highscalability.com

Do you have too many microservices? - Five Design Attributes that can Help - High Scalability -

11-14 minutes

This is a guest Post by Jake Lumetta, Founder and CEO, ButterCMS, an API-first CMS. For more content like this, follow @ButterCMS on Twitter and subscribe to our blog.

Are your microservices too small or too tightly coupled? Are you confident in your decision-making about service boundaries? In interviews with dozens of experienced CTOs, they offered design attributes that they consider when creating a set of microservices. This article distills that wisdom into five key principles to help you better design microservices.

The importance of microservice boundaries

The design attributes discussed below matter because reaping the benefits of microservices requires designing thoughtful microservice boundaries.

One of the major challenges when it comes to creating a new system with a microservice architecture. It came about when I mentioned that one of the core benefits of developing new systems with microservices is that the architecture allows developers to build and modify individual components independently — but problems can arise when it comes to minimizing the number of callbacks between each API. The solution according to McFadden, is to apply the appropriate service boundaries.

But in contrast to the sometimes difficult-to-grasp and abstract concept of domain driven design (DDD) — a framework for microservices — I'll be as practical as I can in this chapter as I discuss the need for well defined microservice boundaries with some of our industry's tops CTOs.

First, avoid arbitrary rules

When designing and creating a microservice, don't fall into the trap of using arbitrary rules. If you read enough advice, you'll come across some of the rules below. While appealing, these are not proper ways to determine boundaries for microservices.

Arbitrary Rule 1: "A microservice should have X lines of code"

Let's get one thing straight; there are no limitations on how many lines of code there are in a microservice. A microservice doesn't suddenly become a monolith just because you write a few lines of extra code. The key is ensuring there is high cohesion for the code within a service (more on this later).

Arbitrary Rule 2: "Turn each function into a microservice"

If a function that computes something based on three input values, and returns a result, is that a good candidate for a microservice? Should it be a separately deployable application of its own? This really depends on what the function is and how it serves to the entire system.

Other arbitrary rules

Other arbitrary rules include those that don't take into account your entire context such as the team's experience, DevOps capacity, what the service is doing and availability needs of the data.

Five characteristics of a well-design service

If the arbitrary rules outlined above don't guide thoughtful design of microservices, then what does? Many articles will tell you that the answer is simple: high cohesion and loose coupling. This is sound advice, but it's quite abstract. In interviews, CTOs offered some practical advice that I've distilled into five key characteristics of well-designed services.

Characteristic #1: It doesn't share database tables with another service

In the early days of SparkPost Chris McFadden, VP of Engineering at SparkPost, and his team had to solve a problem that every SaaS business has: they needed to provide basic services like authentication, account management, and billing.

The core problem, of course, wasn't how to charge their users money. It was how to design their user account microservices to support everything that goes along with that problem domain: user accounts, API keys, authentication, business accounts, billing, etc.

To tackle this they created two microservices: a Users API and an Accounts API. The Users API would handle user accounts, API keys, authentication and the Accounts API would handle all of the billing related logic. A very logical separation, but before long, they spotted a problem.

"We had one service that was called the User API, and we had another one called the Account API. But the problem was that they were actually having several calls back and forth between them. So you would do something in accounts and have call and endpoint in users or vice versa," he continued.

The two services were too tightly coupled.

"We realized that in most cases, you really don't want to have one service calling another service in a sometimes circular way. That's generally not a good idea," McFadden explained.

When it comes to designing microservices, if you have multiple services referencing the same table, that's a red flag as it likely means your DB is a source of coupling.

It is really about how the service relates to the data, which is exactly what Oleksiy Kovrin, Head of Swiftype SRE, Elastic, told me:

"One of the main foundational principles we use when developing new services is that they should not cross database boundaries. Each service should rely on its own set of underlying data stores. This allows us to centralize access controls, audit logging, caching logic, et cetera," he said.

Kovyrin went on to explain that if a subset of your database tables, "have no or very little connections to the rest of the dataset, it is a strong signal that component could be isolated into a separate API or a separate service."

Darby Frey, Co-founder of Lead Honestly, echoed this sentiment by telling me that, "each service should have its own tables [and] should never share database tables."

Characteristic #2: It has a minimal amount of database tables

The ideal size of a microservice is to be small enough, but no smaller. And the same goes for the number of database tables per service.

Steven Czerwinski, Head of Engineering, Scaylr, explained to me during an interview that the sweet spot for Scaylr is, “one or two database tables for a service.”

“We have a suppression microservices, and it handles, keeps track of, millions and billions of entries around suppressions but it's all very focused just around suppression so there's really only one or two tables there. The same goes for other services like webhooks” explained McFadden.

Characteristic #3: It’s thoughtfully stateful or stateless

When designing your microservice, you need to ask yourself whether it requires access to a database or is it going to be a stateless service processing terabytes of data like emails or logs.

Julien Lemoine, CTO of Algolia, explained, “We define the boundaries of a service by defining its input and output. Sometimes a service is a network API but it can also be a process consuming files and producing records in a database (this is the case of our log processing service).”

Clarity about this upfront will lead to a better designed service.

Characteristic #4: Its data availability needs are accounted for

When designing a microservice, you need to keep in mind what services will rely on this new service and what the system-wide impact will be if that data becomes unavailable. Taking that into account allows you properly design data backup and recovery systems for this service.

When speaking to Steven Czerwinski, he mentioned their critical customer row space mapping data is replicated and separated in different ways due to its importance.

“Whereas the per shard information, that's in its own little partition. It sucks if it goes down because that portion of the customer population is not going to have their logs available, but it's only impacting 5 percent of the customers rather than 100 percent of the customers," Czerwinski explained.

Characteristic #5: It's a single source of truth

The final characteristic to keep in mind is to design a service to be the single source of truth for something in your system.

To give you an example, when you order something from an eCommerce site, an order ID is generated. This order ID can be used by other services to query an Order service for complete information about the order. Using the pub/sub concept, the data that is passed around between services should either be the order ID, not the attributes/information of the order itself. Only the Order service has complete information and is the single source of truth for a given order.

Additional considerations for larger teams

In addition to the five considerations above, larger teams should be aware of a few additional considerations. For larger organizations, where entire teams can be dedicated to owning a service, organizational consideration comes into play when determining service boundaries. And there are two considerations to keep in mind: independent release schedule and different uptime importance.

"The most successful implementation of microservices we've seen is either based on a software design principle like domain-driven design for example, and service-oriented architecture or the ones that reflect an organizational approach," said Khash Sajadi, CEO of Cloud66.

Amazon is a perfect example of a large organization with multiple teams. As mentioned in an article published in API Evangelist, Jeff

Bezos issued a mandate to all employees informing them that every team within the company had to communicate via API. Anyone who didn’t would be fired.

This way, all the data and functionality was exposed through the interface. Bezos also managed to get every team to decouple, define what their resources are, and make them available through the API. Amazon was building a system from the ground up. This allows every team within the company to become a partner of one another.

I spoke to Travis Reeder, CTO of Iron.io, about Bezos’ internal initiative.

"Jeff Bezos mandated that all teams had to build API's to communicate with other teams. He's also the guy who came up with the 'two pizza' rule; a team shouldn't be larger than what two pizzas can feed,” he said.

“I think the same could apply here: whatever a small team can develop, manage and be productive with. If it starts to get unwieldy or starts to slow down, it's probably getting too big," Reeder told me.

Determining whether a service is too small or not properly defined

During the testing and implementation phase of your microservice system, there are a number of indicators to keep in mind.

Is there any over-reliance between services?

If two services are constantly calling back to one another, then that’s a strong indication of coupling and a signal that they might be better off combined into one service.

In Sparkpost’s experience with its two API microservices, accounts and users, the services were constantly communicating with one another. McFadden came up an idea to merge the services and decided to call it the Accuser's API. This turned out to be a fruitful strategy:

"What we started doing was eliminating these links [which were the] internal API calls between them. It's helped simplify the code," McFadden informed me.

Does the overhead of setting up the service outweigh the benefit of having it be independent?

Darby Frey of Lead Honestly explained, "Every app needs to have its logs aggregated somewhere and needs to be monitored. You need to set up alerting for it. You need to have standard operating procedures and run books for when things break. You have to manage SSH access to that thing. There's a huge foundation of things that have to exist in order for an app to just run."

Recap: consider these characteristics

Designing microservices can often feel more like an art than a science, but there are principles to follow. There's lots of general advice out there but at times it can be a bit too abstract so let's quickly recap five specific characteristics to look for when designing your next set of microservices:

1. It doesn't share database tables with another service
2. It has a minimal amount of database tables
3. It's thoughtfully stateful or stateless
4. Its data availability needs are accounted for
5. It's a single source of truth

Next time you're tasked with creating the boundaries for new microservices, referring to these characteristics should make that task much easier.