bla bla microservices bla bla

Author: Jonas Bonér

It seems like everyone is talking, writing, and thinking about Microservices—me included. Still, there is more confusion around what it is, and how to deliver on it, than ever. Today we are going to take a look at Microservices from *first principles* and its true context: *distributed systems.*

Reality Has Caught Up On Us

Truth be told, we have been spoiled by the once-believed almighty Monolith, with its single SQL database, for way too long. Its fairytale world where we could assume strong consistency, one single globally consistent «now» where we could comfortably forget our university classes on distributed systems.

Knock, knock. Who's there? Reality. We have been *living in an illusion*—far from reality.

Today's applications are deployed to everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and close to 100% uptime. And by «user» I mean both humans and machines.

Traditional architectures, tools and products such as JEE and Spring Framework simply won't cut it anymore. We can't make the horse faster anymore, we need cars for where we are going.

So it's time to wake up, to retire the monolith, to decompose the system into manageable, discrete services that can be scaled individually, that can fail, be rolled out and upgraded in isolation.

The Reactive Essence of a Microservice

Asynchronous Communication, Isolation, Autonomicity, Single Responsibility, Exclusive State, and Mobility. These are the core traits of Microservices. Let's discuss each one of these in a bit more detail.

Go Asynchronous

It is unfortunate that *synchronous HTTP* is widely considered as the go-to Microservice communication protocol. Its synchronous nature introduces strong coupling between services which makes it a very *bad default protocol* for inter-service communication.

Instead, communication between Microservices needs to be based on *Asynchronous Message-Passing.* Having an asynchronous boundary between services is necessary in order to decouple them, and their communication flow,

- in *time*: for concurrency, and
- in *space*: for distribution and mobility

Isolate All the Things

Isolation is the most important trait, the foundation for many of the high-level benefits in Microservices, and it is also the trait that has the biggest impact on your design and architecture. It will slice up your architecture. It will impact the way you organize your teams and their responsibilities, as Melvyn Conway discovered in the 60's and was later turned into Conway's Law.

Isolation of failure—being able to *contain and manage failure* without having it cascade—is a pattern sometimes referred to as *Bulkheading.*

Bulkheading has been used in the ship construction industry for centuries as a way to divide the ship into isolated watertight compartments, so that if a few compartments are filled up with water, the leak does not spread and the ship can continue to function and reach its destination.

Resilience—the ability to heal from failure—depends on compartmentalization and containment of failure, and can only be achieved by breaking free from the strong coupling of synchronous communication.

The good news is that today we have a much more refined foundation for isolation of services, using virtualization, Docker, Unikernels.

Act Autonomously

Isolation is a prerequisite for *autonomy* and mobility. Only when services are isolated can they be fully autonomous and make decisions independently, act independently, and cooperate and coordinate with others to solve problems.

Do One Thing, and Do It Well

The word Microservice is a terrible word. It implies size which make people argue about how many LOC a service can have and still be micro. We need to stop this madness.

It is about one thing: *scope of responsibility*—adhering to the Single Responsibility Principle. A service should only *do one thing and do it well*; have one reason to change.

If a service only has one single reason to exist, providing a single composable piece of functionality, then business domains and responsibilities are not tangled. Which makes the code and system easier to understand, compose, extend and maintain over time.

Own Your State, Exclusively

Ok, but what about state?

What is needed is that each Microservice take sole responsibility for their own state and the persistence thereof. Which storage medium is used does not matter; what matters is that a service can be treated as a single unit. That it *own its state, exclusively.*

Stay Mobile, But Addressable

Mobility is the possibility of moving services around at runtime, while they are being used.

One requirement for this is that the services are *addressable through virtual, stable addresses*, that always work, even if a service is failing, is being relocated or upgraded. This is called Location Transparency, and is one of the cornerstones in Reactive system design.

It is paramount that the service is seen and moved around *as a single unit*—including its behaviour and persistent state—to remain oblivious to how the system is deployed, which topology it currently has—something that changes dynamically.

Microservices Come In Systems

Now we have a pretty good understanding of what characterizes a single Reactive Microservice. However, one Microservice is not of much use, *they come in systems.*

Like humans they *act autonomously* and therefore need to *communicate and collaborate with others* to solve problems—and as with humans, it is in collaboration that both the most interesting opportunities and challenging problems arise.

Individual Microservices are fairly easy to design and implement—what is hard in Microservices is all the things around them: discovery, coordination, security, replication, data consistency, failover, deployment, and integration with other systems, just to name a few.

Systems Need To Exploit Reality

One of the major benefits of Microservices-based Architecture is that it gives you a set of *tools to exploit reality*, to create systems that closely mimic how the world works, including all its constraints and opportunities.

One subtle, but important fact to embrace, is that *reality is not consistent*—there is no single absolute present—everything is relative, including time and our experience of now.

Information cannot travel faster than the speed of light, and most often travels considerably slower, which means that communication of information has latency. Information is always from the past, and *«now» is in the eye of the beholder.*

Understanding this fact can be both terrifying and liberating.

As soon as we exit the boundary of the Microservice we enter a *wild ocean of non-determinism*—the world of distributed systems—where systems fail in the most spectacular and intricate ways, where information gets lost, reordered, garbled, and where failure detection is a guessing game. But it is also the world that gives us solutions for resilience, elasticity, isolation amongst others.

But the Microservice can become an *escape route from reality.* Within each Microservice, we can live on a *safe island of determinism* and strong consistency—an island where we can live happily under the illusion that time and the present is absolute.

Embrace the Constraints of Distributed Systems

Minimize Coupling and Communication

Strong consistency requires *coordination*, which is *extremely expensive in a distributed system*, and puts an upper bound on scalability, throughput, low latency and availability.

The need for coordination means that individual services can't make progress individually, but has to wait for consensus. When designing Microservices-based systems we should therefore strive to minimize the service-to-service coordination of state and to allow them to *comfortably share silence.*

Exploiting reality means coming at peace with the fact that *information is always from the past*; always representing another present than ours; another view of the world. To model this we *have to rely on Eventual Consistency.*

It might sound like we are giving up a lot, and we are, but we are also raising the ceiling on what can be done in terms of loose coupling, scalability and availability—as stated by the CAP theorem.

Guess and Apologize

Grace Hopper once said:

> "It is easier to ask for forgiveness than it is to get permission."

If you can't coordinate, and be certain about something, then *take an educated guess*, a bet that a condition will hold, and *if you were wrong then apologize* and perform a *compensating action*.

This approach matches reality very well. It's how we humans collaborate all the time. Other examples include ATMs—allowing you to withdraw money in the case of network disconnect, and then later charging your account—and how airlines are over-booking flights—and then trying to bribe themselves out of the problem through vouchers.

What About Transactions?

But what about transactions? Don't we need transactions?

Transactions are fine within the individual microservice, but to quote Pat Helland:

> "In general, application developers simply do not implement large scalable applications assuming distributed transactions."

A practical, scalable and resilient alternative to distributed transactions, that makes use of some of the ideas we have discussed, is the Saga Pattern. It is a way to manage long-running business transactions and is based on idea that long-running business transactions often comprise multiple transactional steps in which overall consistency of the whole transaction can be achieved by grouping these steps into an overall distributed transaction. The technique is to pair every stage's transaction with a compensating reversing transaction, so that the whole distributed transaction can be reversed—in reverse order—if one of the stage's transactions fails.

Summary

To sum things up: bla bla microservices bla bla. Thank you.