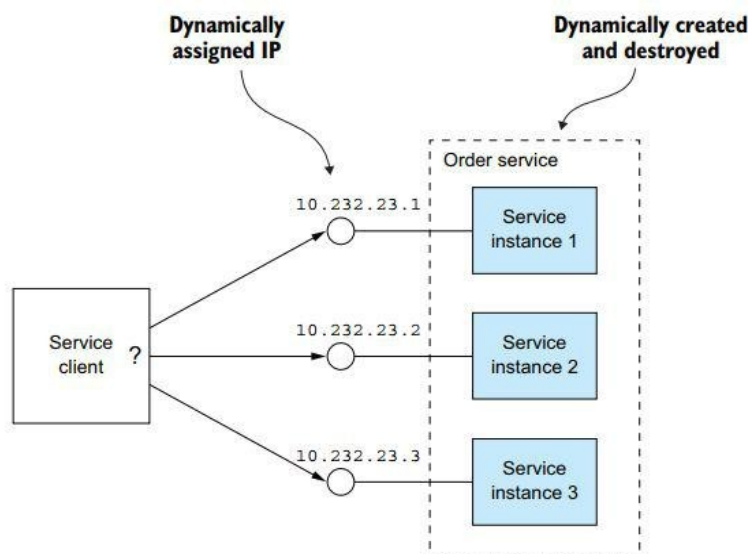


Mikroservisi part 1

1. Service Discovery

Mikroservisna arhitektura je karakteristična po tome što je sačinjena od skupa nezavisnih servisa nastalih u procesu dekomponovanja sistema na servise po nekom šablonu (poslovne funkcionalnosti, poddomeni itd.). Za razliku od monolitne arhitekture gde je skaliranje moguće vršiti jednostavno, odnosno pokretanjem dodatnih instanci monolita, kod mikroservisa mogućnosti skaliranja uzimaju još širi obim. Broj pokrenutih servisa može biti izuzetno velik i dinamika dodavanja i uklanjanja instanci je na višem nivou, što dodatno produbljuje problem koji se odnosi na određivanje lokacije servisa. Primera radi, ako želimo da pošaljemo neki zahtev nekom REST servisu, moramo da znamo njegovu IP adresu i port. Ako imamo 5 mikroservisa, pri čemu je svaki pokrenut u 5 instanci, moramo znati 25 adresa. Prvi način koji je uglavnom moguć, ali zbog dinamike i prirode arhitekture nije poželjan jeste da imamo neki konfiguracioni fajl u kome će biti zapisane konkretne lokacije i koji ćemo periodično održavati. Zbog kompleksnosti današnjih sistema, održavati ovako nešto je jako teško.

Naravno, rešenje bi bio neki vid automatizacije registrovanja lokacije servisa. Umesto da manuelno registrujemo servise, odnosno njihove lokacije, oni će se sami (ili će ih neko drugi) javiti nekom centralnom registru koji će zabeležiti javljenu adresu i port. Taj registar je moguće kontaktirati i dobiti kompletan spisak svih pokrenutih instanci i na osnovu podataka onda odrediti koju konkretnu instancu kontaktirati.

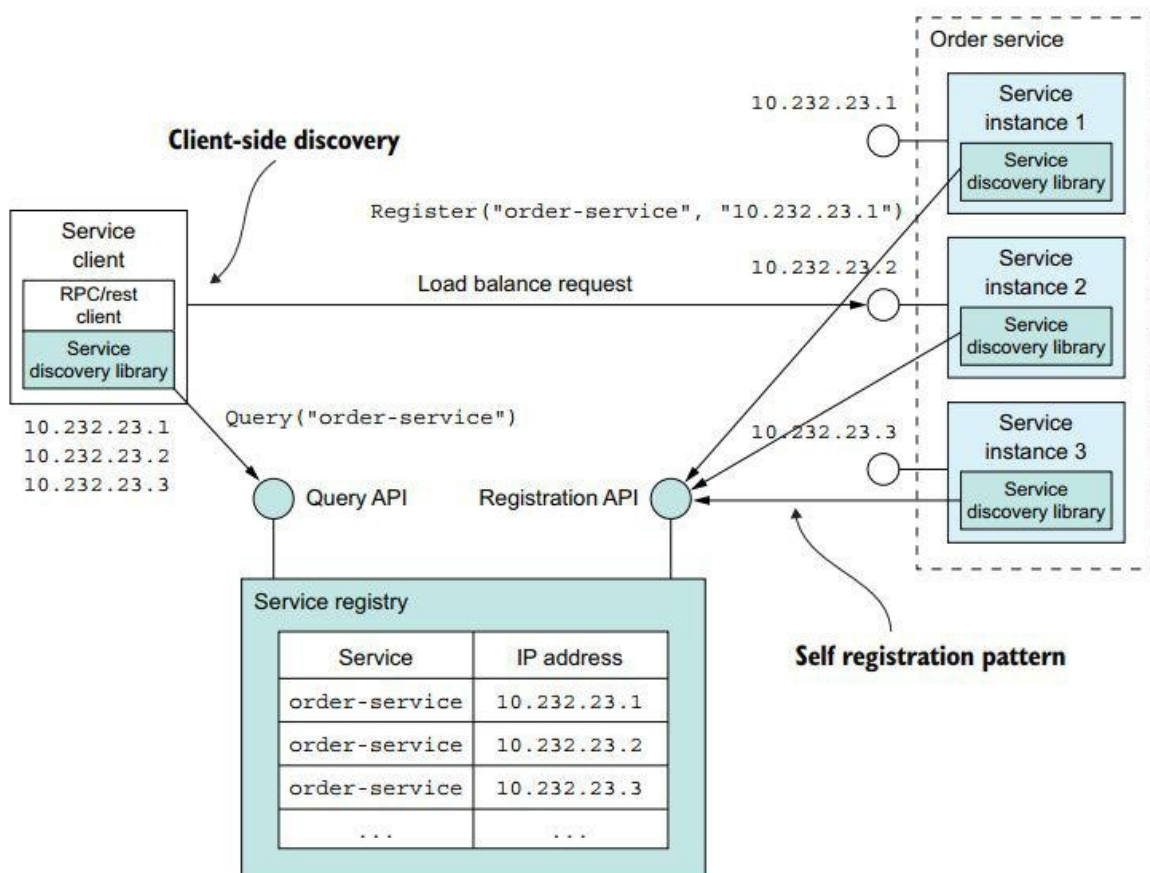


Postoje dva načina da se implementira service discovery:

1. Klijenti i servisi direktno interaguju sa registrom (Application-Level Service Discovery).
2. Deployment infrastruktura se brine za discovery (Platform-Provided Service Discovery).

Prvi način se sastoji u već opisanom toku. Dakle instanca se javlja nekom centralnom registru, koji beleži lokaciju. Pored beleženja, stavke koje su bitne jesu Health-Checking, odnosno u slučaju otkaza neke instance, ne želimo da je kontaktiramo i samim tim dobijemo timeout ili viši latency usled dodatnog pronalaženja neke aktivne instance. Takođe ovi sistemi se koriste i za potrebe distribuirane konfiguracije, što dodatno može olakšati održavanje infrastrukture.

Klijent kada želi da se obrati nekoj instanci, kontaktira „centralno čvorište,, i dobija podatke na osnovu kojih može da pronađe tačan servis, odnosno konkretnu instancu upotrebom nekom load-balancing algoritma.

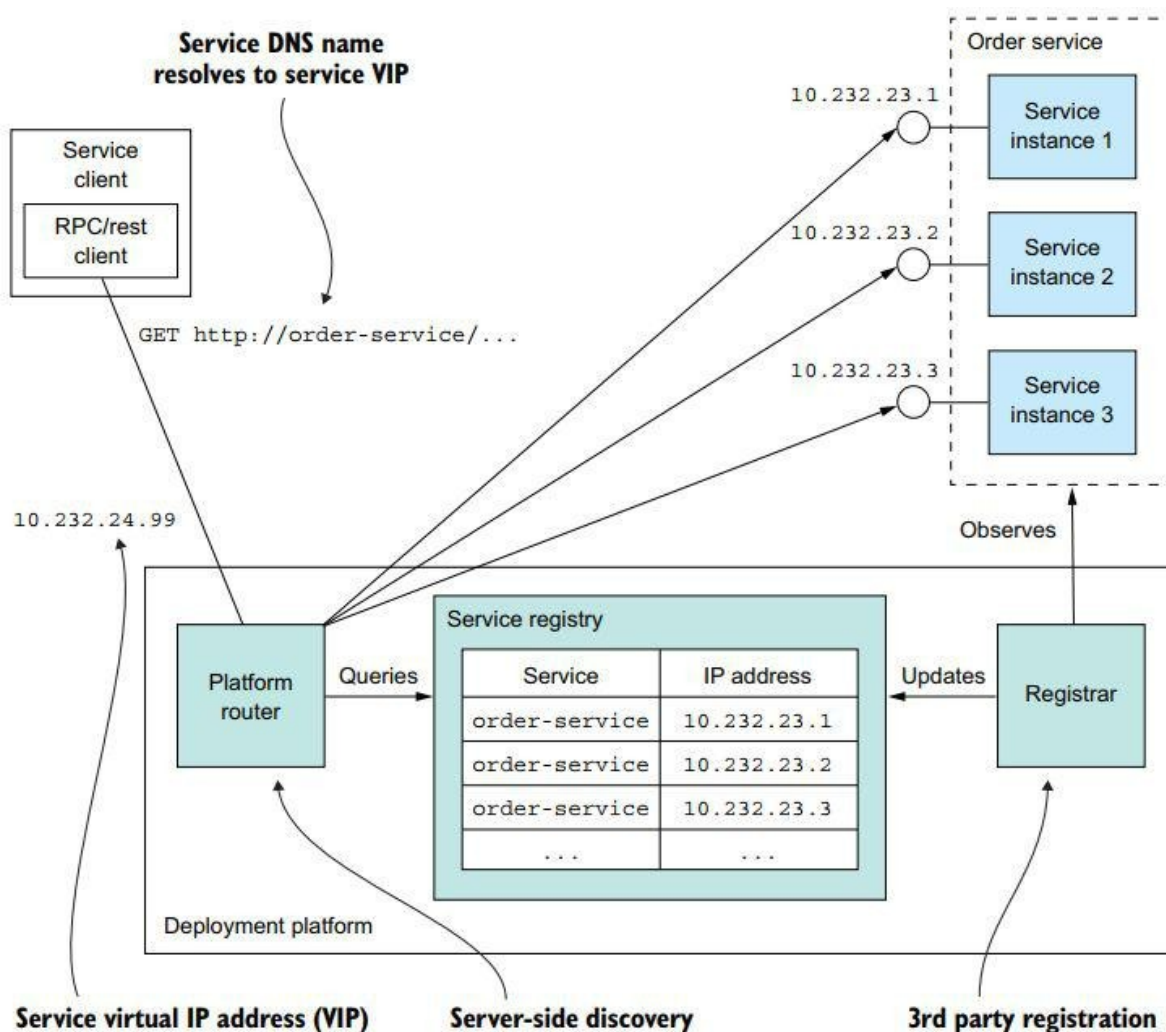


Prednosti načina (Application-Level Service Discovery) leže u tome da apsolutno ne zavisimo od deployment platforme. Ali nedostaci se ogledaju u tome da su često library-iji usko vezani za neki jezik ili framework. Dodatno, opet u zavisnosti od tehnologija, često je potrebna konfiguracija samih servisa, a nekad sadrže i dodatan kod koji se odnosi na service discovery, što želimo da izbegnemo ako je moguće. Takođe, jesu još jedan moving part u arhitekturi koji treba održavati i za koji

je neophodno obezbediti high-availability (odnosno izbeći SPOF - **Single Point Of Failure**).

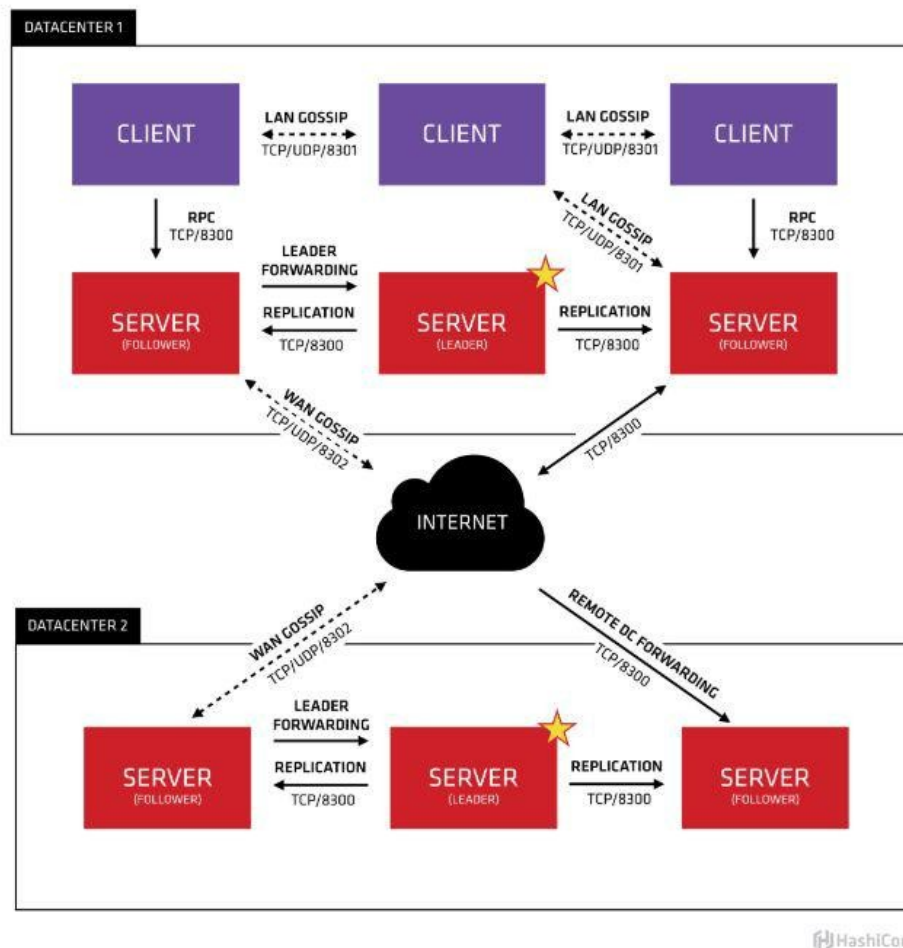
Drugi oblik discovery-a je pružen od strane deployment platforme kao što su Docker i Kubernetes koji su u stanju da u potpunosti odrade Service Discovery i routing što ne zahteva nikakve dodatne elemente koje treba održavati u arhitekturi. Ovo uključuje Server-side discovery (što nije nužno povezano sa deployment platformom). Umesto da instance direktno komuniciraju jedna sa drugom, zahtev se šalje ka ruteru, koji predstavlja ulaznu tačku, koji će se onda obratiti Service Registry-u za neophodne informacije. Sa stanovišta registrovanja instanci one ne moraju same da se registruju, već ih može neko drugi registrovati (3rd party registration - registar).

Pored očiglednih benefita, u ovom obliku postoji i jedan očigledan nedostatak, a to je da je moguće vršiti discovery samo onih servisa koji su deploy-ovani upotrebom konkretne platforme.



1.1 HashiCorp Consul

Za potrebe Service Discovery-a danas postoji veći broj rešenja, a jedno veoma zgodno rešenje je Consul. Jedan kompleksan sistem, koji se sastoji od većeg broja moving part-ova koji zajedno omogućuju našoj infrastrukturi da se ponaša fleksibilno i reaktivno. Arhitektura Consul-a izgleda na sledeći način:



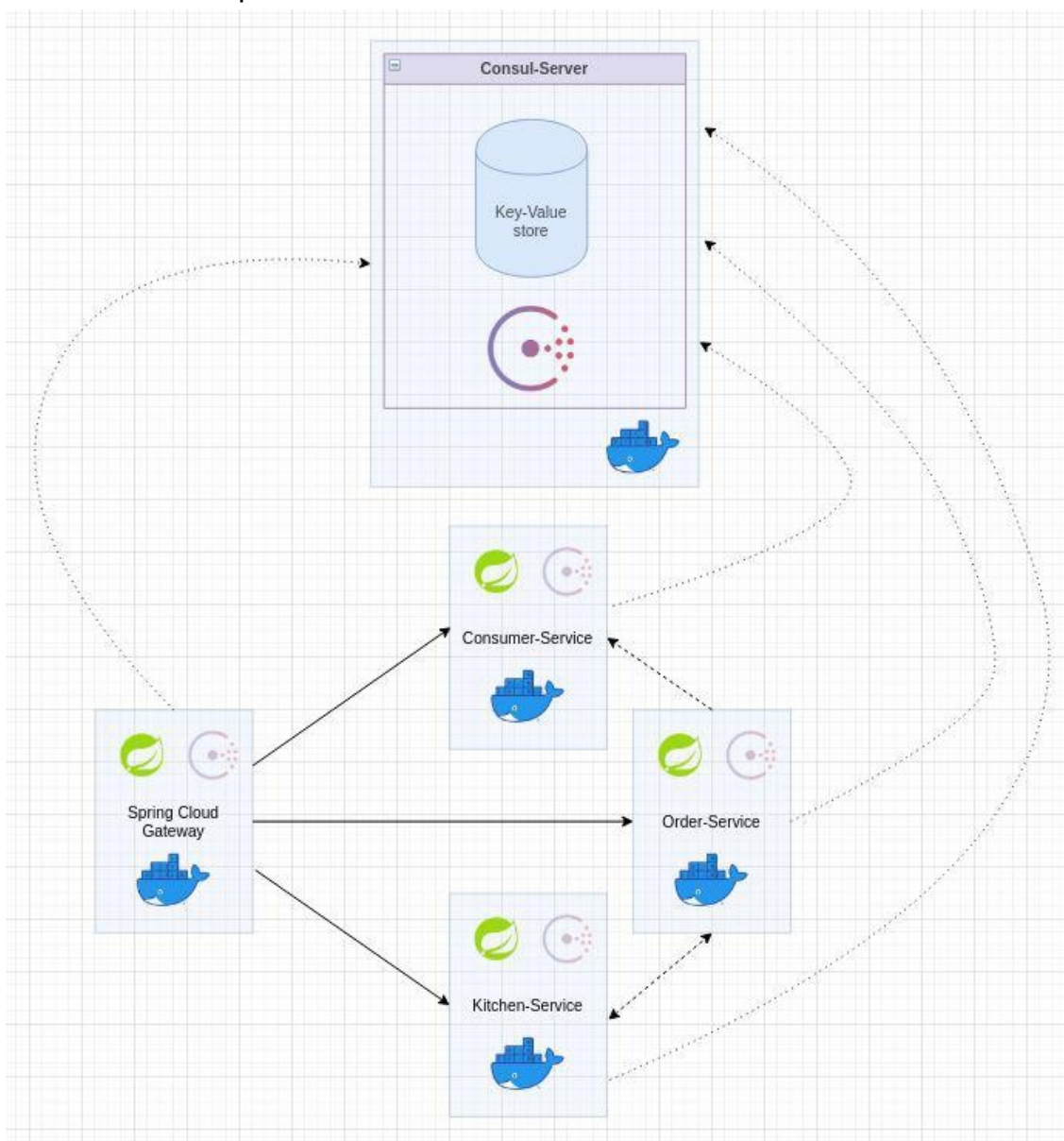
Ono što je evidentno sa same slike, je da Consul pruža podršku za više Datacentre-ara i pored toga postoje dva tipa agenta. Agent je long running daemon koji se izvršava na svakom članu Consul klastera. On se može pokrenuti u Server i Client modu, pri čemu postoje drastične razlike između njih. Zajedničko im je da svi međusobno komuniciraju (unutar jednog Datacentre-a).

Server-i su zaduženi za čuvanje stanja klastera (Key-Value Store), odnosno čuvaju podatke o registrovanim servisima, konfiguraciji servisa, podatke o tome ko se nalazi u klasteru pri čemu dodatno odgovaraju na upućene RPC zahteve i učestvuju u Raft konsenzus protokolu ([Link1](#), [Link2](#)) zajedno sa drugim Consul Server-ima. Ako bismo pokrenuli samo jedan Consul Server, u slučaju njegovog pada, ne bi mogli da dobavimo podatke o registrovanim servisima, što može uticati da kompletan sistem postane nedostupan. Da bismo se od takvih scenarija zaštitili, možemo pokrenuti više servera, gde u slučaju pada jednog, može uskočiti drugi.

Međutim, da bi naš sistem funkcionisao kako treba, podaci moraju biti adekvatno replicirani između čvorova. Detalje o Raft konsenzus protokolu možete naći na prethodno navedenim linkovima. U Cross-Datacentre komunikaciji su uključeni samo Server-i.

Consul Client-i imaju lakša zaduženja. Oni ne učestvuju u konsenzus protokolu, već im je namena prvenstveno Health-Checking i prosleđivanje RPC poziva i kao Server-i učestvuju u LAN Gossip pool-u. Poenta Gossip protokola je da omogući članstvo, detekciju otkaza i emitovanje događaja. Dovoljno je razumeti da gossip-ing uključuje slučajnu node-to-node komunikaciju preko UDP protokola.

Upotreba Consul-a na primeru za vežbe je prikazana na slici koja grubo oslikava arhitekturu primera:




U jednom Docker kontejneru je pokrenut Consul Server koji će se koristiti za Service Discovery i koji će voditi računa o informacijama o našem klasteru. Ono što

je primetno je da je pokrenuta samo jedna instanca, što znači da naša arhitektura nije Highly-Available, ali trenutno nije akcenat na tome (Za detalje [Link1](#), [Link2](#)). U svakom od ostala četiri kontejnera su pokrenuti Consul Client koji lokalno radi Health-Checking i prosleđuje podatke o registrovanom servisu i Spring Boot aplikacija (servis) koji se samostalno „javlja“, lokalnom Consul Client-u.

Svaki od agenata (Client ili Server) se pokreće komandom **consul agent** pri čemu je moguće zadati dodatne opcije putem komandne linije ([Link](#)). Za oba tipa agenta, nevedene su putanje do konfiguracionih fajlova, pri čemu je na jedan način navedena putanja do fajla, a u drugom načinu je naveden kompletan direktorijum gde se učitavaju svi konfiguracioni fajlovi koji se u njemu nalaze.

```
home > stefan > primer1-xws > consul-server >  entrypoint.sh
```

```
1  #!/bin/sh
2
3  consul agent -config-file=/consul/config/consul-server.json
```

```
home > stefan > primer1-xws > services > Consumer-Service >  entrypoint.sh
```

```
1  #!/bin/sh
2
3  ./consul agent -config-dir=/consul-config &
```

Konfiguracija servera:

```
home > stefan > primer1-xws > consul-server > {} consul
1  {
2      "server": true,
3      "bootstrap": true,
4      "client_addr": "0.0.0.0",
5      "data_dir": "/consul/data",
6      "ui": true
7  }
8
```

- “server”: true - Označava da se Consul agent pokreće u server režimu.
- “bootstrap”: true - Omogućava serveru da se samoizabere u lidera (podešavanje vezano za raft konsenzus protokol).
- “client_addr”: “0.0.0.0” - Učiniti agenta dostupanog svima.
- “data_dir”: “/consul/data” - Putanja na kojoj će adekvatni podaci biti čuvani.
- “ui”: true - Omoguće UI prikaz kompletnog klastera i registrovanih servisa na portu 8500.

Konfiguracija klijenta:

```
home > stefan > primer1-xws > services > Consumer-Service
1  {
2      "retry_join": [
3          "172.20.0.2"
4      ],
5      "data_dir": "/consul-data",
6      "client_addr": "0.0.0.0"
7  }
8
```

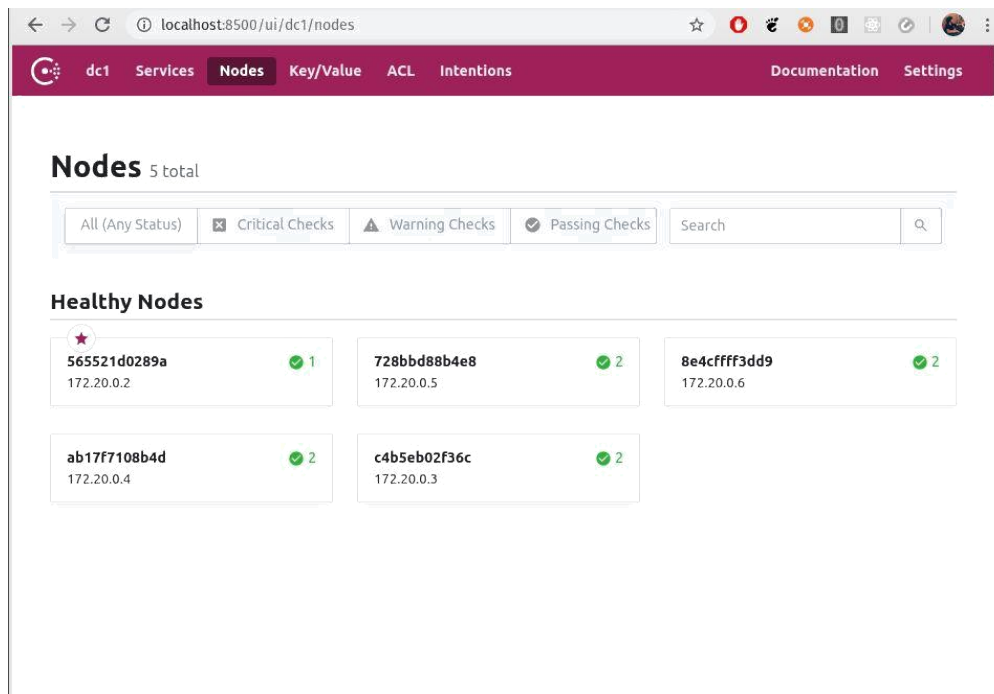
- “retry_join”: [“172.20.0.2”] - Adresa nekog agenta koja omogućuje ućlanavanje u klaster. Ako inicijalno ne može da join-uje, pokušavaće u određenim vremenskim intervalima ponovo.
- “data_dir”: “/consul/data” - Putanja na kojoj će adekvatni podaci biti čuvani.
- “client_addr”: “0.0.0.0” - Učiniti agenta dostupanog svima.

Kompletan primer se može pokrenuti **docker-compose up** komandom. Ako se na bilo kom čvoru pokrene **consul members** komanda, biće prikazani svi članovi klastera:

```
stefan@pop-os ~/primer1-xws docker exec primer1-xws_consul-server_1 consul members
Node      File Edit Address      Port Format T Status  ID-c Type  Build  Protocol  DC Segment
6fd3d9e07f65 172.20.0.2:8301 alive server 1.7.2 2      dc1 <all>
15fda8601f91 172.20.0.5:8301 alive client 1.7.2 2      dc1 <default>
df3a023f4acc 172.20.0.4:8301 alive client 1.7.2 2      dc1 <default>
f2b3910fd64b 172.20.0.6:8301 alive client 1.7.2 2      dc1 <default>
f75eb7837814 172.20.0.3:8301 alive client 1.7.2 2      dc1 <default>
stefan@pop-os ~/primer1-xws
```

Na putanji <http://localhost:8500> je moguće videti UI prikaz registrovanih servisa:

Service	Health Checks	Tags
consul	1	
consumer	2	secure=false
gateway	2	secure=false
kitchen	2	secure=false
order	2	secure=false



Po pitanju naših mikroservisa (Spring-Boot) aplikacija, pored adekvatnog dodatka u pom.xml fajlu (videti u kodu), neophodno je specificirati gde će se izvršiti self-registration, koja putanja će biti korišćena za Health-Check, na koliko će se sekundi isti vršiti, kao i neki manje bitni podaci vezani za preferenciju IP adrese naspram simboličke i na koliko sekundi će biti pokušano ponovno povezaivanje sa lokalnim konzul agentom. To je navedeno u bootstrap.properties fajlu. Bitna podešavanja su naziv aplikacije i port koji su definisani u application.properties fajlu.

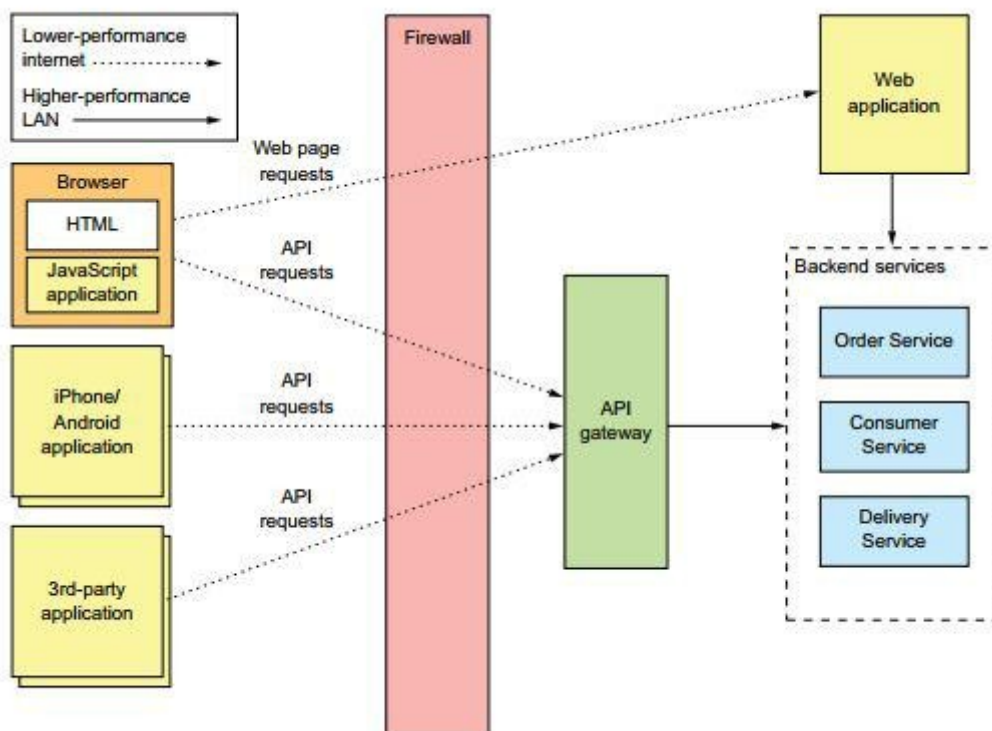
```
home > stefan > primer1-xws > services > Kitchen-Service > src > main > resources > bootstrap.properties
1  spring.cloud.consul.host=localhost
2  spring.cloud.consul.port=8500
3
4  spring.cloud.consul.discovery.health-check-path=/health
5  spring.cloud.consul.discovery.health-check-interval=5s
6  spring.cloud.consul.discovery.catalog-services-watch.enabled=false
7  spring.cloud.consul.discovery.prefer-ip-address=true
8  spring.cloud.consul.retry.initial-interval=2000
```

```
home > stefan > primer1-xws > services > Kitchen-Service > src > main
1  server.port=8080
2  spring.application.name=kitchen
3
```


2. API Gateway

U slučaju da naš sistem treba da koriste različiti klijenti (web aplikacija, android aplikacije itd.), iako je Service Discovery relativno lako rešiv problem, tj. svaki od klijenata može da pristupi adekvatnom servisu, potrebe istih se mogu poprilično razlikovati. Android aplikacija za neku funkcionalnost možda kontaktira drugačiji skup servisa ili možda način upotrebe našeg sistema od strane web aplikacije nije adekvatan za Android aplikaciju tj. Treba na neki način napraviti razliku pri konzumiranju naše aplikacije od strane različitih klijenata. Direktan pristup servisima u nekim situacijama nije dovoljno dobar, što zbog loše enkapsulacije, što zbog toga što to često zahteva neku dodatnu logiku na klijentu u zavisnosti od osobina istih.

U rešavanju ovih problema, od pomoći nam je API Gateway patern koji predstavlja ulaznu tačku u našu aplikaciju. Služi kao fasada iza koje se nalaze mikroservisi što odaje utisak upotrebe jedne monolitne aplikacije (enkapsulacija). Ovaj dodatni sloj je zadužen rutiranje, odnosno prosleđivanje zahteva ka adekvatnim mikroservisima (radi i load-balancing), predstavlja tačku koja se može koristiti za API Composition, radi translaciju protokola, omogućuje bolje sprovođenje bezbednosti itd.



Međutim kao i svi patterni, ni ovo nije silver bullet. Kao i sa Application-Level Service Discovery-em, API Gateway predstavlja još jednu komponentu koju treba održavati i o kojoj voditi računa po pitanju visoke dostupnosti. S obzirom da predstavlja ulaznu tačku, odnosno klijenti će slati zahteve na adresu koja je vezana

za Gateway i s obzirom da je relativno statička, skaliranje je drugačije prirode i pri lošoj konfiguraciji, Gateway može postati usko grlo. Pošto je to dodatna tačka između klijenta i mikroservisa, biće i veći latency što može ugroziti korisničko iskustvo.

2.1 NGINX

Za potrebe primera korišćen je NGINX. NGINX je open-source veb server koji može biti korišćen i kao reverse proxy, load balancer i mail proxy.

Konfiguracija počinje sa glavnim konfiguracionim fajlom, **nginx.conf**. Za uključivanje API gateway konfiguracije koristi se **include** direktiva u **http** bloku u **nginx.conf** koja referencira fajl koji sadrži gateway konfiguraciju, **api_gateway.conf**.

Izgled **nginx.conf** fajla:

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include        /etc/nginx/mime.types;
    default_type   application/octet-stream;

    log_format     main '$remote_addr - $remote_user [$time_local] "$request" '
                        '$status $body_bytes_sent "$http_referer" '
                        '"$http_user_agent" "$http_x_forwarded_for"';
    access_log     /var/log/nginx/access.log  main;

    sendfile       on;
    #tcp_nopush    on;

    keepalive_timeout  65;

    include /etc/nginx/api_gateway.conf; # All API gateway configuration
    include /etc/nginx/conf.d/*.conf;   # Regular web traffic
}
```

API gateway konfiguracioni fajl definiše virtuelni server koji klijentima omogućava korišćenje NGINX-a kao API Gateway-a. Omogućava pristup svim API-jima putem jedne pristupne tačke. U primeru, server “sluša” na svim adresama na portu 8080. U **upstream** blokovima za svaki servis se koriste parovi adresa-port da bi se definisalo gde je API deploy-ovan. Mogu biti korišćeni i hostnames.

Kompletan API je definisan kolekcijom **location** blokova koji specificiraju URI-je koji bivaju rutirani do servisa. Redosled location direktiva nije bitan, bira se najspecifičniji match. **Proxy_pass** direktiva vrši rutiranje do odgovarajuće upstream grupe. **Rewrite** direktiva omogućava izmenu URI-ja zahteva. Prvi parametar je regularan izraz sa kojim treba da se poklapa URI zahteva. Drugi parametar je zamena za taj URI. Treći parametar, flag, u ovom slučaju zaustavlja procesiranje rewrite direktiva i obustavlja potragu za lokacijama koje se poklapaju sa novim URI-jem.

Izgled **api_gateway.conf** fajla:

```
upstream restaurant-order {
    zone upstream-ecommerceApp 64k;
    least_conn;
    server order-service:8080 max_fails=3 fail_timeout=60 weight=1;
}

upstream restaurant-consumer {
    zone upstream-imageApp 64k;
    least_conn;
    server consumer-service:8080 max_fails=3 fail_timeout=60 weight=1;
}

upstream restaurant-kitchen {
    zone upstream-productApp 64k;
    least_conn;
    server kitchen-service:8080 max_fails=3 fail_timeout=60 weight=1;
}

server {
    access_log /var/log/nginx/api_access.log main;

    listen 8080 default_server;

    location /api/order {
        proxy_pass http://restaurant-order;
        rewrite ^/api/order/(.*)$ /$1 break;
    }

    location /api/consumer {
        proxy_pass http://restaurant-consumer;
        rewrite ^/api/consumer/(.*)$ /$1 break;
    }

    location /api/kitchen {
        proxy_pass http://restaurant-kitchen;
        rewrite ^/api/kitchen/(.*)$ /$1 break;
    }
}
```