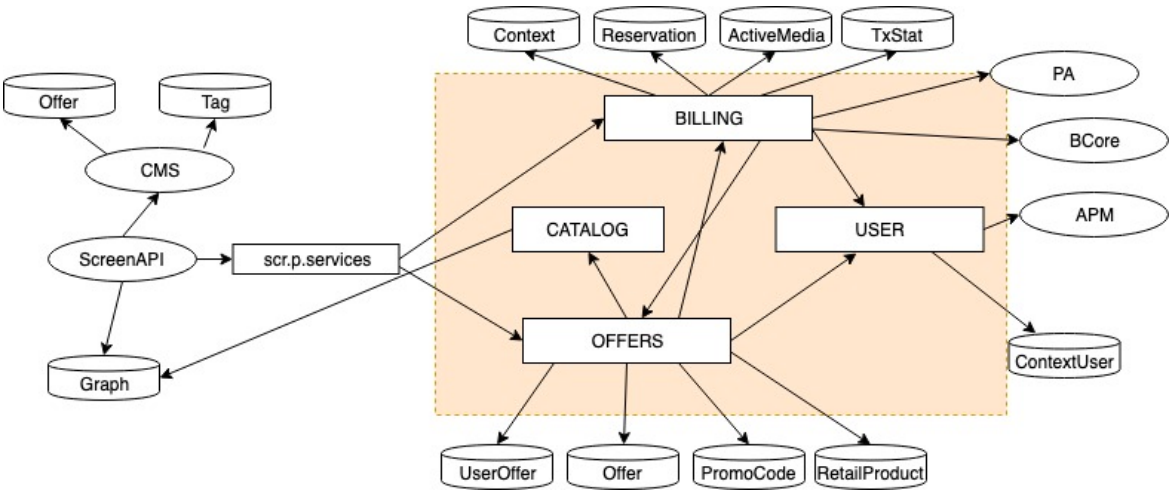

Figure 3. Architecture “before”.

There are obvious problems with the monolithic architecture scheme presented above (Fig. 3):

- Strong interconnection between services.
- If the services are just divided into separate applications, the number of network operations is too high, and transactions fall apart.
- Often different services get the same information - this is inefficient and, moreover, makes services “all-knowing”.
- A lot of unnecessary checks, performance may suffer.
- The life cycle of bugs is increased because it is difficult to analyze any problems.

3.2 Architecture “after”

Microservice architecture is an approach to creating applications, meaning the rejection of a single, monolithic structure (Fig. 4). That is, instead of executing all the limited contexts, several small applications can be used, each of which corresponds to some limited context.

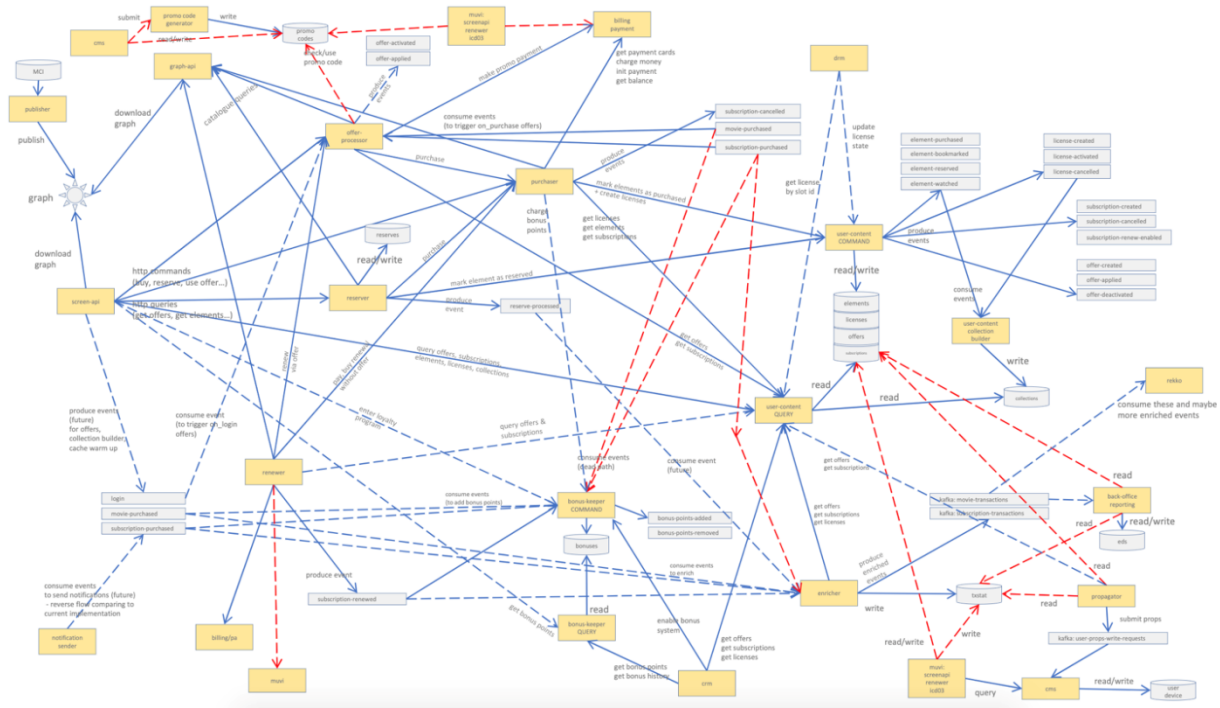

Figure 4. Architecture "after".

Another view on the microservices system is where every component is connected to a message bus (kafka/rabbit/etc) and sends and receives messages from it (Fig 5). It is a less scary image of the diagram above.

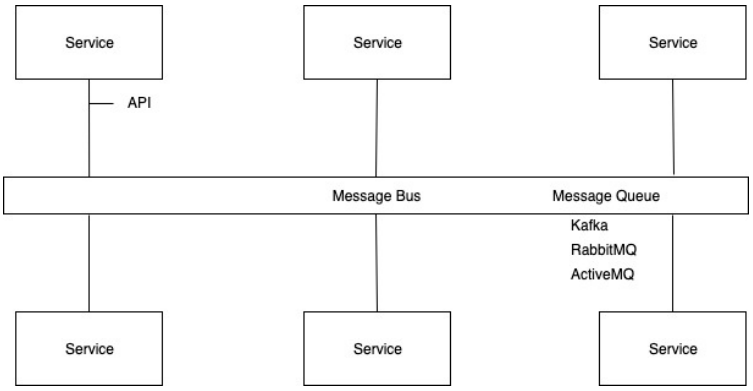

Figure 5. The diagram of a messaging system between microservices.

Microservices were gradually separated and new functionality was written independently from the main application. The new components provide:

- Independence of change.
- Testing independence.
- Independence from any technologies.
- Isolation of business logic and work with the database from other components (since one component now communicates with the database separately), which in future will allow moving to any other database particularly.
- Ability to optimize the execution time of complex scenarios.
- It is also possible to scale in/out every component of the system individually.
- Microservices allow for separate (isolated) testing, when the rest of environment is fake.

In the case of a monolith, in order to replace any piece in this system, we cannot roll it out separately. You need to rebuild and completely update the backend. This is not always rational and convenient. What happens when switching to microservice architecture? We take the backend and divide it into its component components, dividing them by functionality. We determine the interactions between them, and we get a new system with the same front-end and the same databases. Microservices interact with each other and provide the same business processes. For the user and system testing, everything remained as before, the internal organization has changed.

What would be the point? You can make a change in the microservice X, which implements some of its functionality, and immediately roll it out to work. It all sounds very good. But, as always, there is one catch: you need to check how microservices interact with each other.

As a result, for testing, we get a number of services, each of which can pull the rest, and which exchange the data randomly:

- A set of functionally complete services.
- Interaction between them occurs either synchronously via http/https or asynchronously through the manager, if a mixed type of architecture was selected, when services interact either directly; or via a messenger, when a component subscribes to a message queue and listens to it.
- Standard messaging interface for all services.

4 How to test it?

Expectation:

Today, there are several ways to structure tests on microservices: for example, a honeycomb test that focuses on integration tests. This is the theory of moving to a higher level in the test pyramid, which also contains the priority of integration tests. A lot was mentioned about the test-division approach on levels - modular, contract, integration, functional and end-to-end - a huge number of ideas and proposals. It sounds excellent, but in practice, not everything turned out to be so easy.

Reality: «situation of absolute mess is normal: all tests pass, but nothing works».

Based on the definition of microservices, it may seem that microservices for testing is a great idea, but things need to be clarified a bit.

Further I will discuss some basic misconceptions and pitfalls of the transition to microservices so that you could take them into account when you decide whether microservice architecture is right for you or not.

4.1 The Illusion «It is Easier»: distributed transactions are never easier

Every time a new technology appears, there is a feeling that now we have to study everything from the very beginning. But this is usually not the case. For example, when mobile applications appeared, almost all the technologies that we knew before remained unchanged. We had to deal only with the new client part, in connection with the advent of new operating systems and programming languages. And when the Internet of Things appeared, in addition to hardware and embedded code, the server part also changed a little - new protocols and new ways of processing large amounts of data in real-time appeared. But when it came to be building a microservice architecture, really serious changes appeared at once on many levels, and not only with regard to technology.

Starting the transition to microservices, the first basic thing that will have to face is the separation of the old monolithic code into small services and placing them into an architecture already familiar to us (for example, SOA). Dividing code is not too difficult. Difficulties begin when setting up load balancing, restoring, and automatically scaling these services, but the greatest difficulties can arise when testing such new applications. Here, new approaches and testing tools are required, as well as a lot of expertise and knowledge in the field of microservices.

On the surface, everything may seem simple: you have clearly defined inputs and outputs for the API that the service should provide - what could be the difficulty?

When several remote services are involved in a single request, the complexity greatly increases. Can I refer to them in parallel or do I need to apply sequentially? Are you aware of all possible errors in advance (at the application level and at the network level) that can occur at any time, at any part of the chain, and what will these errors mean for the request itself? Each of the distributed transactions often requires its own approach to error handling. And to understand all the mistakes and how to solve them is a very, very big job.

Before you start testing microservices, you need to determine approaches and the number of tests, based on the existing architecture and application features. Based on the experience of testing the video delivery application discussed above, I would like to highlight the main points in testing microservices:

4.1.1 Unit testing

For the most part, the developers are responsible for the unit tests. They make sure that the individual unit of the code base (test subject) is working properly. You can also use imitations (mocks) or stubs, which, instead of implementation, give out believable data, simply hard-coded. In our case, the tester acts more as a reviewer. And the unit tests are not enough here since errors are maybe at the junction of services.

4.1.2 Integration testing

The basis of our test suite consisted of integration testing. During integration testing, the correctness of interaction of various microservices with each other is verified. This is one of the most crucial tests of the whole architecture. With a positive test outcome, we can be sure that the whole architecture is designed correctly and all independent microservices function as a whole in accordance with expectations.

4.1.3 You have a lot of services - you have a lot of APIs. What is the problem with the API?

An API is not just code that is accessible to an external customer, it is a large number of supporting tasks that need to be addressed, and the code that stands behind the API is not the biggest problem. You will need:

- Documentation with examples of how to use your API, including a comprehensive description of all possible errors.
- Means of validating requests (declarative schemes, at least for oneself, at most - publicly available).
- Tools for testing and experimenting with APIs - isolated environments, sandboxes for debugging.

If you have one application, then the API publishing point is about the same. If you have a service architecture, each service requires all this infrastructure support. If each service is created using different technologies, your problems have just tripled.

4.1.4 A hybrid testing approach was chosen, and testing was added in production.

It is necessary to monitor the entire service, collect as much data as possible - both for analyzing the current situation and for analyzing incidents. The tester is responsible for monitoring using the modern Jaeger (distributed query tracking system), there are more research testing and user interaction.

In addition to these 4 points, you also need new tools and approaches, like system observability techniques and tools (assigning every message in a scenario a traceable id across all components) Thus, there are no fewer tests, it has not become easier: since microservices work simultaneously alone and in conjunction with other loosely coupled services, it is necessary to test each component both separately and as a part of a whole.

4.2 The Illusion «It is faster»

Each service is a separate, functionally complete unit and it has its own set of tests. Although now we do not have to emulate user actions, it is enough just to build the REST request correctly, but we live under the conditions of monolith and microservices that work together. All of this must be managed simultaneously, which gives us rise to an even greater number of points of failure, the number of tests requires more expertise.

Suppose you decide to do everything conscientiously and provide your services with tests.

With unit tests, everything is good and even better than with ordinary applications - you have clearly described inputs and outputs.

But now you need to emulate external services with which you interact - you do not want to re-raise the entire ecosystem of services after each change (moreover, if you have resorted to using separate development teams for each service, you personally may not require infrastructure / knowledge in order to raise the services of another team).

The problem is that your stubs and mocks, which successfully emulate your immediate environment, actually protect you far from as much as you think, even if you update them neatly and they always exactly match the interfaces of the services you use directly.

Imagine this situation:

Your service A goes to service B, which in turn relies on service C, which you do not directly interact with.

Service B was written by smart guys, and it knows how to handle different variants of data from service C, without noticeable damage to himself, and even give you something similar to the result. Your tests pass; on test environments with real services, services A, B, and C have no problems with each other.

But here your distant colleagues from service C roll out new formats that are successfully processed by the current version of service B, and are returned to you after being processed by service B. No one has any idea of how it can respond – but it still works.

And you (service A) - fail. All your tests pass, all tests of services B and C pass. Even pairwise integration pass. But all together it does not work - this is why system observability tools are necessary, to understand first which component exactly did not work in this scenario, to avoid browsing logs of every component individually.

At the same time, the versions of services A and B, at the moment of the interaction of which the problem occurs, have not changed.

Even if all the data formats that you exchange are covered with schemas and validators, they cannot protect from everything, and you will encounter the described problem.

Some kind of more or less noticeable confidence that everything will work gives us the deployment of a full set of services in a test environment and run of typical scenarios on a fully working system.

Even if we forget about the fact that such checks are slow, every service can be developed with different technologies, it can take a considerable amount of time to initialize or require specially configured databases. And with such data sets that will allow you to get rid of your end-to-end scenarios (these sets will need to be maintained by various teams up to date, by the way), all you know is that a specific set of service configurations seems to be working. And if you want to be constantly confident in the performance of the entire complex, you need to roll it all out together.

Thus, we spend a lot of time on interaction between teams, on finding errors, new requirements, but the worst thing is that when we divide microservices between test engineers (each tester is in charge for and understands his own field), we get an uncertain area of responsibility:

In this case, when testing microservices there can be one big global difficulty - the interaction between teams.

It requires constant synchronization with the teams. Errors can live simultaneously in several services and require changes at the level of several teams - synchronization and coordination of efforts are necessary. In addition, people may lose their sense of responsibility and try to push as many problems as possible to another team.

4.3. The illusion «This is better for testing»

We encounter some key issues when testing the microservice architecture:

4.3.1 Team coordination.

As many independent teams manage their own microservices, it becomes very difficult to coordinate the entire software development and testing process.

4.3.2 The complexity.

There are many microservices that communicate with each other (Fig. 6). We must make sure that each of them works properly and is resistant to slow reactions or failures from other microservices. You definitely need to test full system threads to see if the services understand each other, in particular when one of the services responds with error messages.

4.3.3 Performance.

Since there are many independent services, it is important to test the entire architecture under traffic conditions that are close to production. If you work with queues, many of them may have asynchronous events and timeouts that may come into work. You can try to model each of such scenarios individually, but performance tests, when many parallel threads load one of the queues with an unexpected number of messages, can reveal many errors.

4.3.4 The problem of collecting logs in one place.

To test a problem, you need to download all the involved microservices. Debugging becomes a non-trivial task, and all the logs need to be collected somewhere in one place. At the same time, you need as many logs as possible to figure out what happened. To track the problem down, you need to understand the whole path that the message went. Unit tests are not enough here since errors are likely to be at the junction of services.

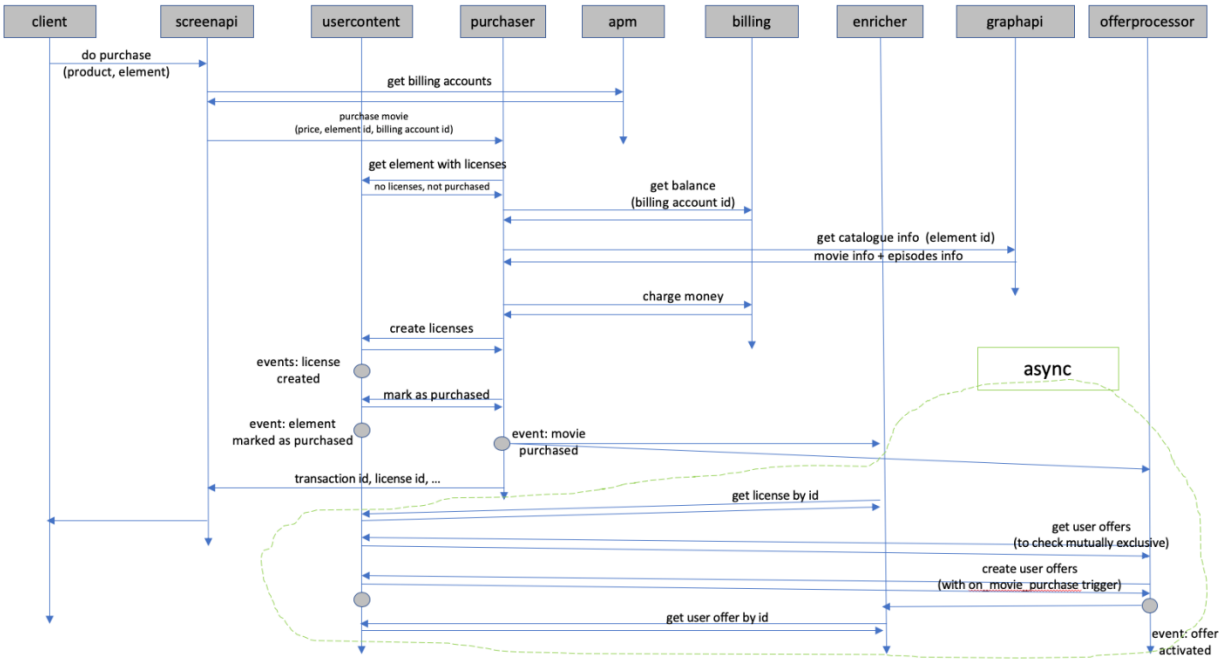

Figure 6. Sample scenario: movie purchase.

Automated testing plays a very important role in the modern software development process. It allows you to always maintain the working of the product, make changes and refactor. The microservice approach adds its difficulties to the testing issue since it is necessary to verify the correct interaction of microservices under various working conditions. For full testing of microservices, it is necessary to deploy a complex testing environment on special servers, which requires large human and material costs.

Many scenarios are easier to automate at unit testing level, where you do not need to start an HTTP server to host a service or service mock. This is particularly the case for negative scenarios. Usually, we can work closely with developers of such services, and every time QA finds that automating a scenario is too expensive at the system level, we can ask developers to add a test to their unit testing suite.

5 Conclusion

All considered illusions do not detract from the obvious advantages of microservice architecture. Consider an example.

When we came up with the idea of switch to microservices, it was decided to first separate the billing and the offers granting service - the so-called free periods of using the service.

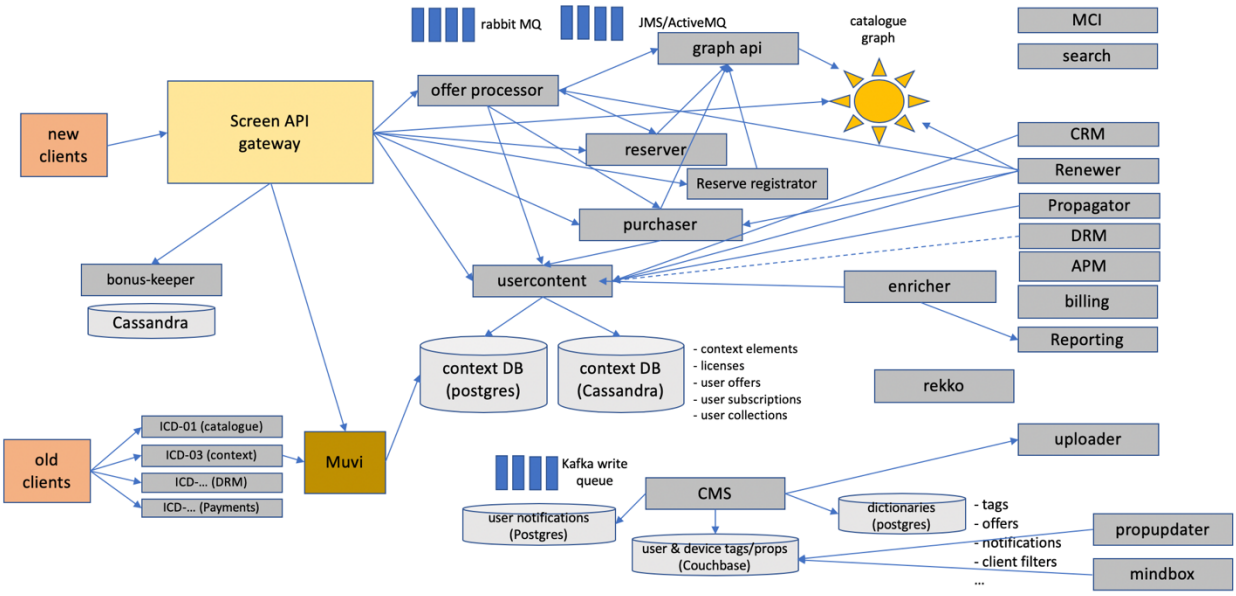

Figure 7. Switch to microservices: first iteration.

To output the component data from the monolith, it was necessary to isolate a couple of services, and in the end, we got this (Fig.7):

Graph API:

- provides GraphQL API for catalogue graph
- listens to publication event from publisher
- downloads catalogue graph and keeps it in memory
- if your app cannot load graph in memory, use this service

User-content:

- proxies read and write access to user entities (context):
    - user offers
    - user subscriptions
    - licenses & context elements (purchases, bookmarks, etc.)
    - user collections
- dumb service, CQRS (command-query) pattern
- supports Postgres and Cassandra implementations
- produces events, but nobody consumes them yet but itself
- does not interact with other services

Purchaser:

- wraps movie/subscription purchase logic
- handles purchase/pay requests of main API(ScreenAPI) and offer-processor
- works with Graph API (to get info about prices, etc.)
- works with User-content (checks/creates licenses, subscriptions)
- works with billing/APM (checks balance, linked cards)

Offer-processor

- wraps offer activation and offer usage logic
- also handles promo code activation requests
- works with user content
- listens to purchaser pay events to activate specific offers
- listens to the main API (ScreenAPI) events (login/register) to activate offers
- creates user notifications when offer is activated

Business logic was isolated from the monolith, but, of course, not removed. Everything seemed to be tested. Both the monolith and the microservices included the A / B split (using the CMS internal split service) and worked on the test environment.

Initially, the path of individual component testing was chosen: each microservice was covered with unit tests, plus a contract test of the interaction of microservices inspired by trends was conducted. Of course, no one forgot about integration, but more attention was still paid to the integration between monolith and microservices, and not between different microservices. The main task was to bring microservices to the monolith without harm.

On the appointed day X, everything was implemented in the production environment. And it lasted only five minutes. Because such promising microservices did not go off the first try, and not at the second, and even not at the third.

It was a great fiasco experience, from which we drew certain conclusions:

- Contract testing is a lot of sizzle and no steak, because in practice, we have not found a long-lasting good solution here. Although in some cases it certainly works.
- Sometimes the test environment does not coincide with the production environment for various reasons: from the configuration of hardware to the load. Therefore, it is necessary to carefully consider the model of the load on services (and not to repeat our mistake).

- To test microservices, a higher-level approach is needed, more system testing, as well as end-to-end tests.
- Each service should fall as painlessly as possible for the system as a whole and scale separately: what really worked. The fallen part of microservices did not destroy the entire service. Chaos Monkey testing was successful.

Microservices provide flexibility in choosing the type and level of testing that can be used since microservices can implement some central business functions or something small, when the level of testing may be lower. But before moving on to microservices, you need to think about all the possible details and issues in order not to face any problems in the future. In testing, there are pros and cons of this approach, so you need to take into account the current situation in the team and be prepared for huge amounts of work. Although we still live under conditions of monolith and microservices, now I can say that the transition experience may be very positive.

In our case, microservices are a dream and salvation: it is an opportunity to organize and structure testing processes, strengthen automation and, eventually, conduct testing faster and better.

Experience says: you cannot fight trends, but you can fight fluctuations. Let your choice of architecture be the personal trend, based on the requirements of the system, and not the point's position on the technology's hype-cycle.