Adam Drake

Subscribe to newsletter? Yes!

- Latest |
- About |
- Case Studies |
- Contact |
- Press

Struggling to hire developers? Check out ApplyByAPI!

Enough with the microservices

May 20, 2017

Share this: twitter // facebook // linkedin // google+

Tl;dr:

> Don't even consider microservices unless you have a system that's too complex to manage as a monolith. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.
>
> – Martin Fowler
>
> If you can't build a well-structured monolith, what makes you think microservices is the answer?
>
> – Simon Brown

Intro

Much has been written on the pros and cons of microservices, but unfortunately I'm still seeing them as something being pursued in a cargo cult fashion in the growth-stage startup world. At the risk of rewriting Martin Fowler's Microservice Premium article, I thought it would be good to write up some thoughts so that I can send them to clients when the topic arises, and hopefully help people avoid some of the mistakes I've seen. The mistake of choosing a path towards a given architecture or technology on the basis of so-called *best practices* articles found online is a costly one, and if I can help a single company avoid it then writing this will have been worth it.

Context

Microservices are still (unfortunately) currently a big thing and a tech buzzword *du jour*. The approach has been around forever (Service-Oriented architecture anyone?) but for most of the companies I encounter, microservices aren't simply a waste of time or a distraction. They actually make things **worse**.

This might seem strange, because most of the articles on microservices are extolling their countless virtues like decoupling tech systems, better horizontal scalability, removing dependencies between development teams, and so on. If you're the size of Uber, Airbnb, Facebook, or Twitter, then that is all probably true. I've helped larger organizations with their microservices transitions, including helping them set up messaging systems and other technologies which allow for amazing scalability properties. However, for growth-stage startups, all of that technology, and those microservices, are rarely needed.

> You are not Netflix, stop trying to be them!
>
> –Russ Miles

Russ Miles makes this his first point in his post on 8 ways to lose at microservices adoption, and it's something I see all the time. Growth-stage startups often want to emulate the *best practices* they see at those companies, often to their own detriment. Best practices are context dependent. What is a best practice for a company like Facebook, may or may not be a best practice for a startup with a total engineering team size of 100 or less.

Even if you are smaller than the tech giants, you still might, on the balance, get some benefit from moving towards a microservices architecture. However, a growth-stage startup doing a wholesale migration to microservices should be a firing-level offense for the tech people involved.

Why microservices?

Usually, in growth-stage startups, the main motivation for moving to microservices is that hope that doing so will remove dependencies between development teams and/or improve the ability of the system to handle larger traffic loads (i.e., scalability). Common complaints and symptoms are things like merge conflicts, bugs in deployment due to parts of the application not being ready to use partially implemented features, and horizontal scalability. Let's break these down individually.

Dependencies

In an early-stage startup, the dev team is small and the tech is small. People can work well together without stepping on toes and it seems like everything is relatively fast to implement. Life is good.

As the startup grows, the dev team grows, and the codebase grows, and soon there are multiple teams working on the same codebase. These teams are often largely composed of the people who were around at the earlier stage of the startup. Since many early-stage startups are a first job for many junior developers, they don't realize that the communication effectiveness has to increase as the team size and codebase size increases. As is often the case for tech people who have limited experience, they reach for a technical solution to a people problem, and decide they need microservices in order to reduce dependencies or coupling between dev teams.

In reality, they need to address the people-related problems via more effective communication. When a startup has multiple dev teams, it is a requirement that they stay coordinated and informed about everyone's work. They need to collaborate. Software development, in any organization of this size, is a social endeavor. If there is little communication or information sharing between teams, they will have the same dependency problems, with or without the microservices. However, with the microservices, they will also have all of the negative technical problems that come attached.

It is true that keeping the code modular, as a tech solution to the problem, can mitigate some of the inter-team dependencies inherent in software development, but the communication component still must be grown and improved as the team size grows.

Don't confuse **decoupling** with **distribution**. You can achieve decoupling by having a monolith, composed of well-defined modules, with well-defined interfaces, and you **should**. You don't need to **distribute** your application as separate services in order to benefit from the **decoupling** you get by having modules with clear interfaces.

Partial-feature implementation

This point is often addressed by feature flags, which are something you might need to be familiar with to successfully implement microservices anyway. Especially as you get into **rapid deployment** (discussed more below), you may need to deploy parts of features which are not yet ready on certain platforms, or where the frontend implementation is complete but that backend is not, and so on. As companies grow, and deployment and ops systems become more automated and complex, feature flags are something important to use and to use wisely.

Horizontal scalability

This point has some merit in that multiple copies of the same microservice can be deployed in order to achieve a form of scalability. However, most companies that adopt microservices too early will use the same storage subsystem (most often a database) to back all of their microservices. What that means is that you don't really have horizontal scalability for your application, only for your service. If this is the scalability method you plan to use, why not just deploy more copies of your monolith behind a load balancer? You'll accomplish the same goal with less complexity. Not only that, but the complexity that accompanies horizontal scalability should only be borne as a last resort. Your first effort should be in taking reasonable steps to improve your application performance. Often, even basic things can result in performance hundreds of times faster than the original system, and that also includes wisely using services which support your application. I mentioned in my post on Redis performance triage, for example.

Are we ready for microservices?

This question is often never even considered when deliberating what an architecture should resemble, but it should be. Many times, the senior technologists in a company simply identify complaints or pain points of developers or of the business, and then find something on the Internet claiming microservices architectures address those issues. This claim has many caveats. Microservices, like many things, come with positive and negative effects. If your organization is mature enough, and has the tech in place, then the challenges associated with having microservices can be minimized, making the positive effects all the more apparent. So what does it mean to be ready for microservices? Martin Fowler wrote down his Microservice Prerequisites years ago, and in my experience most growth-stage startups have totally ignored him. Martin's prerequisites are a good place to start, so let's consider them.

1. Rapid provisioning
2. Basic monitoring
3. Rapid deployment

I can tell you from experience with dozens of growth-stage startups that almost none of them have even one of these prerequisites in place, never mind all of them. If your tech team doesn't have the ability to quickly provision, deploy, and monitor all of your current systems, you **must** gain that capability before you consider migrating to microservices. Let's consider each of these prerequisites in a bit more detail.

Rapid provisioning

If your organization has only one or a few people in your entire dev team who can set up new services, virtual or otherwise, you are not ready for microservices. You will need multiple members in each team with the ability to provision infrastructure and deploy services to that infrastructure without requiring outside assistance. Remember, if you have a *DevOps Team*, then you are absolutely not doing DevOps. Developers should be involved in managing everything about their applications, including infrastructure.

Likewise, if your current architecture is not backed by flexible infrastructure that is easy to scale up and down and can be managed by various people in the teams, you must address this before moving towards microservices. You can of course have microservices running on your own bare metal machines, and you

may have superior performance for lower cost, but you must still be able to have flexibility on ops and deployment of your services.

Basic monitoring

If you don't monitor the system and application performance of your monolith, then you will have a miserable time with microservices. You need familiarity with system level metrics (such as CPU and RAM), application level metrics (such as request latency per endpoint, or errors per endpoint), and business level metrics (such as transactions per second, or revenue per second) to understand the performance of your systems. For all the complexities of the monolith, an ensemble of microservices is far more complex to understand, let alone troubleshoot, when it comes to performance. Set up something like Prometheus and add all the necessary instrumentation to your monolith before carving parts of it out into microservices.

Rapid deployment

If you don't have a good continuous integration and deployment process and system in place for your monolith, then trying to manage integration and deployment for your microservices will be nearly impossible. Imagine having 10 teams and 100 services, all of which require manual integration testing and deployment. Now imagine the same manual work, but with only one monolith. How many ways can things go wrong with 100 services? How many ways with 1 monolith? This prerequisite is an excellent example of the complexity that comes along with a microservices approach.

Fowler's list has also been extended with a couple of additional prerequisites by Phil Calcado, but I would say those are more along the lines of important extensions than true prerequisites.

What if we have the prerequisites?

Even with the prerequisites in place, it is important to consider the negatives of microservices in order to make sure the approach really makes sense for your business. The simple fact is that a lot of tech people pretend that the fallacies of distributed computing are somehow not a concern in the microservices world, but all of those things must be taken into account in order to be successful. For most growth-stage startups, there are just too many reasons to avoid microservices.

Increased operational overhead

This is partially covered by the **Rapid Deployment** prerequisite, but consider that with microservices often come lofty aims of containerizing everything (probably with Docker) and using something like Kubernetes to orchestrate all of it. While both systems are wonderful pieces of technology in many ways, for most growth-stage startups they can be a distraction. I've seen startups scale up to fantastic levels using `rsync` for their deployment and orchestration. I've also seen many more startups get stuck in the quagmire of hugely complex ops tooling, which ends up stealing away valuable time they could be using to build features for customers.

Your app can get slower

If you have multiple modules in your monolith, with well-defined APIs, then you have nearly zero overhead when interacting with those APIs. This is definitely not the case with microservices, since they are often running on other machines and require a network hop between your services. This can slow down your whole system considerably. This situation becomes even worse if you have some services which need to contact multiple other services, synchronously, in order to complete a request. I have worked with companies that had nearly 10 services, which had to be called, in order, for a request to be serviced. At each step they have network overhead and other delays to service the request, and they could have easily put all of those services into one artifact, perhaps as different modules, and probably done some additional redesign to make things asynchronous. It could have saved them an order of magnitude on their infrastructure costs.

Local development is more difficult

If you have one monolith, probably backed by a database, then getting your application to run locally during your development process is pretty easy. If you have 100 services, possibly with multiple datastores that may have dependencies, now local development can be an absolute nightmare. Even Docker containers won't save you from this level of complexity. They can make things easier of course, but you'll still have to deal with the dependency issue somehow. Microservices, in theory, remove this requirement because each service is supposed to be independent from the start. However, for growth-stage startups, that is almost never the case. People usually need to have all (or nearly all) the services running on their machine in order to properly develop and test new features. This complexity is extremely wasteful.

It can be harder to scale

The easiest way to scale a monolith is to simply deploy additional copies of your monolith behind a load balancer. This is a dead-simple way to scale up if your system receives more traffic, and it involves minimal additional complexity from an operations perspective. The longer your system can survive on something like Elastic Beanstalk, the better. That will keep you and the team free to work on actually building things for customers instead of battling with your deployment pipeline. Some of this pain can be mitigated by having the proper CI/CD systems as in the **Rapid Deployment** prerequisite, but things get a lot more complex when you're in the microservices world, and often that complexity is more trouble than it's worth.

Now what?

If you're in a growth-stage startup with the need to make some changes to your architecture and microservices aren't the answer they seem to be, what is it that you **should** be doing?

It's important to note that Fowler's prerequisites are something of a Capability Maturity Model for tech, and of course Fowler does have his own article on the topic of a Maturity Model. **IF** it makes sense for the company, we can use his prerequisites and take other intermediate steps to prepare for a move to microservices. To quote Fowler:

> The vital point here is that the true outcome of a maturity model assessment isn't what level you are but the list of things you need to work on to improve. Your current level is merely a piece of intermediate work in order to determine that list of skills to acquire next.

So how do we look for things to improve, and what path should we take to get there? There are a few simple, general steps. The first two **alone** will typically solve many of the problems that cause teams to move towards microservices, and without all the associated complexity.

1. Clean up the application. Make sure it has good automated tests and is using the current versions of all libraries, frameworks, and languages.
2. Refactor the application into clear modules with clear APIs. Don't allow bits of the code to reach into the modules directly. All interaction should be via the APIs presented by the module.
3. Choose one module in the application and split it into its own application on the same host. This starts to give you some of the usefulness of totally separate microservices, but with fewer of the operations headaches. You will, however, still have to cope with communication between two different components, albeit on the same host. That fact allows you to disregard some of the complexity inherent in network partitions and availability on fully distributed systems like a microservices architecture.
4. Take the separated module and put it on a different host system. Now you'll have to deal with the issues surrounding communication over a network, but you will have bought yourself a little less coupling between the two systems.
5. If possible, refactor the data storage system so that the module on the other host now has total responsibility for storage of data within its context.

Even at larger scales, almost every company I've seen only really needs the first two steps in order to be happy. If they can do well with those, the remaining steps aren't always as important as they originally thought. Better still, if you decide to stop at any point along the way, the system is still maintainable and probably in better shape than it was when you started.

Outro

I can't claim that any of these ideas are unique, or that I came up with them. This is simply a summary of observations and thoughts I've collected from others who seem to have encountered the same thing. Many others who have been around a lot longer than I have written about these topics, and some with a lot more clarity, like Sander Mak's article on Modules and Microservices. Either way, these lessons are important and hopefully useful for companies considering what to do with their architectural future. Consider all options carefully, and make sure that microservices are an appropriate path for your organization.

At least start with the first two steps in the section above, and **after** those are complete, then consider again whether microservices are the right direction for your organization. Chances are, a lot of the issues you had previously will simply disappear.

Adam Drake leads technical business transformations in global and multi-cultural environments. His passion is to help companies become more productive by improving internal leadership capabilities and accelerating product development through technology and data architecture guidance. He has a background in Applied Mathematics.