Service Discovery & Registry

Nikola Hristovski 161101

Oliver Dimitriov 161535

Anton Kahwaji 161555

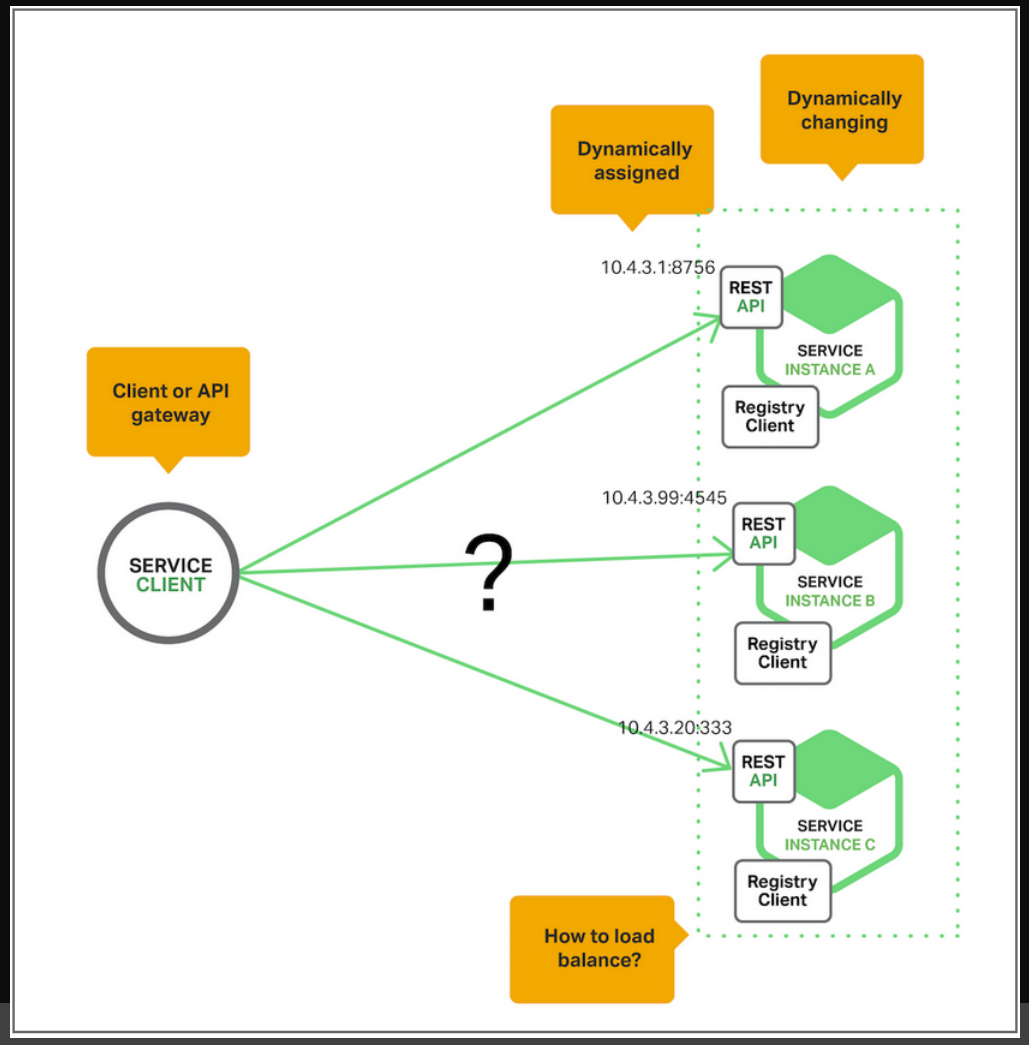
Service discovery and registry

- Automatic detection of devices and services offered on a computer network
- It is a database for up-to-date lists of microservices
- Two types of service discovery: server-side discovery or client-side discovery

Traditional applications

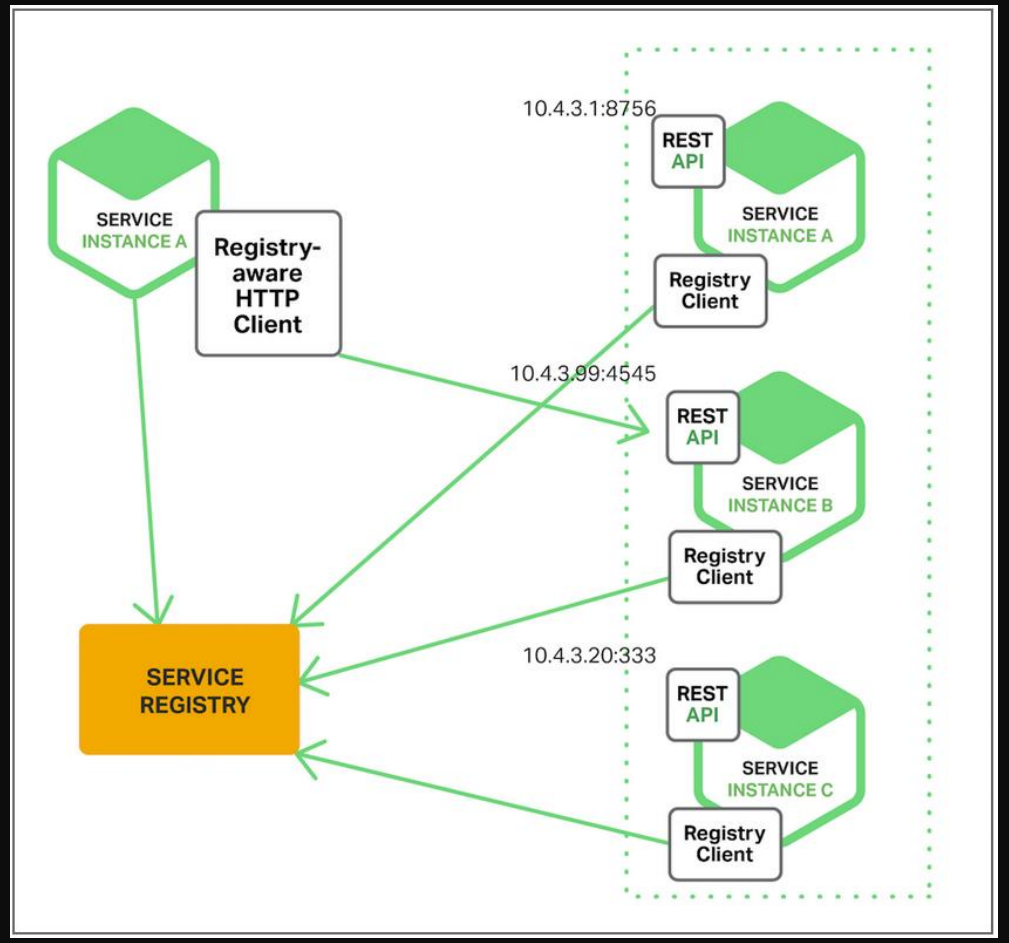
- The network locations of service instances are relatively static.
- They can be read from the configuration file in each service.
- In modern, cloud-based microservices application this is much more difficult problem.

Cloud-based microservice application



Client-side Discovery

- The client is responsible for determining the network locations of available service instances.
- The client is load balancing request across them.

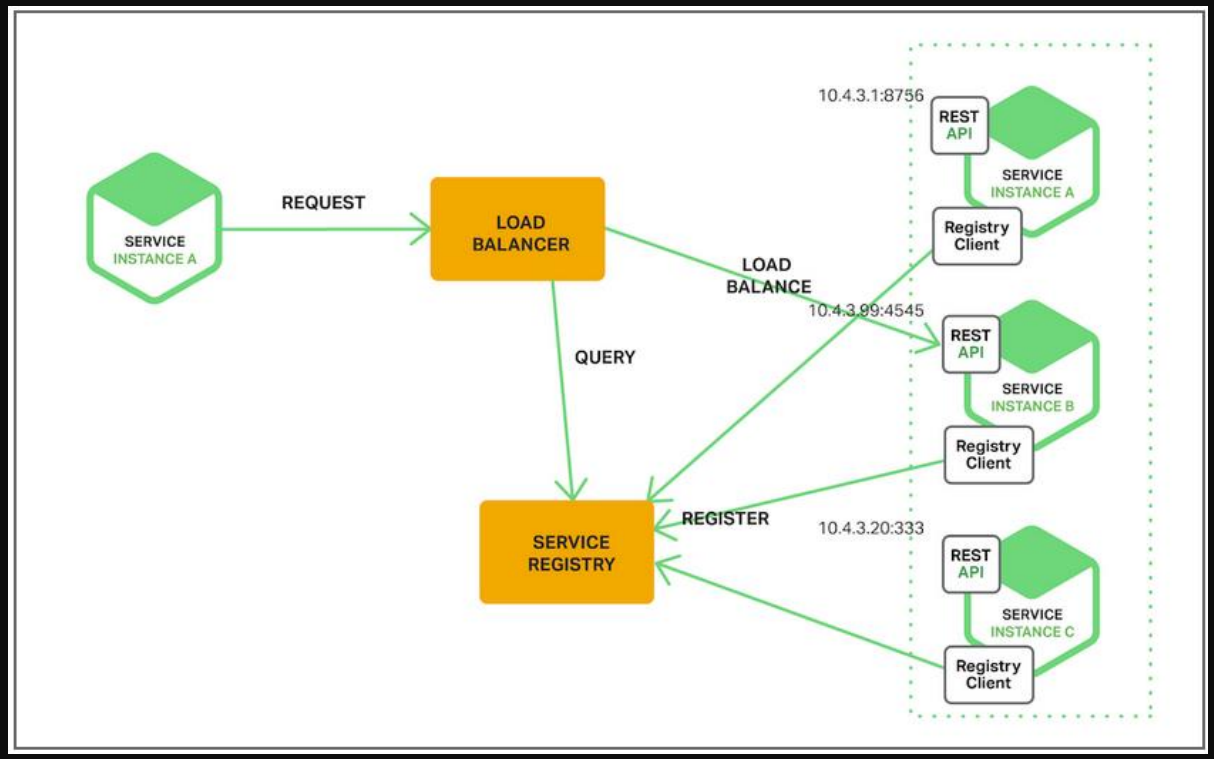


Client-side Discovery Pattern

- When the service registry starts up, the network location of service instance is registered.
- When the service registry terminates, the information about all registered service instances is removed.
- The service registry periodically checks the health for each registered service.

Server-side Discovery

- The client makes request to a service using load balancer.
- The load balancer queries the service registry and routes each request to an available service instance.
- Service instances are registered and deregistered with the service registry.

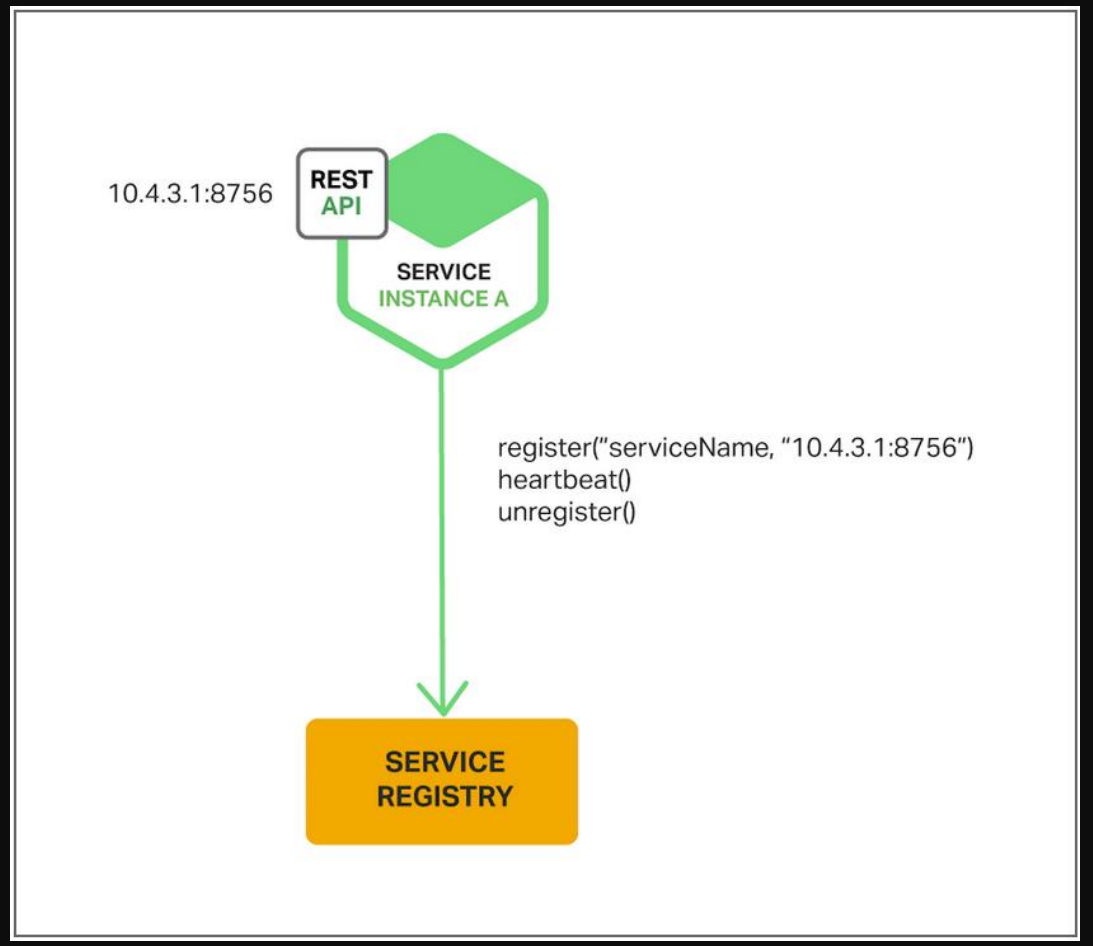


Service registry

- Key part of service discovery
- Database containing the network locations of service instances
- Needs to be highly available and up to date
- Consists of a cluster of servers that use a replication protocol to maintain consistency

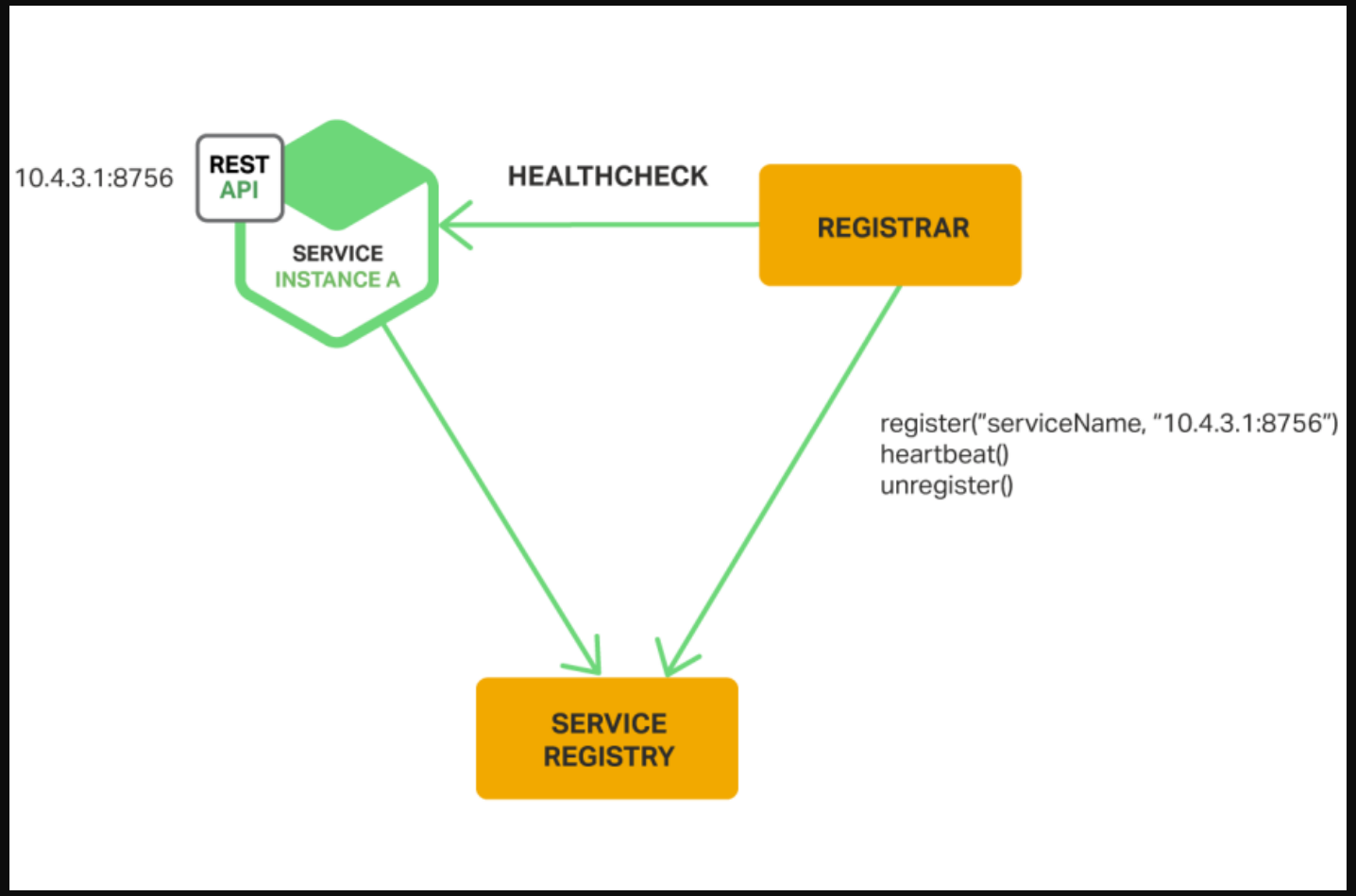
Self-Registration Pattern

- A service instance is responsible for registering and deregistering itself with the service registry.
- Service instance sends heartbeat requests to prevent its registration from expiring.



Third-Party Registration Pattern

- Another system, service registrar, handles the registration.
- The service registrar tracks changes by either polling the deployment environment or subscribing to events.
- Service registrar deregisters terminated service instances



Netflix OSS services

- Netflix OSS provides great example of the client-side discovery pattern.
- Netflix Eureka is service registry, providing REST API for managing service-instance registration.
- Netflix Ribbon is an IPC client that load balance requests across the available service instances.

Service registry services

- Registrator – open source project, that automatically registers and deregisters service instances that are deployed as Docker containers.
- Netflix OSS Prana – primarily intended for services written in non-JVM languages.

Netflix Eureka

- Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

- Open source:
  - https://github.com/Netflix/eureka

Components

- Eureka Server

- Eureka Client, Java-based client component
  - makes interactions with the service much easier
  - has a built-in load balancer that does basic round-robin load balancing

Why

- Very easy to use for JVM Applications (Spring Cloud)
- Netflix Prana - a “side car” application that runs along side a non-JVM application and registers the application with Eureka
- Netflix

How (Server)

- Maven dependency

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

- Configuration

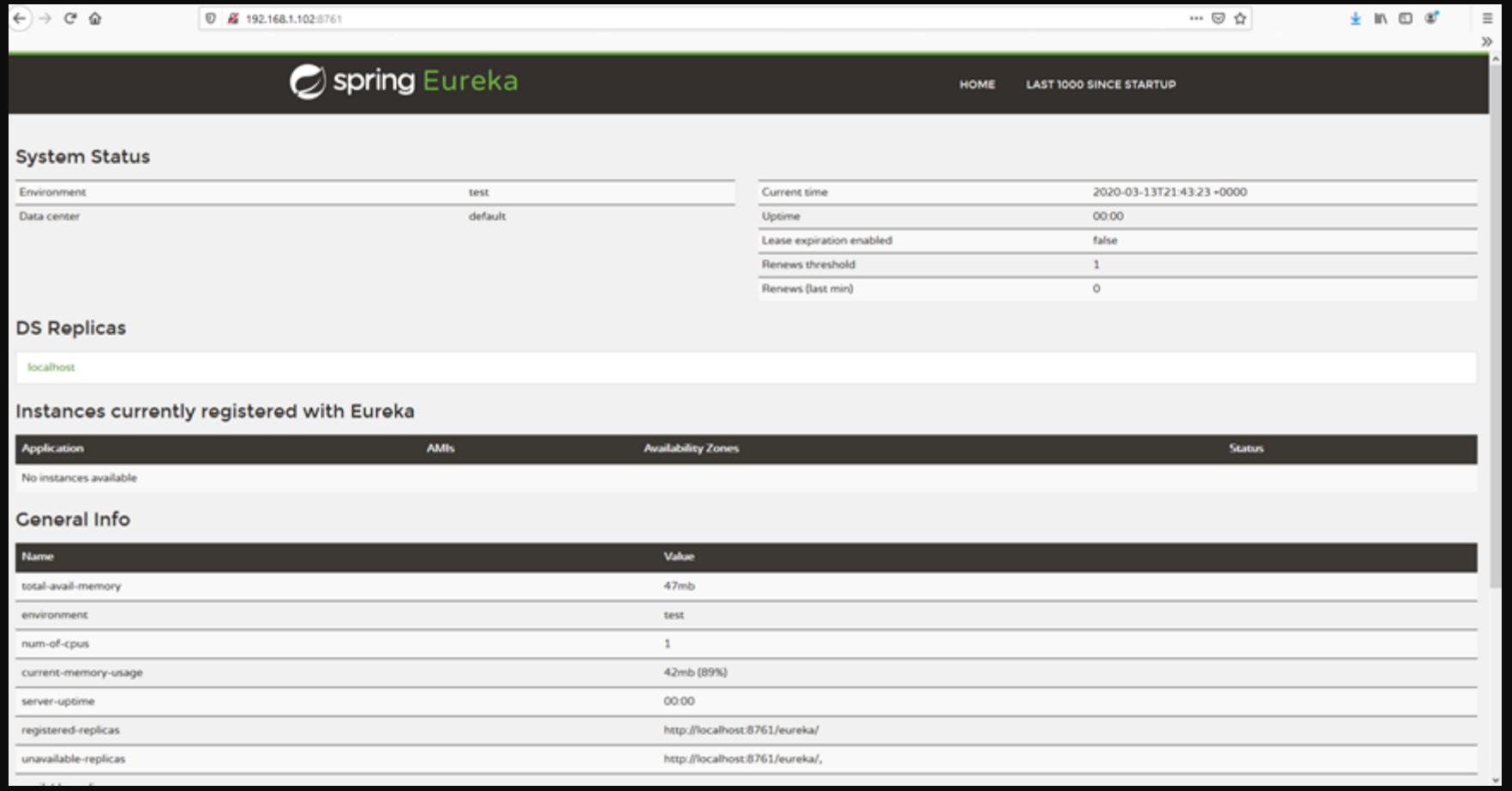
```
spring.application.name=eureka-server
server.port=8761

# eureka by default will register itself as a client. So, we need to set it
to false.
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

How (Server) (Cont.)

- Application

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class ServiceRegistryApplication {

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

Eureka dashboard



Contoso Suites

How (Client)

- Maven dependency

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- Configuration

```
spring.application.name=auth-service
server.port=9100
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

How (Client) (Cont.)

- Application

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class AuthApplication {

    public static void main(String[] args) {
        SpringApplication.run(AuthApplication.class, args);
    }

}
```

Dashboard after registrations

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
AUTH-SERVICE	**n/a** (1)	(1)	**UP** (1) - kube-slave1:auth-service:9100
ZUUL-SERVER	**n/a** (1)	(1)	**UP** (1) - kube-slave1:zuul-server:8762

Application	AMIs	Availability Zones	Status
AUTH-SERVICE	**n/a** (2)	(2)	**UP** (2) - 192.168.56.1:auth-service:9101 , 192.168.56.1:auth-service:9100

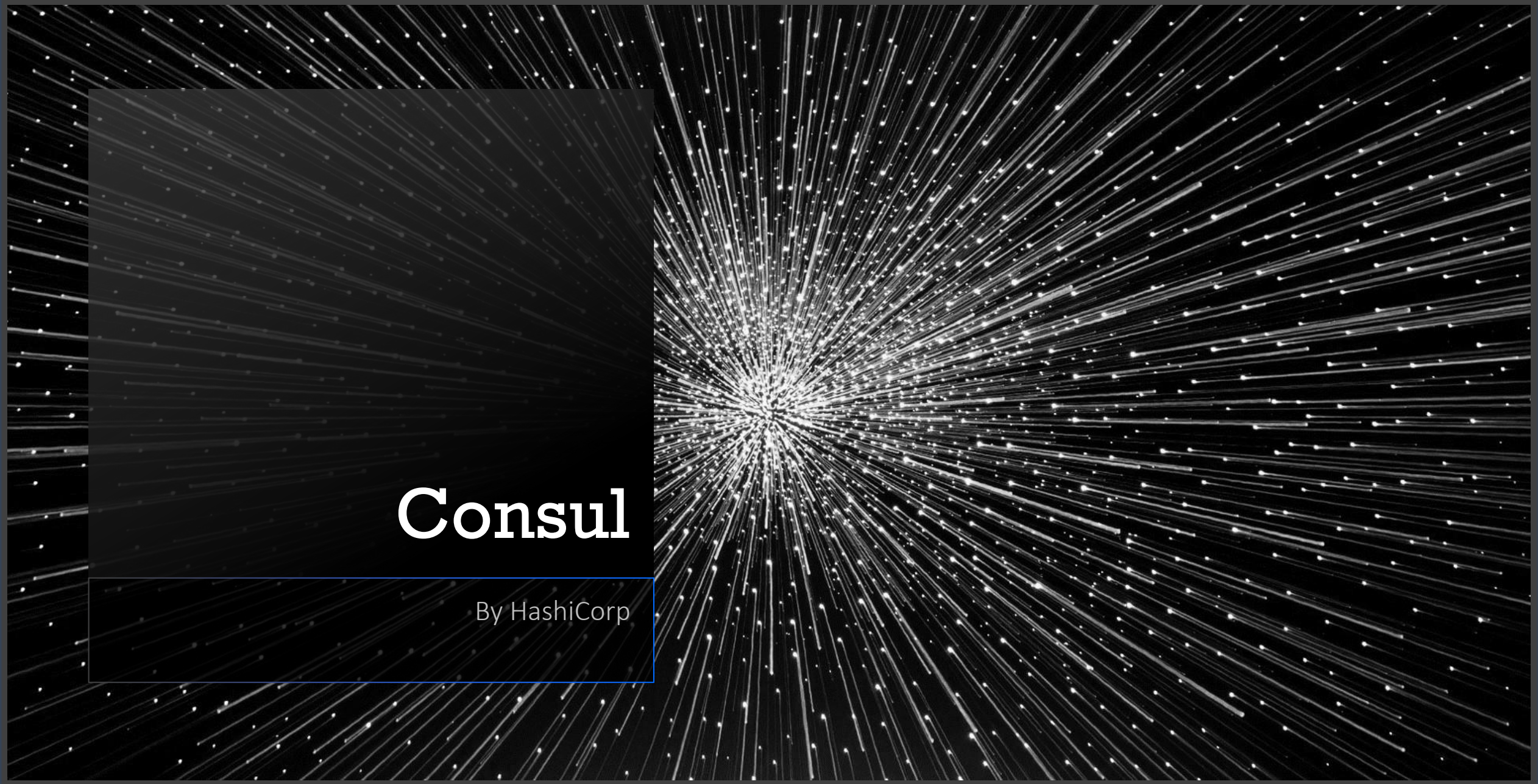
Contoso
Suites

Eureka client for discovering services

```
private static final String SERVICE = "users-service";
private final EurekaClient discoveryClient;
private final RestTemplate restTemplate;

@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

    try {
        InstanceInfo instanceInfo = discoveryClient.getNextServerFromEureka(SERVICE, false);

        String url = instanceInfo.getHomePageUrl() + username;

        ApplicationUser applicationUser = restTemplate.getForObject(url, ApplicationUser.class);
    }

    // TODO: Your code goes here …
}
```

Service registry tools

- Etcd – highly available, distributed, consistent, key-value store that is used for shared configuration and service discovery
- Consul – tool for discovering and configuring services
- Apache Zookeeper – high-performance coordination service for distributed applications.

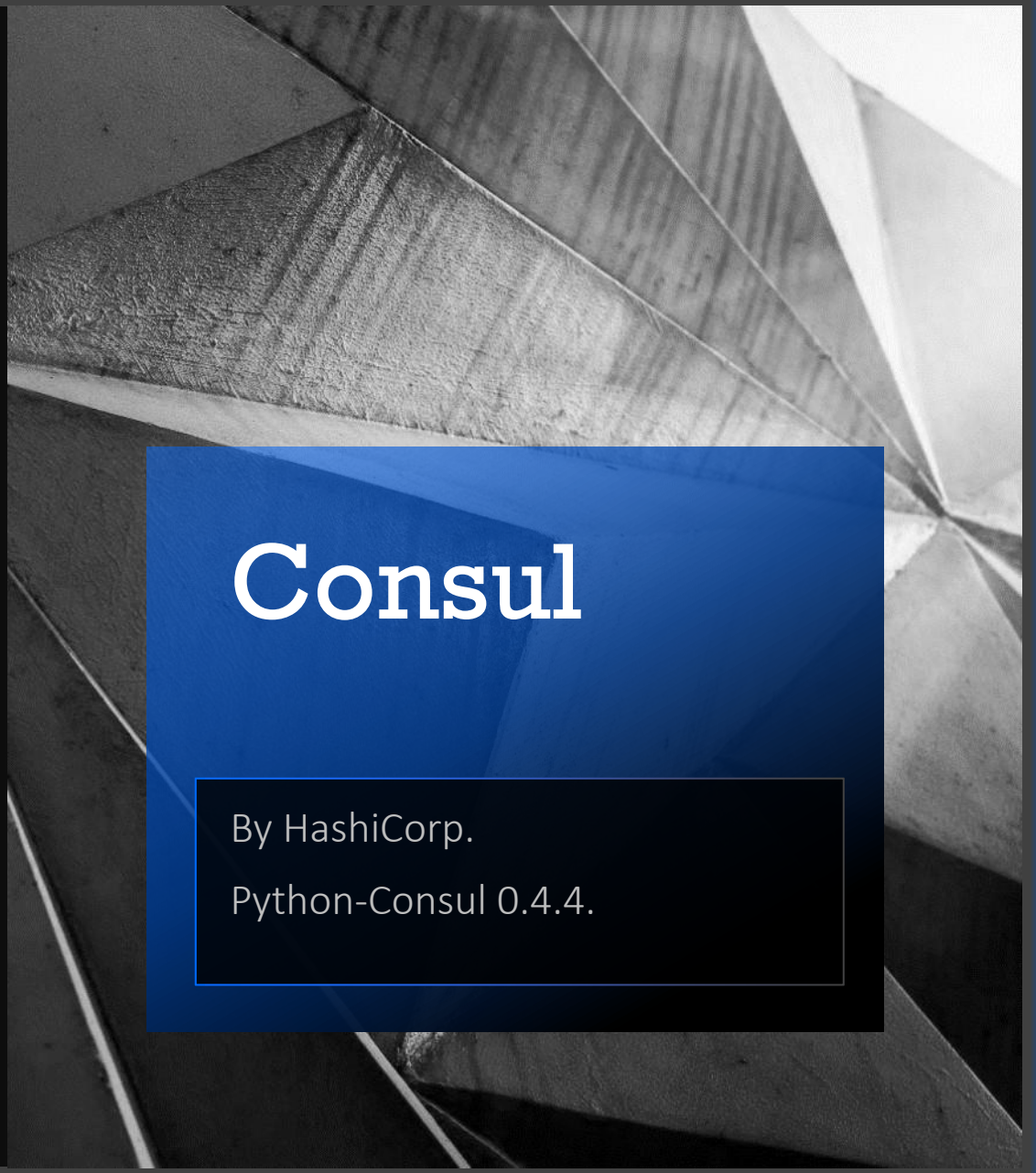

Consul
By HashiCorp

Service networking tool that allows you to discover services and secure network traffic.

- Offers a lot more than just service discovery, like:
  - Key Value Store
  - Health Checks
  - ACL Tokens
  - Sessions (Distributed Locks)

- To register a service in python-consul it is as simple as:

```
from consul import Consul

consul = Consul(host="3.17.67.170", port=8500)

consul.agent.service.register(“gary", service_id="gary",
                              address='localhost', port=8020)
```



- To find info about a service it is as simple as:

```
from consul import Consul

consul = Consul(host="3.17.67.170", port=8500)

service_list = consul.agent.services()

service_info = service_list[“gary"]

address, port = service_info['Address'], service_info['Port']
```

- Which can then be used in the following way:

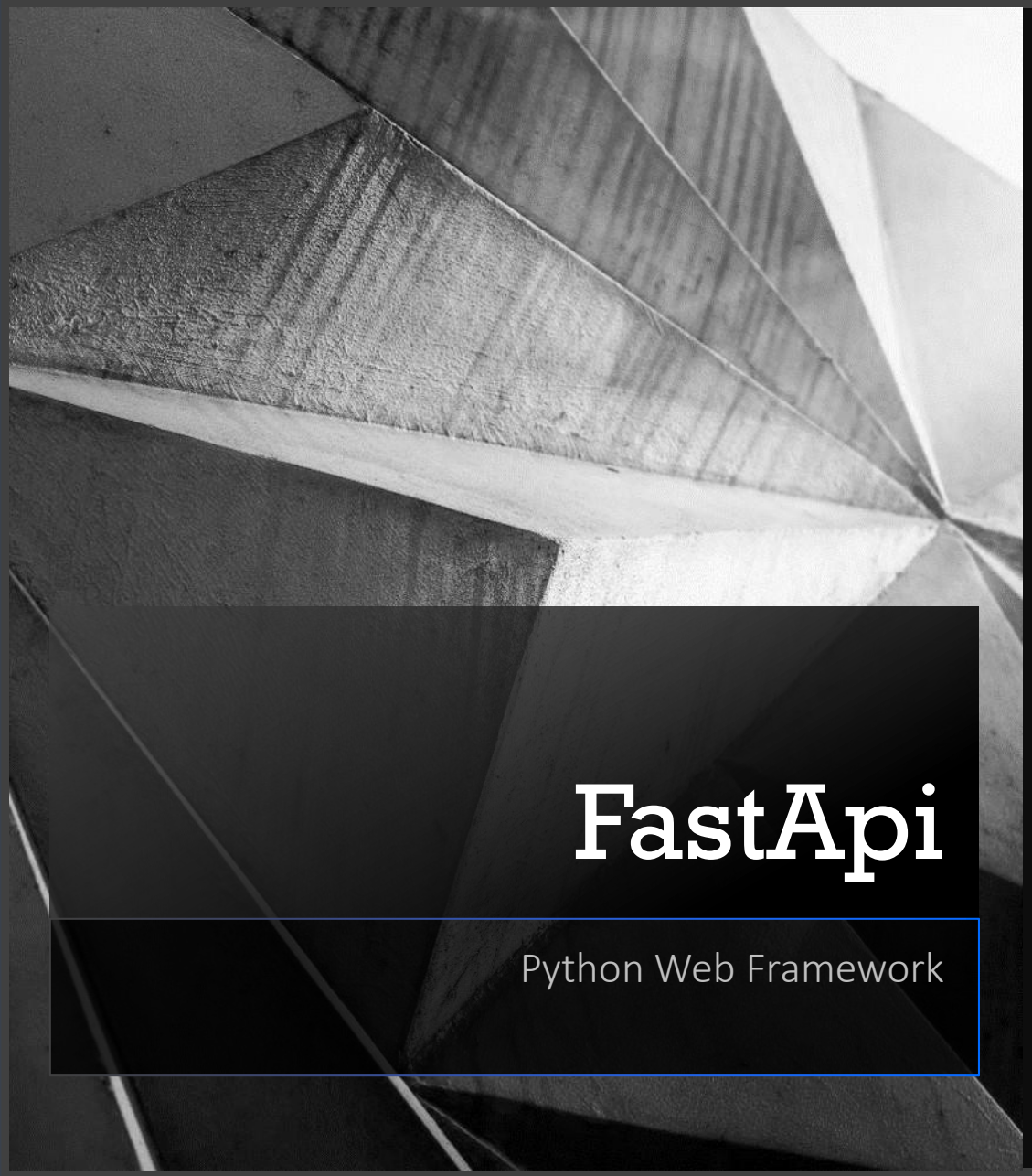
```
import requests

response = requests.get(f"http://{address}:{port}/gary")
```

Consul

By HashiCorp.

Python-Consul 0.4.4.


Unrelated service stuff
FastApi, Docker containers, AWS EC2 hosting



FastApi

Python Web Framework

Modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

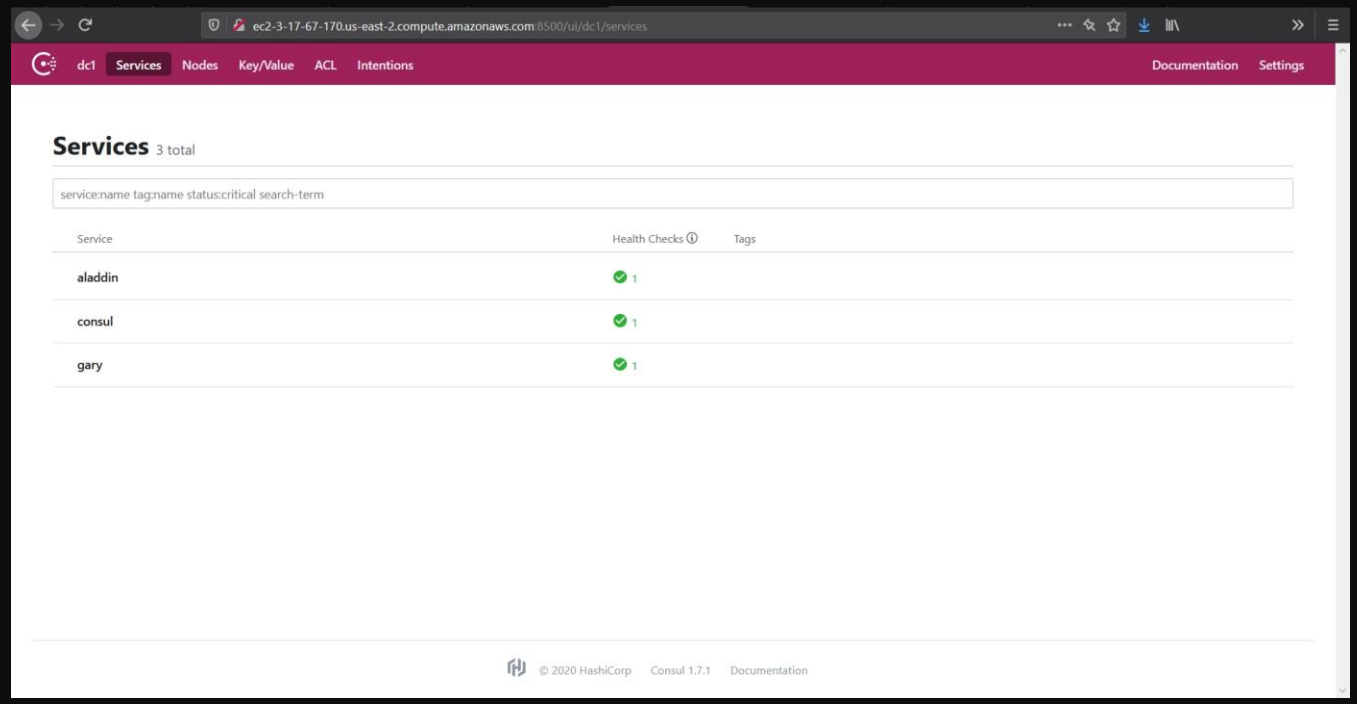
- A few simple lines of code are enough to create an API:

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/gary")
def gary():
    return {"Gary": "meow"}
```

Docker & AWS

Containerized services and hosting on EC2

- To host the simple services
- Made them into docker images
- Built them into containers
- Launch docker on an EC2 instance
- Check if everything is working using the Consul UI Dashboard




Thank You