Technology Conversations

Microservices: When the Stars Aligned

In The History of Failed Initiatives I mentioned that microservices as a concept existed for a long time. And yet, most of those who tried organizing their architecture around microservices failed or, to use different words, realized that benefits are shadowed by the increase in complexity (especially in operations). The spark that was born long ago became a star only recently. In order to understand why microservices became a hot topic not long ago, we need to understand better why they failed in the past. The explanation is simple. We neither had the tools nor understood the logic we had to apply. Do we have the tools today? Are we capable of putting in place the right procedures and have the mindset microservices require. I'd say yes, we have and we do. When I say we I mean people with the will and resources to make this happen. As in most other cases, once something new is proven and adopted by some, years need to pass until that something is applied by many. The good news is that I think that we are getting there and, slowly, the more conservative part of the industry (in other words, most of it) is starting to at least consider microservices as an option. Even Gartner had positive words for it. And when Gartner says jump, enterprise jumps.

Why would microservices work today if they didn't work yesterday (except for a very few)? What do we have now that we didn't have before? The stars aligned and five (mostly unrelated) concepts and technologies are becoming very prominent and, more importantly, proven to work. It's not that they didn't exist before. Some did and some didn't. What they all have in common is that we started to take them seriously and to see them as a whole. Those concepts and technologies are following (in no particular order):

- Domain-driven design (DDD)
- Continuous delivery and deployment (CD)
- Immutable deployments
- Scalable systems
- Small autonomous teams

You might say that, for example, you had small autonomous teams for quite some time. Or you might say that you work in Google and containers are not a new thing. I won't question that since I would risk missing the point that I am about to make.

Let's start with a question. What did those concepts and tools around them bring to the table?

For too long we were designing and developing applications around horizontal layers. We tried to approach everything from technical point of view without truly understanding the context of each entity. Many of us didn't even understand what an entity is. We though that an entity is a set of DB tables, or a class in a business layer. **Domain-driven design (DDD)** thought us how to define and develop around well-defined entities. It thought us what an entity is on a conceptual level that is to be translated to bytes later on.

Continuous integration (CI) existed for a long time but it has no clear goal. It is a vague (but brilliant) idea that applications should be integrated at (almost) all times instead of waiting for the end of development to discover whether everything works well together. While the only way to be able to call something continuous meant that things need to be automated, it also assumed that part of the process is manual. That part was performed after the automated integration pipeline. There was never a clear boundary between the automated and manual parts of the process. As a result, continuous integration become popular and everyone decided that it is something that they must do (in the same way that today everyone wants big data and internet of things). At the same time, we started getting very poor results from adopting continuous integration. That is not to say that some didn't do it right nor that it didn't yield huge benefits. However, what that luck of clear goals or, better said, clear objectives accomplished is that many thought that installing Jenkins and setting up jobs that perform few builds and run some tests is all there is to be done in order to be "continuous integration" certified. And you cannot say that they don't do continuous integration. Are they checking whether applications are integrated? Yes they do. Is the software really working as expected? It often isn't. If it would be, there wouldn't be a round of manual testing. If it would work well, it would be called **continuous delivery (CD)**. Unlike CI, CD has a real and measurable objective. It has something that clearly distinguishes those who practice it from those who want to fake it enough in order to the "certified". In a nutshell, you are doing continuous delivery if every commit that passed the whole automated pipeline can be deployed to production. The key is in the words **every commit**. You are not deploying every build to production due to reasons that have nothing to do with the quality of your code. You might not want to release a feature A until feature B is ready. You might want to wait until the marketing campaign is ready. No matter the reasons, **every build can be deployed to production** with a reasonable assumption that nothing would go wrong. Now, that is something to strive for. That is something that cannot be faked. Same goes for **continuous deployment** which is basically the same process with the same quality but without having that button that says "deploy to production". Every build that passed the whole automated pipeline **is deployed to production**. There is no real difference between the two except that the later doesn't have that button.

Containers with **Docker** are probably the hottest topic these days (not to say months). During many years I've been working in the industry, I cannot think of many examples of a technology that got from nothing to everyone wants it during such a short period of time. There is a reason for it. Containers solve many problems that we were facing before. The whole concept of immutable deployments is a very powerful practice. As with other things, neither containers not immutable deployments can be considered new. Containers existed for a long time. However, until Docker emerged they were too complicated to reason with. On the other hand, immutable deployments have been practiced by some almost since the days virtual machines emerged. The reason they were practiced by some and not by many is that VMs, in the context of immutable deployments, leave a lot to be desired. Being slow to create, cumbersome to move around and their high usage of resources made them more of an experiment in immutable deployments than something that would take the world by the storm. Docker solved those problems and allowed us to practice immutable deployments without ripping our hair out.

Many will tell you that they have a **scalable system**. Scaling is easy after all. Buy a server, install WebLogic (or whichever other monster applications server you're using) and deploy your applications. Then wait for a few days until you discover that everything is so "fast" that you can click a button, go have a coffee and, by the time you get back to your desk, the result will be waiting for you. What do you do? You scale. You buy few more servers, install your monster applications server and deploy your monster applications on top of it. Which part of the system was the bottleneck? Nobody knows. Why did you duplicate everything? Because you must. And then some more time passes and you continue scaling until you run out of money and, simultaneously, people working for you go crazy. Today we do not approach scaling like that. Today we understand that scaling is about many other things. It's about elasticity. It's about being able to quickly and easily scale and de-scale depending on variations in your traffic and growth of your business and that during that process you should not go bankrupt. It's about the need of almost every company to scale their business without thinking that IT department is a liability. It's about getting rid of those monsters.

There's not much to be said about **small autonomous teams** without offending half of the industry. It has been proven over and over that they perform better than big ones and yet many still do not have them. What wasn't so obvious from the start is that small teams and monolithic applications are a bad marriage. It's not enough to simply form small teams. We need to change the approach to software architecture. That is probably the main reason why big companies tried and failed to change the organization of their teams. They were not ready to change their software architecture. In some other cases they even concluded that they had small teams for a long time. There is a team of front-end developers, there is a team of back-end developers, there is a team of testers, and so on. And then you multiply the number of those teams with the number of applications and you end up with an organization where people do their work 20% of their time and dedicate the rest to futile attempts of trying to figure out what is going on. You take a look at their agendas and wonder whether they ever do anything besides attending (or organizing) meetings. They missed the word **autonomous** and ended up with simply **small teams**. If you want small autonomous teams, you need to have small autonomous applications.

Those five concepts, and tools around them, converged into microservices. When joined together the need for microservices becomes real and the benefit of taking that road becomes substantial. Without them, there are no microservices or the need for them.

Domain-driven design, even though not invented for that purpose, is easiest to apply when used in conjunction with microservices. Continuous delivery or deployment is much harder to accomplish with monolithic applications. Containers do not work well with huge application servers. Systems can hardly be scalable if applications are huge and small autonomous teams can not be small or autonomous if applications are not equally scaled down. On the other hand, microservices can hardly be successful if some form of domain-driven design is not applied. They are pointless if their development cycle is long (hence CD). They can be too much of an overhead if the whole system is small (hence applied only on large-scale). Finally, only an insane person would develop them if teams are huge or not autonomous.

This entry was posted in Architecture and tagged CD, Continuous delivery, Continuous Integration, Delivery and Deployment, DDD, Docker, Domain-Driven Design, Immutable deployment, jenkins, Scaling on November 3, 2015 [http://technologyconversations.com/2015/11/03/microservices-when-the-stars-aligned/] by Viktor Farcic.

One thought on “Microservices: When the Stars Aligned”

Pingback: Revue de Presse Xebia | Blog Xebia France