The Saga is Antipattern

5 min read · Jun 19, 2023


Sergiy Yevtushenko Follow

Listen Share

The Saga pattern is often positioned as a better way to handle distributed transactions. I see no point in discussing Saga's advantages and disadvantages because Saga should not be used at all in the microservices -based systems:

> *If you need distributed transactions across a few microservices, most likely you incorrectly defined and separated domains.*

Below is a long explanation why.

Microservices As Distributed System

Any microservices-based system is a distributed system. To be precise — the simplest possible, basic version of it. Such a distributed system does not maintain any form of consensus. In other words, such a system:

- Lacks any built-in means to coordinate nodes
- Lacks any built-in means to get information about nodes
- If nodes need to communicate, such a communication must be a part of business logic

These properties result in an inability of a microservices-based system to perform certain tasks. For example, perform transactions. Or maintain consistency (even eventual). Or get information if all necessary nodes are up and running (i.e. if the system is available). As a result, if one node needs a piece of information from the other node, it should explicitly include a request to remote service as one of the business steps. It looks surprisingly similar to interactions between, for example,

browsers and web servers. Note, that such interaction means that each request is completely independent of each other. All transactions, if they are necessary, never cross request boundaries. Only with such a deep separation of domains (and data), it makes sense to require a microservice to maintain its own data, independently and separately. Or claim independent deployability or testability. Or include a request (which can fail) to another node as an explicit step in business logic (and handle failure at this level).

> *Since no transaction can cross the request boundary, the service must govern all data included in the transaction. This could be considered a validation criteria for the separation of domains. If any cross-service transaction is necessary, then split-up was done incorrectly.*

Domain Size Issue

As soon as we start doing separation of domains according to data governance, we may quickly realize that in the vast majority of cases, microservices look a lot like traditional monoliths and the domain they should handle is big. Or realize that traditional monoliths are, in fact, microservices. This happens because most organizations have only a very limited number of truly independent domains. Most often — one.

Unfortunately, the whole microservices hype ignores this fact, and we get "best practices", "design patterns", books, articles, etc. which are stretching the initial idea of loosely coupled, independently deployable services to areas where it does not fit. This results in a mind-blowing, devastating consequences:

- We build unreliable systems (see above about what kind of distributed systems microservices are) on top of reliable ones (cloud infrastructure)
- We get ugly, inherently broken design, where layers are mixed up, communication error handling/retrying/etc. and transaction handling happens at the business logic level
- We split data into parts and then try to collect them to process the request, introducing unpredictable and barely controllable tail latency
- We get the systems, which are unable to ensure data integrity and consistency
- We ought to perform end-to-end testing before deployment because there are no guarantees that a new version of service does not break the whole system. This

completely obviates any independent testability and deployability of the services

The list above is definitely incomplete. Incorrectly applied microservices can cause all kinds of harm. Especially when combined with "cloud native" "microservices" frameworks like Spring, which turn the whole system into a bunch of slowly moving monoliths.

All advantages and requirements, which are inherent properties and natural fit for the microservices, either disappear or get transformed into quite painful and expensive obstacles.

Unfortunately, all these considerations might result in much bigger domains than could be considered acceptable for traditional microservices design. That's fine and just means that we should not use microservices. What can we use then? Let's take a look.

Handling Big Domain

Since we're going to handle a big, but single domain, we need to use something capable to maintain consensus. There are at least three options:

- Modular monolith (also known as *modulith*)
- Event-driven architecture
- Cluster-based architecture

Modular Monolith

This option addresses most monolith pain points, in particular maintainability and concurrent development. Mostly, this is achieved by improving design via application of DDD and other techniques. The ability to access all data, perform regular transactions and lack of communication errors makes this approach an appealing choice for many use cases. In addition, this approach is much easier to fix/rework/refactor/update in atomic fashion during system evolution. Note that inner sub-services are not required to maintain their own data (but they can, if necessary). Shared data is often inherent and natural in this approach, as we're talking about a single domain. Since this is just a monolith, after all, there are no issues with maintaining the consensus nor deployment/monitoring/maintenance.

The main disadvantage of moduliths is the limited scalability. At some point, there might be a need to switch to another design. Fortunately, modularity significantly

simplifies the transition.

Event-Driven Architecture (EDA)

EDA is quite a popular choice and while often it is mentioned in the context of microservices, this is incorrect. By design, EDA leverages reliable data sharing infrastructure in the form of message brokers or pub-sub service, which explicitly conflicts with how microservices maintain their data. Overall, this architecture is described in detail in many sources, so I see no point to repeating them here.
Its main disadvantage is the similarity to microservices regarding reliance on infrastructure, complex deployment, monitoring, etc.

Cluster-based Architecture

This design is quite a rare animal. It has several advantages, but a few worth being noted separately:

- Simplicity of transition to this architecture from modulith
- No reliance on infrastructure. Often, the whole system is self-contained, could be deployed on premises, in the cloud or across several clouds
- Simplicity of deployment — there is only one deployable artifact
- Nearly linear scalability
- Real fault tolerance — failed node does not bring down the whole system nor results in error being returned from request. Requests are just somewhat slower processed
- Great resource utilization, fine-grained two level scaling

Conclusion

Microservices were thought of as a way to solve problems, but their blind application causes more harm than good. The main issue is that there are no clear criteria, where they are applicable. I have no illusions, my article will not solve this problem, but at least it provides some meaningful criteria to assess where microservices should not be used.

Microservices | Architecture | Design | Saga

Follow

Written by Sergiy Yevtushenko

630 followers · 111 following

Writing code for 35+ years and still enjoy it...

Responses (12)

Write a response

What are your thoughts?

Bernd Kursawe
Aug 21, 2023

Funny: the word 'saga' occurs 4 times: in the title and the first paragraph. So this article is about transactions, not the saga pattern. Should be more clear in the title.

About the content:

"If any cross-service transaction is necessary, then... more

80 1 reply Reply

Yossi Yaari
Aug 31, 2023

Thanks for the article.
For my team the need is to coordinate logical transactions across 3rd party systems. Not internal systems we could have joined ourselves.
Also, the reason the payterns of distribution have become popular in spite of the... more

7 Reply

Tak Yu Chan (Franky)
Aug 31, 2023

Nice article that mentions some smells of a distributed big ball of mud and the subset of consequences.

Some companies like a start up in a non-traditional domain, they often suffer from finding the business direction and strategy, which can have... more

7 Reply

See all responses

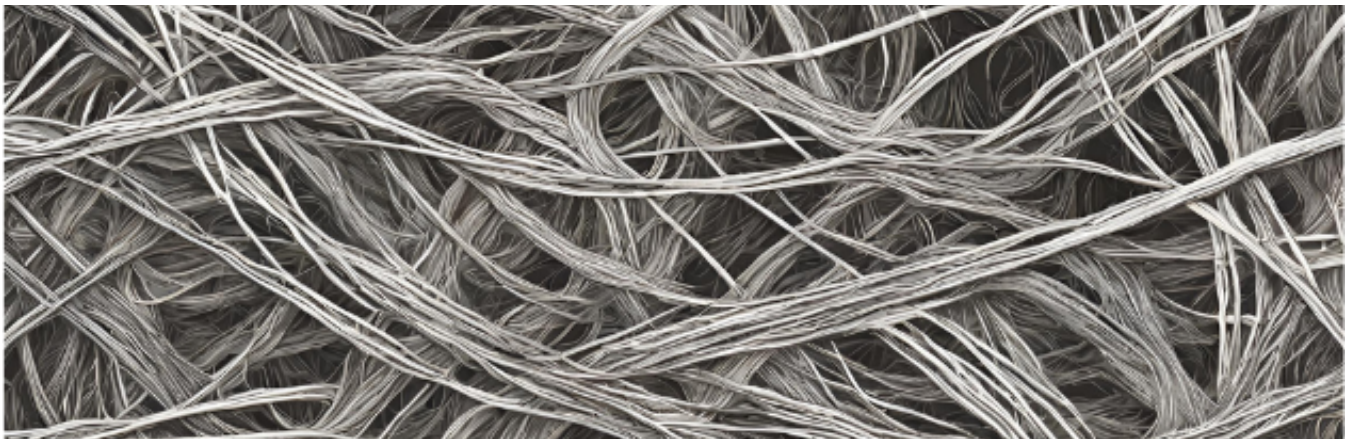
More from Sergiy Yevtushenko

Sergiy Yevtushenko

Asynchronous Processing in Java with Promises

Traditionally, asynchronous processing is considered complex and error-prone. There are several approaches to address this issue:

Apr 13 64 3



Sergiy Yevtushenko

Java Virtual Threads: The Good, The Bad and The Ugly

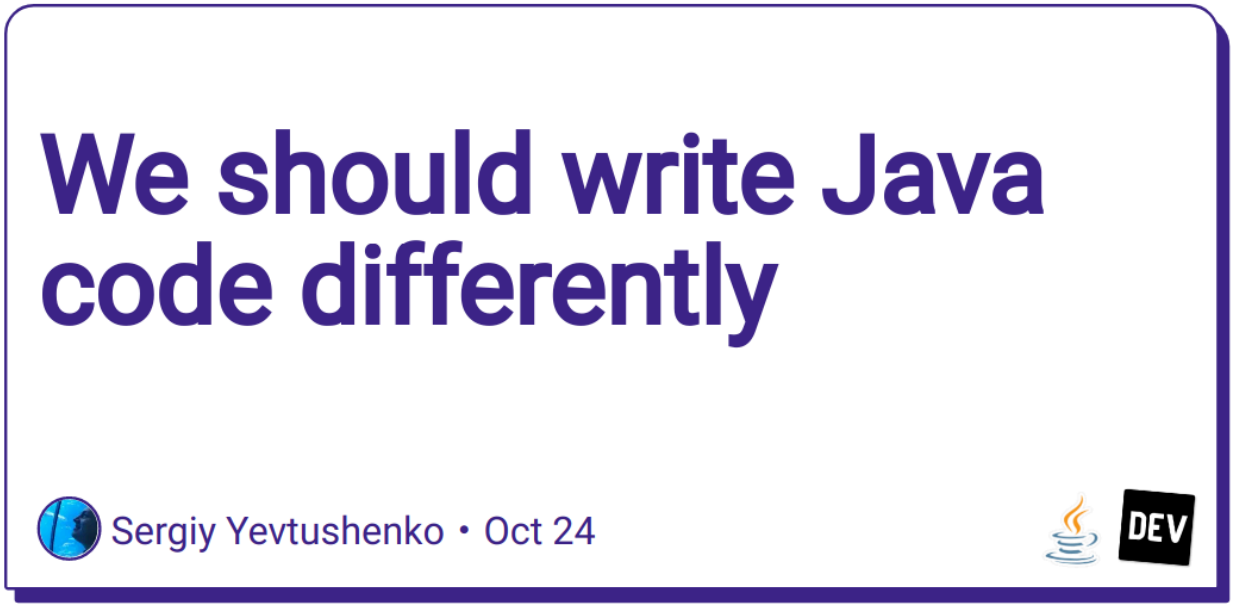
All that glitters is not gold

Oct 17, 2023 203 6

In CodeX by Sergiy Yevtushenko

Introduction To Pragmatic Functional Java.md

Oct 6, 2021 410 6



In CodeX by Sergiy Yevtushenko

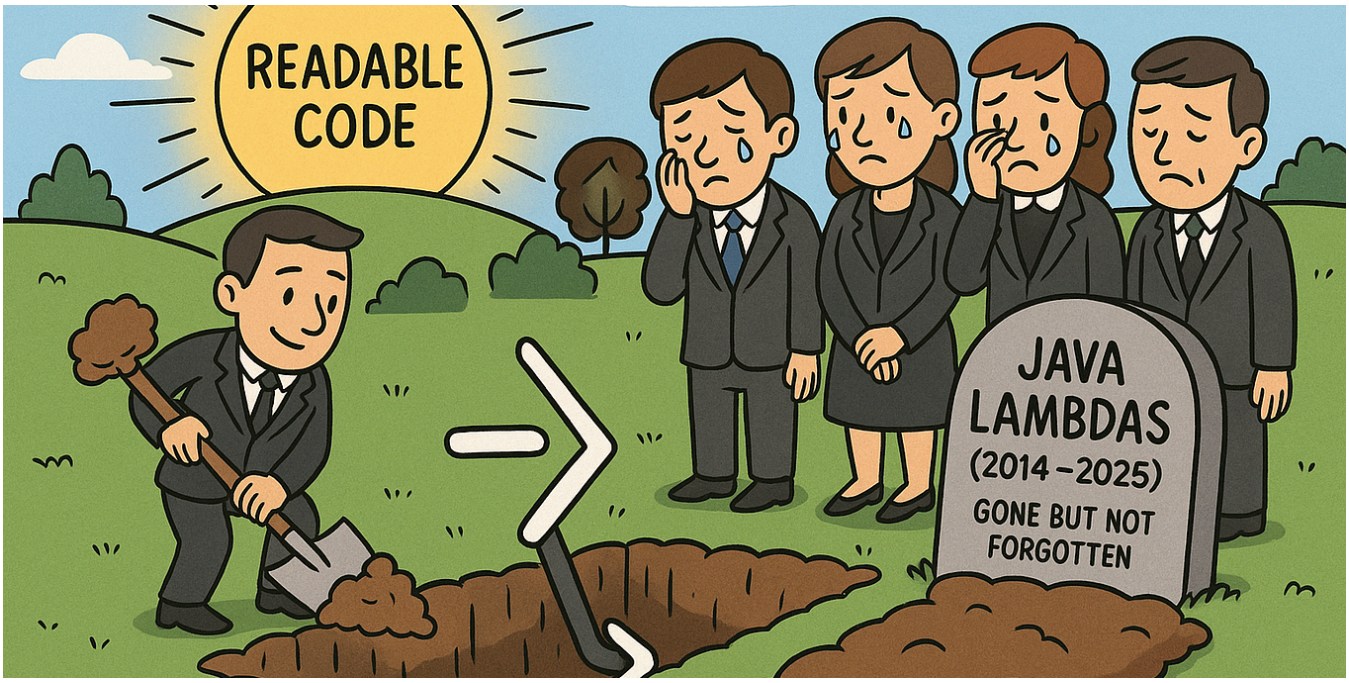
We should write Java code differently

For the last few years, I'm writing articles which describe a new, more functional way to write Java code. But the question of why we...

Oct 25, 2021 333 5

See all from Sergiy Yevtushenko

Recommended from Medium

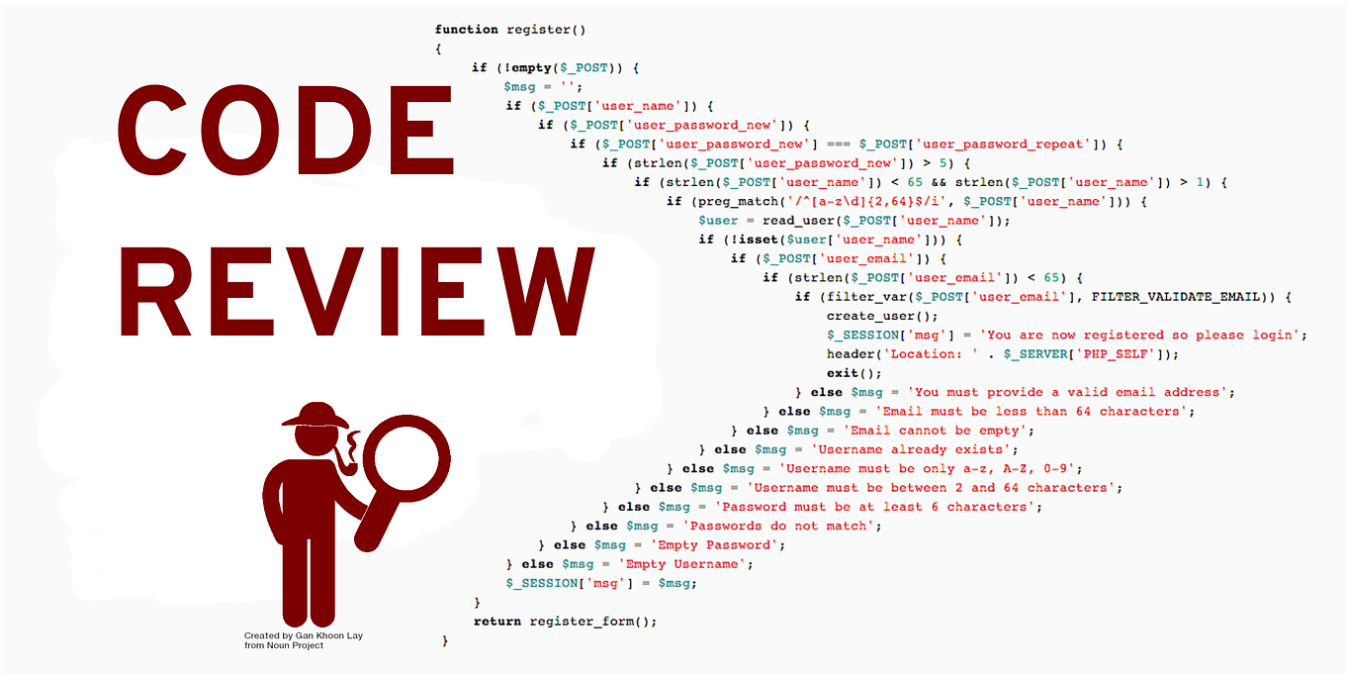


Kavya's Programming Path

🔥 RIP Java Lambdas (2014–2025): You Were Cool… Until You Weren't

Why I finally banned them from my codebase—with real-world bugs, benchmarks, and blood pressure spikes.

6d ago 673 43

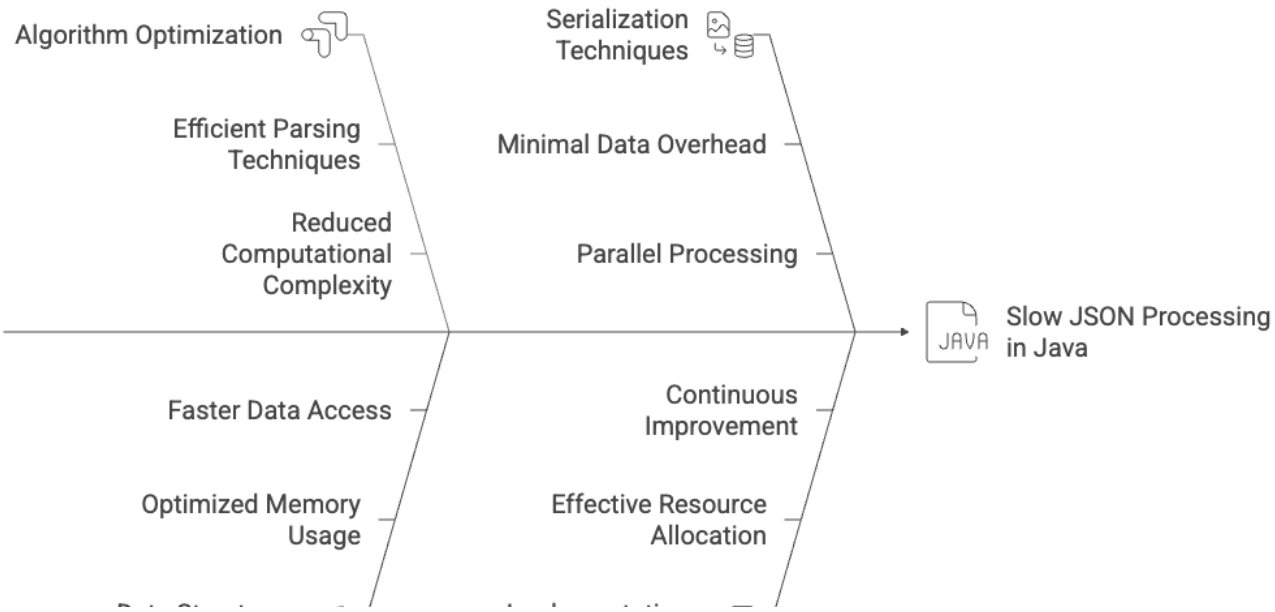


The Latency Gambler

Your Code Was Fine Until a Senior Dev Needed to Feel Smart

We've all been there. You submit a pull request that works perfectly, passes all tests, and solves the problem elegantly. Then a senior...

4d ago 54 2

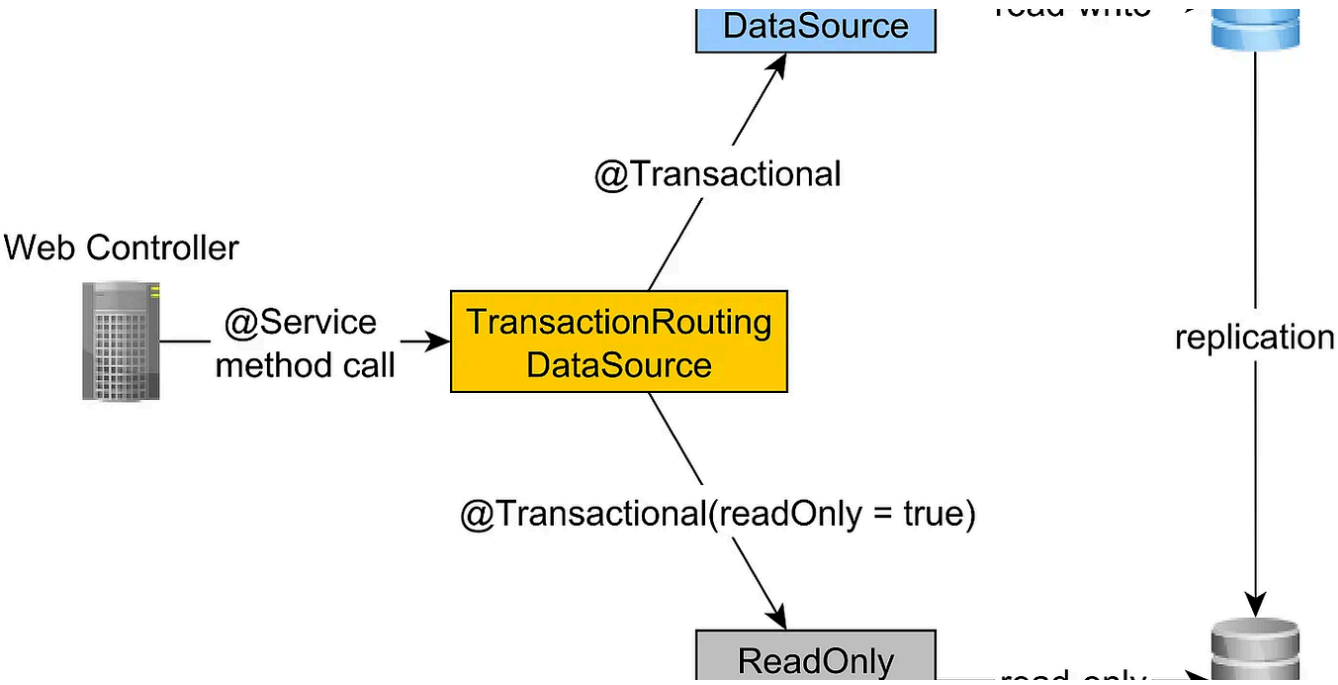


Sachin Kumar

30x Faster JSON in Java? Yes, and Here's How We Did It

At first, our service was fine.

Jul 26 141 4



Andrew Charles

You're Using @Transactional Wrong , and It's Costing You Performance

And why your Spring Boot app feels slower than a Monday morning

6d ago 46 4

h harsh shah

Java Microservices Architecture Guide: Spring Boot Best Practices for Production 2025

Complete implementation guide with REST APIs, Docker deployment, and real-world performance optimization strategies for enterprise Java...

Jul 26 142 1

In Stackademic by Pudari Madhavi

Java 21 Virtual Threads Almost Destroyed My App—Here's What No One Told Me

When I first heard about Java 21's virtual threads, I thought I had struck gold.

3d ago 75 3

See more recommendations