Isolating your microservices through loose coupling

Christopher Laine Follow
May 6, 2019 · 6 min read

Photo by Ksenia Makagonova on Unsplash

One of the biggest issues we can face when dealing in microservices is to ensure each microservice operates in complete isolation from its microservice siblings.

When we first get started with microservice architecture, *it's* easy to imagine that microservices should be like little pals, all of them operating together like chummy relations who all work together to make a product / solution come together.

Well, sort of. *In a way, this is true.* We clearly have a product / solution offering, and in order for this all to come together, we need our microservices to collaborate (sort of) to make this happen. If they don't make it happen, then our product / solution isn't worth it's salt.

So yes, our microservices need to communicate with each other. They need to route messages to one another; they need to ensure they are all able to consume appropriate commands / events of those which interest them. Yet at this point, we must pause and take stock. The question we must ask ourselves is what does this look like?

. . .

Coming from an old-school application background, I can recall the days of SOAP and before. For those unfamiliar, SOAP was (I say 'was' because if you're NOT a lunatic or trapped with a legacy app, there's no chance you'll be using it these days) a communications protocol come about in the latter 90s, which defined a means of two disparate systems to communicate over HTTP via a standard XML syntax.

SOAP was a massive leap forward in RPC, in that it standardised how two systems could talk to one another without code dependencies or specialised systems to translate messages between them. So, in this way, SOAP was a big and critical precursor to our modern usage of REST.

The SOAP definition not only defined the endpoints / web methods one could call against a server, but also the message contract one needed to provide in order to make a method call. At the time, this seemed amazing. Oooh, a standardised way of knowing what kinds of contracts are necessary to call a method? How nifty!

So for many, many years, hundreds of thousands of systems used these SOAP contracts to communicate with one another, and everything was great.

Except when it wasn't. Thing was, in theory these contracts ensured that clientA could call serviceB with just the data it required. In reality, what this meant was that clientA was *tightly-coupled* to serviceB. If serviceB's contract needed to change in any way, then clientA is wrecked, because it's contracts were no longer valid. Only by regenerating / downloading serviceB's new contracts could clientA continue working.

On small projects, this was a hassle. On enterprise systems, it was like death. SOAP, like most old RPC systems, showed it's worst quality in it's very structure and discipline. Tightly-coupled contracts meant that no one could move unless everyone did. A bad solution in the best of times.

With the growing interest and adoption of REST, we saw a way out. REST by its nature is less tightly-coupled, so that clientA can call serviceB, not by a method name but by a URL, and by providing a more nebulous form of data. Definitions of contracts were not enforced *per se*, but instead were defined in consumable documentation (Swagger), and contract versions manageable by content negotiation, HATEOAS, and other means.

Even still, many developers and architects have continued to emotionally cling to the SOAP bad old days, wherein the contracts between a server and a client are to be well-defined and strictly-enforced. Semantically-versioned contracts is a common pattern, where the contracts themselves are held in a separate package, and compiled for consumption by consumers. Another is the published "ServiceClient" which exposes all methods and contracts for the consuming clientA to call serviceB. Both are architecturally troubling, because once more these contracts or clients are nothing more than another form of tight-coupling.

Once again, nobody moves unless everybody does, just not in as rigid a format as SOAP provided.

How to avoid tight coupling?

In any communications between services (microservice or otherwise), it is important to go above and beyond to avoid tight-coupling. In microservice architectures, it is especially important, as we want to ensure that our services keep a cool, wary distance from one another however / whenever possible. Tight-coupling means that changes to resources / contracts on *microserviceA* not only effect itself, but also any / all consumers of its data. If we update the contracts of microserviceA, we must (or at least should) redeploy all consumers of those contracts. In a small system, this might mean the redeployment of five or six consuming sibling microservices. In a massive system, this could mean a gigantic piece of redeployment.

In any architecture, it is risk aversion we are after. Redeployments of any kind are a risk. The more redeployment required, the higher the risk factor. We deploy code with the implicit understanding that the risk of failure is worth taking because the deployment is necessary to advance our product. But the ROI must be balanced. Too much risk from too much redeployment is unacceptable.

So how does one avoid tight coupling? Here are some great techniques / food for thought when considering your communications protocols between microservices. I suggest reviewing these and thinking of how you can bring these to bear in your product / architecture.

Define contract ownership at the consumer, not the publisher

Rather than defining a contract as a rigid thing owned by the exposing service, expose your contracts via Swagger or some such publication. Each consumer of this microservice's data is now responsible for the management and updating of their own copies of these contracts. While this leads to code duplication / data drift across your microservices, it also means that each consumer can be updated to the most recent version of the contracts as they see fit. As long as you do not introduce breaking changes in your contract changes, then the older consumers will know no differently than before.

Allowing the consumer to own the contract means that things can change around it without it being adversely affected. I might have microserviceA (the originator of the data) which changes rapidly, updating its contracts frequently, and then have microserviceB, which is never changing, and doesn't need to be kept up-to-date with all the fancy changes being made to microserviceA's contracts. I might wait a week, a month, even a year, before I decide to update microserviceB's copy of the contracts, and if I've done my job correctly, then this shouldn't matter.

Consider content negotiation as a contract version strategy

Top highlight

If you feel compelled as a dev or architect that contracts should be versioned, then consider using content negotiation between client and server. Content negotiation avoids the need for specific contracts in favour of versioned contracts being requested by the client from the server.

Your client, instead of just calling a REST endpoint, provides additional details about which version it is able to consume. As in

```
GET /users/someid HTTP/1.1
Accept: application/vnd.myservice.v1+json
```

This is by no stretch a perfect solution, just an URL versioning is fraught with its own potential issues. Contract versioning / API versioning is a big decision, so take some time to get to know the topic before you choose.

Ben Morris' excellent post on API versioning will give you some good insight into things to consider when embarking down this road.

One big issue one faces with content-negotiation is that now the API responsible for this kind of versioning has to handle two or more versions appropriately. This can lead to bloat in your API controllers, handling lots of versions of contracts.

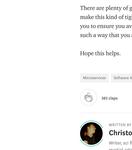
Keep this to a minimum. If you have two contract versions, you should be okay. Beyond that, you're asking for a kick in the teeth with your own boot.

Avoid DNS / URLs in favour of discovery services / service registries

Traditionally, in order to find and communicate with an API, we would use DNS as a means of getting to the service in question. This is old school, and if you can do so, dodge this like a speeding train about to hit you.

You see, DNS and such URLs are just another form of tight-coupling. If I need to reach http://someurl/api/somemodpoint (this being the DNS entry for my microserviceA), and this URL / DNS is missing, down, or changed, my consumer is SOL. In modern microservices, DNS is a much more fluid thing, and one which should not be used as a hard address to your microservice. Your microservice could be in flux, or made up of a container cluster, each assigned its own route and held as a single logical unit by a load balancer.

All modern cloud platforms offer some version of the Discovery Service and/or the Service Registry. In them, your microserviceA is registered as a discoverable service, using some unique key as the name definition for it. A Service Registry stores this relationship for you, so that other services can reach your microserviceA by this unique key, rather than by DNS. When a client calls, it goes through a router / Discovery Service to reach microserviceA. This in turn calls on our Service Registry to give us access to the microserviceA we are after. Chris Richardson provides a tidy introduction to Discovery Services and Service Registries.



Keep the chattiness to a minimum

Yes, your microservices need to talk to one another. Clearly this is the case. However, if you find this communication is excessive / chatty, then something is wrong. Excessive communication between two or more microservices is, in and of itself, a form of tight-coupling. Excessive communications means that whatever microserviceA is doing is inextricably linked to its sibling(s). This means you've not created a series of microservices, but instead a single piece of functionality which is dressed up to look like microservices. If you need this kind of communication, decouple it through a ServiceBus and Events (as in, OrderCreatedEvent, fired and forgotten by microserviceA, which is then consumed by microserviceB,C, or D). If direct communication is required, try to ensure that you minimise calls as best you can, through caching.

This one is tricky, as the right balance can be hard to quantify. Let me put this another way, then: You'll know very quickly that the chattiness is too much when every call to microserviceA spawns at least one additional call to *microserviceB*. If this is the case, step back and look at the problem you are solving, as well as how you've solved it. Think you'll find that there is a better way of modeling your architecture so that microserviceA is not so dependent on its sibling(s) to do its job.

In Conclusion

There are any number of ways to reduce the type of tight-coupling I am talking about, and I've covered just a few. I chose the ones I see the most often, and have suffered the most from in my career. However, the best rule of thumb I tell developers / architects alike is always imagine your microservice stands alone. It should be kept apart from its siblings (either consumers or required publishers) as often and as discreetly as possible. Any direct connection to / from a microservice is a difficult proposition, so take your time and model this out as best you can at the start.

There are plenty of great tools and cloud services out there nowadays which make this kind of tight-coupling totally unnecessary. However, it is up to you to ensure you avail yourself of them, and you build your platform in such a way that you avoid this kind of pain before it begins.

Hope this helps.

Microservices Software Architecture Software Development DevOps Advice

Written by
Christopher Laine Follow
Writer, sci-fi / Lowcraftian nutbag, "master" chef, gym rat, martial artist, Dungeon Master, and programmer. I cover all the useless bases