# 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

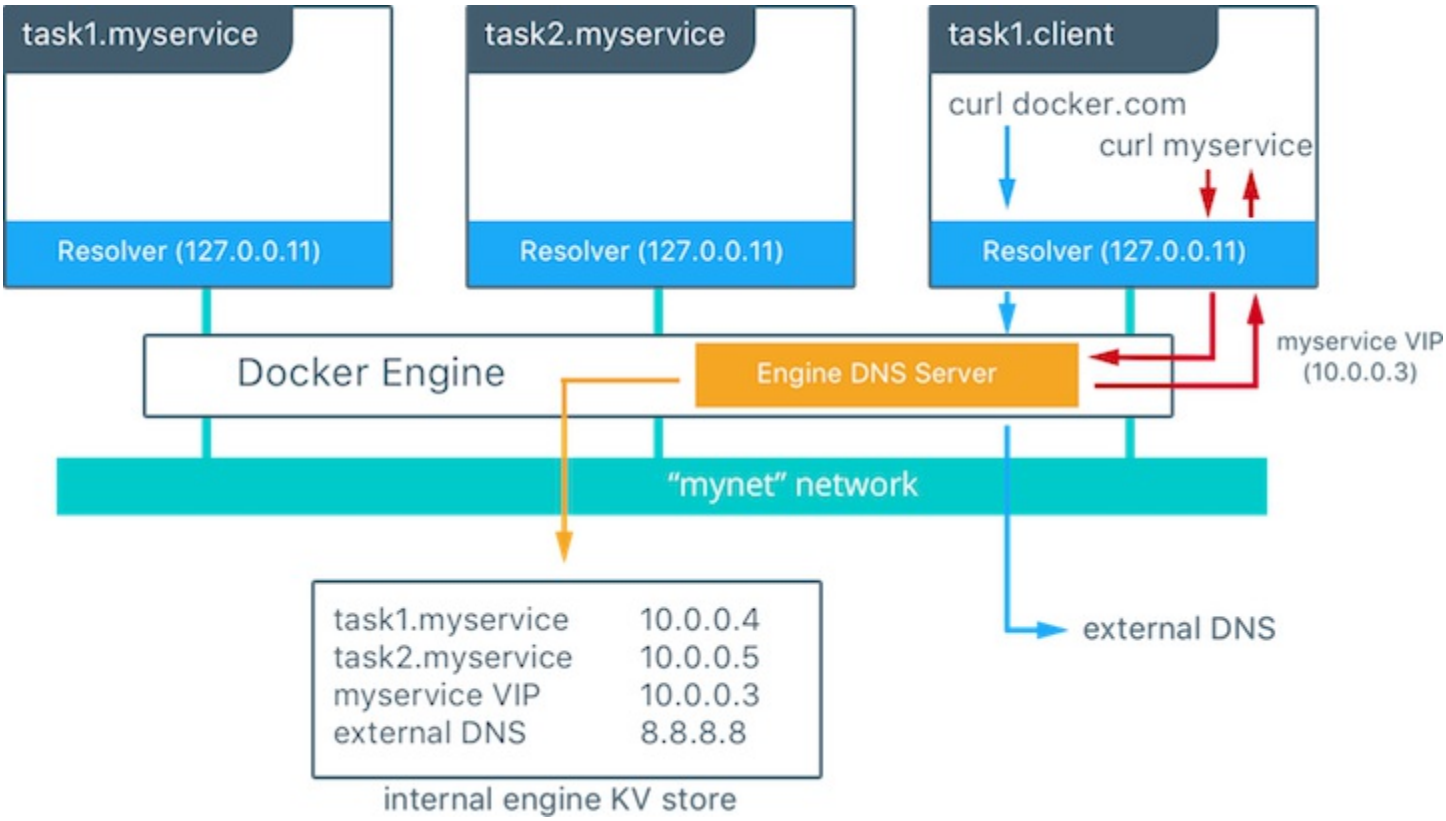
This reference architecture covers the solutions that Docker EE provides in the topic areas of service discovery and load balancing. Docker uses DNS for service discovery as services are created, and different routing meshes are built into Docker to ensure your application remains highly available. The release of UCP 2.0 introduces a new application layer routing mesh called the **HTTP Routing Mesh (HRM)** that routes HTTP traffic based on DNS hostname. After reading this document, you will have a good understanding of how the HRM works and how it integrates with the other Docker service discovery and load balancing features.

# 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 Docker Engine, which acts as a DNS server. 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 Docker Engine looks up the IP address that matches a container's, `task`'s, or`service`'s **name** 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 other's 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, 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"
                },
]
```

> 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.

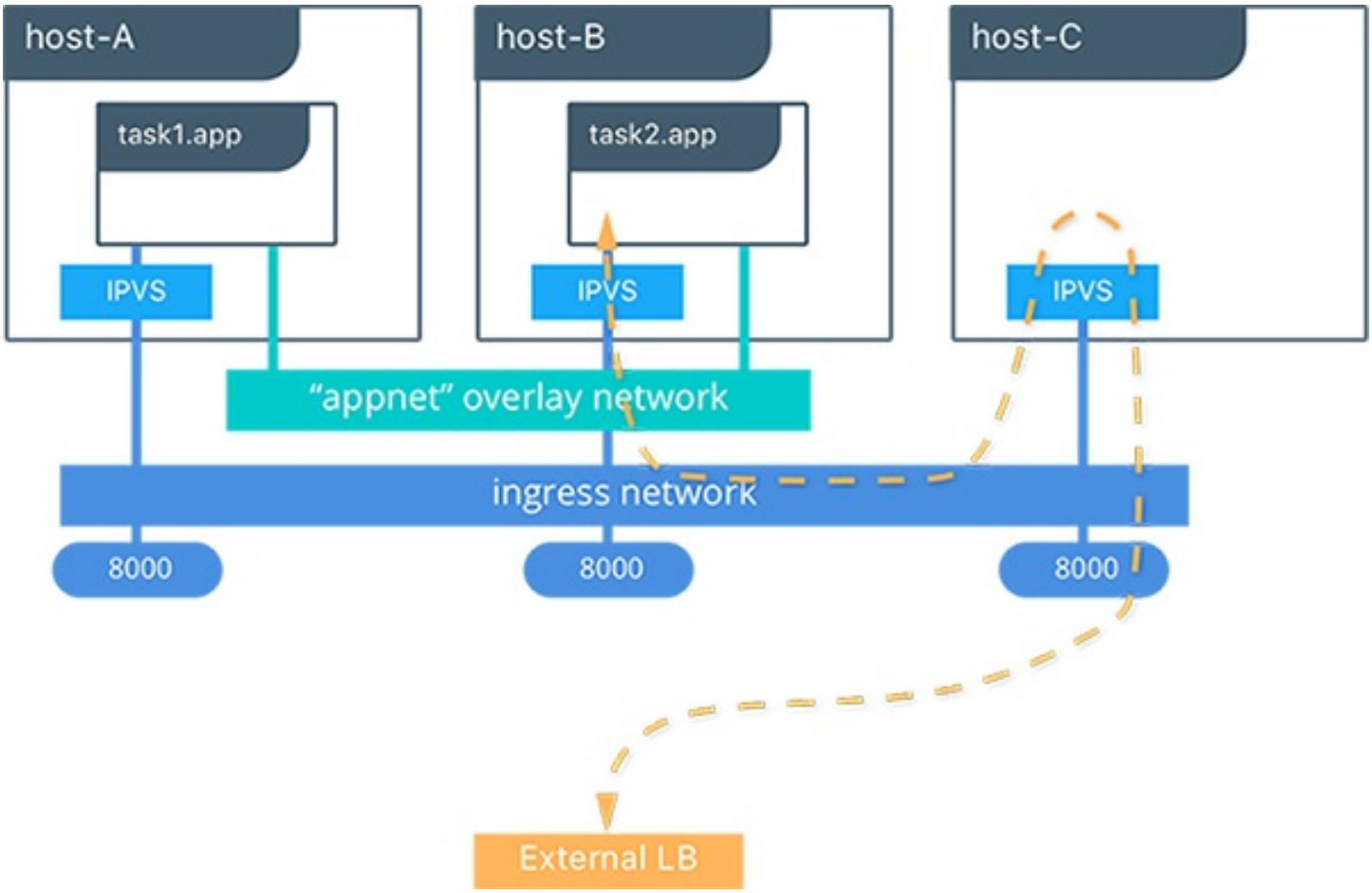
# External Load Balancing (Swarm Mode Routing Mesh)

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. Routing mesh is a new feature in Docker Engine 1.12 that 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 HTTP Routing Mesh

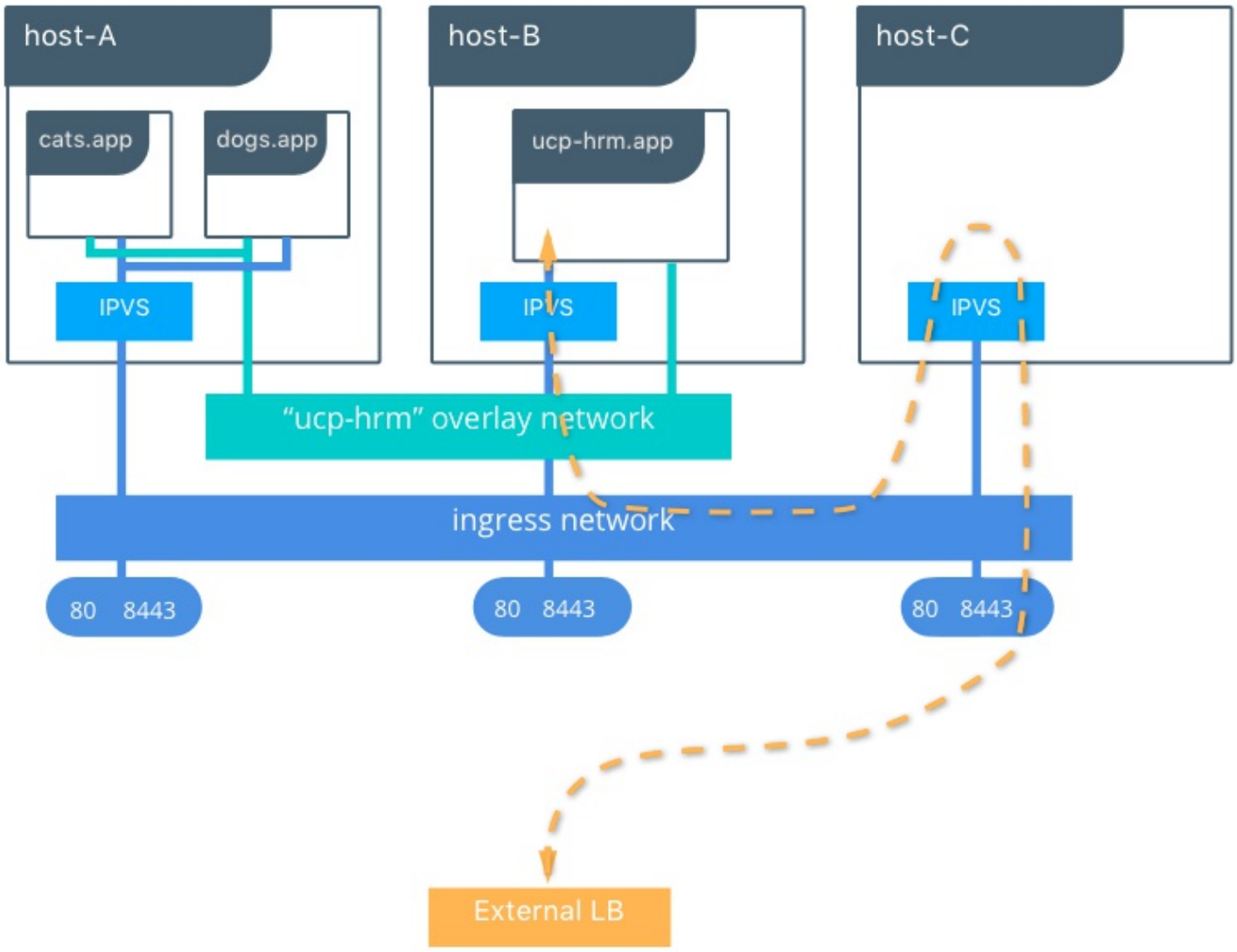
The swarm mode routing mesh is great for transport-layer routing. It routes to services using the service's published ports. But what if you wanted to route traffic to services based on hostname instead? The HTTP Routing Mesh (HRM) is a new feature that enables service discovery on the application layer (L7). The HRM extends upon the swarm mode routing mesh by adding application layer capabilities such as inspecting the HTTP header. The HRM and swarm mode routing meshes are both used together for flexible and robust service delivery. The addition of the HRM 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 HRM 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 HRM with HTTP requests, both the swarm mode routing mesh and the HRM are used in tandem. When a service is created using the `com.docker.ucp.mesh.http` label, the HRM 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 HRM is a service, the HRM is accessible on any node in the cluster using the configured published port.

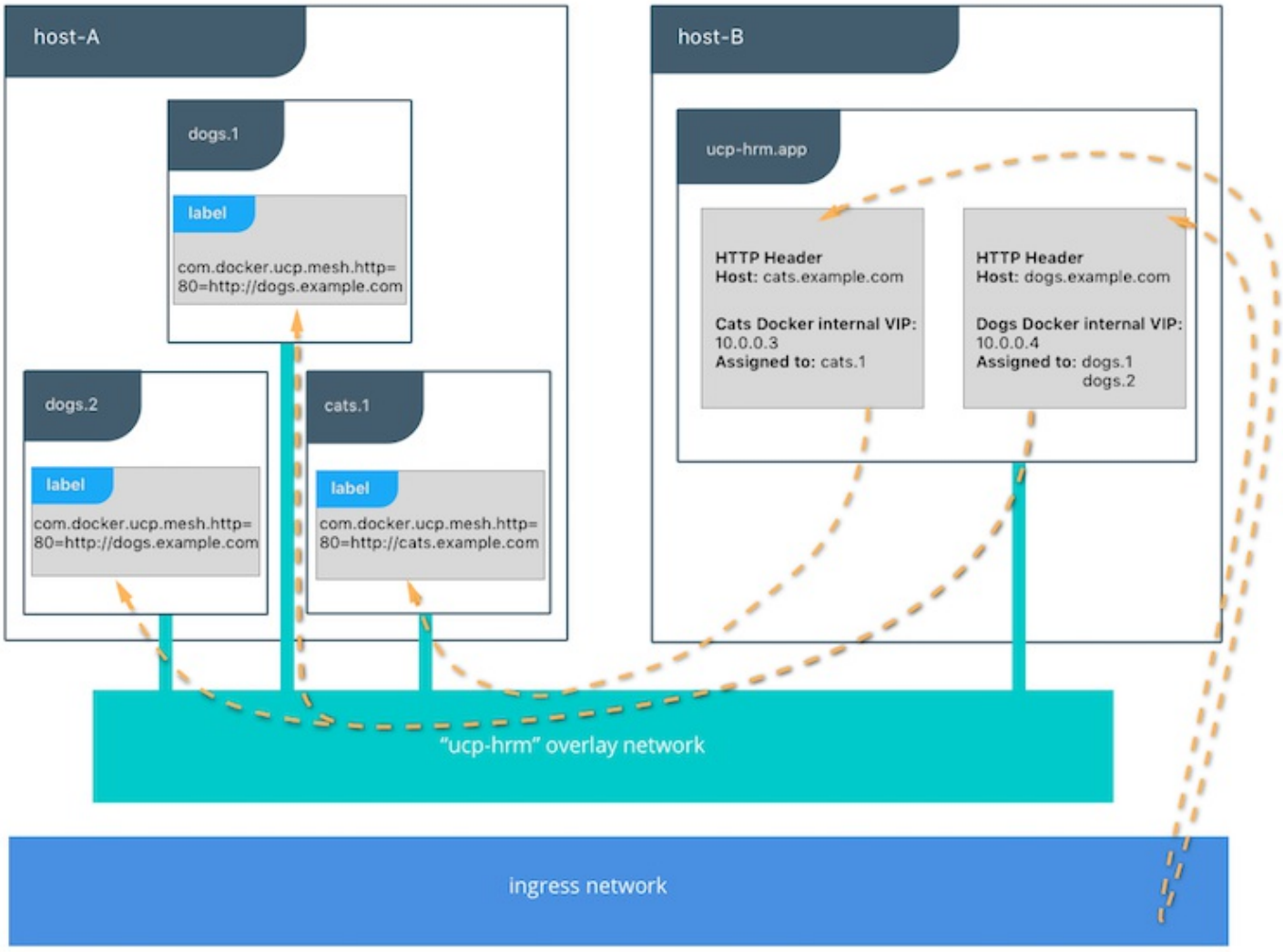
Below is a diagram that displays a higher level view of how the swarm mode routing mesh and HRM work together.




- The traffic comes in from the external load balancer into the swarm mode routing mesh.
- The HRM service is configured to listen on port 80 and 8443, so any request to port 80 or 8443 on the UCP cluster go to the HRM service.

- All services attached to a network that is enabled for "Hostname based routing" can utilize the HRM to have traffic routed based on the HTTP `Host:` header.

Looking closer, you can see what the HRM is doing. The following graphic represents a closer look of the previous diagram.




- Traffic comes in through the swarm mode routing mesh on the `ingress` network to the HRM service's published port.
- As services are created, they are assigned VIPs on the swarm mode routing mesh (L4).
- The HRM receives the TCP packet and inspects the HTTP header.
  - Services that contain the label `com.docker.ucp.mesh.http` are checked to see if they match the HTTP `Host:` header.
  - If a `Host:` header and service label label match, then traffic is routed to the service's VIP 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.

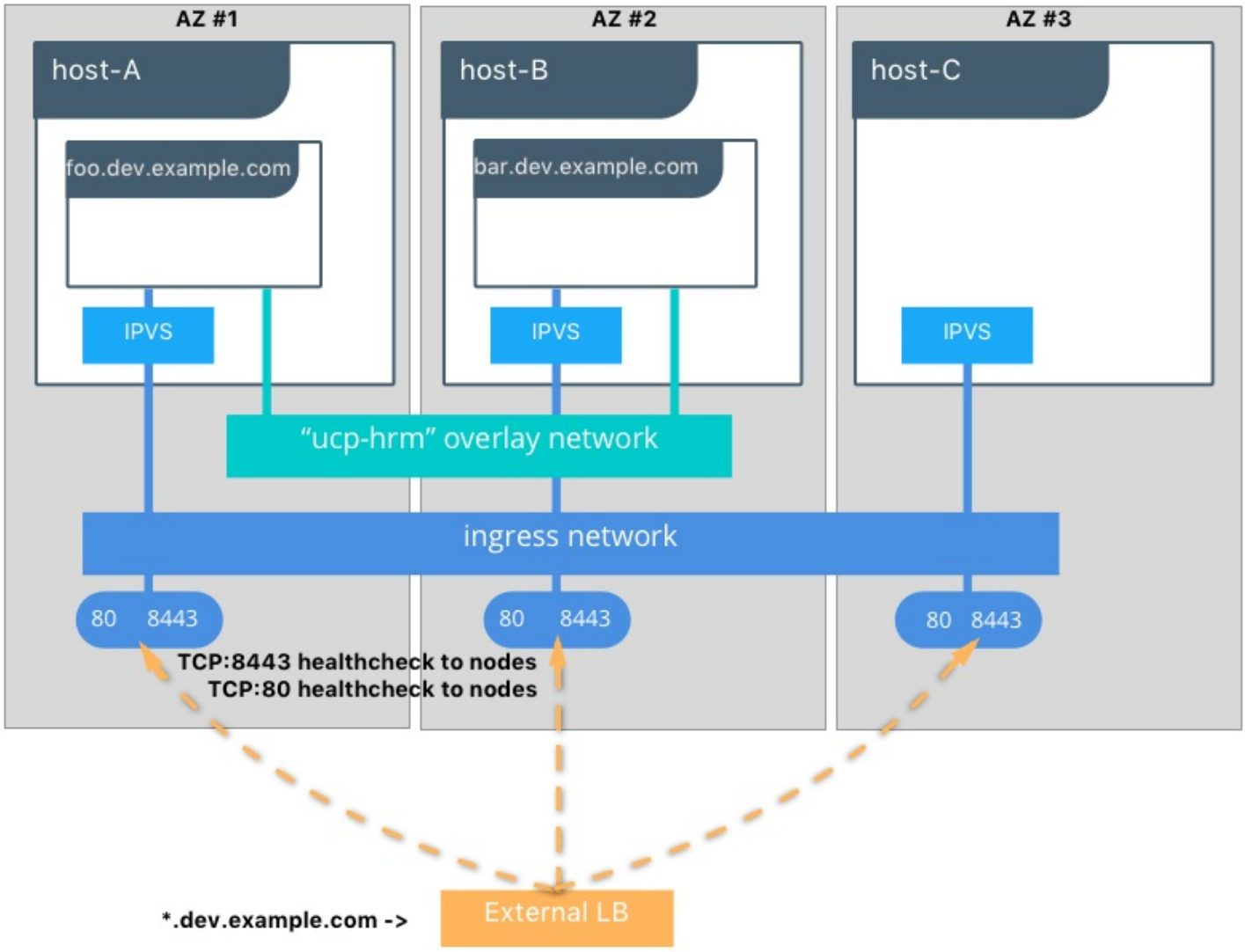
## Differences Between the HRM and Swarm Mode Routing Mesh

The main difference between the HRM and swarm mode routing mesh is that the HRM 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 HRM could be a good fit. If mutual TLS is required to the back-end application, then using the transport layer would probably be preferred.

Another advantage of using the HRM 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 front-end load balancer can be configured to send traffic to the service's published port.

The following diagram shows an example with wildcard DNS:

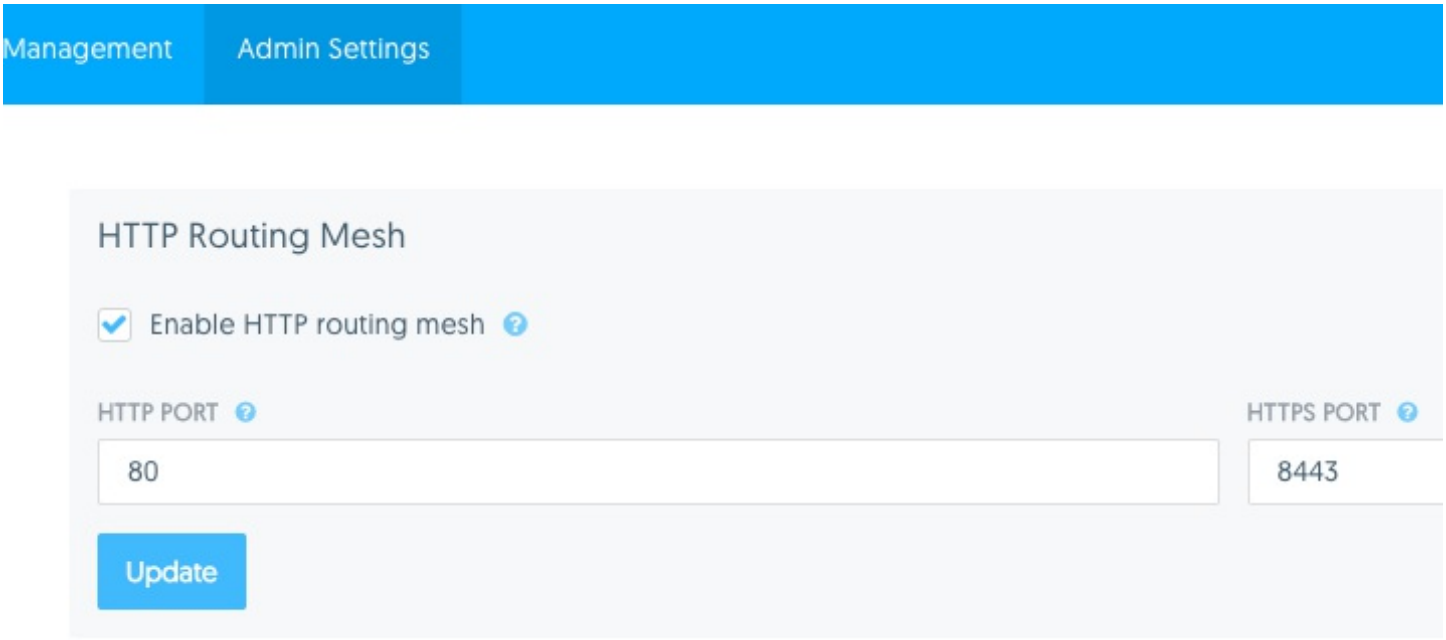



## Enabling the HRM

The HTTP Routing Mesh can be enabled from the UCP web console. To enable it:

1. Login to the UCP web console.

2. Navigate to **Admin Settings** > **Routing Mesh**.
3. Check **Enable HTTP Routing Mesh**.
4. Configure the ports for HRM 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 a service on the swarm cluster to route traffic to the specified container based on the HTTP `Host:` header. Since the HRM service is a *swarm mode* service, every node in the UCP cluster can route traffic to the HRM by receiving traffic from ports `80` and `8443`. The HRM service exposes ports 80 and 8443 cluster-wide, and any requests on ports 80 and 8443 to the cluster are send to the HRM.

## Networks and Access Control

The HTTP routing mesh uses one or more overlay networks to communicate with the back-end services. By default, a single network is created called `ucp-hrm`, with the access control label `ucp-hrm`. Adding a service to this network either requires administrator-level access or the user must be in a group that gives them `ucp-hrm` access.

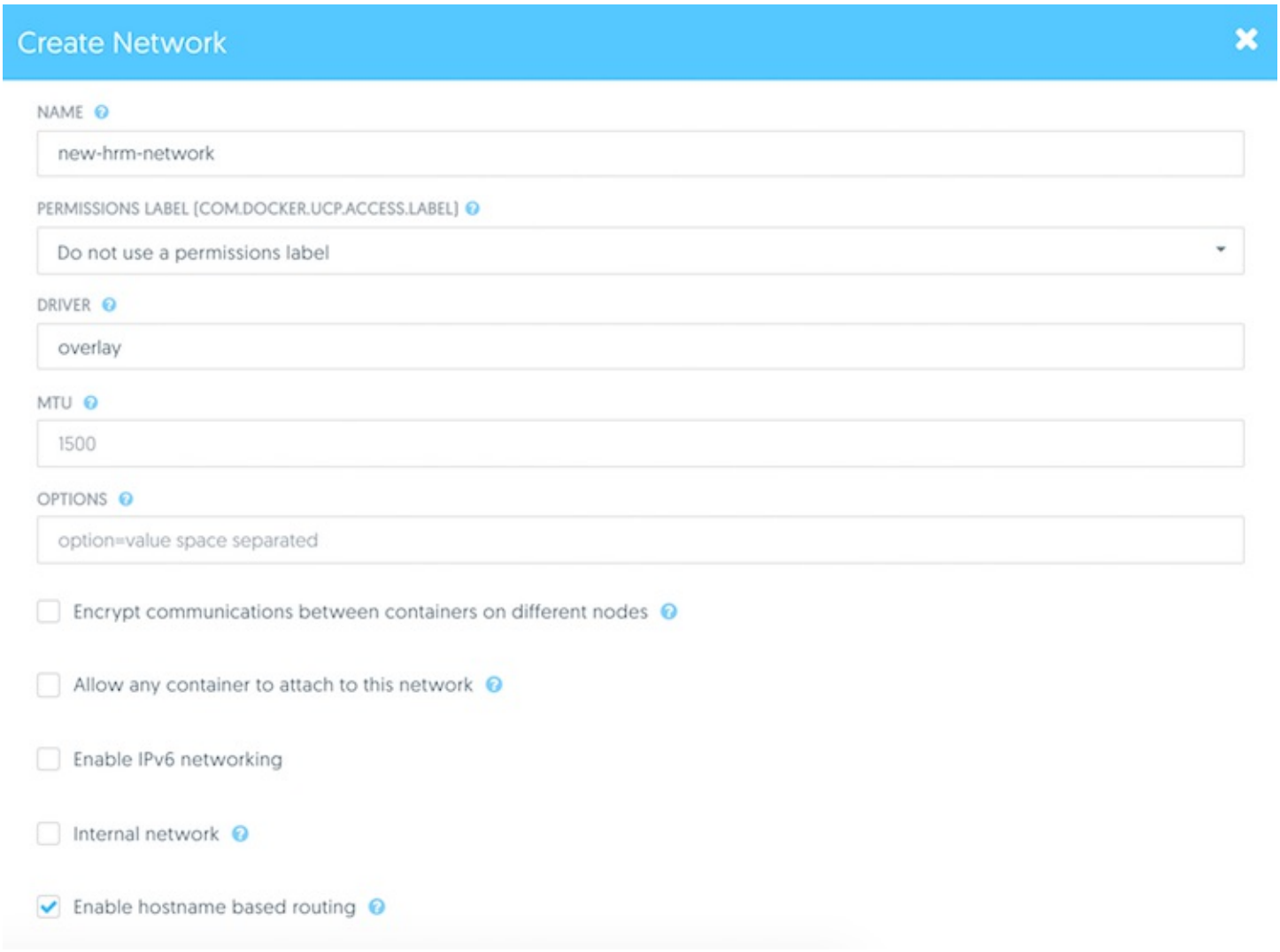
This default configuration does not provide any isolation between services using the HTTP routing mesh, since services share the `ucp-hrm` network.

Isolation between services may be implemented by creating one or more overlay networks with the label `com.docker.ucp.mesh.http` prior to enabling the HTTP Routing Mesh. Once the HRM is enabled, it is able to route to all services attached to any of these networks, but services on different networks can't communicate directly. The only way to have the HRM available on a new network is to disable and then re-enable the HRM.

The following is an example of creating an overlay network that contains the `com.docker.mesh.http` label. When using a UCP client bundle for an admin user, or a user with administrator privileges, you can run the following command:

```
docker network create -d overlay --label com.docker.ucp.mesh.http=true new-hrm-network
```

The same can be accomplished with the UCP UI by selecting **Enable hostname based routing** when creating a network.




# HRM Requirements

There are three requirements for services to use the HRM.

1. The service must be connected to a network with the `com.docker.ucp.mesh.http` label
2. The service must publish one or more ports
3. The service must contain one or more labels prefixed with `com.docker.ucp.mesh.http` to specify the ports to route

# Configuring DNS with the HRM

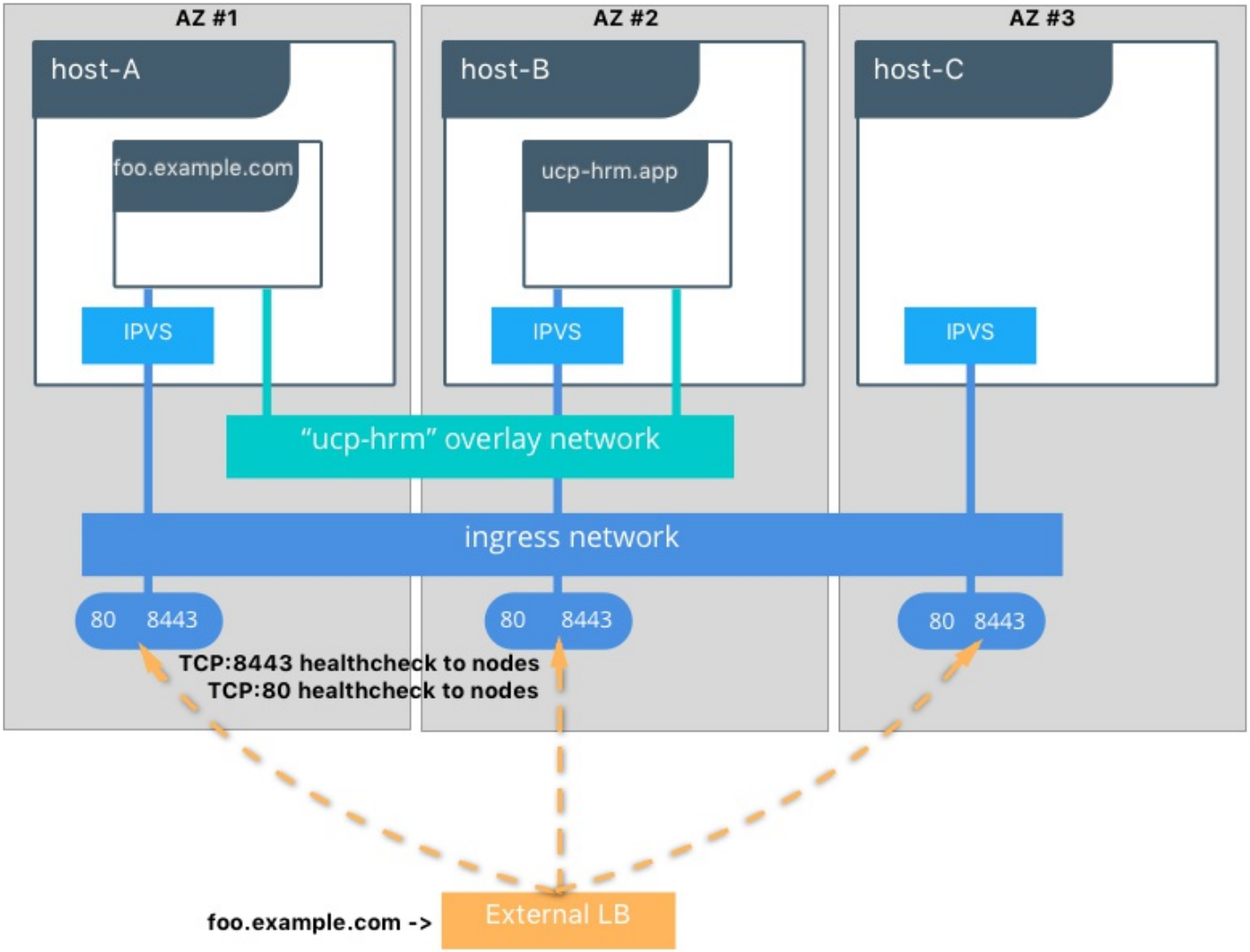
This section covers how to configure DNS for services using the HRM. To use the HRM, a DNS record for the service needs to point to the UCP cluster. This can be accomplished through a variety of different ways because of the flexibility that the swarm mode routing mesh provides.

If a service needs to be publicly accessible for requests to `foo.example.com`, then the DNS record for that service can be configured 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 HRM.
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 HRM.

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 HRM service's configured published port 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**: They generally don't shift around (new hosts, new IPs, etc.) often, and it is easier to keep load balancers pointing to the same nodes.
   - **Disadvantage**: They 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**: They don't manage the entire cluster, so there's less additional networking overhead.

- **Disadvantage**: They fall more into the "cattle" territory 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.

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

# HRM Usage

Now that you have an understanding on how the HRM works and understand the requirements associated with it, this section covers the syntax for the HRM for HTTP routing, logging, monitoring, and replicas.

## HTTP Routing

Services must contain a label where the key of the label begins with `com.docker.ucp.mesh.http`. If a service needs to expose multiple ports, then multiple labels can be used such as `com.docker.ucp.mesh.http.80` and `com.docker.ucp.mesh.http.443`. Here `80` and `443` are used to differentiate the HRM labels via port numbers. You can use whatever values you want, just make sure that they are different from each other and that you can keep track of them.

The key of the label attached to a service to be used by HRM must begin with `com.docker.ucp.mesh.http`, for example `com.docker.ucp.mesh.http.80` and `com.docker.ucp.mesh.http.443`.

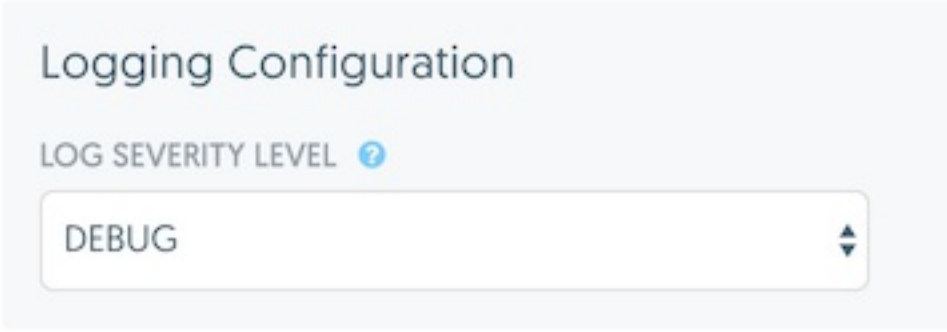
The value of the label is a comma-separated list of key/value pairs separated by equals signs. The following pairs can be used:

- `external_route` **(required)** — The external URL to route to this service. Examples: `[http://myapp.example.com](http://myapp.example.com "http://myapp.example.com")` and `[sni://myapp.example.com](https://success.docker.com/api/asset/.%2Frefarch%2Fucp-2-0-service-discovery%2Fsni:%2F%2Fmyapp.example.com "sni:%2F%2Fmyapp.example.com")`
- `internal_port` — The internal port to use for the service. This is **required** if more than one port is published by the service. Examples: `80` and `8443`
- `sticky_sessions` — If present, use the named cookie to route the user to the same back-end task for this service. Details are in the *Sticky Sessions* section later in this document.
- `redirect` — If present, perform redirection to the specified URL. See the *Redirection* section later in this document.

## Logging

It's possible to log the traffic passing through HRM by performing these steps:

1. In the UCP UI go to **Admin Settings** -> **Logs**.
2. Set the **LOG SEVERITY LEVEL** to **DEBUG**.




3. Update the HRM server 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-hrm
```

## Monitoring

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

## HRM HA Considerations

This section discusses a few usage considerations with HRM.

If you are utilizing the sticky sessions feature, the stick table that HRM uses for persistence is not shared between all replicas — thus, only one replica of HRM can be run. In other words, if cookie-based persistence is used, then HRM can only be run as one replica.

If only HTTP routing (without sticky sessions) and HTTPS routing are going to be used, then the HRM can be scaled to more than one replica for HA purposes.

If you don't need to use cookie-based persistence, you can scale the HRM service to more than one replica. For example, to use 3 replicas:

```
docker service update --replicas 3 ucp-hrm
```

# HRM Usage Examples

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

- HTTP Routing
- Sticky Session
- HTTPS
- Multiple Ports
- 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 them using the published HRM ports (80 and 8443 by default).

> **Note:** The repository for the sample application can be found on GitHub (https://github.com/ahromis/spring-session-docker-demo).

## HTTP Routing Example

Consider an example 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:

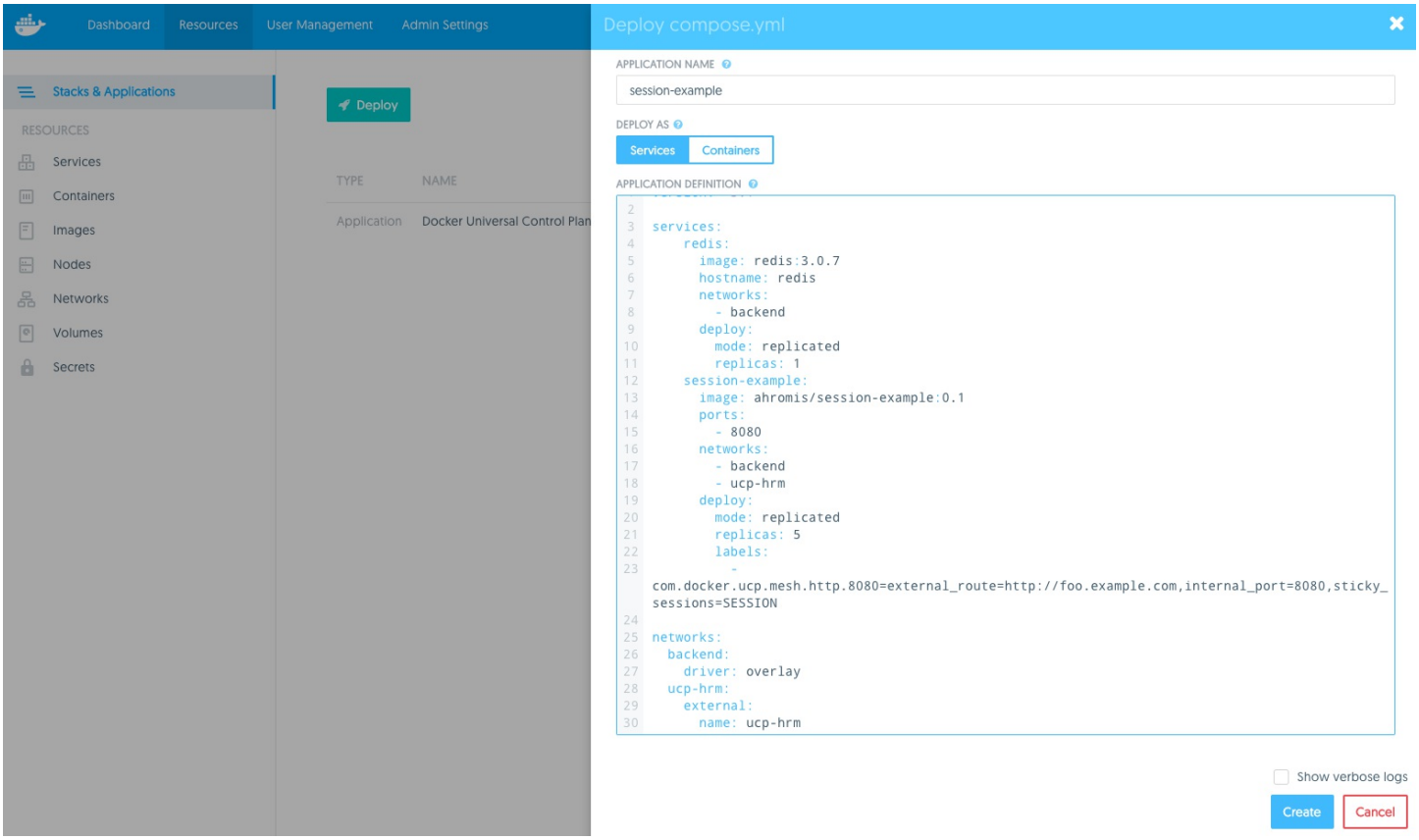
```
$ wget https://raw.githubusercontent.com/ahromis/spring-session-docker-demo/master/ucp-demo/docker-compose.hrm.http.yml
$ DOMAIN=<domain-to-route> docker stack deploy -c docker-compose.hrm.http.yml hrm-http-example
```

Then access the example application at `http://<domain-to-route>`, and log into it with **user:** `user` and **password:** `password`.

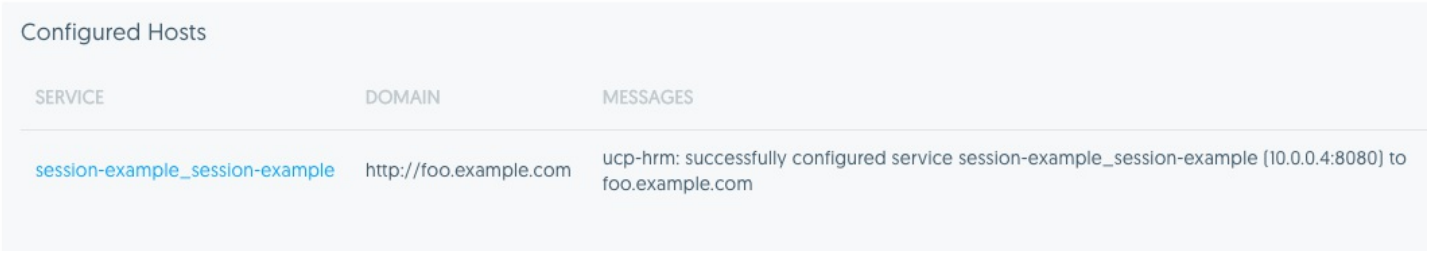
This is the contents of the compose file if you just want to copy/paste into the UCP UI instead:

```
version: "3.1"

services:
    redis:
      image: redis:3.0.7
      hostname: redis
      networks:
        - back-end
      deploy:
        mode: replicated
        replicas: 1
    session-example:
      image: ahromis/session-example:0.1
      ports:
        - 8080
      networks:
        - back-end
        - ucp-hrm
      deploy:
        mode: replicated
        replicas: 5
        labels:
          - com.docker.ucp.mesh.http.8080=external_route=http://${DOMAIN},internal_port=8080

networks:
  back-end:
    driver: overlay
  ucp-hrm:
    external:
      name: ucp-hrm
```

It's possible to deploy through the UCP UI be going to **Resources** -> **Stacks & Applications** -> **Deploy**. Name the stack, and copy/paste the above compose file into the open text field. Be sure to substitute the `${DOMAIN}` variable when deploying through the UI.




Once HRM discovers the newly created service, it will list it in the UCP. Go to **Admin Settings** -> **Routing Mesh**. The new application should be listed under **Configured Hosts**.




## HRM Service Deployment Breakdown

The HRM polls every 30 seconds, so once the application launches, HRM polls for the new service and finds it at `http://<domain-to-route>`.

When the compose file is deployed, the following happens:

1. Two services are created, a front-end Spring Boot application and a Redis service to store session data.
2. A new overlay network specific to this particular application stack is created called `<stack-name>_backend`.
3. The Redis task creates a DNS A Record of `redis`, on the `backend` network. This DNS record directs to the IP address of the Redis container.
    - The configuration for the front-end application doesn't need to change for every stack when accessing Redis. It remains as `spring.redis.host=redis` in the application configuration for every environment.
4. A DNS entry of the front-end service name, `session-example`, then registers on the `ucp-hrm` network.
5. The front-end service is created and attached to the `ucp-hrm` service.
6. The declared healthy state for the service is 5 replicas, so 5 replica tasks are created.

7. The HRM polls every 30 seconds for Docker events, and it picks up the `com.docker.ucp.mesh.http.8080` label on the newly created service.
8. The HRM creates an entry so that all 5 front-end replicas are load balanced backed on the `$DOMAIN` environment variable that was passed for the stack deploy.
9. By doing a refresh of `[http://$DOMAIN](http://$DOMAIN "http://$DOMAIN")` in a web browser, a new hostname should show up with every request. It is load balancing across all of the front-end service replicas.
10. Click on the link to log in. The credentials are **User:** `user`, **Password:** `password`.

## HRM Sticky Session Example

The HTTP Routing Mesh has the ability to route to a specific back-end 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 passing `sticky_sessions=JSESSIONID` to the HRM label. Sticky connections are accomplished in HRM by using stick tables, where HRM learns and uses the application session cookie to persist connections to a specific back-end replica.

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 back-end 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 won't have a "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 back-end 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.

> **Note:** Sticky sessions are not available when using HTTPS as the value of the cookie is encrypted in the HTTP header.

To deploy the example application stack for sticky sessions, run these commands with the UCP client bundle loaded:

```
wget https://raw.githubusercontent.com/ahromis/spring-session-docker-demo/master/ucp-demo/docker-compose.hrm.sticky.yml
DOMAIN=<domain-to-route> docker stack deploy -c docker-compose.hrm.http.yml hrm-sticky-example
```

Access the example application at `http://<domain-to-route>`, and log into it with **user:** `user`, **password:** `password`.

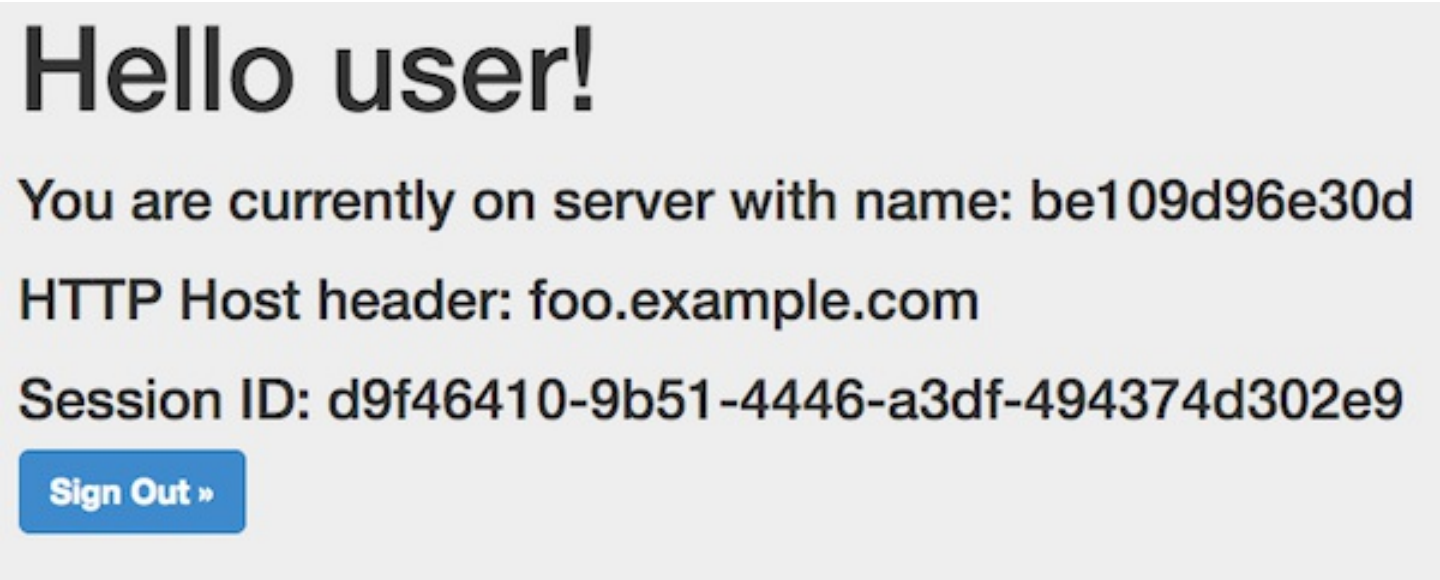
This is the contents of the compose file if you want to copy/paste into the UCP UI instead:

```
version: "3.1"

services:
    redis:
      image: redis:3.0.7
      hostname: redis
      networks:
        - back-end
      deploy:
        mode: replicated
        replicas: 1
    session-example:
      image: ahromis/session-example:0.1
      ports:
        - 8080
      networks:
        - back-end
        - ucp-hrm
      deploy:
        mode: replicated
        replicas: 5
        labels:
          -
com.docker.ucp.mesh.http.8080=external_route=http://${DOMAIN},internal_port=8080,sticky_sessions=SESSION


networks:
  back-end:
    driver: overlay
  ucp-hrm:
    external:
      name: ucp-hrm
```

Be sure to substitute the `${DOMAIN}` variable when deploying through the UI.

## HRM Sticky Session Breakdown

When accessing and logging into the application, you should see a page similar to the following:



This is the same as the HTTP routing example but with the additional key value entry of `sticky_sessions=SESSION`.

What does adding `sticky_sessions` to the `com.docker.ucp.mesh.http` do?

1. The HRM creates an entry so that all 5 front-end replicas have their IPs added to the configuration. In addition to this configuration, the name of the session cookie to base persistence on is added.
2. Load `[http://$DOMAIN](http://$DOMAIN "http://$DOMAIN")` in a web browser, login with **User:** `user`, **Password:** `password`. With sticky sessions, refreshing the page should show the same back-end server. Connections to the back-end instance persists based on the value of the `SESSION` cookie.

This demo application uses Redis as the distributed storage for the session data. It's possible to see the `SESSION` cookie stored in Redis by opening a console to the Redis container in the UCP UI.

1. Login to the UCP UI.
2. Click on **Stacks & Applications** in the left-hand navigation pane.
3. Select your stack from the list.
4. Select the redis service from the list.
5. Click on **Tasks** on the top.
6. Select the Redis container.
7. Select **Console** on the top.
8. Use `sh` to connect to the console stack.
9. Run `redis-cli keys "*"` on the console.


# HRM HTTPS Example

The HTTP routing mesh has support for routing using HTTPS. Using a feature of HTTPS called *Server Name Indication*, the HRM is able to route connections to service back-ends without terminating the HTTPS connection. SNI is an extension of the TLS protocol, where the client indicates which hostname it is attempting to connect to at the start of the handshake process.

To use HTTPS support, no certificates for the service are provided to the HTTP routing mesh. Instead, the back-end service must handle HTTPS connections directly. Services that meet this criteria can use the SNI protocol to indicate handling of HTTPS in this manner. By terminating at the application servers, encrypted traffic goes all the way to the application. However, this also means that applications must manage the TLS certificates. By leveraging Docker secrets, certificates for application servers can be securely and easily be managed.

When using HRM with HTTPS, the connections are re-used to reduce the overhead of re-negotiating new TLS connections. The HTTPS connections are re-used using the SSL session ID, and they persist to a service task until the connection needs to be re-established (i.e. the application server connection timeout is reached).

Secrets and certificates are almost always involved when dealing with encrypted communications. Before deploying the example application, create a Java keystore.

> **Tip:** Here's some helpful commands to create a key store for use with Java applications.

```
# create PKCS#12 file format
$ openssl pkcs12 -export -out keystore.pkcs12 -in fullchain.pem -inkey privkey.pem
# convert PKCS file into Java keystore format
$ docker run --rm -it -v $(pwd):/tmp -w /tmp java:8 \
    keytool -importkeystore -srckeystore keystore.pkcs12 -srcstoretype PKCS12 -destkeystore
keystore.jks
```

Now that a Java keystore has been created, it's time to turn it into a Docker secret so that it can securely be used by this application.

```
$ docker secret create session-example_keystore.jks_v1 keystore.jks
$ echo "<your-key-store-password>" | docker secret create session-example_keystore-password.txt_v1 -
$ echo "<your-key-password>" | docker secret create session-example_key-password.txt_v1 -
```

That's it! Now the secrets are 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/Architecture/Docker_Reference_Architecture%3A_Securing_Docker_EE_and_

To add more context, this is what the application configuration for the example application is using:

```
spring.redis.host=redis
spring.redis.port=6379
server.port=8443
server.ssl.key-store=file:/run/secrets/keystore.jks
server.ssl.key-store-password=${KEY_STORE_PASSWORD}
server.ssl.key-password=${KEY_PASSWORD}
```

The environment variables are from an `ENTRYPOINT` script that reads the secrets then exposes it to Spring Boot. More information can be found in the GitHub repository (https://github.com/ahromis/spring-session-docker-demo) for this application.

Now that the certs are securely created and stored in the Docker kv store, it's time to create a service that can use them:

```
wget https://raw.githubusercontent.com/ahromis/spring-session-docker-demo/master/ucp-demo/docker-
compose.hrm.ssl.yml
DOMAIN=<domain-to-route> docker stack deploy -c docker-compose.hrm.http.yml hrm-sticky-example
```

Access the example application at `https://<domain-to-route>`. If you aren't using a load balancer placed in front of your UCP cluster, then use `https://<domain-to-route>:8443`. Then log into it with **user:** `user`, **password:** `password`.

Here is the full compose file if you want to copy/paste from the UI instead (remember to replace the `${DOMAIN}` variables):

```
version: "3.1"

services:
    redis:
      image: redis:3.0.7
      hostname: redis
      networks:
        - back-end
      deploy:
        mode: replicated
        replicas: 1
    session-example:
      image: ahromis/session-example:0.1
      ports:
        - 8443
      environment:
        - SPRING_PROFILES_ACTIVE=https
      networks:
        - back-end
        - ucp-hrm
      deploy:
        mode: replicated
        replicas: 5
        labels:
          - com.docker.ucp.mesh.http.1=external_route=http://${DOMAIN},redirect=https://${DOMAIN}
          - com.docker.ucp.mesh.http.8443=external_route=sni://${DOMAIN},internal_port=8443
      secrets:
        - source: session-example_keystore.jks_v1
          target: keystore.jks
          mode: 0400
        - source: session-example_keystore-password.txt_v1
          target: keystore-password.txt
          mode: 0400
        - source: session-example_key-password.txt_v1
          target: key-password.txt
          mode: 0400

networks:
  back-end:
    driver: overlay
  ucp-hrm:
    external:
      name: ucp-hrm

secrets:
    session-example_keystore.jks_v1:
        external: true
    session-example_keystore-password.txt_v1:
        external: true
    session-example_key-password.txt_v1:
        external: true
```

The above example also shows how redirection from HTTP to HTTPS can be handled by using the `redirect` keypair.

It's worth mentioning how easily repeatable this is, even for existing applications. Only a few simple steps need to be performed to Modernize Traditional Applications (MTA), and the application stack can be deployed multiple times with minimal configuration changes.

## HRM Multiple Ports Example

Sometimes a service has multiple ports it can listen on. With HRM each listen port can be routed to independently.

Here's an example of a service with multiple listen ports:

```
$ docker service create \
  -l com.docker.ucp.mesh.http.8000=external_route=http://site1.example.com,internal_port=8000 \
  -l com.docker.ucp.mesh.http.8001=external_route=http://site2.example.com,internal_port=8001 \
  -p 8000 \
  -p 8001 \
  --network ucp-hrm \
  --replicas 3 \
  --name twosite ahromis/nginx-twosite:latest
```

In this example an Nginx service is created that has two web roots that listen on different ports. The HRM routes traffic to each of the sites independently based on the HTTP `Host:` header it receives.

## HRM Redirection Example

Redirect from HTTP to HTTPS when you want to force all connections to be secure. The redirect option indicates that all requests to this route should be redirected to another domain name using an HTTP redirect.

One use of this feature is for a service which only listens using HTTPS, with HTTP traffic to it being redirected to HTTPS. If the service is on example.com, then this can be accomplished with two labels:

```
com.docker.ucp.mesh.http.1=external_route=http://example.com,redirect=https://example.com
com.docker.ucp.mesh.http.2=external_route=sni://example.com
```

Another use is a service expecting traffic only on a single domain, but there are other domains that should be redirected to it. For example, a website that has been renamed might use this functionality. The following labels accomplish this for `new.example.com` and `old.example.com`.

```
com.docker.ucp.mesh.http.1=external_route=http://old.example.com,redirect=http://new.example.com
com.docker.ucp.mesh.http.2=external_route=http://new.example.com
```

Below is an example redirecting a website to a different domain using a different example application.

```
$ docker service create \
  -l
com.docker.ucp.mesh.http.1=external_route=http://oldsite.example.com,redirect=http://foo.example.com \
  -l com.docker.ucp.mesh.http.8080=external_route=http://foo.example.com,internal_port=8080 \
  --replicas 3 \
  --name lbinfo \
  ahromis/lbinfo:latest
```

# Non Swarm Mode Containers

The HRM and swarm mode routing mesh are only supported for applications deployed using "services." For non-swarm mode containers, such as containers running on pre-1.12 Docker Engines and applications deployed not using services (e.g using `docker run`), interlock (https://github.com/ehazlett/interlock) and NGINX must be used.

Interlock is a containerized, event-driven tool that connects to the UCP controllers and watches for events. In this case, events are the containers being spun up or going down. Interlock also looks for certain metadata that these containers have such as hostnames or labels configured for the container. It then uses the metadata to register/de-register these containers to a load-balancing back-end (NGINX). The load balancer uses updated back-end configurations to direct incoming requests to healthy containers. Both Interlock and the load balancer containers are stateless and, hence, can be scaled horizontally across multiple nodes to provide a highly-available load balancing services for all deployed applications.

There are three requirements for containers to use Interlock and NGINX:

**I. Interlock and NGINX need to be deployed on one or more UCP worker nodes.**

The easiest way to deploy Interlock and NGINX is by using Docker Compose in the UCP portal:

1. Log into the UCP web console.
2. On top, go to the **Resources** tab.
3. Go to **Applications** in the left pane, and click the **Deploy compose.yml** button.
4. Enter `interlock` as the **Application Name**.
5. For the `compose.yml` file, enter the following sample `compose.yml` file. You can alter the Interlock or NGINX config as you desire. Full documentation is on GitHub (https://github.com/ehazlett/interlock/tree/master/docs).

```
interlock:
        image: ehazlett/interlock:1.3.0
        command: -D run
        tty: true
        ports:
            - 8080
        environment:
            constraint:node==${UCP_NODE_NAME}:
            INTERLOCK_CONFIG: |
                ListenAddr = ":8080"
                DockerURL = "tcp://${UCP_CONTROLLER_IP}:2376"
                TLSCACert = "/certs/ca.pem"
                TLSCert = "/certs/cert.pem"
                TLSKey = "/certs/key.pem"
                PollInterval = "10s"
                [[Extensions]]
                Name = "nginx"
                ConfigPath = "/etc/nginx/nginx.conf"
                PidPath = "/etc/nginx/nginx.pid"
                MaxConn = 1024
                Port = 80
        volumes:
            - ucp-node-certs:/certs
        restart: always


    nginx:
        image: nginx:latest
        entrypoint: nginx
        command: -g "daemon off;" -c /etc/nginx/nginx.conf
        ports:
            - 80:80
        labels:
            - "interlock.ext.name=nginx"
        restart: always
        environment:
            constraint:node==${UCP_NODE_NAME}:
```

> **Note:** Substitute `UCP_NODE_NAME` and `UCP_CONTROLLER_IP`. `UCP_NODE_NAME` is the name of the node that you wish to run Interlock and NGINX on (as displayed under the **Resources**/**Nodes** section). The DNS name for your application(s) needs to resolve to this node. `UCP_CONTROLLER_IP` is the IP or DNS name of one or more UCP controllers.

6. Click **Create** to deploy Interlock and NGINX.

> **Note:** You can deploy Interlock and NGINX on multiple nodes by repeating steps 3-6 above and changing the **Application Name** and `UCP_NODE_NAME`. This allows you to front these nodes with an external load balancer ( e.g ELB or F5) for high availability. The DNS records for your applications would then need to be registered to the external load balancer IP.

**II. The container must publish one or more ports.**

**III. The container must be launched with Interlock labels (https://github.com/ehazlett/interlock/blob/master/docs/interlock_data.md).**

For example, to deploy a container that exposes port `8080` and is accessed on the DNS name `demo.app.example.com`, launch it as follows:

```
docker run --name demo -p 8080 --label interlock.hostname=demo --label interlock.domain=app.example.com
ehazlett/docker-demo:dcus
```

Once you launch your container with the correct labels and published ports, you can access it using the desired DNS name.

# 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 applications in a shorter amount of time.
ck as a Kubernetes workload. We will deploy 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.

Login to 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.

So let's create a `PersistentVolume` using the definition below. 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.

We 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 their 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) etc. 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 how exactly inbound connections reach application services running within the Kubernetes cluster. Configurations like host based URLs, load balancing, SSL terminations or passthrough, or even to configure edge routers. An ingress resource alone will not be sufficient for routing of application traffic; it would invariably require one or more ingress controllers to satisfy the rules and configurations of the ingress resource.

In this example, we will use the standard Kubernetes nginx based Ingress Controller (https://github.com/kubernetes/ingress-nginx) in Docker EE to expose an application over L7.

We will configure 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, we need to grant the `default` service account in the `ingress-nginx` read access to all other namespaces.

## Create the `ingress-nginx` namespace

Login to UCP and navigate to **Kubernetes** —> **Namespace** —> **Create**. In the **Create Kubernetes Object** pane, under **Object YAML**, paste the yaml text as below:

```
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. The steps will create a grant with `Restricted Control` permissions.

- Login to 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 and under `Object YAML`, paste the following text.

```
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 will show a blank page. This indicates that the `ingress-controller` is working appropriately.

## Deploy applications using ingress

In this section, we will deploy the same application as demonstrated before to use ingress' L7 routing.

In order for the example application to work, you would 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 address 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.

We will need to edit the stack so that the frontend (web) service is assigned a **ClusterIP** 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 on **Save**.

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' 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.

We can deploy another application in a separate namespace that will 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 steps below are very similar to the previous application deployment procedure and will result in the deployment of an application called `words` in the `blue` namespace. It will also configure the application to use L7 capabilities by integrating it with the nginx ingress controller.

Create a new namespace called `blue` using the yaml below 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 below 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 we can create the Ingress resource by navigating to **Kubernetes** —> **+ Create**. Select the `blue` namespace and paste the yaml below 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.