**What is Service discovery:**

Service discovery is a key component of most distributed systems and service oriented architectures. The problem seems simple at first:

*How do clients determine the IP and port for a service that exist on multiple hosts?*

Usually, we start off with some static configuration which gets us pretty far. Things get more complicated as we start deploying more services. There are many ways service locations can change in a live system, auto or manual scaling, new deployments of services, as well as hosts failing or being replaced.

Dynamic service registration and discovery becomes very important in these scenarios in order to avoid service interruption.

There are two sides to the problem of locating services. • Service Registration • and Service Discovery.

**Service Registration:**

The process of a service registering its location in a central registry. It usually register its host and port and sometimes authentication credentials, protocols, versions numbers, and/or environment details.

**Service Discovery:**

The process of a client application querying the central registry to learn of the location of services.

Example of popular service discovery systems are Consul, Zookeeper, etcd etc. Out of these, Consul is the only one that really tries to provide a comprehensive solution for service discovery.

**Consul:**

consul is a service discovery tool from [hashicorp](https://www.hashicorp.com/). Consul provides a consistent view of services and configuration. Consul monitors and changes service information based on the health of nodes. Consul provides a REST interface and web UI to see services and service configurations. Consul organizes services in service catalog and provides a DNS/REST/HTTP interface to it.

* **Service Discovery**: Clients of Consul can provide a service, such as API or postgresql, 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.
* **Key/Value 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.

To use Consul you start up an agent process. The Consul agent process is a long running daemon on every member of Consul cluster. The agent process can be run in server mode or client mode. Consul agent clients would run on every physical server or OS virtual machine (if that makes more sense). Client runs on server hosting services. The clients use gossip and RPC calls to stay in sync with Consul.

Consul is built on top of serf, [a full gossip protocol](https://www.serfdom.io/).

**setting up a local consul cluster**

consul can be downloaded from [consul](https://www.consul.io/downloads.html) website . On OSX, if you are using homebrew as a package manager, consul can be installed from homebrew.

brew cask install consul

Verify the installation by running:

$ consul

usage: consul [--version] [--help] <command> [<args>]

Available commands are:

agent Runs a Consul agent

configtest Validate config file

event Fire a new event

exec Executes a command on Consul nodes

force-leave Forces a member of the cluster to enter the "left" state

info Provides debugging information for operators

join Tell Consul agent to join cluster

keygen Generates a new encryption key

keyring Manages gossip layer encryption keys

leave Gracefully leaves the Consul cluster and shuts down

lock Execute a command holding a lock

maint Controls node or service maintenance mode

members Lists the members of a Consul cluster

monitor Stream logs from a Consul agent

reload Triggers the agent to reload configuration files

rtt Estimates network round trip time between nodes

version Prints the Consul version

watch Watch for changes in Consul

If you get an error, set your PATH correctly.

After Consul is installed, the agent must be run. The agent can run either in server or client mode. Each datacenter must have at least one server, a cluster of 3 or 5 servers is recommended.

All other agents run in client mode. A client is a very lightweight process that registers services, runs health checks, and forwards queries to servers. The agent must be run on every node that is part of the cluster so that we can get information from every node.

*-dev* will run consul in dev mode.

consul agent -dev

We can use CTRL+C to stop the consul server.

We have a server running, now let’s add a service to our consul server.

**Consul Service:**

A service can be defined by providing a consul [service definition](https://www.consul.io/docs/agent/services.html) or by making [HTTP calls](https://www.consul.io/docs/agent/http/agent.html#agent_service_register) to consul server.

**Example service definition:**

{

"service": {

"name": "myservice",

"tags": ["prod"],

"address": "127.0.0.1",

"port": 8000,

"enableTagOverride": false,

"checks": [

{

"script":"check",

"interval": "10s"

}

]

}

}

**Querying Consul service**

We can query consul service using DNS or HTTP API.

**DNS API:**

dig @127.0.0.1 -p 8600 myservice.service.consul

**HTTP API:**

curl http://localhost:8500/v1/catalog/service/myservice

Once we have our server and client up and running. We can use consul to find our services

**Service Discovery from Golang:**

We can use both DNS and HTTP [API](https://godoc.org/github.com/hashicorp/consul/api) to discover service information from consul. I have only used HTTP API and that’s what we are going to use today. We create an interface that give us methods to register, deregister, and get services from consul.

Example:

Package consul

import (

"fmt"

"time"

consul "github.com/hashicorp/consul/api"

)

//Client provides an interface for getting data out of Consul

type Client interface {

// Get a Service from consul

Service(string, string) ([]string, error)

// Register a service with local agent

Register(string, int) error

// Deregister a service with local agent

DeRegister(string) error

}

type client struct {

consul \*consul.Client

}

//NewConsul returns a Client interface for given consul address

Func NewConsulClient(addr string) (Client, error) {

config := consul.DefaultConfig()

config.Address = addr

c, err := consul.NewClient(config)

if err != nil {

return nil, err

}

return &client{consul: c}, nil

}

// Register a service with consul local agent

func (c \*client) Register(name string, port int) error {

reg := &consul.AgentServiceRegistration{

ID: name,

Name: name,

Port: port,

}

return c.consul.Agent().ServiceRegister(reg)

}

// DeRegister a service with consul local agent

func (c \*client) DeRegister(id string) error {

return c.consul.Agent().ServiceDeregister(id)

}

// Service return a service

func (c \*client) Service(service, tag string) ([]\*ServiceEntry, \*QueryMeta, error) {

passingOnly := true

addrs, meta, err := c.consul.Health().Service(service, tag, passingOnly, nil)

if len(addrs) == 0 && err == nil {

return nil, fmt.Errorf("service ( %s ) was not found", service)

}

if err != nil {

return nil, err

}

return addrs, meta, nil

}

Now when we have consul running and we know how to interact with consul with golang. It is time to build .

http://varunksaini.com/consul-service-discovery-golang/

https://sreeninet.wordpress.com/2016/04/17/service-discovery-with-consul/

In a Microservices architecture, Services are dynamic, distributed and present in large numbers. It is needed to have a good Service discovery solution to address this dynamic problem. In this blog, I will cover basics of Service discovery and using Consul to do Service discovery.

### What is Service discovery?

Service discovery should provide the following:

1. Discovery – Services need to discover each other to get IP address and port detail to communicate with other services in the cluster.
2. Health check – Only healthy services should participate in handling traffic, unhealthy services need to be dynamically pruned out.
3. Load balancing – Traffic destined to a particular service should be dynamically load balanced to all instances providing the particular service.

Following are the critical components of Service discovery:

* Service Registry – Maintains a database of services and provides an external API(HTTP/DNS) to interact. This is typically Implemented as a distributed key, value store.
* Registrator – Registers services dynamically to Service registry by listening to events.
* Health checker – Monitors Service health dynamically and updates Service registry appropriately.
* Load balancer – Distribute traffic destined to service to active participants.

### Consul for Service Discovery

There are many possible solutions(etcd, zookeeper, skydns) available for Service discovery. Consul is a comprehensive Service discovery solution that covers all the critical components explained above. Following are some architecture details of Consul:

* Has a distributed key, value(KV) store for storing Service database.
* Provides comprehensive service health checking using both in-built solutions as well as user provided custom solutions.
* Provides REST based HTTP api for interaction.
* Service database can be queried using DNS.
* Does dynamic load balancing.
* Supports single data center and can be scaled to support multiple data centers.
* Integrates well with Docker.

### Consul installation

Consul can be installed in standalone mode or in a cluster of servers. For development purpose, we can use a single cluster node. For production purposes, it is necessary to have a multi-node Consul cluster to have built-in redundancy. When starting a multi-node cluster, it is necessary to specify one Consul neighbor for a node after which Consul members use the gossip protocol to communicate with themselves, elect a leader and create a working cluster. Consul can be installed as an application software or as a Docker Container. For this blog, I will use single node [Consul Docker Container](https://github.com/gliderlabs/docker-consul).  
Following command starts Consul Docker Container:

docker run -d -p 8500:8500 -p 172.17.0.1:53:8600/udp -p 8400:8400 gliderlabs/consul-server -node myconsul -bootstrap

Following are some notes on the above command:

* Within Consul, port 8400 is used for RPC, 8500 is used for HTTP, 8600 is used for DNS. By using “-p” option, we are exposing these ports to the host machine.
* “172.17.0.1” is the Docker bridge IP address. We are remapping Consul Container’s port 8600 to host machine’s Docker bridge port 53 so that Containers on that host can use Consul for DNS.
* “bootstrap” option is used to operate Consul in standalone mode.

Following is the Consul version that I am running. This output is inside Consul Container.

# consul --version

Consul v0.6.3

Following output shows the HTTP REST api example listing the active nodes:

$ curl -s http://localhost:8500/v1/catalog/nodes | jq .

[

{

"ModifyIndex": 4,

"CreateIndex": 3,

"Address": "172.17.0.2",

"Node": "myconsul"

}

]

Following output shows that no active services are running:

$ curl -s http://localhost:8500/v1/catalog/services | jq .

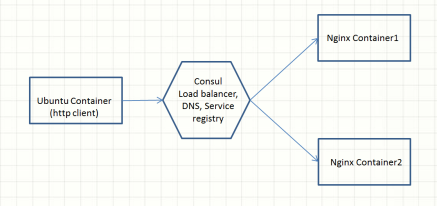
{

"consul": []

}

### Example Application used in this blog

For this blog, we will use the following application.



Following are some notes on the above application:

* Two nginx containers will serve as the web servers. ubuntu container will serve as http client.
* Consul will load balance the request between two nginx web servers.
* DNS and health check will be provided by Consul.
* We will first try this application with manual service registration and later use dynamic service registration using [gliderlabs registrator](https://github.com/gliderlabs/registrator).

Following are some pre-requisites:

To allow Docker Containers to use Docker bridge as the DNS IP address, we need to add the following to Docker start options. For Ubuntu 14.04, we need to specify the following options in “/etc/default/docker”.

DOCKER\_OPTS="--dns 172.17.0.1 --dns 8.8.8.8 --dns-search service.consul"

“172.17.0.1” is the Docker bridge IP address in my case. “dns-search” option allows us to specify the default domain name. When Docker startup options are changed, it is needed to restart Docker engine.

### Example application using manual registration

#### Application with no health check

First, lets start the 3 Docker Containers comprising the application:

docker run -d -P --name=nginx1 nginx

docker run -d -P --name=nginx2 nginx

docker run -ti smakam/myubuntu:v3 sh

Following output shows all running Containers, including 3 application Containers as well as Consul:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

75663b0a2a9f smakam/myubuntu:v3 "sh" 4 seconds ago Up 3 seconds grave\_mahavira

167bb121198e nginx "nginx -g 'daemon off" 42 seconds ago Up 41 seconds 0.0.0.0:32821->80/tcp, 0.0.0.0:32820->443/tcp nginx2

0ea2573fc764 nginx "nginx -g 'daemon off" 44 seconds ago Up 43 seconds 0.0.0.0:32819->80/tcp, 0.0.0.0:32818->443/tcp nginx1

e27c1d7f63db gliderlabs/consul-server "/bin/consul agent -s" About an hour ago Up About an hour 0.0.0.0:8400->8400/tcp, 8300-8302/tcp, 8600/tcp, 8301-8302/udp, 0.0.0.0:8500->8500/tcp, 172.17.0.1:53->8600/udp thirsty\_albattani

Let’s register these 2 services manually:

**http1.json:**

{

"ID": "http1",

"Name": "http",

"Address": "172.17.0.3",

"Port": 80

}

**http2.json:**

{

"ID": "http2",

"Name": "http",

"Address": "172.17.0.4",

"Port": 80

}

curl -X PUT --data-binary @http1.json http://localhost:8500/v1/agent/service/register

curl -X PUT --data-binary @http2.json http://localhost:8500/v1/agent/service/register

Now let’s look at the Consul service registry. We can see below that the service name “http” is composed of 2 web servers(172.17.0.3:80, 172.17.0.4:80)

$ curl -s http://localhost:8500/v1/catalog/service/http | jq .

[

{

"ModifyIndex": 372,

"CreateIndex": 372,

"Node": "myconsul",

"Address": "172.17.0.2",

"ServiceID": "http1",

"ServiceName": "http",

"ServiceTags": [],

"ServiceAddress": "172.17.0.3",

"ServicePort": 80,

"ServiceEnableTagOverride": false

},

{

"ModifyIndex": 373,

"CreateIndex": 373,

"Node": "myconsul",

"Address": "172.17.0.2",

"ServiceID": "http2",

"ServiceName": "http",

"ServiceTags": [],

"ServiceAddress": "172.17.0.4",

"ServicePort": 80,

"ServiceEnableTagOverride": false

}

]

Now, let’s look at DNS database. This shows both the web servers that are part of “http” service.

$ dig @172.17.0.1 http.service.consul

; <<>> DiG 9.9.5-3ubuntu0.7-Ubuntu <<>> @172.17.0.1 http.service.consul

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4972

;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;http.service.consul. IN A

;; ANSWER SECTION:

http.service.consul. 0 IN A 172.17.0.3

http.service.consul. 0 IN A 172.17.0.4

We can also look at DNS service records that will show the port number associated with each service. In this case, “http” service exposes port 80.

$ dig @172.17.0.1 http.service.consul SRV

; <<>> DiG 9.9.5-3ubuntu0.7-Ubuntu <<>> @172.17.0.1 http.service.consul SRV

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34138

;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 2

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;http.service.consul. IN SRV

;; ANSWER SECTION:

http.service.consul. 0 IN SRV 1 1 80 myconsul.node.dc1.consul.

http.service.consul. 0 IN SRV 1 1 80 myconsul.node.dc1.consul.

;; ADDITIONAL SECTION:

myconsul.node.dc1.consul. 0 IN A 172.17.0.4

myconsul.node.dc1.consul. 0 IN A 172.17.0.3

Now, let’s try to ping the dns name and check if load balancing is happening. As we can see below, the ping request shifts between 172.17.0.3 and 172.17.0.4 for service “http.service.consul”

$ ping -c1 http.service.consul

PING http.service.consul (172.17.0.4) 56(84) bytes of data.

64 bytes from 172.17.0.4: icmp\_seq=1 ttl=64 time=0.105 ms

--- http.service.consul ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 0.105/0.105/0.105/0.000 ms

sreeni@ubuntu:~/consul$ ping -c1 http.service.consul

PING http.service.consul (172.17.0.3) 56(84) bytes of data.

64 bytes from 172.17.0.3: icmp\_seq=1 ttl=64 time=0.110 ms

--- http.service.consul ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 0.110/0.110/0.110/0.000 ms

In the example above, we have registered the service manually and also there is no health check configured. Because of the manual registration, the services would stay in Consul database even if the service is removed. Because there is no health check, the service would stay in Consul database even if the service dies.  
We can use the following command to deregister the services manually.

curl -X PUT http://localhost:8500/v1/agent/service/deregister/http1

curl -X PUT http://localhost:8500/v1/agent/service/deregister/http2

#### Application with http health check

With http based health check, Consul takes care of doing periodic http check and remove the service from registry if the health check fails. Consul provides other health check mechanisms other than http.  
Following are two example services with http based health check and the example shows how to register them to Consul.

**http1\_checkhttp.json:**

{

"ID": "http1",

"Name": "http",

"Address": "172.17.0.3",

"Port": 80,

"check": {

"http": "http://172.17.0.3:80",

"interval": "10s",

"timeout": "1s"

}

}

**http2\_checkhttp.json:**

{

"ID": "http2",

"Name": "http",

"Address": "172.17.0.4",

"Port": 80,

"check": {

"http": "http://172.17.0.4:80",

"interval": "10s",

"timeout": "1s"

}

}

curl -X PUT --data-binary @http1\_checkhttp.json http://localhost:8500/v1/agent/service/register

curl -X PUT --data-binary @http2\_checkhttp.json http://localhost:8500/v1/agent/service/register

Following command shows the configured health checks and shows the status as passing.

$ curl -s http://localhost:8500/v1/health/checks/http | jq .

[

{

"ModifyIndex": 424,

"CreateIndex": 423,

"Node": "myconsul",

"CheckID": "service:http1",

"Name": "Service 'http' check",

"Status": "passing",

"Notes": "",

"Output": "",

"ServiceID": "http1",

"ServiceName": "http"

},

{

"ModifyIndex": 427,

"CreateIndex": 425,

"Node": "myconsul",

"CheckID": "service:http2",

"Name": "Service 'http' check",

"Status": "passing",

"Notes": "",

"Output": "",

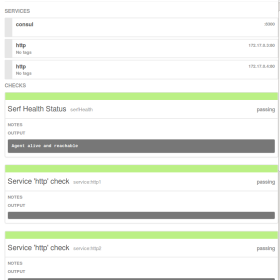
"ServiceID": "http2",

"ServiceName": "http"

}

]

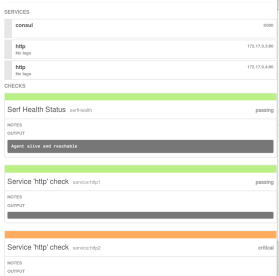
Following picture shows the Consul GUI and services with health check passing for “http” service.



To prove that the health check is working, lets kill one of the Docker containers using:

docker rm -f nginx2

Following picture shows that the health check for “http2” service is critical:



Once the health check fails, the service gets removed from service registry. Following command shows that only 1 active web server is present for “http” service now.

$ dig @172.17.0.1 http.service.consul

; <<>> DiG 9.9.5-3ubuntu0.7-Ubuntu <<>> @172.17.0.1 http.service.consul

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36858

;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;http.service.consul. IN A

;; ANSWER SECTION:

http.service.consul. 0 IN A 172.17.0.3

;; Query time: 1 msec

;; SERVER: 172.17.0.1#53(172.17.0.1)

;; WHEN: Sat Apr 09 22:35:10 PDT 2016

;; MSG SIZE rcvd: 72

#### Application with tcp based check

Following is an example service with TCP based check. Here Consul server will do a periodic TCP check to specified IP and port.

{

"ID": "http",

"Name": "http",

"Address": "172.17.0.3",

"Port": 80,

"check": {

"tcp": "172.17.0.3:80",

"interval": "10s",

"timeout": "1s"

}

}

### Application using script based check

Following is an example service using script based check. Here, Consul server periodically executes the user specified script to verify the health of the service.

{

"ID": "http",

"Name": "http",

"Address": "172.17.0.3",

"Port": 80,

"check": {

"script": "curl 172.17.0.3 >/dev/null 2>&1",

"interval": "10s"

}

}

#### Application using TTL based check

Compared to the previous health checks which are triggered by Consul server, this health check is triggered by the service . The service needs to periodically update a shared TTL counter that Consul server maintains. If the TTL counter is not refreshed within a particular period, the service is assumed unhealthy. Following is an example service with TTL based check.

{

"ID": "web",

"Name": "web",

"Address": "172.17.0.3",

"Port": 80,

"check": {

"Interval": "10s",

"TTL": "15s"

}

}

To manually trigger TTL based keepalive update, we can do the following:

curl http://localhost:8500/v1/agent/check/pass/service:http1

curl http://localhost:8500/v1/agent/check/pass/service:http2

### Example application with automatic registration using Registrator

Manual registration with Consul is error-prone and it can be overcome by an application like [Gliderlabs Registrator](https://github.com/gliderlabs/registrator). Registrator listens for Docker events and dynamically updates Consul service registry.

Choosing the IP address for the registration is critical. There are 2 choices:

1. With internal IP option, Container IP and port number gets registered with Consul. This approach is useful when we want to access the service registry from within a Container. Following is an example of starting Registrator using “internal” IP option.
2. docker run -d -v /var/run/docker.sock:/tmp/docker.sock --net=host gliderlabs/registrator -internal consul://localhost:8500
3. With external IP option, host IP and port number gets registered with Consul. Its necessary to specify IP address manually. If its not specified, loopback address gets registered. Following is an example of starting Registrator using “external” IP option.
4. docker run -d -v /var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator -ip 192.168.99.100 consul://192.168.99.100:8500

First, let’s start Consul server and Registrator:

docker run -d -p 8500:8500 -p 172.17.0.1:53:8600/udp -p 8400:8400 gliderlabs/consul-server -node myconsul -bootstrap

docker run -d -v /var/run/docker.sock:/tmp/docker.sock --net=host gliderlabs/registrator -internal consul://localhost:8500

Let’s start the 2 nginx services with service name as “http” and with http based health checks enabled.

docker run -d -p :80 -e "SERVICE\_80\_NAME=http" -e "SERVICE\_80\_ID=http1" -e "SERVICE\_80\_CHECK\_HTTP=true" -e "SERVICE\_80\_CHECK\_HTTP=/" --name=nginx1 nginx

docker run -d -p :80 -e "SERVICE\_80\_NAME=http" -e "SERVICE\_80\_ID=http2" -e "SERVICE\_80\_CHECK\_HTTP=true" -e "SERVICE\_80\_CHECK\_HTTP=/" --name=nginx2 nginx

In the above example, we have passed environment variables that Registrator will use when registering the service with Consul.  
Let’s look at the service details now:

$ curl -s http://localhost:8500/v1/catalog/service/http | jq .

[

{

"ModifyIndex": 20,

"CreateIndex": 16,

"Node": "myconsul",

"Address": "172.17.0.2",

"ServiceID": "http1",

"ServiceName": "http",

"ServiceTags": [],

"ServiceAddress": "172.17.0.3",

"ServicePort": 80,

"ServiceEnableTagOverride": false

},

{

"ModifyIndex": 19,

"CreateIndex": 17,

"Node": "myconsul",

"Address": "172.17.0.2",

"ServiceID": "http2",

"ServiceName": "http",

"ServiceTags": [],

"ServiceAddress": "172.17.0.4",

"ServicePort": 80,

"ServiceEnableTagOverride": false

}

]

As we can see above, “http” service is composed of “http1” with “172.17.0.3:80” and “http2” with “172.17.0.4:80” service.  
We can look at DNS details to confirm the same:

$ dig @172.17.0.1 http.service.consul SRV

; <<>> DiG 9.9.5-3ubuntu0.7-Ubuntu <<>> @172.17.0.1 http.service.consul SRV

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34138

;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 2

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;http.service.consul. IN SRV

;; ANSWER SECTION:

http.service.consul. 0 IN SRV 1 1 80 myconsul.node.dc1.consul.

http.service.consul. 0 IN SRV 1 1 80 myconsul.node.dc1.consul.

;; ADDITIONAL SECTION:

myconsul.node.dc1.consul. 0 IN A 172.17.0.4

myconsul.node.dc1.consul. 0 IN A 172.17.0.3

Following example shows service registration with TTL based check enabled for service exposing port 80.

docker run -d -p :80 -e "SERVICE\_80\_NAME=http" -e "SERVICE\_80\_ID=http1" -e "SERVICE\_80\_CHECK\_TTL=30s" --name=nginx1 nginx

docker run -d -p :80 -e "SERVICE\_80\_NAME=http" -e "SERVICE\_80\_ID=http2" -e "SERVICE\_80\_CHECK\_TTL=30s" --name=nginx2 nginx

As we can see, Consul with Registrator makes the Service discovery process dynamic and easy to manage.

https://www.consul.io/docs/agent/dns.html

# DNS Interface

One of the primary query interfaces for Consul is DNS. The DNS interface allows applications to make use of service discovery without any high-touch integration with Consul.

For example, instead of making HTTP API requests to Consul, a host can use the DNS server directly via name lookups like redis.service.us-east-1.consul. This query automatically translates to a lookup of nodes that provide the redis service, are located in the us-east-1 datacenter, and have no failing health checks. It's that simple!

There are a number of configuration options that are important for the DNS interface, specifically [client\_addr](https://www.consul.io/docs/agent/options.html#client_addr), [ports.dns](https://www.consul.io/docs/agent/options.html#dns_port), [recursors](https://www.consul.io/docs/agent/options.html#recursors), [domain](https://www.consul.io/docs/agent/options.html#domain), and [dns\_config](https://www.consul.io/docs/agent/options.html#dns_config). By default, Consul will listen on 127.0.0.1:8600 for DNS queries in the consul. domain, without support for further DNS recursion. Please consult the [documentation on configuration options](https://www.consul.io/docs/agent/options.html), specifically the configuration items linked above, for more details.

There are a few ways to use the DNS interface. One option is to use a custom DNS resolver library and point it at Consul. Another option is to set Consul as the DNS server for a node and provide a [recursors](https://www.consul.io/docs/agent/options.html#recursors) configuration so that non-Consul queries can also be resolved. The last method is to forward all queries for the "consul." domain to a Consul agent from the existing DNS server.

You can experiment with Consul's DNS server on the command line using tools such as dig:

$ dig @127.0.0.1 -p 8600 redis.service.dc1.consul. ANY

**Note:** In DNS, all queries are case-insensitive. A lookup of PostgreSQL.node.dc1.consul will find all nodes named postgresql.

## [»](https://www.consul.io/docs/agent/dns.html" \l "node-lookups) Node Lookups

To resolve names, Consul relies on a very specific format for queries. There are fundamentally two types of queries: node lookups and service lookups. A node lookup, a simple query for the address of a named node, looks like this:

<node>.node[.datacenter].<domain>

For example, if we have a foo node with default settings, we could look for foo.node.dc1.consul. The datacenter is an optional part of the FQDN: if not provided, it defaults to the datacenter of the agent. If we know foo is running in the same datacenter as our local agent, we can instead use foo.node.consul. This convention allows for terse syntax where appropriate while supporting queries of nodes in remote datacenters as necessary.

For a node lookup, the only records returned are A records containing the IP address of the node.

$ dig @127.0.0.1 -p 8600 foo.node.consul ANY

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 foo.node.consul ANY

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24355

;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 0

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;foo.node.consul. IN ANY

;; ANSWER SECTION:

foo.node.consul. 0 IN A 10.1.10.12

;; AUTHORITY SECTION:

consul. 0 IN SOA ns.consul. postmaster.consul. 1392836399 3600 600 86400 0

## [»](https://www.consul.io/docs/agent/dns.html" \l "service-lookups) Service Lookups

A service lookup is used to query for service providers. Service queries support two lookup methods: standard and strict [RFC 2782](https://tools.ietf.org/html/rfc2782).

### [»](https://www.consul.io/docs/agent/dns.html" \l "standard-lookup) Standard Lookup

The format of a standard service lookup is:

[tag.]<service>.service[.datacenter].<domain>

The tag is optional, and, as with node lookups, the datacenter is as well. If no tag is provided, no filtering is done on tag. If no datacenter is provided, the datacenter of this Consul agent is assumed.

If we want to find any redis service providers in our local datacenter, we could query redis.service.consul. If we want to find the PostgreSQL primary in a particular datacenter, we could query primary.postgresql.service.dc2.consul.

The DNS query system makes use of health check information to prevent routing to unhealthy nodes. When a service query is made, any services failing their health check or failing a node system check will be omitted from the results. To allow for simple load balancing, the set of nodes returned is also randomized each time. These mechanisms make it easy to use DNS along with application-level retries as the foundation for an auto-healing service oriented architecture.

For standard services queries, both A and SRV records are supported. SRV records provide the port that a service is registered on, enabling clients to avoid relying on well-known ports. SRV records are only served if the client specifically requests them, like so:

$ dig @127.0.0.1 -p 8600 consul.service.consul SRV

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 consul.service.consul ANY

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50483

;; flags: qr aa rd; QUERY: 1, ANSWER: 3, AUTHORITY: 1, ADDITIONAL: 1

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;consul.service.consul. IN SRV

;; ANSWER SECTION:

consul.service.consul. 0 IN SRV 1 1 8300 foobar.node.dc1.consul.

;; ADDITIONAL SECTION:

foobar.node.dc1.consul. 0 IN A 10.1.10.12

### [»](https://www.consul.io/docs/agent/dns.html" \l "rfc-2782-lookup) RFC 2782 Lookup

The format for RFC 2782 SRV lookups is:

\_<service>.\_<protocol>[.service][.datacenter][.domain]

Per [RFC 2782](https://tools.ietf.org/html/rfc2782), SRV queries should use underscores, \_, as a prefix to the service and protocol values in a query to prevent DNS collisions. The protocol value can be any of the tags for a service. If the service has no tags, tcp should be used. If tcp is specified as the protocol, the query will not perform any tag filtering.

Other than the query format and default tcp protocol/tag value, the behavior of the RFC style lookup is the same as the standard style of lookup.

If you registered the service rabbitmq on port 5672 and tagged it with amqp, you could make an RFC 2782 query for its SRV record as \_rabbitmq.\_amqp.service.consul:

$ dig @127.0.0.1 -p 8600 \_rabbitmq.\_amqp.service.consul SRV

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 \_rabbitmq.\_amqp.service.consul ANY

; (1 server found)

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52838

;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; WARNING: recursion requested but not available

;; QUESTION SECTION:

;\_rabbitmq.\_amqp.service.consul. IN SRV

;; ANSWER SECTION:

\_rabbitmq.\_amqp.service.consul. 0 IN SRV 1 1 5672 rabbitmq.node1.dc1.consul.

;; ADDITIONAL SECTION:

rabbitmq.node1.dc1.consul. 0 IN A 10.1.11.20

Again, note that the SRV record returns the port of the service as well as its IP.

### [»](https://www.consul.io/docs/agent/dns.html" \l "prepared-query-lookups) Prepared Query Lookups

The format of a prepared query lookup is:

<query or name>.query[.datacenter].<domain>

The datacenter is optional, and if not provided, the datacenter of this Consul agent is assumed.

The query or name is the ID or given name of an existing [Prepared Query](https://www.consul.io/api/query.html). These behave like standard service queries but provide a much richer set of features, such as filtering by multiple tags and automatically failing over to look for services in remote datacenters if no healthy nodes are available in the local datacenter. Consul 0.6.4 and later also added support for [prepared query templates](https://www.consul.io/api/query.html#templates) which can match names using a prefix match, allowing one template to apply to potentially many services.

To allow for simple load balancing, the set of nodes returned is randomized each time. Both A and SRV records are supported. SRV records provide the port that a service is registered on, enabling clients to avoid relying on well-known ports. SRV records are only served if the client specifically requests them.

### [»](https://www.consul.io/docs/agent/dns.html" \l "udp-based-dns-queries) UDP Based DNS Queries

When the DNS query is performed using UDP, Consul will truncate the results without setting the truncate bit. This is to prevent a redundant lookup over TCP that generates additional load. If the lookup is done over TCP, the results are not truncated.

## [»](https://www.consul.io/docs/agent/dns.html" \l "caching) Caching

By default, all DNS results served by Consul set a 0 TTL value. This disables caching of DNS results. However, there are many situations in which caching is desirable for performance and scalability. This is discussed more in the guide for [DNS Caching](https://www.consul.io/docs/guides/dns-cache.html).

## [»](https://www.consul.io/docs/agent/dns.html" \l "wan-address-translation) WAN Address Translation

By default, Consul DNS queries will return a node's local address, even when being queried from a remote datacenter. If you need to use a different address to reach a node from outside its datacenter, you can configure this behavior using the [advertise-wan](https://www.consul.io/docs/agent/options.html#_advertise-wan) and [translate\_wan\_addrs](https://www.consul.io/docs/agent/options.html#translate_wan_addrs) configuration options.

https://www.consul.io/api/index.html

# HTTP API

The main interface to Consul is a RESTful HTTP API. The API can basic perform CRUD operations on nodes, services, checks, configuration, and more.

## [»](https://www.consul.io/api/index.html" \l "version-prefix) Version Prefix

All API routes are prefixed with /v1/.

This documentation is only for the v1 API.

**Backwards compatibility:** At the current version, Consul does not yet promise backwards compatibility even with the v1 prefix. We'll remove this warning when this policy changes. We expect to reach API stability by Consul 1.0.

## [»](https://www.consul.io/api/index.html" \l "acls) ACLs

Several endpoints in Consul use or require ACL tokens to operate. An agent can be configured to use a default token in requests using the acl\_token configuration option. However, the token can also be specified per-request by using the X-Consul-Token request header or the token query string parameter. The request header takes precedence over the default token, and the query string parameter takes precedence over everything.

## [»](https://www.consul.io/api/index.html" \l "authentication) Authentication

When authentication is enabled, a Consul token should be provided to API requests using the X-Consul-Token header. This reduces the probability of the token accidentally getting logged or exposed. When using authentication, clients should communicate via TLS.

Here is an example using curl:

$ curl \

--header "X-Consul-Token: abcd1234" \

https://consul.rocks/v1/agent/members

Previously this was provided via a ?token= query parameter. This functionality exists on many endpoints for backwards compatibility, but its use is **highly discouraged**, since it can show up in access logs as part of the URL.

## [»](https://www.consul.io/api/index.html" \l "blocking-queries) Blocking Queries

Many endpoints in Consul support a feature known as "blocking queries". A blocking query is used to wait for a potential change using long polling. Not all endpoints support blocking, but each endpoint uniquely documents its support for blocking queries in the documentation.

Endpoints that support blocking queries return an HTTP header named X-Consul-Index. This is a unique identifier representing the current state of the requested resource.

On subsequent requests for this resource, the client can set the index query string parameter to the value of X-Consul-Index, indicating that the client wishes to wait for any changes subsequent to that index.

When this is provided, the HTTP request will "hang" until a change in the system occurs, or the maximum timeout is reached. A critical note is that the return of a blocking request is **no guarantee** of a change. It is possible that the timeout was reached or that there was an idempotent write that does not affect the result of the query.

In addition to index, endpoints that support blocking will also honor a wait parameter specifying a maximum duration for the blocking request. This is limited to 10 minutes. If not set, the wait time defaults to 5 minutes. This value can be specified in the form of "10s" or "5m" (i.e., 10 seconds or 5 minutes, respectively). A small random amount of additional wait time is added to the supplied maximum wait time to spread out the wake up time of any concurrent requests. This adds up to wait / 16 additional time to the maximum duration.

## [»](https://www.consul.io/api/index.html" \l "consistency-modes) Consistency Modes

Most of the read query endpoints support multiple levels of consistency. Since no policy will suit all clients' needs, these consistency modes allow the user to have the ultimate say in how to balance the trade-offs inherent in a distributed system.

The three read modes are:

* [default](https://www.consul.io/api/index.html" \l "default) - If not specified, the default is strongly consistent in almost all cases. However, there is a small window in which a new leader may be elected during which the old leader may service stale values. The trade-off is fast reads but potentially stale values. The condition resulting in stale reads is hard to trigger, and most clients should not need to worry about this case. Also, note that this race condition only applies to reads, not writes.

* [consistent](https://www.consul.io/api/index.html" \l "consistent) - This mode is strongly consistent without caveats. It requires that a leader verify with a quorum of peers that it is still leader. This introduces an additional round-trip to all server nodes. The trade-off is increased latency due to an extra round trip. Most clients should not use this unless they cannot tolerate a stale read.

* [stale](https://www.consul.io/api/index.html" \l "stale) - This mode allows any server to service the read regardless of whether it is the leader. This means reads can be arbitrarily stale; however, results are generally consistent to within 50 milliseconds of the leader. The trade-off is very fast and scalable reads with a higher likelihood of stale values. Since this mode allows reads without a leader, a cluster that is unavailable will still be able to respond to queries.

To switch these modes, either the stale or consistent query parameters should be provided on requests. It is an error to provide both.

To support bounding the acceptable staleness of data, responses provide the X-Consul-LastContact header containing the time in milliseconds that a server was last contacted by the leader node. The X-Consul-KnownLeader header also indicates if there is a known leader. These can be used by clients to gauge the staleness of a result and take appropriate action.

## [»](https://www.consul.io/api/index.html" \l "formatted-json-output) Formatted JSON Output

By default, the output of all HTTP API requests is minimized JSON. If the client passes pretty on the query string, formatted JSON will be returned.

## [»](https://www.consul.io/api/index.html" \l "http-methods) HTTP Methods

Consul's API aims to be RESTful, although there are some exceptions. The API responds to the standard HTTP verbs GET, PUT, and DELETE. Each API method will clearly document the verb(s) it responds to and the generated response. The same path with different verbs may trigger different behavior. For example:

PUT /v1/kv/foo

GET /v1/kv/foo

Even though these share a path, the PUT operation creates a new key whereas the GET operation reads an existing key.

Here is the same example using curl:

$ curl \

--request PUT \

--data 'hello consul' \

https://consul.rocks/v1/kv/foo

## [»](https://www.consul.io/api/index.html" \l "translated-addresses) Translated Addresses

Consul 0.7 added the ability to translate addresses in HTTP response based on the configuration setting for [translate\_wan\_addrs](https://www.consul.io/docs/agent/options.html#translate_wan_addrs). In order to allow clients to know if address translation is in effect, the X-Consul-Translate-Addresses header will be added if translation is enabled, and will have a value of true. If translation is not enabled then this header will not be present.

https://www.consul.io/api/agent/check.html

# Check - Agent HTTP API

The /agent/check endpoints interact with checks on the local agent in Consul. These should not be confused with checks in the catalog.

## [»](https://www.consul.io/api/agent/check.html" \l "list-checks) List Checks

This endpoint returns all checks that are registered with the local agent. These checks were either provided through configuration files or added dynamically using the HTTP API.

It is important to note that the checks known by the agent may be different from those reported by the catalog. This is usually due to changes being made while there is no leader elected. The agent performs active [anti-entropy](https://www.consul.io/docs/internals/anti-entropy.html), so in most situations everything will be in sync within a few seconds.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/checks | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:read,service:read |

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request) Sample Request

$ curl \

https://consul.rocks/v1/agent/checks

### [»](https://www.consul.io/api/agent/check.html" \l "sample-response) Sample Response

{

"service:redis": {

"Node": "foobar",

"CheckID": "service:redis",

"Name": "Service 'redis' check",

"Status": "passing",

"Notes": "",

"Output": "",

"ServiceID": "redis",

"ServiceName": "redis"

}

}

## [»](https://www.consul.io/api/agent/check.html" \l "register-check) Register Check

This endpoint adds a new check to the local agent. Checks may be of script, HTTP, TCP, or TTL type. The agent is responsible for managing the status of the check and keeping the Catalog in sync.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/check/register | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write,service:write |

### [»](https://www.consul.io/api/agent/check.html" \l "parameters) Parameters

* [Name](https://www.consul.io/api/agent/check.html" \l "name) (string: <required>) - Specifies the name of the check.

* [ID](https://www.consul.io/api/agent/check.html" \l "id) (string: "") - Specifies a unique ID for this check in the cluster. This defaults to the "Name" parameter, but it may be necessary to provide an ID for uniqueness.

* [Interval](https://www.consul.io/api/agent/check.html" \l "interval) (string: "") - Specifies the frequency at which to run this check. This is required for HTTP and TCP checks.

* [Notes](https://www.consul.io/api/agent/check.html" \l "notes) (string: "") - Specifies arbitrary information for humans. This is not used by Consul internally.

* [DeregisterCriticalServiceAfter](https://www.consul.io/api/agent/check.html" \l "deregistercriticalserviceafter) (string: "") - Specifies that checks associated with a service should deregister after this time. This is specified as a time duration with suffix like "10m". If a check is in the critical state for more than this configured value, then its associated service (and all of its associated checks) will automatically be deregistered. The minimum timeout is 1 minute, and the process that reaps critical services runs every 30 seconds, so it may take slightly longer than the configured timeout to trigger the deregistration. This should generally be configured with a timeout that's much, much longer than any expected recoverable outage for the given service.

* [Script](https://www.consul.io/api/agent/check.html" \l "script) (string: "") - Specifies a script or path to a script to run on Interval to update the status of the check. If specifying a path, this path must exist on disk and be readable by the Consul agent.

* [DockerContainerID](https://www.consul.io/api/agent/check.html" \l "dockercontainerid) (string: "") - Specifies that the check is a Docker check, and Consul will evaluate the script every Interval in the given container using the specified Shell. Note that Shell is currently only supported for Docker checks.

* [HTTP](https://www.consul.io/api/agent/check.html" \l "http) (string: "") - Specifies an HTTP check to perform a GET request against the value of HTTP (expected to be a URL) every Interval. If the response is any 2xx code, the check is passing. If the response is 429 Too Many Requests, the check is warning. Otherwise, the check is critical. HTTP checks also support SSL. By default, a valid SSL certificate is expected. Certificate verification can be controlled using the TLSSkipVerify.

* [TLSSkipVerify](https://www.consul.io/api/agent/check.html" \l "tlsskipverify) (bool: false) - Specifies if the certificate for an HTTPS check should not be verified.

* [TCP](https://www.consul.io/api/agent/check.html" \l "tcp) (string: "") - Specifies a TCP to connect against the value of TCP (expected to be an IP or hostname plus port combination) every Interval. If the connection attempt is successful, the check is passing. If the connection attempt is unsuccessful, the check is critical. In the case of a hostname that resolves to both IPv4 and IPv6 addresses, an attempt will be made to both addresses, and the first successful connection attempt will result in a successful check.

* [TTL](https://www.consul.io/api/agent/check.html" \l "ttl) (string: "") - Specifies this is a TTL check, and the TTL endpoint must be used periodically to update the state of the check.

* [ServiceID](https://www.consul.io/api/agent/check.html" \l "serviceid) (string: "") - Specifies the ID of a service to associate the registered check with an existing service provided by the agent.

* [Status](https://www.consul.io/api/agent/check.html" \l "status) (string: "") - Specifies the initial status of the health check.

### [»](https://www.consul.io/api/agent/check.html" \l "sample-payload) Sample Payload

{

"ID": "mem",

"Name": "Memory utilization",

"Notes": "Ensure we don't oversubscribe memory",

"DeregisterCriticalServiceAfter": "90m",

"Script": "/usr/local/bin/check\_mem.py",

"DockerContainerID": "f972c95ebf0e",

"Shell": "/bin/bash",

"HTTP": "http://example.com",

"TCP": "example.com:22",

"Interval": "10s",

"TTL": "15s",

"TLSSkipVerify": true

}

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request-1) Sample Request

$ curl \

--request PUT \

--data @payload.json \

https://consul.rocks/v1/agent/check/register

## [»](https://www.consul.io/api/agent/check.html" \l "deregister-check) Deregister Check

This endpoint remove a check from the local agent. The agent will take care of deregistering the check from the catalog. If the check with the provided ID does not exist, no action is taken.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/check/deregister/:check\_id | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write,service:write |

### [»](https://www.consul.io/api/agent/check.html" \l "parameters-1) Parameters

* [check\_id](https://www.consul.io/api/agent/check.html" \l "check_id) (string: "") - Specifies the unique ID of the check to deregister. This is specified as part of the URL.

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request-2) Sample Request

$ curl \

--request PUT \

https://consul.rocks/v1/agent/check/deregister/my-check-id

## [»](https://www.consul.io/api/agent/check.html" \l "ttl-check-pass) TTL Check Pass

This endpoint is used with a TTL type check to set the status of the check to passing and to reset the TTL clock.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/check/pass/:check\_id | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write,service:write |

### [»](https://www.consul.io/api/agent/check.html" \l "parameters-2) Parameters

* [check\_id](https://www.consul.io/api/agent/check.html" \l "check_id-1) (string: "") - Specifies the unique ID of the check to use. This is specified as part of the URL.

* [note](https://www.consul.io/api/agent/check.html" \l "note) (string: "") - Specifies a human-readable message. This will be passed through to the check's Output field.

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request-3) Sample Request

$ curl \

https://consul.rocks/v1/agent/check/pass/my-check-id

## [»](https://www.consul.io/api/agent/check.html" \l "ttl-check-warn) TTL Check Warn

This endpoint is used with a TTL type check to set the status of the check to warning and to reset the TTL clock.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/check/warn/:check\_id | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write,service:write |

### [»](https://www.consul.io/api/agent/check.html" \l "parameters-3) Parameters

* [check\_id](https://www.consul.io/api/agent/check.html" \l "check_id-2) (string: "") - Specifies the unique ID of the check to use. This is specified as part of the URL.

* [note](https://www.consul.io/api/agent/check.html" \l "note-1) (string: "") - Specifies a human-readable message. This will be passed through to the check's Output field.

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request-4) Sample Request

$ curl \

https://consul.rocks/v1/agent/check/warn/my-check-id

## [»](https://www.consul.io/api/agent/check.html" \l "ttl-check-fail) TTL Check Fail

This endpoint is used with a TTL type check to set the status of the check to critical and to reset the TTL clock.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/check/fail/:check\_id | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write,service:write |

### [»](https://www.consul.io/api/agent/check.html" \l "parameters-4) Parameters

* [check\_id](https://www.consul.io/api/agent/check.html" \l "check_id-3) (string: "") - Specifies the unique ID of the check to use. This is specified as part of the URL.

* [note](https://www.consul.io/api/agent/check.html" \l "note-2) (string: "") - Specifies a human-readable message. This will be passed through to the check's Output field.

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request-5) Sample Request

$ curl \

https://consul.rocks/v1/agent/check/fail/my-check-id

## [»](https://www.consul.io/api/agent/check.html" \l "ttl-check-update) TTL Check Update

This endpoint is used with a TTL type check to set the status of the check and to reset the TTL clock.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/check/update/:check\_id | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write,service:write |

### [»](https://www.consul.io/api/agent/check.html" \l "parameters-5) Parameters

* [check\_id](https://www.consul.io/api/agent/check.html" \l "check_id-4) (string: "") - Specifies the unique ID of the check to use. This is specified as part of the URL.

* [Status](https://www.consul.io/api/agent/check.html" \l "status-1) (string: "") - Specifies the status of the check. Valid values are "passing", "warning", and "critical".

* [Output](https://www.consul.io/api/agent/check.html" \l "output) (string: "") - Specifies a human-readable message. This will be passed through to the check's Output field.

### [»](https://www.consul.io/api/agent/check.html" \l "sample-payload-1) Sample Payload

{

"Status": "passing",

"Output": "curl reported a failure:\n\n..."

}

### [»](https://www.consul.io/api/agent/check.html" \l "sample-request-6) Sample Request

$ curl \

--request PUT \

--data @payload.json \

https://consul.rocks/v1/agent/check/update/my-check-id

https://www.consul.io/api/agent.html

# Agent HTTP API

The /agent endpoints are used to interact with the local Consul agent. Usually, services and checks are registered with an agent which then takes on the burden of keeping that data synchronized with the cluster. For example, the agent registers services and checks with the Catalog and performs [anti-entropy](https://www.consul.io/docs/internals/anti-entropy.html) to recover from outages.

In addition to these endpoints, additional endpoints are grouped in the navigation for Checks and Services.

## [»](https://www.consul.io/api/agent.html" \l "list-members) List Members

This endpoint returns the members the agent sees in the cluster gossip pool. Due to the nature of gossip, this is eventually consistent: the results may differ by agent. The strongly consistent view of nodes is instead provided by /v1/catalog/nodes.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/members | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:read |

### [»](https://www.consul.io/api/agent.html#parameters) Parameters

* [wan](https://www.consul.io/api/agent.html" \l "wan) (bool: false) - Specifies to list WAN members instead of the LAN members (which is the default). This is only eligible for agents running in **server mode**. This is specified as part of the URL as a query parameter.

### [»](https://www.consul.io/api/agent.html#sample-request) Sample Request

$ curl \

https://consul.rocks/v1/agent/members

### [»](https://www.consul.io/api/agent.html#sample-response) Sample Response

[

{

"Name": "foobar",

"Addr": "10.1.10.12",

"Port": 8301,

"Tags": {

"bootstrap": "1",

"dc": "dc1",

"port": "8300",

"role": "consul"

},

"Status": 1,

"ProtocolMin": 1,

"ProtocolMax": 2,

"ProtocolCur": 2,

"DelegateMin": 1,

"DelegateMax": 3,

"DelegateCur": 3

}

]

## [»](https://www.consul.io/api/agent.html" \l "read-configuration) Read Configuration

This endpoint returns the configuration and member information of the local agent.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/self | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | agent:read |

### [»](https://www.consul.io/api/agent.html#sample-request-1) Sample Request

$ curl \

https://consul.rocks/v1/agent/self

### [»](https://www.consul.io/api/agent.html" \l "sample-response-1) Sample Response

{

"Config": {

"Bootstrap": true,

"Server": true,

"Datacenter": "dc1",

"DataDir": "/tmp/consul",

"DNSRecursor": "",

"DNSRecursors": [],

"Domain": "consul.",

"LogLevel": "INFO",

"NodeID": "40e4a748-2192-161a-0510-9bf59fe950b5",

"NodeName": "foobar",

"ClientAddr": "127.0.0.1",

"BindAddr": "0.0.0.0",

"AdvertiseAddr": "10.1.10.12",

"Ports": {

"DNS": 8600,

"HTTP": 8500,

"RPC": 8400,

"SerfLan": 8301,

"SerfWan": 8302,

"Server": 8300

},

"LeaveOnTerm": false,

"SkipLeaveOnInt": false,

"StatsiteAddr": "",

"Protocol": 1,

"EnableDebug": false,

"VerifyIncoming": false,

"VerifyOutgoing": false,

"CAFile": "",

"CertFile": "",

"KeyFile": "",

"StartJoin": [],

"UiDir": "",

"PidFile": "",

"EnableSyslog": false,

"RejoinAfterLeave": false

},

"Coord": {

"Adjustment": 0,

"Error": 1.5,

"Vec": [0,0,0,0,0,0,0,0]

},

"Member": {

"Name": "foobar",

"Addr": "10.1.10.12",

"Port": 8301,

"Tags": {

"bootstrap": "1",

"dc": "dc1",

"id": "40e4a748-2192-161a-0510-9bf59fe950b5",

"port": "8300",

"role": "consul",

"vsn": "1",

"vsn\_max": "1",

"vsn\_min": "1"

},

"Status": 1,

"ProtocolMin": 1,

"ProtocolMax": 2,

"ProtocolCur": 2,

"DelegateMin": 2,

"DelegateMax": 4,

"DelegateCur": 4

},

"Meta": {

"instance\_type": "i2.xlarge",

"os\_version": "ubuntu\_16.04"

}

}

## [»](https://www.consul.io/api/agent.html" \l "reload-agent) Reload Agent

This endpoint instructs the agent to reload its configuration. Any errors encountered during this process are returned.

Not all configuration options are reloadable. See the [Reloadable Configuration](https://www.consul.io/docs/agent/options.html#reloadable-configuration) section on the agent options page for details on which options are supported.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/reload | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | agent:write |

### [»](https://www.consul.io/api/agent.html#sample-request-2) Sample Request

$ curl \

--request PUT \

https://consul.rocks/v1/agent/reload

## [»](https://www.consul.io/api/agent.html" \l "enable-maintenance-mode) Enable Maintenance Mode

This endpoint places the agent into "maintenance mode". During maintenance mode, the node will be marked as unavailable and will not be present in DNS or API queries. This API call is idempotent.

Maintenance mode is persistent and will be automatically restored on agent restart.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/maintenance | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | node:write |

### [»](https://www.consul.io/api/agent.html#parameters-1) Parameters

* [enable](https://www.consul.io/api/agent.html" \l "enable) (bool: <required>) - Specifies whether to enable or disable maintenance mode. This is specified as part of the URL as a query string parameter.

* [reason](https://www.consul.io/api/agent.html" \l "reason) (string: "") - Specifies a text string explaining the reason for placing the node into maintenance mode. This is simply to aid human operators. If no reason is provided, a default value will be used instead. This is specified as part of the URL as a query string parameter, and, as such, must be URI-encoded.

### [»](https://www.consul.io/api/agent.html#sample-request-3) Sample Request

$ curl \

--request PUT \

https://consul.rocks/v1/agent/maintenance?enable=true&reason=For+API+docs

## [»](https://www.consul.io/api/agent.html" \l "stream-logs) Stream Logs

This endpoint streams logs from the local agent until the connection is closed.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/monitor | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | agent:read |

### [»](https://www.consul.io/api/agent.html#parameters-2) Parameters

* [loglevel](https://www.consul.io/api/agent.html" \l "loglevel) (string: "info") - Specifies a text string containing a log level to filter on, such as info.

### [»](https://www.consul.io/api/agent.html#sample-request-4) Sample Request

$ curl \

https://consul.rocks/v1/agent/monitor

### [»](https://www.consul.io/api/agent.html" \l "sample-response-2) Sample Response

YYYY/MM/DD HH:MM:SS [INFO] raft: Initial configuration (index=1): [{Suffrage:Voter ID:127.0.0.1:8300 Address:127.0.0.1:8300}]

YYYY/MM/DD HH:MM:SS [INFO] raft: Node at 127.0.0.1:8300 [Follower] entering Follower state (Leader: "")

YYYY/MM/DD HH:MM:SS [INFO] serf: EventMemberJoin: machine-osx 127.0.0.1

YYYY/MM/DD HH:MM:SS [INFO] consul: Adding LAN server machine-osx (Addr: tcp/127.0.0.1:8300) (DC: dc1)

YYYY/MM/DD HH:MM:SS [INFO] serf: EventMemberJoin: machine-osx.dc1 127.0.0.1

YYYY/MM/DD HH:MM:SS [INFO] consul: Handled member-join event for server "machine-osx.dc1" in area "wan"

# ...

## [»](https://www.consul.io/api/agent.html" \l "join-agent) Join Agent

This endpoint instructs the agent to attempt to connect to a given address.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| GET | /agent/join/:address | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | agent:write |

### [»](https://www.consul.io/api/agent.html#parameters-3) Parameters

* [address](https://www.consul.io/api/agent.html" \l "address) (string: <required>) - Specifies the address of the other agent to join. This is specified as part of the URL.

* [wan](https://www.consul.io/api/agent.html" \l "wan-1) (bool: false) - Specifies to try and join over the WAN pool. This is only optional for agents running in server mode. This is specified as part of the URL as a query parameter

### [»](https://www.consul.io/api/agent.html#sample-request-5) Sample Request

$ curl \

https://consul.rocks/v1/agent/join/1.2.3.4

## [»](https://www.consul.io/api/agent.html" \l "graceful-leave-and-shutdown) Graceful Leave and Shutdown

This endpoint triggers a graceful leave and shutdown of the agent. It is used to ensure other nodes see the agent as "left" instead of "failed". Nodes that leave will not attempt to re-join the cluster on restarting with a snapshot.

For nodes in server mode, the node is removed from the Raft peer set in a graceful manner. This is critical, as in certain situations a non-graceful leave can affect cluster availability.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/leave | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | agent:write |

### [»](https://www.consul.io/api/agent.html#sample-request-6) Sample Request

$ curl \

--request PUT \

https://consul.rocks/v1/agent/leave

## [»](https://www.consul.io/api/agent.html" \l "force-leave-and-shutdown) Force Leave and Shutdown

This endpoint instructs the agent to force a node into the left state. If a node fails unexpectedly, then it will be in a failed state. Once in the failed state, Consul will attempt to reconnect, and the services and checks belonging to that node will not be cleaned up. Forcing a node into the left state allows its old entries to be removed.

| **Method** | **Path** | **Produces** |
| --- | --- | --- |
| PUT | /agent/force-leave | application/json |

The table below shows this endpoint's support for [blocking queries](https://www.consul.io/api/index.html#blocking-queries), [consistency modes](https://www.consul.io/api/index.html#consistency-modes), and [required ACLs](https://www.consul.io/api/index.html#acls).

| **Blocking Queries** | **Consistency Modes** | **ACL Required** |
| --- | --- | --- |
| NO | none | agent:write |

### [»](https://www.consul.io/api/agent.html" \l "sample-request-7) Sample Request

$ curl \

--request PUT \

https://consul.rocks/v1/agent/force-leave

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

[microservices](https://www.nginx.com/blog/tag/microservices/), [service discovery](https://www.nginx.com/blog/tag/service-discovery/), [etcd](https://www.nginx.com/blog/tag/etcd/), [DevOps](https://www.nginx.com/blog/tag/devops/), [Netflix](https://www.nginx.com/blog/tag/netflix/), [Amazon Web Services (AWS)](https://www.nginx.com/blog/tag/amazon-web-services-aws/), [ZooKeeper](https://www.nginx.com/blog/tag/zookeeper/), [Consul](https://www.nginx.com/blog/tag/consul/)

# Service Discovery in a Microservices Architecture

1. [*ntroduction to Microservices*](https://www.nginx.com/blog/introduction-to-microservices/)
2. [*Building Microservices: Using an API Gateway*](https://www.nginx.com/blog/building-microservices-using-an-api-gateway/)
3. [*Building Microservices: Inter-Process Communication in a Microservices Architecture*](https://www.nginx.com/blog/building-microservices-inter-process-communication/)
4. *Service Discovery in a Microservices Architecture (this article)*
5. [*Event-Driven Data Management for Microservices*](https://www.nginx.com/blog/event-driven-data-management-microservices/)
6. [*Choosing a Microservices Deployment Strategy*](https://www.nginx.com/blog/deploying-microservices/)
7. [*Refactoring a Monolith into Microservices*](https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/)

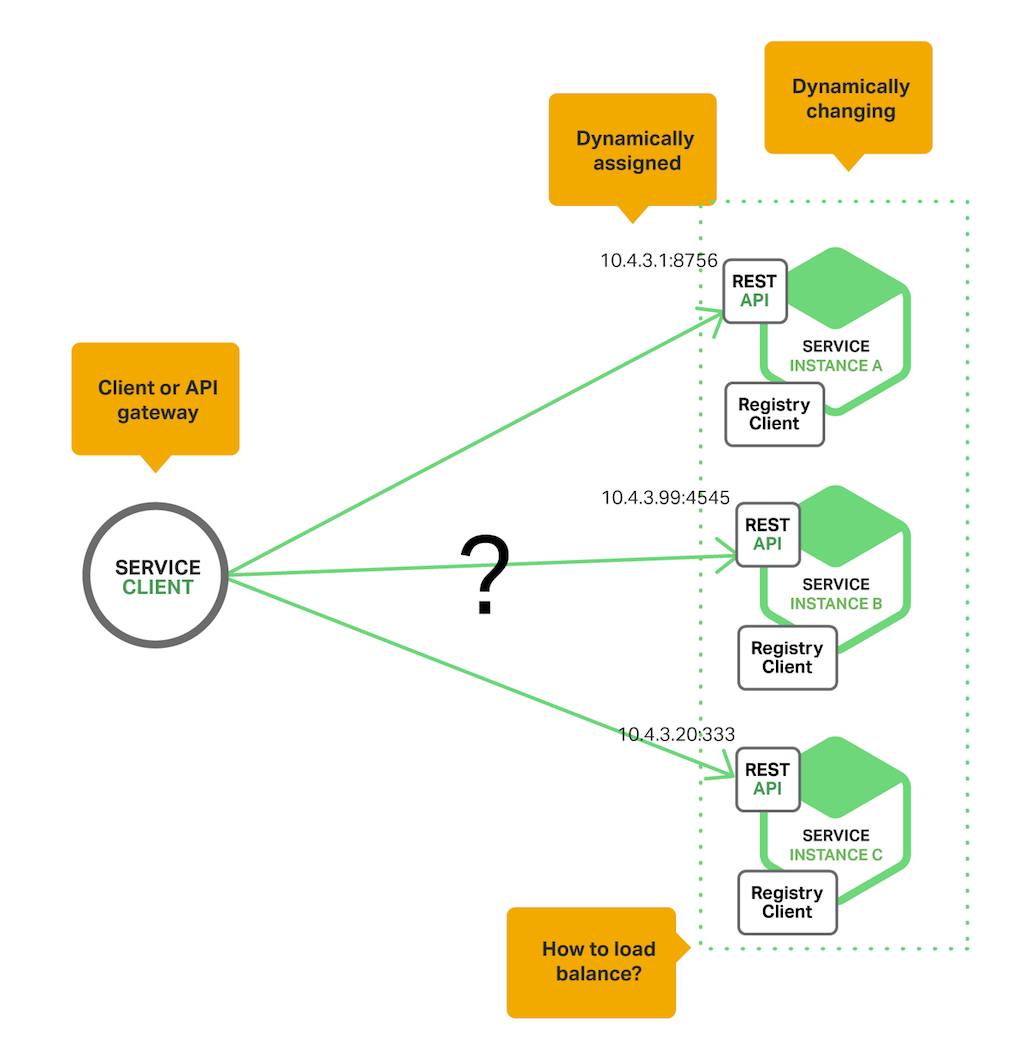
You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – [*Microservices: From Design to Deployment*](https://www.nginx.com/resources/library/designing-deploying-microservices/). Also, please look at the new [Microservices Solutions page](https://www.nginx.com/solutions/microservices/#version_b).

This is the fourth article in our series about building applications with microservices. The [first article](https://www.nginx.com/blog/introduction-to-microservices/) introduces the [Microservices Architecture pattern](http://microservices.io/patterns/microservices.html) and discussed the benefits and drawbacks of using microservices. The [second](https://www.nginx.com/blog/building-microservices-using-an-api-gateway) and [third](https://www.nginx.com/blog/building-microservices-inter-process-communication/) articles in the series describe different aspects of communication within a microservices architecture. In this article, we explore the closely related problem of service discovery.

## Why Use Service Discovery?

Let’s imagine that you are writing some code that invokes a service that has a REST API or Thrift 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.

In a modern, cloud‑based microservices application, however, this is a much more difficult problem to solve as shown in the following diagram.



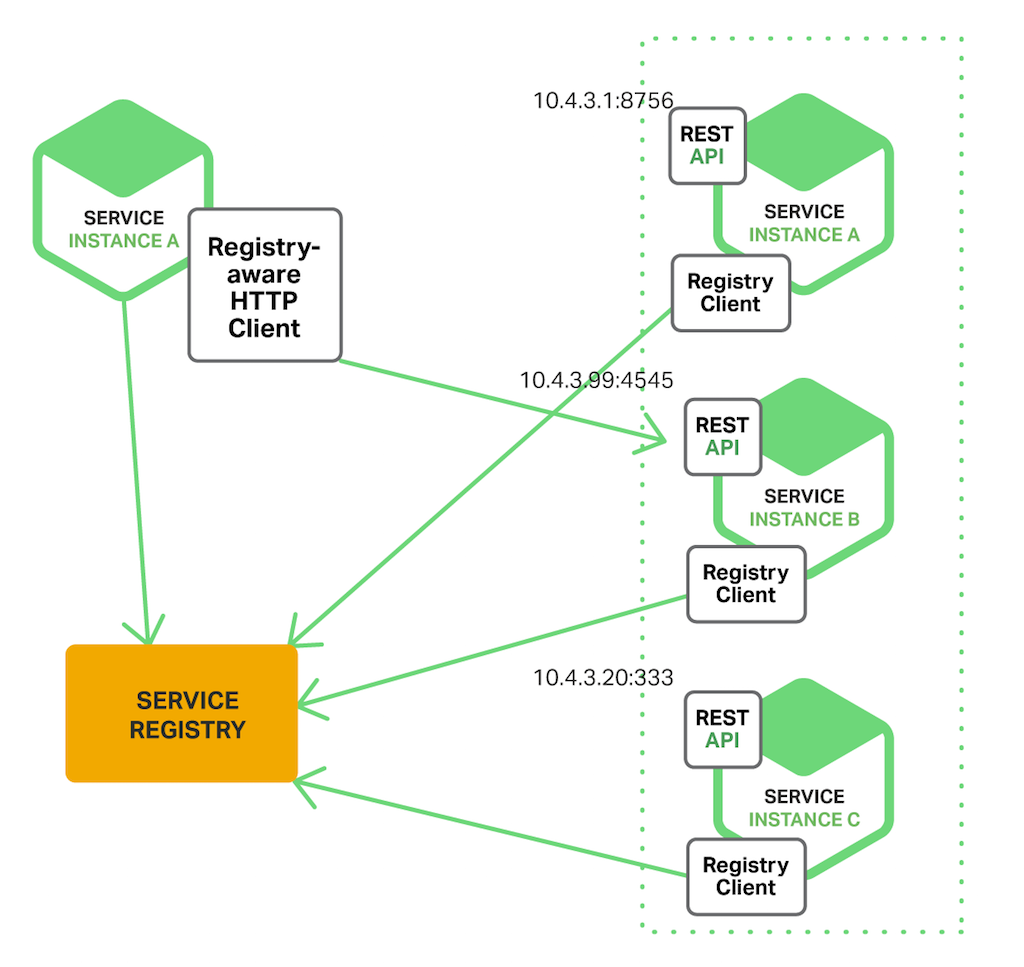
Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

There are two main service discovery patterns: [client‑side discovery](http://microservices.io/patterns/client-side-discovery.html) and [server‑side discovery](http://microservices.io/patterns/server-side-discovery.html). Let’s first look at client‑side discovery.

### The Client‑Side Discovery Pattern

When using [client‑side discovery](http://microservices.io/patterns/client-side-discovery.html), the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load‑balancing algorithm to select one of the available service instances and makes a request.

The following diagram shows the structure of this pattern.



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.

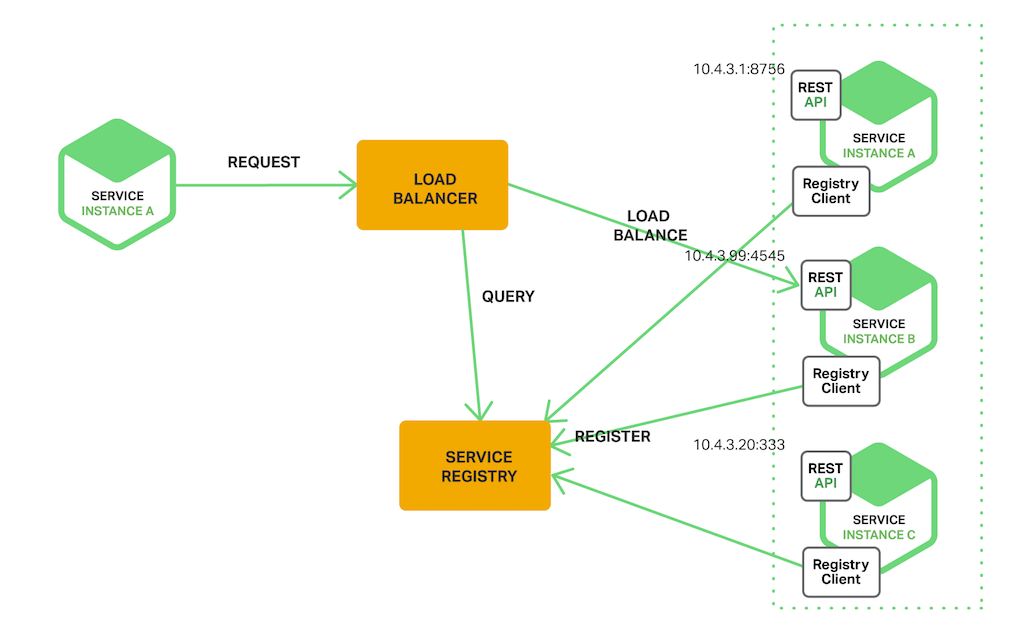
[Netflix OSS](https://netflix.github.io/) provides a great example of the client‑side discovery pattern. [Netflix Eureka](https://github.com/Netflix/eureka) is a service registry. It provides a REST API for managing service‑instance registration and for querying available instances. [Netflix Ribbon](https://github.com/Netflix/ribbon) is an IPC client that works with Eureka to load balance requests across the available service instances. We will discuss Eureka in more depth later in this article.

The client‑side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application‑specific load‑balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client‑side service discovery logic for each programming language and framework used by your service clients.

Now that we have looked at client‑side discovery, let’s take a look at server‑side discovery.

### The Server‑Side Discovery Pattern

The other approach to service discovery is the [server-side discovery pattern](http://microservices.io/patterns/server-side-discovery.html). The following diagram shows the structure of this pattern.



The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client‑side discovery, service instances are registered and deregistered with the service registry.

The [AWS Elastic Load Balancer](https://aws.amazon.com/elasticloadbalancing/) (ELB) is an example of a server-side discovery router. An ELB is commonly used to load balance external traffic from the Internet. However, you can also use an ELB to load balance traffic that is internal to a virtual private cloud (VPC). A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn’t a separate service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

HTTP servers and load balancers such as [NGINX Plus](https://www.nginx.com/products/) and NGINX can also be used as a server-side discovery load balancer. For example, this [blog post](https://www.airpair.com/scalable-architecture-with-docker-consul-and-nginx) describes using [Consul Template](https://github.com/hashicorp/consul-template) to dynamically reconfigure NGINX reverse proxying. Consul Template is a tool that periodically regenerates arbitrary configuration files from configuration data stored in the [Consul service registry](https://www.consul.io/). It runs an arbitrary shell command whenever the files change. In the example described by the blog post, Consul Template generates an **nginx.conf** file, which configures the reverse proxying, and then runs a command that tells NGINX to reload the configuration. A more sophisticated implementation could dynamically reconfigure NGINX Plus using either [its HTTP API or DNS](https://www.nginx.com/products/on-the-fly-reconfiguration/).

Some deployment environments such as [Kubernetes](https://github.com/kubernetes/kubernetes/blob/master/docs/design/architecture.md) and [Marathon](https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html) run a proxy on each host in the cluster. The proxy plays the role of a server‑side discovery load balancer. In order to make a request to a service, a client routes the request via the proxy using the host’s IP address and the service’s assigned port. The proxy then transparently forwards the request to an available service instance running somewhere in the cluster.

The server‑side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. This pattern also has some drawbacks, however. Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.

## The Service Registry

The [service registry](http://microservices.io/patterns/service-registry.html) is a key part of service discovery. It is a database containing the network locations of service instances. A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry. However, that information eventually becomes out of date and clients become unable to discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

As mentioned earlier, [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.

[Netflix achieves high availability](https://github.com/Netflix/eureka/wiki/Configuring-Eureka-in-AWS-Cloud) by running one or more Eureka servers in each Amazon EC2 availability zone. Each Eureka server runs on an EC2 instance that has an [Elastic IP address](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html). DNS TEXT records are used to store the Eureka cluster configuration, which is a map from availability zones to a list of the network locations of Eureka servers. When a Eureka server starts up, it queries DNS to retrieve the Eureka cluster configuration, locates its peers, and assigns itself an unused Elastic IP address.

Eureka clients – services and service clients – query DNS to discover the network locations of Eureka servers. Clients prefer to use a Eureka server in the same availability zone. However, if none is available, the client uses a Eureka server in another availability zone.

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.

Also, as noted previously, 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.

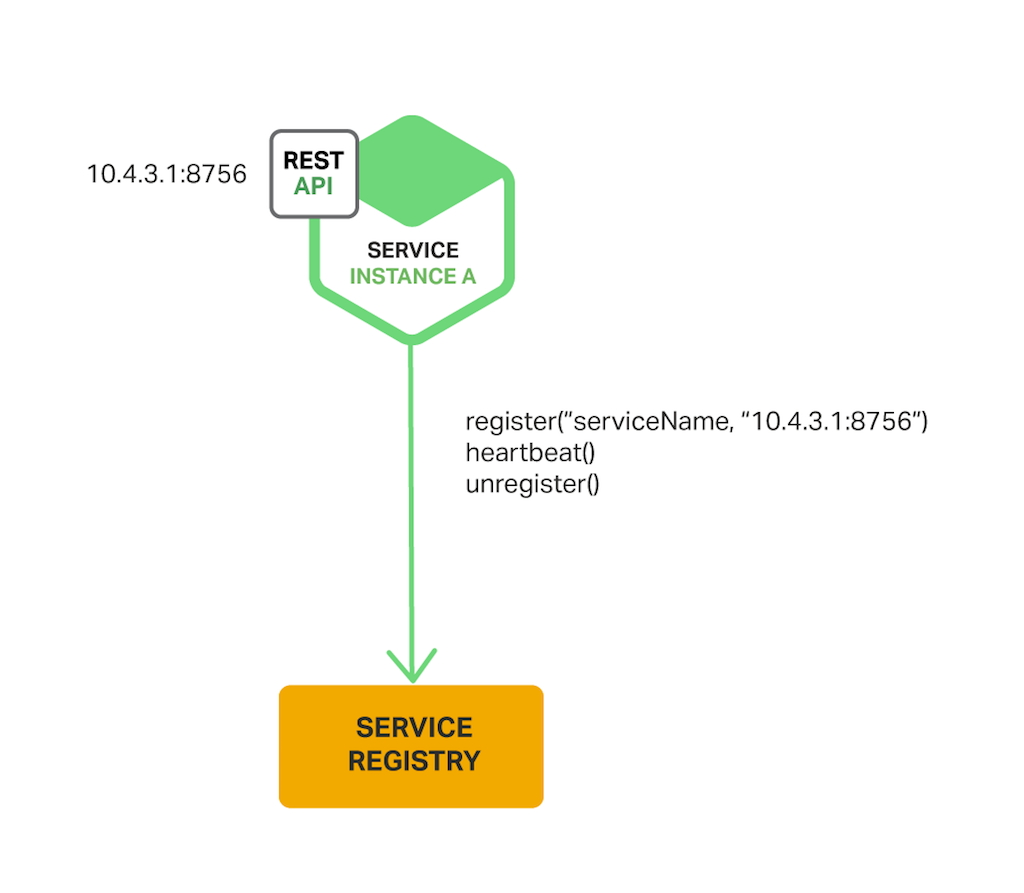
Now that we have looked at the concept of a service registry, let’s look at how service instances are registered with the service registry.

## Service Registration Options

As previously mentioned, service instances must be registered with and deregistered from the service registry. There are a couple of different ways to handle the registration and deregistration. One option is for service instances to register themselves, the [self‑registration pattern](http://microservices.io/patterns/self-registration.html). The other option is for some other system component to manage the registration of service instances, the [third‑party registration pattern](http://microservices.io/patterns/3rd-party-registration.html). Let’s first look at the self‑registration pattern.

### The Self‑Registration Pattern

When using the [self‑registration pattern](http://microservices.io/patterns/self-registration.html), a service instance is responsible for registering and deregistering itself with the service registry. Also, if required, a service instance sends heartbeat requests to prevent its registration from expiring. The following diagram shows the structure of this pattern.



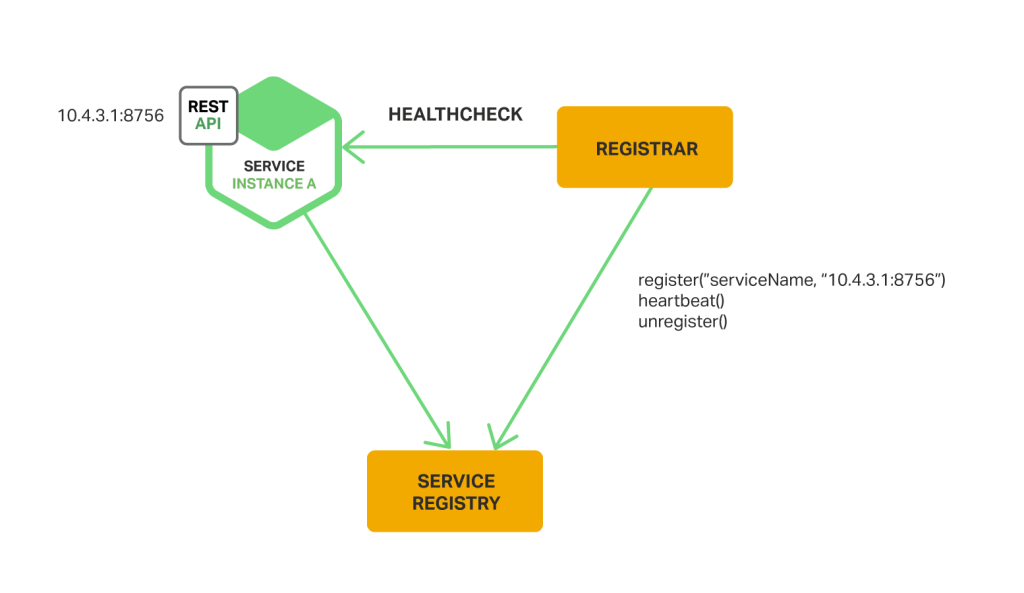
A good example of this approach is the [Netflix OSS Eureka client](https://github.com/Netflix/eureka). The Eureka client handles all aspects of service instance registration and deregistration. The [Spring Cloud project](http://projects.spring.io/spring-cloud/), which implements various patterns including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

The self‑registration pattern has various benefits and drawbacks. One benefit is that it is relatively simple and doesn’t require any other system components. However, a major drawback is that it couples the service instances to the service registry. You must implement the registration code in each programming language and framework used by your services.

The alternative approach, which decouples services from the service registry, is the third‑party registration pattern.

### The Third‑Party Registration Pattern

When using the [third-party registration pattern](http://microservices.io/patterns/3rd-party-registration.html), 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. Registrator supports several service registries, including etcd and Consul.

Another example of a service registrar is [NetflixOSS Prana](https://github.com/netflix/Prana). Primarily intended for services written in non‑JVM languages, it is a sidecar application that runs side by side with a service instance. Prana registers and deregisters the service instance with Netflix Eureka.

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.

The third‑party registration pattern has various benefits and drawbacks. A major benefit is that services are decoupled from the service registry. You don’t need to implement service‑registration logic for each programming language and framework used by your developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

One drawback of this pattern is that unless it’s built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

## Summary

In a microservices application, the set of running service instances changes dynamically. Instances have dynamically assigned network locations. Consequently, in order for a client to make a request to a service it must use a service‑discovery mechanism.

A key part of service discovery is the [service registry](http://microservices.io/patterns/service-registry.html). The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and deregistered from the service registry using the management API. The query API is used by system components to discover available service instances.

There are two main service‑discovery patterns: client-side discovery and service-side discovery. In systems that use [client‑side service discovery](http://microservices.io/patterns/client-side-discovery.html), clients query the service registry, select an available instance, and make a request. In systems that use [server‑side discovery](http://microservices.io/patterns/server-side-discovery.html), clients make requests via a router, which queries the service registry and forwards the request to an available instance.

There are two main ways that service instances are registered with and deregistered from the service registry. One option is for service instances to register themselves with the service registry, the [self‑registration pattern](http://microservices.io/patterns/self-registration.html). The other option is for some other system component to handle the registration and deregistration on behalf of the service, the [third‑party registration pattern](http://microservices.io/patterns/3rd-party-registration.html).

In some deployment environments you need to set up your own service‑discovery infrastructure using a service registry such as [Netflix Eureka](https://github.com/Netflix/eureka), [etcd](https://github.com/coreos/etcd), or [Apache Zookeeper](http://zookeeper.apache.org/). In other deployment environments, service discovery is built in. For example, [Kubernetes](https://github.com/kubernetes/kubernetes/blob/master/docs/design/architecture.md) and [Marathon](https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html) handle service instance registration and deregistration. They also run a proxy on each cluster host that plays the role of [server‑side discovery](http://microservices.io/patterns/server-side-discovery.html) router.

An HTTP reverse proxy and load balancer such as NGINX can also be used as a server‑side discovery load balancer. The service registry can push the routing information to NGINX and invoke a graceful configuration update; for example, you can use [Consul Template](https://hashicorp.com/blog/introducing-consul-template.html). NGINX Plus supports [additional dynamic reconfiguration mechanisms](https://www.nginx.com/products/on-the-fly-reconfiguration/) – it can pull information about service instances from the registry using DNS, and it provides an API for remote reconfiguration.

In future blog posts, we’ll continue to dive into other aspects of microservices. Sign up to the NGINX mailing list (form is below) to be notified of the release of future articles in the series.

***Editor*** – This seven‑part series of articles is now complete:

1. [*Introduction to Microservices*](https://www.nginx.com/blog/introduction-to-microservices/)
2. [*Building Microservices: Using an API Gateway*](https://www.nginx.com/blog/building-microservices-using-an-api-gateway/)
3. [*Building Microservices: Inter-Process Communication in a Microservices Architecture*](https://www.nginx.com/blog/building-microservices-inter-process-communication/)
4. *Service Discovery in a Microservices Architecture (this article)*
5. [*Event-Driven Data Management for Microservices*](https://www.nginx.com/blog/event-driven-data-management-microservices/)
6. [*Choosing a Microservices Deployment Strategy*](https://www.nginx.com/blog/deploying-microservices/)
7. [*Refactoring a Monolith into Microservices*](https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/)

*You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook –* [*Microservices: From Design to Deployment*](https://www.nginx.com/resources/library/designing-deploying-microservices/)*.*

Guest blogger Chris Richardson is the founder of the original [*CloudFoundry.com*](http://cloudfoundry.com/), an early Java PaaS (Platform as a Service) for Amazon EC2. He now consults with organizations to improve how they develop and deploy applications. He also blogs regularly about microservices at [*http://microservices.io*](http://microservices.io/).

http://daviddawson.me/getting/started/with/microservices/2015/06/10/service-discovery-overview.html

|  |
| --- |
| This article was originally posted at simplicityitself.com.  Simplicity Itself has now closed, and so I have moved my articles here.  If you would like to read up on why it closed - See [here](http://daviddawson.me/blog/2017/02/04/simplicity-itself-shutdown.html)  I now operate as a freelancer, specialising in Microservices, event systems, cloud foundry, IoT in many languages.  I run the [Muon microservices platform project](http://muoncore.io/) and the [London Microservices User Group](http://meetup.com/London-Microservices-User-Group/)  I’m [available to hire](http://daviddawson.me/hireme.html) for small and large projects |

When building microservices, you have to naturally distribute your application around a network. It is almost always the case that you are building in a cloud environment, and often using *immutable infrastructure*. Ironically, this means that your virtual machines or containers are created and destroyed much more often than normal, as this immutable nature means that you don’t maintain them.

These properties together mean that your services need to be reconfigured with the location of the other services they need to connect to. This reconfiguration needs to be able to happen on the fly, so that when a new service instance is created, the rest of the network can automatically find it and start to communication with it.  This process is called *Service Discovery*.

This concept is one of the key underpinnings of a Microservice architecture. Attempting to create Microservices without a service discovery system will lead to pain and misery as you will, in effect, be working as a manual replacement.

As well as the standalone solutions presented here, most platforms, whether full PaaS or the more minimal container managers, have some form of Service Discovery.

Which system to choose?

There are currently several key contenders to choose from, ZooKeeper, Consul, Etcd, Eureka and RollYourOwn

**ZooKeeper**

[*http://zookeeper.apache.org/*](http://zookeeper.apache.org/)

ZooKeeper is an Apache project providing a distributed, eventually consistent hierarchical configuration store.

ZooKeeper originated out of the world of Hadoop, where it was built to help in the maintenance of the various components in a hadoop cluster. It is not a service discovery system per se, but is instead a distributed configuration store that provides notifications to registered clients.  With this, it is possible to build a service discovery infrastructure, however every service must explicitly register with ZooKeeper, and the clients must then check in the configuration.

Netflix have invested a lot of time and resources into ZooKeeper, and so a significant amount of Netflix OSS projects have some ZooKeeper integration.

**I Recommend**

ZooKeeper is a well understood clustered system. It is a consistent configuration store, and so being well designed and built, a network partition will cause the smaller side of the partition to shut down. For that reason, you must choose whether consistency or availability is more important to you.

If you do choose a consistent system for service discovery, such as Zookeeper, then you need to understand the implications on your services. You have tied them to the lifecycle of the discovery system, and also exposed them to any failure conditions it may have. You should not assume that 'consistent' means free from failure. Zookeeper is among the older cluster managers, and consensus (pun intended) is that it’s implementation of master selection is robust and well behaved.

If you do choose to use ZooKeeper, investigate the Netflix OSS projects, staring with <a title="Curator" href="https://github.com/Netflix/curator" target="\_blank">Curator</a> as a first point of call, and only use bare ZooKeeper if they don’t fit your needs.

Since Zookeeper is mature and established, there is a large ecosystem of good quality (mostly!) clients and libraries to enrich your projects.

**Consul**

[*http://www.consul.io*](http://www.consul.io)

Consul is a peer to peer, strongly consistent data store that uses a gossip protocol to communicate and form dynamic clusters. It is based on the Serf library.

It provides a hierarchical key/value store that you can place data in and register watches against to be notified when something changes within a particular key space. In this, it is similar to ZooKeeper.

As opposed to ZooKeeper and Etcd, however, Consul implements a full service discovery system in the library, and so you don’t need to implement your own use use a third party library. This includes health checks on both nodes and services.

It implements a DNS server interface, allowing you to perform service lookups using the DNS protocol. It also allows 'clients' to run as independent processes and register/ monitor services on their behalf. This removes the need to add explicit Consul support into your applications.  This is similar in concept to the Netflix OSS *Sidecar* concept that allows services with no ZooKeeper support to be registered and be discoverable in ZooKeeper.

Deployment wise, Consul agents are deployed onto the systems that services are running on, and not in a centralised fashion.

**I Recommend**

This is a newer product, and one that we like a lot. Generally recommended if you are able to adopt it.

As with the other strongly consistent systems in the list, care must be taken that you understand the implications of adopting it, including understanding it’s potential failure modes.

If you would like something similar that chooses availability rather than consistency, investigate the related Serf project. The Serf library serves as the basis of Consul, but has chosen different data guarantees guarantees. It is nowhere near as full featured, but can handily survive a split brain scenario and reform afterwards without any ill effects.

**Etcd**

[*http://www.etcd.io*](http://www.etcd.io)

Etcd is an HTTP accessible key/ value store. In that, it is similar in concept to ZooKeeper and the K/V portion of Consul.  It functions as a distributed, hierarchical configuration system, and can be used to build a Service Discovery system.

It originally grew out of the CoreOS project, is maintained by them and recently achieved a stable major release

***Simplicity Itself Recommends:***

If you are primarily using HTTP as your communication mechanism, then Etcd can’t be easily beaten. It provides a well distributed, fast HTTP based system, and has query and push notifications on change, via long polling.

**Eureka**

[*https://github.com/Netflix/eureka*](https://github.com/Netflix/eureka)

Eureka is a 'mid tier load balancer' built by Netflix and released as open source. It is designed to allow services to be able to register with a Eureka server and then locate each other via that server.

Eureka has several capabilities beyond other solutions presented here. It contains a built in load balancer which, although fairly simple, certainly does it’s job.  Netflix state that they have a secondary load balancer implementation internally that uses Eureka as a data source and is much more full featured. This hasn’t been released.

If you use Spring for your projects, then <a href="http://projects.spring.io/spring-cloud/">Spring Cloud</a> is an interesting project to look into, in order to be able to automatically register and resolve services in Eureka.

**I Recommend**

Like all Netflix OSS projects, it was written to run on the AWS infrastructure first and foremost. While other Netflix OSS projects have been extended to run in other environments, Eureka does not appear to be moving in that direction. We very much like the relation between client and server in the Eureka design, as it leaves the clients with the ability to continue working if the service discovery infrastructure fails.

Server wise, Eureka has also chosen availability rather than consistency. You must also be aware of the implications of this choice as it directly affects your application. Primarily, this manifests as a potentially stale or partial view of the full data set. This is discussed in the Eureka documentation.

For Spring projects, Eureka is now the quickest to get started with due to the investment the Spring team has made in adopting the Netflix OSS components via the Spring Cloud sub-projects.

**RollYourOwn**

If you can’t adapt to that, then you will have to create your own discovery solution within your existing infrastructure.

The basis of this will be :-

* Services must be able to notify each other of their availability and supply connection information</li>
* Periodic updates to the records to strip out stale information</li>
* Easy integration into your application infrastructure, often using a standard protocol such as HTTP or DNS</li>
* Notifications on services starting and stopping.</li>

**I Recommend**

Building your own discovery service should not be taken lightly. If you do need to then we recommend building a system that values availability rather the consistency. These are significantly easier to build, and more likely that you will build something that is functional.

The approach we would recommend would be to use some existing message infrastructure and broadcast notifications on service status. Each service caches the latest information from the broadcasts and uses that as a local set of service discovery data. This has the potential for being stale, but we’ve found this approach to scale reasonably well and is easy to implement.

If you do require consistency, then using some consistent data store could serve as the basis for a distributed configuration system that can be used to build service discovery. You will also want to emit notifications on status changes. You should realise, though, that building a consistent, distributed system is exceptionally hard to get right, and very easy to get subtly wrong.

Overall, really not recommended, but certainly possible.

https://stackshare.io/stackups/consul-vs-zookeeper-vs-eureka

https://www.consul.io/intro/vs/eureka.html

# Consul vs. Eureka

Eureka is a service discovery tool. The architecture is primarily client/server, with a set of Eureka servers per datacenter, usually one per availability zone. Typically clients of Eureka use an embedded SDK to register and discover services. For clients that are not natively integrated, a sidecar such as Ribbon is used to transparently discover services via Eureka.

Eureka provides a weakly consistent view of services, using best effort replication. When a client registers with a server, that server will make an attempt to replicate to the other servers but provides no guarantee. Service registrations have a short Time-To-Live (TTL), requiring clients to heartbeat with the servers. Unhealthy services or nodes will stop heartbeating, causing them to timeout and be removed from the registry. Discovery requests can route to any service, which can serve stale or missing data due to the best effort replication. This simplified model allows for easy cluster administration and high scalability.

Consul provides a super set of features, including richer health checking, key/value store, and multi-datacenter awareness. Consul requires a set of servers in each datacenter, along with an agent on each client, similar to using a sidecar like Ribbon. The Consul agent allows most applications to be Consul unaware, performing the service registration via configuration files and discovery via DNS or load balancer sidecars.

Consul provides a strong consistency guarantee, since servers replicate state using the [Raft protocol](https://www.consul.io/docs/internals/consensus.html). Consul supports a rich set of health checks including TCP, HTTP, Nagios/Sensu compatible scripts, or TTL based like Eureka. Client nodes participate in a [gossip based health check](https://www.consul.io/docs/internals/gossip.html), which distributes the work of health checking, unlike centralized heartbeating which becomes a scalability challenge. Discovery requests are routed to the elected Consul leader which allows them to be strongly consistent by default. Clients that allow for stale reads enable any server to process their request allowing for linear scalability like Eureka.

The strongly consistent nature of Consul means it can be used as a locking service for leader elections and cluster coordination. Eureka does not provide similar guarantees, and typically requires running ZooKeeper for services that need to perform coordination or have stronger consistency needs.

Consul provides a toolkit of features needed to support a service oriented architecture. This includes service discovery, but also rich health checking, locking, Key/Value, multi-datacenter federation, an event system, and ACLs. Both Consul and the ecosystem of tools like consul-template and envconsul try to minimize application changes required to integration, to avoid needing native integration via SDKs. Eureka is part of a larger Netflix OSS suite, which expects applications to be relatively homogeneous and tightly integrated. As a result, Eureka only solves a limited subset of problems, expecting other tools such as ZooKeeper to be used alongside.

https://www.digitalocean.com/community/tutorials/an-introduction-to-using-consul-a-service-discovery-system-on-ubuntu-14-04

### Introduction

**Consul** is a distributed, highly available, datacenter-aware, service discovery and configuration system. It can be used to present services and nodes in a flexible and powerful interface that allows clients to always have an up-to-date view of the infrastructure they are a part of.

Consul provides many different features that are used to provide consistent and available information about your infrastructure. This includes service and node discovery mechanisms, a tagging system, health checks, consensus-based election routines, system-wide key/value storage, and more. By leveraging consul within your organization, you can easily build a sophisticated level of awareness into your applications and services.

In this guide, we will introduce you to some of the basics of using consul. We will cover the general procedures necessary to get consul running on your servers to test it out. In the next guide, we will focus on setting up consul in a production environment.

## Prerequisites and Goals

In this guide, we will be getting familiar with using consul to build out a system of service discovery and configuration for your infrastructure.

For our demonstration, we will be configuring three servers and one client. Servers are used to handle queries and maintain a consistent view of the system. The client is also a member of the system, and can connect to the servers for information about the infrastructure. Clients may also contain services that will be monitored by consul.

For the purposes of this guide, and this series as a whole, we will be configuring 4 computers. The first three will be **consul servers** as described above. The last one will be a **consul agent** that acts as a client and can be used to query information about the system.

For us to implement some of the security mechanisms at a later point, we need to name all of our machines within a single domain. This is so that we can leverage a wildcard SSL certificate in the future.

The details of our machines are here:

| **Hostname** | **IP Address** | **Role** |
| --- | --- | --- |
| server1.example.com | 192.0.2.1 | bootstrap consul server |
| server2.example.com | 192.0.2.2 | consul server |
| server3.example.com | 192.0.2.3 | consul server |
| agent1.example.com | 192.0.2.50 | consul client |

We will be using 64-bit Ubuntu 14.04 servers for this demonstration, but any modern Linux server should work equally well.

## Downloading and Installing Consul

The first step that we need to take is to download and install the consul software on each of our machines. The following steps should be taken on each of the machines listed above. You should be logged in as root.

Before we look into the consul application, we need to get unzip to extract the executable. We will also use the screen application to allow us to easily have multiple sessions in a single terminal window. This is useful for our introduction, since consul typically takes up the entire screen when not run as a service.

Update the local systems package cache and then install the package using apt:

apt-get update

apt-get install unzip screen

So we do not forget to do so later, start your screen session now:

screen

Press enter if you get a copyright message. You will be dropped back into a terminal window, but you are now inside a screen session.

Now, we can go about getting the consul program. The [consul project's page](http://www.consul.io/downloads.html) provides download links to binary packages for Windows, OS X, and Linux.

Go to the page above and right-click on the operating system and architecture that represents your servers. In this guide, since we are using 64-bit servers, we will use the "amd64" link under "linux". Select "copy link location" or whatever similar option your browser provides.

In your terminal, move to the /usr/local/bin directory, where we will keep the executable. Type wget and a space, and then paste the URL that you copied from the site:

cd /usr/local/bin

wget https://dl.bintray.com/mitchellh/consul/0.3.0\_linux\_amd64.zip

Now, we can extract the binary package using the unzip command that we installed earlier. We can then remove the zipped file:

unzip \*.zip

rm \*.zip

You should now have the consul command available on all of your servers.

## Starting the Bootstrap Server

To begin working with consul, we need to get our consul servers up and running. When configuring this in the recommended multi-server environment, this step will have to be done in stages.

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 without an election (since it will be the only server available).

In the table that specifies our hosts, we designated our server1 as the bootstrap server. On server1, start the bootstrap instance by typing:

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

The server will start up in the current terminal and log data will be output as events occur. Towards the end of the log data, you will see these lines:

. . .

2014/07/07 14:32:15 [ERR] agent: failed to sync remote state: No cluster leader

2014/07/07 14:32:17 [WARN] raft: Heartbeat timeout reached, starting election

2014/07/07 14:32:17 [INFO] raft: Node at 192.0.2.1:8300 [Candidate] entering Candidate state

2014/07/07 14:32:17 [INFO] raft: Election won. Tally: 1

2014/07/07 14:32:17 [INFO] raft: Node at 192.0.2.1:8300 [Leader] entering Leader state

2014/07/07 14:32:17 [INFO] consul: cluster leadership acquired

2014/07/07 14:32:17 [INFO] consul: New leader elected: server1.example.com

2014/07/07 14:32:17 [INFO] consul: member 'server1.example.com' joined, marking health alive

As you can see, no cluster leader was found since this is the initial node. However, since we enabled the bootstrap option, this server was able to enter the leader state by itself in order to initiate a cluster with a single host.

## Starting the Other Servers

On server2 and server3, we can now start the consul service without the bootstrap option by typing:

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

For these servers, you will also see the log entries. Towards the end, you will see messages like this:

. . .

2014/07/07 14:37:25 [ERR] agent: failed to sync remote state: No cluster leader

2014/07/07 14:37:27 [WARN] raft: EnableSingleNode disabled, and no known peers. Aborting election.

2014/07/07 14:37:53 [ERR] agent: failed to sync remote state: No cluster leader

This happens because it cannot find a cluster leader and is not enabled to become the leader itself. This state occurs because our second and third server are enabled, but none of our servers are connected with each other yet.

To connect to each other, we need to join these servers to one another. This can be done in any direction, but the easiest is from the our server1 machine.

Since we are running the consul server in the current terminal window of server1, we will have to create another terminal with screen in order to do additional work. Create a new terminal window within the existing screen session of server1 by typing:

CTRL-A C

This will open a fresh terminal instance while keeping our previous session running. You can step through each of the existing terminal sessions by typing:

CTRL-A N

Back in your fresh terminal, join the other two instances by referencing their IP addresses like this:

consul join 192.0.2.2 192.0.2.3

This should instantly join all three servers into the same cluster. You can double check this by typing:

consul members

Node Address Status Type Build Protocol

server1.example.com 192.0.2.1:8301 alive server 0.3.0 2

server2.example.com 192.0.2.2:8301 alive server 0.3.0 2

server3.example.com 192.0.2.3:8301 alive server 0.3.0 2

You can get this information from any of the other servers as well by creating a new terminal session in screen as we described above and issuing the same command.

## Removing the Bootstrap Server and Re-Joining as a Regular Server

We have all three of our servers joined in a cluster, but we are not done yet.

Currently, since server1 was started in bootstrap mode, it has the power to make decisions without consulting the other servers. Since they are supposed to operate as equals and make decisions by quorum, we want to remove this privilege after the cluster has been bootstrapped.

To do this, we need to stop the consul service on server1. This will allow the remaining machines to select a new leader. We can then restart the consul service on server1 without the bootstrap option and rejoin the cluster.

On server1, switch back to the terminal running consul:

CTRL-A N

Stop the service by typing:

CTRL-C

Now, restart the service without the bootstrap option:

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

Switch back to your open terminal and rejoin the cluster by connecting with one of the two servers in the cluster:

CTRL-A N

consul join 192.0.2.2

You should now have your three servers available in equal standing. They will replicate information to each other and will handle situations where a single server becomes unavailable. Additional servers can now join the cluster as well by simply starting the server without bootstrap and joining the cluster.

## Joining the Cluster as a Client and Serving the Web UI

Now that the server cluster is available, we can go ahead and connect using the client machine.

We are going to put the consul web UI on our client machine so that we can interact with the cluster and monitor its health. To do this, [visit the download page for the web UI](http://www.consul.io/downloads_web_ui.html). Right-click on the download button and select "copy link location" or whatever similar option you have available.

On your client machine, change into your home directory. Type wget and a space and then paste the URL you copied from the page:

cd ~

wget https://dl.bintray.com/mitchellh/consul/0.3.0\_web\_ui.zip

When the download is complete, unzip and delete the archive:

unzip \*.zip

rm \*.zip

There will be a directory called dist that contains all of the files necessary to render the consul web UI. We just need to specify this directory when we are connecting to the cluster.

To connect to the cluster, we will use a similar call to consul agent that we used for the servers. We will use different flags however.

We will not use the server flag, since we want to operate in client mode. By default, each node's client interface is accessible using the local loopback interface. Since we want to access the web UI remotely, we'll have to specify the public IP address of the client instead.

We'll have to point consul to the directory that houses the web UI in order to serve that content. Additionally, we're going to join the cluster right away by passing the IP address of one of the servers in the cluster. This will allow us to avoid having to join afterwards. We could have done this earlier with the server examples as well.

In the end, our connection command is quite long. It will look like this:

consul agent -data-dir /tmp/consul -client 192.0.2.50 -ui-dir /home/your\_user/dir -join 192.0.2.1

This will connect our client machine to the cluster as a regular, non-server agent. This agent will respond to requests on its public IP address instead of the usual 127.0.0.1 interface. Because of this, you will need to add an additional flag to any consul commands specifying the rpc-addr.

For instance, if you want to query the list of members from the client, you'll have to do so by passing in the alternative interface and port that you selected:

consul members -rpc-addr=192.0.2.50:8400

Node Address Status Type Build Protocol

agent1 192.0.2.50:8301 alive client 0.3.0 2

server2 192.0.2.2:8301 alive server 0.3.0 2

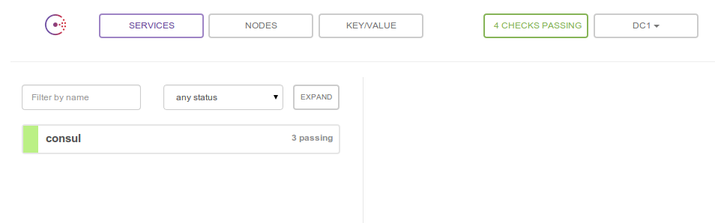
server1 192.0.2.1:8301 alive server 0.3.0 2

server3 192.0.2.3:8301 alive server 0.3.0 2

This may seem like a hassle, but it provides us with the opportunity to access the consul web interface. You can get to the web interface by visiting your client's IP address, followed by :8500/ui in your web browser:

http://192.0.2.50:8500/ui

The main interface will look like this:



You can click through various menus and explore the interface. This provides you with a good way to visualize your cluster and the health of your machines and services.

## Adding Services and Checks

Now, we want to add services to consul, which is the primary use-case for setting this up. You can add services in a number of ways, but the easiest is to create a configuration directory to store your service definitions.

A service is associated with the node that contains the service definition. So if we have a web server, we should install the consul agent on that server and create a service definition file there.

For our purposes, we'll install Nginx on our client to demonstrate this. Kill the current client session by typing:

CTRL-C

Install Nginx on the client by typing:

apt-get install nginx

Now, we can create a configuration directory to store our service definitions:

mkdir ~/services

Inside this directory, we will create a JSON file that describes our web service. We will call this web.json:

nano ~/services/web.json

Inside this file, we need include a structure for our service definition. Within this structure, we'll define a sub-structure for a health check of the service in order for us to reliably be able to tell whether its running or not.

The basic outline looks like this:

{

"service": {

. . .

"check": {

. . .

}

}

}

For the service, we need to define a name for the service and tell consul what port it should be checking. Additionally, we can give it a list of tags that we can use to arbitrarily categorize the service for our own sorting purposes.

For our example, this looks like this:

{

"service": {

"name": "web server",

"port": 80,

"tags": ["nginx", "demonstration"],

"check": {

. . .

}

}

}

This is all we need to define the service itself. However, we also want to define a method by which consul can verify the health of the service. This is usually fairly simple and will replicate a normal system administrator's manual checks.

For our service, we will implement a simple web request with curl as the consul project lists in [its own documentation](http://www.consul.io/intro/getting-started/checks.html). We don't actually need to know what curl is able to retrieve, we only care about whether the command was able to execute without any errors. Because of this, we can throw away any output.

We also need to set the interval at which the check will be run. This is always a compromise between performance and up-to-date information. We'll use 10 seconds, since we want to know relatively soon if something is wrong:

{

"service": {

"name": "web server",

"port": 80,

"tags": ["nginx", "demonstration"],

"check": {

"script": "curl localhost:80 > /dev/null 2>&1",

"interval": "10s"

}

}

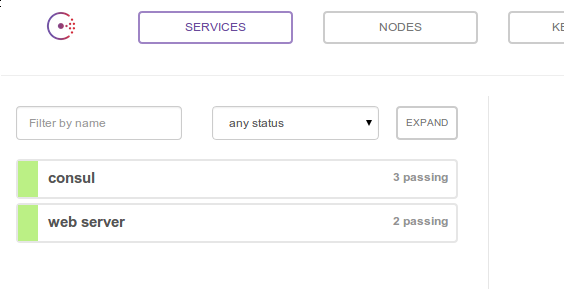
}

Save and close the file when you are finished.

Now, we can simply restart the client consul session, and point to this directory as having service definitions:

consul agent -data-dir /tmp/consul -client 192.0.2.50 -ui-dir /home/your\_user/dist -join 192.0.2.1 -config-dir /home/your\_user/services

This will restart the node and connect it to the cluster. If you return to the web interface, you should now now see a service:

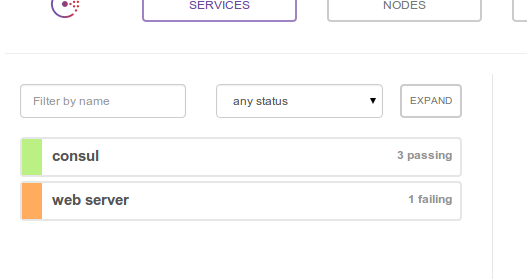


Back on your client, you can create a new terminal and temporarily stop the web server:

CTRL-A C

service nginx stop

When you refresh the web UI, you can see that web service check is now failing, as expected:



This shows that our health check is working as expected.

## Conclusion

You should now have a basic idea of how consul works. The demonstration we have provided in this guide is not exactly the best way to handle consul in production, but was used to let you see the useful features of the software quickly.

In the [next guide](https://www.digitalocean.com/community/tutorials/how-to-configure-consul-in-a-production-environment-on-ubuntu-14-04), we'll cover how to use consul in a production environment. We will put all of our configuration details in files for easy reference and create upstart scripts to start the service at boot.

https://www.consul.io/intro/getting-started/next-steps.html

# Next Steps

That concludes the getting started guide for Consul. Hopefully you're able to see that while Consul is simple to use, it has a powerful set of features. We've covered the basics for all of these features in this guide.

Consul is designed to be friendly to both the DevOps community and application developers, making it perfect for modern, elastic infrastructures.

As a next step, the following resources are available:

* [Documentation](https://www.consul.io/docs/index.html) - The documentation is an in-depth reference guide to all the features of Consul, including technical details about the internals of how Consul operates.
* [Guides](https://www.consul.io/docs/guides/index.html) - This section provides various getting started guides with Consul, including how to bootstrap a new datacenter.
* [Examples](https://github.com/hashicorp/consul/tree/master/demo) - The work-in-progress examples folder within the GitHub repository for Consul contains functional examples of various use cases of Consul to help you get started with exactly what you need.

https://www.consul.io/intro/getting-started/join.html

# Consul Cluster

We've started our first agent and registered and queried a service on that agent. This showed how easy it is to use Consul but didn't show how this could be extended to a scalable, production-grade service discovery infrastructure. In this step, we'll create our first real cluster with multiple members.

When a Consul agent is started, it begins without knowledge of any other node: it is an isolated cluster of one. To learn about other cluster members, the agent must join an existing cluster. To join an existing cluster, it only needs to know about a single existing member. After it joins, the agent will gossip with this member and quickly discover the other members in the cluster. A Consul agent can join any other agent, not just agents in server mode.

## [»](https://www.consul.io/intro/getting-started/join.html" \l "starting-the-agents) Starting the Agents

To simulate a more realistic cluster, we will start a two node cluster via [Vagrant](https://www.vagrantup.com/). The Vagrantfile we will be using can be found in the [demo section of the Consul repo](https://github.com/hashicorp/consul/tree/master/demo/vagrant-cluster).

We first boot our two nodes:

$ vagrant up

Once the systems are available, we can ssh into them to begin configuration of our cluster. We start by logging in to the first node:

$ vagrant ssh n1

In our previous examples, we used the [-dev flag](https://www.consul.io/docs/agent/options.html#_dev) to quickly set up a development server. However, this is not sufficient for use in a clustered environment. We will omit the -dev flag from here on, and instead specify our clustering flags as outlined below.

Each node in a cluster must have a unique name. By default, Consul uses the hostname of the machine, but we'll manually override it using the [-node command-line option](https://www.consul.io/docs/agent/options.html#_node).

We will also specify a [bind address](https://www.consul.io/docs/agent/options.html#_bind): this is the address that Consul listens on, and it must be accessible by all other nodes in the cluster. While a bind address is not strictly necessary, it's always best to provide one. Consul will by default attempt to listen on all IPv4 interfaces on a system, but will fail to start with an error if multiple private IPs are found. Since production servers often have multiple interfaces, specifying a bind address assures that you will never bind Consul to the wrong interface.

The first node will act as our sole server in this cluster, and we indicate this with the [server switch](https://www.consul.io/docs/agent/options.html#_server).

The [-bootstrap-expect flag](https://www.consul.io/docs/agent/options.html#_bootstrap_expect) hints to the Consul server the number of additional server nodes we are expecting to join. The purpose of this flag is to delay the bootstrapping of the replicated log until the expected number of servers has successfully joined. You can read more about this in the [bootstrapping guide](https://www.consul.io/docs/guides/bootstrapping.html).

Finally, we add the [config-dir flag](https://www.consul.io/docs/agent/options.html#_config_dir), marking where service and check definitions can be found.

All together, these settings yield a [consul agent](https://www.consul.io/docs/commands/agent.html) command like this:

vagrant@n1:~$ consul agent -server -bootstrap-expect=1 \

-data-dir=/tmp/consul -node=agent-one -bind=172.20.20.10 \

-config-dir=/etc/consul.d

...

Now, in another terminal, we will connect to the second node:

$ vagrant ssh n2

This time, we set the [bind address](https://www.consul.io/docs/agent/options.html#_bind) address to match the IP of the second node as specified in the Vagrantfile and the [node name](https://www.consul.io/docs/agent/options.html#_node) to be agent-two. Since this node will not be a Consul server, we don't provide a [server switch](https://www.consul.io/docs/agent/options.html#_server).

All together, these settings yield a [consul agent](https://www.consul.io/docs/commands/agent.html) command like this:

vagrant@n2:~$ consul agent -data-dir=/tmp/consul -node=agent-two \

-bind=172.20.20.11 -config-dir=/etc/consul.d

...

At this point, you have two Consul agents running: one server and one client. The two Consul agents still don't know anything about each other and are each part of their own single-node clusters. You can verify this by running [consul members](https://www.consul.io/docs/commands/members.html) against each agent and noting that only one member is visible to each agent.

## [»](https://www.consul.io/intro/getting-started/join.html" \l "joining-a-cluster) Joining a Cluster

Now, we'll tell the first agent to join the second agent by running the following commands in a new terminal:

$ vagrant ssh n1

...

vagrant@n1:~$ consul join 172.20.20.11

Successfully joined cluster by contacting 1 nodes.

You should see some log output in each of the agent logs. If you read carefully, you'll see that they received join information. If you run [consul members](https://www.consul.io/docs/commands/members.html) against each agent, you'll see that both agents now know about each other:

vagrant@n2:~$ consul members

Node Address Status Type Build Protocol

agent-two 172.20.20.11:8301 alive client 0.5.0 2

agent-one 172.20.20.10:8301 alive server 0.5.0 2

**Remember:** To join a cluster, a Consul agent only needs to learn about one existing member. After joining the cluster, the agents gossip with each other to propagate full membership information.

## [»](https://www.consul.io/intro/getting-started/join.html" \l "auto-joining-a-cluster-on-start) Auto-joining a Cluster on Start

Ideally, whenever a new node is brought up in your datacenter, it should automatically join the Consul cluster without human intervention. Consul facilitates auto-join by enabling the auto-discovery of instances in AWS, Google Cloud or Azure with a given tag key/value. To use the integration, add the [retry\_join\_ec2](https://www.consul.io/docs/agent/options.html?#retry_join_ec2), [retry\_join\_gce](https://www.consul.io/docs/agent/options.html?#retry_join_gce) or the [retry\_join\_azure](https://www.consul.io/docs/agent/options.html?#retry_join_azure) nested object to your Consul configuration file. This will allow a new node to join the cluster without any hardcoded configuration. Alternatively, you can join a cluster at startup using the [-join flag](https://www.consul.io/docs/agent/options.html#_join) or [start\_join setting](https://www.consul.io/docs/agent/options.html#start_join) with hardcoded addresses of other known Consul agents.

## [»](https://www.consul.io/intro/getting-started/join.html" \l "querying-nodes) Querying Nodes

Just like querying services, Consul has an API for querying the nodes themselves. You can do this via the DNS or HTTP API.

For the DNS API, the structure of the names is NAME.node.consul or NAME.node.DATACENTER.consul. If the datacenter is omitted, Consul will only search the local datacenter.

For example, from "agent-one", we can query for the address of the node "agent-two":

vagrant@n1:~$ dig @127.0.0.1 -p 8600 agent-two.node.consul

...

;; QUESTION SECTION:

;agent-two.node.consul. IN A

;; ANSWER SECTION:

agent-two.node.consul. 0 IN A 172.20.20.11

The ability to look up nodes in addition to services is incredibly useful for system administration tasks. For example, knowing the address of the node to SSH into is as easy as making the node a part of the Consul cluster and querying it.

## [»](https://www.consul.io/intro/getting-started/join.html" \l "leaving-a-cluster) Leaving a Cluster

To leave the cluster, you can either gracefully quit an agent (using Ctrl-C) or force kill one of the agents. Gracefully leaving allows the node to transition into the left state; otherwise, other nodes will detect it as having failed. The difference is covered in more detail [here](https://www.consul.io/intro/getting-started/agent.html#stopping).

## [»](https://www.consul.io/intro/getting-started/join.html" \l "next-steps) Next Steps

We now have a multi-node Consul cluster up and running. Let's make our services more robust by giving them [health checks](https://www.consul.io/intro/getting-started/checks.html)!

https://www.consul.io/intro/getting-started/install.html

# Install Consul

Consul must first be installed on your machine. Consul is distributed as a [binary package](https://www.consul.io/downloads.html) for all supported platforms and architectures. This page will not cover how to compile Consul from source, but compiling from source is covered in the [documentation](https://www.consul.io/docs/index.html) for those who want to be sure they're compiling source they trust into the final binary.

## [»](https://www.consul.io/intro/getting-started/install.html" \l "installing-consul) Installing Consul

To install Consul, find the [appropriate package](https://www.consul.io/downloads.html) for your system and download it. Consul is packaged as a zip archive.

After downloading Consul, unzip the package. Consul runs as a single binary named consul. Any other files in the package can be safely removed and Consul will still function.

The final step is to make sure that the consul binary is available on the PATH. See [this page](https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux) for instructions on setting the PATH on Linux and Mac. [This page](https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows) contains instructions for setting the PATH on Windows.

## [»](https://www.consul.io/intro/getting-started/install.html" \l "verifying-the-installation) Verifying the Installation

After installing Consul, verify the installation worked by opening a new terminal session and checking that consul is available. By executing consul you should see help output similar to this:

$ consul

usage: consul [--version] [--help] <command> [<args>]

Available commands are:

agent Runs a Consul agent

event Fire a new event

# ...

If you get an error that consul could not be found, your PATH environment variable was not set up properly. Please go back and ensure that your PATH variable contains the directory where Consul was installed.

## [»](https://www.consul.io/intro/getting-started/install.html#next-steps) Next Steps

Consul is installed and ready for operation. Let's [run the agent](https://www.consul.io/intro/getting-started/agent.html)!

https://www.consul.io/intro/index.html

# Introduction to Consul

Welcome to the intro guide to Consul! This guide is the best place to start with Consul. We cover what Consul is, what problems it can solve, how it compares to existing software, and how you can get started using it. If you are familiar with the basics of Consul, the [documentation](https://www.consul.io/docs/index.html) provides a more detailed reference of available features.

## [»](https://www.consul.io/intro/index.html" \l "what-is-consul-) What is Consul?

Consul has multiple components, but as a whole, it is a tool for discovering and configuring services in your infrastructure. 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.

Consul is designed to be friendly to both the DevOps community and application developers, making it perfect for modern, elastic infrastructures.

## [»](https://www.consul.io/intro/index.html" \l "basic-architecture-of-consul) Basic Architecture of Consul

Consul is a distributed, highly available system. This section will cover the basics, purposely omitting some unnecessary detail, so you can get a quick understanding of how Consul works. For more detail, please refer to the [in-depth architecture overview](https://www.consul.io/docs/internals/architecture.html).

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.

Components of your infrastructure that need to discover other services or nodes can query any of the Consul servers or any of the Consul agents. The agents forward queries to the servers automatically.

Each datacenter runs a cluster of Consul servers. When a cross-datacenter service discovery or configuration request is made, the local Consul servers forward the request to the remote datacenter and return the result.

## [»](https://www.consul.io/intro/index.html#next-steps) Next Steps

* See [how Consul compares to other software](https://www.consul.io/intro/vs/index.html) to assess how it fits into your existing infrastructure.
* Continue onwards with the [getting started guide](https://www.consul.io/intro/getting-started/install.html) to get Consul up and running.

In the [previous tutorial](https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html) we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs -- the first will emit log messages and the second will receive and print them.

In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.

Essentially, published log messages are going to be broadcast to all the receivers.

**Exchanges**

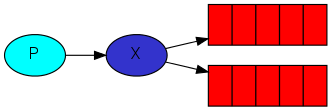
In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.

Let's quickly go over what we covered in the previous tutorials:

* A *producer* is a user application that sends messages.
* A *queue* is a buffer that stores messages.
* A *consumer* is a user application that receives messages.

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an *exchange*. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the *exchange type*.



There are a few exchange types available: direct, topic, headers and fanout. We'll focus on the last one -- the fanout. Let's create an exchange of this type, and call it logs:

channel.ExchangeDeclare("logs", "fanout");

The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

**Listing exchanges**

To list the exchanges on the server you can run the ever useful rabbitmqctl:

sudo rabbitmqctl list\_exchanges

In this list there will be some amq.\* exchanges and the default (unnamed) exchange. These are created by default, but it is unlikely you'll need to use them at the moment.

**The default exchange**

In previous parts of the tutorial we knew nothing about exchanges, but still were able to send messages to queues. That was possible because we were using a default exchange, which we identify by the empty string ("").

Recall how we published a message before:

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "",

routingKey: "hello",

basicProperties: null,

body: body);

The first parameter is the the name of the exchange. The empty string denotes the default or *nameless* exchange: messages are routed to the queue with the name specified by routingKey, if it exists.

Now, we can publish to our named exchange instead:

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "logs",

routingKey: "",

basicProperties: null,

body: body);

**Temporary queues**

As you may remember previously we were using queues which had a specified name (remember hello and task\_queue?). Being able to name a queue was crucial for us -- we needed to point the workers to the same queue. Giving a queue a name is important when you want to share the queue between producers and consumers.

But that's not the case for our logger. We want to hear about all log messages, not just a subset of them. We're also interested only in currently flowing messages not in the old ones. To solve that we need two things.

Firstly, whenever we connect to Rabbit we need a fresh, empty queue. To do this we could create a queue with a random name, or, even better - let the server choose a random queue name for us.

Secondly, once we disconnect the consumer the queue should be automatically deleted.

In the .NET client, when we supply no parameters to queueDeclare() we create a non-durable, exclusive, autodelete queue with a generated name:

var queueName = channel.QueueDeclare().QueueName;

At that point queueName contains a random queue name. For example it may look like amq.gen-JzTY20BRgKO-HjmUJj0wLg.

https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html

A new registration module went live and did work like a charm initially. One of the early problems we encountered was that the registration module showed an error once a user finished the registration process. Subsequent investigations revealed that the third-party SMTP APIs were down for some time. It was during this downtime that a user registered but never received confirmation emails.

Keeping the above in mind, we also had to take into account the following scenarios:  
– Third-party API stops working  
– Audit log table gets locked due to high traffic  
– User information is sent to other internal systems which are written in a different language  
– And many more such cases where direct communication is required between applications.

**Can our code scale to meet new requirements or remove dependencies on third-party APIs?**

We assumed that if our RegisterUser method will only register the user, other tasks can be performed by other applications without impacting the user registration process.

The code was changed to:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public void RegisterUser(User user)  {     //Register user     userService.Register(user);     //Send message about user's registration    var message = new Message()     {     //user's registration information    }    messagingService.SendMessage(message)  } |

These messages can now be stored in a central place so that other applications can have quick access. Now email services, third-party APIs, and audit log service can run independently and perform their tasks without impacting the user registration module.

Even in case of any issues encountered in email or third-party services, our registration module will keep working, and once these services are up again they can continue their tasks without losing any data.

**What is Messaging?**

It is a way of exchanging messages from point A to point B or many points C. Messaging enables distributed communication that is loosely coupled. A message sender sends a message to a destination, and the message recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. To understand better, email is a great example of messaging which enables people to communicate.

You may want to use messaging in the following scenarios:

* Send data to many applications without calling their API directly into your application
* Want to do things in certain order like a transactional system
* Monitor data feeds like the number of registrations in any application

The component which receives the message from the sender and recipient retrieves the message from a message broker or messaging middleware.

**Message Broker/Middleware**

According to Wikipedia:

“Message-oriented middleware (MOM) is software or hardware infrastructure supporting sending and receiving messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols.”

Message brokers do many things such as:

* Decouple message publisher and consumer
* Store the messages
* Routing of messages
* Monitoring and management of messages

Such message broker services typically used to lack standards, and existing commercial implementations had proprietary implementation and API issues to look at. This was up until recently an issue which can now be addressed through AMQP. AMQP is an open standard application layer protocol for message-oriented middleware.

From [the AMQP website](http://www.amqp.org/):

“AMQP is an Open Standard for Messaging Middleware.

By complying to the AMQP standard, middleware products written for different platforms and in different languages can send messages to one another. AMQP addresses the problem of transporting value-bearing messages across and between organizations in a timely manner.

AMQP enables complete interoperability for messaging middleware; both the networking protocol and the semantics of broker services are defined in AMQP.”

In null-set, AMQP defines:

* Where to send messages (Routing)
* How to get there (Delivery)
* What goes in must come out (Fidelity)

[AMQP](http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol) is a standard  wire level protocol (communicating with a remote machine or getting data from point to point) and has many [implementations](http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Implementations).

**RabbitMQ** is one such open source message broker software that implements AMQP.

**Hello RabbitMQ**

RabbitMQ is a message broker that takes messages and sends them to other places in a pretty smart way. AMQP is the protocol that RabbitMQ implements. It is completely language-neutral and while using it you can write and read to them in any language just like you would while using TCP or HTTP.

Another great advantage is that RabbitMQ runs on all [major operating systems](http://www.rabbitmq.com/platforms.html) and supports a large number of developer platforms like [Java, .NET, Python, PHP, Erlang and many more](http://www.rabbitmq.com/devtools.html).

The RabbitMQ server is written in the *Erlang* programming language and is built on the *Open Telecom Platform*(OTP) framework for clustering and failover.

As defined on [the Erlang website](http://www.erlang.org/):

“Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang’s runtime system has built-in support for concurrency, distribution and fault tolerance.”

“OTP is set of Erlang libraries and design principles providing middle-ware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release handling tools.”

Example: WhatsApp uses Erlang to run messaging servers, achieving up to 2 million connected users per server.

RabbitMQ has the following advantages:

* Fast
* Polyglot
* Simple management
* No Erlang knowledge needed
* Great documentation and community

**Installation Steps:**

**Erlang:** Download and install Erlang (OTP R16B03-1 Windows 64bit Binary Release) from <http://www.erlang.org/download.html>

**RabbitMQ:** Download and install RabbitMQ server for Windows from <http://www.rabbitmq.com/download.html>

RabbitMQ will be installed as a Windows service

RabbitMQ also comes with web-based management plugin which is quite handy to manage RabbitMQ.

Steps to install the management plugin:

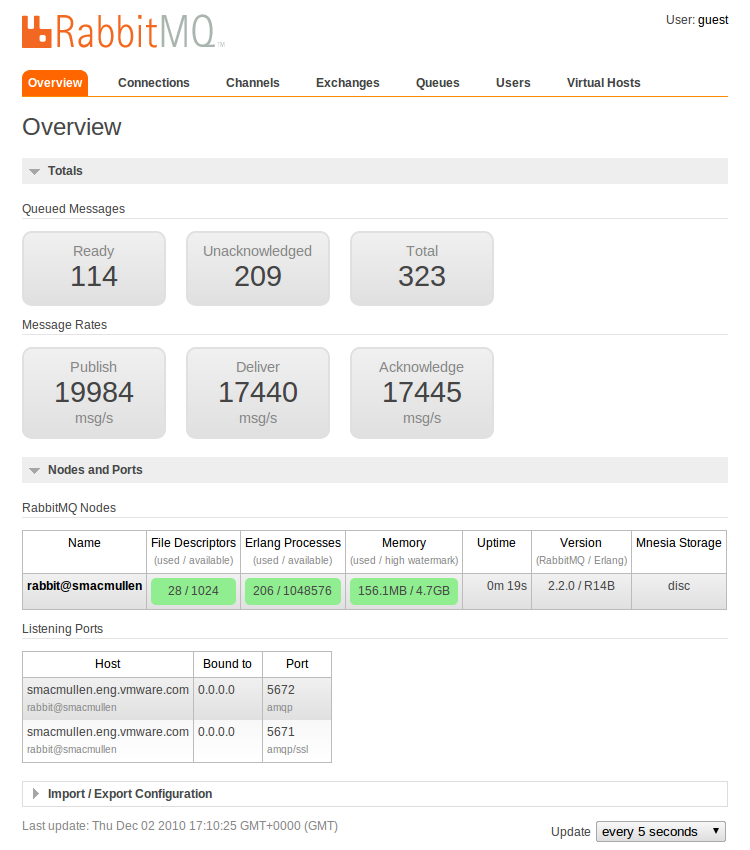
* Open command prompt
* Run the following command from /sbin

`rabbitmq-plugins.bat enable rabbitmq\_management`

* Restart the RabbitMQ by following command

rabbitmq-service.bat stop  
rabbitmq-service.bat start

By default management plugin will run on http://localhost:15672/  
Default user name and password is guest & guest



**Basic operation**

* To stop RabbitMQ along with Erlang

***`rabbitmqctl stop`***

* Stop/Start RabbitMQ only

***rabbitmqctl stop\_app***  
***rabbitmqctl start\_app***

**Elements**

**Producer** creates message and sends (publishes) into the message broker (RabbitMQ). A message must have two parts: a payload and a label. Payload is data and it can be anything from a simple JSON to MPEG-4 file. Label describes the payload and how RabbitMQ will determine who should get a copy of the message. The communication between publisher and RabbitMQ is one directional and fire and forget.

**Consumer** on the other hand attaches to the broker and subscribes to a queue to get the message.

Your app can be producer when it needs to send messages to other applications, or it can be consumer when it needs to receive the message.

**Connect to RabbitMQ**

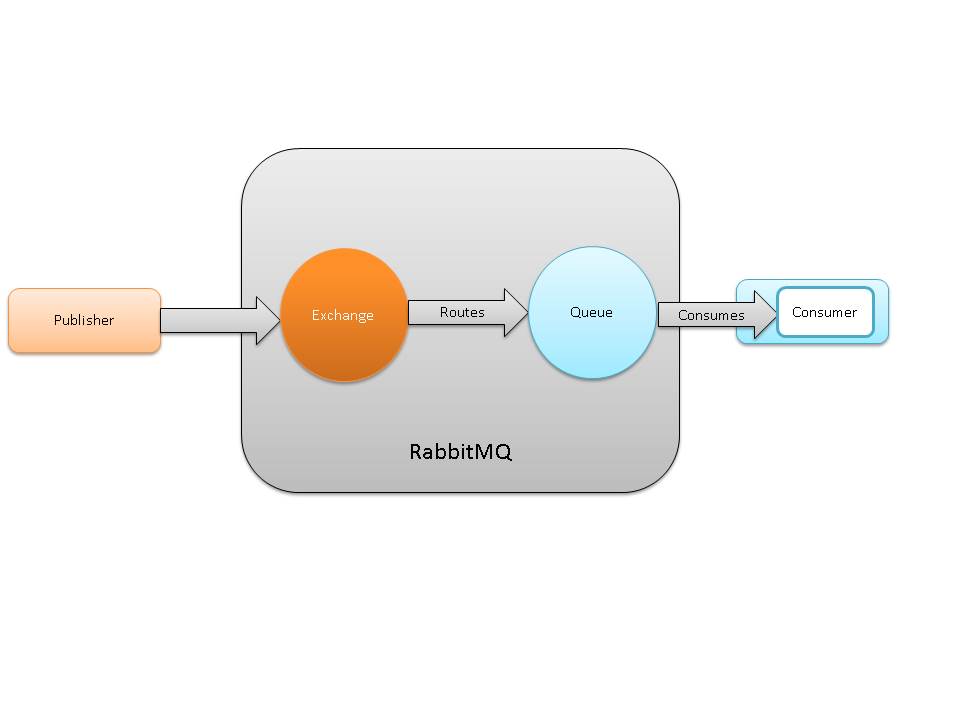
Applications will connect to RabbitMQ by creating TCP connections and getting authenticated. Setting up and tearing down TCP session is a time-consuming process for OS. AMQP also uses what is called a  **channel** which is nothing but a virtual connection inside a real TCP connection. The Publisher/Consumer apps use channel to issue AMQP command to the broker. A single TCP connection can be used to establish multiple communication paths between the application and the broker. Publisher writes to channel and consumer reads through the channel.

**Exchanges, Queues and Binding**

**Exchange:** Producer publishes messages in exchange. Producer never directly communicates through consumer applications.

**Queue:** Messages end up in the queue and are received by the consumer.

**Binding:** Rule to route the message into one or more queue. This is a relationship between exchange and a queue.

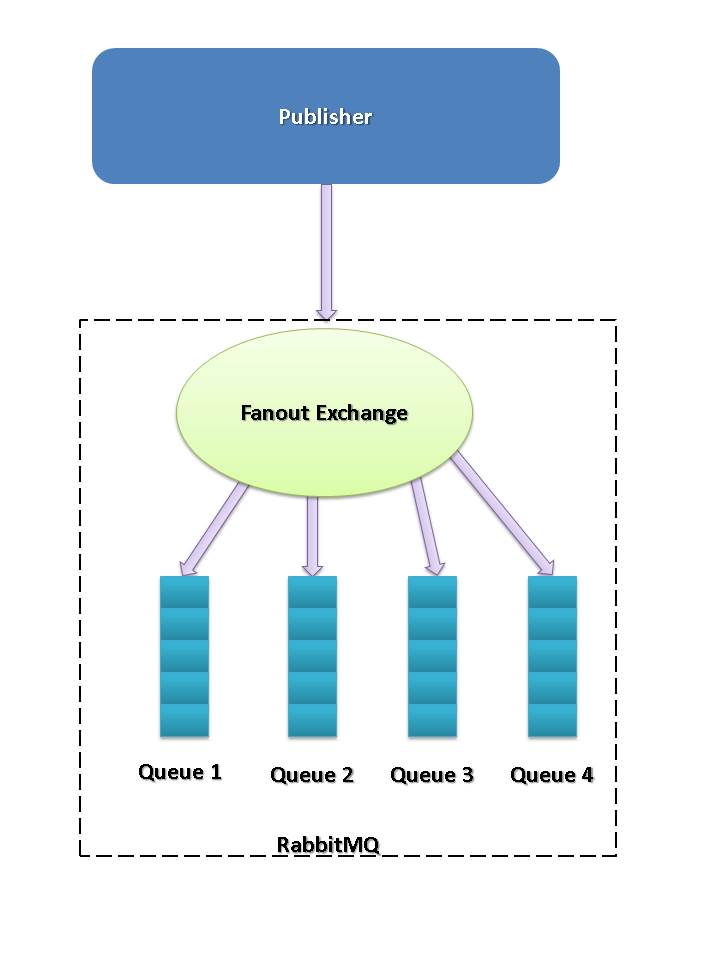


When producer publishes a message in exchange it contains two parts; a payload and a label. Label is called the routing key. Each queue is bound to a routing key or a pattern of routing keys. This routing key binding with queue are rules that allow the exchange to put messages into the queue.

The exchange receives messages from producer and from the other side it pushes them to the queues. The exchange must know exactly what to do with the messages it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded? The rules for determining that are defined by the exchange type.

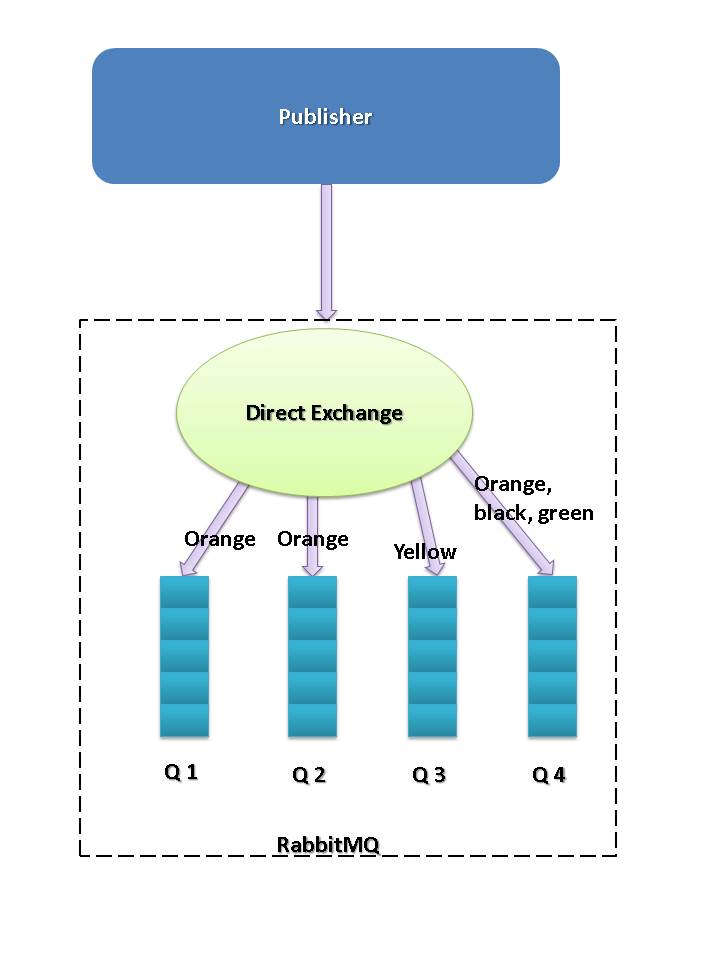
There are mainly four type of exchanges available: direct, topic, headers and fanout.

**Fanout Exchange:** Fanout exchange brodcasts all the messages it receives to all the queues it knows.

**Direct exchange:**  
As we have seen, Fanout exchange broadcasts all the messages to all queues. In direct exchange, message goes to the queues whose binding key exactly matches the routing key (label) of the message. In direct exchange it is also perfectly legal to bind multiple queues with the same binding key.

In the below example Q1 & Q2 are binded with the routing key “Orange,” Q3 with yellow, and Q4 with Orange, black, and green.

* Exchange receives message with “Orange” routing key and it will deliver message to Q1, Q2, and Q4
* Exchange receives message with “Yellow” routing key and it will deliver message to Q3.
* Exchange receives message with “Green” routing key and it will deliver to Q4.



**Topic exchange:**  
Direct exchange gives flexibility to bind routing key with queues, but still it lacks binding based upon pattern. In topic exchanges will route messages to one or many queues based on the pattern that was used to bind a queue to an exchange. Messages sent to a topic exchange can’t have an arbitrary routing\_key – it must be a list of words, delimited by dots. A valid routing key examples are “stock.usd.nyse”, “nyse.vmw“, “quick.orange.rabbit“. Queues can bind with exchange bu using patterns like “\*.usd.\*” to get all the messages where “usd” is a middle part of the message.

In the above example:

1. Q1 will get all the messages which start with a log. E.g. log.debug, log.error, log.info, and log.warn etc.
2. Q2 will get message with routing key log.error.
3. Q3 will only get log.info message.
4. Q4 will get all the messages.

**Summary**

Messaging enables software applications to connect and scale. RabbitMQ is fast, reliable, and a flexible messaging solution and therefore my preferred choice for enterprise-level message broker service.

https://www.3pillarglobal.com/insights/rabbitmq-understanding-message-broker

A message can include any kind of information. It could, for example, have information about a process/task that should start on another application (that could be on another server), or it could be just a simple text message. The queue-manager software stores the messages until a receiving application connects and takes a message off the queue. The receiving application then processes the message in an appropriate manner.

All articles from **Getting Started with RabbitMQ** can be downloaded as a free ebook [here.](https://www.cloudamqp.com/rabbitmq_ebook.html)

#### Table of Contents

1. [RabbitMQ for beginners - What is RabbitMQ?](https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html)

Gives a brief understanding of messaging and important RabbitMQ concepts are defined

1. [RabbitMQ step-by-step coding instructions](https://www.cloudamqp.com/blog/part2-rabbitmq-for-beginners_example-and-sample-code.html)

Step-by-step instructions which shows how to set up a connection, how to publish to a queue, and how to subscribe from the queue

* + [Ruby sample code](https://www.cloudamqp.com/blog/2015-05-19-part2-1-rabbitmq-for-beginners_example-and-sample-code-ruby.html)
  + [Node.js sample code](https://www.cloudamqp.com/blog/2015-05-19-part2-2-rabbitmq-for-beginners_example-and-sample-code-node-js.html)
  + [Python sample code](https://www.cloudamqp.com/blog/2015-05-21-part2-3-rabbitmq-for-beginners_example-and-sample-code-python.html)

1. [The management interface](https://www.cloudamqp.com/blog/2015-05-27-part3-rabbitmq-for-beginners_the-management-interface.html)

Describes how to monitor and handle your RabbitMQ server from a web browser

1. [Exchanges, routing keys and bindings](https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html)

Explains the different types of exchanges in RabbitMQ and how exchanges and queues are associated with each other

[Download the ebook **Getting started with RabbitMQ** for free](https://www.cloudamqp.com/rabbitmq_ebook.html)

### RabbitMQ Example

A message brooking solutions can act like a middleman for various services (e.g. a web application, as in this example). They can be used to reduce loads and delivery times by web application servers since tasks, which would normally take quite a bit of time to process, can be delegated to a third party whose only job is to perform them.

In this guide, we follow a scenario where a web application allows users to upload information to a web site. The site will handle this information and generate a PDF and email it back to the user. Handling the information, generating the PDF and sending the email will in this example case take several seconds and that is one of the reasons of why a message queue will be used.

When the user has entered user information into the web interface, the web application will put a "PDF processing" - task and all information into a message and the message will be placed onto a queue defined in RabbitMQ.

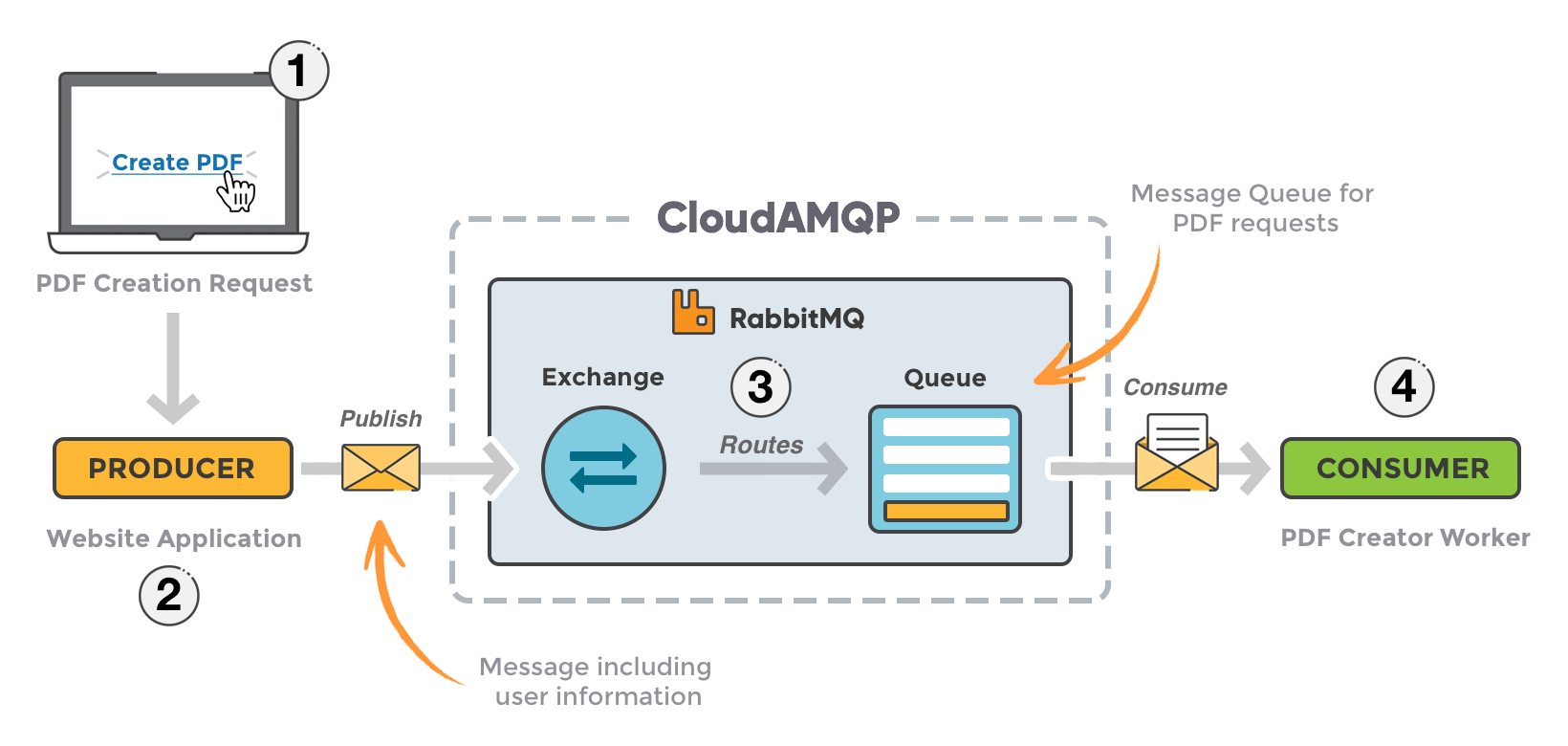


The basic architecture of a message queue is simple, there are client applications called producers that create messages and deliver them to the broker (the message queue). Other applications, called consumers, connects to the queue and subscribes to the messages to be processed. A software can be a producer, or consumer, or both a consumer and a producer of messages. Messages placed onto the queue are stored until the consumer retrieves them.

##### When and why should you use RabbitMQ?

Message queueing allow web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures on the spot. Message queueing is also good when you want to distribute a message to multiple recipients for consumption or for balancing loads between workers.

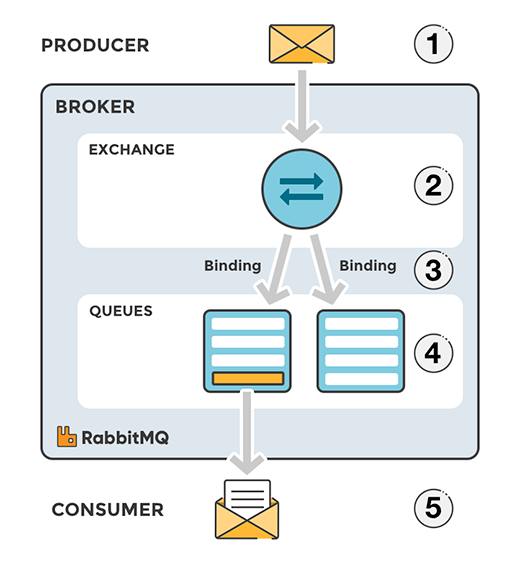
The consumer can take a message of the queue and start the processing of the PDF at the same time as the producer is queueing up new messages on the queue. The consumer can be on a totally different server than the publisher, or they can be located on the same server. Request can be created in one programming language and handled in another programming language - the two applications will only communicate through the messages they are sending to each other. Due to that, the two applications will have a low coupling between the sender and the receiver.



1. The user sends a PDF creation request to the web application.
2. The web application (the producer) sends a message to RabbitMQ, including data from the request, like name and email.
3. An exchange accepts the messages from a producer application and routes them to correct message queues for PDF creation.
4. The PDF processing worker (the consumer) receives the task and starts the processing of the PDF.

#### Exchanges

Messages are not published directly to a queue, instead, the producer sends messages to an exchange. An exchange is responsible for the routing of the messages to the different queues. An exchange accepts messages from the producer application and routes them to message queues with the help of bindings and routing keys. A binding is the link between a queue and an exchange.

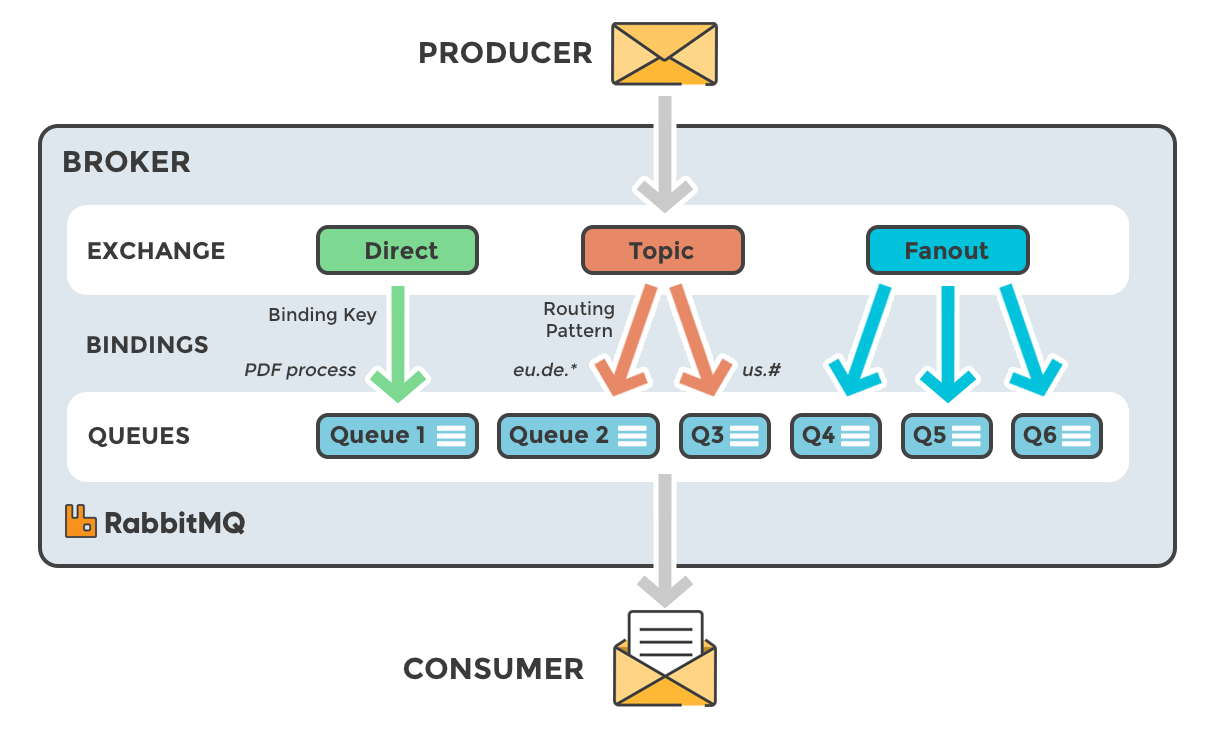


##### Message flow in RabbitMQ

1. The producer publish a message to an exchange. When you create the exchange, you have to specify the type of it. The different types of exchanges are explained in detail later on.
2. The exchange receives the message and is now responsible for the routing of the message. The exchange take different message attributes into account, such as routing key, depending on the exchange type.
3. Bindings have to be created from the exchange to queues. In this case we see two bindings to two different queues from the exchange. The Exchange routes the message in to the queues depending on message attributes.
4. The messages stay in the queue until they are handled by a consumer
5. The consumer handles the message.

#### Types of exchanges

In Part 2 of the tutorial are only direct exchanges used. Deeper understanding about the different exchange types, binding keys, routing keys and how/when you should use them can be found in part 4 about exchanges: [Part 4: RabbitMQ for beginners - Exchanges, routing keys and bindings.](https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html)



* **Direct:** A direct exchange delivers messages to queues based on a message routing key. In a direct exchange, the message are routed to the queues whose binding key exactly matches the routing key of the message. If the queue is bound to the exchange with the binding key *pdfprocess,* a message published to the exchange with a routing key *pdfprocess* will be routed to that queue.
* **Fanout:** A fanout exchange routes messages to all of the queues that are bound to it.
* **Topic:** The topic exchange does a wildcard match between the routing key and the routing pattern specified in the binding.
* **Headers** Headers exchanges use the message header attributes for routing.

#### RabbitMQ and server concepts

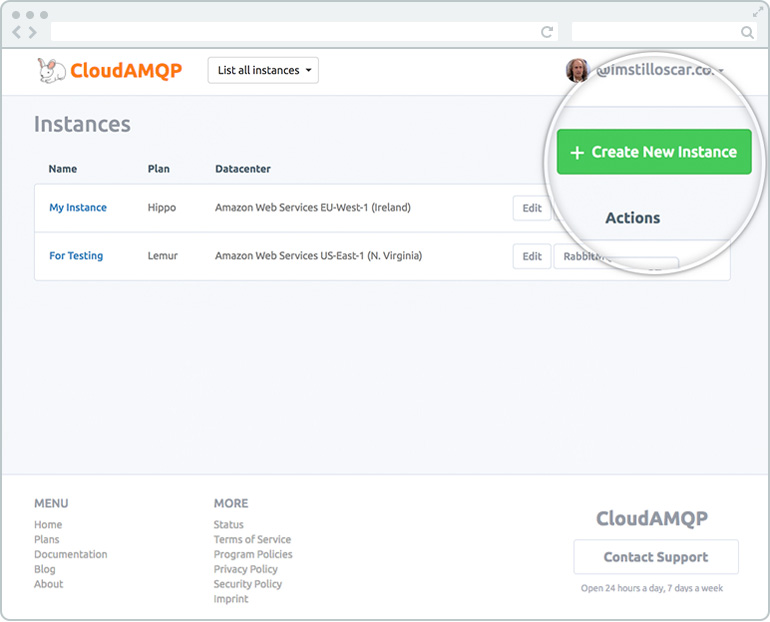
Here are some important concepts that needs to be described before we dig deeper into RabbitMQ. The default virtual host, the default user, and the default permissions are used in the examples that follow, but it is still good to have a feeling of what it is.

* **Producer:** Application that sends the messages.
* **Consumer:** Application that receives the messages.
* **Queue:** Buffer that stores messages.
* **Message:** Information that is sent from the producer to a consumer through RabbitMQ.
* **Connection:** A connection is a TCP connection between your application and the RabbitMQ broker.
* **Channel:** A channel is a virtual connection inside a connection. When you are publishing or consuming messages from a queue - it's all done over a channel.
* **Exchange:** Receives messages from producers and pushes them to queues depending on rules defined by the exchange type. In order to receive messages, a queue needs to be bound to at least one exchange.
* **Binding:** A binding is a link between a queue and an exchange.
* **Routing key:** The routing key is a key that the exchange looks at to decide how to route the message to queues. The routing key is like an *address* for the message.
* **AMQP:** AMQP (Advanced Message Queuing Protocol) is the protocol used by RabbitMQ for messaging.
* **Users:** It is possible to connect to RabbitMQ with a given username and password. Every user can be assigned permissions such as rights to read, write and configure privileges within the instance. Users can also be assigned permissions to specific virtual hosts.
* **Vhost, virtual host:** A Virtual host provide a way to segregate applications using the same RabbitMQ instance. Different users can have different access privileges to different vhost and queues and exchanges can be created so they only exists in one vhost.

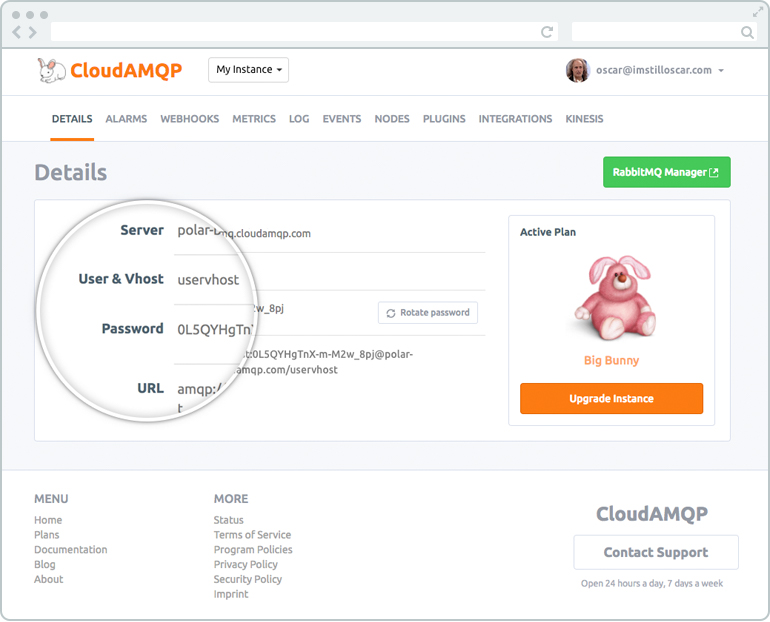
At the beginning of this article series we had one producer (the website application) and one consumer (the PDF processing application). If the PDF processing application crashes, or if there is a lot of PDF requests coming in the same time, messages would continue to stack up in the queue until the consumer starts again. It would then process all the messages, one by one.

### Set up a RabbitMQ instance

To be able to follow this guide you need to set up a CloudAMQP instance or you need to download and install RabbitMQ. CloudAMQP is a hosted RabbitMQ solution, meaning that all you need to do is sign up for an account and create an instance. You do not need to set up and install RabbitMQ or care about cluster handling, CloudAMQP will do that for you. CloudAMQP can be used for free with the plan little lemur. Go to the [plan](https://www.cloudamqp.com/plans.html) page and sign up for any plan and create an instance.



When your instance is created, press on details for your instance to find your username, password and connection URL for your cloud hosted RabbitMQ instance.



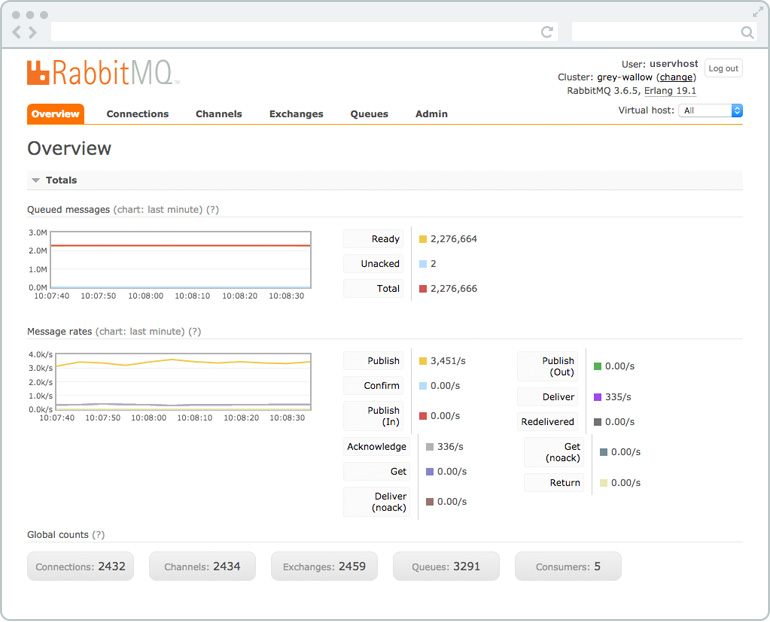
### Getting started with RabbitMQ

Immediately after a RabbitMQ instance has been created it is possible to send a message cross languages, platforms and OS. This way of handling messages decouple your processes and creates a highly scalable system. You can now start by opening the management interface to get an overview of your RabbitMQ server.

#### The Management Interface - Management and Monitoring

RabbitMQ provides a web UI for management and monitoring of your RabbitMQ server. The RabbitMQ management interface is enabled by default in CloudAMQP and a link can be found on the details page for your CloudAMQP instance.

From the management interface, it is possible to handle, create, delete and list queues. It is possible to monitor queue length, check message rate, change and add users permissions much more.



More information about the management interface can be found in [Part 3 - The management interface.](https://www.cloudamqp.com/blog/2015-05-27-part3-rabbitmq-for-beginners_the-management-interface.html)

#### Publish and subscribe messages

RabbitMQ speaks a protocol called AMQP by default. To be able to communicate with RabbitMQ you need a library that understands the same protocol as RabbitMQ. You need to download the client-library for the programming language that you intend to use for your applications. A client-library is an applications programming interface (API) for use in writing client applications. A client library has several methods that can be used, in this case to communicate with RabbitMQ. The methods should be used when you, for example, connect to the RabbitMQ broker (using the given parameters, host name, port number, etc) or when you declare a queue or an exchange. There is a choice of libraries for almost every programming language.

Steps to follow when setting up a connection and publishing a message/consuming a message:

1. First of all, we need to set up/create a connection object. Here, the username, password, connection URL, port etc, will be specified. A TCP connection will be set up between the application and RabbitMQ when the *start* method is called.
2. Secondly a channel needs to be opened. A channel needs to be created in the TCP connection. The connection interface can be used to open a channel and when the channel is opened it can be used to send and receive messages.
3. Declare/create a queue. Declaring a queue will cause it to be created if it does not already exist. All queues needs to be declared before they can be used.
4. **In subscriber/consumer:** Set up exchanges and bind a queue to an exchange. All exchanges needs to be declared before they can be used. An exchange accepts messages from a producer application and routes them to message queues. For messages to be routed to queues, queues need to be bound to an exchange.
5. **In publisher:** Publish a message to an exchange   
   **In subscriber/consumer:** Consume a message from a queue.
6. Close the channel and the connection.

#### Sample code

Sample code will be given in the part 2, starting with [Part 2.1 - Ruby,](https://www.cloudamqp.com/blog/2015-05-19-part2-1-rabbitmq-for-beginners_example-and-sample-code-ruby.html) followed by [Part 2.2 - Node.js,](https://www.cloudamqp.com/blog/2015-05-19-part2-2-rabbitmq-for-beginners_example-and-sample-code-node-js.html) and [Part 2.3 Python,](https://www.cloudamqp.com/blog/2015-05-21-part2-3-rabbitmq-for-beginners_example-and-sample-code-python.html) It is possible to have different programming languages on different parts of the system. The publisher could, for example, be written in node.js and the subscriber in Python.

Hope this article helped you get some understanding about RabbitMQ!

https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html