# Introduction to Microservices

Microservices are small, autonomous services that work together. They communicate with each other via networks, and as an architecture choice offers many options for solving the problems, you may face. It follows that a microservice architecture should base on multiple collaborating microservices.

# Independent Deployability

It is the idea that we can make a change to a microservice and deploy it into a production environment without having to utilize any other services. More importantly, it’s not just that we can do this; it’s that this is how you manage deployments in your system. It’s a discipline you practice for the bulk of your releases. It is a simple idea that is nonetheless complex in execution.

To guarantee independent deployability, we need to ensure our services are loosely coupled—in other words; we need to be able to change one service without having to change anything else. It means we need explicit, well-defined, and stable contracts between services. Some implementation choices make this problematic—the sharing of databases, for example, is especially problematic. The desire for loosely coupled services with stable interfaces guides our thinking about how we find service boundaries in the first place.

# Modeled Around a Business Domain

Making a change across a process boundary is expensive. If you need to make a change to two services to roll out a feature and orchestrate the deployment of these two changes, that takes more work than making the same change inside a single service (or, for that matter, a monolith). It, therefore, follows that we want to find ways of ensuring we make cross-service changes as infrequently as possible.

Following the same approach, I used in Building Microservices, this book uses a fake domain and company to illustrate certain concepts when it isn’t possible to share real-world stories. The company in question is Music Corp, a large multi-national organization that somehow remains in business, despite it focusing almost entirely on selling CDs.

.

# Own Their Own Data

One of the things I see people having the hardest time with is the idea that microservices should not share databases. If one service wants to access data held by another service, then it should go and ask that service for the data it needs. This gives the service the ability to decide what is shared and what is hidden. It also allows the service to map from internal implementation details, which can change for various arbitrary reasons, to a more stable public contract, ensuring stable service interfaces. Having stable interfaces between services is essential if we want independent deployability—if the interface a service exposes keeps changing, this will have a ripple effect causing other services to need to change as well.

# Advantages of Microservices

The advantages of microservices are many and varied. The independent nature of the deployments opens up new models for improving the scale and robustness of systems and allows you to mix and match technology. As services can be worked on in parallel, you can bring more developers to bear on a problem without them getting in each other’s way. It can also be more comfortable for those developers to understand their part of the system, as they can focus their attention on just one part of it. Process isolation also makes it possible for us to vary the technology choices we make, perhaps mixing different programming languages, programming styles, deployment platforms, or databases to find the right mix.

Perhaps, above all, microservice architectures give you flexibility. They open up many more options regarding how you can solve problems in the future.

However, it’s important to note that none of these advantages come for free. There are many ways you can approach system decomposition, and fundamentally what you are trying to achieve will drive this decomposition in different directions. Understanding what you are trying to get from your microservice architecture, therefore, becomes essential.

# Disadvantages

Service-oriented architecture became a thing partly because computers got cheaper, so we had more of them. Rather than deploy systems on single, giant mainframes, it made more sense to make use of multiple cheaper machines. The service-oriented architecture was an attempt to work out how best to build applications that spanned multiple machines. One of the main challenges in all of this is how these computers talk to each other: networks.

Communication between computers over networks is not instantaneous (this has something to do with physics). This means we have to worry about latencies—and specifically, latencies that far outstrip the latencies we see with local, in-process operations. Things get worse when we consider that these latencies will vary, which can make system behavior unpredictable. And we also have to address the fact that networks sometimes fail—packets get lost; network cables are disconnected.