For a single container deployment the approach we have seen is absolutely feasible, but once you repeat this process you will notice the redundancy. **Why should we install and configure NGINX for every single container instance we are running in our environment?** The logical thing to do is to configure this common task of authentication and encryption only once for all deployed containers. This can be achieved by utilizing **docker swarm** (<https://docs.docker.com/engine/swarm/>).

Docker swarm is able to run a collection of containers simultaneously such that they can communicate with each other over a shared virtual network. Docker swarm has a multitude of features which makes it a powerful tool even in large scale deployment, but in this tutorial we will apply a minimal setup to provide secured access for two or more containers running *plumber APIs*. To achieve this we will make use of the **AHUB deployment framework**. AHUB provides a pre-configured swarm setup to deploy analytical containers (based on R or any other language) with minimal effort.

The setup we want to achieve is the following:

Multiple containers running plumber APIs will be accessed via a single **NGINX** instance. The principle of AHUB is to provide a service stack for common tasks like access control & encryption and providing a GUI for manual API execution. The node stack contains multiple containers, all dedicated to a different analytical task. In this example we will run three node stack containers:

* **qunis/ahub\_rnode**: A minimal R plumber configuration with the endpoints /thread and /batch showcasing the use of the *ahubr* package
* **qunis/plumberdemo**: A minimal R plumber configuration with the endpoints /echo, /plot and /sum
* **qunis/prophetdemo**: A more elaborate container running a timeseries forecast with the fantastic *prophet* library and producing an interactive *dyplot*

**Let‘s start!**

**Generating certificates and user credentials**

AHUB comes with a pre-generated certificate and password file. But of course you want to change these. This is very quickly done, with two little helper containers. All you need to do is navigate to the subfolder *./configs* and run the following commands (please fill in your *username* and *password*). This will create a new **SSL certificate** **and key** along with a **.htpasswd** file containing the MD5 hashed credentials for your user in the subfolder *./configs*.

docker run --mount type=bind,src=$pwd,dst=/var qunis/openssl

docker run --mount type=bind,src=$pwd,dst=/var qunis/htpasswd username password

**Configuring the stack**

Docker swarm operates with a recipe, telling it which containers to spin up, which ports to publish, which volumes to mount, et cetera. Everything you would normally  configure in a single „*docker run …*“ statement for a singular container instance, we instead write down in the so called **Compose file** when working with docker swarm.

Please inspect the demo file in the main folder

version: '3.3'

services:

# -------------------------------------------

# NODE STACK (add analytical modules here)

# -------------------------------------------

# For compatibility with AHUB, container images

# need to comply with the following:

# - publish a REST API on port 8000

# - provide a swagger.json file in the "/" path (only for GUI)

# -------------------------------------------

node1:

image: qunis/ahub\_rnode:1.0

# -------------------------------------------

node2:

image: qunis/plumberdemo

# -------------------------------------------

node3:

image: qunis/prophetdemo

# -------------------------------------------

# SERVICE STACK (DO NOT TOUCH)

# -------------------------------------------

nginx:

image: nginx

ports:

- "80:80"

- "443:443"

configs:

- source: nginx\_template.conf

target: /etc/nginx/nginx.conf

secrets:

- source: server.crt

target: server.crt

- source: server.key

target: server.key

- source: htpasswd

target: .htpasswd

deploy:

placement:

constraints: [node.role == manager]

...(continues)...

The **first block** defines the **node stack**. Here you can add as many container images as you like. For compatibility with AHUB it is only required that plumber (or any other API) publishes on port 8000 and provides the Swagger definition file (if you want to use the GUI functionality). The latter is achieved by running the plumber **$run** command with parameter swagger=TRUE.

**IMPORTANT**: If you want to add your own images, you need to make sure, that they are hosted in a container-registry, otherwise docker swarm will not be able to find them. Either you use the public [Docker Hub](https://hub.docker.com/) or set up a private registry with one of the cloud providers ([Google](https://cloud.google.com/container-registry/?hl=de), [Azure](https://azure.microsoft.com/de-de/services/container-registry/) or [AWS](https://aws.amazon.com/de/ecr/)).

The analytical nodes do not have to be R based. A python node running a combination of *flask/flasgger* would be compatible as well.

The **second block** constitutes the **service stack** and does not need to be changed, if you stick to the basic scenario with self-signed certificates and basic authentication. Changes here need to be made if you want to use more elaborate functionality like auto-refreshing *Let‘s Encrypt* certificates or Active Directory Authentication. These use-cases will be covered in future tutorials.

For now you can either leave the demo file as is or add/substitute your own container images in the node stack! Note: There is no need to configure *nginx* when adding containers in the node stack. **This is all taken care of by AHUB.**

**Ramping up the swarm**

Before we launch AHUB we need to prepare the docker daemon to run in swarm mode:

> docker swarm init

Swarm initialized: current node (XXX) is now a manager.

Then the whole stack can be launched by docker in swarm mode with the following command

docker stack deploy -c ./ahub.yaml mystack

This command references the Compose file *ahub.yaml* to deploy a stack called „*mystack*„. Of course you can change the name of your stack to your liking.

You should see the following output on the shell:

> docker stack deploy -c ./ahub.yaml mystack

Creating network mystack\_default

Creating secret mystack\_server.key

Creating secret mystack\_htpasswd

Creating secret mystack\_server.crt

Creating config mystack\_location\_template.conf

Creating config mystack\_nginx\_template.conf

Creating config mystack\_upstream\_template.conf

Creating service mystack\_portainer

Creating service mystack\_node1

Creating service mystack\_node2

Creating service mystack\_node3

Creating service mystack\_nginx

Creating service mystack\_redis

Creating service mystack\_boss

Creating service mystack\_gui

Creating service mystack\_updater

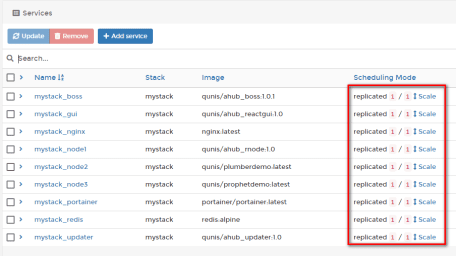
>

Here is a quick intro on the components of the service stack (the code and Dockerfiles for them are located in the subfolder *./modules*):

* **boss**: The central management service. Besides other things, handles the detection of the node stack and fits the nginx configuration to that.
* **nginx**: Our reverse proxy handling authentication, encryption and endpoint routing
* **redis**: NoSQL DB as a process repository: For this minimal setup of no real importance. Comes to play when using the *ahubr* package and activating the advanced logging functionality of AHUB. I will dive into that in a subsequent tutorial.
* **gui**: Provides a very basic GUI for interaction with the API endpoints. You can manually set parameters and view the output of an API call (only JSON output currently supported).
* **updater**: Time triggering of certain activities, like node stack discovery or certificate renewal. Will be covered in future tutorials as well.
* and finally **portainer**…

**Portainer** is a very powerful browser-based container management tool. We can start checking if everything ramped up fine, by navigating to http://localhost::9000 (Portainer listens on this port). As you are starting this cluster for the first time, you need to set an admin account and then choose the **„Local“** mode. After that you get to the Portainer main page, where you can click through the items *Services*, *Containers* and what else piques your interest. With Portainer you can do almost anything you can do from the docker command line interface.

Under the *Services* tab you should see 9 services if you stuck to the demo file. Three of them being the nodestack comprising of *node1*, *node2* and *node3*. Everything is fine when you see a 1/1 behind each service.



**Checking the API endpoints**

You can now navigate to your endpoints via **https://localhost/{nodename}/{endpoint}?{parameters}**. For example <https://localhost/node2/plot> or <https://localhost/node3/?n=24>. You will be warned by your browser about the insecure certificate (because we have self-signed it, skip this warning) and be asked for the user credentials.

There is also a rudimentary GUI at [https://localhost](https://localhost/) (still under development) showing you the various nodes and their endpoints so you can manually trigger a GET request for testing purposes.

Finally if you want to tear down the whole stack, you can do so with the following command

docker stack rm mystack

**Summary**

We were able to ramp up a set of analytical containers providing RESTful APIs by just changing a few lines in a Docker compose file. The AHUB deployment framework takes care of common tasks like **encryption**, **authentication** and a **graphical user interface** to the endpoints. In the next installment of this tutorial series, I will show how to run a stack on a cloud-based virtual machine with public DNS address and retrieve a proper SSL certificate from Let‘s Encrypt.