Service Discovery in Microservices

Last updated: November 6, 2022

Written by: Simone Cusimano (https://www.baeldung.com/cs/author/simonecusimano)

Software Architecture (https://www.baeldung.com/cs/category/software-architecture)

Microservices (https://www.baeldung.com/cs/tag/microservices)

1. Introduction

Dealing with microservices (/cs/microservices-cross-cutting-concerns), the need arises for a mechanism that allows a service to use another one without knowing its exact location. In this tutorial, we'll explore the concept of Service Discovery.

2. What Is Service Discovery?

Let's imagine several microservices that make up a more or less complex application. These will communicate with each other somehow (e.g. API Rest (/rest-with-spring-series), gRPC (/grpc-introduction)).

A microservices-based application (/spring-microservices-guide) typically runs in virtualized or containerized environments. The number of instances of a service and its locations changes dynamically. We need to know where these instances are and their names to allow requests to arrive at the target microservice. This is where tactics such as Service Discovery come into play.

The Service Discovery mechanism helps us know where each instance is located. In this way, a Service Discovery component acts as a registry in which the addresses of all instances are tracked. The instances have dynamically assigned network paths. Consequently, if a client wants to make a request to a service, it must use a Service Discovery mechanism.

3. The Need for Service Discovery

A microservice needs to know the location (IP address and port) of every service it communicates with. If we don't employ a Service Discovery mechanism, service locations become coupled, leading to a system that's difficult to maintain. We could wire the locations or inject them via configuration in a traditional application, but it isn't recommended in a modern cloud-based application of this kind.

Dynamically determining the location of an application service isn't a trivial matter. Things become more complicated when we consider an environment where we're constantly destroying and distributing new instances of services. This may well be the case for a cloud-based application that's

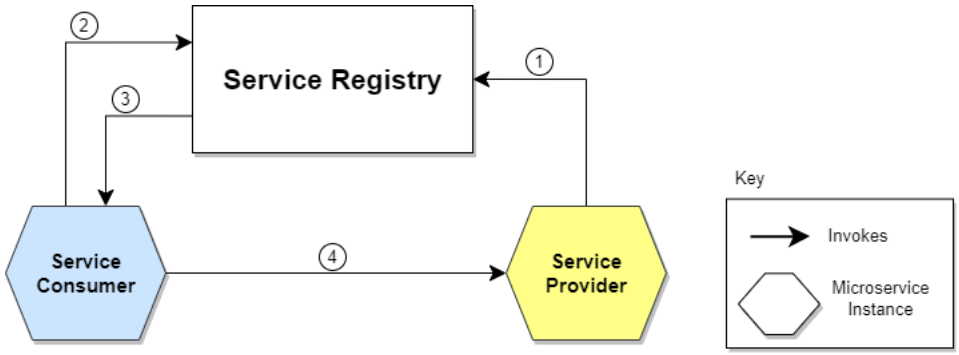
continuously changing due to horizontal autoscaling to meet peak loads, or the release of a new version. **Hence, the need for a Service Discovery mechanism.**

4. How Does Service Discovery Works?

Service Discovery handles things in two parts. First, it provides a mechanism for an instance to register and say, "*I'm here!*" Second, it provides a way to find the service once it has registered.

Let's clarify the concept we've discussed so far with an example: a Service Consumer and a Service Provider (a service exposing REST API (/cs/rest-architecture)). The Service Consumer needs the Service Provider to read and write data.

The following diagram shows the communication flow:



Let's describe the steps illustrated in the diagram:

1. The location of the Service Provider is sent to the Service Registry (a database containing the locations of all available service instances).
2. The Service Consumer asks the Service Discovery Server for the location of the Service Provider.
3. The location of the Service Provider is searched by the Service Registry in its internal database and returned to the Service Consumer.
4. The Service Consumer can now make direct requests to the Service Provider.

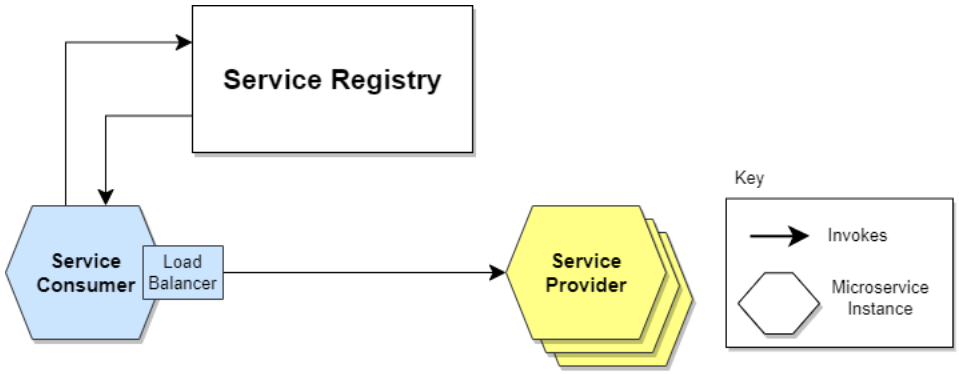
There are two main Service Discovery patterns: Client-Side Discovery and Server-Side Discovery. Let's clarify them.

5. Service Discovery Implementations

5.1. Client-Side Service Discovery

When using Client-Side Discovery, **the Service Consumer is responsible for determining the network locations of available service instances and load balancing requests between them.** The client queries the Service Register. Then the client uses a load-balancing algorithm to choose one of the available service instances and performs a request.

The following diagram shows the pattern just described:



(/wp-content/uploads/sites/4/2022/01/Service-Discovery-Client-Side.png)
Giving responsibility for client-side load balancing is both a burden and an advantage. **It's an advantage because it saves an extra hop that we would've had with a dedicated load balancer. It's a disadvantage because the Service Consumer must implement the load balancing logic.**

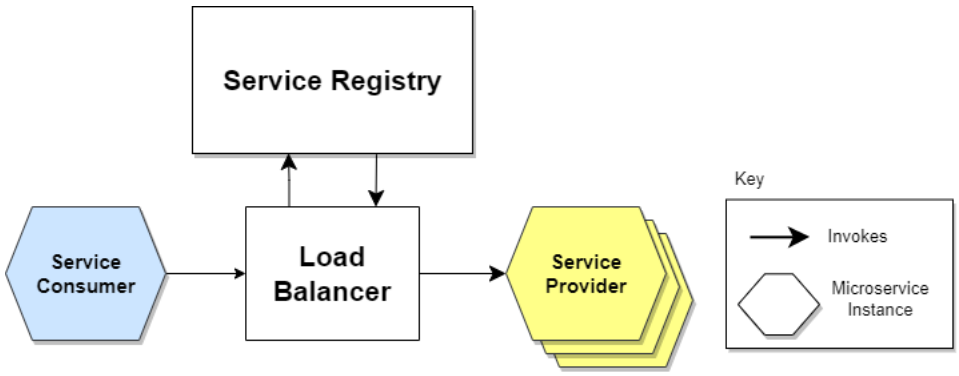
We can also point out that the Service Consumer and the Service Registry are quite coupled. This means that Client-Side Discovery logic must be implemented for each programming language and framework used by the Service Consumers.

Now that we've clarified Client-Side Discovery, let's take a look at Server-Side Discovery.

5.2. Server-Side Service Discovery

The alternate approach to Service Discovery is the Server-Side Discovery model, which uses an intermediary that acts as a Load Balancer (/zuul-load-balancing). The client makes a request to a service via a load balancer that acts as an orchestrator. The load balancer queries the Service Registry and routes each request to an available service instance.

The following diagram shows how communication takes place:



(/wp-content/uploads/sites/4/2022/01/Service-Discovery-Server-Side.png)
In this approach, a dedicated actor, the Load Balancer, does the job of load balancing. This is the main advantage of this approach. Indeed, **creating this level of abstraction makes the Service Consumer lighter, as it doesn't have to deal with the lookup procedure.** As a matter of fact, there's no need to implement the discovery logic separately for each language and framework that the Service Consumer uses.

On the other hand, we must set up and manage the Load Balancer, unless it's already provided in the deployment environment.

Now that we've delved into the different approaches to the discovery mechanisms, let's move on to registration mechanisms.

6. What Is Service Registry?

So far, we've assumed that the Service Registry already knew the locations of each microservice. But how do this registration and de-registration operation take place?

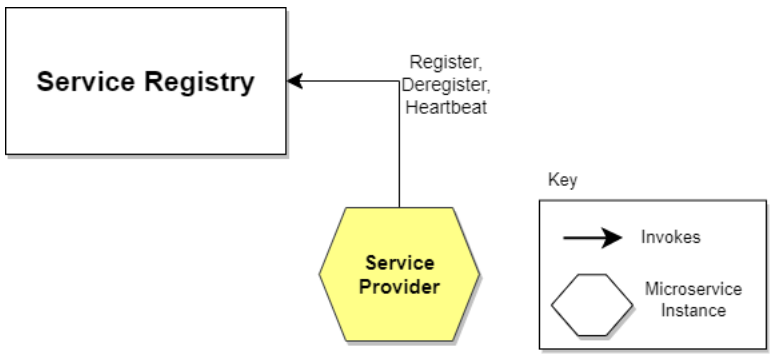
The Service Register is a crucial part of service identification. **It's a database containing the network locations of service instances.** A Service Registry must be highly available and up-to-date. Clients can cache the network paths obtained from the Service Registry; however, this information eventually becomes obsolete, and clients won't reach the service instances. Consequently, a Service Registry consists of a cluster of servers that use a replication protocol to maintain consistency.

Let's look at it in more detail, describing two possible approaches.

7. Service Registration Options

7.1. Self-Registration

When using the self-registration model, a service instance is responsible for registering and de-registering itself in the Service Registry. In addition, if necessary, a service instance sends heartbeat (https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html) requests to keep its registration alive. The following diagram shows the structure of this pattern:

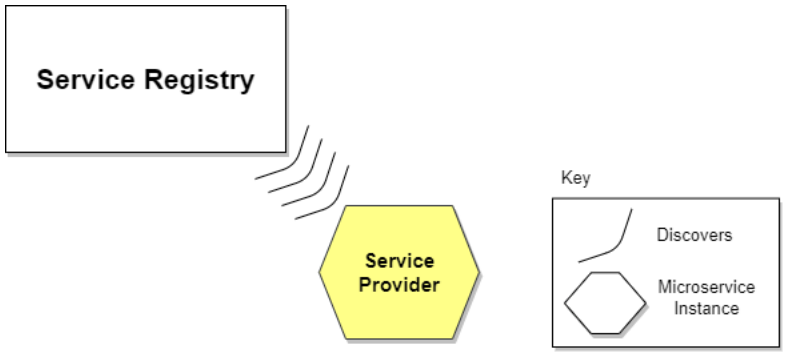


(/wp-content/uploads/sites/4/2022/01/Service-Discovery-Self-Registration.png)
The self-registration model has several pros and cons. One advantage is that it's relatively simple and doesn't require other system components as intermediaries. However, a significant disadvantage is that it couples service instances to the Service Registry, which means we must implement the registration code in each language and framework used.

An alternate approach, which decouples services from the Service Registry, is the third-party registration scheme.

7.2. Third-party Registration

When using the third-party registration model, the service instances aren't responsible for registration in the Service Registry. Instead, another system component known as the Service Register is responsible for registration. **The Service Register keeps track of changes to running instances by polling the deployment environment or subscribing to events.** When it detects a newly available service instance, it records it in its database. The Service Registry also de-registers terminated service instances. The following diagram illustrates this:



(/wp-content/uploads/sites/4/2022/01/Service-Discovery-3rd-Registration.png)
Like self-registration, the third-party registration scheme also has various pros and cons. One of the main advantages is that services are decoupled from the Service Registry. There's no need to implement service registration logic for each programming language and framework. Instead, the registration of service instances is managed centrally within a dedicated service.

One disadvantage of this model is that, unless it's embedded in the deployment environment, it's yet another highly available system component that needs to be set up and managed.

8. Conclusion

This article outlines how a microservice application contains several service instances running with dynamic changes. These instances have dynamic network locations and need a way to communicate or locate. Therefore, a client needs a mechanism, such as Service Discovery, to make a request.

Service Discovery helps by providing a database of available service instances so that services can be discovered, registered, and de-registered based on usage.

CATEGORIES

ALGORITHMS(/cs/category/algorithms)

ARTIFICIAL INTELLIGENCE(/cs/category/ai)

CORE CONCEPTS(/cs/category/core-concepts)

DATA STRUCTURES(/cs/category/data-structures)

GRAPH THEORY(/cs/category/graph-theory)

LATEX(/cs/category/latex)

NETWORKING(/cs/category/latex)

SECURITY(/cs/category/security)

SERIES

SITES

BAELDUNG(/)

LINUX(/linux)

SCALA(/scala)

KOTLIN(/kotlin)

ABOUT

ABOUT BAELDUNG(/about)

THE FULL ARCHIVE(/cs/full_archive)

EDITORS(/editors)

TERMS OF SERVICE(/terms-of-service) | PRIVACY POLICY(/privacy-policy) | COMPANY INFO(/company-info) | CONTACT(/contact)