# Why I stopped using Microservices

APRIL 13, 2020 BY ROBIN WIERUCH - EDIT THIS POST



I've always been fascinated by APIs. In fact APIs, distributed systems, and web services are the reason why I learned to code. When I started my first job as a junior frontend developer, it wasn't foreign to me to interact with a backend API. I was always fond of decoupled client-server architectures. When I started to be self-employed in 2017 and started to work as a consultant for clients, I got confronted with microservice architectures more often. This brought the whole decoupling of services on a next level for me.

While I was working with clients on *their* projects, I extended my online portfolio with websites and side-projects too. One of these side-projects turned out to be a platform for selling self-published ebooks as courses, which I had started at the same time as getting self-employed. It goes beyond just the selling, because it also offers features like coupon codes and partner programs.

So what does my web application have to do with microservices? Since I was always drawn towards decoupling my APIs, I went down this road as a solo developer too. As it turns out, it was too much for one single person. Who would have thought this ;-) Hence the clickbaity title why I stopped using Microservices, which doesn't imply that companies with multiple teams should stop using them.

*Disclaimer: I do not claim to be a microservice expert here. I am just experimenting. As a solo developer on my side-projects, I didn't push the microservice scenario too far, which means I didn't go beyond 5 microservices before I gave up on this approach and I didn't use K8S or any other of these more sophisticated tech stacks. I host all my projects, whether they are websites, web applications or APIs, on DigitalOcean.*

Let's start with the good parts and end with the bad parts.

- **Software Craftsmanship (+):** I like to build things. Going beyond a decoupled client-server architecture with just two entities (frontend and backend) was something I always wanted to explore. It's a challenge and whenever you start a side-project not only for the sake of generating an income stream from it, it should be there for learning purposes. So I asked myself: Is it possible to treat user authentication, payment processing, and coupon codes for my web application as decoupled microservices?
- **Decoupling (+):** Beyond the learning, it's all about API design which fascinates me. Is it possible to decouple my payment API from my course API without them not knowing about each others domain specifics? After all, once a payment was successful, it needs to notify the course domain to create the course in the database. In a common monolithic backend application, it's easy to overlook this clear separation of concerns, because one service (module) can easily creep into another service without proper dependency injection. However, if such a service becomes a microservice with just a REST or GraphQL API, you are forced to avoid these missteps.
- **Reusability (+):** Beyond the decoupling of services for *one* project, I was wondering whether it's possible to reuse my payment API or authentication API for other side-projects of mine. After all, it's just too tiresome to develop all these things from scratch for every project. It turned out to be possible, but with a huge caveat (see Abstraction and Mental Overhead).
- **Abstraction (-):** If a microservice should be repurposed for the sake of reusability, one has to treat the microservice with some level of abstraction in mind, because it doesn't handle one specific case anymore. For instance, if the authentication microservice should be repurposed, the API and the service have to distinguish between the projects for which a user authenticates. While this abstraction allows us to avoid the implementation of multiple authentication APIs, which are all doing essentially the same, it adds another level of

complexity to the authentication API which becomes more difficult to maintain for a solo developer.

- **Feature Creep (-):** Starting out with a payment API and course API that have to work together wasn't too difficult. But it doesn't end there in a growing application. Eventually more features and therefore more APIs make their way into your microservice composition. Once I started to use a coupon API, the feature creep started to become more obvious, because it wasn't only a unidirectional communication between payment API and course API anymore. The coupon API had to be used for the frontend application to verify the coupon code, while also being used by the payment API to process the discounted price when a course has been purchased.

- **Mental Overhead (-):** With all this Abstraction and Feature Creep in mind, it became too difficult to reason about all the microservices as a solo developer. The decoupling of the microservices turned into a negative asset. After all, it's just easier to reason about all these things in a monolithic application where everything is closer, even though it just feels mentally closer to each other, and nothing is abstracted for some kind of reusability scenario.

- **Code (-):** Instead of having all the code in one monolithic application, it was distributed among several microservices now. This may turn out as a great benefit when working on an application with multiple teams, because teams can declare certain ownership of their domains optionally, however, going through this as a solo developer was just not sustainable. Everything felt too far away and reusing more general code in one microservice from another microservice wasn't feasible without managing yet another external dependency (e.g. library).

- **Robustness (-):** In theory, having decoupled microservices sounds amazing for isolated testing purposes and robustness of each individual service. However, working alone on this thing and scaling it to multiple microservices didn't make it in any more robust for me. In contrast, managing all these individual code bases with their APIs felt brittle. Not only because they were loosely coupled, but also because the API isn't typed. In a monolithic backend application I can at least make sure that all service to service communication works when using a typed language.

- **Multiple Points of Failure (-):** Without using a more sophisticated tech stack for microservices, over time the composition of services resulted in multiple points of failure. For instance, when having one monolithic application deployed, you immediately know when things are down. However, when having multiple microservices deployed, you have to make sure to get notified properly for every service when things go south. An offline payment API isn't obvious when you navigate through the rest of the application without any errors. However, here again, I guess it would help tremendously to have the resources for a proper infrastructure set-up.

infrastructure setup.

- **Infrastructure Management (-):** Managing all the infrastructure as a solo developer on a side-project is just too much. I did it all by hand with a dedicated DigitalOcean instance which hosts all my APIs, but it not easy to guarantee that everything works as expected. All the CIs and CDs need to work properly when scaling this up, all the code needs to be at the latest stage, and there shouldn't be a flaw for any of the deployed services (see Multiple Points of Failure).

. . .

As you can see, my experience as a solo developer is very much different from companies with multiple teams that I work with, who are able to manage their microservice composition with lots of resources. If I had all the time in the world, I would continue using microservices. However, as a solo developer I stick to one monolithic application which offers me more advantages.





KEEP READING ABOUT JAVASCRIPT›

BEST TIME TO BECOME A JAVASCRIPT DEVELOPER

I started out with JavaScript -- and web development in general -- when everything felt brittle. Developing entire applications with jQuery felt just wrong, the JavaScript APIs (e.g. DOM API) weren't…

NOBODY INTRODUCED ME TO THE API

It might be a common issue in teaching computer science at universities: While you learn about bubble sorts, lambda calculus and permutations, nobody mentors you about common developer subjects. In…


## THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK

Get it on Amazon.

## TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers

✔ Learn Web Development

✔ Learn JavaScript

✔ Access Tutorials, eBooks and Courses

✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.



**PORTFOLIO**

Online Courses

Open Source

Tutorials

**ABOUT**

About me

What I use

How to work with me

How to support me