In this session, we'll explore how you can use the NGINX web server to load balance requests between two containers running on the host.

With Docker, there are two main ways for containers to communicate with each other. The first is via links which configure the container with environment variables and host entry allowing them to communicate. The second is using the Service Discovery pattern where uses information provided by third parties, in this scenario, it will be Docker's API.

The Service Discovery pattern is where the application uses a third party system to identify the location of the target service. For example, if our application wanted to talk to a database, it would first ask an API what the IP address of the database is. This pattern allows you to quickly reconfigure and scale your architectures with improved fault tolerance than fixed locations.

The machine name Docker is running on is called docker. If you want to access any of the services, then use docker instead of localhost or 0.0.0.0.

Step 1 - NGINX Proxy
In this scenario, we want to have a NGINX service running which can dynamically discovery and update its load balance configuration when new containers are loaded. Thankfully this has already been created and is called nginx-proxy.

Nginx-proxy accepts HTTP requests and proxies the request to the appropriate container based on the request Hostname. This is transparent to the user with happens without any additional performance overhead.

Properties
There are three keys properties required to be configured when launching the proxy container.

The first is binding the container to port 80 on the host using -p 80:80. This ensures all HTTP requests are handled by the proxy.

The second is to mount the docker.sock file. This is a connection to the Docker daemon running on the host and allows containers to access its metadata via the API. Nginx-proxy uses this to listen for events and then updates the NGINX configuration based on the container IP address. Mounting file works in the same way as directories using -v /var/run/docker.sock:/tmp/docker.sock:ro. We specify :ro to restrict access to read-only.

Finally, we can set an optional _-e DEFAULTHOST=<domain>. If a request comes in and doesn't make any specified hosts, then this is the container where the request will be handled. This enables you to run multiple websites with different domains on a single machine with a fall-back to a known website.

Task
Use the command below to launch nginx-proxy.

docker run -d -p 80:80 -e DEFAULT_HOST=proxy.example -v /var/run/docker.sock:/tmp/docker.sock:ro --name nginx jwilder/nginx-proxy

Because we're using a DEFAULT_HOST, any requests which come in will be directed to the container that has been assigned the HOST proxy.example.

Request
You can make a request to the web server using- curl http://docker.
As we have no containers, it will return a 503 error.

```
See 'docker --help'.
$ docker run -d -p 80:80 -e DEFAULT_HOST=proxy.example -v /var/run/docker.sock:/tmp/docker.sock:ro --name nginx jwilder/nginx-proxy
1ceab313222e8da94ea4383a57ce445e9ce0c72277332f9bd71743c47715fc87
$ curl http://docker
<html>
<head><title>503 Service Temporarily Unavailable</title></head>
<body>
<center><h1>503 Service Temporarily Unavailable</h1></center>
<hr><center>nginx</center>
</body>
</html>
$
```

docker.sock is a Unix socket file used by the Docker daemon to communicate with the Docker client. It is a special file that allows processes to communicate with the Docker daemon without requiring network connectivity.

When the Docker daemon starts, it creates a Unix socket file called docker.sock in the /var/run/docker.sock directory. The Docker client communicates with the daemon by sending requests to this socket file, and the daemon responds by sending data back through the same socket.

By default, the Docker daemon listens on the docker.sock file with root-owned permissions. This means that only the root user or users in the docker group have permission to interact with the Docker daemon. This is done for security reasons, as allowing non-privileged users to interact with the Docker daemon can pose a security risk.

However, you can configure the Docker daemon to listen on a network port instead of the docker.sock file. This allows non-root users to interact with the Docker daemon over the network. This approach is less secure than using the docker.sock file, but it can be useful in certain scenarios where you need to interact with the Docker daemon remotely.docker.sock is a Unix socket file used by the Docker daemon to communicate with the Docker client. It is a special file that allows processes to communicate with the Docker daemon without requiring network connectivity.

When the Docker daemon starts, it creates a Unix socket file called docker.sock in the /var/run/docker.sock directory. The Docker client communicates with the daemon by sending requests to this socket file, and the daemon responds by sending data back through the same socket.

By default, the Docker daemon listens on the docker.sock file with root-owned permissions. This means that only the root user or users in the docker group have permission to interact with the Docker daemon. This is done for security reasons, as allowing non-privileged users to interact with the Docker daemon can pose a security risk.

However, you can configure the Docker daemon to listen on a network port instead of the docker.sock file. This allows non-root users to interact with the Docker daemon over the network. This approach is less secure than using the docker.sock file, but it can be useful in certain scenarios where you need to interact with the Docker daemon remotely.

Step 2 - Single Host
Nginx-proxy is now listening to events which Docker raises on start / stop. A sample website called katacoda/docker-http-server has been created which returns the machine name its running on. This allows us to test that our proxy is working as expected. Internally its a PHP and Apache2 application listening on port 80.

Starting Container
For Nginx-proxy to start sending requests to a container you need to specify the VIRTUAL_HOST environment variable. This variable defines the domain where requests will come from and should be handled by the container.

In this scenario we'll set our HOST to match our DEFAULT_HOST so it will accept all requests.

docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server

Testing
Sometimes it takes a few seconds for NGINX to reload but if we execute a request to our proxy using curl http://docker then the request will be handled by our container.

```
$ docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server
ac5d4076531e231f3c643c6a7f060431a6873046ddd6335030caa98118637cee
$ curl http://docker
<h1>This request was processed by host: ac5d4076531e</h1>
$
```

Step 3 - Cluster
We now have successfully created a container to handle our HTTP requests. If we launch a second container with the same VIRTUAL_HOST then nginx-proxy will configure the system in a round-robin load balanced scenario. This means that the first request will go to one container, the second request to a second container and then repeat in a circle. There is no limit to the number of nodes you can have running.

Task
Launch a second container using the same command as we did before.

docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server

Testing

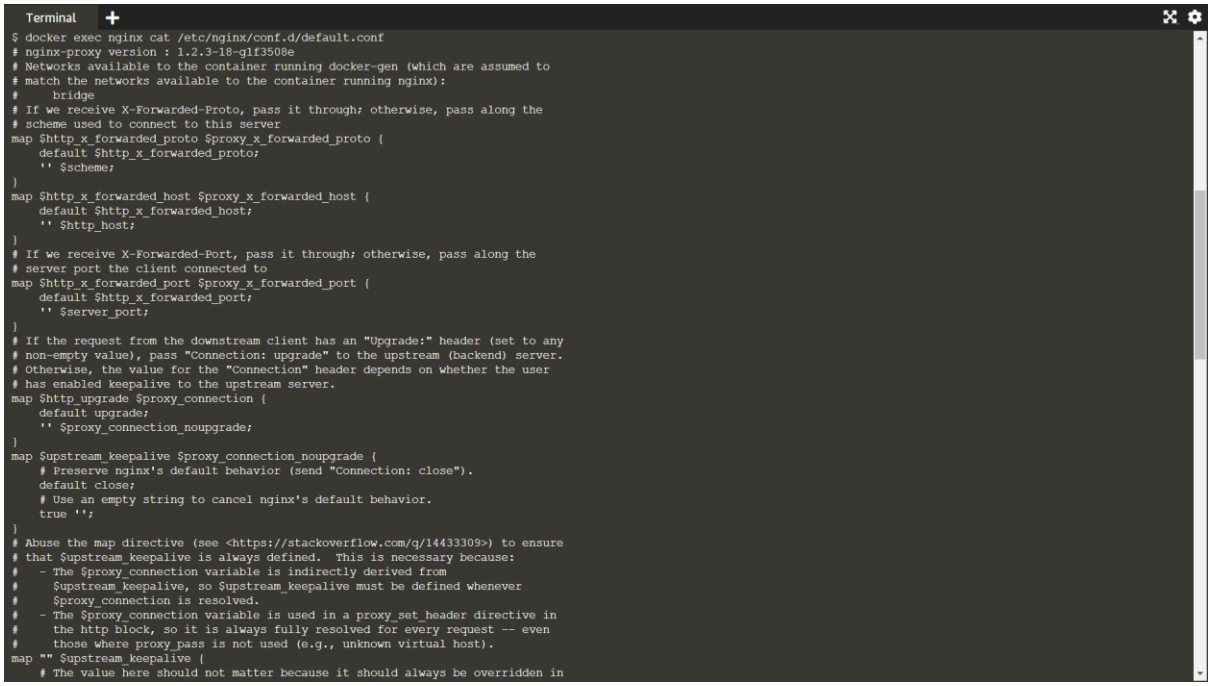
If we execute a request to our proxy using curl http://docker then the request will be handled by our first container. A second HTTP request will return a different machine name meaning it was dealt with by our second container.

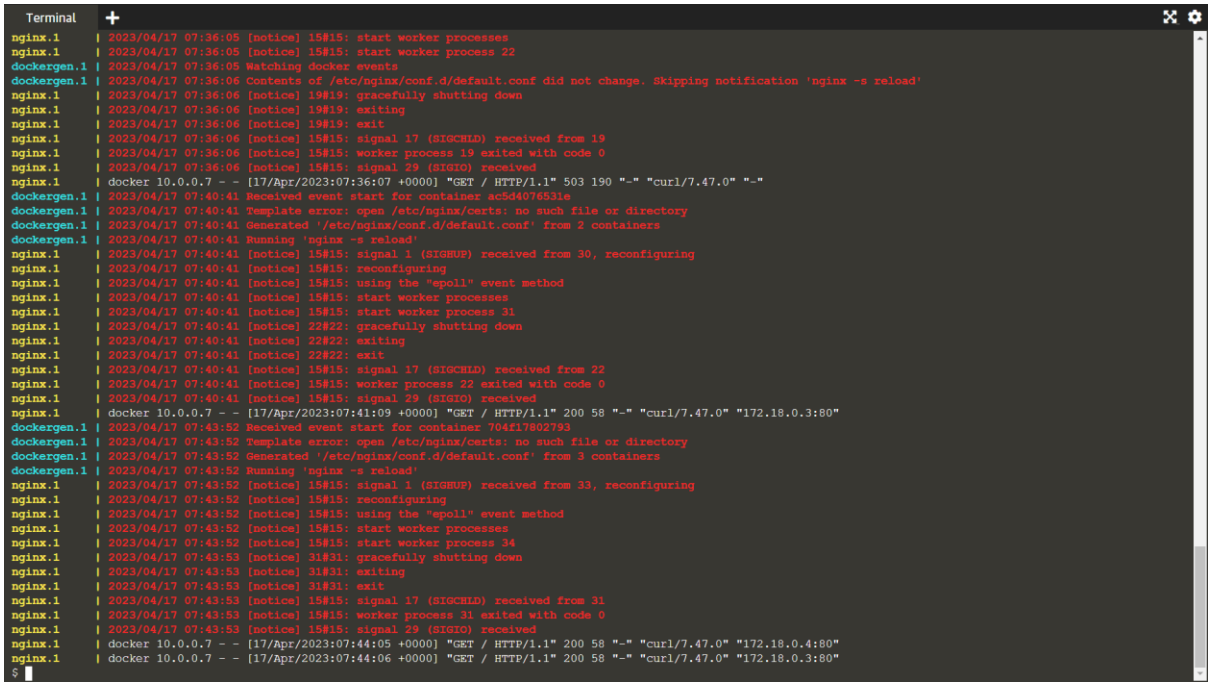
```
$ docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server
ac5d4076531e231f3c643c6a7f060431a6873046ddd6335030caa98118637cee
$ curl http://docker
<h1>This request was processed by host: ac5d4076531e</h1>
$ docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server
704f17802793e427933e69cdad0f858724a864e56c971e89e77b6bf878db4a2d
$ curl http://docker
<h1>This request was processed by host: 704f17802793</h1>
$ curl http://docker
<h1>This request was processed by host: ac5d4076531e</h1>
$
```

Step 4 - Generated NGINX Configuration
While nginx-proxy automatically creates and configures NGINX for us, if you're interested in what the final configuration looks like then you can output the complete config file with docker exec as shown below.

docker exec nginx cat /etc/nginx/conf.d/default.conf

Additional information about when it reloads configuration can be found in the logs using
docker logs nginx





In this scenario we introduced how you can use nginx-proxy to dynamically load balance requests between two containers. The containers to handle each request is discovery using the Docker API and is triggered when new containers are started/stopped.

This pattern allows us to quickly add additional nodes to serve a single website or use the same server to run multiple different websites.