International Journal of Advances in Software Engineering & Research Methodology– IJSERM
Volume 2: Issue 2 [ISSN : 2374-1619]
Publication Date : 30 October, 2015

Internal and external structure of microservices architecture

[Ionut Gheorghe Hrinca]

***Abstract*—Microservices are one of the hot topics around architectural styles in software development. Its main philosophy is not something new but lately there was a name assigned to this architectural style and people started to talk about it and adopt it. Many development teams seem to be too eager to embrace microservices without realizing the complexity that is added to the system by them.**

The main focus in this paper is to define the characteristics of the microservices architecture, its external and internal structure, all in the context of cost and benefits and good practices.

***Keywords*—microservices, architecture, microservices architecture, hexagonal architecture**

I. Introduction

The architecture oriented on microservices is a new hot topic in approaching the architecture of high complex systems. As per Martin Fowler explanation, “the term *microservice* was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on *microservices* as the most appropriate name.”[3]

It took several years for microservices architecture to be adopted. One of the first to experiment this new architectural style was Netflix in 2013 in one of the projects run by Adrian Cockcroft (one of the architects present at the workshop in Venice in 2011)

II. Microservices architectural style

A. *Definition of the microservices architectural style*

At the logical level, the microservices architectural style can be defined as the functional decomposition of the system in components that can be managed and deployed individually.

The first part of this definition refers to the functional decomposition and actually it refers to vertically slicing of the system. This is one of the main differences in approach in contrast with the classical SOA.

The second part of this definition refers to the independence in management and deployment. This implies that, in designing microservices, one must consider that microservices shouldn’t share the state and also there shouldn’t be any inter-process communication. Usually this is achieved using REST interfaces over http.

Figure 1. Logical view of microservices architecture

B. *Characteristics of microservices architectural style*

Analyzing the current literature around microservices and also testing the theories in practice, the microservice architectural style have the following set of characteristics that differentiate it from the classical SOA approach [3].

a) Componentization through services

The classical approach was that the components were included in some libraries, which were linked as part of a program and using their functions in the internal memory space of the machine. Everything was happening in the same process.

The main difference is that services are components outside the process, which run and communicate through a mechanism (like web service calls or remote procedure calls)

b) Organized around business capabilities

In contrast with other architectural styles where the applications and systems had different layers (UI, technologic, application/server logic, data) with the business logic spread through almost all the layers, the microservices style approach is focused on splitting the business logic in business capabilities. Each business capability is incorporated in only one microservice, which contains in it all layers from interfaces to data.

Ionut Gheorghe Hrinca *(Author)*
PhD Economic Informatics - Bucharest University of Economic Studies
Romania

c) Smart endpoints and dump communication channels

The systems developed based on microservices approach aim to be as decoupled and as cohesive as possible, containing inside of them all the business logic for the domain that they serve.

The approach is similar with the filter approach in Unix. Each microservice act as a filter in the sense that at the moment of receiving a call, it applies the necessary logic and provides a reply.

In classical SOA implementation one can notice that complex ESB solutions, among the main purpose for which they were created, they were in charge of a big part of orchestration, choreography, sophisticated routing or even of some business logic. In the microservice architectural style only lightweight messaging solutions must be used (e.g. message buses that handle only simple message routing).

d) Decentralized governance

Ideally, following the principle *develop it and use it*, the teams that develop the microservices are in charge also of operating them. From the practice one can say that centralized governance leads to technological platform standardization creating constraints for development of the business requirements, which is exactly the opposite direction that microservices architecture follow.

e) Decentralized data management

In the most abstract way, this means different views of different systems on the conceptual model of the entity. Decentralizing the data through the microservices creates lots of problems in table update management. Distributed transactions are a solution to this but they are well known as being extremely hard to implement and they bring a lot of complexity to the system. Inconsistency management is the new challenge that the team responsible for the microservice has. The most easy and common practice is that business to accept a certain data inconsistency as a trade off for quick reply to the customer request. A reversal mechanism is also put in place to solve later the inconsistency issues. Always one should balance the cost of solving all the consistency issues with the cost of business loss due to the consistency issues.

f) Infrastructure automation

To reach the level of agility that it promises, the development of a system based on microservices should rely on an automated infrastructure to reduce the burden of deploying and testing microservices. Technics like continuous delivery should be used in order to reduce the effort by automating the build, the test and the deploy stages.

g) Design to fail

Due to a sum of reasons or circumstances, any microservice can be unavailable and this should be in every development team member mind when they design the application or system. The system should be design in such way to handle this kind of failures. To prevent negative effects due to unavailability, a real time monitoring system needs to be in place in order to detect the problem and to start the execution of the failover procedures or to restore the service. Semantic monitoring is important to identify eventually problems that could appear in the future based on certain patterns.

h) Evolutionary design

The practitioners see the microservices architecture as a tool to progressively decompose the applications in such a way that developers could control the changes in their application without slowing down the whole change process. This change control don't mean a slowing down of each microservice but rather, using the right automated tools and the right mindset on the system partitioning, means often and faster deliveries in a controlled way.

C. *Costs and benefits of microservices architecture*

The main benefits of microservices architecture are:

- Strong modularization - microservices reinforce the modular structure of the system
- Independence in deployment - microservices are easier to deploy because they are autonomous and the risk of breaking down the whole system is very low.
- Technological diversity - using this architectural style different technologies, frameworks, data access technologies and coding languages can be used and can be mixed in order to target the best solution for each requirement.

All the benefits come always with the attached costs. Among these costs we can mention:

- Cost of distributed systems - distributed systems are hard to develop and also it is hard to change the developers' mind set in regards to asynchronous function calls and to the fact that the functions can be unavailable and their component need to handle the situation accordingly.
- Cost of consistency - keeping the consistency level high implies a high cost
- Operational complexity - the operations teams need to be mature enough and they need a set of automated tools for automated deployment, automated testing, monitoring, etc.

Analyzing the costs and the benefits of microservices architecture for the decision for adopting the architectural style one needs to take in consideration also the complexity of the system but also its dynamic and frequency of change through which business value can be delivered faster. Martin Fowler presents in his article [4] the fact that, for the classical architectural styles, once the complexity of the system increases the productivity decreases with a much higher correlation coefficient compared to microservice architecture.

III. External structure of microservices architecture

When you use microservices for decomposing the system the expectations are that you end up with a loosely coupled system in which you have agility in development, flexibility in operations, you can scale microservices independently, you can deploy microservices independently (different deployment options should be made available by the architecture). In case of a microservice failure that needs to be isolated and shouldn't affect other microservices and shouldn't bring the whole system down until the issue is fixed.

The next diagram represents a proposed external structure in designing microservices-oriented system architecture. Using this approach the system can be decomposed into loosely coupled microservices that:

- Have an interface or an explicit contract - REST / SOAP (1)
- Boundaries are aligned with business capabilities (2)
- Use asynchronous communication between microservices (3)
- Have their own storage - they are the authoritative source of data for their domain (4)

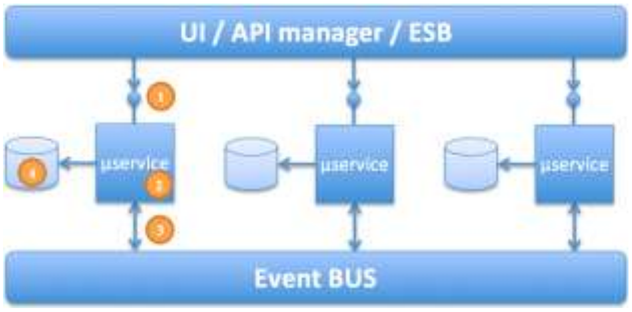

Figure 2. Microservices architecture - external structure

Decomposition of the system needs to be done in a way that a microservice incorporates all the necessary functions of the covered domain [1]. If done this way, decomposition doesn't just break the system into small and independent units, but in autonomous units with capabilities that can be used by other components.

Applying the approach described above in the diagram, multiple touch points or any other third party applications (part of the system) can consume the microservices capabilities, exposed by their APIs, and offer to the customer the functionality and the experience that he or she expects.

Synchronous communication leads to a high level of coupling between components of the system and also reduces the throughput of the system. In order to avoid this microservices have to communicate asynchronously exchanging events using the publisher-subscriber integration pattern [2]. To facilitate the communication between microservices it is recommended to use a lightweight event bus. The advantages of using this pattern are:

- Temporal decupling
- Microservices don't depend on each other's availability
- Better business analytics by capturing the history of business events

IV. Internal structure of a microservice

A. *Hexagonal architecture microservices*

A microservice should follow the hexagonal architecture pattern. This pattern is also called ports and adapters. The main reason in choosing this kind of architecture is the separation between business aspects and technological aspects.

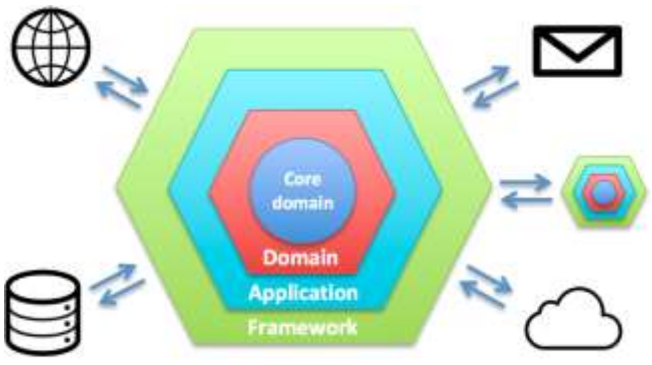

Figure 3. Hexagonal architecture of a microservice

Hexagonal architecture defines the conceptual layers of code responsibility and indicates ways to decouple between the layers. It also defines when and how to use interfaces. The hexagonal architecture is not a new development pattern inside a framework but rather is a way to describe good practices. It describe ways to decouple code from the framework, means to expose the application and how to use frameworks only as means to create functionality into the application.

In a port and adapters design pattern, the port is an expression of component's interface. *In* ports expose the functionality of the core. *Out* ports describe how the core sees the outside world (the rest of the components of the system with which it interacts).

The adaptors are located outside the hexagon (component). Their role is to ensure that the transport of the information between the port and the destination component is happening according to the contract of the port interface. When the microservice is tested, the only thing that needs to be replaced is the port. This can be done using the DI (dependency injection) technique.

B. Structural modularity of a microservice

Considering its hexagonal architecture, a microservice should contain the following components around its core domain model:

- Interface adaptors - REST / SOAP
- Serialization / deserialization component
- Helper function
- Third party libraries
- Persistence component
- Logging component
- DI container
- Publish / subscribe event component

All these components and the way they interact are depicted in the following diagram. All asynchronous communication is marked using dashed lines.

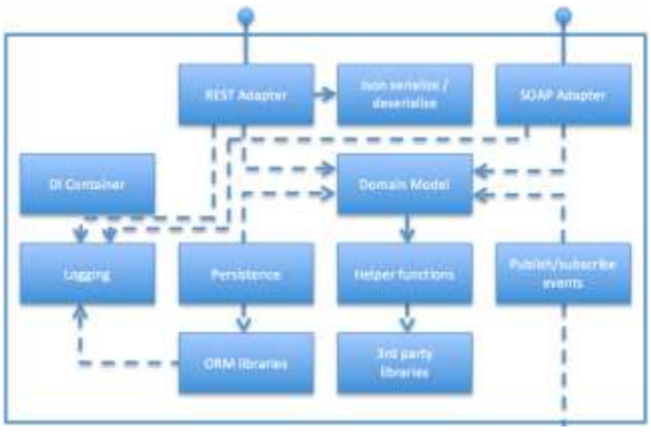

Figure 4. Microservice - internal structure

C. Evolutionary steps in the direction of mature modularity

System decomposition into microservices alone is not enough for reaching a high agility level in development. Microservices bring along some additional complexity that needs to be taken care of. In order react quickly and cheap to the business change need, the dependency between microservice modules must be managed.

Modularity saves the complexity problems created by the microservices. Creating a mature level of modularity requires following the next four steps:

- Creation of modules - dependencies on others modules' identity (Maven style) solve problems like transitive dependency problem and "jar hell" problem
- Module encapsulation - dependencies on packages exported by other modules. Modules are isolated form each other by having public and private packages and dedicated class loaders.
- Module dependency management - minimizing coupling between modules by applying modularity patterns and enforcing the desired dependencies.
- Module dynamism - the modules can be upgraded/replaced on the fly without downtime.

References

[1] Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, New York 2011

[2] Gregor Hohpe, Bobby Wolf , Enterprise Integration Patterns, 1st ed., Addison-Wesley Professional, 2003, pp151-153, 137-141

[3] Martin Fowler - Microservices - March 2014 - http://martinfowler.com/articles/microservices.html

[4] Martin Fowler - Microservices premium - May 2015 - http://martinfowler.com/bliki/MicroservicePremium.html