Level 400

This document focuses on Microservice patterns for Service Registry

Problem Statement:

How do clients of a service (in the case of [Client-side discovery](https://microservices.io/patterns/client-side-discovery.html)) and/or routers (in the case of [Server-side discovery](https://microservices.io/patterns/server-side-discovery.html)) know about the available instances of a service?

* Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port)
* The number of services instances and their locations changes dynamically. Virtual machines and containers are usually assigned a dynamic IP address. An EC2 Autoscaling Group, for example, adjusts the number of instances based on load.

Solution:

Implement a service registry, which is a database of services, their instances and their locations. Service instances are registered with the service registry on startup and deregistered on shutdown. Client of the service and/or routers query the service registry to find the available instances of a service. A service registry might invoke a service instance’s [health check API](https://microservices.io/patterns/observability/health-check-api.html) to verify that it is able to handle requests

Examples:

[Netflix Eureka](https://github.com/Netflix/eureka) is good example of a service registry. It provides a REST API for registering and querying service instances. A service instance registers its network location using a POST request. Every 30 seconds it must refresh its registration using a PUT request. A registration is removed by either using an HTTP DELETE request or by the instance registration timing out. As you might expect, a client can retrieve the registered service instances by using an HTTP GET request.

Other examples of service registries include:

* [etcd](https://github.com/coreos/etcd) – A highly available, distributed, consistent, key‑value store that is used for shared configuration and service discovery. Two notable projects that use etcd are Kubernetes and [Cloud Foundry](http://pivotal.io/platform).
* [consul](https://www.consul.io/) – A tool for discovering and configuring services. It provides an API that allows clients to register and discover services. Consul can perform health checks to determine service availability.
* [Apache Zookeeper](http://zookeeper.apache.org/) – A widely used, high‑performance coordination service for distributed applications. Apache Zookeeper was originally a subproject of Hadoop but is now a top‑level project.

some systems such as Kubernetes, Marathon, and AWS do not have an explicit service registry. Instead, the service registry is just a built‑in part of the infrastructure.

You need to decide how service instances are registered with the service registry. There are two options:

* [Self-registration pattern](https://microservices.io/patterns/self-registration.html) - service instances register themselves.
* [3rd party registration pattern](https://microservices.io/patterns/3rd-party-registration.html) - a 3rd party registers the service instances with the service registry.

Self-Registration:

When using the self-registration pattern, a service instance is responsible for registering and deregistering itself with the service registry. Also, if required, a service instance can send heartbeat requests to prevent its registration from expiring. If a service instance gracefully goes down, it should deregister itself. However, if it terminates abruptly, the service registry should deregister the instance in absence of heartbeat.

A good example of this approach is the [Netflix OSS Eureka client](https://github.com/Netflix/eureka). The Eureka client handles all aspects of service instance registration and de-registration. The [Spring Cloud project](http://projects.spring.io/spring-cloud/), which implements various patterns including service discovery, makes it easy to automatically register a service instance with Eureka. You simply annotate your Java Configuration class with @EnableEurekaClient annotation.

Reference Implementation: <https://github.com/spring-guides/gs-service-registration-and-discovery>

### Third-Party Registration:

When using the third-party registration pattern, service instances aren’t responsible for registering themselves with the service registry. Instead, another system component known as the service registrar handles the registration. The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events. When it notices a newly available service instance it registers the instance with the service registry. The service registrar also deregisters terminated service instances. The following diagram shows the structure of this pattern.

One example of a service registrar is the open source [Registrator](https://github.com/gliderlabs/registrator) project. It automatically registers and deregisters service instances that are deployed as Docker containers. Registrar supports several service registries, including etcd and Consul.

The service registrar is a built‑in component of deployment environments. The EC2 instances created by an Autoscaling Group can be automatically registered with an ELB. Kubernetes services are automatically registered and made available for discovery.

Istio Service Registry:

Using Service Entry, you can add or remove a service from the service registry of Istio. Once defined, the service names become well known to Istio, and they can then be mapped to other Istio configurations. By default, Istio allows services to send requests to hostnames not present in the service registry.

Creating a service entry does not create a DNS record in Kubernetes, so you won't be able to perform a DNS lookup from the application to fetch the resolved IP address. A CoreDNS Istio plugin can generate DNS records from service entry records, enabling Istio to populate DNS records outside Istio. With service entry records, you can implement traffic management patterns such as timeouts on external services.

Here’s an example of a ServiceEntry resource that declares an external API (api.external-svc.com) we can access over HTTPS.

apiVersion: networking.istio.io/v1alpha3

kind: ServiceEntry

metadata:

name: external-svc

spec:

hosts:

- api.external-svc.com

ports:

- number: 443

name: https

protocol: TLS

resolution: DNS

location: MESH\_EXTERNAL

specification will create a record in the Istio service registry.

The hosts field can contain multiple external APIs, and in that case, the Envoy sidecar will do the checks based on the hierarchy below. If Envoy cannot inspect any of the items, it moves to the next item in the order.

* HTTP Authority header (in HTTP/2) and Host header in HTTP/1.1),
* SNI,
* IP address and port

Envoy will either blindly forward the request or drop it if none of the above values can be inspected, depending on the Istio installation configuration

With MESH\_INTERNAL setting in the location field, we say that this service is part of the mesh. This value is typically used in cases when we include workloads on unmanaged infrastructure (VMs). The other value for this field, MESH\_EXTERNAL, is used for external services consumed through APIs. The MESH\_INTERNAL and MESH\_EXTERNAL settings control how sidecars in the mesh attempt to communicate with the workload, including whether they’ll use Istio mutual TLS by default.

Let's create a destination rule to ensure that other services communicate with our service over TLS.

spec:

host: api.external-svc.com

trafficPolicy:

loadBalancer:

consistentHash:

httpCookie:

name: user

ttl: 0s

the destination rule will use a cookie to route consecutive requests to the same instance of the **Independent** service. If the cookie does not already exist, it will create one and return it with the response. The client should pass the same cookie in all consecutive requests to reach the same destination service.