**Five tips to move your project to Kubernetes**

19 MARCH 2018 on [community](https://blog.alexellis.io/tag/community/), [kubernetes](https://blog.alexellis.io/tag/kubernetes/), [k8s](https://blog.alexellis.io/tag/k8s/), [learn-k8s](https://blog.alexellis.io/tag/learn-k8s/)

Here are five tips to help you move your projects to Kubernetes with learnings from the OpenFaaS community over the past 12 months. The following is compatible with Kubernetes 1.8.x. and is being used with [OpenFaaS - Serverless Functions Made Simple](https://www.openfaas.com/).

Disclaimer: the Kubernetes API is something which changes frequently and you should always refer to the official documentation for the latest information.

**1. Put everything in Docker**

It might sound obvious but the first step is to create a Dockerfile for every component that runs as a separate process. You may have already done this in which case you have a head-start.

If you haven't started this yet then make sure you use multi-stage builds for each component. A multi-stage build makes use of two separate Docker images for the build-time and run-time components of your code. A base image may be the Go SDK for example which is used to build binaries and the final stage will be a minimal Linux user-space like Alpine Linux. We copy the binary over into the final stage, install any packages like CA certificates and then set the entry-point. This means that your final is smaller and won't contain unused packages.

Here's an example of a multi-stage build in Go for the [OpenFaaS](https://www.openfaas.com/) API gateway component. You will also notice some other practices:

* Uses a non-root user for runtime
* Names the build stages such as build
* Specifies the architecture of the build i.e. linux
* Uses specific version tags i.e. 3.6 - if you use latest then it can lead to unexpected behaviour

FROM golang:1.9.4 as build

WORKDIR /go/src/github.com/openfaas/faas/gateway

COPY . .

RUN CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o gateway .

FROM alpine:3.6

RUN addgroup -S app \

&& adduser -S -g app app

WORKDIR /home/app

EXPOSE 8080

ENV http\_proxy ""

ENV https\_proxy ""

COPY --from=build /go/src/github.com/openfaas/faas/gateway/gateway .

COPY assets assets

RUN chown -R app:app ./

USER app

CMD ["./gateway"]

*Note: If you want to use OpenShift (a distribution of Kubernetes) then you need to ensure that all of your Docker images are running as a non-root user.*

**1.1 Get Kubernetes**

You'll need Kubernetes available on your laptop or development machine. Read my blog post on Docker for Mac which covers all the most popular options for working with Kubernetes locally.

<https://blog.alexellis.io/docker-for-mac-with-kubernetes/>

If you've worked with Docker before then you may be used to hearing about *containers*. In Kubernetes terminology you rarely work directly with a container, but with an abstraction called a Pod.

A Pod is a group of one-to-many containers which are scheduled and deployed together and get direct access to each other over the loopback address 127.0.0.1.

An example of where the Pod abstraction becomes useful is where you may have an existing legacy application without TLS/SSL which is deployed in a Pod along with Nginx or another web-server that is configured with TLS. The benefit is that multiple containers can be deployed together to extend functionality without having to make breaking changes.

**2. Create YAML files**

Once you have a set of Dockerfiles and images your next step is to write YAML files in the Kubernetes format which the cluster can read to deploy and maintain your project's configuration.

These are different from Docker Compose files and can be difficult to get right at first. My advice is to find some examples in the documentation or other projects and try to follow the style and approach. The good news it that it does get easier with experience.

Every Docker image should be defined in a *Deployment* which specifies the containers to run and any additional resources it may need. A Deployment will create and maintain a Pod to run your code and if the Pod exits it will be restarted for you.

You will also need a *Service* for each component which you want to access over HTTP or TCP.

It is possible to to have multiple Kubernetes definitions within a single file by separating them with --- and a new line, but prevailing opinion suggests we should spread our definitions over many YAML files - one for each API object in the cluster.

An example may be:

* gateway-svc.yml - for the Service
* gateway-dep.yml - for the Deployment

If all of your files are in the same directory then you can apply all the files in one step with kubectl apply -f ./yaml/ for instance.

When working with additional operating systems or architectures such as the Raspberry Pi - we find it useful to separate those definitions into a new folder such as yaml\_arm.

* Deployment example

Here is a simple example of a Deployment for [NATS Streaming](https://github.com/nats-io/nats-streaming-server) which is a lightweight streaming platform for distributing work:

apiVersion: apps/v1beta1

kind: Deployment

metadata:

name: nats

namespace: openfaas

spec:

replicas: 1

template:

metadata:

labels:

app: nats

spec:

containers:

- name: nats

image: nats-streaming:0.6.0

imagePullPolicy: Always

ports:

- containerPort: 4222

protocol: TCP

- containerPort: 8222

protocol: TCP

command: ["/nats-streaming-server"]

args:

- --store

- memory

- --cluster\_id

- faas-cluster

A deployment can also state how many replicas or instances of the service to create at start-up time.

* Service definition

apiVersion: v1

kind: Service

metadata:

name: nats

namespace: openfaas

labels:

app: nats

spec:

type: ClusterIP

ports:

- port: 4222

protocol: TCP

targetPort: 4222

selector:

app: nats

Services provide a mechanism to balance requests between all the replicas of your Deployments. In the example above we have one replica of NATS Streaming but if we had more they would all have unique IP addresses and tracking those would be problematic. The advantage of using a Service is that it has a stable IP address and DNS entry which can be used to access one of the replicas at any time.

Services are not directly mapped to Deployments, but are mapped to labels. In the example above the Service is looking for a label of app=nats. Labels can be added or removed from Deployments (and other API objects) at runtime making it easy to redirect traffic in your cluster. This can help enable A/B testing or rolling deployments.

The best way to learn about the Kubernetes-specific YAML format is to look up an API object in the documentation where you will find examples that can be used with YAML or via kubectl.

Find out more about the various API objects here:

<https://kubernetes.io/docs/concepts/>

**2.1 helm**

[Helm](https://helm.sh/) describes itself as a package manager for Kubernetes. From my perspective it has two primary functions:

* To distribute your application (in a Chart)

Once you are ready to distribute your project's YAML files you can bundle them up and submit them to the Helm repository so that other people can find your application and install it with a single command. Charts can also be versioned and can specify dependencies on other Charts.

Here are three example charts: [OpenFaaS](https://github.com/openfaas/faas-netes/tree/master/chart/openfaas), [Kakfa](https://github.com/kubernetes/charts/tree/master/incubator/kafka) or [Minio](https://github.com/kubernetes/charts/tree/master/stable/minio).

* To make editing easier

Helm supports in-line templates written in Go, which means you can move common configuration into a single file. So if you have released a new set of Docker images and need to perform some updates - you only have to do that in one place. You can also write conditional statements so that flags can be used with the helm command to turn on different features at deployment time.

This is how we define a Docker image version using regular YAML:

**image**: functions/**gateway**:0.7.5

With Helm's templates we can do this:

**image**: {{ .Values.images.gateway }}

Then in a separate file we can define the value for "images.gateway". The other thing helm allows us to do is to use conditional statements - this is useful when supporting multiple architectures or features.

This example shows how to apply either a ClusterIP or a NodePort which are two different options for exposing a service in a cluster. A NodePort exposes the service outside of the cluster so you may want to control when that happens with a flag.

If we were using regular YAML files then that would have meant maintaining two sets of configuration files.

spec:

type: {{ .Values.serviceType }}

ports:

- port: 8080

protocol: TCP

targetPort: 8080

{{- if contains "NodePort" .Values.serviceType }}

nodePort: 31112

{{- end }}

In the example "serviceType" refers to ClusterIP or NodePort and then we have a second conditional statement which conditionally applies a nodePort element to the YAML.

**3. Make use of ConfigMaps**

In Kubernetes you can mount your configuration files into the cluster as a ConfigMap. ConfigMaps are better than "bind-mounting" because the configuration data is replicated across the cluster making it more robust. When data is bind-mounted from a host then it has to be deployed onto that host ahead of time and synchronised. Both options are much better than building config files directly into a Docker image since they are much easier to update.

A ConfigMap can be created ad-hoc via the kubectl tool or through a YAML file. Once the ConfigMap is created in the cluster it can then be attached or mounted into a container/Pod.

Here's an example of how to define a ConfigMap for Prometheus:

**kind**: ConfigMap

**apiVersion**: v1

**metadata**:

**labels**:

**app**: prometheus

**name**: prometheus-config

**namespace**: openfaas

**data**:

prometheus.**yml**: |

**scrape\_configs**:

- **job\_name**: 'prometheus'

**scrape\_interval**: 5s

**static\_configs**:

- **targets**: ['localhost:9090']

You can then attach it to a Deployment or Pod:

volumeMounts:

- mountPath: /etc/prometheus/prometheus.yml

name: prometheus-config

subPath: prometheus.yml

volumes:

- name: prometheus-config

configMap:

name: prometheus-config

items:

- key: prometheus.yml

path: prometheus.yml

mode: 0644

See the full example here: [ConfigMap](https://github.com/openfaas/faas-netes/blob/master/yaml/alertmanager_config.yml)  
[Prometheus config](https://github.com/openfaas/faas-netes/blob/master/yaml/prometheus.yml)

Read more in the docs: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>

**4. Use secure secrets**

In order to keep your passwords, API keys and tokens safe you should make use of Kubernetes' secrets management mechanisms.

If you're already making use of ConfigMaps then the good news it that secrets work in almost exactly the same way:

* Define the secret in the cluster
* Attach the secret to a Deployment/Pod via a mount

The other type of secrets you need to use is when you want to pull an image in from a private Docker image repository. This is called an ImagePullSecret and you can find out [more here](https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/).

You can read more about how to create and manage secrets in the Kubernetes docs: <https://kubernetes.io/docs/concepts/configuration/secret/>

**5. Implement health-checks**

Kubernetes supports health-checks in the form of liveness and readiness checking. We need these mechanisms to make our cluster self-healing and resilient to failure. They work through a probe which either runs a command within the Pod or calls into a pre-defined HTTP endpoint.

* Liveness

A liveness check can show whether application is running. With OpenFaaS functions we create a lock file of /tmp/.lock when the function starts. If we detect an unhealthy state we can remove this file and Kubernetes will re-schedule the function for us.

Another common pattern is to add a new HTTP route like /\_/healthz. The route of /\_/ is used by convention because it is unlikely to clash with existing routes for your project.

* Readiness checks

If you enable a readiness check then Kubernetes will only send traffic to containers once that criteria has passed.

A readiness check can be set to run on a periodic basis and is different from a health-check. A container could be healthy but under too much load - in which case it could report as "not ready" and Kubernetes would stop sending traffic until resolved.

You can read more in the docs: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/>

**Wrapping up**

In this article, we've listed some of the key things to do when bringing a project over to Kubernetes. These include:

* Creating good Docker images
* Writing good Kubernetes manifests (YAML files)
* Using ConfigMaps to decouple tunable settings from your code
* Using Secrets to protect sensitive data such as API keys
* Using liveness and readiness probes to implement resiliency and self-healing

For further reading I'm including a comparison of Docker Swarm and Kubernetes and a guide for setting up a cluster fast.

* [Kubernetes vs Docker/Swarm](https://blog.alexellis.io/you-need-to-know-kubernetes-and-swarm/)

Compare Kubernetes with Docker and Swarm and get a good overview of the tooling from the CLI to networking to the component parts

* [You instant Kubernetes cluster](https://blog.alexellis.io/your-instant-kubernetes-cluster/)

If you want to get up and running with Kubernetes on a regular VM or cloud host - this is probably the quickest way to get a development cluster up and running.