(/)

OFFICIAL REPOSITORY

consul (/r/_/consul/) ☆

Last pushed: 6 days ago

Repo Info (/_/consul/)

Short Description

Consul is a datacenter runtime that provides service discovery, configuration, and orchestration.

Full Description

Supported tags and respective `Dockerfile` links

- `1.3.1` (*0.X/Dockerfile*) (https://github.com/hashicorp/docker-consul/blob/2da194710e10e02277cd6b4fb38fb1d1f3ec2fc7/0.X/Dockerfile)
- `1.4.0`, `latest` (*0.X/Dockerfile*) (https://github.com/hashicorp/docker-consul/blob/a9fc2295556404f409d16de58be611b77079a870/0.X/Dockerfile)

Quick reference

- **Where to get help**:
  the Docker Community Forums (https://forums.docker.com/), the Docker Community Slack (https://blog.docker.com/2016/11/introducing-docker-community-directory-docker-community-slack/), or Stack Overflow (https://stackoverflow.com/search?tab=newest&q=docker)

- **Where to file issues**:
  https://github.com/hashicorp/docker-consul/issues (https://github.com/hashicorp/docker-consul/issues)

- **Maintained by**:
  HashiCorp (https://github.com/hashicorp/docker-consul)

- **Supported architectures**: (more info (https://github.com/docker-library/official-images#architectures-other-than-amd64))
  `amd64` (https://hub.docker.com/r/amd64/consul/), `arm32v6` (https://hub.docker.com/r/arm32v6/consul/), `arm64v8` (https://hub.docker.com/r/arm64v8/consul/), `i386` (https://hub.docker.com/r/i386/consul/)

- **Published image artifact details**:
  repo-info repo's `repos/consul/` directory (https://github.com/docker-library/repo-info/blob/master/repos/consul) (history (https://github.com/docker-library/repo-info/commits/master/repos/consul))
  (image metadata, transfer size, etc)

- **Image updates**:
  official-images PRs with label `library/consul` (https://github.com/docker-library/official-images/pulls?q=label%3Alibrary%2Fconsul)
  official-images repo's `library/consul` file (https://github.com/docker-library/official-images/blob/master/library/consul) (history (https://github.com/docker-library/official-images/commits/master/library/consul))

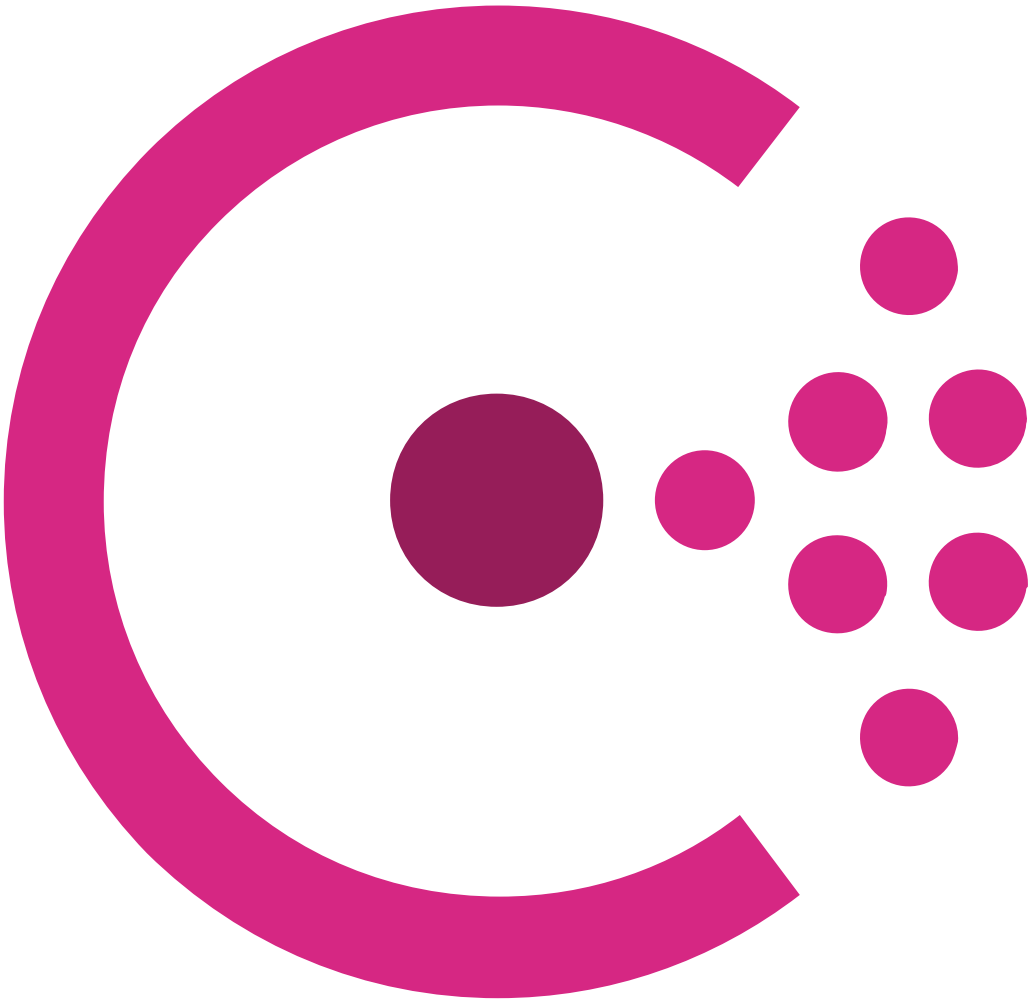
- **Source of this description**:
  docs repo's `consul/` directory (https://github.com/docker-library/docs/tree/master/consul) (history (https://github.com/docker-library/docs/commits/master/consul))

- **Supported Docker versions**:
  the latest release (https://github.com/docker/docker-ce/releases/latest) (down to 1.6 on a best-effort basis)

Consul

Consul is a distributed, highly-available, and multi-datacenter aware tool for service discovery, configuration, and orchestration. Consul enables rapid deployment, configuration, and maintenance of service-oriented architectures at massive scale. For more information, please see:

- Consul documentation (https://www.consul.io/)
- Consul on GitHub (https://github.com/hashicorp/consul)



Consul and Docker

Consul has several moving parts so we'll start with a brief introduction to Consul's architecture and then detail how Consul interacts with Docker. Please see the Consul Architecture (https://www.consul.io/docs/internals/architecture.html) guide for more detail on all these concepts.

Each host in a Consul cluster runs the Consul agent, a long running daemon that can be started in client or server mode. Each cluster has at least 1 agent in server mode, and usually 3 or 5 for high availability. The server agents participate in a consensus protocol (https://www.consul.io/docs/internals/consensus.html), maintain a centralized view of the cluster's state, and respond to queries from other agents in the cluster. The rest of the agents in client mode participate in a gossip protocol (https://www.consul.io/docs/internals/gossip.html) to discover other agents and check them for failures, and they forward queries about the cluster to the server agents.

Applications running on a given host communicate only with their local Consul agent, using its HTTP APIs or DNS interface. Services on the host are also registered with the local Consul agent, which syncs the information with the Consul servers. Doing the most basic DNS-based service discovery using Consul, an application queries for `foo.service.consul` and gets a randomly shuffled subset of all the hosts providing service "foo". This allows applications to locate services and balance the load without any intermediate proxies. Several HTTP APIs are also available for applications doing a deeper integration with Consul's service discovery capabilities, as well as its other features such as the key/value store.

These concepts also apply when running Consul in Docker. Typically, you'll run a single Consul agent container on each host, running alongside the Docker daemon. You'll also need to configure some of the agents as servers (at least 3 for a basic HA setup). Consul should always be run with `--net=host` in Docker because Consul's consensus and gossip protocols are sensitive to delays and packet loss, so the extra layers involved with other networking types are usually undesirable and unnecessary. We will talk more about this below.

We don't cover Consul's multi-datacenter capability here, but as long as `--net=host` is used, there should be no special considerations for Docker.

Using the Container

We chose Alpine as a lightweight base with a reasonably small surface area for security concerns, but with enough functionality for development, interactive debugging, and useful health, watch, and exec scripts running under Consul in the container. As of Consul 0.7, the image also includes `curl` since it is so commonly used for health checks.

Consul always runs under dumb-init (https://github.com/Yelp/dumb-init), which handles reaping zombie processes and forwards signals on to all processes running in the container. We also use gosu (https://github.com/tianon/gosu) to run Consul as a non-root "consul" user for better security. These binaries are all built by HashiCorp and signed with our GPG key (https://www.hashicorp.com/security.html), so you can verify the signed package used to build a given base image.

Running the Consul container with no arguments will give you a Consul server in development mode (https://www.consul.io/docs/agent/options.html#_dev). The provided entry point script will also look for Consul subcommands and run `consul` as the correct user and with that subcommand. For example, you can execute `docker run consul members` and it will run the `consul members` command inside the container. The entry point also adds some special configuration options as detailed in the sections below when running the `agent` subcommand. Any other command gets `exec` -ed inside the container under `dumb-init` .

The container exposes `VOLUME /consul/data`, which is a path were Consul will place its persisted state. This isn't used in any way when running in development mode. For client agents, this stores some information about the cluster and the client's health checks in case the container is restarted. For server agents, this stores the client information plus snapshots and data related to the consensus algorithm and other state like Consul's key/value store and catalog. For servers it is highly desirable to keep this volume's data around when restarting containers to recover from outage scenarios. If this is bind mounted then ownership will be changed to the consul user when the container starts.

The container has a Consul configuration directory set up at `/consul/config` and the agent will load any configuration files placed here by binding a volume or by composing a new image and adding files. Alternatively, configuration can be added by passing the configuration JSON via environment variable `CONSUL_LOCAL_CONFIG`. If this is bind mounted then ownership will be changed to the consul user when the container starts.

Since Consul is almost always run with `--net=host` in Docker, some care is required when configuring Consul's IP addresses. Consul has the concept of its cluster address as well as its client address. The cluster address is the address at which other Consul agents may contact a given agent. The client address is the address where other processes on the host contact Consul in order to make HTTP or DNS requests. You will typically need to tell Consul what its cluster address is when starting so that it binds to the correct interface and advertises a workable interface to the rest of the Consul agents. You'll see this in the examples below as the `-bind=<external ip>` argument to Consul.

The entry point also includes a small utility to look up a client or bind address by interface name. To use this, set the `CONSUL_CLIENT_INTERFACE` and/or `CONSUL_BIND_INTERFACE` environment variables to the name of the interface you'd like Consul to use and a `-client=<interface ip>` and/or `-bind=<interface ip>` argument will be computed and passed to Consul at startup.

Running Consul for Development

```
$ docker run -d --name=dev-consul -e CONSUL_BIND_INTERFACE=eth0 cor
```

This runs a completely in-memory Consul server agent with default bridge networking and no services exposed on the host, which is useful for development but should not be used in production. For example, if that server is running at internal address 172.17.0.2, you can run a three node cluster for development by starting up two more instances and telling them to join the first node.

```
$ docker run -d -e CONSUL_BIND_INTERFACE=eth0 consul agent -dev -jo
... server 2 starts
$ docker run -d -e CONSUL_BIND_INTERFACE=eth0 consul agent -dev -jo
... server 3 starts
```

Then we can query for all the members in the cluster by running a Consul CLI command in the first container:

```
$ docker exec -t dev-consul consul members
Node          Address          Status  Type    Build  Protocol  DC
579db72c1ae1  172.17.0.3:8301  alive   server  0.6.3  2         dc1
93fe2309ef19  172.17.0.4:8301  alive   server  0.6.3  2         dc1
c9caabfd4c2a  172.17.0.2:8301  alive   server  0.6.3  2         dc1
```

Remember that Consul doesn't use the data volume in this mode - once the container stops all of your state will be wiped out, so please don't use this mode for production. Running completely on the bridge network with the development server is useful for testing multiple instances of Consul on a single machine, which is normally difficult to do because of port conflicts.

Development mode also starts a version of Consul's web UI on port 8500. This can be added to the other Consul configurations by supplying the `-ui` option to Consul on the command line. The web assets are bundled inside the Consul binary in the container.

Running Consul Agent in Client Mode

```
$  docker run -d --net=host -e 'CONSUL_LOCAL_CONFIG={"leave_on_term
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
         Node name: 'linode'
        Datacenter: 'dc1'
            Server: false (bootstrap: false)
       Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RP
      Cluster Addr: <external ip> (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
             Atlas: <disabled>
...
```

This runs a Consul client agent sharing the host's network and advertising the external IP address to the rest of the cluster. Note that the agent defaults to binding its client interfaces to 127.0.0.1, which is the host's loopback interface. This would be a good configuration to use if other containers on the host also use `--net=host`, and it also exposes the agent to processes running directly on the host outside a container, such as HashiCorp's Nomad.

The `-retry-join` parameter specifies the external IP of one other agent in the cluster to use to join at startup. There are several ways to control how an agent joins the cluster, see the agent configuration (https://www.consul.io/docs/agent/options.html) guide for more details on the `-join`, `-retry-join`, and `-atlas-join` options.

Note also we've set `leave_on_terminate` (https://www.consul.io/docs/agent/options.html#leave_on_terminate) using the `CONSUL_LOCAL_CONFIG` environment variable. This is recommended for clients to and will be defaulted to `true` in Consul 0.7 and later, so this will no longer be necessary.

At startup, the agent will read config JSON files from `/consul/config`. Data will be persisted in the `/consul/data` volume.

Here are some example queries on a host with an external IP of 66.175.220.234:

```
$ curl http://localhost:8500/v1/health/service/consul?pretty
[
    {
        "Node": {
            "Node": "linode",
            "Address": "66.175.220.234",
...
```

```
$ dig @localhost -p 8600 consul.service.consul
; <<>> DiG 9.9.5-3ubuntu0.7-Ubuntu <<>> @localhost -p 8600 consul.s
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61616
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL:
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;consul.service.consul.         IN      A

;; ANSWER SECTION:
consul.service.consul.  0       IN      A       66.175.220.234
...
```

If you want to expose the Consul interfaces to other containers via a different network, such as the bridge network, use the `-client` option for Consul:

```
docker run -d --net=host consul agent -bind=<external ip> -client=<
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
         Node name: 'linode'
        Datacenter: 'dc1'
            Server: false (bootstrap: false)
       Client Addr: <bridge ip> (HTTP: 8500, HTTPS: -1, DNS: 8600,
      Cluster Addr: <external ip> (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
             Atlas: <disabled>
...
```

With this configuration, Consul's client interfaces will be bound to the bridge IP and available to other containers on that network, but not on the host network. Note that we still keep the cluster address out on the host network for performance. Consul will also accept the `-client=0.0.0.0` option to bind to all interfaces.

Running Consul Agent in Server Mode

```
$ docker run -d --net=host -e 'CONSUL_LOCAL_CONFIG={"skip_leave_on_
```

This runs a Consul server agent sharing the host's network. All of the network considerations and behavior we covered above for the client agent also apply to the server agent. A single server on its own won't be able to form a quorum and will be waiting for other servers to join.

Just like the client agent, the `-retry-join` parameter specifies the external IP of one other agent in the cluster to use to join at startup. There are several ways to control how an agent joins the cluster, see the agent configuration (https://www.consul.io/docs/agent/options.html) guide for more details on the `-join`, `-retry-join`, and `-atlas-join` options. The server agent also consumes a `-bootstrap-expect` option that specifies how many server agents to watch for before bootstrapping the cluster for the first time. This provides an easy way to get an orderly startup with a new cluster. See the agent configuration (https://www.consul.io/docs/agent/options.html) guide for more details on the `-bootstrap` and `-bootstrap-expect` options.

Note also we've set `skip_leave_on_interrupt` (https://www.consul.io/docs/agent/options.html#skip_leave_on_interrupt) using the `CONSUL_LOCAL_CONFIG` environment variable. This is recommended for servers and will be defaulted to `true` in Consul 0.7 and later, so this will no longer be necessary.

At startup, the agent will read config JSON files from `/consul/config`. Data will be persisted in the `/consul/data` volume.

Once the cluster is bootstrapped and quorum is achieved, you must use care to keep the minimum number of servers operating in order to avoid an outage state for the cluster. The deployment table in the consensus (https://www.consul.io/docs/internals/consensus.html) guide outlines the number of servers required for different configurations. There's also an adding/removing servers (https://www.consul.io/docs/guides/servers.html) guide that describes that process, which is relevant to Docker configurations as well. The outage recovery (https://www.consul.io/docs/guides/outage.html) guide has steps to perform if servers are permanently lost. In general it's best to restart or replace servers one at a time, making sure servers are healthy before proceeding to the next server.

Exposing Consul's DNS Server on Port 53

By default, Consul's DNS server is exposed on port 8600. Because this is cumbersome to configure with facilities like `resolv.conf`, you may want to expose DNS on port 53. Consul 0.7 and later supports this by setting an environment variable that runs `setcap` on the Consul binary, allowing it to bind to privileged ports. Note that not all Docker storage backends support this feature (notably AUFS).

Here's an example:

```
$ docker run -d --net=host -e 'CONSUL_ALLOW_PRIVILEGED_PORTS=' cons
```

This example also includes a recursor configuration that uses Google's DNS servers for non-Consul lookups. You may want to adjust this based on your particular DNS configuration. If you are binding Consul's client interfaces to the host's loopback address, then you should be able to configure your host's `resolv.conf` to route DNS requests to Consul by including "127.0.0.1" as the primary DNS server. This would expose Consul's DNS to all applications running on the host, but due to Docker's built-in DNS server, you can't point to this directly from inside your containers; Docker will issue an error message if you attempt to do this. You must configure Consul to listen on a non-localhost address that is reachable from within other containers.

Once you bind Consul's client interfaces to the bridge or other network, you can use the `--dns` option in your *other containers* in order for them to use Consul's DNS server, mapped to port 53. Here's an example:

```
$ docker run -d --net=host -e 'CONSUL_ALLOW_PRIVILEGED_PORTS=' cons
```

Now start another container and point it at Consul's DNS, using the bridge address of the host:

```
$ docker run -i --dns=<bridge ip> -t ubuntu sh -c "apt-get update &
...
;; ANSWER SECTION:
consul.service.consul.  0       IN      A       66.175.220.234
...
```

In the example above, adding the bridge address to the host's `/etc/resolv.conf` file should expose it to all containers without running with the `--dns` option.

Service Discovery with Containers

There are several approaches you can use to register services running in containers with Consul. For manual configuration, your containers can use the local agent's APIs to register and deregister themselves, see the Agent API (https://www.consul.io/docs/agent/http/agent.html) for more details. Another strategy is to create a derived Consul container for each host type which includes JSON config files for Consul to parse at startup, see Services (https://www.consul.io/docs/agent/services.html) for more information. Both of these approaches are fairly cumbersome, and the configured services may fall out of sync if containers die or additional containers are started.

If you run your containers under HashiCorp's Nomad (https://www.nomadproject.io/) scheduler, it has first class support for Consul (https://www.nomadproject.io/docs/jobspec/servicediscovery.html). The Nomad agent runs on each host alongside the Consul agent. When jobs are scheduled on a given host, the Nomad agent automatically takes care of syncing the Consul agent with the service information. This is very easy to manage, and even services on hosts running outside of Docker containers can be managed by Nomad and registered with Consul. You can find out more about running Docker under Nomad in the Docker Driver (https://www.nomadproject.io/docs/drivers/docker.html) guide.

Other open source options include Registrator (http://gliderlabs.com/registrator/latest/) from Glider Labs and ContainerPilot (https://www.joyent.com/containerpilot) from Joyent. Registrator works by running a Registrator instance on each host, alongside the Consul agent. Registrator monitors the Docker daemon for container stop and start events, and handles service registration with Consul using the container names and exposed ports as the service information. ContainerPilot manages service registration using tooling running inside the container to register services with Consul on start, manage a Consul TTL health check while running, and deregister services when the container stops.

Running Health Checks in Docker Containers

Consul has the ability to execute health checks inside containers. If the Docker daemon is exposed to the Consul agent and the `DOCKER_HOST` environment variable is set, then checks can be configured with the Docker container ID to execute in. See the health checks (https://www.consul.io/docs/agent/checks.html) guide for more details.

License

View license information (https://raw.githubusercontent.com/hashicorp/consul/master/LICENSE) for the software contained in this image.

As with all Docker images, these likely also contain other software which may be under other licenses (such as Bash, etc from the base distribution, along with any direct or indirect dependencies of the primary software being contained).

Some additional license information which was able to be auto-detected might be found in the `repo-info` repository's `consul/` directory (https://github.com/docker-library/repo-info/tree/master/repos/consul).

As for any pre-built image usage, it is the image user's responsibility to ensure that any use of this image complies with any relevant licenses for all software contained within.

Docker Pull Command

```
docker pull consul
```