# **Micro services, what even are they?**

I’m going to start by saying I’ve probably got no right to be leading the charge for a definition for micro services, but I do have a lot of skin in this game, as it’s the direction that I’ve been pushing REA development for the past 2-3 years. Much of this is my personal perspective, but I do think it’s broadly applicable and does provide what I consider an alternate viewpoint on the vision for micro services that exist.

To answer Jonathan’s second question “How is it any different to SOA?”, my immediate response is “the intent is different”. With SOA, the intent is a layered architecture of co-operating services where SOA focuses on describing the organisation and co-ordination of the services. With micro services, the intent is to describe the nature of the services themselves and not quite so much the organisation and co-ordination of them.

While SOA is used as a comparison, SOA itself has no “one true definition” but merely a collection of patterns/principles and attributes regarding the organisation and co-ordination between services. I should point out that I see micro services and SOA working well together, with micro services describing attributes of the services themselves and SOA providing useful guidance on how to arrange them.

**So, why do I think this way?**

I’m a software developer, designer and architect. I like to think a lot about the human factors of software development and how can I put systems in place to encourage development teams to “do the right thing” when building software. There’s far too much shit software out there, and I like to have teams not contribute to that. With that in mind, why did I think micro services was a “good approach”? My definition is meant to be used to guide *development*. The benefits that we get operationally is wonderful—but that’s not the primary reason. It’s to get developers to stop building Borgified software with unclear responsibilities and brittle coupling.

First it’s probably worth providing my definition of what a micro service is, so that there’s at least some context around the discussions that may, or may not ensue. After defining the attributes, I’ll expand on why I consider them important.

Desirable attributes of a micro service are:

1. The responsibility is narrow. The service does one thing, and one thing well.
2. The code base is small. The service can be rewritten and redeployed in 2 weeks.
3. There is no 3.

I tried to think of more, but most of them were derived from these. A valuable attribute is the ease of upgrade and redeployment. This is directly related to #1. Another valuable attribute is the ease of change. Both #1 and #2 provide support here. There is also the ability for services to be re-used effectively. This is related to #1. A person much smarter than I am once said “The unit of reuse is the unit of release”.

There’s possibly some rambly hipster crap about “REST services” and “HATEOAS” but really, that’s such flavour of the month and not really something that I think is that important. Certainly no more interesting than JSON vs XML vs ASN.1. All of these things can be done well, or badly—but don’t provide a defining point on if an implementation has desirable attributes.

**The responsibility is narrow**

This key point relates to design and the fundamental architectural principles. If the responsibility is narrow, then hopefully it follows that the codebase will be small. If the responsibility is narrow, then the understanding of where to make changes is clearer and design intent can be carried forward. If the responsibility is narrow, then understanding how the service fits in the broader network of services, or how the service can be reused is much clearer.

The second important part here is the ability to release the services often, cheaply and without needing to have a deep graph of dependencies. Having a narrow responsibility means that any systems that want to use the services are only coupled to that service for that responsibility. There’s no undesirable coupling.

Like object oriented software, services are best with high cohesion and low coupling. Creating services as micro services helps in this regard.

**The code base is small**

When I first started proposing micro services I wanted to appeal to the developers, so I said that services could be written in any language they choose, The only caveats were that the component had to conform to our monitoring and logging interfaces (to aid with deployment and operations) and that it could be re-written in 2 weeks.

This created significant consternation, not by developers, but by management. They were concerned about the “explosion of software that nobody could understand”. I did laugh while explaining my reasoning. I laughed mostly because their basis of concern was that “it would take too long”. Sadly this shows the lack of understanding about software that pervades our industry.

Most developers are perfectly capable of understanding new syntax, and generally can understand new syntax in a relatively short period of time. What takes much, much longer is understanding twisted and tortured domain logic, scattered across 6 packages and libraries all bundled together in one monolithic application.

My rationale is that if software is written according to the simple rules—narrow responsibility and small codebase—then the actual language choice is for the most part irrelevant in terms of defect fixing and extension. Sadly, I don’t have a lot of data points in this regard, as developers seem to want to choose the path of least resistance (which is normally keep writing the same old shit in the same way), but I do have a great example written by my team.

We had the need to write a service and one of the team wrote it in Go. It was working well, performed as expected and when it came to adding some additional monitoring we hit a snag because the Go runtime wasn’t supported by NewRelic. The developer who wrote it had sadly departed the team so another team member re-wrote the service and had it redeployed in 2 weeks. Written in Java, using Dropwizard. It was a perfect example of exactly what I was proposing.

There are some really useful patterns that we developed while creating the service, not really suitable for addition here, but if there is enough interest I can expand on it in another post. However, the way we thought about building the initial service and more importantly the automated testing around that service made re-development trivial, and incredibly safe.