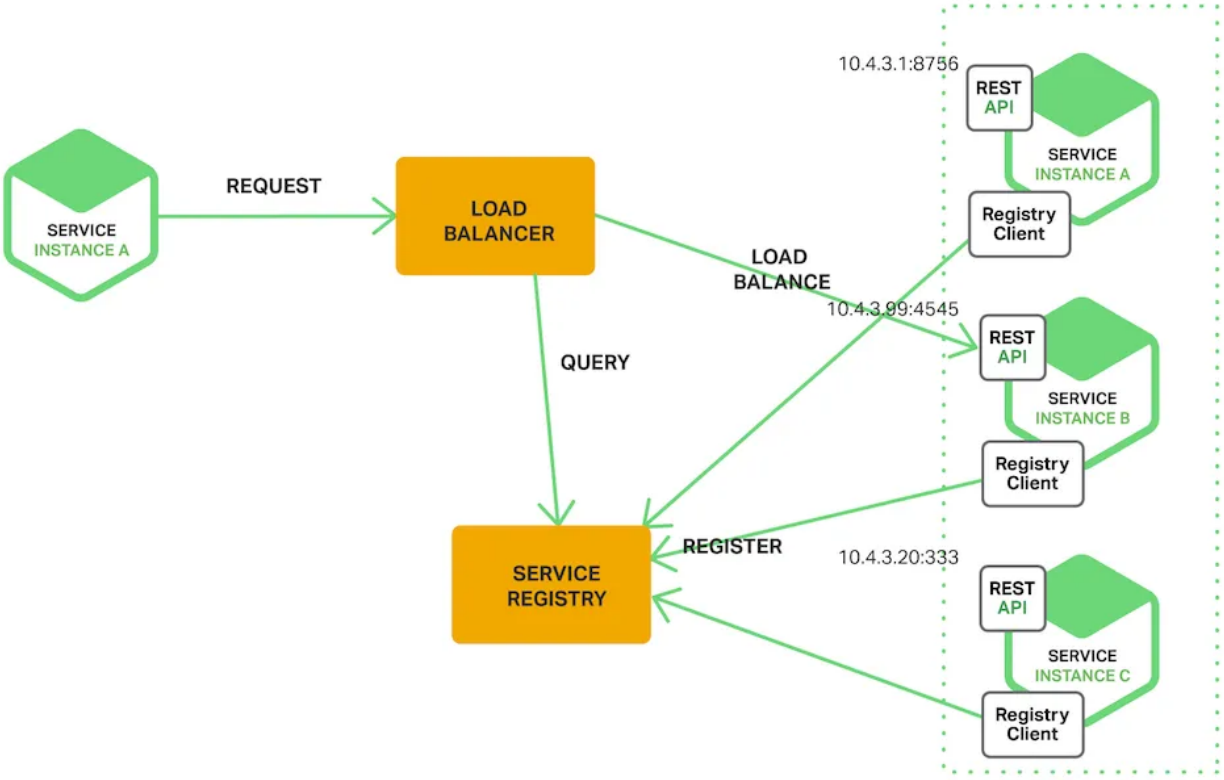
Member-only story

Service Registry Design Pattern in Microservices Explained

Discover, Load Balance, and Scale: Understanding the Service Registry Pattern in Microservices Architecture

Soma · Follow
Published in Javarevisited
15 min read · Apr 16

Listen Share More



Hello folks, if you are wondering What is Service Registry pattern in Microservices and how does it work then you have come to the right place. In past articles, we have seen popular Microservice design patterns like Event Sourcing, CQRS, SAGA,

Database Per Microservices, API Gateway, Circuit-Breaker and also best practices to design Microservices and in this article, we will learn about Service registry pattern.

Service Registry Pattern is a design pattern commonly used in microservices architecture to enable service discovery and dynamic load balancing. In this pattern, **microservices register themselves with a service registry**, which acts as a central repository for service metadata.

> *This metadata includes information such as service endpoint URLs, version numbers, and other configuration details.*

When a microservice needs to interact with another microservice, it queries the service registry to obtain the location and configuration of the target service. This allows microservices to dynamically discover and communicate with each other without hard-coding service endpoint URLs or configuration details in their code.

The **Service Registry Pattern provides several benefits** in a microservices architecture, including:

1. **Dynamic Service Discovery**: Microservices can discover the location and configuration of other services at runtime, allowing for flexible and dynamic communication between services without the need for static configuration.
2. **Load Balancing**: Service registry can also implement load balancing strategies, such as round-robin, to distribute requests across multiple instances of the same service, improving performance and scalability.
3. **Fault Tolerance**: Service registry can detect and remove failed or unavailable services from the registry, allowing microservices to automatically adapt to changes in the availability of services.
4. **Scalability**: Service registry can handle a large number of services and instances, making it suitable for large-scale microservices architectures.

However, there are also some considerations to keep in mind when using the Service Registry Pattern, **such as potential single point of failure,** increased complexity in managing and monitoring the service registry, and potential performance overhead in querying the registry.

Now, that we have learned what is Service registry pattern and what benefits it provide, let's deep dive into it and understand how it works and how we can setup a service registry in Java Microservices using Spring Cloud Eureka Server and Eureka client.

But, 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**

How does Service Registry design pattern works?

The Service Registry pattern, a **central registry or service registry acts as a directory for all the services in the system**, providing a way for services to register themselves and for clients to discover and locate the available services.

Here's how the Service Registry pattern typically works:

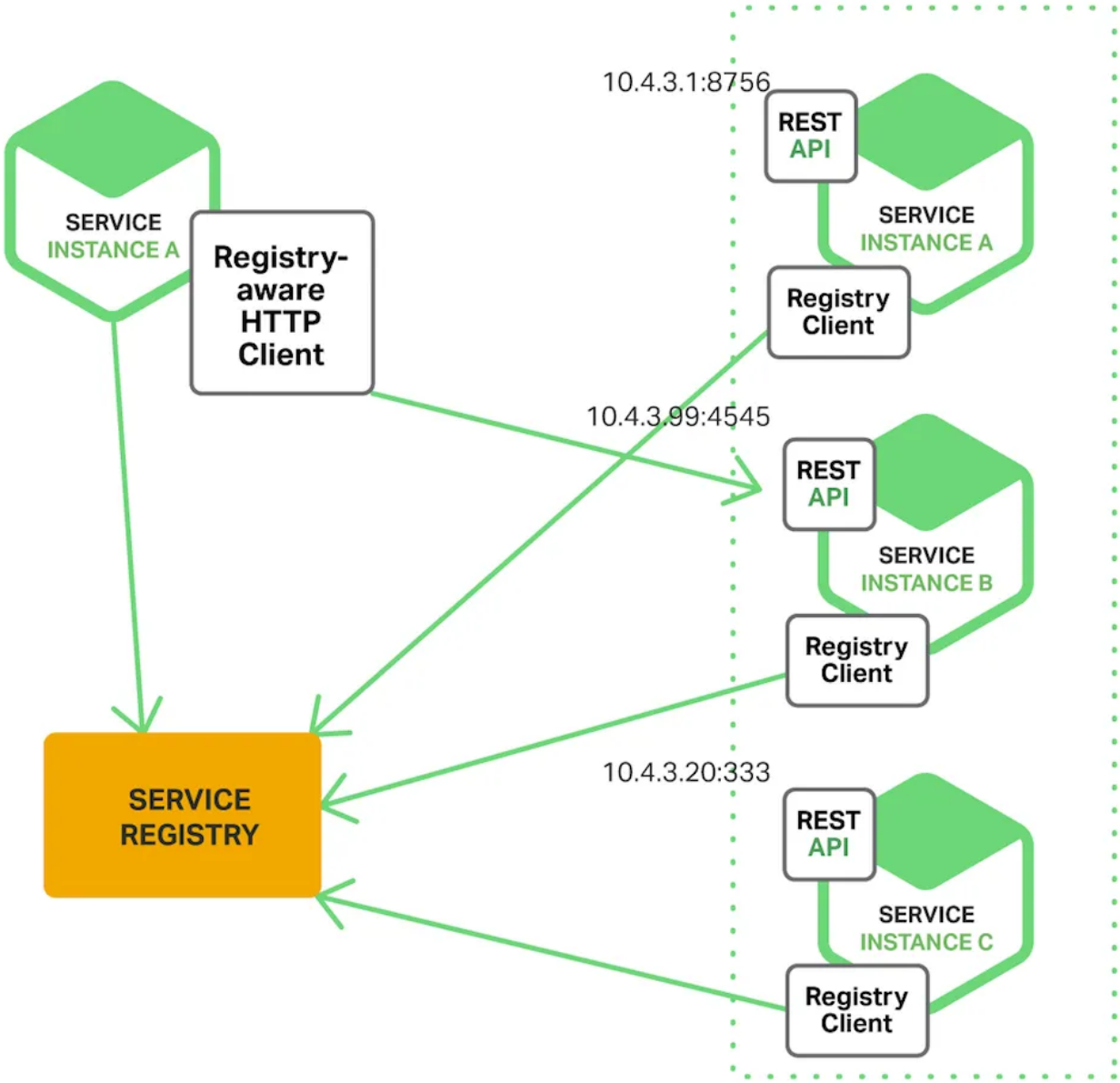
1. **Service Registration**: When a microservice starts, it registers itself with the Service Registry by providing its metadata, such as service name, endpoint URL, version, and other relevant information.
2. **Service Discovery:** Clients, which are other microservices or applications, can then query the Service Registry to discover the available services and their respective endpoints. This allows services to dynamically discover and locate other services without hard-coding their endpoints, enabling flexibility and scalability in a distributed system.
3. **Load Balancing**: The Service Registry can also be used for load balancing. It can keep track of the health status of registered services and distribute incoming requests to available and healthy instances of the service. This helps in achieving load balancing across multiple instances of a service, ensuring optimal utilization of resources and high availability.

4. **Dynamic Updates:** The Service Registry allows services to dynamically update their registration information, such as IP address, port, or health status, so that clients always have up-to-date information about the available services.

5. **Fault Tolerance:** The Service Registry can also provide fault tolerance by monitoring the health of registered services and automatically removing failed or unresponsive services from the registry. This ensures that clients do not attempt to communicate with unavailable or unhealthy services.

Overall, the Service Registry pattern plays a crucial role in enabling dynamic discovery, load balancing, and fault tolerance in a microservices architecture, making it a key component for building scalable and resilient microservices-based applications.

Here is a nice diagram which shows **how Service registry pattern works in Microservice architecture**:



How to use Service Registry Pattern in Java Microservice using Spring cloud? Example

Here's an example of implementing the Service Registry pattern using Spring Cloud and Netflix Eureka, which is a popular service registry and discovery solution for building microservices-based applications in Spring

In this example, the Eureka server acts as the service registry, and microservices can register themselves with the Eureka server by using the `@EnableDiscoveryClient` annotation.

Clients can then discover the available services by querying the Eureka server, which provides the necessary metadata, such as service name and endpoint URL,

for locating the services.

Here are the **steps to setup Eureka Server as Service Registry in Java Microservices with Spring Boot and Spring Cloud:**

1. Add the required dependencies to your Spring Boot project's `pom.xml` file:

```
<!-- Spring Cloud Eureka Server -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

2. Add the **`@EnableEurekaServer` annotation** to your Spring Boot application's main class to enable the Eureka server:

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }
}
```

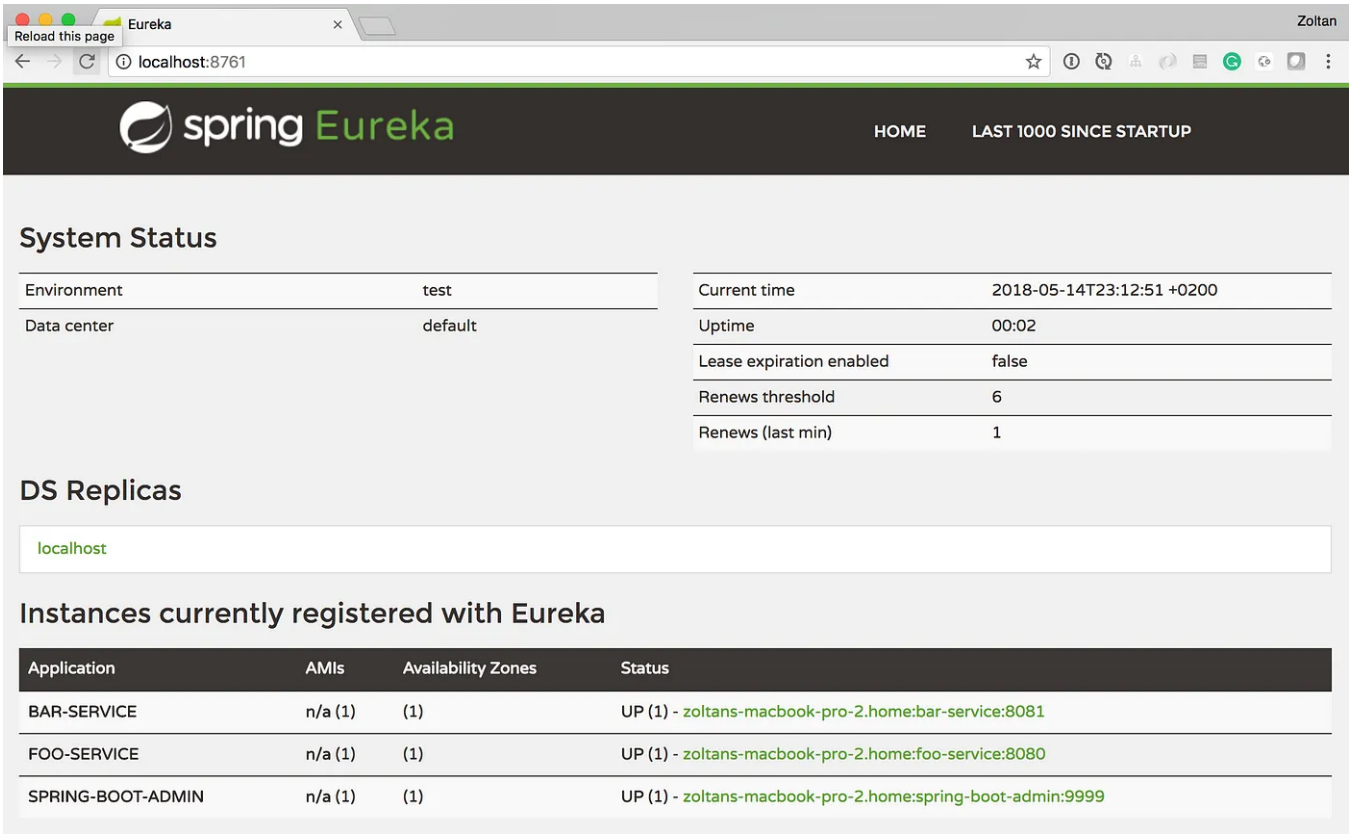
3. Configure the **Eureka server properties** in the `application.properties` or `application.yml` file:

```
spring.application.name=service-registry
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

4. **Register your microservices with the Eureka server using the `@EnableDiscoveryClient` annotation** in your microservices' main classes:

```
@SpringBootApplication
@EnableDiscoveryClient
public class MyMicroserviceApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyMicroserviceApplication.class, args);
    }
}
```

5. **Access the Eureka server's dashboard to view the registered services** and their endpoints. The dashboard is typically available at `http://localhost:8761` by default.



Btw, This is just a simple example, and there are many other configurations and features available in Spring Cloud and Netflix Eureka for advanced use cases, such as *load balancing*, service health monitoring, and security. Please refer to the official documentation for more information.

Pros and Cons of Service Registry Design Pattern in Microservice Architecture

The Service Registry pattern in microservices architecture has several pros and cons:

Here are the pros and key benefits of using Service Registry pattern in Microservices:

1. **Dynamic Service Discovery**: Services can register and un-register themselves with the service registry at runtime, allowing for dynamic service discovery. This makes it easier to add, remove, or update services without having to manually configure and update service endpoints in clients.
2. **Load Balancing**: Service registry can provide load balancing capabilities by distributing requests among multiple instances of the same service. This can improve the scalability and availability of microservices, as well as optimize resource utilization.
3. **Fault Tolerance**: Service registry can detect and automatically remove failed or unresponsive services from its registry, reducing the impact of service failures and improving the overall fault tolerance of the system.
4. **Scalability:** Service registry can be scaled horizontally to handle large numbers of services and clients, making it suitable for large-scale microservices architectures.
5. **Centralized Configuration**: Service registry can store metadata and configuration information about services, allowing for centralized configuration management and reducing the need for duplicating configuration in each microservice.

Now that we have seen all the pros, its time to look into compromises you need to make and all the cons which comes with Service registry pattern in Microservice architecture:

1. **Additional Infrastructure**: Service registry introduces additional infrastructure components, such as the service registry server, that need to be deployed, managed, and maintained. This may add complexity to the overall system architecture and operational overhead.
2. **Single Point of Failure**: This is probably the biggest risk when you use Service registry pattern in Microservices. If the service registry server becomes unavailable, it can impact the availability of the entire system, as clients may not be able to discover services or perform load balancing. This requires careful

design and configuration to ensure high availability and fault tolerance of the service registry.

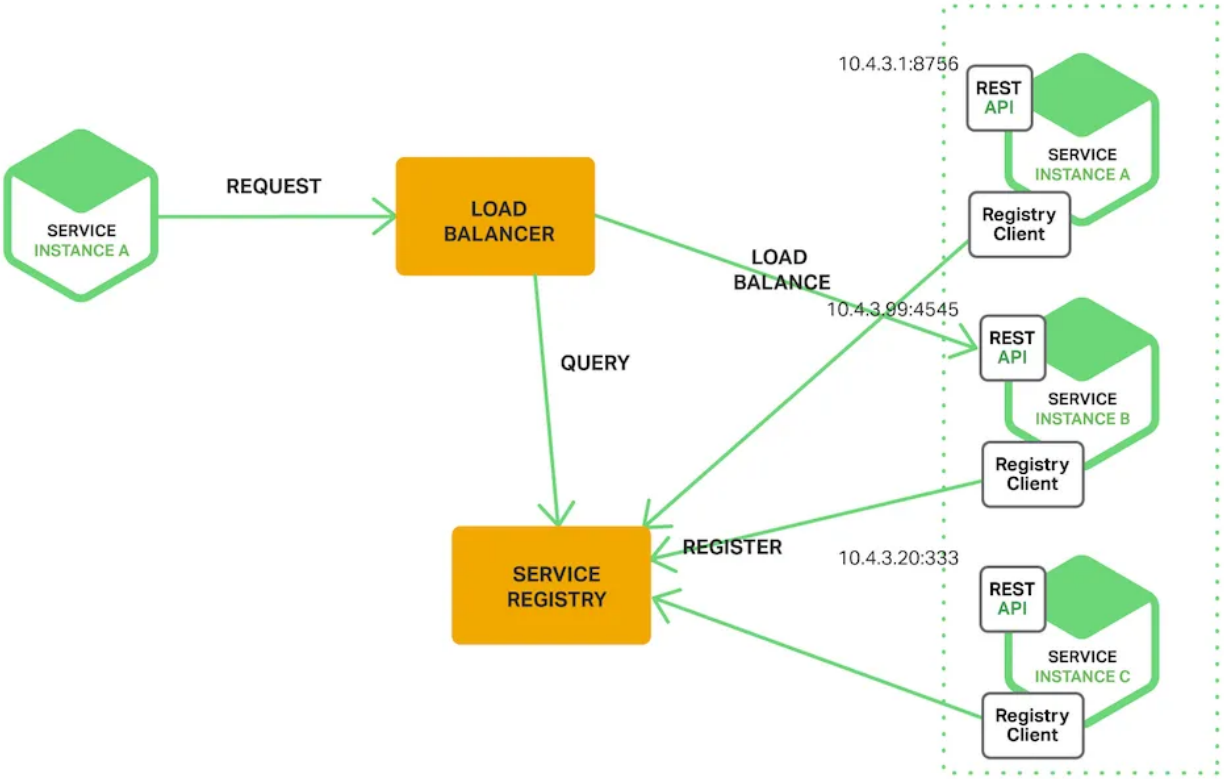
3. **Increased Latency**: Service discovery through a service registry may introduce additional network latency, as clients need to query the service registry to discover the location of services. This may impact the performance of the system, especially in high-traffic scenarios.

4. **Coupling to Service Registry**: Clients need to be aware of the service registry and use its APIs to discover services, which may introduce coupling between clients and the service registry. This can make it harder to switch to a different service registry or change the discovery mechanism in the future.

5. **Complexity**: Implementing and managing a service registry adds complexity to the microservices architecture, and requires additional development efforts, testing, and operational considerations. It may not be necessary for small or simple microservices architectures where service discovery can be achieved through other means, such as static configurations or DNS-based approaches.

Overall, the *Service Registry pattern provides dynamic service discovery, load balancing, and fault tolerance benefits* in microservices architectures, but also introduces additional complexity and operational overhead.

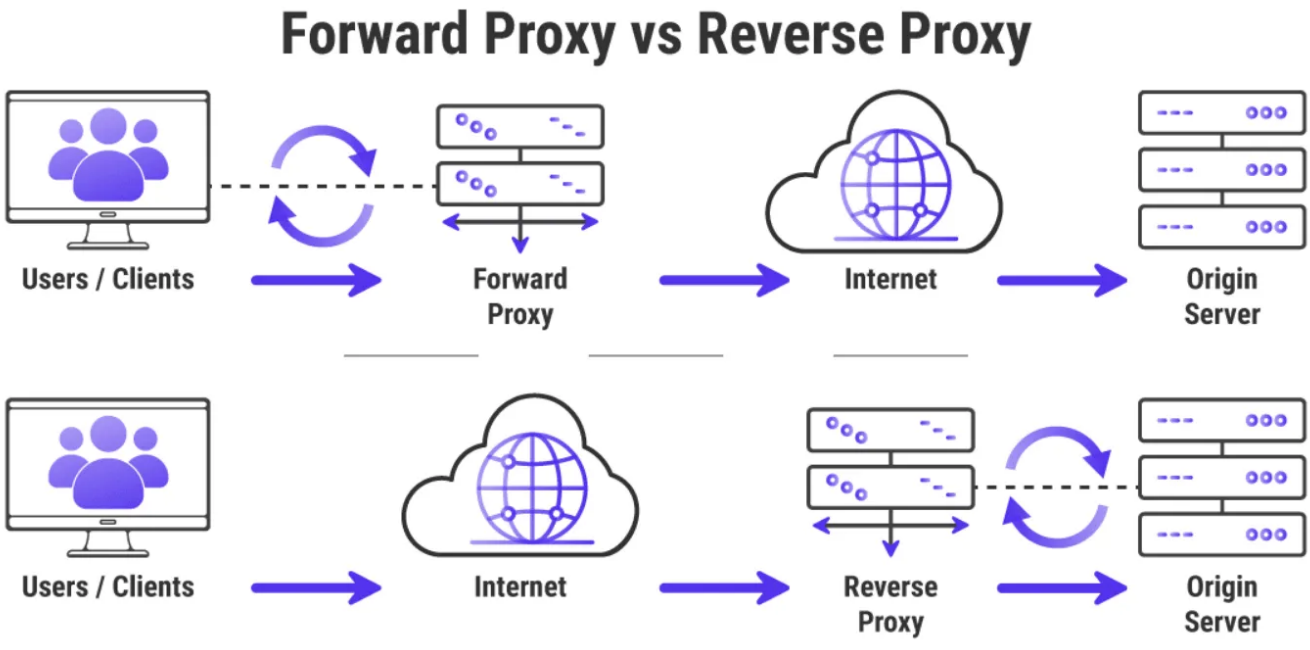
You need to take a call whether to us this pattern or not depending upon the specific requirements and trade-offs of the system architecture before adopting the Service Registry pattern.



Difference between Forward Proxy and Reverse Proxy in System Design

Understanding the Distinctions between Forward and Reverse Proxies in System Design and when to use them

---



image_credit — https://kinsta.com/wp-content/uploads/2020/08/Forward-Proxy-vs-Reverse-Proxy-Servers.png

Hello folks, if you are preparing for System design interview then knowing the difference between forward proxy and reverse proxy is very important, its one of the most frequently asked question on System Design, along with **difference between API Gateway and Load Balancer,** which we have seen earlier.

When designing complex systems, it's common to use proxy servers to improve performance, security, and reliability. Proxy servers sit between clients and servers and help manage traffic between them.

Two types of proxies that are often used are forward proxies and reverse proxies. While both are designed to **improve the performance and security of a system**, they work in different ways and are used in different contexts. In this article, we'll explore the differences between forward proxies and reverse proxies in system design.

By the way, if you are preparing for senior developer interviews then along with System Design you should also familiar with different architectures like Microservices and various Microservice design patterns like **Event Sourcing, CQRS, SAGA, Database Per Microservices, API Gateway, Circuit-Breaker** they will help you immensely during interview as they are often used to gauge your seniority level.

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**

Now, let's come back to the topic and find out more about what is forward and reverse proxies, how to use them, what are their pros and cons and most importantly difference between forward and reverse proxies.

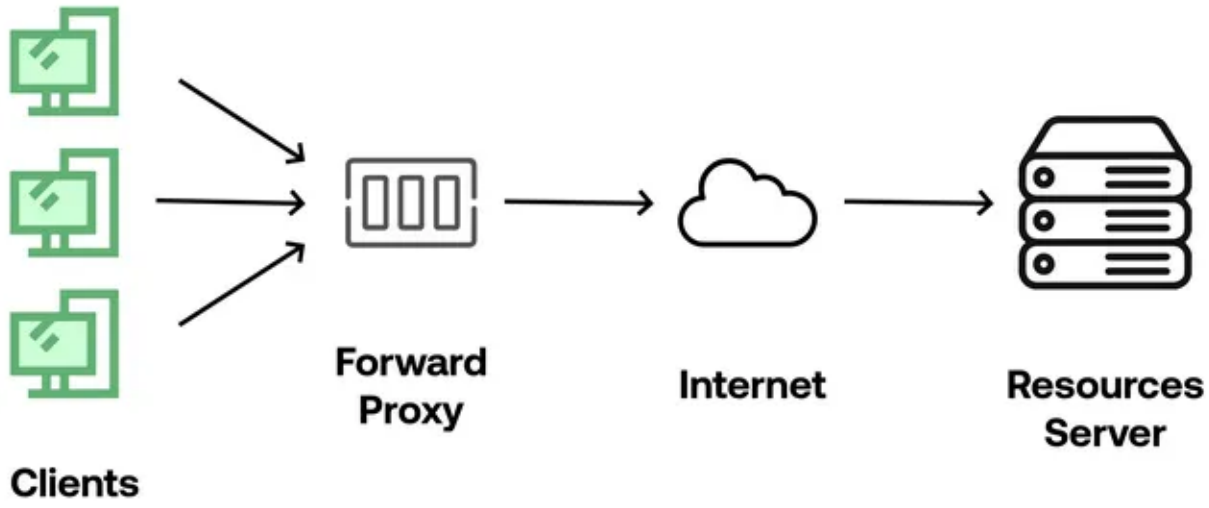
What is Forward Proxy? when to use it?

A forward proxy is a proxy server that sits between a client and the internet. The client requests a resource or service from the internet through the forward proxy,

which acts as an intermediary, forwarding the request on to the internet and then returning the response to the client.

Forward proxies are typically used t**o control access to the internet, filter content, or provide anonymity for the client**. They can also be used to speed up access to resources by caching frequently requested content.

Here is an example of forward proxy:



You can see that clients connect to forward proxy and then it routes request to internet.

What is Pros and Cons of Forward Proxy?

Now that you are familiar with the location of proxy in forward proxy, you can easily find out the pros and cons of it. Here are some pros and cons of using a forward proxy:

Pros:

1. **Enhanced security:** Forward proxy can provide an additional layer of security by hiding the original IP address of the clients accessing the internet.
2. **Improved speed and performance**: Caching frequently requested resources can improve response times for clients accessing the internet.

3. **Access control**: By restricting access to certain resources, organizations can use forward proxies to prevent unauthorized access to sensitive data.
4. **Anonymity:** Users can remain anonymous while browsing the internet, as their IP addresses are hidden.

Cons:

1. **Complex configuration**: Forward proxies require a more complex configuration as they have to be set up on individual devices to be effective.
2. **Single point of failure**: If the forward proxy fails, all the devices that rely on it will also fail to access the internet.
3. **Increased latency**: Forward proxies can increase latency and slow down the overall performance of internet access.
4. **Limited control:** Forward proxies can limit users' ability to access certain resources, leading to frustration and reduced productivity.

Apart from the limitation the access control and security benefit it provides is the main reason for using forward proxy on various architecture. Now, let's see what is Reverse proxy and how does it work.

What is Reverse Proxy? when to use it?

A **reverse proxy is a server that sits between the client and the origin server**, receiving requests from clients and forwarding them to the appropriate server. The response from the server is then returned to the proxy and forwarded to the client. In essence, it helps to protect the origin server from direct access by clients.

Reverse proxies are commonly u**sed as load balancer to balance the load across multiple servers**, improve security by hiding the details of the server infrastructure, and provide other value-added services such as caching and SSL termination.

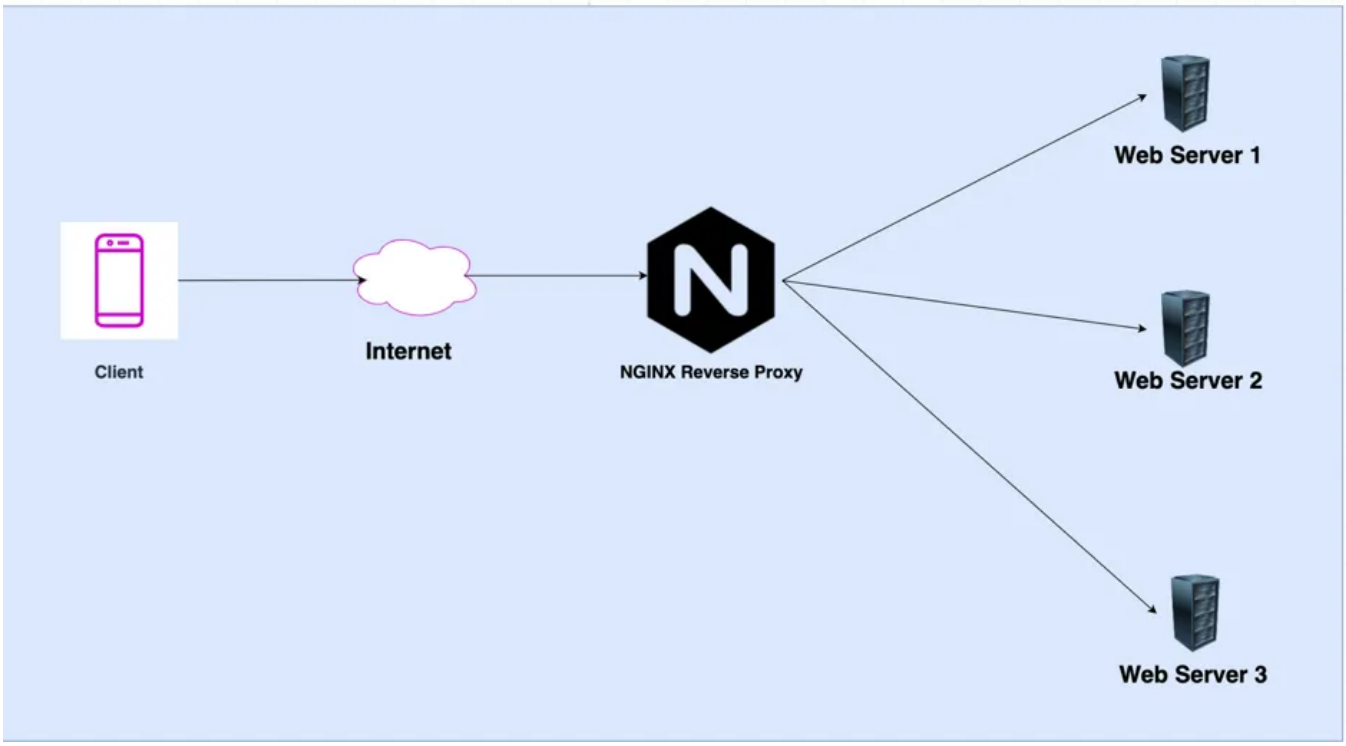
Reverse proxies are often used in the following scenarios:

- **Load balancing:** distributing incoming traffic across multiple servers to improve performance and availability.

- **Security**: protecting the backend servers from direct exposure to the internet and preventing unauthorized access.
- **Scalability**: allowing horizontal scaling of the server infrastructure without affecting the clients.

Reverse proxies **provide a single entry point for clients**, making it easier to manage and monitor the traffic to the backend servers. They also provide a level of abstraction between the clients and servers, allowing the server infrastructure to be modified or upgraded without affecting the clients.

Here is an example of NGINX reverse proxy setup:



You can see that client connects to internet directly but servers are behind proxy so client will never know which server their request are processed, hence security your infrastructure from outside.

What is Pros and Cons of Reverse Proxy Architecture?

Here are some pros and cons of using a reverse proxy architecture:

Pros:

1. **Increased security**: Reverse proxies can provide an additional layer of security by masking the identity and location of backend servers, preventing direct access to them from external clients.

2. **Better scalability:** Reverse proxies can distribute traffic evenly across multiple backend servers, ensuring that no single server becomes overloaded and causing the application to crash.

3. **Improved performance:** By caching and compressing data, reverse proxies can reduce the amount of data that needs to be transferred between clients and servers, leading to faster response times.

4. **Simplified architecture:** Reverse proxies can be used to consolidate multiple backend servers into a single endpoint, simplifying the overall architecture of the application.

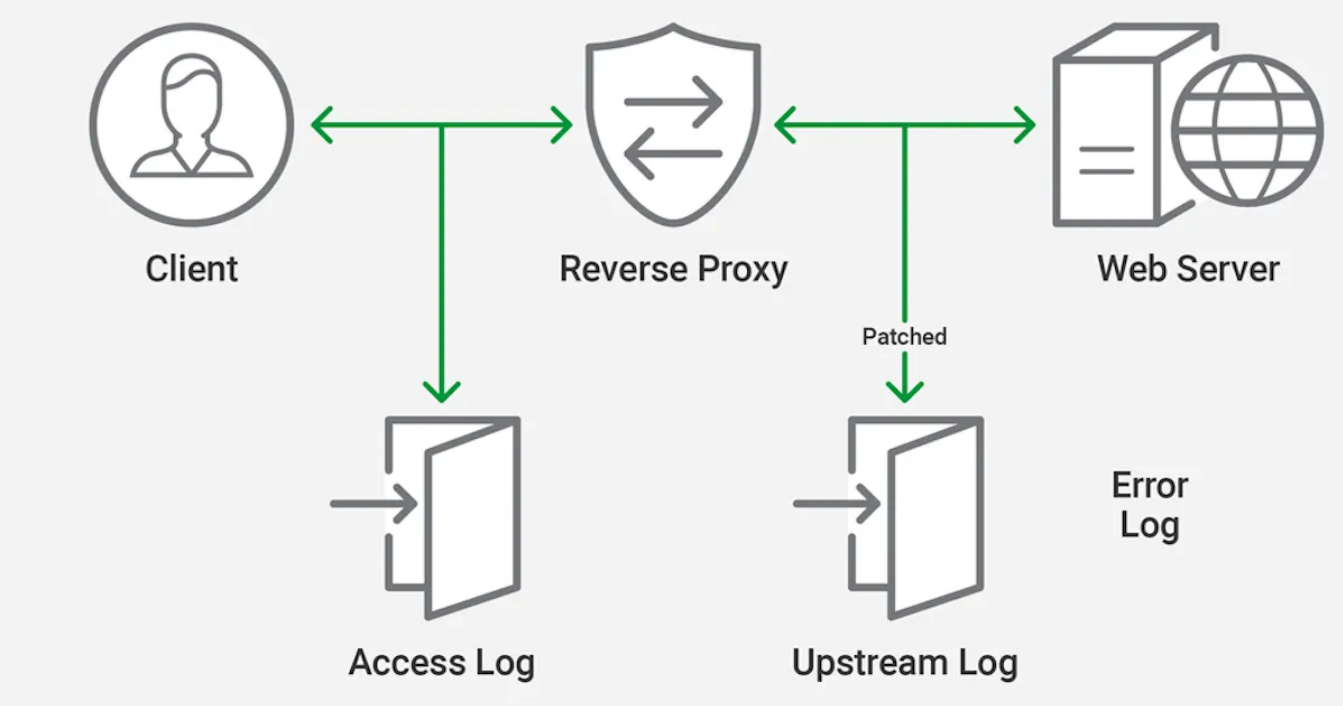
Cons:

1. **Single point of failure:** If the reverse proxy fails, the entire application may become unavailable. This is its biggest drawback.

2. **Increased complexity:** Implementing and maintaining a reverse proxy can be more complex than a simple client-server architecture.

3. **Limited customization:** Reverse proxies may not offer the same level of customization as a direct connection between clients and servers, which could limit the functionality of the application.

4. **Additional cost:** Implementing a reverse proxy can require additional hardware and software, which could increase the cost of the overall system.



What is difference between Forward Proxy and Reverse Proxy?

Now that you have basic idea of what is forward and reverse proxy, their location as well their function, now is the time to look into differences to understand them better:

1. Direction

The main difference between a forward proxy and a reverse proxy is the direction of traffic flow. Forward proxy is used to forward traffic from a client to the internet, while a reverse proxy is used to forward traffic from the internet to a web server.

2. Client Access

With a forward proxy, clients have to be explicitly configured to use the proxy server. In contrast, a reverse proxy is transparent to the client, and clients can access the web server directly without needing to know the proxy server's address.

3. Load Balancing

Reverse proxies can distribute incoming requests across multiple servers to balance the load, while forward proxies cannot.

4. Caching

Forward proxies can cache frequently accessed resources to reduce the load on the web server and speed up the response time for subsequent requests. Reverse proxies can also cache resources, but the caching is typically done closer to the client to improve performance.

5. Security

A forward proxy can be used to protect a client's identity by hiding their IP address from the internet. A reverse proxy can be used to protect a server by hiding its identity and exposing a single IP address to the internet.

6. SSL/TLS Termination

A reverse proxy can terminate SSL/TLS connections on behalf of a web server to reduce the load on the server and simplify certificate management. Forward proxies typically do not terminate SSL/TLS connections.

7. Content Filtering

Forward proxies can be used to filter content, block access to specific websites, and enforce access policies. Reverse proxies can also perform content filtering, but this is typically done closer to the client to reduce the load on the web server.

8. Routing

Forward proxies can be used to route traffic to different servers based on predefined rules. Reverse proxies can also perform routing, but this is typically done based on the requested URL and other criteria.

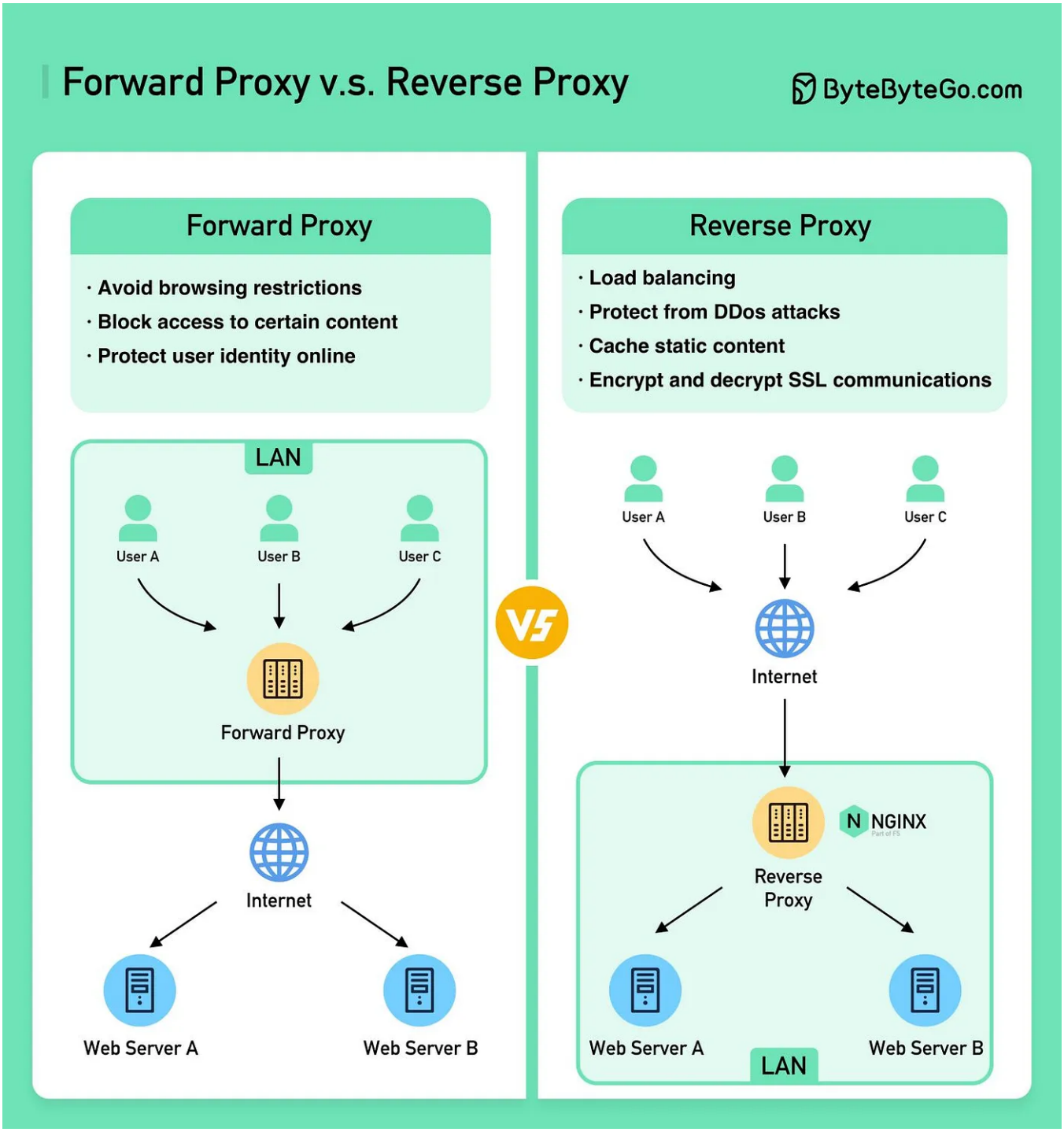
9. Scalability

Reverse proxies can be used to scale web applications horizontally by distributing traffic across multiple servers. Forward proxies do not provide this scalability feature.

10. Network Complexity

Forward proxies are relatively simple to set up and manage, while reverse proxies can be more complex due to their load balancing, SSL/TLS termination, and caching features.

Here is a nice diagram I found online which also highlight the difference between Forward and Reverse Proxy



image_credit — ByteByteGo.com

ByteByteGo is a great place to learn System Design, they also run a newsletter and have a YouTube channel, if you are preparing for System Design interview, you can also check that as a resource. Also, if you want to learn more about reverse proxy and forward proxy you can watch this YouTube video from their channel

Java and Spring Interview Preparation Material

Before any Java and Spring Developer interview, I always read the below two 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**

Conclusion

In conclusion, the **Service Registry pattern is a popular approach in microservices architecture for dynamic service discovery, load balancing, and fault tolerance**. It provides several benefits, such as enabling services to register and un-register themselves at runtime, distributing requests among multiple service instances, and automatically detecting and removing failed services.

However, it also introduces additional complexity, operational overhead, and potential single points of failure. It is important to carefully consider the specific requirements and trade-offs of the system architecture before adopting the Service Registry pattern.

Proper configuration, monitoring, and maintenance of the service registry are also critical to ensure high availability and reliability of the microservices ecosystem.

Overall, the Service Registry pattern can be a powerful tool in building scalable and resilient microservices architectures, but it should be used judiciously and in alignment with the overall system design and requirements.

And, If you like this Microservice article and want to read more such article but you are not a Medium member then I highly recommend you to join Medium and many such articles from many Java and Microservice experts. 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

Other **Microservices articles** you may like:

Top 10 Microservices Problem Solving Questions for 5 to 10 years experienced Developers

From Service Discovery to Security: 10 Problem-Solving Questions to Test the Microservices Skills of Developers with 5...

medium.com

Securing Microservices with OAuth2 and Spring Security

An Overview of Implementing OAuth2 and Spring Security to Enhance Microservices Security

medium.com

Difference between @Controller, @Service, and @Repository Annotations in Spring Framework?

While all three are stereotype annotation in Spring and can be use to represent bean where exactly they are used is the...

medium.com

Microservices | Microservice Architecture | Java | Programming | Spring Boot

Follow

Written by Soma

4.1K 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

Implementation	implementation	an implementation	features
Persistence API	Hibernate provides its own API	Defines a set of standard APIs for ORM	Builds on JPA APIs, adds more functionality
Database Support	Supports various databases through dialects	Depends on the JPA implementation used	Depends on the JPA implementation used
Transaction Management	Provides its own transaction management	Depends on the JPA implementation used	Depends on the JPA implementation used
Query Language	Hibernate Query Language (HQL)	JPQL (Java Persistence Query Language)	JPQL (Java Persistence Query Language)
Caching	Provides first-level and second-level caching	Depends on the JPA implementation used	Depends on the JPA implementation used
Configuration	XML, annotations, or Java-based configuration	XML, annotations, or Java-based configuration	XML, annotations, or Java-based configuration
Integration	Can be used independently or with JPA	Works with any JPA-compliant implementation	Works with any JPA-compliant implementation

Soma in Javarevisited

Difference between Hibernate, JPA, and Spring Data JPA?

Hello folks, if you are preparing for Java Developer interviews then part from preparing Core Java, Spring Boot, and Microservices, you...

· 10 min read · May 26

253 1

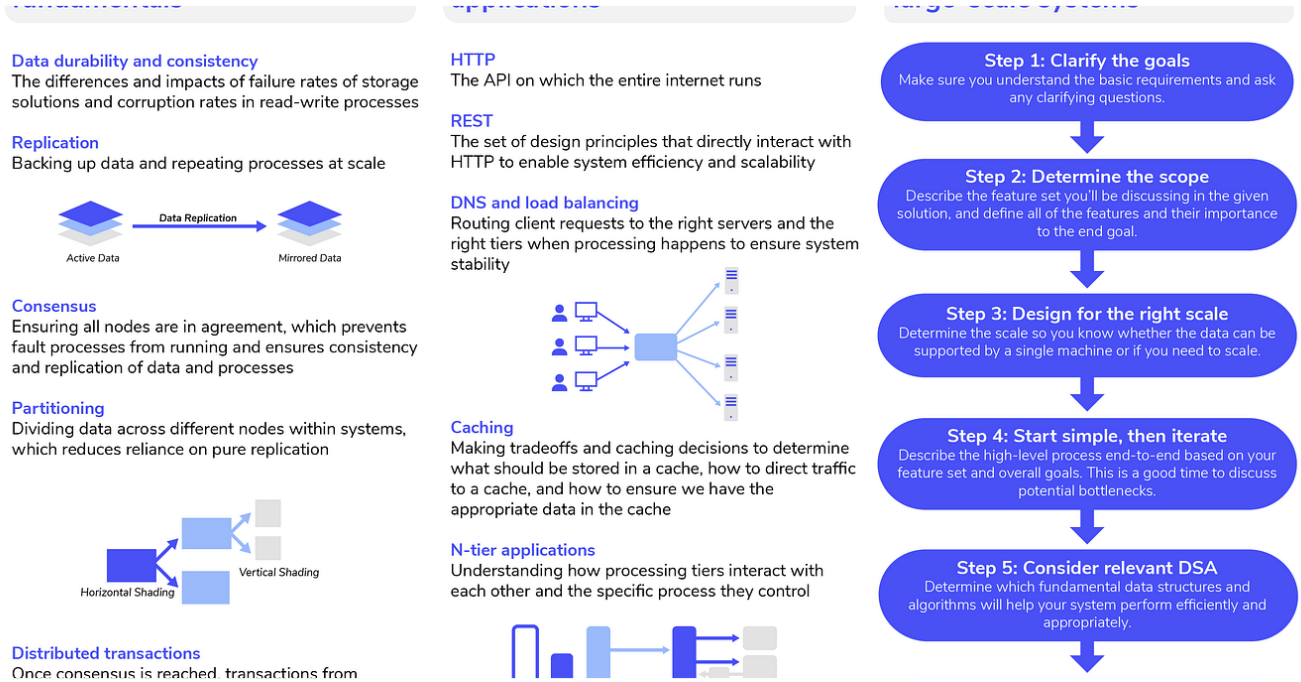
javinpaul in Javarevisited

Top 10 Websites to Learn Python Programming for FREE [2023]

Hello guys, if you are here then let me first congratulate you for making the right decision to learn Python programming language, the...

13 min read · Apr 22, 2020

328 6



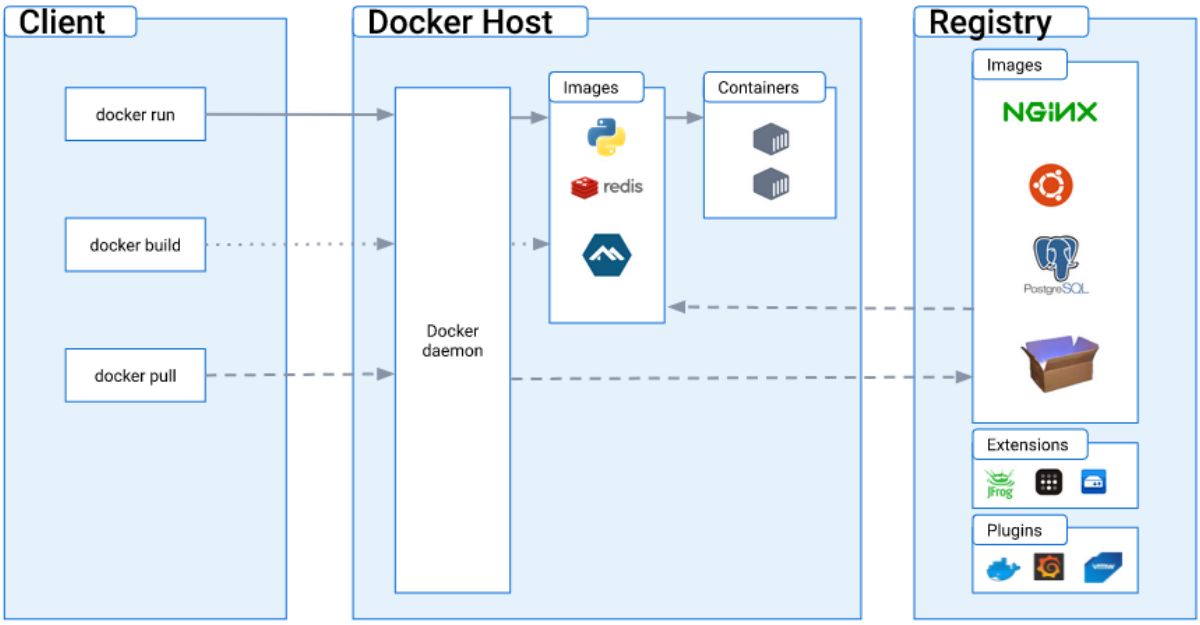
javinpaul in Javarevisited

Top 3 System Design Cheat Sheets for Developers

3 System Design Cheat Sheet you can print and put on your desktop to revise before Tech interviews

6 min read · Jul 19

204 1



Soma in Javarevisited

How Docker works internally? Magic Behind Containerization

Exploring the Inner Workings of Docker: Unveiling the Magic Behind Containerization

· 6 min read · Jun 30

184

See all from Soma

See all from Javarevisited

Recommended from Medium

Rupert Waldron

Create a non-blocking REST Api using Spring @Async and Polling

Get the great asynchronous, non-blocking experience you deserve with Spring's basic Rest Api, polling and @Aysnc annotation.

12 min read · Feb 26

17

Hamza Nassour in Javarevisited

What Happens Internally When You Start A Spring Boot Application(Part1)

ApplicationContext creation/registration , AutoConfiguration .. .

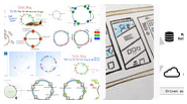
4 min read · Feb 19

93

Lists

It's never too late or early to start something

13 stories · 67 saves

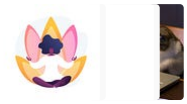


General Coding Knowledge

20 stories · 194 saves

Coding & Development

11 stories · 101 saves



Stories to Help You Grow as a Software Developer

19 stories · 265 saves

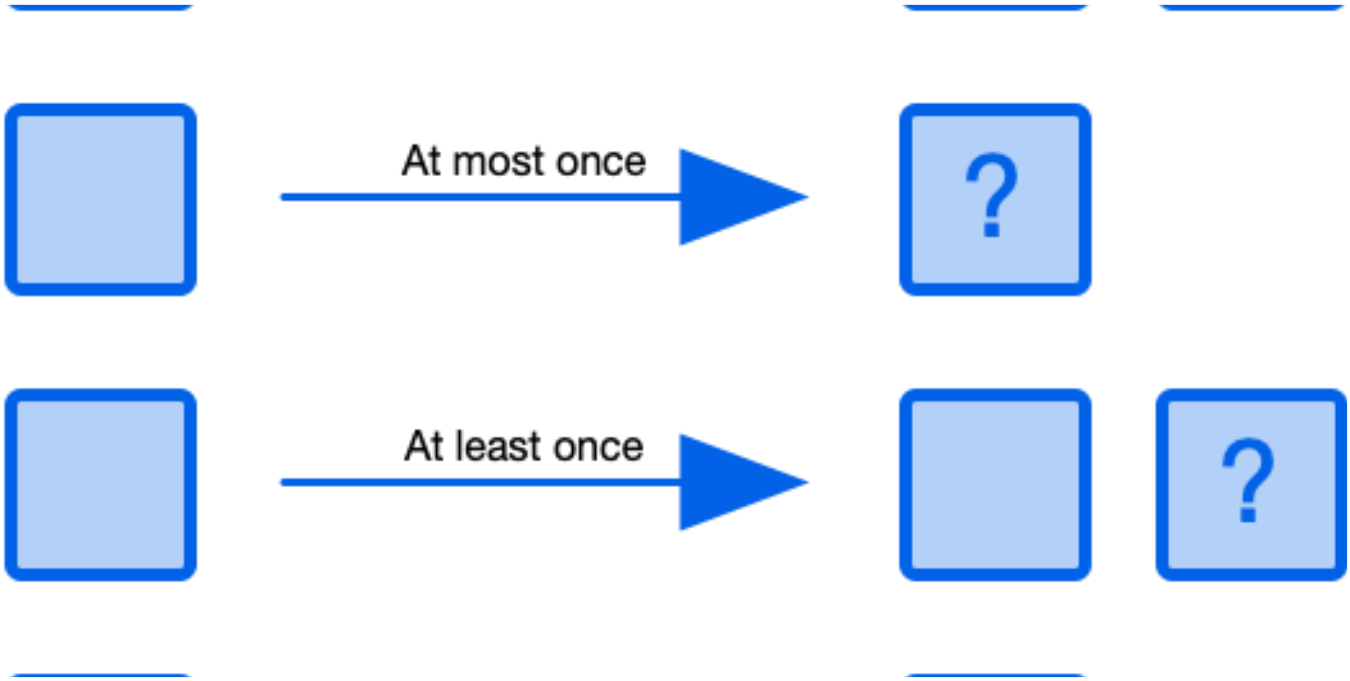
Amit Himani

Distributed Transaction in Spring Boot Microservices

Distributed transactions in microservices refer to transactions that involve multiple microservices, each handling a part of the...

· 5 min read · Feb 17

49 3



Andy Bryant

Processing guarantees in Kafka

Each of the projects I've worked on in the last few years has involved a distributed message system such as AWS SQS, AWS Kinesis and more...

21 min read · Nov 16, 2019

1.91K 5

Sowmya.L.R

Docker—Container era (PART-I)

In earlier days people rarely used web apps. But in this digital era, every single task is done by any particular app. Ex grocery buying...

4 min read · 2 days ago

10

Virendra Oswal

Spring Boot Apache Ignite Caching: Boost Performance & Scale Effortless

Banner

10 min read · Jul 21