Spring Cloud consul

**Spring Cloud Consul?**

Let’s imagine that you are writing some code that invokes a service that has a REST API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

Service Discovery:

In a modern, cloud‑based microservices application, however, this is a much more difficult problem to solve. Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.  
 Service Discovery:

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance’s registration is typically refreshed periodically using a heartbeat mechanism.

Examples of service registries include: [Netflix Eureka](https://github.com/Netflix/eureka), etcd, [consul](https://www.consul.io/), [Apache Zookeeper](http://zookeeper.apache.org/)

<https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>

[**consul**](https://www.consul.io/)**:** A tool for discovering and configuring services. It provides an API that allows clients to register and discover services. It provides several key features:

Service Discovery: Clients of Consul can provide a service, such as api or mysql, and other clients can use Consul to discover providers of a given service. Using either DNS or HTTP, applications can easily find the services they depend upon.

Health Checking: Consul clients can provide any number of health checks, either associated with a given service ("is the webserver returning 200 OK"), or with the local node ("is memory utilization below 90%"). This information can be used by an operator to monitor cluster health, and it is used by the service discovery components to route traffic away from unhealthy hosts.

KV Store: Applications can make use of Consul's hierarchical key/value store for any number of purposes, including dynamic configuration, feature flagging, coordination, leader election, and more. The simple HTTP API makes it easy to use.

Multi Datacenter: Consul supports multiple datacenters out of the box. This means users of Consul do not have to worry about building additional layers of abstraction to grow to multiple regions.

**Using Spring cloud consul?**

These are following component to provide basic idea of Spring Batch.

Click with ctrl to navigate to these topics.

1. [Installing Consul](#installconsul1)
2. [Creating Service with Spring Cloud](#createservice2)
3. [Consul client properties/configuration](#configuration3)
4. [Checking Application health & status](#healthstatus4)
5. [Service Discovery](#servicediscovery5)
6. [Customizing health check](#customhealth6)
7. [Custom Health Indicator](#customhealthindicator7)
8. **Installing Consul**

Consul is a distributed, highly available system. Every node that provides services to Consul runs a Consul agent. Running an agent is not required for discovering other services or getting/setting key/value data. The agent is responsible for health checking the services on the node as well as the node itself.

The agents talk to one or more Consul servers. The Consul servers are where data is stored and replicated. The servers themselves elect a leader. While Consul can function with one server, 3 to 5 is recommended to avoid failure scenarios leading to data loss. A cluster of Consul servers is recommended for each datacenter.

Consul must first be installed on your machine. Consul is distributed as a binary package for all supported platforms and architectures. After downloading Consul, unzip the package. Consul runs as a single binary named consul.

The first thing we need to do is start the consul program on one of our servers in server and bootstrap mode. The server mode means that the consul will start up as a server instance instead of a client. The bootstrap option is used for the first server. This allows it to designate itself as the "leader" for the cluster.

consul agent -server -bootstrap -data-dir /tmp/consul

To start consul server with **ui** append the above command option with ***-ui***. The same configuration can be added by using configuration file which is more feasibile as follows.

consul agent -config-file config.json config.json

{

"bind\_addr": "10.219.28.53",

"data\_dir": "/tmp/consul",

"server": true,

"ui\_dir": "../ui",

"bootstrap\_expect":1

}

The beside code shows consul configuration to load from. The first bind addr is showing current system ip address. Data\_dir is to store service information.

This above two wasy will start the consul server with ui. To verify, open a browser & paste this url:

http://localhost:8500

To start the consul in development mode use this command: consul agent -dev

It will start the Consul agent in development mode for now. This mode is useful for bringing up a single-node Consul environment quickly and easily. It is not intended to be used in production.

**Service Configurations:**

By using above configuration file, we can register the any service along with its checks definition. It is useful in scenario like running consul with predefined service configuration and later these services will start remotely or locally.

In configuration, we can give information of services to get registered with consul like service name, address of service, port, and check definition. Checks definition is used check health of the service instance. Below is sample configuration that will start consul agent with a sample service `test-consul` at given location. After that consul agent will look for that service instance actually running at given address & port.

config.json

{

"bind\_addr": "10.219.28.149",

"data\_dir": "/tmp/consul",

"server": true,

"ui": true,

"bootstrap\_expect":1,

"services": [

{

"name": "test-consul",

"address":"127.0.0.1",

"port":8910,

"checks": [

{

"name": "service `test-consul` check",

"http": "http://localhost:8910/health",

"method": "GET",

"interval": "10s"

}

]

}

]

}

<https://www.consul.io/intro/getting-started/agent.html>

1. **Creating Service with Spring Cloud**

Now, We’ll create a service which will exposed over rest call. This spring boot based rest application will register itself with consul server running in the system automatically by using spring cloud consul api. To begin with first add these jars to your application build.gradle file.

compile("org.springframework.boot:spring-boot-starter-web") {

exclude module: "spring-boot-starter-tomcat"

}

compile("org.springframework.boot:spring-boot-starter-jetty")

compile group: 'org.springframework.boot', name: 'spring-boot-actuator', version: '1.5.6.RELEASE'

compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-consul-all', version: '1.0.2.RELEASE'

Now, as second step expose one rest service as you wish to. By annotating your class as @RestController. To register this application with consul, Spring cloud consul comes with annotation @EnableDiscoveryClient which will make this service as Discoveryclient to register itself with consul.

This annotation looks for implementations of the DiscoveryClient interface via META-INF/spring.factories. Implementations of Discovery Client will add a configuration class to spring.factories under the org.springframework.cloud.client.discovery.EnableDiscoveryClient key. Examples of DiscoveryClient implementations: are Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery and Spring Cloud Zookeeper Discovery.

By default, implementations of DiscoveryClient will auto-register the local Spring Boot server with the remote discovery server. This can be disabled by setting autoRegister=false in @EnableDiscoveryClient.

<https://cloud.spring.io/spring-cloud-commons/multi/multi__spring_cloud_commons_common_abstractions.html#__enablediscoveryclient>

Start the application, and navigate to browsers localhost:8500. It will show the service started with default configuration.

Note: consul will show service & its health only if consul is started with ui.

1. **Consul client properties/configuration**

As discussed in previous section, application will be created to register with consul. To modify the default behavior, add some config in application.properties or bootstrap.properties file at classpath resource.

bootstrap.properties is used only with spring cloud consul to load the configuration before loading properties from application.properties. It is loaded by parent application context before the child one which holds application.properties.

To change the server port & application name in spring boot application, below two properties are given.

#server port specification

Server.port=8910

spring.application.name=test-consul

The above will start the application at port 8910 with application name as test-consul. Consider below properties which will configure the application to register with consul with default host & port name as follows.

The complete minimal configuration is:

#server port specification

server.port=8910

spring.application.name=test-consul

#spring cloud configuration

spring.cloud.consul.host=http://localhost

spring.cloud.consul.port=8500

spring.cloud.config.enabled=true

spring.cloud.config.discovery.enabled=true

spring.cloud.consul.config.prefix=private

#spring cloud configuration

management.security.enabled=false

spring.cloud.consul.host=http://localhost

spring.cloud.consul.port=8500

These properties will discovery behavior with consul. We will disable the default spring security for spring boot by using last properties given.

spring.cloud.config.enabled=true

spring.cloud.config.discovery.enabled=true

spring.cloud.consul.config.prefix=private

management.security.enabled=false

1. **Checking Application health & status**

By running application, we will service status at <http://localhost:8500/ui/#/dc1/services>, will show service along with all nodes & services. We can view health and other stats by click on those links.

![A screenshot of a cell phone

Description generated with very high confidence]()

Health :

Health of a service instance can be checked by <http://localhost:8500/v1/health/service/test-consul> path where test-consul is service instance name. It will fetch json like result regarding service at a node.

Checks :

Above health rest call of consul of any service instance will entire information regarding health of service. But to get the only checks which is nothing but actual call to /health of actuator service given by spring, use checks. It will json containing checks result.

<https://www.consul.io/docs/agent/checks.html>

A check is defined in a configuration file or added at runtime over the HTTP interface. Checks created via the HTTP interface persist with that node.

To perform only checks on service use: <http://localhost:8500/v1/health/checks/test-consul> where test-consul is service name.

/health

The overall health of a client running can be fetched by <http://localhost:8910/health>. where 8910 is client’s port. It shows application health information. This particular health is not bound to consul. It is coming due to spring actuator service. They’re mainly used to expose different types of information about the running application – health, metrics, info, dump, env etc. It will contain info like:

Description

Status

Discoverycomposite

Diskspace

Refrechscope

Hystrix

Consul

<http://www.baeldung.com/spring-boot-actuators>

1. **Service Discovery**

Till now, we have created application which will be registered as a service. But whole point of this scenario is to get the application instance access to other client which will make use of the service without knowing physical path or ip address with port.

1. Service instance register itself with consul which is running in background.
2. Now, client application which wants to consume the service, will get service details (IP add & port) from consul using discovery service.
3. After getting ip add & port, client can consume service [ without getting the physical location of service running ].

Create a simple spring boot application to over rest api to consume registered service above. Generally, we user RestTemplate to consume any rest over spring application. RestTemplate required service uri in order to consume it which is usually hard-coded.



client

Service

instance

String url = "<http://localhost:8080/rest/city/capital>";

restTemplate.getForObject(url, String[].class);

Using service discovery

So this is the moment to introduce service discovery. To enable this, client application should have consul dependencies and @EnableDiscoveryClient over boot main app as service instance. Now we can autowired DiscoveryClient.

DiscoveryClient

This DiscoveryClient interface can be used to request instances of all services which have a certain name. So by adding the following code to the capital method:

ServiceInstance serviceInstance =

client.getInstances("testconsul")

.stream()

.findFirst()

.orElseThrow(() -> new Exception());

String uri = serviceInstance.getUri().toString() + "/consul/get/";

This can be done in other ways also, as getInstance(“service-id”); returns list of serviceintances belongs to that serviced-id. If number of instances are there, we can choose anyone of it.

**long** count = client.getInstances("test-consul").size();

<http://blog.trifork.com/2016/12/14/service-discovery-using-consul-and-spring-cloud/>

1. **Customizing health check**

As we seen, consul agent running in background will frequently checking the application health after fixed interval. Default behavior is to check health with endpoint available from spring-actuator which /health. Based on actuator end-points for health checking consul service will be up or down.   
To customize the health check for consul, add below properties in application.properties or bootstrap.properties.

#enable custom health check api for this service

spring.cloud.consul.discovery.healthCheckUrl=http://localhost:${server.port}/status

spring.cloud.consul.discovery.healthCheckInterval=15s

The above configuration will make the consul agent to check for current service health by making rest call to http://localhost:${server.port}/status. This call can be customized to return HttpStatus which may depend on application requirement.

Sample implementation for above rest call:

@RequestMapping("/status")

**public** ResponseEntity<?> checkHealth() {

//add logic to check health

**return** **new** ResponseEntity<>( HttpStatus.***INTERNAL\_SERVER\_ERROR*** );

}

1. **Customizing HealthIndicator**

By default, if you add spring actuator to your application then it will support few endpoints to get some application details. Here are some of the most common endpoints Boot provides out of the box:

/health – Shows application health information (a simple ‘status’ when accessed over an unauthenticated connection or full message details when authenticated). It is not sensitive by default.

/info – Displays arbitrary application info. Not sensitive by default.

/metrics – Shows ‘metrics’ information for the current application. It is also sensitive by default.

/trace – Displays trace information (by default the last few HTTP requests).

Internal Flow of health checking:

If /health is accessed for boot application, request lands to HealthMvcEndPoint’ invoke() method which will call getHealth() of same class. Here it will call invoke() of HealthEndPoint. Now invoke() of HealthEndPoint will call all health() from all HealthIndicator’ implementations.

@Override

**public** Health invoke() {

**return** **this**.healthIndicator.health();

}

This above code will call all healthindicator implementations including custom implementation also.

The above diagram depicts two spring implementations of HealthIndicator interface & custom implementation by user, if any.

AbstractHealthIndicator is an abstract implementation. So, it is further implemented by many more healthindicator which will perform actual health check logic.

Custom Health Indicator

To have custom health indicator, implement your class from HealthIndicator interface by overriding only abstract method health().

**public** **interface** HealthIndicator {

Health health();

}

Sample implementation of HealthIndicator:

@Component

**public** **class** CustomHealthIndicator **implements** HealthIndicator {

@Override

**public** Health health() {

// Actual logic to return health with status.

**if**( (HealthStatus.*code*).value() == 503 )

**return** Health.*down*().withDetail("Error code", "Out of memory error").build();

**return** Health.*up*().build();

}

}

The above code will be executed by invoke() of HealthEndPoint, resulting in custom health status for consul and actuator also.

**OutOfMemoryError scenario:**

Like normal use case of service access by any client using consul, consul ui will reflect the health status based on /health of actuator as default behavior. We have already seen how to customize consul health check in previous section.

Actuator health check will call all the health indicator & based on their health staus, consul will also flag its health. But, sometimes any unexcepted exception may occur which may affect the application from giving coded result. All the given health indicator implementations provide health status based on existing implementation. But it is not sufficient in some cases. One case may be **OutOfMemoryError.**

A running application may throw OutOfMemoryError which cannot be handled using try/catch block. This will make the application’s health as ok but application won’t be in state of giving result. For handling these cases, custom health indicator may come handy.

One implementation may be to use *@ExceptionHandler* along with *@RestControllerAdvice.* This can handle any exception including OutOfMemoryError.

@RestControllerAdvice

**public** **class** RestExceptionHandler {

@ExceptionHandler({ OutOfMemoryError.**class** })

**public** ResponseEntity<?> handle() {

HealthStatus.*msg* = "test-consul out of memory error";

//HealthStatus.code = HttpStatus.valueOf(503);

HealthStatus.*code* = HttpStatus.***SERVICE\_UNAVAILABLE***;

**return** **new** ResponseEntity<>(HttpStatus.***BAD\_REQUEST***);

}

}

Observe, HealthStatus.code, it is the common flag used by custom health indicator to determine where to return status as up or down. Check previous code block [ customhealth indicator ].

To generate outofmemory in application, make a rest call to : <http://localhost:8910/consul/out>

Which will land to this code to generate this error:

@GetMapping(path = "/out")

**public** ResponseEntity<?> getoutofmemory(){

*logger*.info("Start : " + getClass().getName() + " : getoutofmemory()");

**long**[] ar = **new** **long**[Integer.***MAX\_VALUE***];

**return** **new** ResponseEntity<>("success" , HttpStatus.***OK***);

}

Which will handled by exception handler.