Open in app ↗

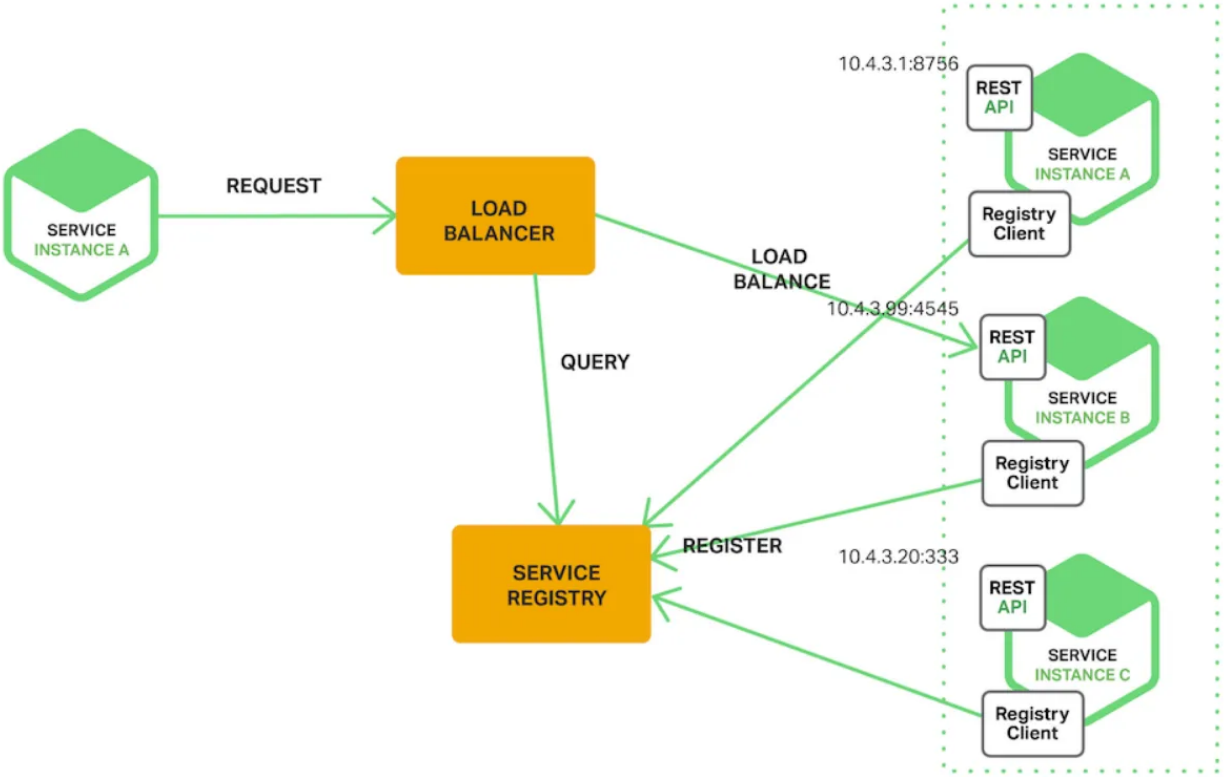
Member-only story

Top 10 Microservice Architecture Design Patterns Every Developer Should Learn

10 essential Microservices design pattern to create robust, scalable, and reliable Microservices

Soma · Follow
Published in Javarevisited
14 min read · Feb 21

Listen Share More



Hello folks, while industry trend is to split your monolithic application to microservices to segregate data, code, and interface its not an easy task to do, especially if you don't have any experienced in Microservice development and not

familiar with best practices and essential Microservices design patterns and principles.

Liked *Object Oriented Design Patterns*, Microservice Patterns are also tried and tested solution of common problems people have encountered while developing, deploying and scaling their Microservices.

For example, **SAGA pattern** solves the problem of distributed transaction failures and API gateway makes client side code easier and also act as a front controller and load balancer for many of your Microservices, and thus making them more maintainable.

In the past, I have shared **50 Microservice Interview Questions** for beginners and experienced developers and *10 essential Microservice design principles* and in this article, I am going to give you brief overview of essential Microservice patterns and when to use them with simple examples and scenarios.

This is the minimum, you need to know and remember as a Microservice developer, if you know the basics and have bit of idea, you can always go back and search and learn them in depth before you actually use them in your project.

Here are some popular Microservice design patterns that a programmer should know:

1. Service Registry
2. Circuit Breaker
3. API Gateway
4. Event-Driven Architecture
5. Database per Service
6. Command Query Responsibility Segregation (CQRS)
7. Externalized Configuration
8. Saga Pattern
9. Bulkhead Pattern
10. Backends for Frontends (BFF)

Now, let's see them in more detail and learn how and when to use them but before that, I suggest you to **join Medium** to read your favorite stories without interruption and learn from great authors and developers like Google Engineers and Tech experts from FAANG companies. You can **join Medium here**

Join Medium with my referral link - Soma

Read this and every story from thousands of great writers on Medium). Read more from great writers on Medium like...

medium.com

10 Essential Microservices Design Patterns for Experienced Developers

Here is a list of popular Microservices design patterns which every developer should know and learn if they are working in Microservices or breaking their monolithic application into Microservices to separate code, data and interfaces:

1. Service Registry Pattern in Microservices

The Service Registry Pattern provides a central repository to discover Microservices by name. It is a microservice architecture pattern that enables services to discover other Microservices and communicate with each other.

In this pattern**, a central service registry or directory is used to keep a record of the available services and their locations.** Microservices can register themselves with the registry, and other Microservices can look up the registry to find the location of the required services.

For example, suppose we have a **large e-commerce website that consists of many microservices**, such as an order service, a payment service, a shipping service, and a customer service. Each of these services has its own REST API that other services can use to communicate with it.

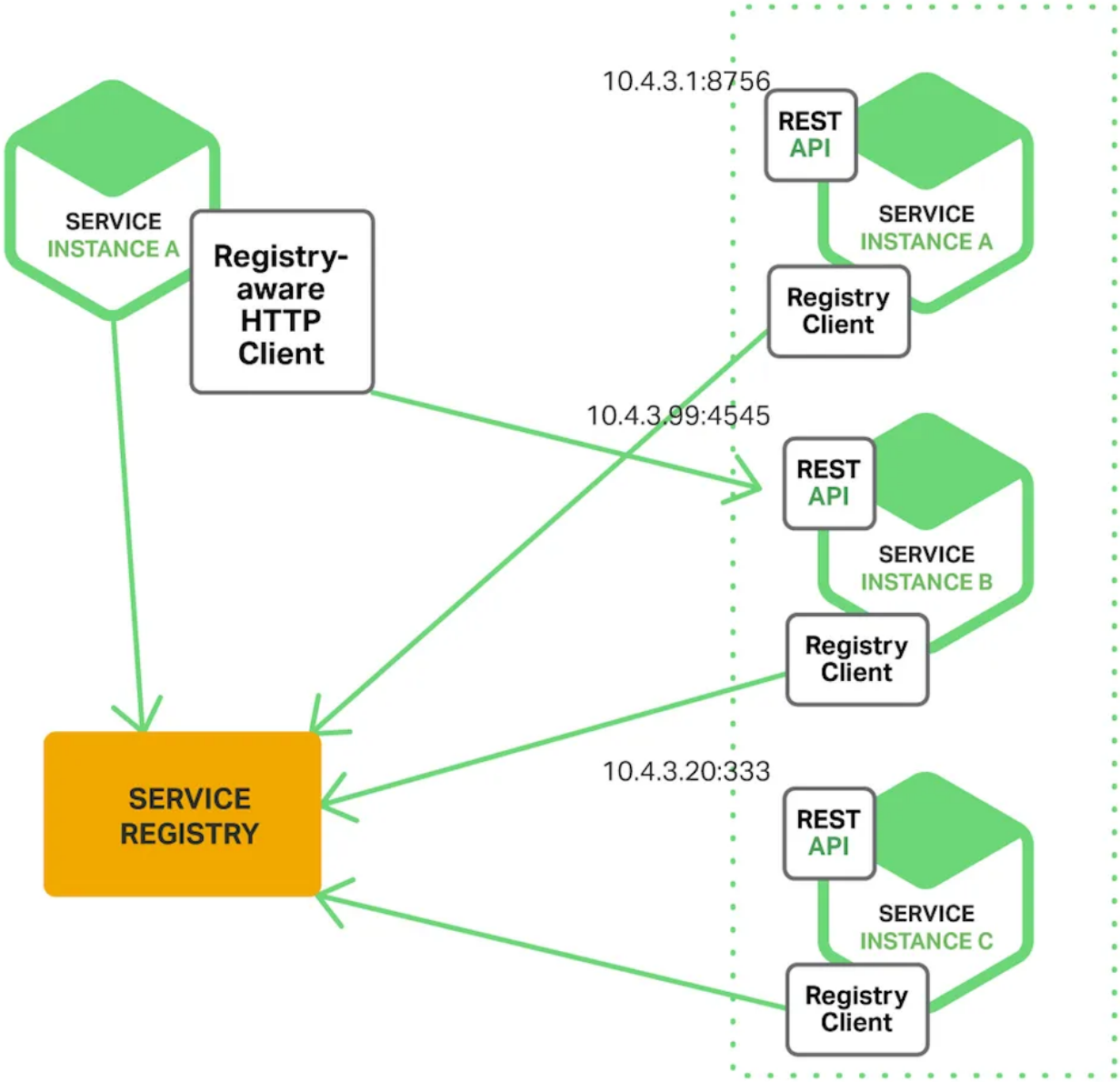
To make it easier for these services to discover each other, we can use the **Service Registry Pattern**. We can set up a service registry, such as Consul or Eureka (Spring cloud provides this), that maintains a list of all the available services and their endpoints.

When a service starts up, it can register itself with the registry by providing its name and endpoint.

For example, the order service might register itself as “order-service” with an endpoint of “http://order-service:8080". Other services that need to communicate with the order service can then look up its endpoint in the registry using its name.

For instance*, the payment service might look up the “order-service” endpoint in the registry to send payment information to the order service.* Similarly, the shipping service might look up the “order-service” endpoint in the registry to get shipping information for an order.

This way, each service can be developed and deployed independently, without hard-coding the endpoints of other services in its code. The Service Registry Pattern enables the services to locate each other dynamically, making the system more flexible and resilient to changes.



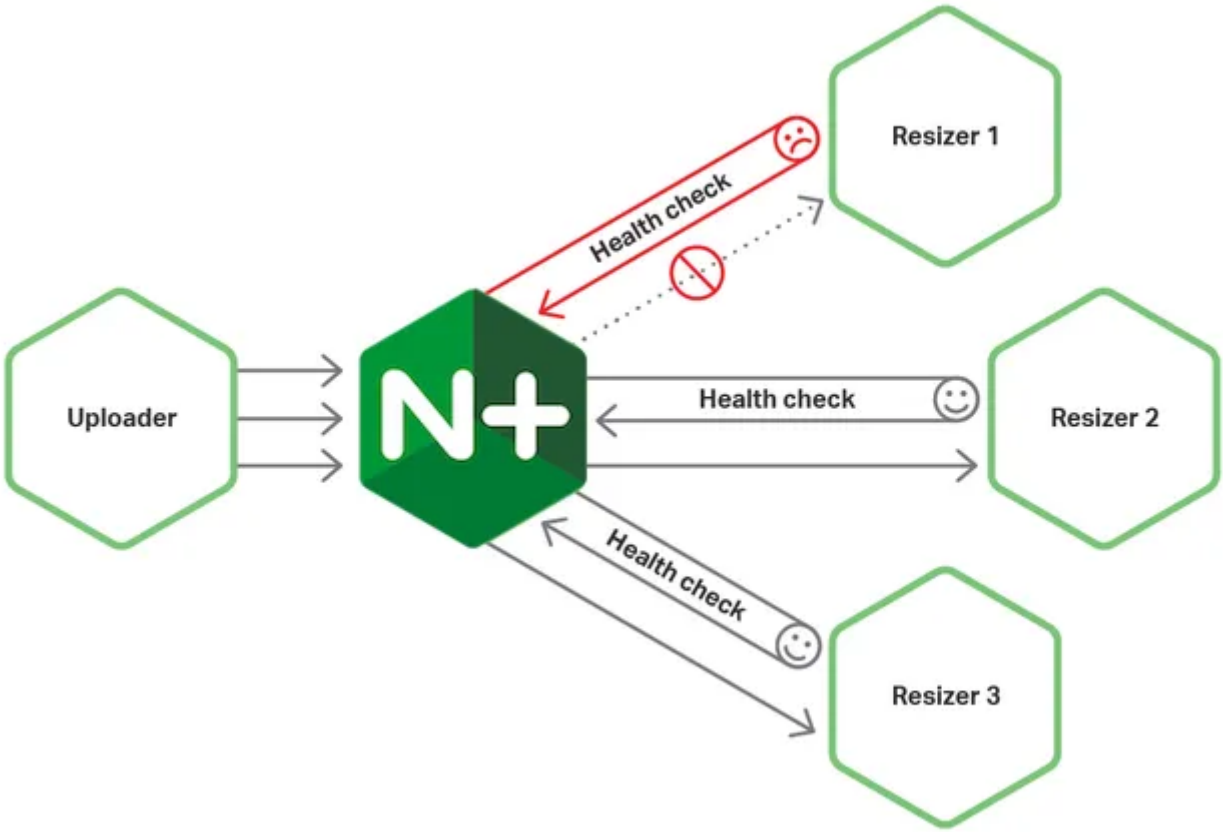
2. Circuit Breaker Pattern

As the name suggest, the Circuit Breaker pattern prevents cascading failure by *breaking circuit and enables applications to continue functioning when one or more services fail*. It is used to handle faults that may occur in a microservice architecture.

In this pattern, a circuit breaker acts as a safety net between the client and the service, protecting the client from failures in the service. The circuit breaker monitors the status of the service and, if it detects that the service is failing, it can open the circuit and prevent further requests from being sent to the service until the service has recovered.

For example, suppose **a microservice application is using an external service that is unreliable, and the application needs to continue functioning even if the external service fails.**

In this scenario, the *Circuit Breaker pattern can be used to detect when the external service is unavailable* and switch to an alternative service or a fallback service until the external service becomes available again.



In a Microservice architecture, the Circuit Breaker pattern can be implemented using tools such as Netflix's Hystrix or **Spring Cloud Circuit Breaker,** which provide a way to manage the circuit breaker's behavior and allow the application to react to service failures in a controlled manner.

3. API Gateway Pattern

The API Gateway Pattern is another common design pattern used in microservices architecture that **involves an API gateway,** which acts as an entry point for all incoming API requests. It provides a single point of entry for all the microservices and acts as a proxy between the clients and the microservices, routing requests to the appropriate service.

The main purpose of an API gateway is to **decouple the clients from the microservices,** abstracting the complexity of the system behind a simplified and consistent API. This also means that you don't need to find and remember address of 100+ Microservice REST APIs.

It also provides an additional layer of security and governance, allowing organizations to control and manage access to their services, monitor the performance of the system, and enforce policies across all the services.

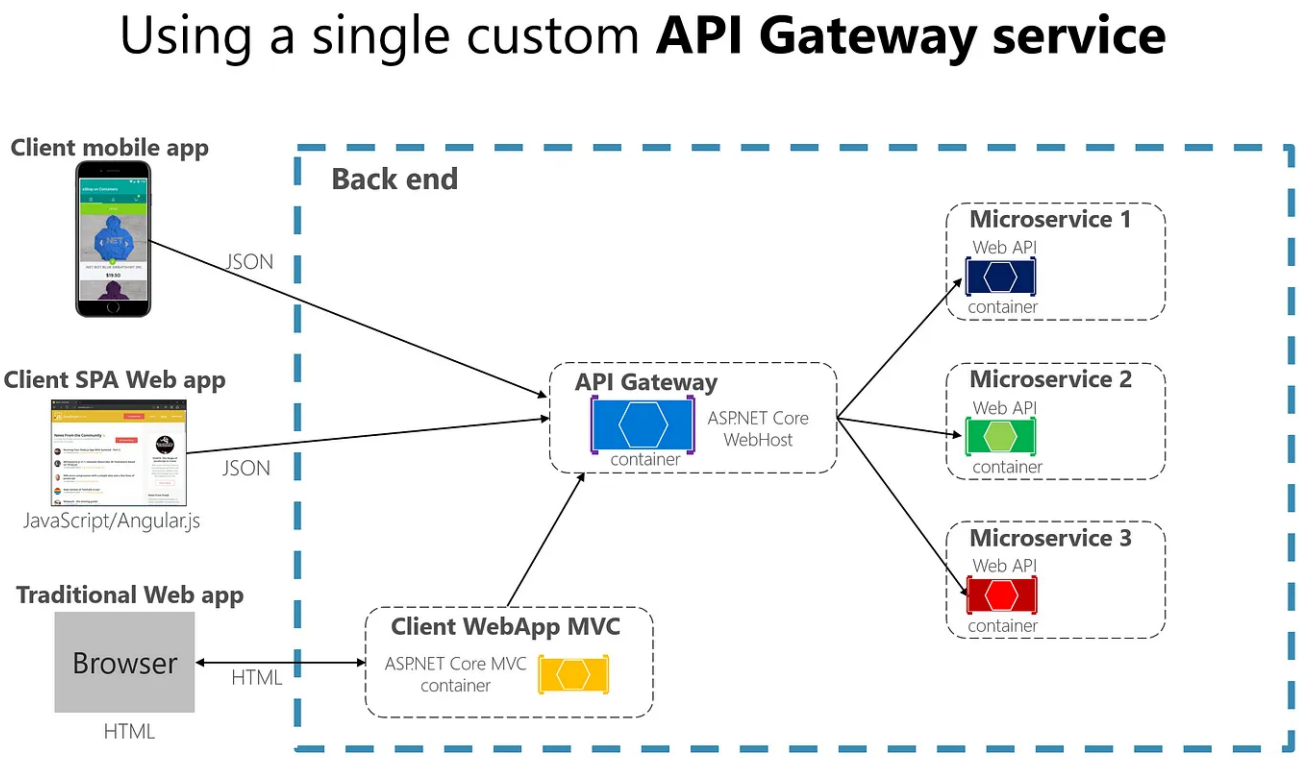
Here is an example of *how the API Gateway pattern works in a simple e-commerce system:*

Suppose an e-commerce system has multiple microservices to handle different functions such as order management, product catalog, and user authentication. Each microservice has its own API endpoint for handling requests. However, the client, which could be a web or mobile application, needs to access all these microservices through a single entry point.

This is where an API Gateway comes into play. The API Gateway acts as a reverse proxy that receives all incoming requests from the clients. It then routes each request to the appropriate microservice based on the endpoint requested.

For example, an API Gateway might route requests to `/orders` endpoint to the order management microservice, and requests to `/products` endpoint to the product

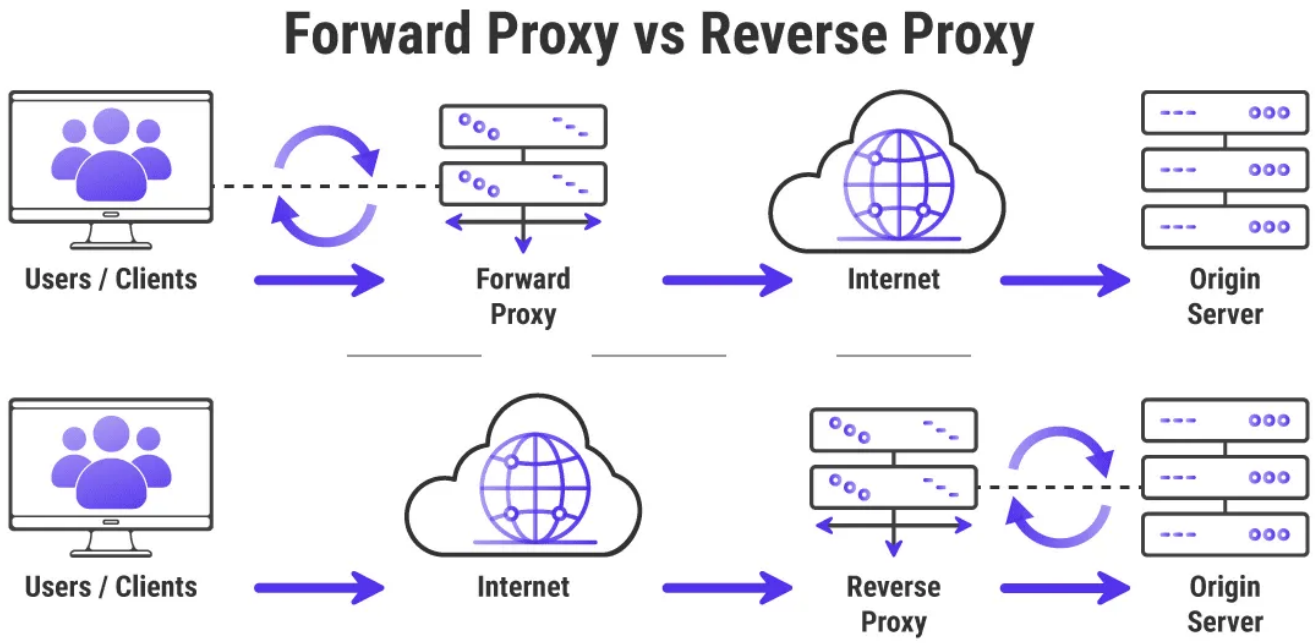
catalog microservice.



The API Gateway can also perform additional functions such as request and response transformation, rate limiting, authentication and authorization, and caching.

It can also provide a unified API that hides the internal details of the Microservices and presents a simpler and consistent interface to the clients.

Overall, the **API Gateway pattern provides a scalable, flexible, and secure way to manage microservices in a complex system,** making it easier to develop, deploy, and maintain Microservices-based applications.



4. Saga Pattern

The Saga pattern provides a way to manage transactions that involve multiple microservices. It is used to ensure that a series of transactions across multiple services are completed successfully, and if not, to roll back or undo all changes that have been made up to that point.

The Saga pattern consists of a **sequence of local transactions, each of which updates the state of a single service**, and a corresponding set of compensating transactions that are used to undo the effects of the original transactions in case of a failure.

Here's an example of how the Saga pattern is used in A Microservice based e-commerce application:

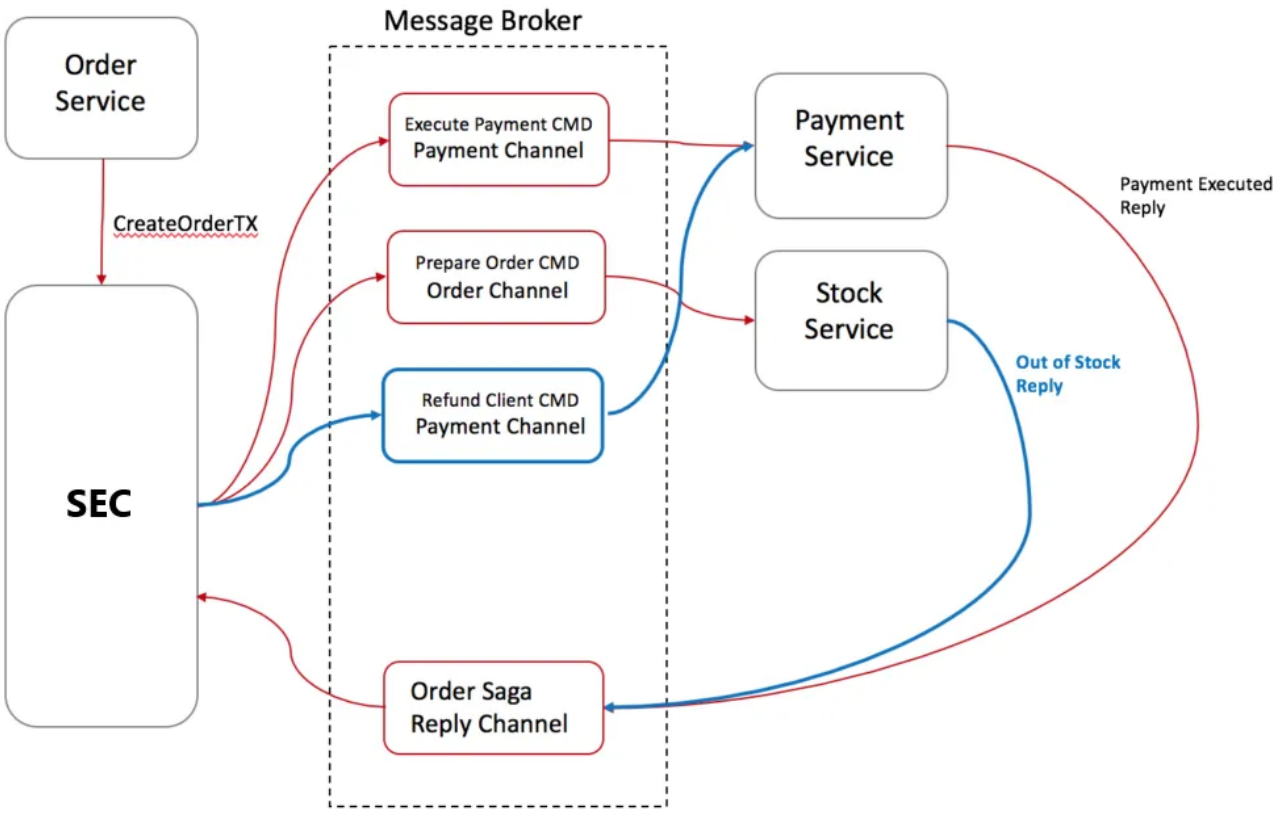
Suppose **you have two microservices, one responsible for processing orders and another responsible for shipping orders.**

When a new order is placed, the order *processing service is responsible for validating the order* and ensuring that the items are in stock, while the *shipping service is responsible for packaging the order a*nd sending it to the customer.

If the order processing service determines that the order is valid and all items are in stock, it sends a message to the shipping service to initiate the shipping process. At this point, the Saga pattern comes into play.

The shipping service will create a new transaction to package and ship the order, and if the transaction is successful, it will mark the order as shipped.

If, on the other hand, **the transaction fails (perhaps due to a problem with the shipping provider), the shipping service will initiate a compensating transaction to undo the effects of the original transaction,** such as canceling the shipment and restocking the items.



Meanwhile, the order processing service is also using the **Saga pattern to manage its own transaction.** If the shipping service reports that the order has been shipped successfully, the order processing service will mark the order as completed.

If the shipping service reports a failure, the order processing service will initiate a compensating transaction to cancel the order and return any funds that were paid.

Overall, the Saga pattern provides a way to manage complex transactions across multiple microservices in a way that ensures consistency and reliability. If you have to just learn one patter, you better learn SAGA patterns as its immensely helpful in Microservice applications.

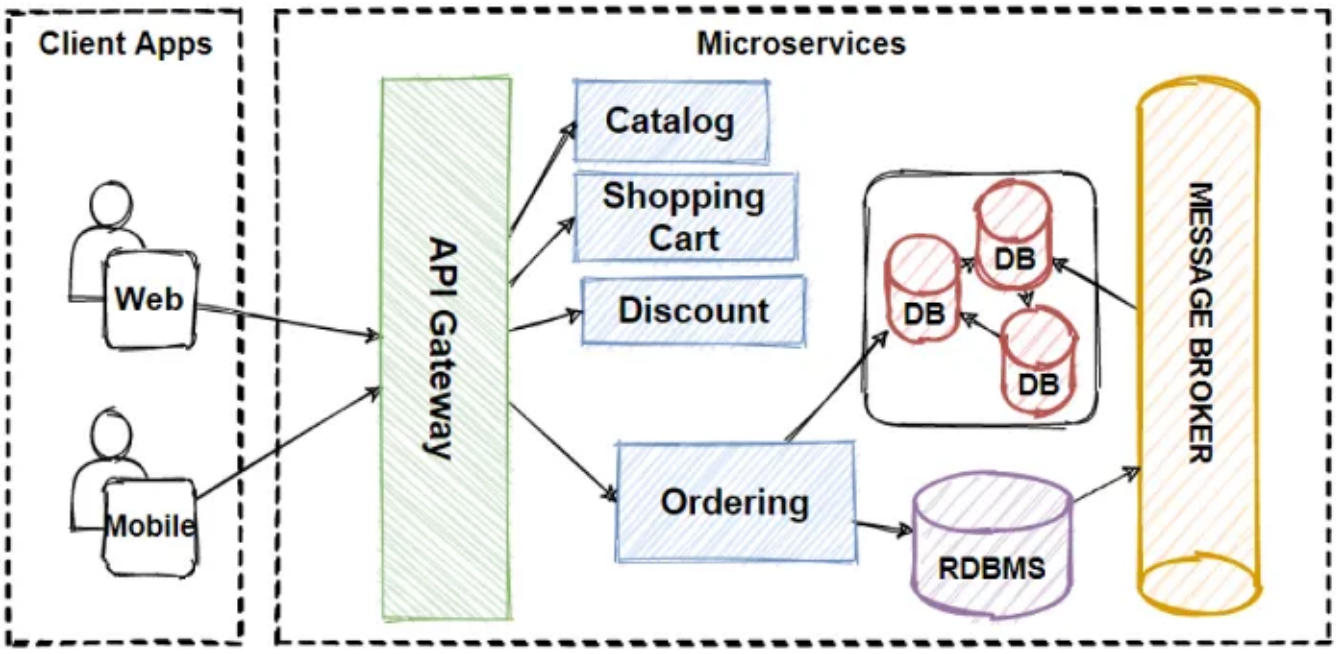
5. Event Sourcing Pattern

Event Sourcing is a Microservice pattern used for persisting and querying data in an application. Instead of storing the current state of an object, Event Sourcing persists all events that occur in the application, allowing the state of the object to be reconstructed at any point in time.

In this pattern, every state change in the application is captured as an event and stored as a log of events. The state of the application can be reconstructed by replaying these events. This means that **Event Sourcing provides an audit log of all changes that occur in the application.**

For example, consider an e-commerce application. When a user places an order, an `OrderPlaced` event is generated and stored in the log. When the order is shipped, a `ShipmentMade` event is generated and stored in the log.

If the order is canceled, a `OrderCanceled` event is generated and stored in the log. By replaying the events, the current state of the order can be determined.



Event Sourcing has several benefits:

1. **Auditability**: All changes to the system can be audited and traced back to their source.
2. **Scalability**: Events can be processed in parallel, allowing for better scalability.

3. **Flexibility**: Because the events are the source of truth, it's possible to change the way the data is queried and persisted without changing the data itself.
4. F**ault-tolerance**: Because the events are immutable, they cannot be modified, ensuring that the data is always correct.

However, implementing Event Sourcing can be complex and requires careful planning. Additionally, querying the data can be slower because it involves replaying all events so you make sure that you really need it before using it. More often then not, there will be an alternative solution.

6. Command Query Responsibility Segregation (CQRS) Pattern:

Command Query Responsibility Segregation (CQRS) pattern is another popular Microservice design pattern that separates **commands (write operations)** and **queries (read operations)** into separate models, each with *its own database.*

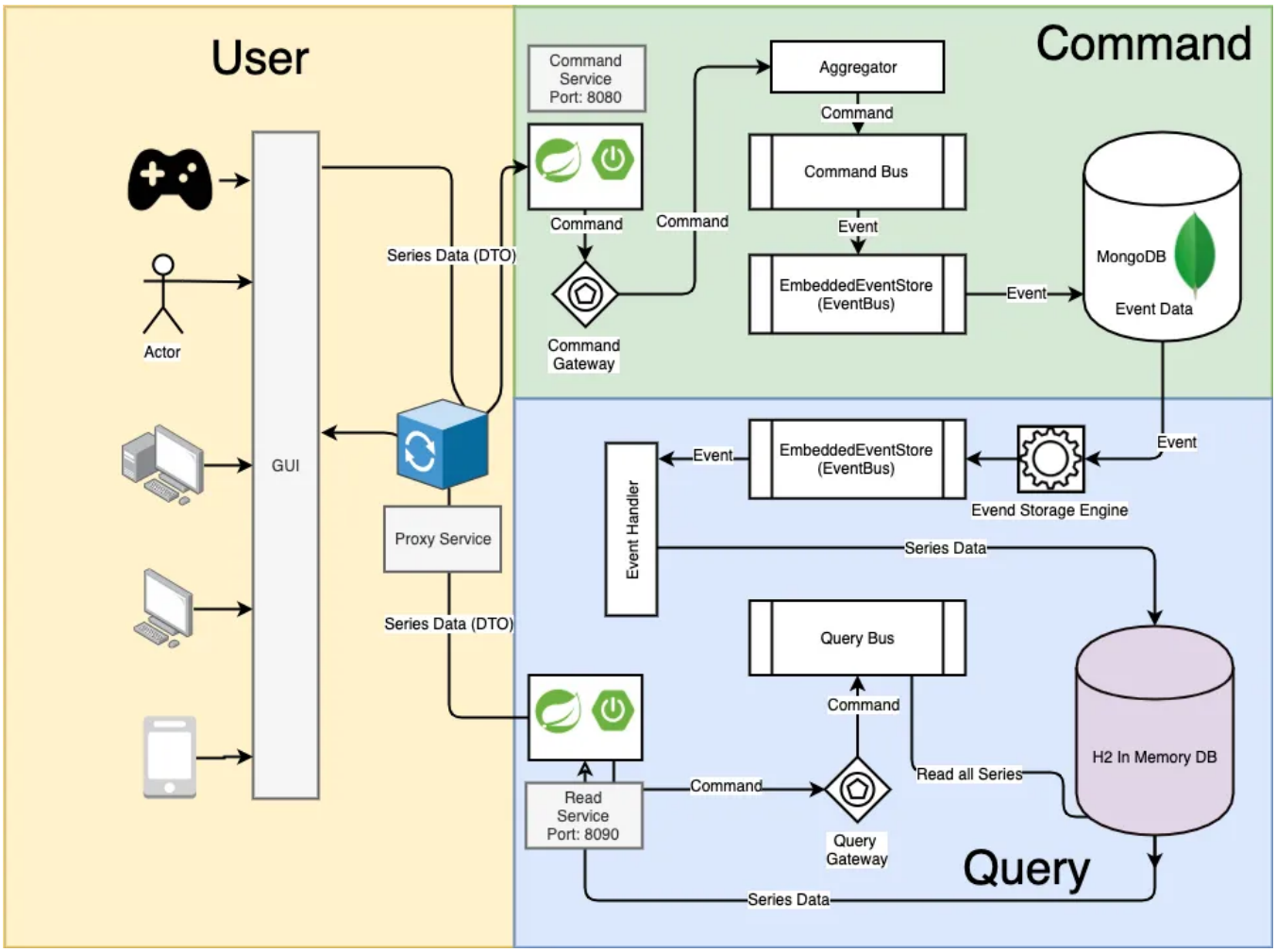
The pattern is based on the idea that the *models used for writing data are not the same as the models used for reading data.*

In this pattern, a **Command Model receives commands from the client** and writes to the database. The Query Model reads from the database and sends data to the client. The pattern can be used to improve the performance and scalability of a system, as each model can be optimized for its specific task.

For example, consider an e-commerce application that uses a traditional CRUD-based approach to managing product information. The same model and database are used to handle both reading and writing product information. As the application grows, the model becomes increasingly complex and the database becomes a performance bottleneck.

Using ***CQRS***, the application would have a separate Command Model for writing product information, and a separate Query Model for reading product information. The Command Model would be optimized for fast writes, while the Query Model would be optimized for fast reads.

The Command Model would store data in a write-optimized database, while the Query Model would store data in a read-optimized database. The two models would communicate through an event bus or message queue.

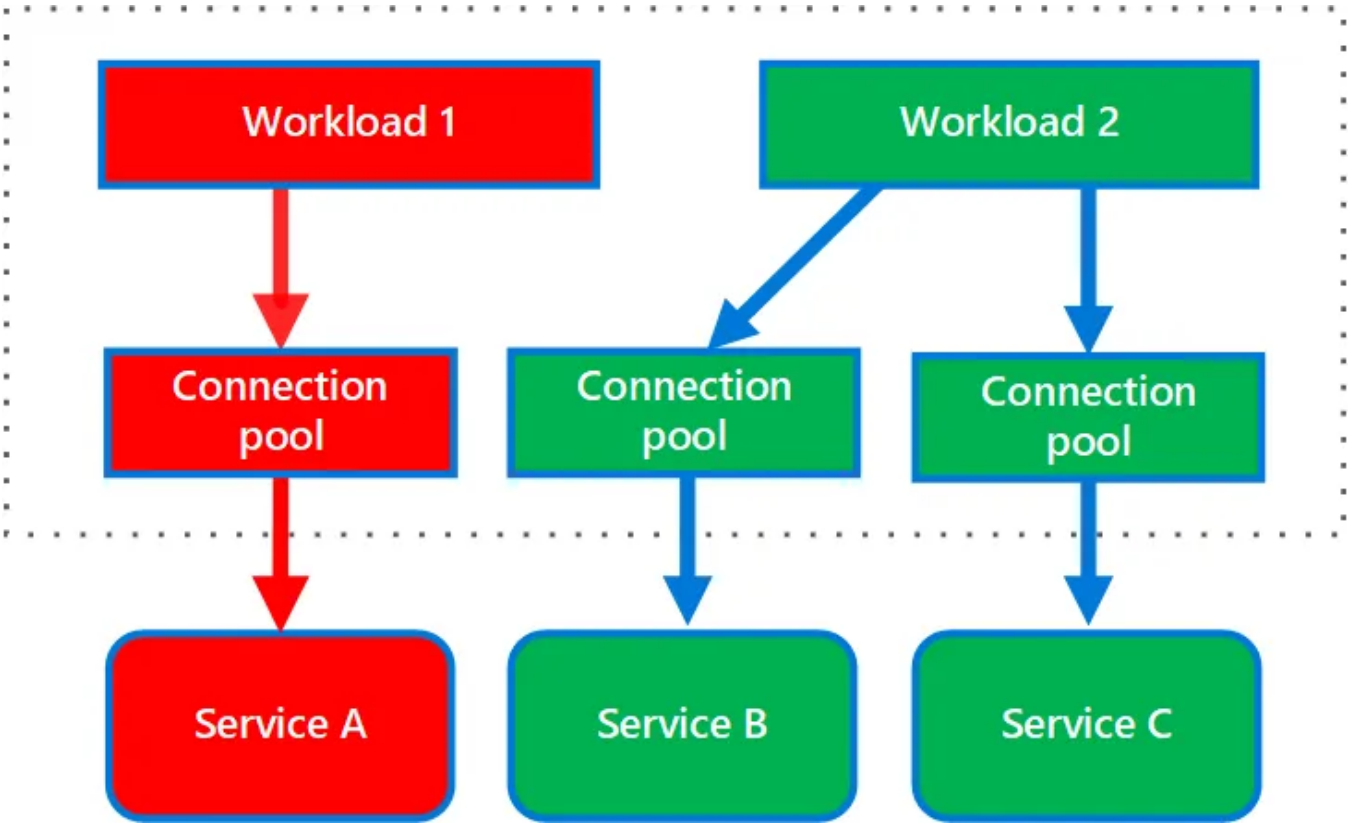


Overall, CQRS can improve the scalability and performance of a system, as well as simplify the code base by separating concerns. However, it can also add complexity and require more development effort, as separate models and databases are required.

7. Bulkhead Pattern

The bulkhead design pattern is a way of isolating different parts of a system so that a failure in one part does not affect the rest of the system. In a Microservice architecture, the bulkhead pattern can be used to isolate different microservices so that a failure in one microservice does not bring down the entire system.

This looks quite similar to circuit breaker pattern which also prevents cascade failure but instead of breaking circuit this design pattern focuses on isolation and self-sufficient Microservices as shown in below diagram than Service A has its own connection pool which is not shared between Service B and Service C.



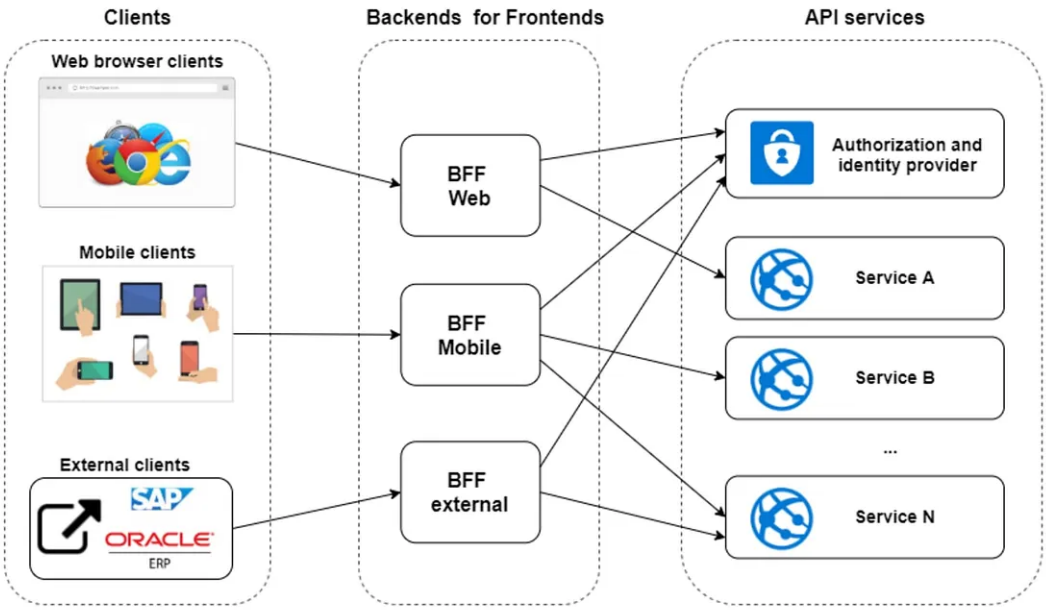
8. Backends for Frontends (BFF)

Backends for Frontends (BFF) is a design pattern used in microservice architecture to handle the complexity of client-server communication in the context of multiple user interfaces. It suggests having a separate back-end service for each frontend to handle the specific needs of that interface.

This **allows developers to optimize the data flow, caching, and authentication mechanisms for the unique needs of the front-end** while keeping the back-end services modular and decoupled.

For example, suppose you have a web application and a mobile application that need to access the same set of services. In this case, you can create separate back-end services for each application, each optimized for the specific platform.

The web application back-end can handle large amounts of data for faster loading, while the mobile application back-end can optimize for lower latency and network usage.



This pattern enables teams to optimize the user experience for each interface by using separate back-end services for each of them. It also allows them to avoid having a single back-end service that needs to serve many different interfaces with different needs, which can become increasingly complex and hard to maintain.

Summary

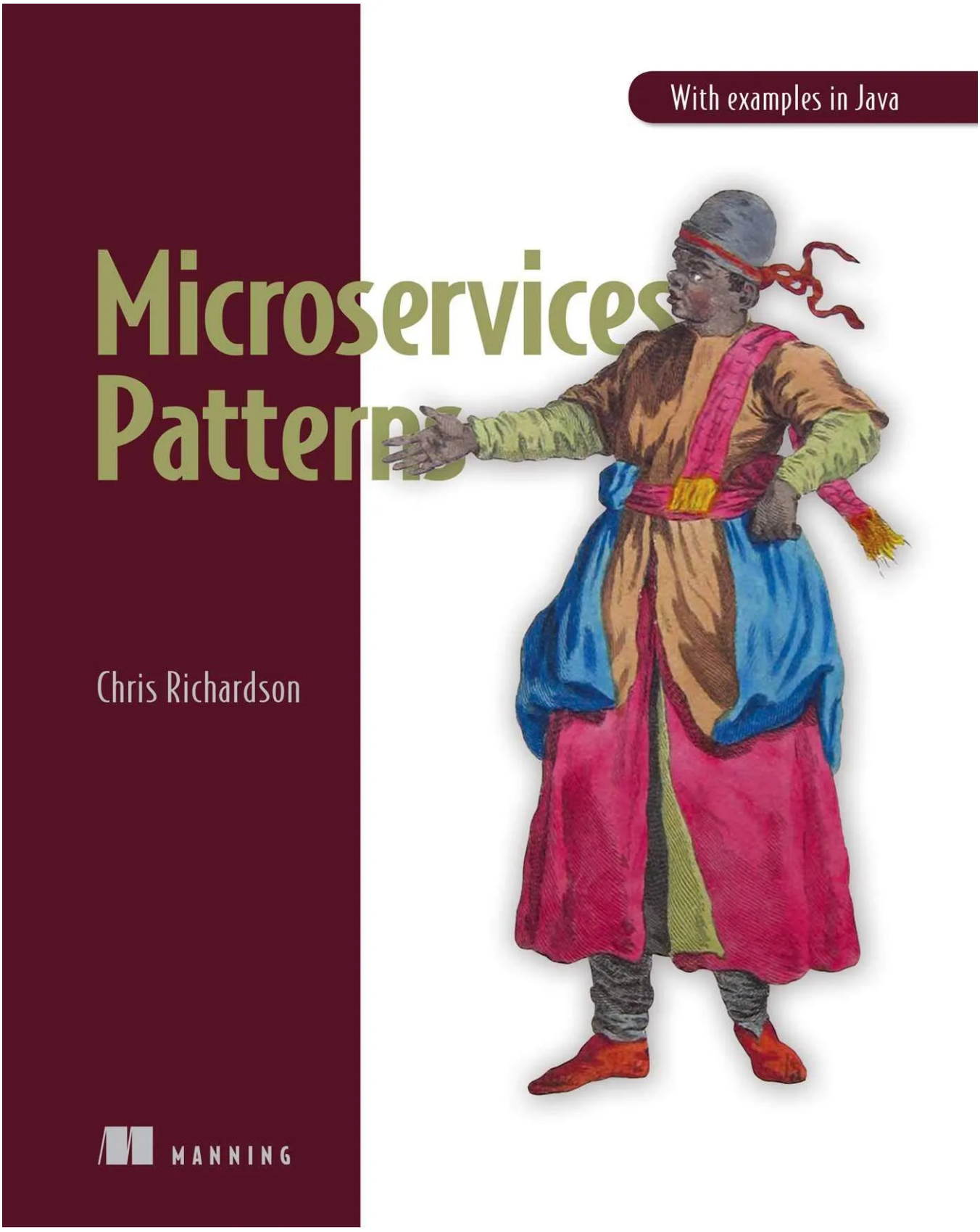
Wow, that's a long write and read but now that we have learned basic stuff about essential Microservice design patterns and now is the time to summarize it in one words so that you will remember them easily and decide when to use them

Here are some popular Microservice design patterns that a programmer should know:

1. **Service Registry** pattern provides a **central location** for services to register themselves.
2. **Circuit Breaker** allows your services to fail fast and **prevent cascading failures**, the circuit breaker pattern is used to isolate a failing service.
3. **API Gateway** provides a **common entry point** for all the requests and responses from the system. So, client only remember one host/port address rather than multiple IPs for each Microservice.
4. **Event-Driven Architecture** allows services to communicate with each other by emitting events.

5. In **Database per Service** *each service has its own database*, which allows services to operate independently.
6. Command Query Responsibility Segregation (**CQRS**) s*eparates read and write requests for an application*, allowing better scaling and performance.
7. **Externalized Configuration** allows *storing configuration data outside of the application code*, making it easier to manage configuration changes.
8. **Saga Pattern** manages the transaction for **long-running transactions** that span multiple services.
9. **Bulkhead Pattern** isolates failures within a microservice, so a single failure does not bring down the entire system.
10. B**ackends for Frontends (BFF)** design pattern provides a specific backend for each client. It allows the front-end team to develop features and add new client-specific functionality quickly.

And, if you want to learn more, there is a nice book called **Microservices Pattern**s by Chris Richardson which is definitely worth reading for Java developers.



Java and Spring Interview Preparation Material

Before any Java and Spring Developer interview, I always use to read the below resources

Grokking the Java Interview

Grokking the Java Interview: click here

I have personally bought these books to speed up my preparation.

You can get your sample copy here, check the content of it and go for it

Grokking the Java Interview [Free Sample Copy]: click here

> *If you want to prepare for the Spring Boot interview you follow this consolidated ebook, it also contains microservice questions from spring boot interviews.*

Grokking the Spring Boot Interview

You can get your copy here — **Grokking the Spring Boot Interview**

That's all about **10 essential Microservice design patterns** which I think every programmer should know and learn. Each pattern has its own advantages and disadvantages, and the choice of pattern will depend on the specific needs of the application but unless you know what they do and what problem they solve, you cannot use them in real world that's where this article help.

By the way, these are just a few of the many microservice design patterns that are available which you can learn on your own, I will also be sharing more patterns in future. .

And, if you are not a Medium member then I highly recommend you to join Medium so that you can not only read these articles but also from many others like Google Engineers and Tech experts from FAANG companies. You can **join Medium here**

Join Medium with my referral link - Soma

Read this and every story from thousands of great writers on Medium). Read more from great writers on Medium like...

medium.com

Microservices Programming Java Software Development

Microservice Architecture

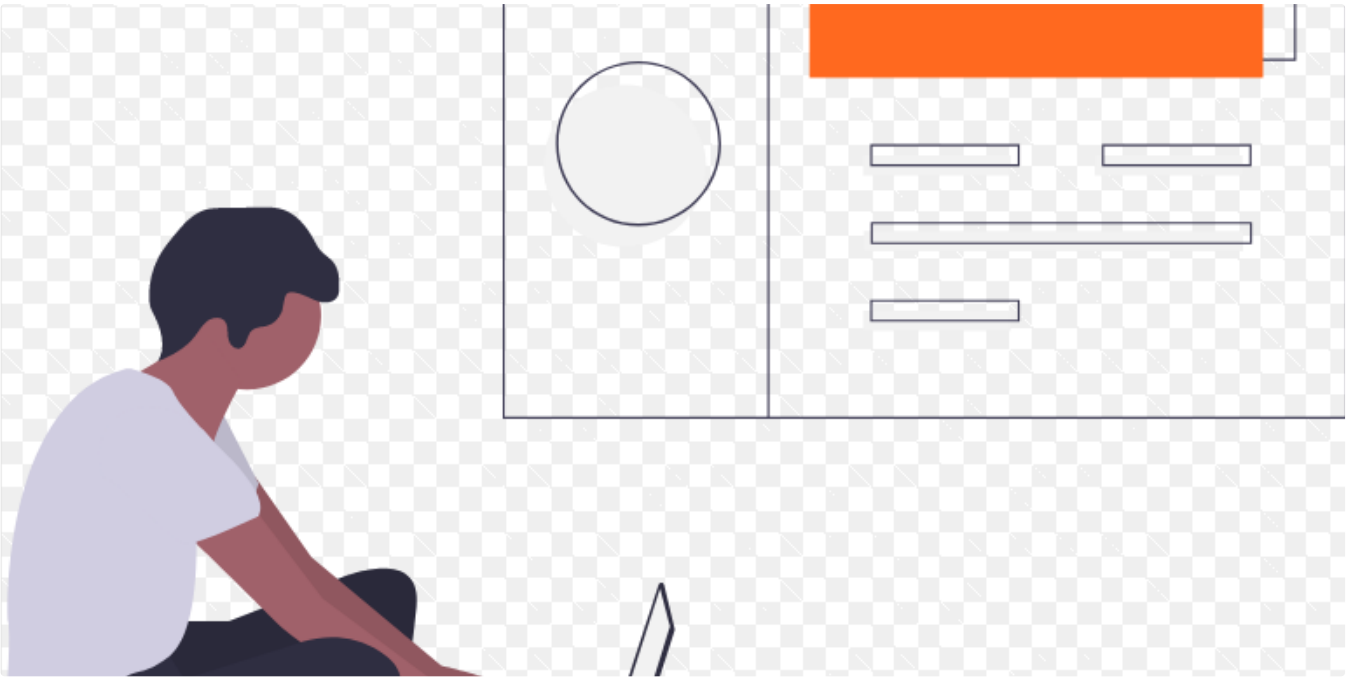
Follow

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



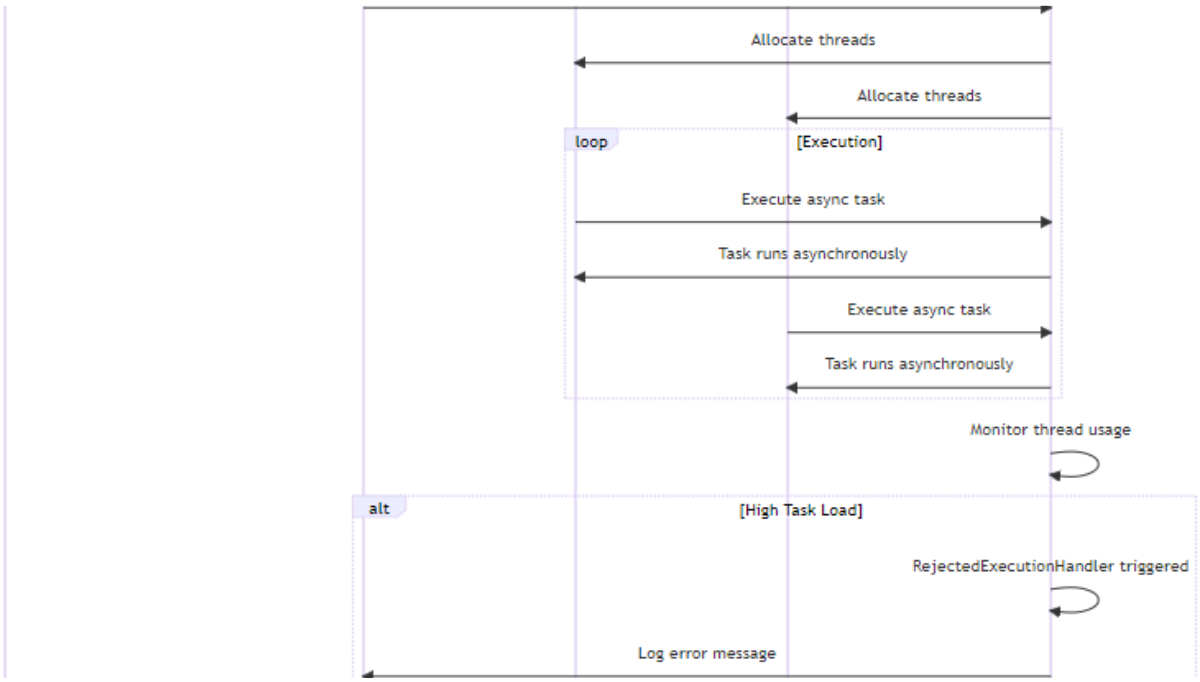
Soma in Javarevisited

Difference between @RequestMapping and @GetMapping Annotations in Spring MVC Web Framework

Hello friends , if you are preparing for Java Developer and Spring Developer interviews then you must prepare for questions like...

✨ · 6 min read · Aug 19

49



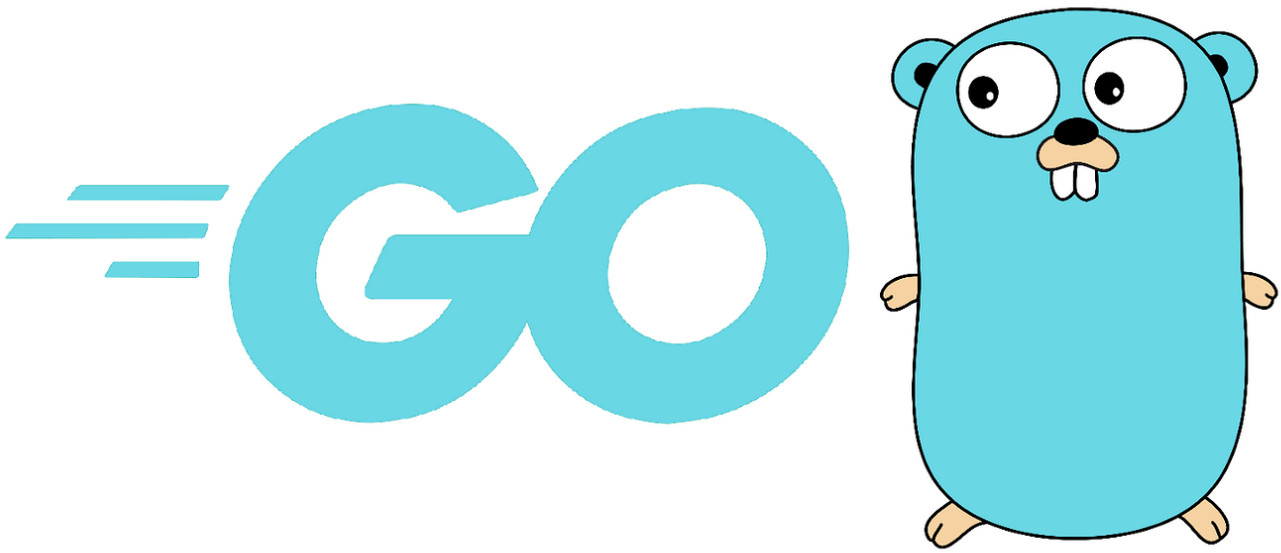
Srikanth Dannarapu in Javarevisited

Using Async Schedulers in Spring Boot

Problem Statement: Efficiently Managing Multiple Asynchronous Schedulers in a Spring Boot Application

5 min read · Aug 7

109 1



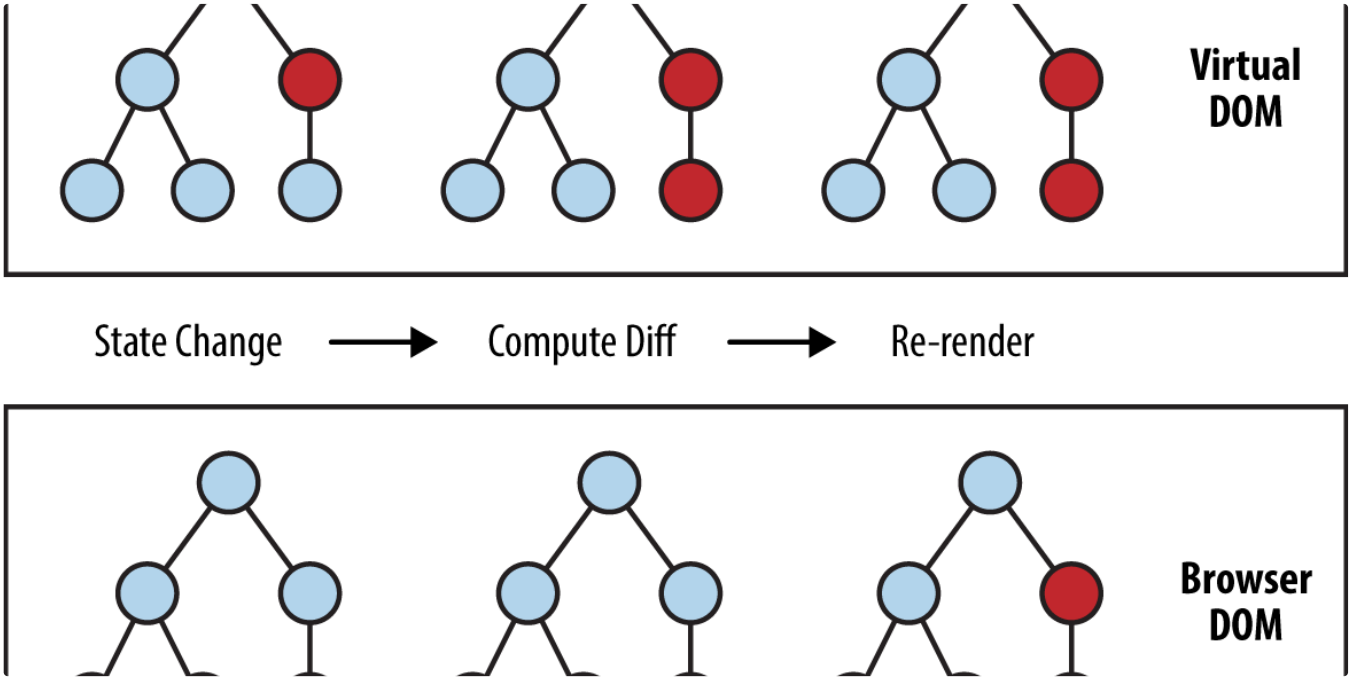
javinpaul in Javarevisited

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



Soma in JavaScript in Plain English

10 React Features That Make Frontend Development Easier

Few reasons why React.js is considered one of the best frontend library of our time

· 7 min read · Jul 27

155

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

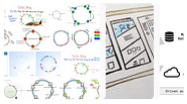
Ionut Anghel

REST Endpoint Best Practices Every Developer Should Know

5 min read · Mar 18

79 1

Lists



General Coding Knowledge

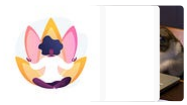
20 stories · 257 saves

It's never too late or early to start something

15 stories · 93 saves

Coding & Development

11 stories · 129 saves



Stories to Help You Grow as a Software Developer

19 stories · 309 saves

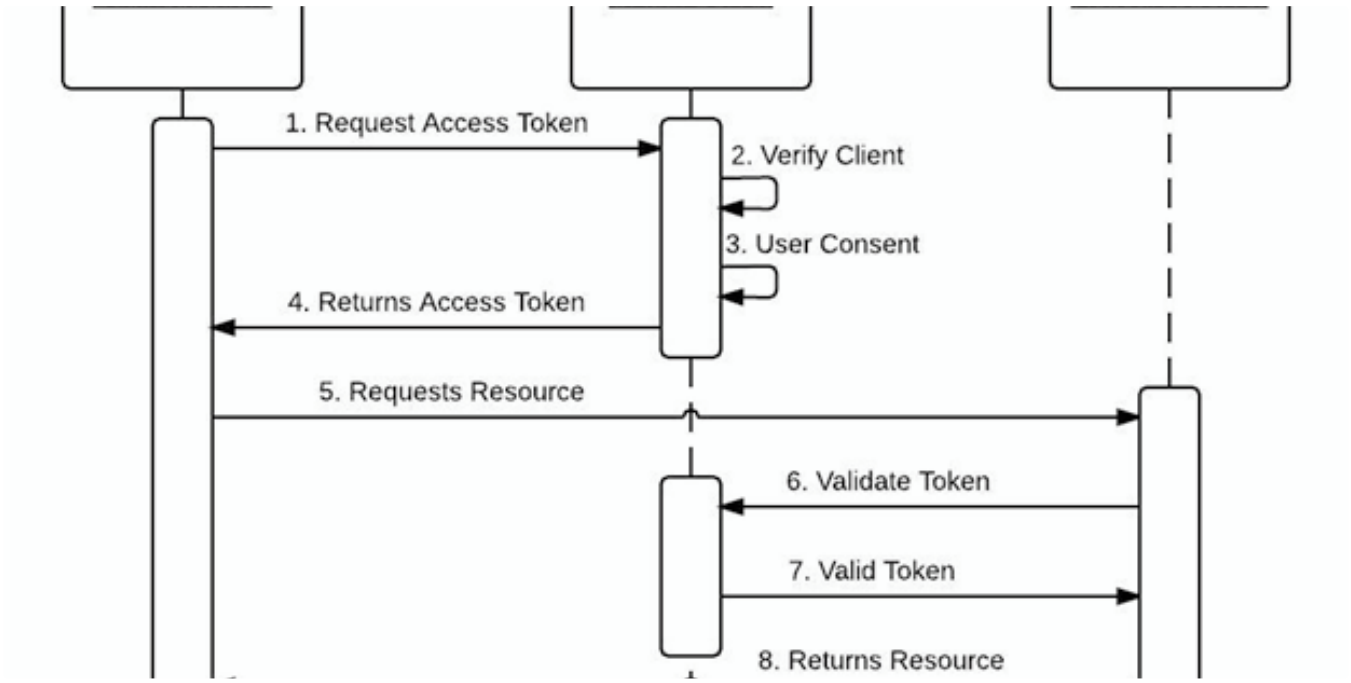
Bubu Tripathy

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



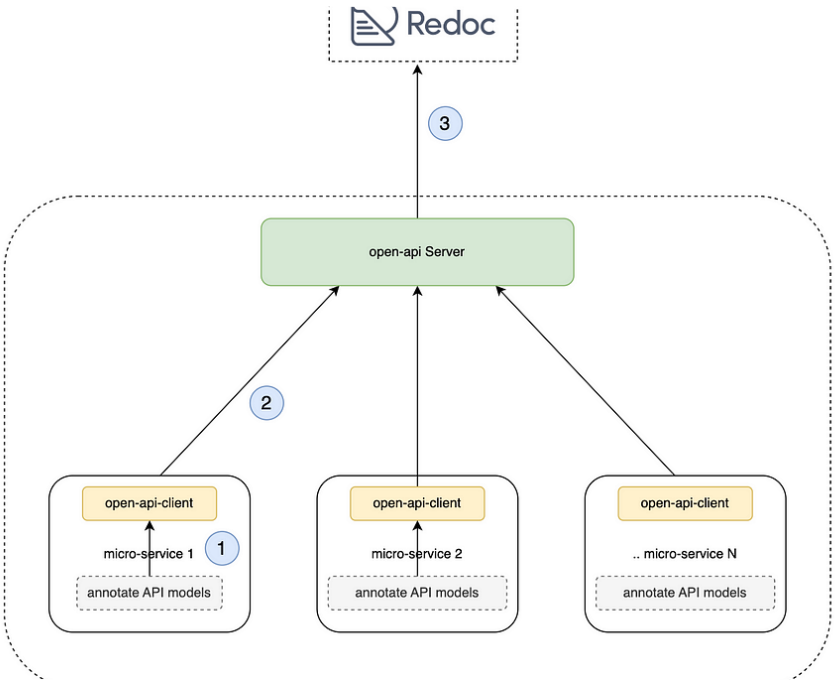
Somal Chakraborty in Dev Genius

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



Rajesh Gupta

Centralised API Documentation

Problem Statement

4 min read · Jun 18

14

Eray Araz

Spring Webhook

Introduction

3 min read · Apr 2

 109  1

See more recommendations