MICROSERVICES ARCHITECTURAL STYLE

Artur CURNICOV

Departamentul Inginerie Software și Informatică, IS-231M, Facultatea Calculatoare Informatică și Microelectronică, UTM, mun. Chișinău, Republica Moldova

Autorul corespondent: Artur CURNICOV, e-mail: artur.curnicov1@isa.utm.md

Abstract. *This article contains a deep exploration of microservices in software development. It weighs the pros and cons, evaluating numerous challenges and solutions met during the implementation phase. The investigation covers analysis of the communication styles between services, offering advice on how to choose the right communication style depending on the business logic and requirements. Additionally, the article examines ways of migration from a monolithic architecture to microservices architecture, addressing crucial considerations, potential issues, and cost-saving strategies during the transition process. It provides an overview of testing challenges and solutions specific to microservices-based software, what types of testing is efficient for microservice architectural style components. Finally, the article considers how microservices based architecture influences the structure of the data used in the system as well as the team shape of the projects itself. Through examples and straightforward analysis, this article offers precious guidance for architects, developers and other technical persons and navigating the challenges of microservices architectural style adoption.*

Keywords: *microservices, architecture, software architecture, monolith*

Introduction

Microservices is an architectural style from the service-oriented architecture family, each service is modeled around a business domain. The microservice architecture consists of a network of services, each modeled around a business domain. Each microservice is a black box that hides all the inner operations, data sources and any other internal implementation details via encapsulation.

The only way to interact with a service is using its communication interface, which exposes points of interaction with the business domain, without leaking any implementation detail. This means that shared databases are used in rare cases [1]. This further helps defining a strong line between what can be easily changed (internal implementation) and what should be changed with caution and be more rigid (API layer of service). This leads to working in parallel on multiple microservices, as each is isolated as soon as each service communication interface is not changed.

Advantages and disadvantages of microservices

Being a distributed system, the microservices open a world of disadvantages, however combining the distributed systems with the benefits of domain driven design, service encapsulation and isolation, microservices bring a list of significant advantages in the game.

Advantages:

- Technology diversity - Right tools can be chosen for each independent service, meaning that different services in the system may use different technology: programming languages, frameworks, databases, patterns, low level architecture. Migrating and upgrading is easier. Performance can be increased due to right picking of technologies.
- Independent deployment - Each service is independently deployable due to its encapsulation and isolation. This makes it easier to add new or rollback features. Each

service has its own pipeline and can be deployed without affecting the work of other services and teams [2].
- Reusability and composition - Functionality can be reused with the usage of the same service. This can be achieved using the same communication interfaces or even adding new ones to the existing services. Composing multiple services' business logic, a business can have multiple applications as most of the services can be common.
- Independent scaling - Scaling each service apart from others allows to increase the resources and performance of a particular business area depending on traffic load or other constraints. This reduces costs as each service is independently scalable.
- Robustness - Due to high isolation, the failures of a service can be stopped from being cascaded to other services, so the entire system can work while one of the services is under maintenance.
- Physical team structure - Development teams can be restructured in such a way that a team is responsible only on a few services instead of the entire system. This helps in performance of features delivery and domain knowledge.

Disadvantages:
- Development complexity - Microservices introduce new complex things in the game: data consistency patterns, cloud usage, different programming languages for each service. This requires that the development team is senior enough to keep all these things workable in a timely manner.
- Cost - Each service is a separate process that needs to run, things that lead to increasing costs. Besides that, each environment requires instances for each service that increases the costs again. Pipelines running tests also need instances of services, databases or other tools such as queues.
- Data analyzation and reporting - There is more than a single database in a microservice system, meaning that there is not a single source of truth. Analyzation of data and reporting needs a merging mechanism to unite data from different services in order to produce a true report. Sometimes data can be in an inconsistent state, this requires handling and testing.
- Logging and monitoring - Each service requires its monitoring mechanism, alerting system in case of issues. Also, a logging aggregator is required to unite logs across all the services and group logs by “flows” to understand where some things may go wrong.
- Security and latency - A lot of information is going through the network instead of going inside a process, this means that security should be increased so that attacks cannot steal some information that is being exchanged between services. Also network communication against process communication adds latency, meaning that in some cases a microservice solution can be a bit slower compared to a monolithic one.
- Testing - Testing is required in all the services to ensure quality. We need to ensure that each service integrates well with other ones, we need to test the contracts of the services, and also the unit test each service. Besides that, component tests of each individual service will assure that the service itself is working properly. Another layer of testing could be performance testing including soak testing, load testing and stress testing.
- Data consistency - As there are more databases spread across microservices, ACID properties of a database are not applicable anymore at the entire system level, meaning other mechanisms are required to handle data consistency such as distributed transactions of saga pattern.

Communication styles

Communication styles are divided in:

- Synchronous - one to one communication that blocks the client until the service responds with a message. This kind of communication implies a coupling between the client and the service, because the client is required to wait for the response, thus blocking the running thread [3]. This may lead to a stagnation depending on the load. If the service is able to auto scale, it may also increase the costs on high demand.
- Asynchronous - one to one or one to many communications that does not require a response. The response can also come as an async operation, but this does not block the client from performing other actions until the response is achieved. This type of communication is less coupled, but requires more maintenance. The asynchronous communication can be implemented using queues.

Monolith to microservices transition

Transition to a microservice may be required due to several reasons: latency in release of the software, hard scalability of some areas of the application, slow delivery of the product, required knowledge across all the domain areas of the project.

The transitioning strategy depends on multiple factors. A rewrite could be useful in case the project is small and has the right number of resources for the transition. A bigger project could not afford a full rewrite due to high costs of the development team and teams around it.

A less costly strategy can be used to migrate iteratively - tactical forking. It is used to by copying the entire system into a new one and deleting the unnecessary code from the old system and the new one. This leads to a split that can be orchestrated by a load balancer. In this way, one service’s logic can be extracted into multiple services.

Another less costly strategy for migration is implementing the new functionality and features as separate services. The old functionality can be iteratively broken down into other services.

To correctly draw the bounds of the new services 2 required things should be taken into consideration: loose coupling and stronger cohesion. The services should be split in such a way that 2 or more services are not tightly coupled and the inner implementation should contain things that are changed together. By doing this, the interfaces of the services can change less, meaning that the deployment and scaling process can be run in isolation.

Testing microservices

Microservices require different types of tests at different levels to ensure the system is working as expected. Each type of test requires more or less tests depending on the scope of the test.

Unit tests should take the biggest amount of code in terms of written lines of the code in the codebase due to the fact that unit tests are testing the smallest amount of the services, meaning each individual class will be well tested [4].

The next type of test, which is broader in terms of the area of testing, are component tests. A component test is an acceptance test for a single service that checks its behavior as a whole unit. Classes, functions or modules within the service are viewed as a unit. Component test can be called a unit test at a service level.

Integration tests are required to validate that the services are well integrated between themselves. The integration tests are also testing integration of a service with its third-party components such as databases, queues of third-party tools. A subtype of integration tests are contract tests, this kind of tests ensure that one service’s response meets the contract it exposes. It is usually used for REST contract testing.

The broadest side of tests in terms of coverage area are end-to-end tests. These kinds of tests are expensive in terms of time and things required for implementation, thus there are usually a smaller number of end-to-end tests. However, end-to-end tests are usually run as a sanity check to verify that the whole system is working as expected and to validate that the most important and critical parts of the service are working well.

To ensure that the system is working as expected and the tests to run in a timely manner, the number of tests should decrease starting from unit tests and reaching to end-to-end tests, due to the fact that unit tests are faster and cheaper in terms of time and end-to-end tests are slower and cost more.

Data, transactions

Due to its distributed nature, each service has its internal database transactions ensuring an internal consistency of the database, however the services often share the same piece of data which is spread across multiple databases from multiple services. This requires a "transaction" across multiple services, which will span across multiple databases.

Distributed transactions is a way of handling such cases, which is outdated nowadays, because new technologies do not support them, meaning that the project could potentially be limited in the used technologies.

Another approach for managing transactions is the saga pattern. This is a way of handling transactions that span across multiple microservices by creating an ordered sequence of actions for each microservice, each action executes its internal transaction which benefits from ACID properties of a relational database.

There are 2 types of implementations of the saga pattern:

- Choreography - each service communicates via events and triggers the transactions in other services. In case something went wrong in a service, it can undo the previous transactions by emitting special events, which are handled in other services, that are aware of these events and know exactly how to handle the rollback.
- Orchestration - a special type of saga that has a main service that coordinates with the events. This service can send events for starting the transactions as well as events for undoing them, however, the service may become a single point of failure, which may be dangerous.

Conclusions

In many cases the disadvantages of the microservice architecture can outweigh the advantages due to the introduced complexity, meaning that microservices are not a silver bullet architecture that solves all the issues. Microservices represent an approach to design and implement scalable, isolated and independently deployable software services. It requires a deep understanding of the domain and the requirements in order to start a fresh project using microservices architecture or to migrate an existing system to microservices.

References:

[1] LEWIS, James, FOWLER, Martin, *Microservices a definition of this new architectural term* [online]. Available: https://martinfowler.com/articles/microservices.html

[2] NEWMAN, Sam, *Building Microservices.* United States of America, Sebastopol, CA 95472: Ed. O'Reilly Media Inc, 2021. ISBN 978-1-492-03402-5

[3] RICHARDSON, Chris, *Microservices Patterns.* United States of America, Shelter Island, NY 11964: Ed. Manning Publications, 2018. ISBN 978-1-617-29454-9

[4] RICHARDS, Mark, FORD, Neal, *Fundamentals of Software Architecture.* United States of America, Sebastopol, CA 95472: Ed. O'Reilly Media Inc, 2020. ISBN 978-1-492-04345-4