


# Microservices for Dummies



Konstantin Vassilev

Nov 28, 2019 · 7 min read ★

Follow

🐦

🌐

📘

📺

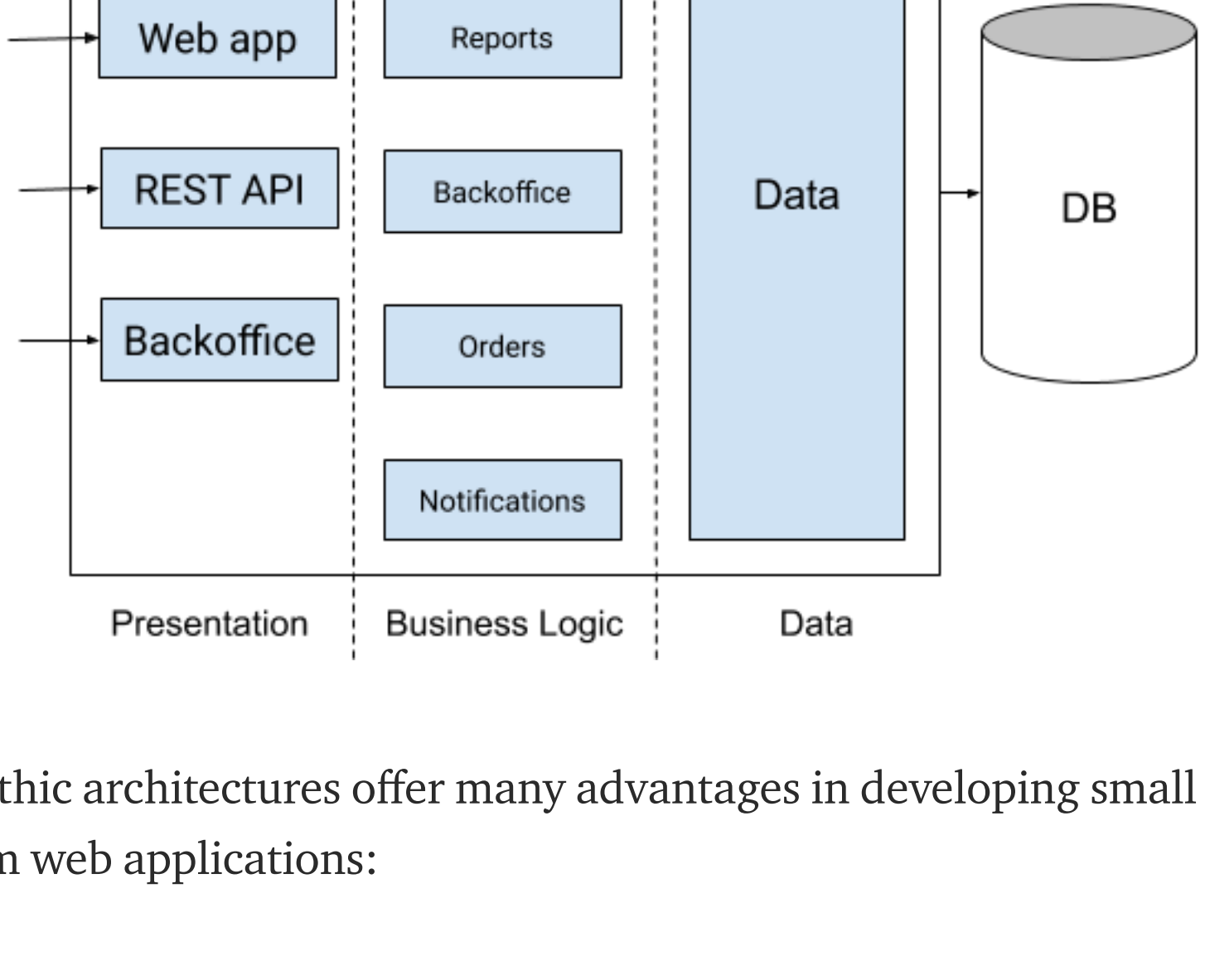
🔖

⋮

co-author [Nikola Toshev](#)

## Why Microservices?

The concept of Microservices came out of a need of solutions to the **problems with monolithic architectures**. We refer to an architecture as monolithic if the **entire app is built into one executable/package, deployed all or nothing, using one or very few data stores**. These applications usually use tiered architectures (e.g. presentation, business logic, data layer) and internal modularization.



Monolithic architectures offer many advantages in developing small to medium web applications:

- Single code base for the entire app, using the same language and libraries
- Simple to deploy — you are deploying just one big service
- Easy to on-board new developers — just build and run the app locally to learn, test and develop

However, once the app reaches a certain age, size and complexity, the drawbacks of the monolithic approach become more significant:

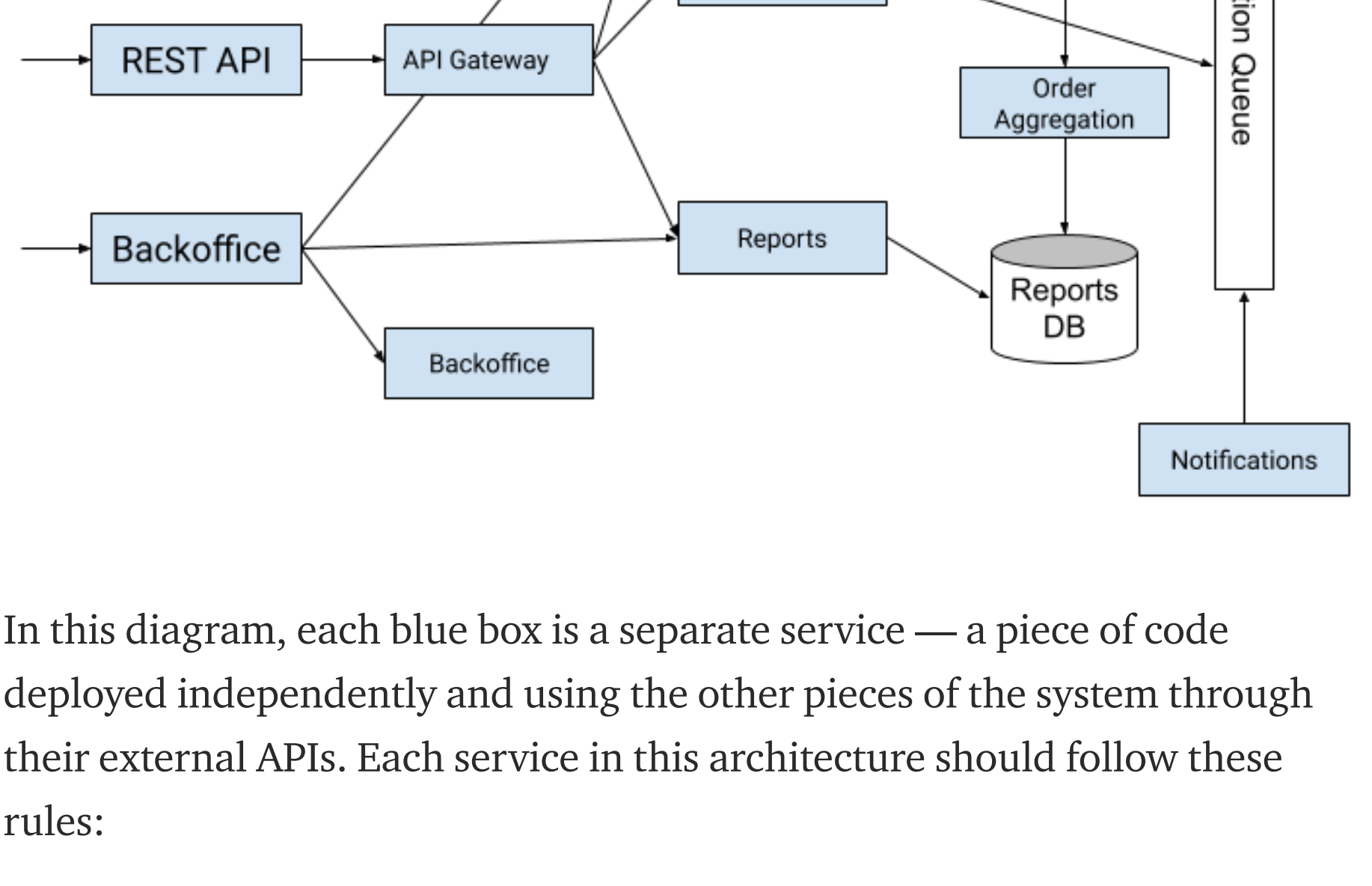
- Development speed slows down — a single code base lacks tools to enforce strict module boundaries and they blur over time. Code becomes **hard to understand and modify safely**. New team members take longer to on-board.
- Large teams working on an app that is deployed in one shot (all or nothing) tend to go into branching hell and some **release train** mechanisms. This **slows features making it to production** and **increases dependencies between teams**, leading to coordination inefficiencies.
- As the application gets very large, the advantage of being able to easily run, modify and debug diminishes. Having the entire system locally at this point leads to **huge startup times** and inflated requirements for the development hardware as you need more memory/CPU/storage to run everything.
- In the early stages of the development of a monolith application, scaling it vertically (a fancy way of saying throw more CPU/RAM in a single machine) is safe and easy. But if the service you are building hits the limits of vertical scaling and you have to resort to horizontal scaling, that becomes not so trivial, since you have just one dimension you can scale. You cannot scale individual components (e.g. scale up the order taking service on Black Friday), so you have to scale the entire monolith (e.g. run it behind a load balancer), but this just shifts the scalability problem to the single database.
- Reliability of the app suffers — as the system is very complex bugs (both in source code and third party libraries) are inevitable. However, due to the monolithic approach, a **bug in any component or library could take the entire app** down. So you end up with your users not being able to log in just because a bug in a library you use for sending notification emails crashes the entire app server.
- Applications that have monolithic architecture tend to have a monolithic code base to reflects that. Usually the code for a monolith is in one language, using same SDK and third party library versions. This makes it both risky and costly to change even simple things like a third party library, since **all** modules of the system must be changed at same point in time. Using a new programming language at some point is prohibitively costly for large monoliths, so they are stuck on an old technology stack for decades.

## What are Microservices?

Microservices are an architectural approach or style to designing large distributed systems aimed at solving the above mentioned deficiencies of the monolithic approach. Microservice architectures enable faster feature delivery and scaling for large applications.

The core idea of microservices is to split the large system into loosely coupled services that can be **deployed independently**. That's it.

So what could the same monolith system from the example above look like in the microservices world? Here is an example:

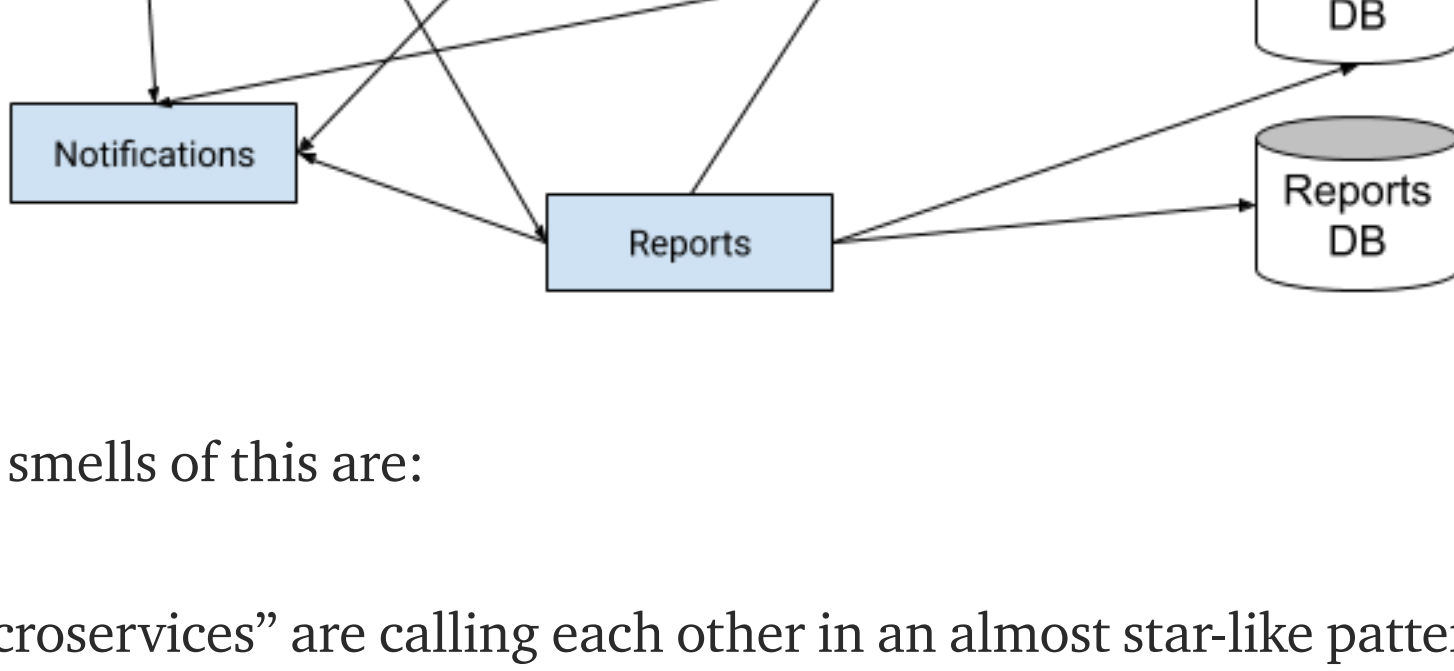


In this diagram, each blue box is a separate service — a piece of code deployed independently and using the other pieces of the system through their external APIs. Each service in this architecture should follow these rules:

### Microservices should be micro

That is, they should be small, specialized services with minimal set of features. But how small is small? There is no magic recipe, and it depends a lot on the domain and the team composition. Here are some guidelines:

- Its common to cap the size at such that it can be developed and maintained by a relatively small team (let's say less than 8)
  - If you are doing a business app, it makes sense to decompose the system by business capabilities.
  - Don't mix services with different reliability requirements into the same microservice. For example, if user-facing services live in the same microservice as report generation, you will not be able to deploy new reports without risk of affecting an important user-facing service.
  - But mostly, use common sense...
- Microservices should be as independent as possible**
- This should be very well covered under “deployed independently”, but many times **too agile** teams overlook this when rushing to pump out features. Its frequent that you end with a spaghetti-microservice architecture, with is architecturally a **monolith with extra RPC calls**.



Typical smells of this are:

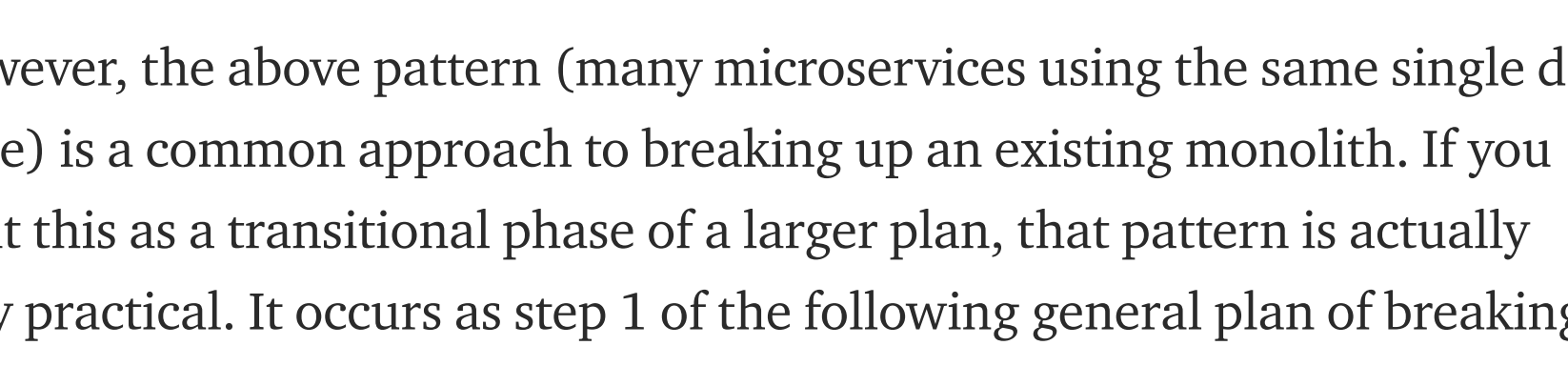
- “microservices” are calling each other in an almost star-like pattern
- “microservices” are accessing multiple data sources, some of them shared with other “microservices”
- Someone took the modules of a monolith and made each class into a microservice, resulting in huge network traffic between these “microservices”.

### Microservices should have their own data store

Top highlight

This follows directly from the requirement to be able to deploy independently. If many services are sharing the same data store, and a change requires modifying the data model, independence is lost. Services using the same data store are **strongly coupled** because of this. While there are valid use cases where 2–3 microservices need to share the same data store, if it's more than that probably some unnecessary coupling has crept in.

If your architecture diagram has a part that looks like this:



you are probably not doing microservices right.

However, the above pattern (many microservices using the same single data store) is a common approach to breaking up an existing monolith. If you treat this as a transitional phase of a larger plan, that pattern is actually very practical. It occurs as step 1 of the following general plan of breaking up a legacy monolith

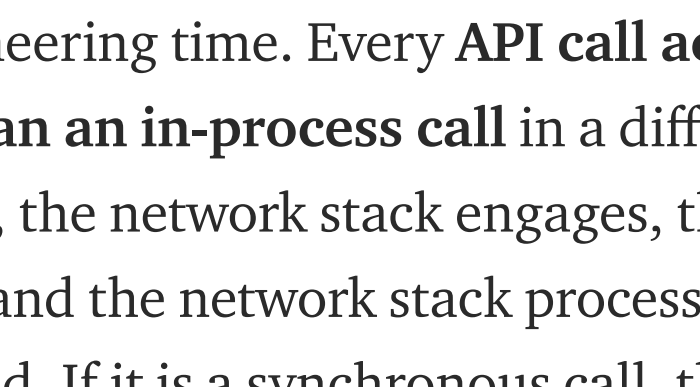
1. Rewrite the service code as microservices, using the old database model. This allows to run the monolith and microservice code side by side on the same DB and test it for feature parity and regressions
2. Microservice by microservice, separate its part of the data in a new DB
3. Repeat 2 until monolith is broken down

### Microservices are typically owned by a small team

One of the main advantages of Microservices is that new developers get on-boarded much faster. To get productive, they need to understand a much simpler system than their counterparts working on monolith systems.

To benefit from this efficiency it is best that one Microservice is owned by a small team. By “owned” here we are talking not only developing, but also deploying and monitoring of the service. Of course, one team can own more than one Microservice, and this is usually the case.

If a service requires a team of more than 10 to develop and maintain, it's worth considering if it is possible to split it in two services and two teams.



## When Microservices are a bad idea?

Microservices **impose tax on crossing service boundaries** on both performance and engineering time. Every **API call across two services is much less efficient than an in-process call** in a different module: data need to me marshalled, the network stack engages, there is latency of the physical data transfer, and the network stack processing and unmarshalling happen on the other end. If it is a synchronous call, there is a thread waiting for a response, keeping stack and other resources allocated (if you went async, then the code is harder to write and you pay in engineering time here). The impact on latency is more severe, but throughput also suffers.

The engineering tax manifests in:

1. Added friction of writing distributed code. Besides the need to implement APIs/RPC, this means also losing static type checking at the API boundary. Since persistent state is distributed, you need to use distributed transactions (which are complicated) or manage failures without transactions (which is also not easy).
2. It becomes harder to implement features that span across more than one service. You typically need to implement downstream services first, and only when they are ready and deployed you can actually implement the UI. This requires team and deployment coordination, which negates the benefits of the microservices in the first place.

This tax pays for easier scaling of engineering time. It is very important to get the boundaries right. When you don't understand the problem very well, which is at the start of pretty much every project, **you are unlikely to get the boundaries correct**. You are also typically starting with a small team that grows and has to build out the product. Therefore you should almost always start with a monolith, but look for well defined boundaries you can split into separate services as the team and the codebase grows.


Thanks to Nikola Toshev.

Microservices

Software Development

Software Architecture

Engineering



321 claps

🐦


🌐

📘

📺

🔖

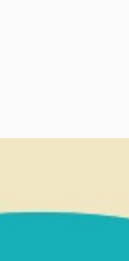
⋮



WRITTEN BY

Konstantin Vassilev

Follow



Sciant

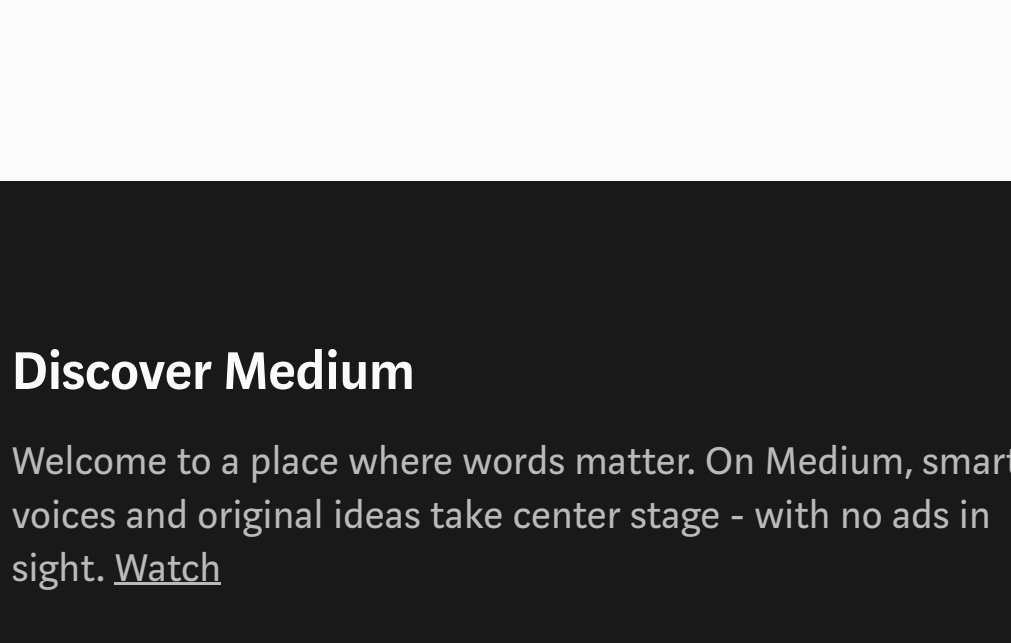
IT and Software Development Services Company

Follow

Write the first response

## More From Medium

Related reads

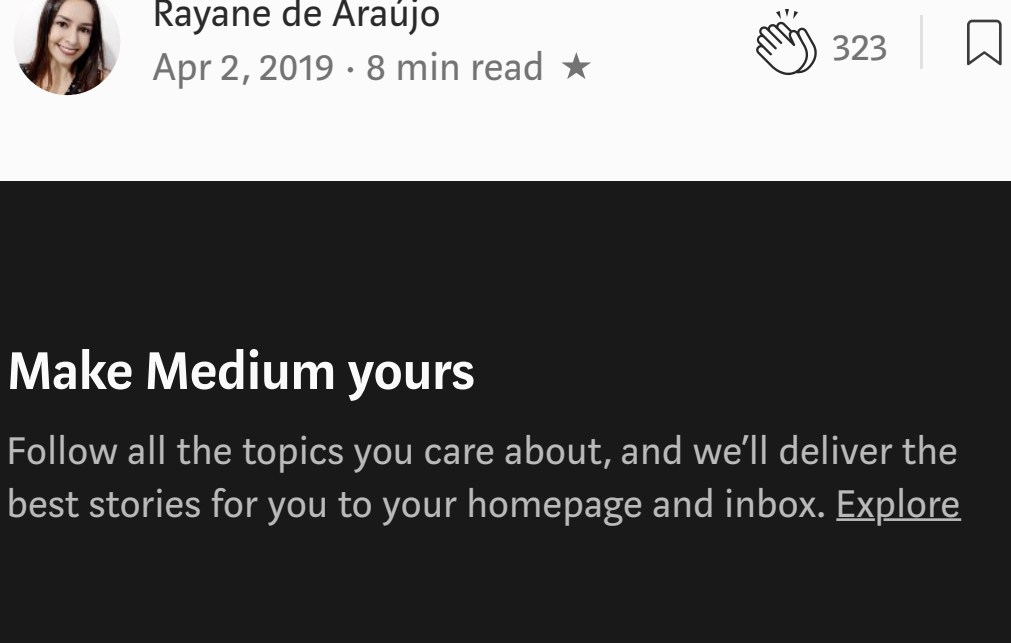


When to Use Event Sourcing

Dirk Hoekstra in ... Programming

Nov 11, 2019 · 3 min read ★

Related reads

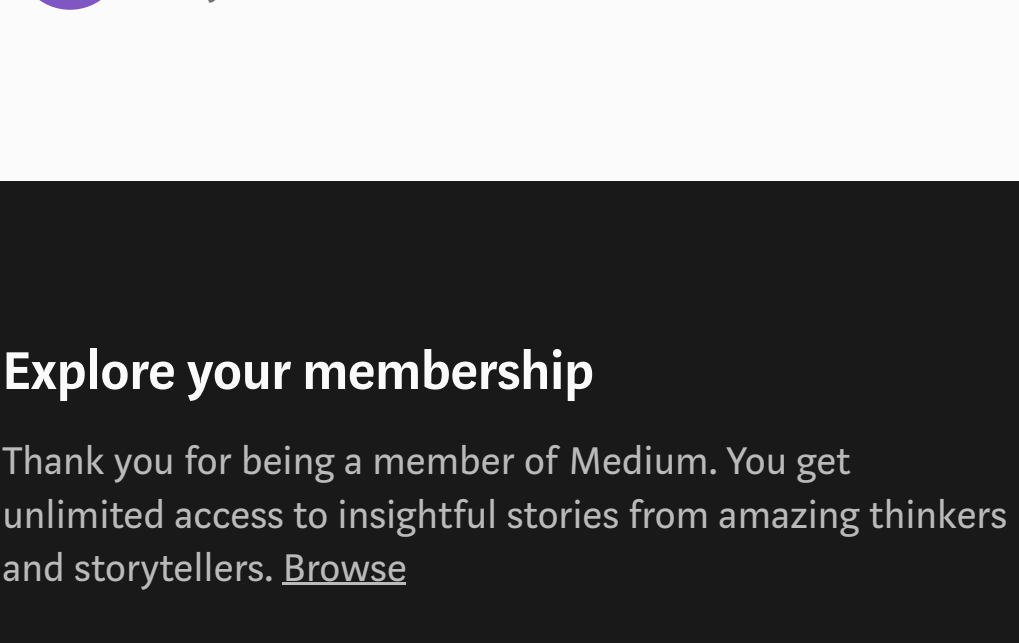


Clean Architecture: the solution to have a reusable, flexible and testable code

Rayane de Araújo

Apr 2, 2019 · 8 min read ★

Also tagged Microservices



Securing Microservices With Assymetric JWTs

Christopher Kuech in The Startup

May 17 · 8 min read ★

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. [Watch](#)

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. [Explore](#)

Explore your membership

Thank you for being a member of Medium. You get unlimited access to insightful stories from amazing thinkers and storytellers. [Browse](#)

Medium

AboutHelpLegal