Introduction

Service discovery registers a service and publishes its connectivity information so that other services are aware of how to connect to the service. As applications move toward microservices and service-oriented architectures, service discovery has become an integral part of any distributed system, increasing the operational complexity of these environments.

Docker Enterprise Edition (Docker EE) includes service discovery and load balancing capabilities to aid the devops initiatives across any organization. Service discovery and load balancing make it easy for developers to create applications that can dynamically discover each other. Also, these features simplify the scaling of applications by operations engineers.

Docker uses a concept called **services** to deploy applications. Services consist of containers created from the same image. Each service consists of tasks that execute on worker nodes and define the state of the application. When deploying a service, a service definition is included upon service creation. The service definition consists of information that includes, among other things, the containers that comprise the service, which ports are published, which networks are attached, and the number of replicas. All of these tasks together make up the desired state of the service. If a node fails a health check or if a specific service task defined in a service definition fails a health check, then the cluster reconciles the service state to another healthy node. Docker EE includes service discovery, load balancing, scaling, and reconciliation events so that this orchestration works seamlessly.

What You Will Learn

This reference architecture covers the solutions that Docker EE 2.0 provides in the topic areas of service discovery and load balancing for both swarm mode as well as Kubernetes workloads. In swarm mode, Docker uses DNS for service discovery as services are created, and different routing meshes are built into Docker to ensure your applications remain highly available. The release of UCP 3.0 introduces a versatile and an enhanced version of application layer (Layer 7) routing mesh called the **Interlock Proxy** that routes HTTP traffic based on DNS hostname. After reading this document, you will have a good understanding of how Interlock Proxy works and how it integrates with the other service discovery and load balancing features native to Docker.

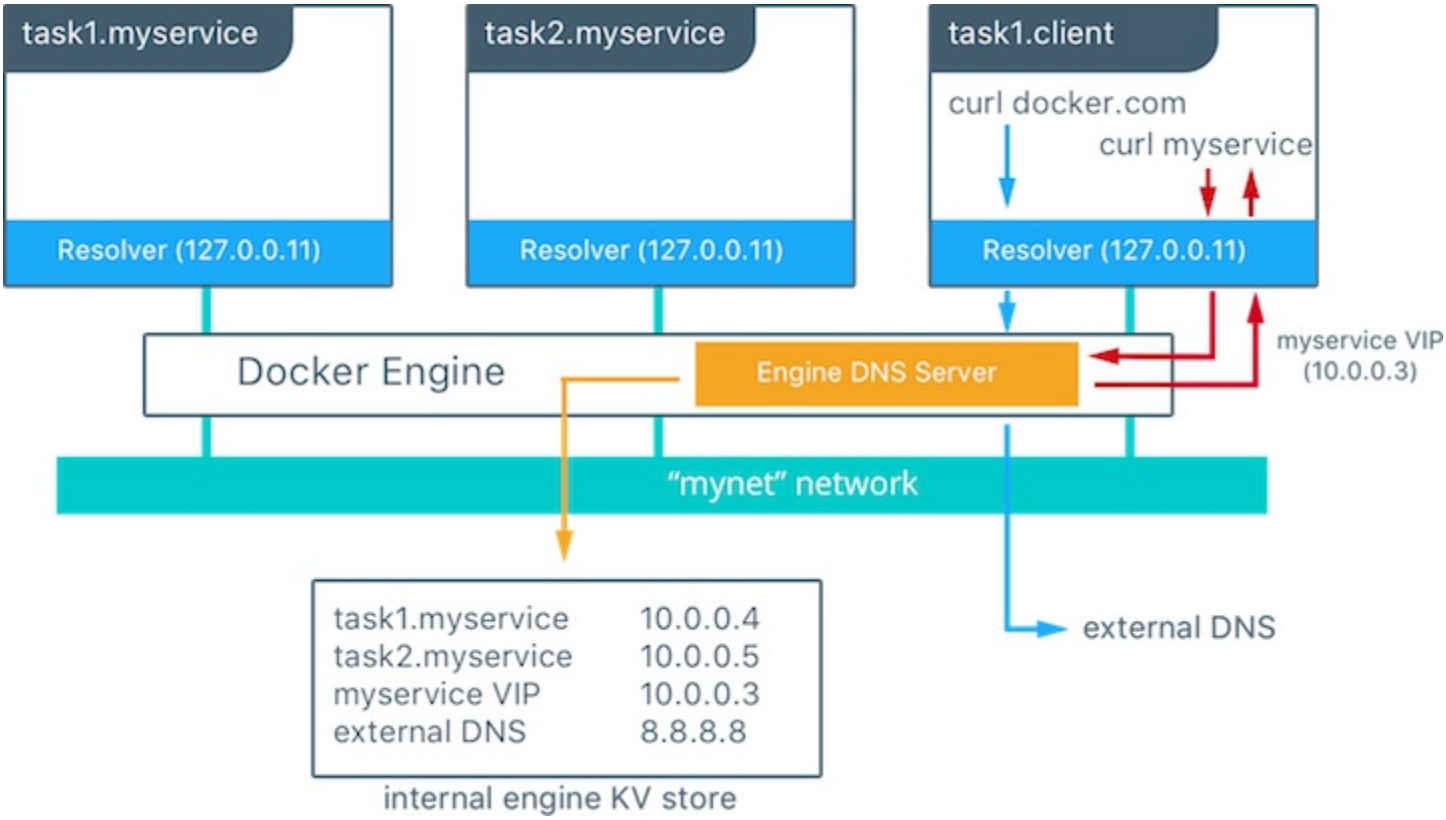
Additionally, UCP 3.0 introduces enterprise support for using Kubernetes as the orchestrator of your application workloads. This document will also provide a good understanding of how to use Kubernetes resource objects within Docker EE to deploy, run, and manage application workloads.

Service Discovery with DNS

Docker uses embedded DNS to provide service discovery for containers running on a single Docker engine and `tasks` running in a Docker swarm. Docker engine has an internal DNS server that provides name resolution to all of the containers on the host in user-defined bridge, overlay, and MACVLAN networks. Each Docker container (or `task` in swarm mode) has a DNS resolver that forwards DNS queries to the Docker engine, which acts as a DNS server. The Docker engine then checks if the DNS query belongs to a container or `service` on each network that the requesting container belongs to. If it does, then the Docker engine looks up the IP address that matches the name of a container, `task`, or`service` in its key-value store and returns that IP or `service` Virtual IP (VIP) back to the requester.

Service discovery is *network-scoped*, meaning only containers or tasks that are on the same network can use the embedded DNS functionality. Containers not on the same network cannot resolve each others' addresses. Additionally, only the nodes that have containers or tasks on a particular network store that network's DNS entries. This promotes security and performance.

If the destination container or `service` and the source container are not on the same network, the Docker engine forwards the DNS query to the default DNS server.



In this example, there is a service of two containers called `myservice`. A second service (`client`) exists on the same network. The `client` executes two `curl` operations for `docker.com` and `myservice`. These are the resulting actions:

- DNS queries are initiated by `client` for `docker.com` and `myservice`.
- The container's built-in resolver intercepts the DNS queries on `127.0.0.11:53` and sends them to Docker Engine's DNS server.
- `myservice` resolves to the Virtual IP (VIP) of that service which is internally load balanced to the individual task IP addresses. Container names are resolved as well, albeit directly to their IP addresses.
- `docker.com` does not exist as a service name in the `mynet` network, so the request is forwarded to the configured default DNS server.

Internal Load Balancing

When services are created in a Docker swarm cluster, they are automatically assigned a Virtual IP (VIP) that is part of the service's network. The VIP is returned when resolving the service's name. Traffic to the VIP is automatically sent to all healthy tasks of that service across the overlay network. This approach avoids any client-side load balancing because only a single IP is returned to the client. Docker takes care of routing and equally distributes the traffic across the healthy service tasks.

To get the VIP of a service, run the `docker service inspect myservice` command like so:

```
# Create an overlay network called mynet
$ docker network create -d overlay mynet
a59umzkdj2r0ua7x8jxd84dhr

# Create myservice with 2 replicas as part of that network
$ docker service create --network mynet --name myservice --replicas 2 busybox ping localhost
8t5r8cr0f0h6k2c3k7ih4l6f5

# Get the VIP that was created for that service
$ docker service inspect myservice
...

"VirtualIPs": [
                {
                    "NetworkID": "a59umzkdj2r0ua7x8jxd84dhr",
                    "Addr": "10.0.0.3/24"
                },
              ]
```

> **Note:** DNS round robin (DNS RR) load balancing is another load balancing option for services (configured with `--endpoint-mode`). In DNS RR mode, a VIP is not created for each service. The Docker DNS server resolves a service name to individual container IPs in round robin fashion.

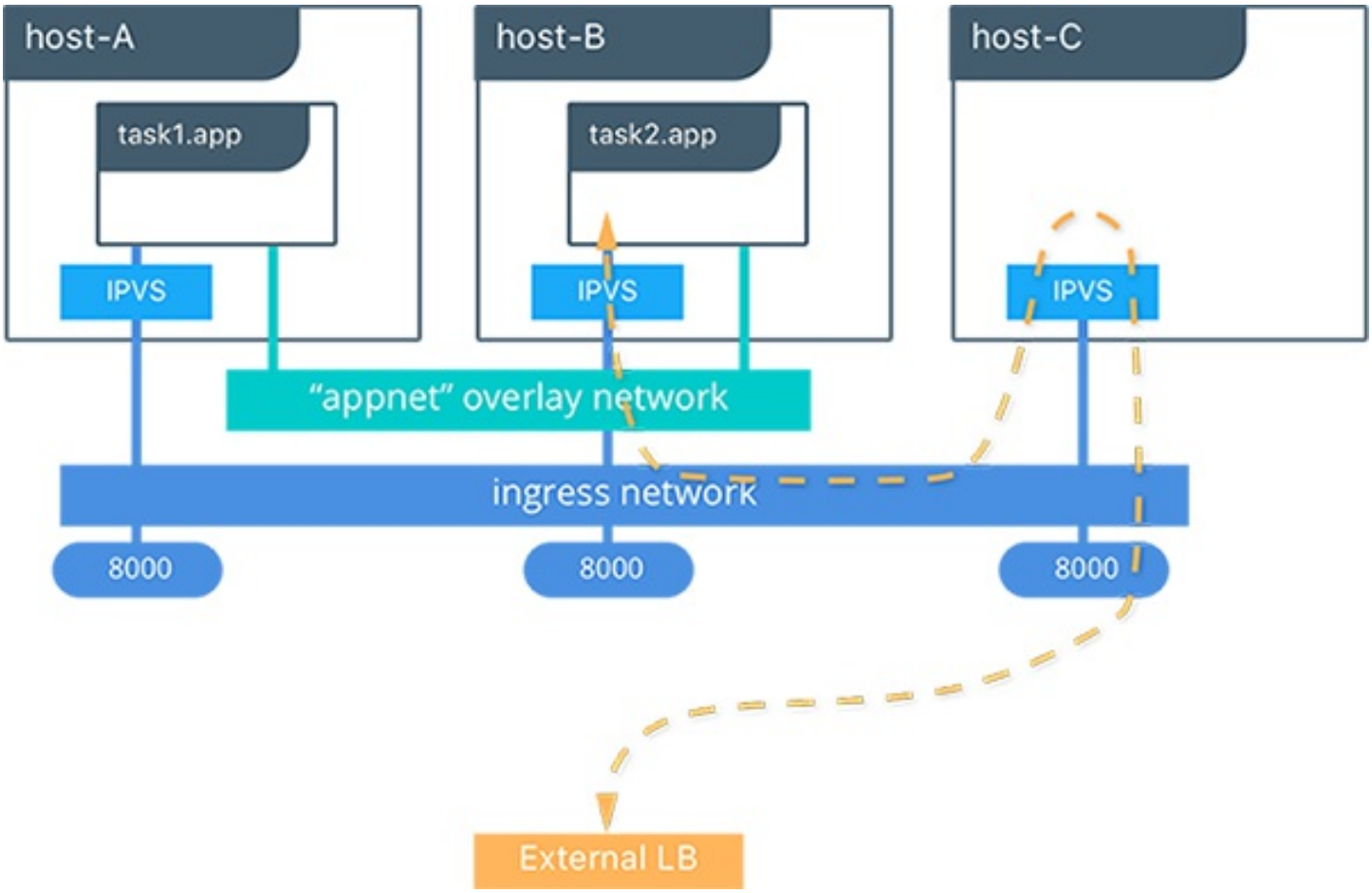
External Load Balancing (Swarm Mode Routing Mesh)

You can expose services externally by using the `--publish` flag when creating or updating the service. Publishing ports in Docker swarm mode means that every node in your cluster is listening on that port, but what happens if the service's task isn't on the node that is listening on that port?

This is where routing mesh comes into play. Introduced in Docker Engine 1.12, routing mesh combines `ipvs` and `iptables` to create a powerful cluster-wide transport-layer (L4) load balancer. It allows all the swarm nodes to accept connections on the services published ports. When any swarm node receives traffic destined to the published TCP/UDP port of a running `service`, it forwards the traffic to the service's VIP using a pre-defined overlay network called `ingress`. The `ingress` network behaves similarly to other overlay networks, but its sole purpose is to transport mesh routing traffic from external clients to cluster services. It uses the same VIP-based internal load balancing as described in the previous section.

Once you launch services, you can create an external DNS record for your applications and map it to any or all Docker swarm nodes. You do not need to worry about where your container is running as all nodes in your cluster look as one with the routing mesh routing feature.

```
# Create a service with two replicas and export port 8000 on the cluster
$ docker service create --name app --replicas 2 --network appnet --publish 8000:80 nginx
```



This diagram illustrates how the routing mesh works.

- A service is created with two replicas, and it is port mapped externally to port `8000`.
- The routing mesh exposes port `8000` on each host in the cluster.
- Traffic destined for the app can enter on any host. In this case the external LB sends the traffic to a host without a service replica.
- The kernel's IPVS load balancer redirects traffic on the `ingress` overlay network to a healthy service replica.

The Swarm Layer 7 Routing (Interlock Proxy)

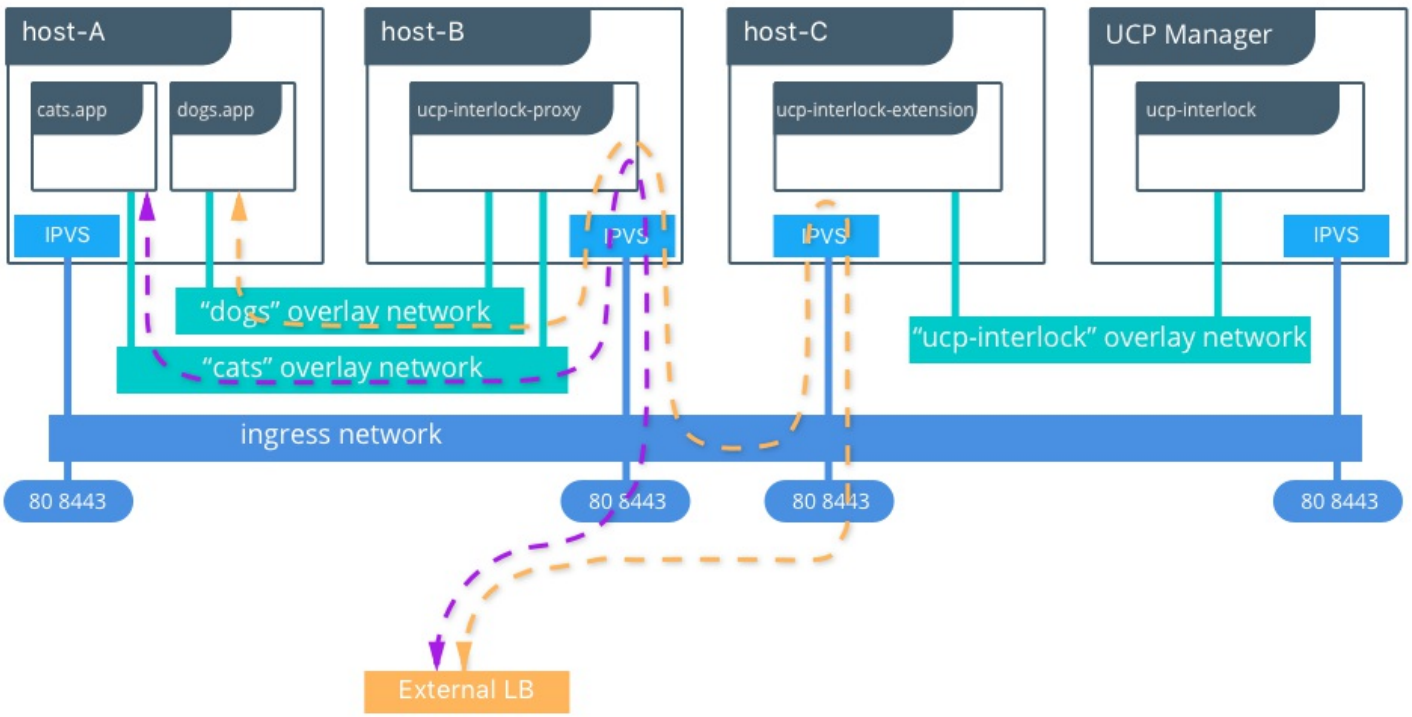
The swarm mode routing mesh is great for transport-layer routing. It routes to services using the service's published ports. But what if you wanted to route traffic to services based on hostname instead? The "Swarm Layer 7 Routing (Interlock)" (called HRM or just interlock in previous versions) is a new feature that enables service discovery on the application layer (L7). This Layer 7 Routing extends upon the swarm mode routing mesh by adding application layer capabilities such as inspecting the HTTP header. Interlock and swarm mode routing meshes are used together for flexible and robust service delivery. The addition of Interlock allows for each service to be accessible via a DNS label passed to the service. As the service scales horizontally and more replicas are added, the service uses round-robin load balancing as well.

The Interlock Proxy works by using the `HTTP/1.1` header field definition. Every `HTTP/1.1` TCP request contains a `Host:` header. A HTTP request header can be viewed using `curl`:

```
$ curl -v docker.com
* Rebuilt URL to: docker.com/
*   Trying 52.20.149.52...
* Connected to docker.com (52.20.149.52) port 80 (#0)
> GET / HTTP/1.1
> Host: docker.com
> User-Agent: curl/7.49.1
> Accept: */*
```

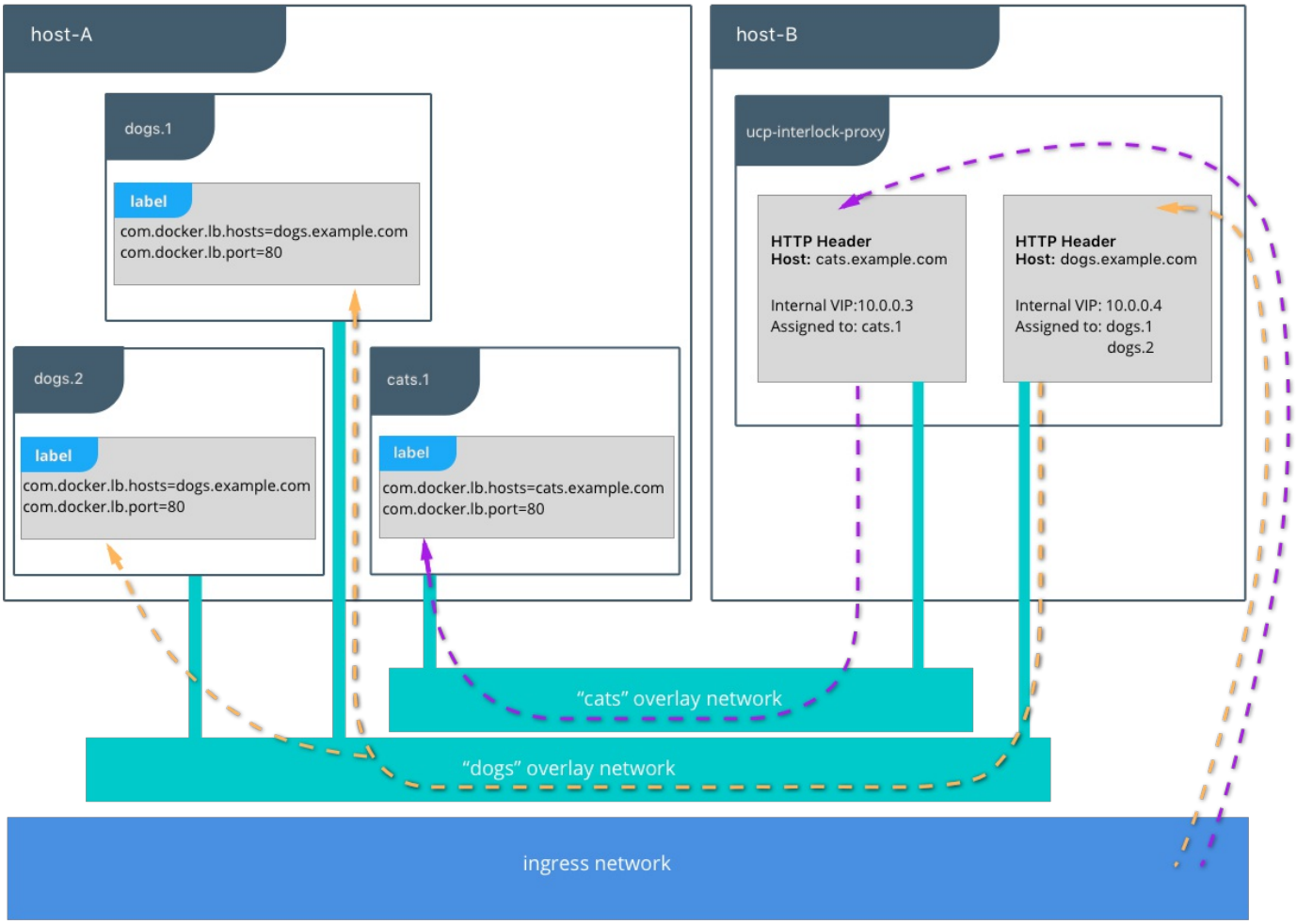
When using Interlock with HTTP requests, both the swarm mode routing mesh and Interlock are used in tandem. When a service is created using the `com.docker.ucp.mesh.http` label, the Interlock configuration is updated to route all HTTP requests that contain the `Host:` header specified in the `com.docker.ucp.mesh.http` label to route to the VIP of the newly created services. Since the Interlock is a service, it is accessible on any node in the cluster using the configured published port.

The following is an overview diagram to show how the swarm mode routing mesh and Interlock work together.



- The traffic comes in from the external load balancer into the swarm mode routing mesh.
- The `ucp-interlock-proxy` service is configured to listen on port 80 and 8443, so any request to port 80 or 8443 on the UCP cluster will hit this service first.
- All services attached to a network that is enabled for "Hostname-based routing" can utilize the Interlock proxy to have traffic routed based on the HTTP `Host:` header.

The following graphic represents a closer look of the previous diagram. You can see how Interlock works under the hood.



- Traffic comes in through the swarm mode routing mesh on the `ingress` network to the Interlock Proxy service's published port.
- As services are created, they are assigned VIPs on the swarm mode routing mesh (L4).
- There are three services within Interlock that communicate with one another. All the services are automatically deployed and updated in response to changes to application services.
  - The core service is called `ucp-interlock`. This listens to the Docker remote API for events and configures an upstream service that is accessed by another service called `ucp-interlock-extension`.
  - The extension service is called `ucp-interlock-extension`. This service queries the core `ucp-interlock` service and uses the response information from that service to generate the configuration file appropriate for the proxy service called `ucp-interlock-proxy`. The configuration file is generated in the form of a Docker config object which will be used by the proxy service.
  - The proxy service is called `ucp-interlock-proxy`. This is the reverse proxy and is responsible for handling the actual requests for application services. The proxy uses the configuration generated by the corresponding extension service `ucp-interlock-extension`.
- The `ucp-interlock-proxy` service receives the TCP packet and inspects the HTTP header.
  - Services that contain the label `com.docker.lb.hosts` are checked to see if they match the HTTP `Host:` header.
  - If a `Host:` header and service label match, then the value of the label `com.docker.lb.port` is queried for that service. This instructs what port the `ucp-interlock-proxy` should use to access the application service.

    - Traffic is routed to the service's VIP on its port using the swarm mode routing mesh (L4).
- If a service contains multiple replicas, then each replica container is load balanced via round-robin using the internal L4 routing mesh.

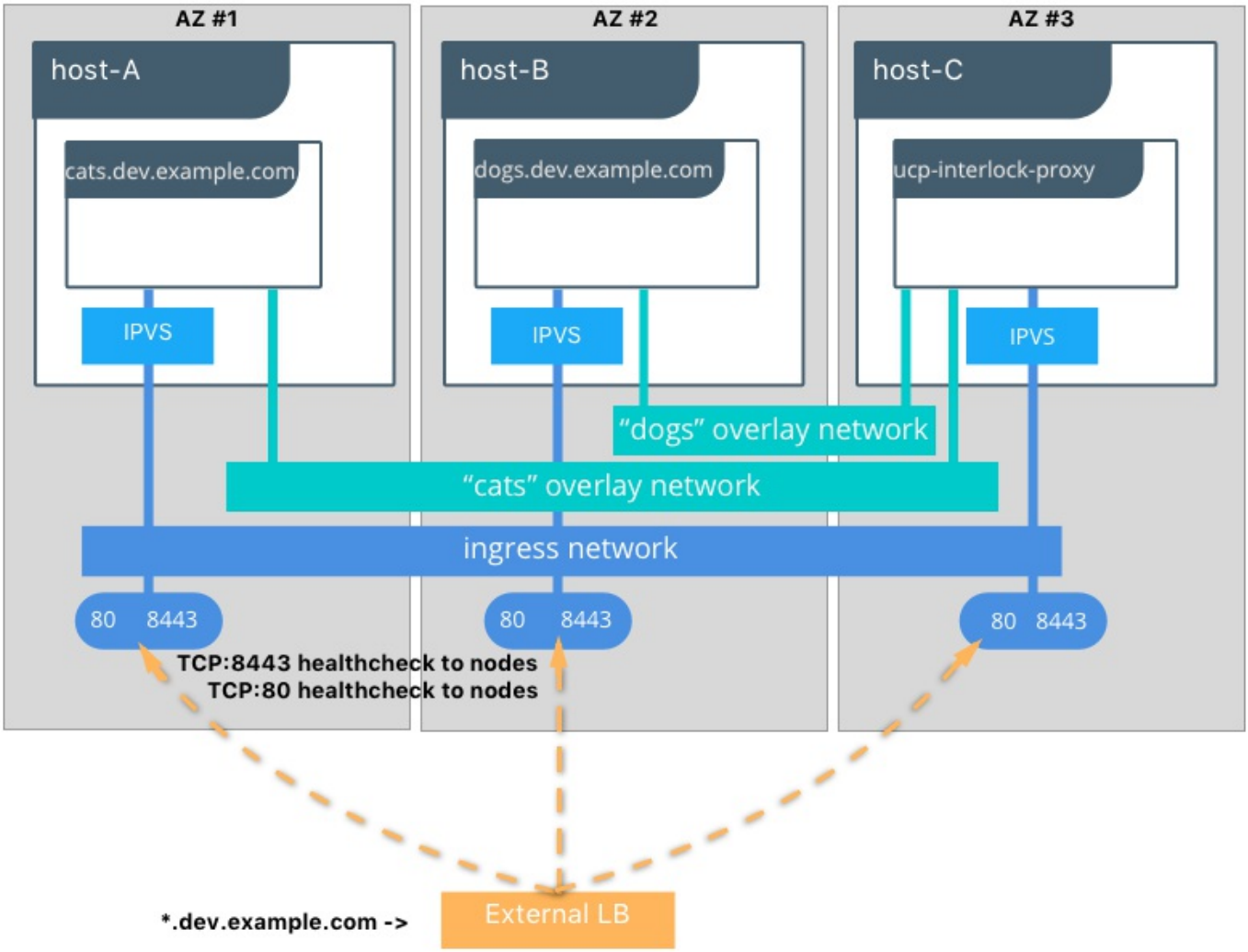
Differences Between the Interlock Proxy and Swarm Mode Routing Mesh

The main difference between the Interlock Proxy and swarm mode routing mesh is that the Interlock Proxy is intended to be used only for HTTP traffic at the application layer, while the swarm mode routing mesh works at a lower level on the transport layer.

Deciding which to use depends on the application. If the application is intended to be publicly accessible and is an HTTP service, then the Interlock Proxy could be a good fit. If mutual TLS is required for the backend application, then using the transport layer would probably be preferred.

Another advantage of using the Interlock Proxy is that less configuration is required for traffic to be routed to the service. Often times only a DNS record is needed along with setting the label on the service. If a wildcard DNS entry is used, then no configuration outside of setting the service label is necessary. In many organizations, access to load balancers and DNS is restricted. Being able to control requests to applications by just a service label can empower developers to quickly iterate over changes. With the swarm mode routing mesh, any frontend load balancer can be configured to send traffic to the service's published port.

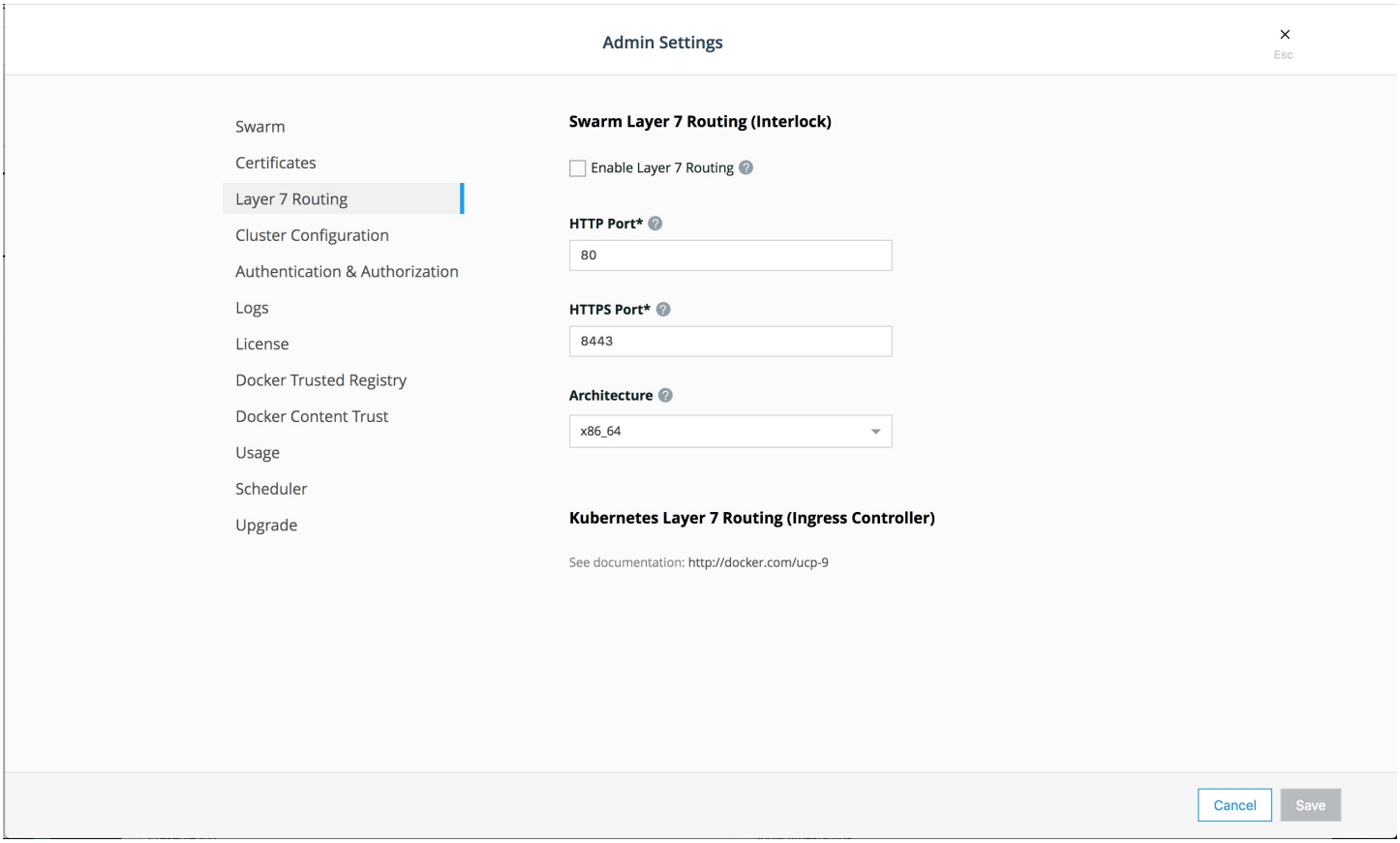
The following diagram shows an example with wildcard DNS:



Enabling the Swarm Layer 7 Routing (Interlock Proxy)

The Interlock Proxy can be enabled from the UCP web console. To enable it:

1. Log into the UCP web console.
2. Navigate to **Admin Settings** > **Layer 7 Routing**.
3. Check **Enable Layer 7 Routing** under the section titled **Swarm Layer 7 Routing (Interlock)**.
4. Configure the ports for Interlock Proxy to listen on, with the defaults being 80 and 8443. The HTTPS port defaults to 8443 so that it doesn't interfere with the default UCP management port (443).



Once enabled, UCP creates three services on the swarm cluster: `ucp-interlock`, `ucp-interlock-extension`, and `ucp-interlock-proxy`. The `ucp-interlock-proxy` service is responsible for routing traffic to the specified container based on the HTTP `Host:` header. Since the Interlock Proxy service is a *swarm mode* service, every node in the UCP cluster can route traffic to it by receiving traffic from ports `80` and `8443`. By default the Interlock Proxy service exposes ports 80 and 8443 cluster-wide, and any requests on ports 80 and 8443 to any node in the cluster are sent to the Interlock Proxy service.

Networks and Access Control

The Interlock Proxy uses one or more overlay networks to communicate with the backend application services. To allow the Interlock Proxy to communicate with and consequently forward requests to application frontend services, it needs to share a network with that service. This is accomplished by setting a label `com.docker.lb.network` to a value which is the name of the network the Interlock Proxy service should attach to for upstream connectivity. As such this action does not require administrator level access within UCP to be performed.

This configuration also allows the isolation between frontend services using the Interlock Proxy since the exposed application services do not share a common network with other similar exposed services.

Swarm Layer 7 Routing (Interlock Proxy) Requirements

There are three requirements that services must satisfy to use the Interlock Proxy:

1. The service must be connected to a network which is also defined as the value for the service label `com.docker.lb.network`.
2. The service must listen on a port. This port need not be exposed to the outside, but it should be configured as the value of the label `com.docker.lb.port` in the service.
3. The service must define a service label `com.docker.lb.hosts` to specify the host (or FQDN) served by the

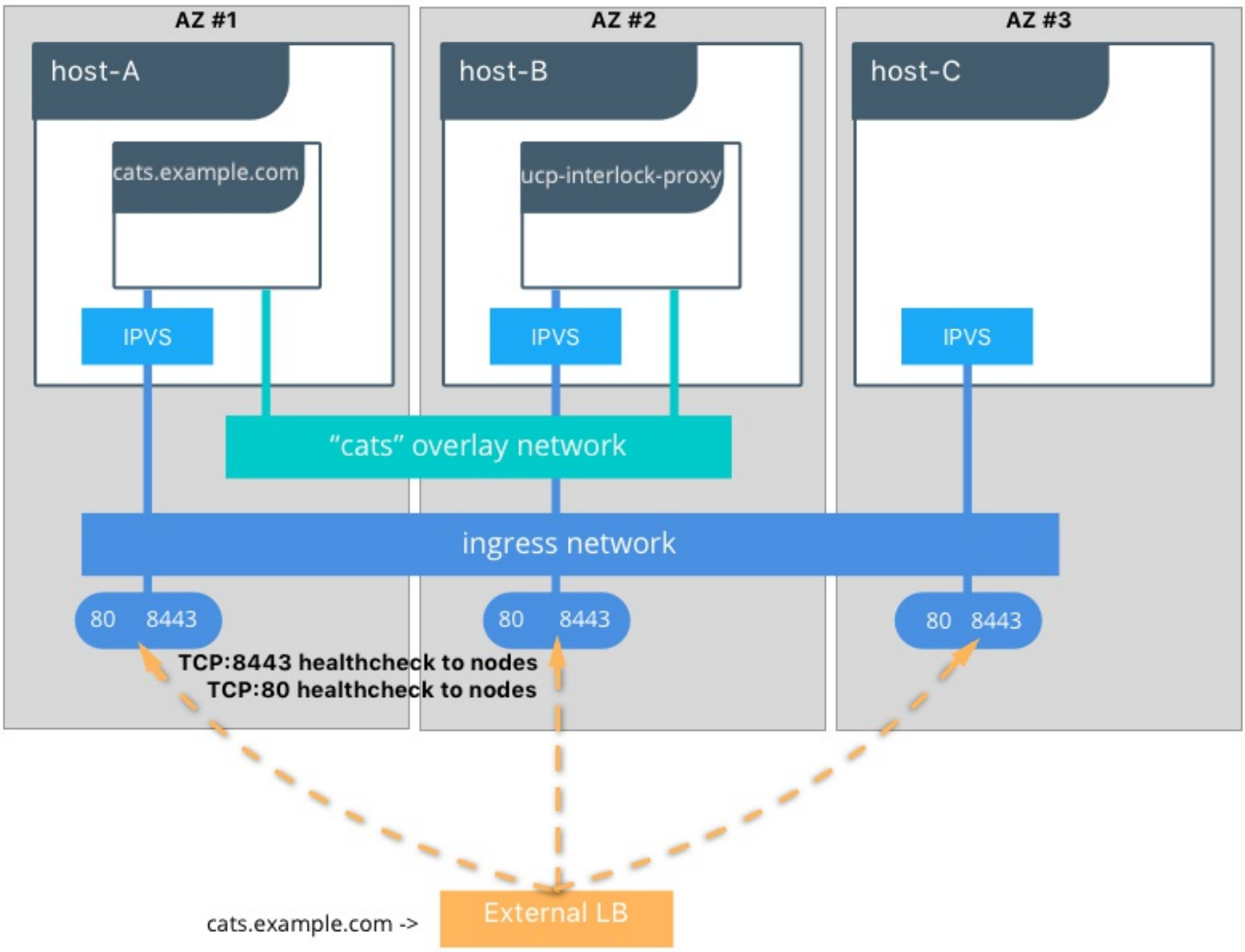
service. Multiple hosts can optionally be configured using a comma-separated list.

Configuring DNS with the Swarm Layer 7 Routing (Interlock Proxy)

This section covers how to configure DNS for services using the Interlock Proxy. To use the Interlock Proxy, a DNS record for the service needs to point to the UCP cluster. This can be accomplished through a variety of different ways because of the flexibility that the swarm mode routing mesh provides.

If a service needs to be publicly accessible for requests to `foo.example.com`, then the DNS record for that service can be configured in one of the following ways:

1. Configure DNS to point to any single node on the UCP cluster. All requests for `foo.example.com` will get routed through that node to the Interlock Proxy.
2. Configure round-robin DNS to point to multiple nodes on the UCP cluster. Any node that receives a request for `foo.example.com` will get routed through the Interlock Proxy.
3. Or, the **best** solution for **high availability**, is to configure an external HA load balancer to reside in front of the UCP cluster. There are some considerations to keep in mind when using an external HA load balancer:
    - Set the DNS record for `foo.example.com` to point to the external load balancer.
    - The external load balancer should point to multiple UCP nodes that reside in different availability zones for increased resiliency.
    - Configure the external load balancer to perform a TCP health check on the Interlock Proxy service's configured exposed port(s) so that traffic will route through healthy UCP nodes.



Which nodes should be used to route traffic, managers, or workers? There are a few ways to approach that question.

1. Routing through the manager nodes is fine for smaller deployments since managers are generally more static in nature.
    - **Advantage**: Manager nodes generally do not shift around (new hosts, new IPs, etc.) often, and it is easier to keep load balancers pointing to the same nodes.
    - **Disadvantage**: Manager nodes are responsible for the control plane traffic. If application traffic is large, you don't want to saturate traffic to these nodes and cause adverse affects to your cluster.
2. Routing through the worker nodes.
    - **Advantage**: Worker nodes do not manage the entire cluster, so there's less additional networking overhead.
    - **Disadvantage**: Worker nodes fall more into the "cattle" category when it comes to nodes. Any automation built around destroying and building nodes needs to take this into account if load balancers are pointing to worker nodes.
3. Routing through specific worker nodes (Host mode). This setup is also referred to as **Service clusters**. In this setup, specific worker nodes will route traffic for a predetermined set of applications. Consequently, another set of worker nodes can be setup to route traffic for a different set of applications, potentially using a different configuration or extension.
    - **Advantages**:

        - Application traffic is completely segregated from one another.
        - Flexibility exists when applying different extensions / configurations for each application or application cluster.
        - Higher control exists over which hosts will route traffic to specific sets of applications or services. For instance, this feature can be leveraged to constrain the Interlock Proxy on hosts that are in different regions to offer high availability.
        - Updates or changes to the Interlock Proxy can be better planned such that it does not impact all applications / services. In other words, an outage to a service cluster will be localized to a failure of applications only within that service cluster.
    - **Disadvantages**: Due to managing the proxy hosts as fixed (host mode, without ingress), there is a cost around slight reduction in reliability for greater flexibility. If using ingress, then you will need to manage the ports corresponding to each service cluster. In both the cases, there is a small overhead of managing the configurations in this mode of routing.

Regardless of which type of instance your frontend load balancer is directing traffic to, it's important to make sure the instances have an adequate network connection.

Interlock Proxy Usage

The following sections cover various use cases and the deployment syntax for the Interlock Proxy for HTTP routing, logging, monitoring, and setting up of secure application clusters (also known as service clusters).

Routing in Interlock Proxy

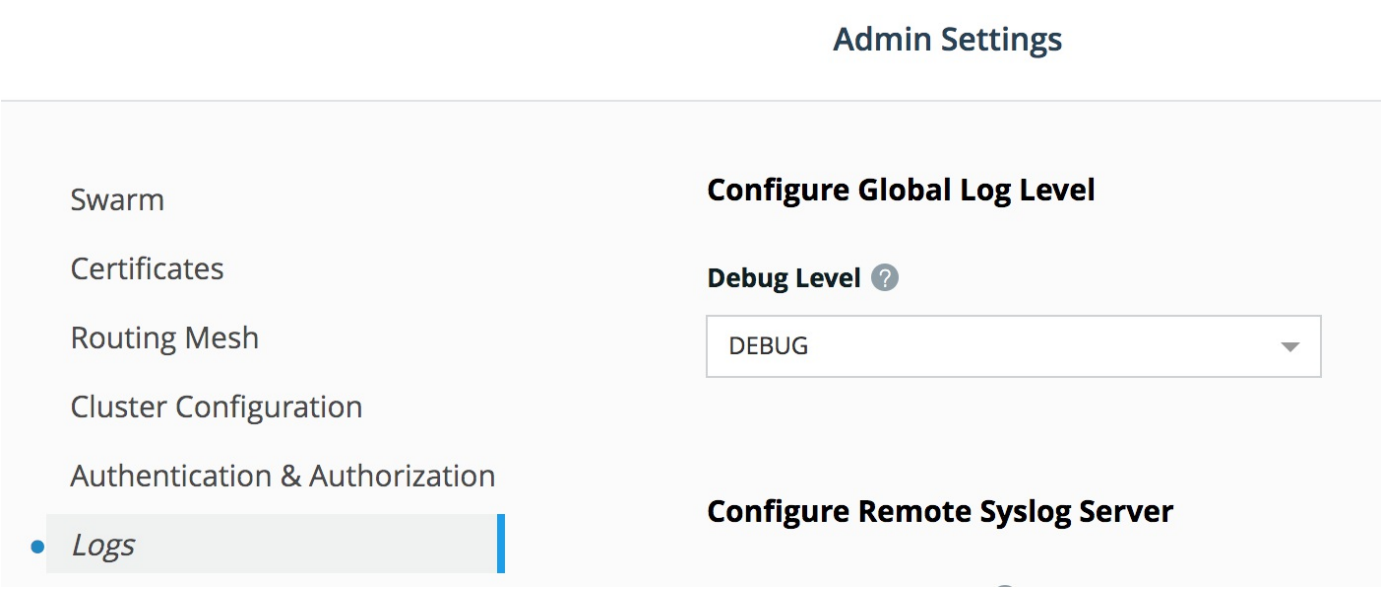
For services to be published using Interlock Proxy, they must contain, among other labels, at least two labels where the keys are `com.docker.lb.hosts` and `com.docker.lb.port`.

- The value of the label `com.docker.lb.hosts` should be the host that the service should serve. Optionally it can also be a comma-separated list of the hosts that the service should serve.
- The value of the label `com.docker.lb.port` should contain the port to use for the internal upstream communication with the service. Note that this port on the service need not be published externally.
- Optionally, the label `com.docker.lb.network` can be set to point to the name of the network that Interlock Proxy service needs to attach to for upstream connectivity. This label is required only if the service to be published using Interlock Proxy is attached to multiple overlay networks.

Logging

It is possible to log the traffic passing through Interlock Proxy by performing these steps:

1. In the UCP UI go to **Admin Settings** —> **Logs**.
2. Set the **Debug Level** to **DEBUG**.



3. Update the Interlock services to use any of the available Docker logging drivers (https://docs.docker.com/engine/admin/logging/overview/). Here's an example using the `syslog` driver:

```
docker service update --log-driver=syslog --log-opt syslog-address=udp://<ip_address>:514 ucp-interlock-proxy
```

Monitoring

To monitor the Interlock Proxy from a frontend load balancer, set the load balancer to monitor the exposed Interlock Proxy ports on the cluster using a TCP health check. If Interlock Proxy is configured to listen on the default ports of `80` and `8443`, then the frontend load balancer would need to simply perform a TCP health check on all nodes that are in its pool.

Interlock Proxy HA Considerations

This section discusses a few usage considerations with regards to Interlock Proxy running in high-availability mode.

The `ucp-interlock-proxy` can be scaled up to have more replicas, and those replicas can be constrained to only those nodes that have high performance network interfaces. The additional benefit of this architecture is that it improves security by avoiding all application traffic from routing through the managers.

The following steps needed to accomplish this design:

1. Update the high performance nodes by using node labels to identify them:

```
docker node update --label-add nodetype=loadbalancer <node>
```

2. Constrain the Interlock Proxy service tasks to only run on the high performance nodes using the node labels. This is done by updating the `ucp-interlock` service configuration to deploy the interlock proxy service with the updated constraints in the `ProxyConstraints` array as explained below:
    - Retrieve the configuration that is currently being used for the `ucp-interlock` service and save it to a file (`config.toml` for example):

```
CURRENT_CONFIG_NAME=$(docker service inspect --format \
'{{ (index .Spec.TaskTemplate.ContainerSpec.Configs 0).ConfigName }}' \
ucp-interlock)

docker config inspect --format '{{ printf "%s" .Spec.Data }}' \
$CURRENT_CONFIG_NAME > config.toml
```

- Update the `ProxyConstraints` array in the `config.toml` file as shown below:

```
[Extensions]
  [Extensions.default]
    ProxyConstraints = ["node.labels.com.docker.ucp.orchestrator.swarm==true",
"node.labels.nodetype==loadbalancer"]
```

- Create a new Docker `config` from the file that was just edited:

```
docker config create $NEW_CONFIG_NAME config.toml
```

- Update the `ucp-interlock` service to start using the new configuration:

```
docker service update \
  --config-rm $CURRENT_CONFIG_NAME \
  --config-add source=$NEW_CONFIG_NAME,target=/config.toml \
ucp-interlock
```

3. Configure the upstream load balancer to direct requests to only those high performance nodes on the Interlock Proxy ports. This ensures all traffic is directed to only these nodes.

Interlock Proxy Usage Examples

This section explains the following types of applications, using all of the available networking modes for Interlock Proxy:

- HTTP Routing
- Websockets
- Sticky Sessions
- HTTPS / SSL
- Redirection

To run through these examples showcasing service discovery and load balancing, the following are required:

1. A Docker client that has the UCP client bundle loaded and communicating with the UCP cluster.
2. DNS pointing to a load balancer sitting in front of your UCP cluster. If no load balancer can be used, then direct entries in your local hosts' file to a host in your UCP cluster. If connecting directly to a host in your UCP cluster, connect over the published Interlock Proxy ports (80 and 8443 by default).

> **Note:** The repository for the sample application can be found on GitHub (https://github.com/dockersuccess/counter-demo/blob/master/interlock-docker-compose.yml).

Interlock Proxy Routing Example

Consider an example, standard 3-tier application that showcases service discovery and load balancing in Docker EE.

To deploy the application stack, run these commands with the UCP client bundle loaded:

```
$ wget https://raw.githubusercontent.com/dockersuccess/counter-demo/master/interlock-docker-compose.yml
$ DOMAIN=<domain-to-route> docker stack deploy -c interlock-docker-compose.yml http-example
```

Then access the example application at `http://<domain-to-route>/`.

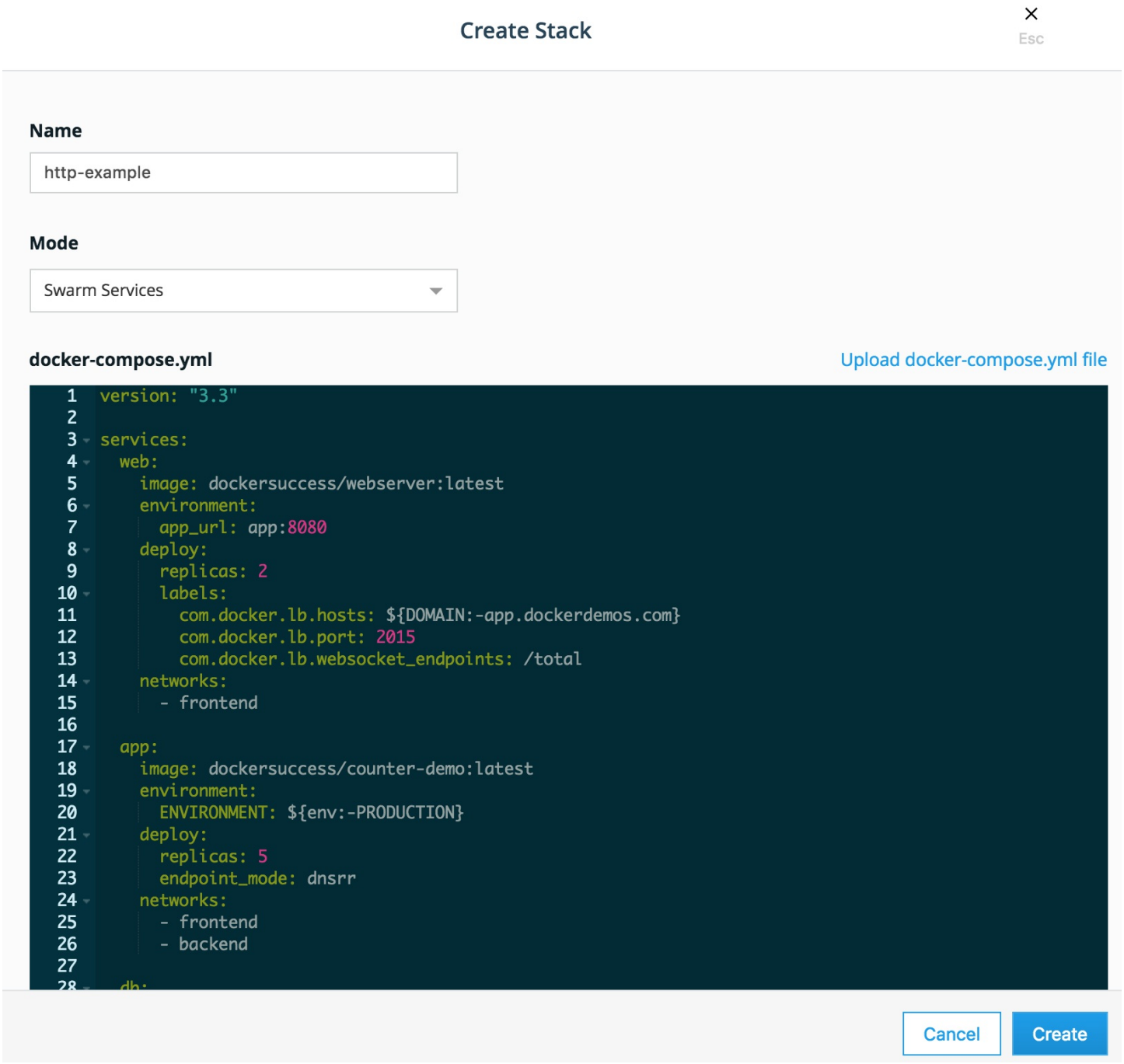
Websockets

The example also demonstrates support for websockets using the label `com.docker.lb.websocket_endpoints` with its value set to `/total` as shown in this section. The value can also be a comma-separated list of endpoints to configure to be upgraded for websockets.

This is the contents of the compose file if you just want to copy/paste into the UCP UI instead:

```
version: "3.3"

services:
  web:
    image: dockersuccess/webserver:latest
    environment:
      app_url: app:8080
    deploy:
      replicas: 2
      labels:
        com.docker.lb.hosts: ${DOMAIN:-app.dockerdemos.com}
        com.docker.lb.port: 2015
        com.docker.lb.websocket_endpoints: /total
    networks:
      - frontend

  app:
    image: dockersuccess/counter-demo:latest
    environment:
      ENVIRONMENT: ${env:-PRODUCTION}
    deploy:
      replicas: 5
      endpoint_mode: dnsrr
    networks:
      - frontend
      - backend

  db:
    image: redis:latest
    volumes:
      - data:/data
    networks:
      backend:

networks:
  frontend:
    driver: overlay
  backend:
    driver: overlay

volumes:
  data:
```

It is also possible to deploy through the UCP UI by going to **Shared Resources** -> **Stacks** -> **Create Stack**. Name the stack, change the **Mode** to **Swarm Services**, and copy/paste the above compose file into the open text field. Be sure to replace `${DOMAIN:-app.dockerdemos.com}` with the correct domain name when deploying through the UI. Click on the **Create** button to deploy the stack.



Interlock Proxy Service Deployment Breakdown

The Interlock Proxy polls the Docker API for changes every 3 seconds (default), so once an application is deployed, Interlock Proxy polls for the new service and finds it at `http://<domain-to-route>`.

When the application stack is deployed using the compose file used as an the example in this section, the following happens:

1. Three services are created — a service called `web` that is the frontend running a Caddy webserver, a service called `app` which contains the application logic, and another service `db` running a Redis database to store data.

2. Multiple overlay networks specific to the application stack are created called `<stack-name>_frontend` and `<stack-name>_backend`. The `web` and `app` services share the `<stack-name>_frontend` network, and the `app` and `db` services share the `<stack-name>_backend` network. In other words the `web` service cannot connect to the `db` service directly; it needs to connect to the `app` service, which is the only service that can connect to the `db` service.

3. The `app` service creates a DNS A Record of `app` on the `<stack-name>_frontend` network. This DNS record directs to the IP address(es) of the `app` containers.

    - The `web` service uses an environment variable `app_url`, the value of which is set to `app:8080`. This value points to the `app` service and does not need to change regardless of the stack name.
    - Similarly, the Redis task creates a DNS A Record of `db` on the `<stack-name>_backend` network.
    - The `app` service does not need to change for every stack when accessing Redis DB. It will always connect to `db`, and this can be hardcoded inside the `app` service. This is also independent of the stack name.

4. The frontend service `web` contains the labels `com.docker.lb.hosts` and `com.docker.lb.port`. The value for the label `com.docker.lb.hosts` is set to the domain(s) where the application needs to be made available. This can be conveniently set using the `$DOMAIN` environment variable. The value for the label `com.docker.lb.ports` is set to the port where the `web` service is running, which is `2015` in this example. Because the `web` service is connected to a single overlay network `<stack-name>_frontend`, Interlock Proxy was able to attach itself to the `<stack-name>_frontend` network.

5. These labels above act as triggers for integration with Interlock Proxy. Once this service is available, Interlock Proxy will detect it and publish it. Any requests to the configured domain on the Interlock Proxy port (80 by default) will be forwarded to one of the `web` service replicas on the configured port.

6. The declared healthy state for the `web` service is 2 replicas, so 2 replica tasks are created. The two `web` service replicas are configured as upstreams. Interlock Proxy is responsible for balancing traffic across all published service replicas.

7. Interlock Proxy creates an entry so that all 5 frontend replicas are load balanced backed on the `$DOMAIN` environment variable that was passed for the stack deploy.

8. By doing a refresh of `http://$DOMAIN` in a web browser, the hit counter should increment with every request. It is load balancing across all of the frontend `web` service replicas.

9. Interlock Proxy polls every 3 seconds for Docker events, and it picks up the `com.docker.lb.*` labels on any newly created or updated services.

Interlock Proxy Sticky Session Example

Cookie-Based Session Persistence

Interlock Proxy has the ability to route to a specific backend service based on a named cookie. For example, if your application uses a cookie named `JSESSIONID` as the session cookie, you can persist connections to a specific service replica task by setting the value of the label `com.docker.lb.sticky_session_cookie` to `JSESSIONID`.

Why would cookie-based persistence need to be used? It can reduce load on the load balancer. The load balancer picks a certain instance in the backend pool and maintains the connection instead of having to re-route on new requests. Another use case could be for rolling deployments. When you bring in a new application server into the load balancer pool you can avoid the "thundering herd" of new instances. Instead, it eases connections to the new instances into load balancing as sessions expire.

In general, sticky sessions are better suited for improving cache performance and lessening the load on certain aspects of the system. If you need to hit the same backend every time because your application is not using distributed storage, then you can run into more problems down the road when swarm mode reschedules your tasks. It's important to keep this in mind while using application cookie-based persistence.

To deploy the example service for sticky sessions, run these commands with the UCP client bundle loaded. Note the label `com.docker.lb.sticky_session_cookie` used to indicate the cookie to use to enable sticky sessions.

```
# Create an overlay network so that service traffic is isolated and secure
docker network create -d overlay demo

# Next create the service with the cookie to use for sticky sessions. Replace <domain-to-route> with a
valid domain.
docker service create \
--name demo \
--network demo \
--detach=false \
--replicas=5 \
--label com.docker.lb.hosts=<domain-to-route> \
--label com.docker.lb.sticky_session_cookie=session \
--label com.docker.lb.port=8080 \
--env METADATA="demo-sticky" \
dockersuccess/docker-demo
```

Access the example application at `http://<domain-to-route>` in a browser tab or use `curl` as shown below:

> **Note:** To test using a browser you need a DNS entry to <domain-to-route> pointing to a load balancer sitting in front of your UCP cluster. If no load balancer can be used, then direct entries in your local hosts' file to a host in your UCP cluster. If connecting directly to a host in your UCP cluster, you need to connect over the published Interlock Proxy ports (80 and 8443 by default). The following examples use `demo.local` in lieu of `<domain-to-route>`.

```
{{content}}gt; curl -vs -c cookie.txt -b cookie.txt http://demo.local/ping
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 80 (#0)
GET /ping HTTP/1.1
Host: demo.local
User-Agent: curl/7.54.0
Accept: */*
Cookie: session=1510171444496686286

< HTTP/1.1 200 OK
< Server: nginx/1.13.6
< Date: Wed, 08 Nov 2017 20:04:36 GMT
< Content-Type: text/plain; charset=utf-8
< Content-Length: 117
< Connection: keep-alive
* Replaced cookie session="1510171444496686286" for domain demo.local, path /, expire 0
< Set-Cookie: session=1510171444496686286
< x-request-id: 3014728b429320f786728401a83246b8
< x-proxy-id: eae36bf0a3dc
< x-server-info: interlock/2.0.0 (147ff2b1) linux/amd64
< x-upstream-addr: 10.0.2.5:8080
< x-upstream-response-time: 1510171476.948
<
{"instance":"9c67a943ffce","version":"0.1","metadata":"demo-
sticky","request_id":"3014728b429320f786728401a83246b8"}
```

In the output of `curl` above, the `Set-Cookie` attribute from the application is sent with subsequent requests, which are pinned to the same instance. The same `x-upstream-addr` is used for new requests.

IP Hashing-Based Sticky Sessions

Interlock Proxy also supports `IP Hashing`. In this mode a unique hash key is generated using the IP addresses of the source and destination. This hash is used by the load balancer (Interlock Proxy in our case) to allocate clients to a particular backend server.

Below is an example that uses IP hashing to enable sticky sessions. The label to use in this case is `com.docker.lb.ip_hash` with its value set to true.

Run the following commands with the UCP bundle loaded.

```
{{content}}gt; docker network create -d overlay demo
1se1glh749q1i4pw0kf26mfx5

{{content}}gt; docker service create \
--name demo \
--network demo \
--detach=false \
--replicas=5 \
--label com.docker.lb.hosts=demo.local \
--label com.docker.lb.port=8080 \
--label com.docker.lb.ip_hash=true \
--env METADATA="demo-sticky" \
dockersuccess/docker-demo
```

Interlock Proxy HTTPS Example

Interlock Proxy has support for routing using HTTPS / SSL. Both "SSL Termination" and "SSL Passthrough" can be setup to provide different configurations of load balancing encrypted web traffic.

In SSL Terminations, SSL encrypted requests are decrypted by Interlock Proxy (the Load Balancer layer), and the unencrypted request is sent to the backend servers.

In SSL Passthrough, the SSL requests are sent as-is directly to the backend servers. The requests remain encrypted, and the backend servers become responsible for the decryption. This also implies that the backend servers have the necessary certificates and libraries to perform the decryption.

Secrets and certificates are almost always involved when dealing with encrypted communications. Before deploying the example application, generate the necessary certificates. You can also use one of several Certificate Authorities like Let's Encrypt to generate the certificates.

Here are some helpful commands to generate self-signed certificates to use with the application using `openssl`.

```
{{content}}gt; openssl req \
-new \
-newkey rsa:4096 \
-days 3650 \
-nodes \
-x509 \
-subj "/C=US/ST=SomeState/L=SomeCity/O=Interlock/CN=demo.local" \
-keyout demo.local.key \
-out demo.local.cert
```

Now that the certificates are generated, you can use the resulting files as input to create two Docker secrets using the following commands:

```
{{content}}gt; docker secret create demo.local.cert demo.local.cert
ywn8ykni6cmnq4iz64um1pj7s
{{content}}gt; docker secret create demo.local.key demo.local.key
e2xo036ukhfapip05c0sizf5w
```

The secrets are now encrypted in the cluster-wide key value store. The secrets are encrypted at rest and using TLS while in motion to nodes that need the secret. Secrets can only be viewed by the application that needs to use them.

> **Tip:** For more details on using Docker secrets please refer to the Reference Architecture covering DDC Security and Best Practices (https://success.docker.com/article/security-best-practices#secrets).

Now create an overlay network so that service traffic is isolated and secure:

```
{{content}}gt; docker network create -d overlay demo
1se1glh749q1i4pw0kf26mfx5

{{content}}gt; docker service create \
--name demo \
--network demo \
--label com.docker.lb.hosts=demo.local \
--label com.docker.lb.port=8080 \
--label com.docker.lb.ssl_cert=demo.local.cert \
--label com.docker.lb.ssl_key=demo.local.key \
dockersuccess/docker-demo
6r0wiglf5f3bdpcy6zesh1pzx
```

Interlock Proxy detects when the service is available and publish it. Once the tasks are running and the proxy service has been updated the application should be available via `https://demo.local`.

> **Note:** To test using a browser you need a DNS entry to <domain-to-route> pointing to a load balancer sitting in front of your UCP cluster. If no load balancer can be used, then direct entries in your local hosts' file to a host in your UCP cluster. If connecting directly to a host in your UCP cluster, you would need to connect over the published Interlock Proxy ports (80 and 8443 by default). The following examples use `demo.local` in lieu of <domain-to-route>.

```
{{content}}gt; curl -vsk https://demo.local/ping
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to demo.local (127.0.0.1) port 443 (#0)
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/certs/ca-certificates.crt
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use http/1.1
* Server certificate:
*  subject: C=US; ST=SomeState; L=SomeCity; O=Interlock; CN=demo.local
*  start date: Nov  8 16:23:03 2017 GMT
*  expire date: Nov  6 16:23:03 2027 GMT
*  issuer: C=US; ST=SomeState; L=SomeCity; O=Interlock; CN=demo.local
*  SSL certificate verify result: self signed certificate (18), continuing anyway.
> GET /ping HTTP/1.1
> Host: demo.local
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx/1.13.6
< Date: Wed, 08 Nov 2017 16:26:55 GMT
< Content-Type: text/plain; charset=utf-8
< Content-Length: 92
< Connection: keep-alive
< Set-Cookie: session=1510158415298009207; Path=/; Expires=Thu, 09 Nov 2017 16:26:55 GMT; Max-Age=86400
< x-request-id: 4b15ab2aaf2e0bbdea31f5e4c6b79ebd
< x-proxy-id: a783b7e646af
< x-server-info: interlock/2.0.0 (147ff2b1) linux/amd64
< x-upstream-addr: 10.0.2.3:8080

{"instance":"c2f1afe673d4","version":"0.1",request_id":"7bcec438af14f8875ffc3deab9215bc5"}
```

Since the certificate and key are stored securely within the Docker swarm, you can safely scale this service as well as the proxy service, and the Docker swarm will handle granting access to the credentials only as needed.

Support for Kubernetes in Docker EE

Docker EE 2.0 has support for the Kubernetes orchestrator in addition to swarm mode, which is the foundational aspect of Interlock Proxy. Kubernetes workloads and applications can co-exist within the same cluster along with swarm / Interlock proxy services. While swarm / Interlock proxy services are supported over

Windows as well as Linux applications, Kubernetes support within Docker EE is limited to Linux applications only.

Service discovery and DNS

Service discovery within Kubernetes is available via the built-in DNS service. This built-in DNS is used within pods to find other services running on the same cluster. All Kubernetes pods running within Docker EE are part of the same cluster. Containers within a pod can, however, contact each other using `localhost:port` without needing to use the DNS service. For service discovery to work, the destination pod should define a `service` resource type; only then will the pod be accessible using Kubernetes' DNS. Further, every node, including manager nodes in a Docker EE cluster, runs a component as `kube-proxy`. `kube-proxy` is responsible for watching the `kube-api` server for the addition and removal of `Service` objects, opening random ports on the nodes as necessary, proxying ports to the backend pods, configuring iptables, and fullfilling load balancing requests across multiple replicas (pods) of a deployment.

Additional reference material on DNS is available at https://kubernetes.io/docs/concepts/services-networking/service/#dns (https://kubernetes.io/docs/concepts/services-networking/service/#dns).

> **Note:** The concept of Kubernetes Services is not the same as Docker Swarm Services. Kubernetes Services are a "layer 4" construct that can be extended to add "layer 7" features using an Ingress Controller.

Exposing Services

For parts of the application that need to interact with or be called from other pods or from external services, or simply make them available on an external IP address, it is necessary to expose them as services. Another consideration for using services is that services provide an abstraction layer to reach pods. This is especially important due to the dynamic nature of pods. Pods are ephemeral and can come up and go as necessary, due to scaling operations, rolling updates, or backups etc. Services provide a layer of access that is immune to the dynamic nature of pods from the perspective of consumers.

Kubernetes offers several types of services. The specific type can be passed as the value of the `type` attribute in the `Service` resource definition. The default service type is `ClusterIP`. The different types of services in the context of Docker EE are described in the following sections.

Cluster IP and Headless Services

This is the default `type`. Using this type creates a virtual IP address that is only accessible form within the cluster.

In some cases, it may be desirable to customize the service registration, service discovery, load balancing, and proxying of services. This can be done by building a plugin that interacts with the Kubernetes API. For such instances, a cluster IP is not necessary, and its creation can be prevented by specifying `None` for the cluster IP. Such a service where the cluster IP is not created is called as a "headless" service.

Nodeport

A service type of NodePort externally exposes a port on every node in the cluster. By default, this port is a random high port (30000 to 32767). Each node proxies this port into the corresponding service.

A Cluster IP service is still created alongside the NodePort service unless a "headless" service was explicitly configured by specifying `None` for the cluster IP.

> **Note:** The NodePort in Kubernetes is conceptually similar to the "Routing Mesh" in Docker swarm mode.

Load Balancer

This service type is usually used in cloud environments. It provisions and uses the native cloud infrastructure (eg: ELB on AWS) to route external traffic to every node in the cluster. Both the NodePort and Cluster IP services are created unless it is a "headless" service in which case a Cluster IP is not created.

Deploying Kubernetes Applications in Docker EE

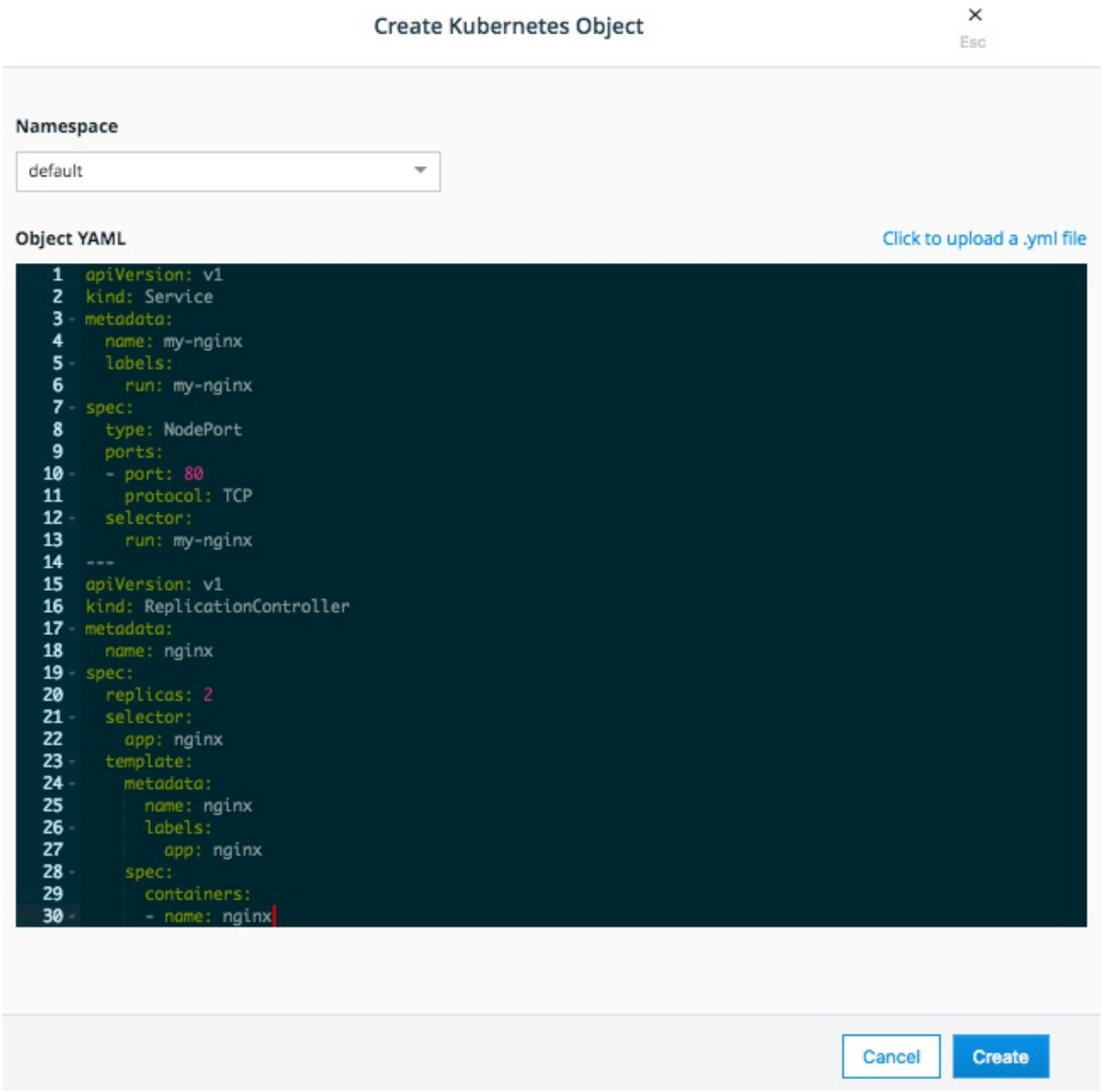
Docker EE 2.0 conforms to the Certified Kubernetes program (https://www.cncf.io/certification/software-conformance/). What this means is that Docker EE offers a fully-conformant, unmodified Kuberenetes experience. All Kubernetes APIs are guaranteed to function per their specifications offering the same benefits of resiliency, consistency, and portability as one would expect in the Docker EE platform. In other words, you can run vanilla Kubernetes resource definitions using `kubectl` or through the UCP UI.

In the following example, a simple Kubernetes application is deployed using the UCP UI. Below is the content of a simple YAML file that defines a Deployment resource with two pods (replicas) and a Service exposed as a NodePort.

To deploy this, log into UCP and click on **Kubernetes** in the left navigation menu to open up its sub-menu. Click on **+ Create**. You should now be in the **Create Kubernetes Object** pane.

In the **Create Kubernetes Object** pane, select "default" from the **Namespace** dropdown and in the **Object YAML** textbox, paste the following text:

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
   matchLabels:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  labels:
    name: nginxservice
  name: nginxservice
spec:
  ports:
    - port: 80
  selector:
    app: nginx
  type: NodePort
```



> **Note:** The same can be done using `kubectl apply` after loading the UCP client bundle.

To access the application URL just deployed, follow these steps:

1. Click on the Escape key to exit out of the **Create Kubernetes Object** pane.
2. Now click on the **Load Balancers** sub-menu item in the left navigation pane under **Kubernetes**. A new load balancer entry named `nginxservice` should be created now.
3. Click on it, and in the newly appeared right pane, scroll down to the **Spec** section. Under **Ports** click on

the **URL:** link.

The earlier example demonstrated deploying a Kubernetes application (an nginx webserver in this case) in Docker EE using native Kubernetes resource definitions. It is also possible to deploy an application using a regular `docker-compose.yml` (https://docs.docker.com/compose/compose-file/) file. In essence, starting from compose version 3.3, the same `docker-compose.yml` file used for swarm deployments can be deployed as a Kubernetes stack by specifying Kubernetes workloads at time of deploying the stack. The result is a true Kubernetes app that can be managed using `kubectl` commands if necessary.

> **Note:** Docker EE 2.0 has implemented the "stack" concept for Kubernetes workloads as a Custom Resource Definition (https://kubernetes.io/docs/concepts/api-extension/custom-resources/) by extending the Kubernetes API. This is what makes it possible to run commands such as `kubectl get stacks`.

Below is an example of deploying a compose stack as a Kubernetes workload. This example deploys this stack in a separate namespace. Namespaces (https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/) provide logical separation of applications within a cluster.

Log into UCP and create a new namespace. Navigate to **Kubernetes** —> **Namespaces** —> **Create**. In the **Create Kubernetes Object** pane, enter the following and click on **Create**:

```
apiVersion: v1
kind: Namespace
metadata:
  name: demo
```

The example application is the same as demonstrated with Interlock Proxy under swarm mode. It has three services: `web`, `app`, and `db`. The `db` service is a stateful service and requires a PersistentVolume (https://kubernetes.io/docs/concepts/storage/persistent-volumes/) to store data.

Create a `PersistentVolume` using the following definition. Navigate to **Kubernetes** —> **+ Create** and paste this in the **Object YAML** textbox.

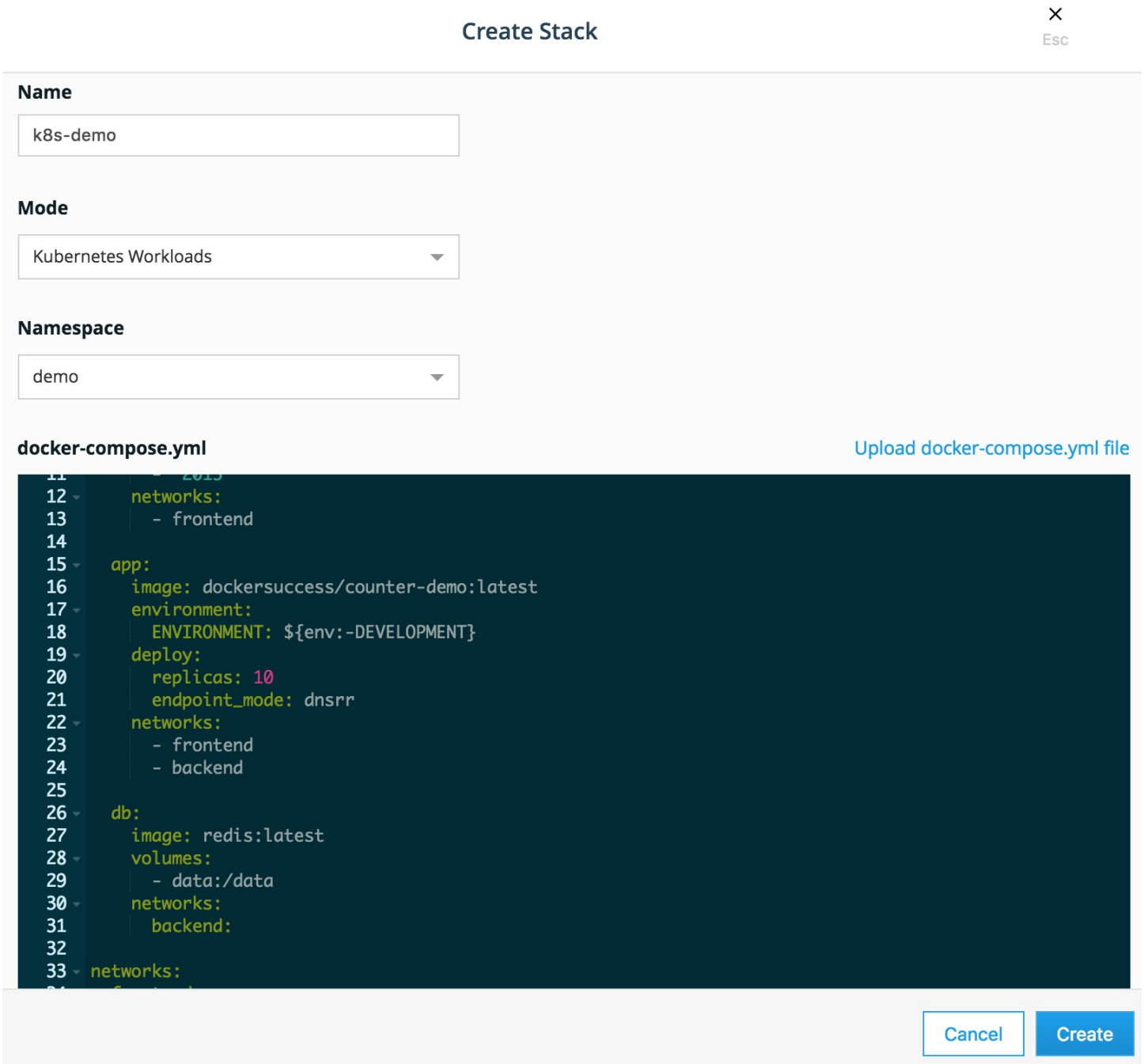
```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: var-tmp-volume
  labels:
    type: local
spec:
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/var/tmp"
```

> **Note:** The above resource definition assumes the path `/var/tmp` is available on every node in the cluster. Normally, a shared, durable storage such as NFS or cloud storage (https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes) is used to store data pertaining to PersistentVolumes. Also note that namespaces do not apply to PersistentVolumes.

You can now deploy the application. Click on **Shared Resources** in the left navigation pane. In the sub-menu click on **Stacks** —> **Create Stack**.

In the **Create Stack** pane, enter a name for the stack, such as `k8s-demo`. Select "Kubernetes Workloads** under **Mode**, and select the newly created "demo" namespace under **Namespace**. Paste the following compose file contents in the **docker-compose.yml** textbox and click on **Create**.

```
version: "3.3"

services:
  web:
    image: dockersuccess/webserver:latest
    environment:
      app_url: app:8080
    deploy:
      replicas: 5
    ports:
      - "2015"
    networks:
      - frontend

  app:
    image: dockersuccess/counter-demo:latest
    environment:
      ENVIRONMENT: ${env:-DEVELOPMENT}
    deploy:
      replicas: 10
      endpoint_mode: dnsrr
    networks:
      - frontend
      - backend

  db:
    image: redis:latest
    volumes:
      - data:/data
    networks:
      backend:

networks:
  frontend:
    driver: overlay
  backend:
    driver: overlay

volumes:
  data:
```



After a few minutes, the stack should be deployed, and all Pods should be available and healthy. To access the application, change the **Namespace** to `demo` by navigating to **Kubernetes** —> **Namespaces** —> **demo** —> **Action** —> **Set Context**. Now click on **Kubernetes** —> **Load Balancers** —> **web-random-ports**. In the right side pop-up, under **Spec** —> **Ports** —> **URL**, you can access the URL and port to access the appplication.

Deploying an Ingress Controller

When applications are deployed using NodePort as the Service type, as in the example above, Kubernetes exposes the application endpoints on a random port on every node in the cluster. Each port (range between 30000 to 32767) can be assigned to one service only. This port does not change as long as the service exists. If the service is recreated or for new applications onboarded, this port would need to be queried from Kubernetes API so that requests can be sent to the appropriate application on its exposed port (NodePort). It is possible to specify the NodePort explicitly, but then you will be responsible for managing port conflicts as well as maintaining the mapping between applications and their NodePorts. Directly exposing the NodePort in

Production is not recommended due to security and usability concerns. A standard way of exposing the service to the Internet is by using a `LoadBalancer` service. This is easier to do in a cloud-based environment by leveraging infrastructure such as AWS Elastic Load Balancer (ELB) or GKE's Network Load Balancer (NLB). But provisioning a `LoadBalancer` for every application may get prohibitively expensive.

Instead, the recommended approach is to deploy and use one or more *Ingress Controllers*. An Ingress Controller is an implementation of a reverse proxy that watches for services being created and destroyed to auto-configure itself. The Ingress Controller works in conjunction with another resource called "Ingress", and together they expose multiple services over L7 (Layer 7). L7 allows for the implementation of many advanced capabilities like redirections, SSL/HTTPS, Authentication, etc.

The Ingress Resource

Ingress resources are collections of rules and configurations related to exactly how inbound connections reach application services running within the Kubernetes cluster — configurations like host-based URLs, load balancing, SSL terminations or passthrough, or even configuring edge routers. An ingress resource alone is not sufficient for the routing of application traffic; it would invariably require one or more ingress controllers to satisfy the rules and configurations of the ingress resource.

This example uses the standard Kubernetes nginx based Ingress Controller (https://github.com/kubernetes/ingress-nginx) in Docker EE to expose an application over L7.

It configures an ingress controller in a separate namespace `ingress-nginx`. The ingress controller needs to be able to watch for events for service creations and deletions in any namespace, so you need to grant the `default` service account in the `ingress-nginx` read access to all other namespaces.

Create the `ingress-nginx` Namespace

Log into UCP and navigate to **Kubernetes** —> **Namespace** —> **Create**. In the **Create Kubernetes Object** pane, under **Object YAML**, paste this YAML text:

```
apiVersion: v1
kind: Namespace
metadata:
  name: ingress-nginx
```

Grant the `default` Service Account in `ingress-nginx` Namespace Access to All Namespaces

The `default` service account associated with the `ingress-nginx` namespace needs access to Kubernetes resources. These steps create a grant with `Restricted Control` permissions.

- Log into UCP and click on **User Management**.
- Navigate to the **Grants** page, and click **Create Grant**.
- In the left pane, click **Resource Sets**, and in the Type section, click **Namespaces**.
- Enable the **Apply grant to all existing and new namespaces** option.
- In the left pane, click **Roles**. In the Role dropdown, select **Restricted Control**.
- In the left pane, click **Subjects**, and select **Service Account**.
- In the **Namespace** dropdown, select `ingress-nginx`, and in the **Service Account** dropdown., select `default`.
- Click **Create**.

Create the nginx Ingress Controller

In UCP, navigate to **Kubernetes** —> **+ Create**. In the **Create Kubernetes Object** page, select `ingress-nginx` from the **Namespace** dropdown. Under **Object YAML**, paste the following text:

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: default-http-backend
  labels:
    app: default-http-backend
  namespace: ingress-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: default-http-backend
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: default-http-backend
        # Any image is permissable as long as:
        # 1. It serves 200 on a /healthz endpoint
        # 2. It serves a 404 page for all other endpoints
        image: gcr.io/google_containers/defaultbackend:1.4
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 30
          timeoutSeconds: 5
        ports:
        - containerPort: 8080
        resources:
          limits:
            cpu: 10m
            memory: 20Mi
          requests:
            cpu: 10m
            memory: 20Mi
---
apiVersion: v1
kind: Service
metadata:
  name: default-http-backend
  namespace: ingress-nginx
  labels:
    app: default-http-backend
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: default-http-backend
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
  namespace: ingress-nginx
```

```
  labels:
    app: ingress-nginx
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: tcp-services
  namespace: ingress-nginx
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: udp-services
  namespace: ingress-nginx
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ingress-nginx
  template:
    metadata:
      labels:
        app: ingress-nginx
      annotations:
        prometheus.io/port: '10254'
        prometheus.io/scrape: 'true'
    spec:
      initContainers:
      - command:
        - sh
        - -c
        - sysctl -w net.core.somaxconn=32768; sysctl -w net.ipv4.ip_local_port_range="1024 65535"
        image: alpine:3.6
        imagePullPolicy: IfNotPresent
        name: sysctl
        securityContext:
          privileged: true
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.10.2
          args:
            - /nginx-ingress-controller
            - --default-backend-service=$(POD_NAMESPACE)/default-http-backend
            - --configmap=$(POD_NAMESPACE)/nginx-configuration
            - --tcp-services-configmap=$(POD_NAMESPACE)/tcp-services
            - --udp-services-configmap=$(POD_NAMESPACE)/udp-services
            - --annotations-prefix=nginx.ingress.kubernetes.io
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
```

```
                fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        ports:
        - name: http
          containerPort: 80
        - name: https
          containerPort: 443
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /healthz
            port: 10254
            scheme: HTTP
          initialDelaySeconds: 10
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 1
        readinessProbe:
          failureThreshold: 3
          httpGet:
            path: /healthz
            port: 10254
            scheme: HTTP
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 1
---
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    nodePort: 35000
    targetPort: 80
    protocol: TCP
  - name: https
    port: 443
    nodePort: 35443
    targetPort: 443
    protocol: TCP
  selector:
    app: ingress-nginx
```

The following are defined in the above configuration:

- A deployment resource for the nginx ingress controller, named `nginx-ingress-controller`
- A service that exposes the deployment, named `ingress-nginx` on ports `35000` and `35443` on all nodes in the cluster, for HTTP and HTTPS respectively

- An application named `default-http-backend` that provides a simple web service that serves 200 on the `/healthz` endpoint (and 404 everywhere else)

After a few minutes, all the resources from the above configuration would be provisioned and functioning. Navigate to the **Load Balancers** page and click the `ingress-nginx` service. In the details pane, click the first URL in the **Ports** section. A new page opens, displaying **default backend - 404**. Navigating to the `/healthz` URL should show a blank page. This indicates that the `ingress-controller` is working appropriately.

Deploy Applications using Ingress

This section shows how to deploy the same application as demonstrated before to use ingress' L7 routing.

For the example application to work, you need to have a DNS registered domain/hostname or insert an entry for the domain/hostname in your local hosts file, which points to one of the hosts' IP addresses in the cluster. Alternatively, you can simulate the hostname by passing it as a flag (`-H` or `--header`) using tools like `curl`. The domain/hostname in DNS should point to all the hosts in the cluster.

> **Tip:** A simplified option is to setup a wildcard DNS entry that points to all the hosts in the cluster. In this example, the setup is such that `*.dockerdemos.com` is a wildcard DNS entry pointing to all the external IP addresses of all the nodes in the Docker EE cluster.

You will need to edit the stack so that the frontend (web) service is assigned a Cluster IP instead of being headless. To do this, ensure the `demo` namespace is selected under **Kubernetes** —> **Namespaces**. Next navigate to **Shared Resources** —> **Stacks**, and click on the stack named `demo/k8s-demo`. In the right pop-up pane, click on **Configure**. In the resulting editor pane with the contents of the `compose.yaml` file, edit the ports configuration for the `web` service to be `- "2015:2015"` instead of `- 2015`. Click **Save**.

Object YAML

```
1  version: "3.3"
2
3  services:
4    web:
5      image: dockersuccess/webserver:latest
6      environment:
7        app_url: app:8080
8      deploy:
9        replicas: 5
10     ports:
11       - "2015:2015"
12     networks:
13       - frontend
14
15   app:
16     image: dockersuccess/counter-demo:latest
```

To create the ingress resource, click on **Kubernetes** —> **+ Create** in the left navigation pane. In the **Create Kubernetes Object** pane, select `demo` as the Namespace, and paste the following YAML in the **Object YAML** textbox:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: noop-ingress
spec:
  rules:
  - host: app.dockerdemos.com
    http:
      paths:
      - backend:
          serviceName: web-published
          servicePort: 2015
```

> **Note:** You can have as many ingress resources as necessary, but the ingress should be in the same namespace as the application being integrated with the ingress controller.

In a few moments, the nginx ingress controller will be reconfigured to proxy requests for `app.dockerdemos.com` to the application on port 2015. In other words, the url `http://app.dockerdemos.com:35000` will allow access into the application. The port `35000` is fixed because this is the port the nginx ingress controller listens on.

You can deploy another application in a separate namespace that also auto-configure itself with the nginx ingress controller. The url will be different, but the port (`35000`) will be same because the requests are proxied through the ingress controller.

The following steps are very similar to the previous application deployment procedure and result in the deployment of an application called `words` in the `blue` namespace. They also configure the application to use L7 capabilities by integrating it with the nginx ingress controller.

Create a new namespace called `blue` using the following YAML in **Kubernetes** —> **Namespaces** —> **Create**.

```
apiVersion: v1
kind: Namespace
metadata:
  name: blue
```

In **Stacks** —> **Create Stack**, enter **Name** as `words`, **Mode** as `Kubernetes Workloads`, and select `blue` as the **Namespace**. Paste the following YAML, and click on **Save**:

```
version: '3.3'

services:
  web:
    build: web
    image: dockerdemos/lab-web
    volumes:
     - "./web/static:/static"
    ports:
     - "80:80"

  words:
    build: words
    image: dockerdemos/lab-words
    deploy:
      replicas: 5
      endpoint_mode: dnsrr
      resources:
        limits:
          memory: 16M
        reservations:
          memory: 16M

  db:
    build: db
    image: dockerdemos/lab-db
```

Finally, create the Ingress resource by navigating to **Kubernetes** —> **+ Create**. Select the `blue` namespace, paste the following YAML inside the editor, and click on **Create**:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: words-ingress
spec:
  rules:
  - host: words.dockerdemos.com
    http:
      paths:
      - backend:
          serviceName: web-published
          servicePort: 80
```

Just as before, the newly deployed `words` application will be available at the URL: `http://words.dockerdemos.com:35000`. Both applications are routed through the nginx ingress controller and can leverage all L7 capabilities that are available within nginx. The ports `35000` and `35443` will be fixed regardless of the application. The nginx ingress controllers can be scaled up to provide for high availability and better throughput. An external or physical load balancer can now be configured to perform SSL terminations/passthrough or forward requests on these ports on all nodes in the cluster.

Summary

The ability to scale and discover services in Docker is now easier than ever. With the service discovery and load balancing features built into Docker, engineers can spend less time creating these types of supporting capabilities on their own and more time focusing on their applications. Instead of creating API calls to set DNS for service discovery, Docker automatically handles it for you. If an application needs to be scaled, Docker takes care of adding it to the load balancer pool. By leveraging these features, organizations can deliver highly available and resilient applications in a shorter amount of time.