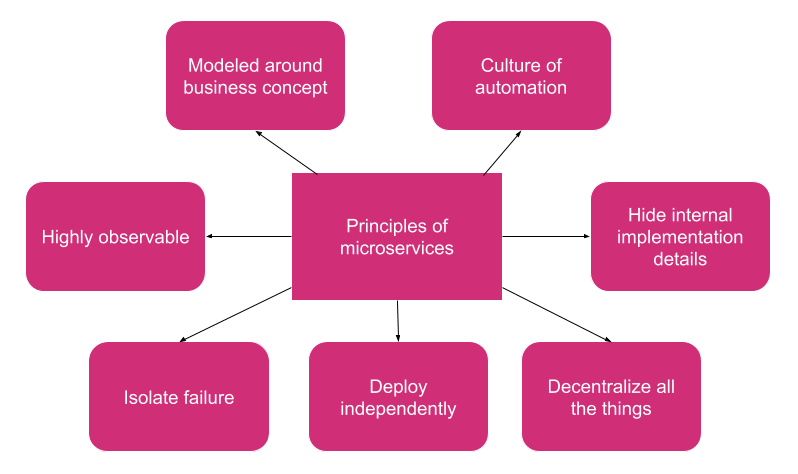
I have been doing some research on best practices in microservice architecture, and I mainly got interested in observability aspect. One of the reason is that I had an opportunity to work in the enterprise system which lacks good observability and I know for a fact how difficult it is to reason about what is going on in production.

It's been a long time since observability was simple as logging to the file system, sending email alerts or using some OS capabilities for diagnostics. Nowadays, in complex distributed systems with lots of small services which could be replicated on different servers or using a different OS or even written in different programming languages, is essential to maintain a good level of observability. A common approach is to gather all information in one central place which you can later use to better understanding of how your system behaves in production. Unfortunately, this is where it gets a little bit tricky. It goes without saying but you will need to invest in infrastructure to support this. Going big costs time and money, and you better be sure you need it.



Sam Newman wrote about principles of microservices in his book “*Building Microservices*”, and one of the principles was about observability of whom I’m writing today. What struck me the most is the following:

“*You can choose to adopt these principles wholesale, or perhaps tweak them to make sense in your own organization. But note the value that comes from using them in combination: the whole should be greater than the sum of the parts. So if you decide to drop one of them, make sure you understand what you’ll be missing.*”

High observability is something you must not drop. I think of this as something you must have in place before you start working on features (like cross-cutting concerns in monolith). If you drop it, prepare for a production nightmare when you get a phantom bug ticket, or your system suddenly starts to misbehave on peak time. What I like to do is, forget about using a debugger while I’m working in localhost. This usually forces me to have meaningful information in tracing, logging or metric systems.

**Four pillars of observability**

Usually, articles on the internet are writing about three pillars (tracing, logging, and metrics) and they are without a doubt the most important ones. I would like to include one more, and that is “Healthcheck”.

**Healthcheck**

Health checks were usually designed to answer a simple question:

“Is my service up or down?”.

The check was usually implemented by merely checking is there something listening on this end-point and simple Yes/No answer was sufficient for a long time, but we can add more information in health check result. We can make a subtle shift from the previous question to this one:

“Is my service ready to take more workload?”

and instead of sending simple Yes/No response, we can structure our response something like this

* Code block

If you decide to go this way you have to make sure that your downstream service health check does not take to long and that you don’t propagate health checks further down the stack.

There are three techniques you can use to instrument your services

Broadcast - Broadcast health events to everyone who is interested

Register - Register your service in zookeeper or etcd with some TTL. If service is not registered after TTL expired, service is considered to be down

Expose - Make service expose health info and get that info by pooling

**Logging**

We have been doing logging for a long time, and usually, we were logging to the file system, or we were using some OS functionality. In the distributed world, we need to get those log events in one central place which we can later use for querying.

There are few properties of log events that are important in distributed systems:

* Centralized
* Structured
* Correlated

Centralized logging

Since we have lots of services in our data center, it is reasonable to have log events centralized in one place.

Structured logging

Structured means that log events are always logged in a standard format with common information. Log events are usually represented using JSON and sent to the centralized logging system. Whenever you are logging application event think about person who is looking at the logs and try to provide as much context as possible.

In ASP.NET core we would typically log application event something like this:

logger.Information($“Provisioning subscription for customer {customer.Id}”)

but luckily ASP.NET Core supports structured logging out of the box:

logger.Information( “Provisioning subscription for customer {customer\_id}”, customer.Id)

It is important to be consistent with structured logging so it might be useful to create a small library which could deal with consistency and distribute this library using some package manager like nuget.

Correlated logging

**Tracing**

**Metrics**

Metrics are optimized for long storage.They are suitable for dashboards which shows threands across time