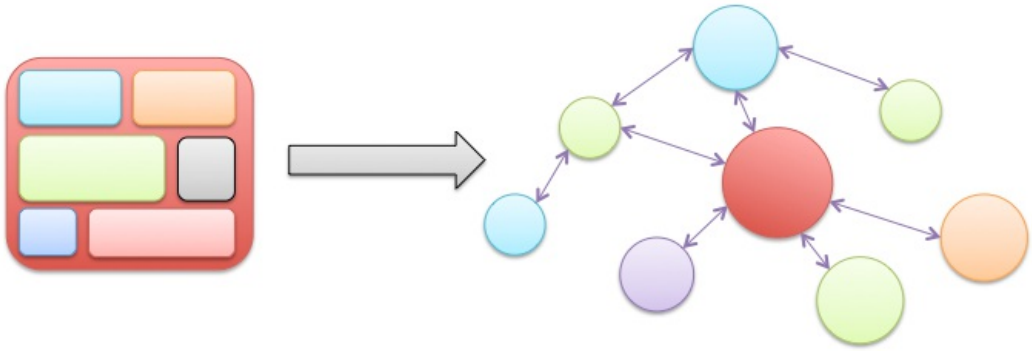
JULY 25, 2016

Microservices testing

Modern software engineering is all about scalability, product delivery time and cross-platform abilities. These are not just fancy terms – since internet boom, smartphones and all this post-pc era in general, software development turns from monolith, centralised systems into multi-platform applications, with necessity of adjust to constantly changing market requirements. Microservices quickly becames new architecture standard, providing implementation independence and short delivery cycles. New architecture style creates also new challenges in testing and quality assurance.



In this article I would like to outline strategies for testing microservices architecture. Most important thing is to understand difference between specific testing layers and their complementarity.

Why microservices?

What's the architecture of typical, corporate IT system in sectors like financial services, insurances or banking? Typically it's monolith-like architecture, based on single, relational database and SOAP API. One system is responsible for many domains and contexts. Those kind of systems have been working on production for years, and big companies are not willing to risk and change their proven-in-battle architecture. Although, the market requirements are constantly changing, so are technical requirements. Monolith architecture falls short when it comes to scalability or continuous deployment, not to mention technology stack commitment or single point of failure.

Here comes *microservices*! Are they a silver bullet for all your architectural problems? Well, no – they also have their own issues. What sets them apart though, is ability to quickly adjust your architecture to constantly changing requirements through independent development, deployment and scaling.

In simple words, microservices are small, independent applications with single-domain responsibility. Those services communicate between each other through HTTP, usually with use of REST protocol.

Microservice architecture is not just a new buzz word. World leading tech companies, like Amazon, Netflix or Twitter base their key system architecture on microservices. Even more significant is that not only new systems are build with use of microservices, but companies also rewrite their old, proven solutions into new architecture.

Testing strategies

New approach to architecture requires new approach to testing and quality assurance. Focusing mainly on automated testing, we can divide specific layers of tests. Important thing to notice here is that working with single layer is insufficient, and due to their complementarity, you should provide all of them to your project.

unit tests – those are nothing new, and nowadays almost everyone understands advantages of TDD approach. In general, unit tests are automatic checks that provide us a feedback whether our implementation complies with the requirements. Unit tests check only system components in the smallest pieces, like single methods, with other dependencies mocked.

Unit tests are great when it comes to fast, nearly continuous feedback of our implementation correctness in smallest granulation – methods or single system components.

integration tests – as we said, unit tests check single components in isolation. When *method A* is dependent on *method B*, we would mock *method B* in unit test of *method A*, since we want our test to be deterministic and single-responsible. Although it's desirable at this level, we can have a situation where two system components, working correct in isolation, don't meet functional requirements when integrated together.

There we introduce *integration tests*. Those are ones that check a number of system components or methods working together. For example, if we want to test an endpoint at integration level, we'd send a request, see if service layer done its job and validate a response.

Remember that we are in microservices world and our system functionalities depends on few applications communication. So, if the endpoint that we're testing triggers client call to external service, should we let our tests be dependent of application owned by someone else? No – at the level of integration tests, we want to test only one application's components, with external services mocked out.

contract testing – microservices architecture depends on communication between services. Although internal implementation of services is independent, interface and API must remain consistent. When we design our service and expose it to the world, we define a contract for our API.

Let's imagine hypothetical situation: we're building a service for users registration. Before we store new user in our database, we check if user's ID number does't appear on fraud list. We integrate for this with external service, that we do not owned. Now, if someone – intentionally or not – breaks the contract in validation service, our registration functionality stops working correctly, although we haven't done any changes. Our integration tests where green obviously, since they work on external validation service mocks.

Contract tests are automatic checks to assure, that the contract we've agreed on is still preserved. Idea is simple, but how to do that in practice? We depend here largely on the tool that we're using. Worth to mention are tools like **Pact** or **Spring-Cloud Contract** – they provide DSL for mocking service on the client side, and interaction playback and verification on server side.

end-to-end tests – also known as functional tests, e2e tests are known and used probably even longer than unit tests. In world of microservices, they are much more complex subject though, since our functionalities are based on integration between many application and services. We do not want to mock anything at this level, so our tests are dependent on the state of specific applications. Furthermore, in distributed architecture we change system calls, known from monolithic applications, to network calls. This comes with all the network issues, like timeouts or packets loss. Last but not least, debugging tests with lots of dependencies to external services can be challenging.

Nevertheless, end-to-end tests are crucial for our projects quality. To minimize our e2e testing efforts in microservices architecture, we should follow well know, yet rarely kept rules:

- First of all, functional tests are ones that gives you feedback of key functionalities from your user perspective. That kind of tests tend to be difficult to maintain and have long execution time. Therefore, the number of them should be limited. If some functionality can be tested on lower level – in unit or integration tests – it should be moved there.
- Keeping user perspective doesn't mean you should build your e2e framework with WebDriver-based libraries. Selenium has many advantages, but it falls short in terms of maintenance cost or execution time. Since you do microservices, majority of user actions can be simulated with service requests, which are not only faster but also more stable.

- Your test environments should be defined as a code. We don't want to be in situation where test results are dependent to test data or environment state. If you run your functional test suite once a day, good practice would be to build environment from scratch before test execution starts. Obviously, that should be done automatically, with use of tools like ***Ansible***, ***Chef*** or ***Puppet***.

Going beyond

Test layers I've mentioned, are just core aspects of automated testing. Having vast of our test scope automated, leaves us with time to perform complex manual testing. You can do either exploration or context-driven testing.

Another key aspect of quality assurance in microservices are performance tests. Your system architecture depends on distributed network calls, so you should put a great attention to performance of your application. There is a common question, whether you should load-test single endpoints in isolation, or whole functionalities, simulating chain of network calls. In my experience, both of approaches are important. Testing single endpoints gives you knowledge of some technical details of your implementation and lets you to profile service in isolation. Testing chain of service calls draws your attention to performance-bottlenecks of your system and lets you know where scalability effort should be put.

Continue reading

If you want to continue reading and expand your knowledge in area of REST and microservices, I recommend you these books:

- **Building Microservices** – one of the most important books for me, everything you want to know about microservices is here

- **RESTful Web APIs** – another great book about REST architecture. Lot of practical knowledge about designing and consuming RESTful APIs

- **Mastering Microservices with Java** – excellent reference for anyone willing to master microservices in Java world

Summary

Testing in microservices architecture can be more challenging than in traditional, monolithic architecture. In combination with continuous integration and deployment, it's even more complex. It's important to understand layers of tests and how they differ from each other. Putting effort on automation aspect of your tests, doesn't mean you shouldn't drop on manual testing. Only combination of various test approaches gives you confidence in product quality.

Posted in **design patterns**, **devops**, **microservices**, **REST**, **software testing** / Tagged **microservices**, **rest**, **testing** / **2 Comments**

PREVIOUS POST
PUBLIC SPEAKING: LESSONS LEARNED
NEXT POST
SOFTWARE TEST ICATION

Categories

Archives