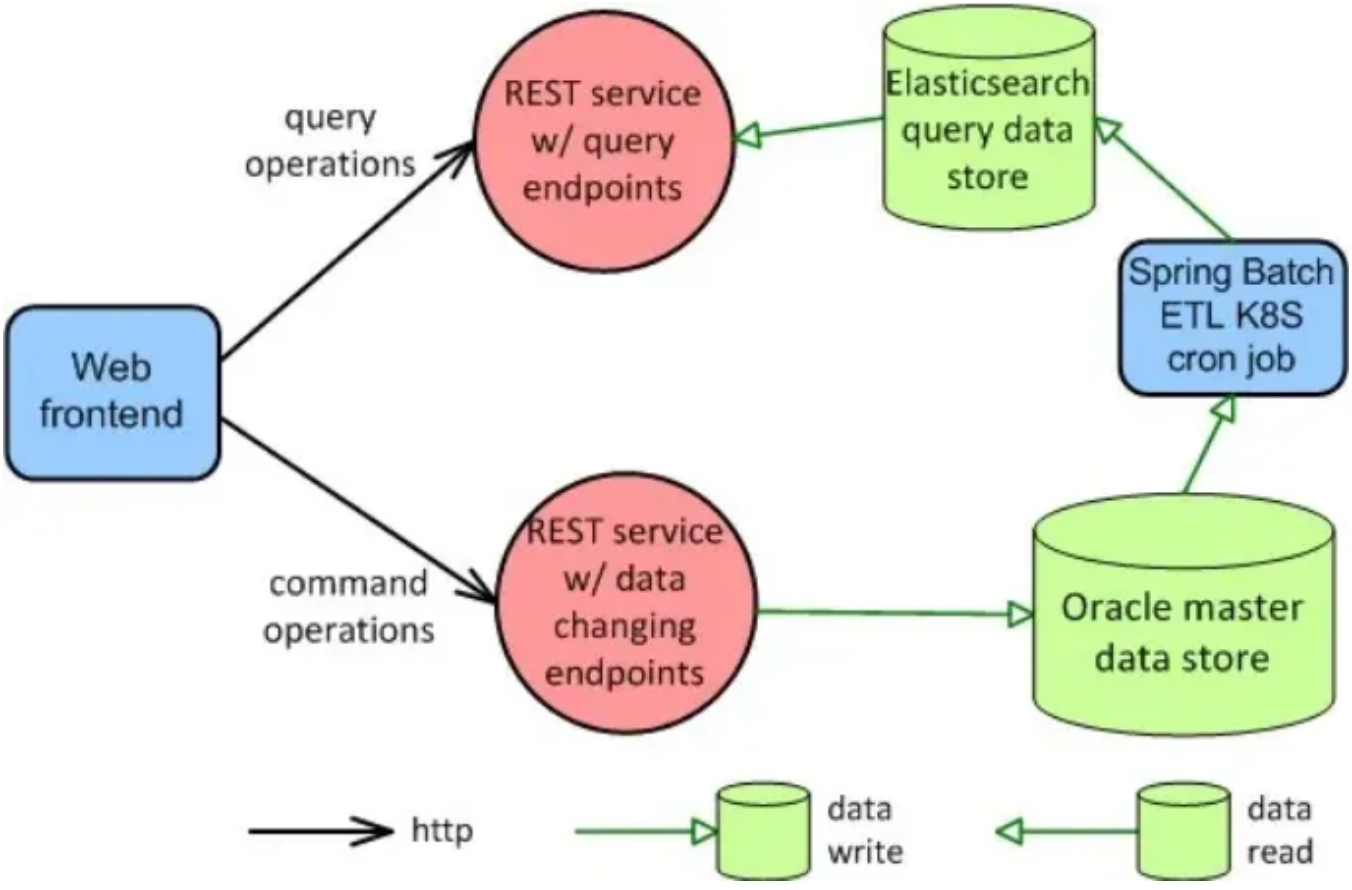
Best Practices for Experienced Developers

Designing Microservices for your organization? Follow these design principle to create a robust and scalable Microservices

Soma · Follow
Published in Javarevisited
9 min read · Feb 11



Hello folks, if you are a senior Java developer or a junior programmer working in Microservices, doesn't have much idea how to create a production grade Microservices the his article is for you. I have been doing Microservices

development in last a couple of years and its been a tough journey as I have to learn most of things hard way.

I made a big mistake when I didn't invested on my learning by joining nay Microservice training courses before jumping into development and due to that my learning was slow and I have to do a lot of Google searches even for simple stuff.

That's why it make more sense to join a couple of courses, read books and articles before you start taking work on a new technology or architecture. Even if you don't learn everything you will get ideas about lot of things which will save a lot f your time even if you have to search.

In past few articles I have been sharing my experience on Microservices like 50 Microservices Interview questions which I shared earlier as well my article about SAGA Design Pattern and Monolitihic vs Microservices architecture.

Difference between Microservices and Monolithic Architecture for Java Interviews

What are differences between Monolith and Microservices architecture? Pros and cons, and when to use each of them

medium.com

To continue that, In this article, I am going to share essential design principles you can follow while designing and developing your Microservices. These design principles are closely related to software redevelopment and it will help you to design robust, scalable and maintainable Microservices.

Bu the way, these design principles are different than *Microservices design pattern patterns* like CQRS, SAGA, or Database per Microservice, we will learn about them later, in this article we will focus on principles which applies to every Microservices.

Database Per Microservice Pattern in Java - Example Tutorial

Hello guys, in the last article, I share with your and 10 essential Microservice design patterns and principles best...

javarevisited.blogspot.com

Btw, if you are not a Medium member then I highly recommend you to join Medium and read great stories from great authors from real field. You can **join Medium here**

10 Essential Microservices Design and Development Principles for Programmers

Here is a list of 10 Microservices design principles you can keep in mind and follow while doing Microservice development. They will help you a lot in long run even after development and deployment.

Earlier, I have also shared ***25 Advanced Java questions***, *25 Spring Framework questions*, *20 SQL queries from Interviews*, *50 Microservices questions*, *60 Tree Data Structure Questions*, 15 System Design Questions, and *35 Core Java Questions* for Java developer interview and if you are preparing for interview you can see them.

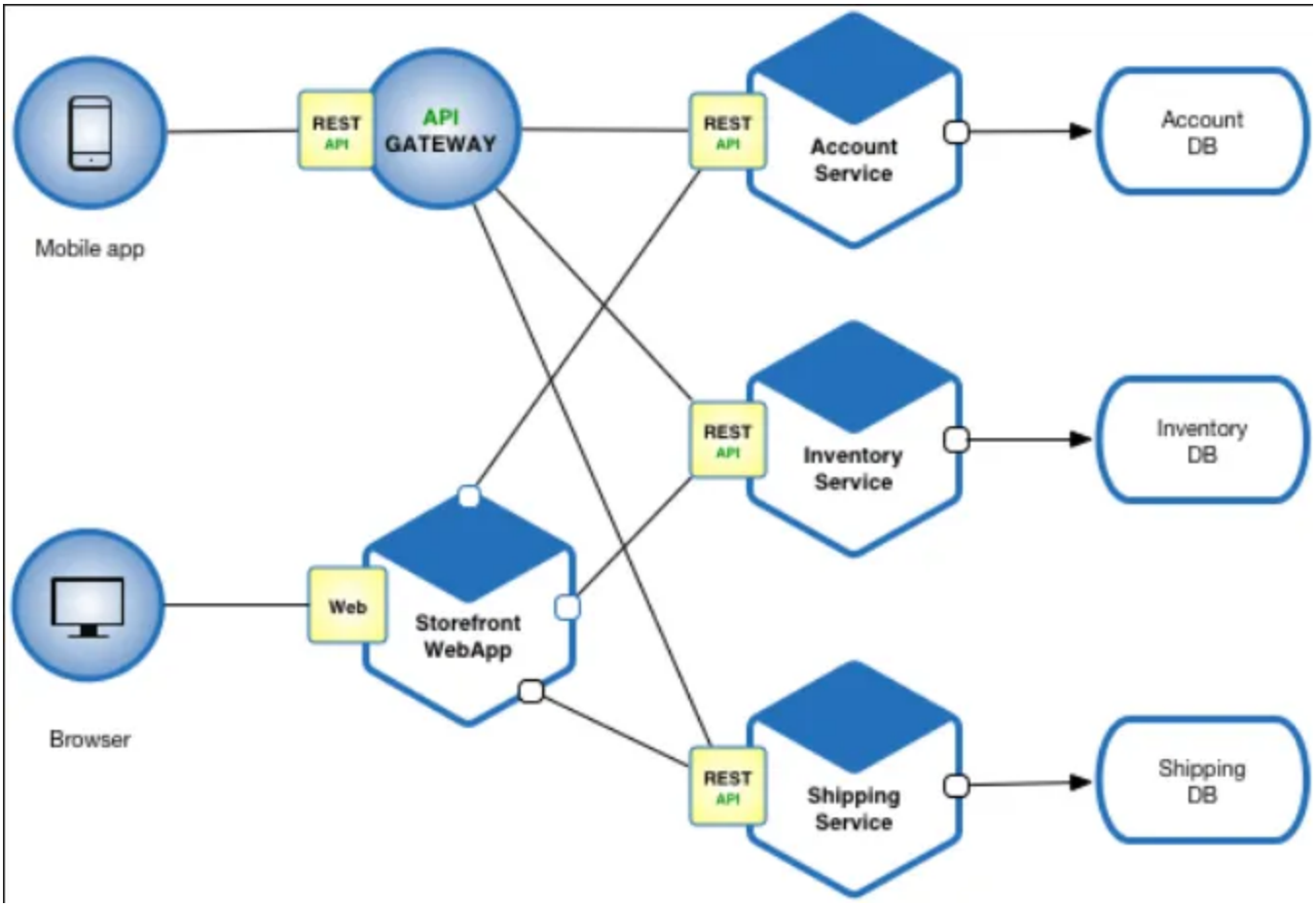
1. Single Responsibility Principle

This is not new to any Software developer, everyone of us are familiar with this as part of learning SOLID design principle but this also applies on Microservices.

As per SRP or Single Responsibility Principle, each Microservice should have a single, well-defined responsibility, and should only communicate with other microservices to accomplish tasks. .

For example, one microservice could handle user authentication, while another handles payment processing. But you shouldn't create a Microservice which is doing both user authentication and Payment handling, that would be violation of SRP.

You can see that in following diagram we have different services to handle different functionality like **Account Service**, **Inventory Services and Shipping Services.**



2. Decentralized Data Management

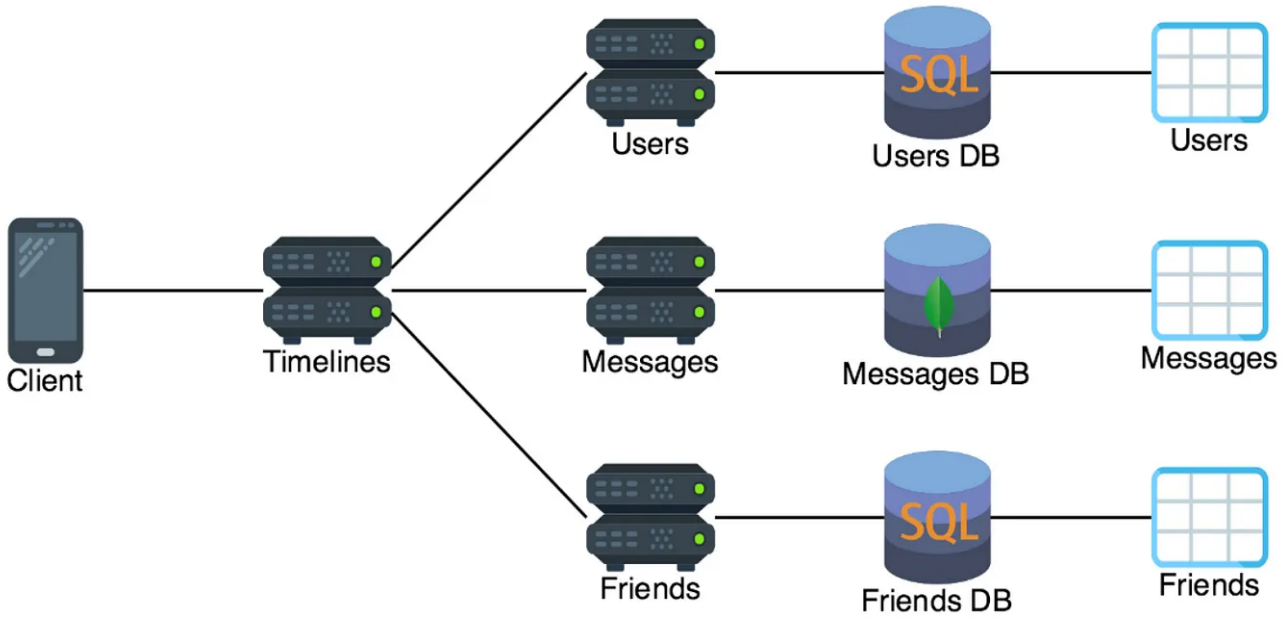
As per Decentralized Data Management principle, each Microservice should manage its own data, without relying on other Microservice, to ensure scalability and reliability. For example, each Microservice could have its own database that it uses to store data.

Sharing Database with other Microservices violate this principle and should be avoided as it will make it difficult to troubleshoot and can result in data inconsistency.

This design principle will help you better manage your database its also a basis of Database per Microservice design pattern, an essential Microservice Design Principle.

If you are thinking about how could then another Microservice get access to the same data as its very much possible that another service do need it? Well, you should always create APIs for that as we going to see in next Microservice design principle.

You can see this in following Microservice architecture we have different database for UserService, MessageService and FriendService.



3. API-Driven Design

This one is my favorite Microservice design principle and it helps a lot during actually designing Microservices. As per this principle, Microservices should be designed around APIs, with each service exposing a well-defined set of APIs for communication with other services.

For example, a Microservice could expose an API for retrieving customer information, which other microservices could use to access that information.

This principle also goes along with the previous design principle which advocate that Microservices should not share databases. You must create APIs for other services which need access to the data and this will then drive your own Microservice design.

4. Statelessness

As per this principle, Microservices should be stateless, meaning that they should not maintain any client-specific state between requests.

For example, if a microservice handles a user's shopping cart, it should not store any information about the user's cart between requests, but should retrieve the cart information from a database each time it processes a request.

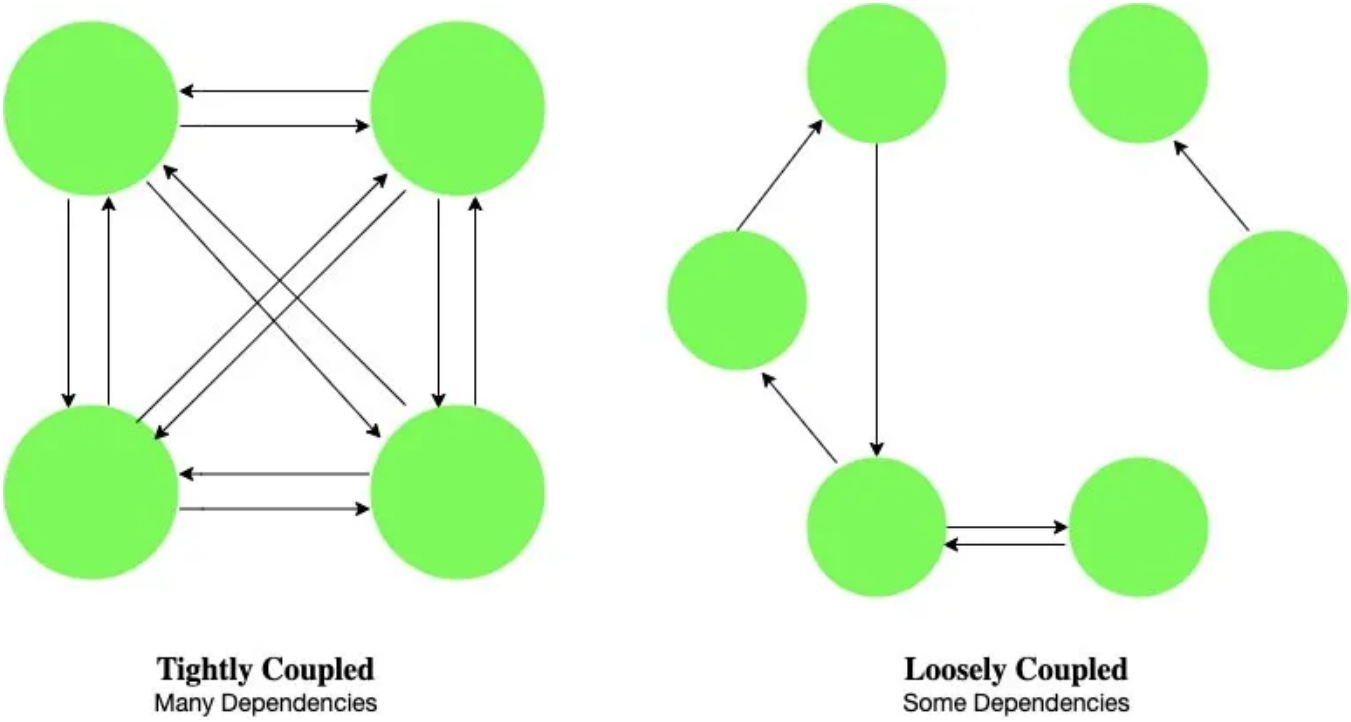
If I could give just one advice of my years of experience in designing Java services then I would say keep it stateless as long as possible. By not Managing state in your service, you will avoid a lot of problems which comes due to that and you can greatly benefit form things like Caching for performance improvement.

5. Loose Coupling

This is another Software deign principle which also applies to Microservices. In Object Oriented design, we aim to lose couple our classes, packages, and modules and we can apply the same rule with Microservices.

As per this principle, Microservices should be loosely coupled, meaning that **they should not have a tight dependency on each other, to ensure scalability and ease of deployment.**

For example, if one microservice is down, the others should still be able to function normally. Here is a diagram which illustrate tightly coupled and loosely coupled Microservices:



6. Smart Endpoints and Dumb Pipes

As per this principle, the data-processing logic should be located in the Microservices themselves, rather than in a centralized hub, to ensure scalability and reliability. If you are wondering what is endpoints and Pipes means here, endpoints are APIs or URLs while pipes are queues an database.

The pattern suggests that endpoints, such as user interfaces or APIs, should be designed to be smart, handling all presentation and business logic, *while pipes, such as queues or databases,* should be designed to be dumb, performing only simple data transfer and storage functions.

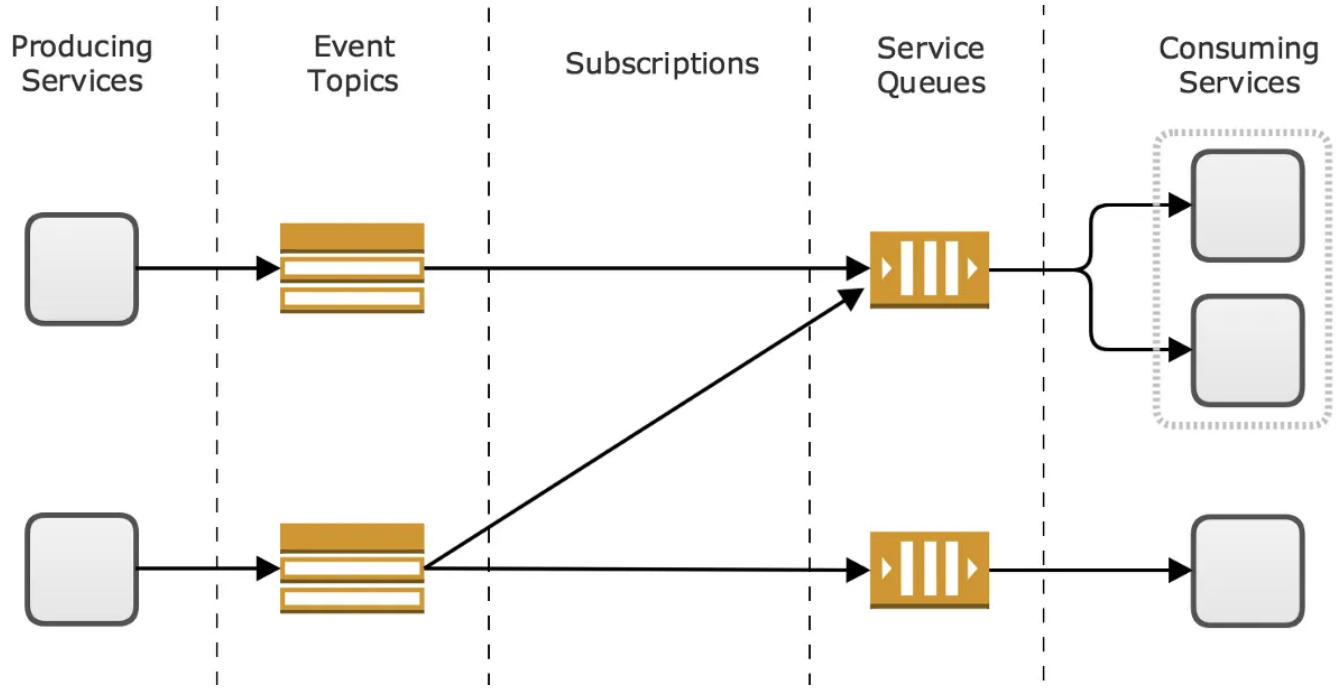
For example, a microservice could be responsible for processing customer orders, rather than having a centralized hub handle all order processing.

The key benefit of using this pattern is that it enables development teams to create applications with loosely coupled components that can be developed and deployed independently, allowing for better scalability and easier maintenance.

This pattern also helps to simplify the development process, as developers can focus on implementing specific features in their smart endpoints rather than worrying about the underlying communication and data storage systems.

However, one drawback of this pattern is that it can make debugging more difficult, as errors may occur in multiple components. Additionally, smart endpoints may require more resources and processing power, which can increase the cost of running the application.

Here is a nice diagram which illustrate this principle:



7. Auto-Scaling

When we think about Microservices, we think about Scaling and this is the principle which takes care of that. As per this best practice and principle, each Microservices should be designed to automatically scale up or down in response to changes in demand, to ensure that the application remains responsive and available.

For example, if the number of users accessing a microservice increases, that microservice could automatically spin up additional instances to handle the increased demand.

In real world, tools like Kubernetes can automatically scale up and scale down by creating new instances of your Microservices and destroying them once they are no longer required.

If your Microservices is not designed for automatic scaling then it wouldn't take full benefit of Cloud and tools like Kubernetes, so you must ensure that they are designed for scaling. If you are wondering how to achieve that, then stay tune, I will share more tips on how to ensure scalability of your Microservices in next article.

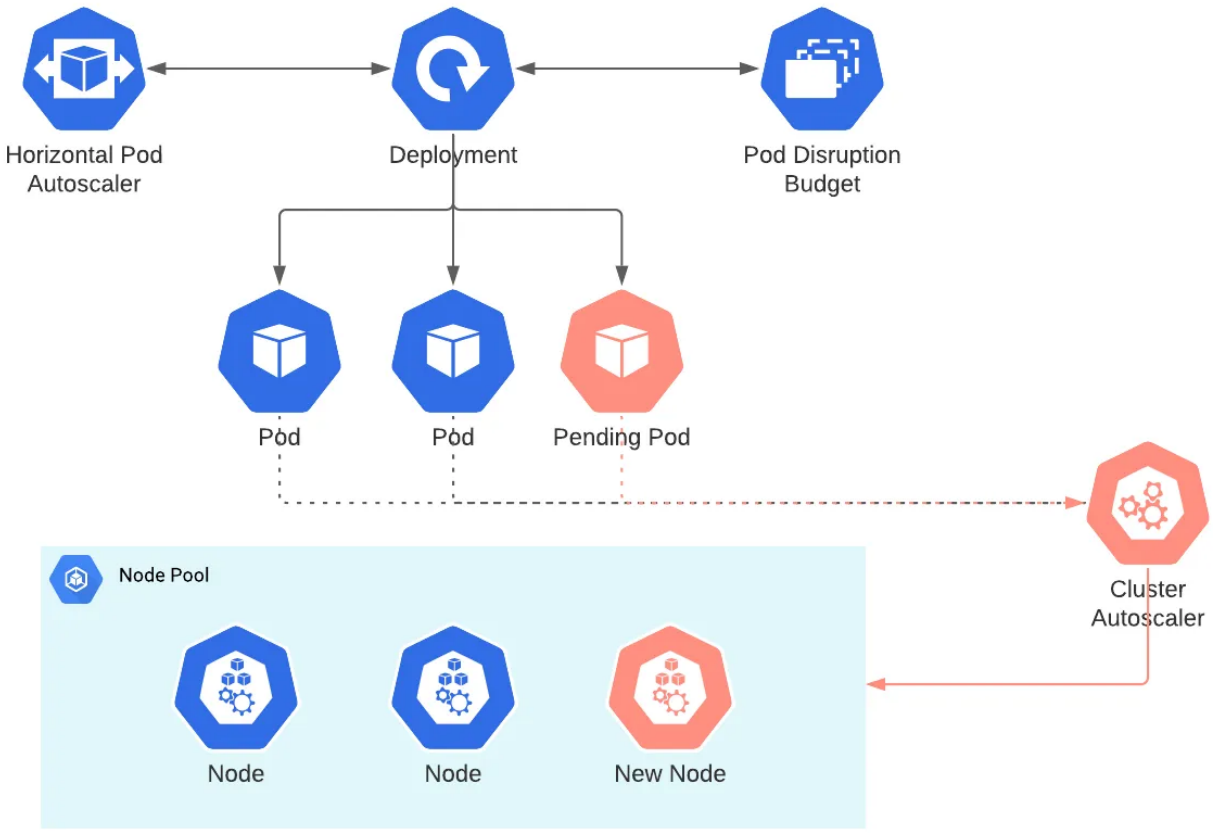


image_ https://timdepater.com

8. Monitoring and Logging

One of the major problem with working in a project which has hundreds of Microservices as debugging is really hard. It's hard to find the cause why a request failed because logs are scattered.

You may see user authentication errors thinking that something wrong with user credential but it because of timeout on service which does that, you will only come to know the true reason by looking at logs of multiple services.

As per this principle. Microservices should have robust monitoring and logging mechanisms in place to help diagnose issues and track performance.

For example, each microservice could log information about its performance and usage, which could be used to identify and diagnose issues. This will help you surely you in long run and day to day support work.

9. Continuously Deployment and Integration

As per this principle, Any Microservices should be continuously deployable, meaning that they should be updated frequently with small, incremental changes like bug fixes, small enhancements etc .

For example, a microservice could be updated to fix a bug or add a new feature, without affecting the rest of the application.

Continuous deployment is achieved through a combination of techniques such as automation of build and deployment processes, testing, and integration with other tools such as version control systems, issue tracking systems, and monitoring tools.

By automating the deployment process, teams can ensure that new changes are deployed quickly and consistently, with minimal human intervention.

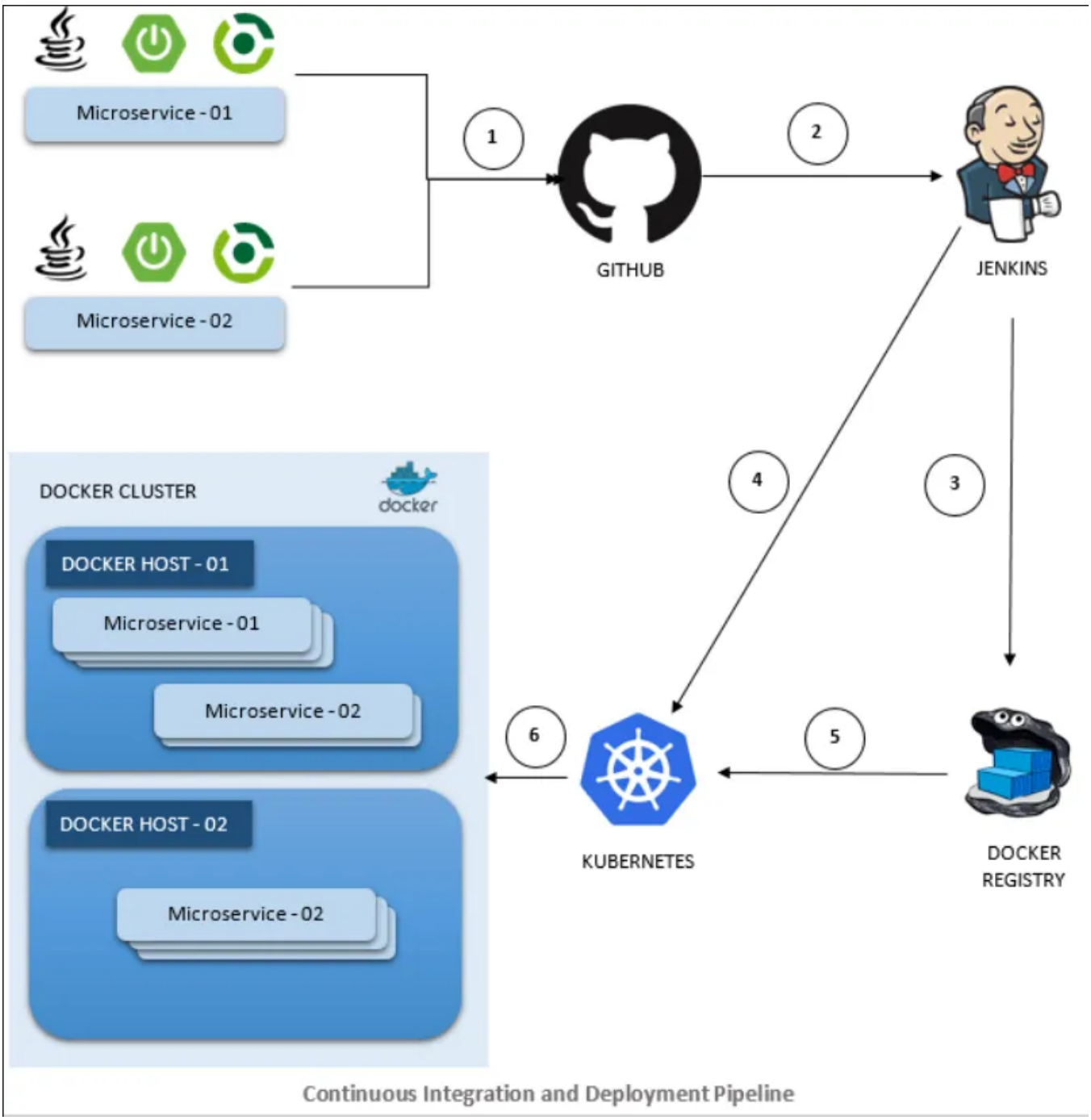
This practice also helps to reduce downtime and minimize the risk of errors, as new changes are thoroughly tested before they are deployed to production. Furthermore, it allows organizations to release new features and bug fixes more frequently, which can result in increased innovation and faster time to market.

10. Infrastructure Automation

This is something for DevOps but as a developer we should also be familiar with this. As per this principle, the deployment and management of microservices should be automated, to ensure consistency and efficiency.

For example, you can use tools like Docker or Kubernetes to automate deployment and scaling of Microservices. This is actually the standard practice now across the firms and it ensure that the deployment process is consistent and efficient.

You can create Jenkins pipelines for automatic deployment and you can also use tools like Terraform to create environment automatically. Here is an example pipelines for automatic deployment of Microservices.



That's all about the **essential Microservices design principles every Java developer should know.** By following these design principles, you can build scalable and reliable micro services applications that can meet the demands of their customers.

They will not only help you to build robust and scalable Microservices but also save time while debugging, monitoring, troubleshooting, and maintaining Microservices, which is a major challenge in the ever growing space of Microservices.

and , if you are not a Medium member then I highly recommend you to join Medium and read great stories from great authors from real field. You can **join Medium here**

Java Microservices Programming Software Development Software Engineering

Follow

Written by Soma

4.2K Followers · Editor for Javarevisited

Java and React developer, Join Medium (my favorite Java subscription) using my link 👉 https://medium.com/@somasharma_81597/membership

More from Soma and Javarevisited

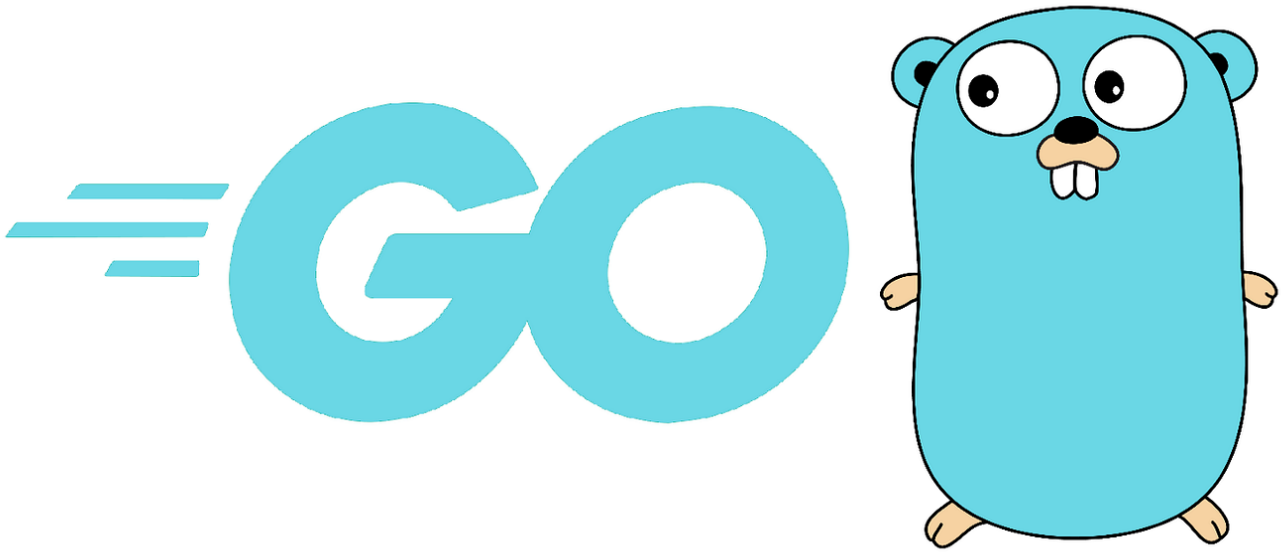
Soma in Javarevisited

Should Programmers Have a YouTube Channel?

Does creating a YouTube Chanel enhance a profile of Programmer or help in career?

· 5 min read · Aug 15

86 1



javinpaul in Javarevisited

10 Projects You Can Build to Learn Golang in 2023

My favorite Golang Projects with courses for beginners and experienced developers

11 min read · Aug 7

98 1

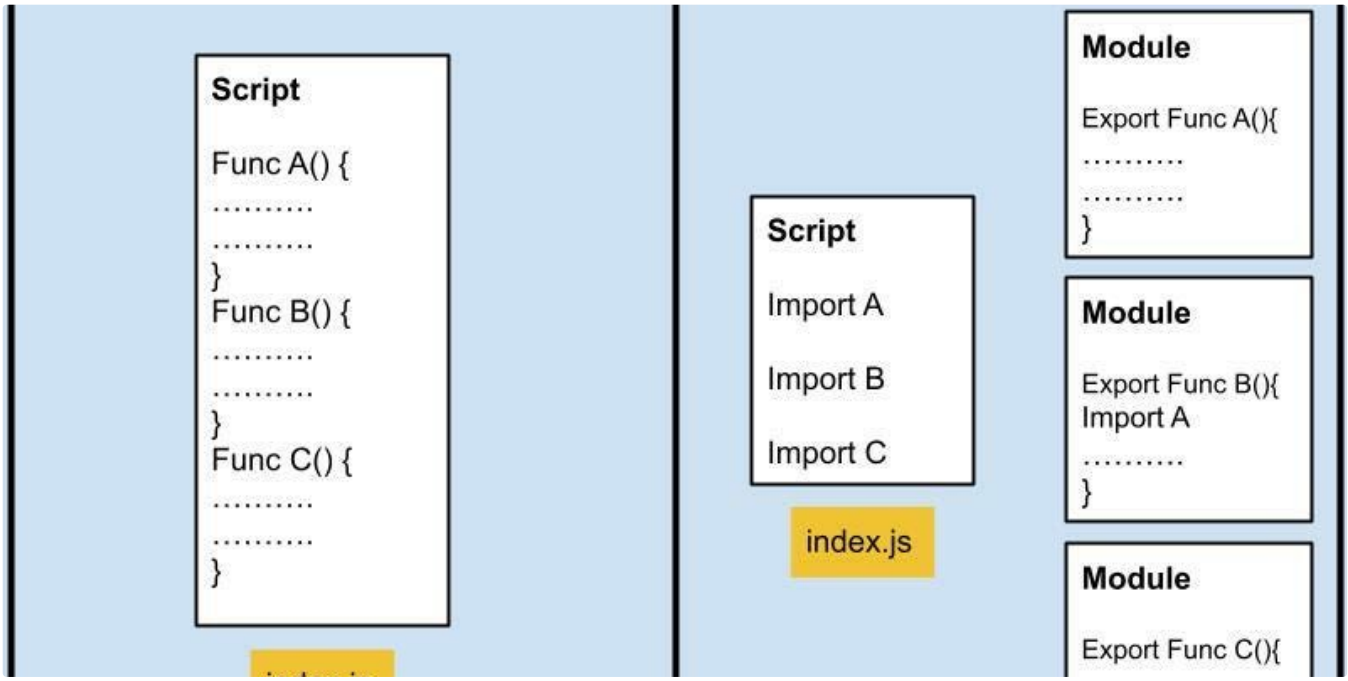
Sivaram Rasathurai in Javarevisited

10 Java Stream Tips—Must Read

Java Stream API is like a Swiss Army knife for Java developers—it's versatile, compact, and can handle a wide variety of tasks with ease.

4 min read · Mar 1

294



Soma in Javarevisited

10 JavaScript Features That Make Web Development Easier

My favorite features of JavaScript programming language

· 6 min read · Aug 18

126 1

See all from Soma

See all from Javarevisited

Recommended from Medium

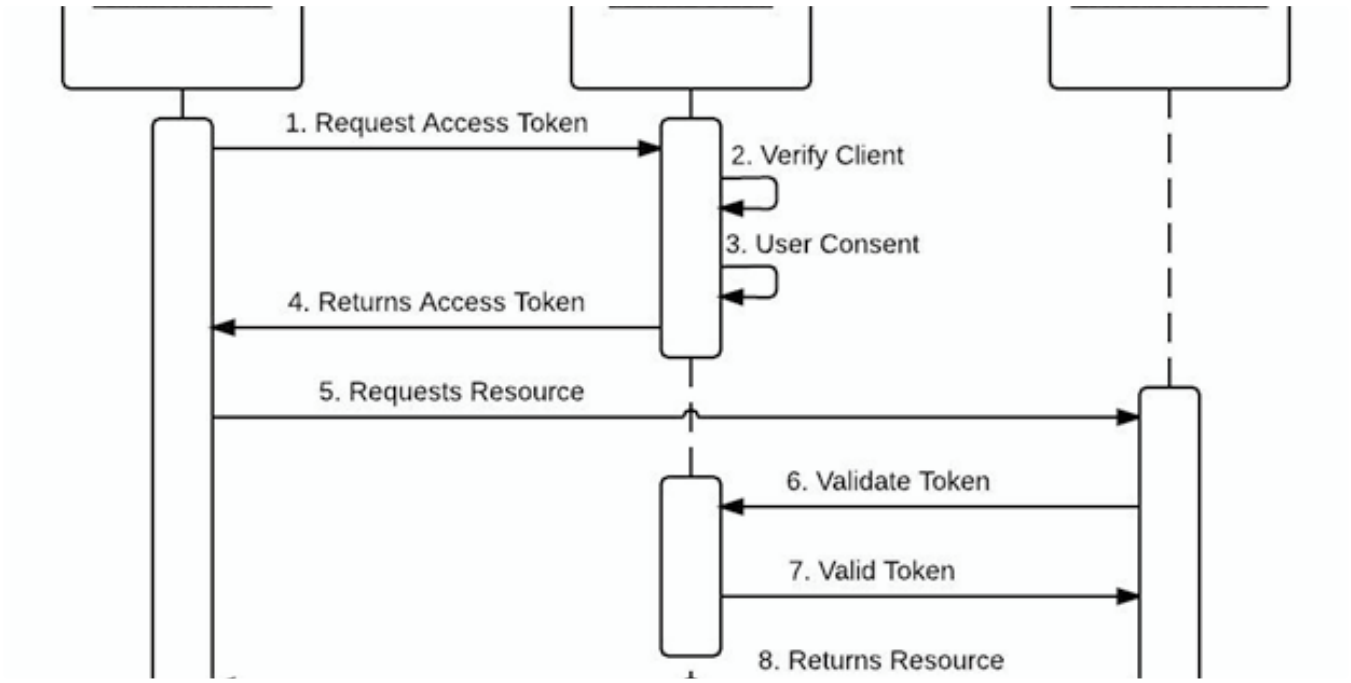
Shubhadeep Chattopadhyay

Best practices for Clean Code

Clean code is a set of programming practices that emphasize the readability, maintainability, and simplicity of code. Writing clean code is...

7 min read · Mar 3

1.1K 29



Somal Chakraborty in Dev Genius

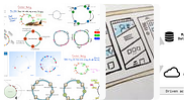
Securing a Microservice with OAuth 2.0 (Part 2 : How Token Based Security Works Internally)

Table of content

5 min read · May 1

37 1

Lists

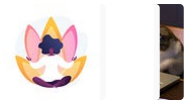


General Coding Knowledge

20 stories · 257 saves

It's never too late or early to start something

15 stories · 93 saves



Stories to Help You Grow as a Software Developer

19 stories · 309 saves

Coding & Development

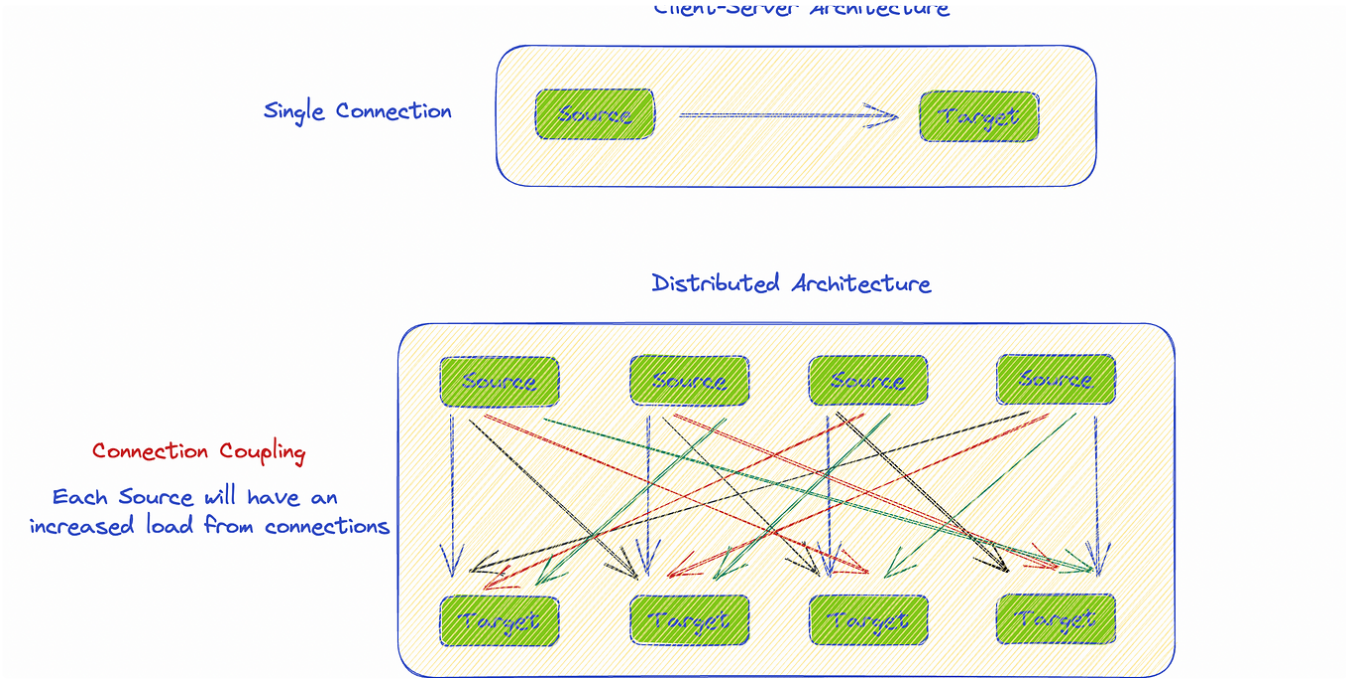
11 stories · 129 saves

Ionut Anghel

REST Endpoint Best Practices Every Developer Should Know

5 min read · Mar 18

79 1



Akshat Vashishtha

Kafka with Spring Boot using docker-compose

Prior understanding to Kafka, we should understand the problem Kafka try to solve. In simple Client-Server architecture source machine...

4 min read · May 9

39

Eray Araz

Spring Webhook

Introduction

3 min read · Apr 2

109 1

Bubu Tripathy

Best Practices for designing REST APIs

In this post, we'll discuss some of the best practices for designing RESTful APIs, including real-world examples.

15 min read · Mar 6

413 12