A practical introduction to containerization and container orchestration with Kubernetes

Gustaf Gunér (gusgun@kth.se)
Moa Nyman (moanym@kth.se)

This guide will introduce you to containerization with Docker and orchestration with Kubernetes by deploying a small app. The three parts of the guide are 1) running the app locally to understand the process, 2) Familiarizing ourselves with Docker by containerizing the app and 3) Using Kubernetes to manage our containers. Let's begin!

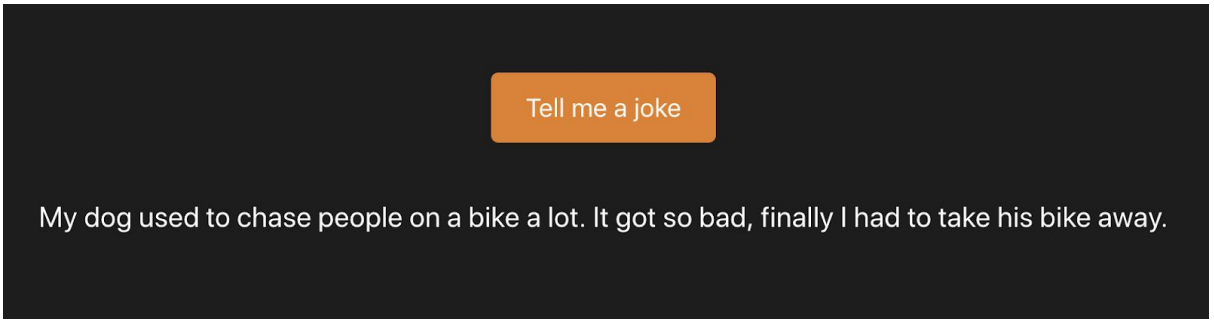
Part 1: Running the app

The app we are going to demonstrate is very basic, its only purpose is to show that the frontend can communicate with the backend. It will consist of two services, one for the frontend and one for the backend. The frontend will be built in React and the backend in Node.js. These two services will be able to communicate with each other over HTTP.

The Node.js app will serve an http server exposing one single GET endpoint `/joke`. This endpoint will respond with a random joke in the following JSON format.

```
{"joke":"A funny joke"}
```

Our React app will consist of a single button titled "Tell me a joke". When clicking this button, a GET request to our backend endpoint `/joke` will be performed. The response of this request, i.e. the joke, will appear beneath the button.



You can find the project here.

To understand the containerization process we must understand the process of running the app in the usual way, i.e. locally on your machine. Let's lay out the steps we would take, starting with the frontend. Both of our services rely on Node.js/NPM, so make sure that you have installed Node.js before proceeding with this guide. If you're unsure how to install it, visit their site: https://nodejs.org. Start by cloning the repo:

```
git clone https://github.com/gustafguner/k8s-demo.git
```

Running the frontend

Step 1: Change directory to `frontend` with the command `cd frontend`.
Step 2: Run `npm install` to install all of the dependencies needed to run the app.
Step 3: Once the installation is complete, we want to run the command `npm run build`. This command builds a production ready version of the React project and generates static files for us to serve.
Step 4: Finally, for us to serve the static files we need some kind of web server. Our weapon of choice will be nginx. We install nginx and and start it up. Nginx is defined to serve files in the directory `/usr/share/nginx/html`. So, to serve our website we simply copy all of our files from our `build` folder (the files that `npm run build` generated) to `/usr/share/nginx/html`.

If we enter `http://localhost:80` (or simply `http://localhost`) in the address bar of our favourite web browser we should be able to see the React app in action. In only four steps we have the frontend up and running on nginx's default port 80. The backend process is even more concise.

Running the backend

Step 1: Change directory to `backend` with `cd ../backend` (assuming your working directory is `frontend`).
Step 2: Install all of the dependencies by running `npm install`.
Step 3: Start the web server by running `npm start`.

You should be prompted with `Server is listening on port 4000` which means that the Node.js server is up and running and that everything is working. If you visit `http://localhost:4000/joke` in your web browser you should see something like:

```
{"joke":"I'm so good at sleeping. I can do it with my eyes closed."}
```

Running the frontend and the backend together

While running the Node.js server and nginx server at the same time, you should now be able to enter `http://localhost`, click the button “Tell me a joke” and a joke should appear beneath the button.

Part 2: Containerizing the app

Okay, we have gotten our frontend and backend up and running locally working perfectly fine. But now, let’s start talking about perhaps the most central concept of guide – Containers!

What is a container? A container is a piece of software that is used to bundle your code with all its dependencies so that is can be run reliably in any environment. It is a way for developers to be a hundred percent sure that if the code works on my machine, it will work on yours too. Containerization is a way to isolate software so that it will run predictably everytime. A benefit of this is less time spent on debugging different environments.

In many ways, containerizing software is similar to running it on a virtual machine, the difference being that a container does not require its own copy of the OS. Instead, the container shares the OS kernel, making startup and teardown faster and cheaper, and requires less memory. As a result, containers are very lightweight compared to VMs.

To get all these benefits we will containerize our app using the most popular containerization tool Docker. To do so, we will remember the steps of starting the apps that we laid out in part one, because our so called *Dockerfiles* will more or less recreate those steps exactly.

Dockerfile

A Dockerfile is a set of instructions on how a container image should be built. At the top of the file is a base *container image* that our container depends on. The base container image is declared using the `FROM` keyword. After the base image, you simply describe what commands should be run in order to build our container image.

To avoid slow build times, you can add another file called `.dockerignore`. This file is very similar to `.gitignore` if you're familiar with it. It lists all files that Docker should ignore in the build process.

Containerizing the backend

We will start the containerization by containerizing the backend. The reason we start with the backend is because it's a bit easier than the frontend. This is how the Dockerfile for the backend will look like:

```
FROM node:latest
WORKDIR /app
COPY . ./
RUN npm install
EXPOSE 4000
CMD npm start
```

We start by defining our base container image with `FROM`, which in this case is `node:latest`. `latest` means that it will use the most recent version of node automatically. If you wish, you could specify a version number instead, but for this example it isn't necessary. We then specify our working directory with `WORKDIR` which we call `/app` (setting a workdir makes all subsequent `ADD`, `COPY`, `CMD`, `ENTRYPOINT` and RUN instructions use this directory). We proceed by copying all files to our working directory with `COPY . ./`. We then execute the command `npm install` (to install our dependencies as we did before). We end the file be exposing the port `4000` (which is the same one as the one we used within backend) before running the command `npm start`. To our `.dockerignore` file we add one single line: `node_modules`. To build the image we run:

```
docker build -t guide-backend .
```

Containerizing the frontend

Containerizing the frontend is a bit more tricky. In this case, the `Dockerfile` will have to be divided into two stages. The first stage will consist of generating the production ready static files via `npm run build` and the second one will consist of copying these files to our magic nginx directory `/usr/share/nginx/html`. The reason the Dockerfile be divided into two phases is because the two phases will have different base container images. That's

right, we will use the `FROM` keyword twice in the `Dockerfile`! The first part will use `node` (for running the `npm` commands) and the second one will use nginx (for serving the static files generated in phase one). Now, you may ask yourself: which base container image will the final container actually use? The answer is nginx. The first one, node, will only be used temporarily within the build process. In the end we're only interested in the static files. The Dockerfile will look like this:

```
# Stage 1
FROM node:latest as react-build
WORKDIR /app
COPY . ./
RUN npm install
RUN npm run build

# Stage 2
FROM nginx:latest
COPY --from=react-build /app/build /usr/share/nginx/html
```

Stage 1 is very similar to the `Dockerfile` in backend. The only difference is that we name our first stage `react-build` with `as react-build` and that we end with running `npm run build` (to generate our static files). In stage 2 we only do two things. We start by saying that we're using the latest version of `nginx` as our base container image. Then, we copy the folder `/app/build` from our stage `react-build` to the folder `/usr/share/nginx/html`. To build the image we run:

```
docker build -t guide-frontend .
```

Making sure that the images work

Now, let's try to run both our built images to make sure that everything still works. Let's start by running the backend. Open a terminal window and run

```
docker run -p 4000:4000 guide-backend:latest
```

You should now be able to visit `http://localhost:4000/joke` in your browser as before. Let's run the frontend image too. Open another terminal window and run

```
docker run -p 80:80 guide-frontend:latest
```

Enter `http://localhost` in your web browser and the app should work as intended!

Pushing the images to Docker Hub

The last part of our Docker journey is to push the images we built to Docker Hub. Docker Hub is a container image library that stores hundreds of thousands of container images that anyone can use.

Step 1: Create an account on https://www.docker.com/.
Step 2: Install the Docker Community Edition: https://www.docker.com/products/container-runtime.
Step 3: Open a terminal window and login to docker by running `docker login -u="$USERNAME" -p="$PASSWORD"` (replace $USERNAME and $PASSWORD with your Docker username and password).
Step 4: Re-build both of your images with the commands `docker build -t $USER_ID/guide-backend .` and `docker build -t $USER_ID/guide-frontend .` (replace $USER_ID with your Docker user ID).
Step 5: Push both of the images to Docker Hub by running `docker push $USER_ID/guide-backend` followed by `docker push $USER_ID/guide-frontend`.

Voilá! Both of our container images are now located on Docker Hub ready for anyone to use!

Part 3: Orchestrate the containers

As the number of containers in a project grows, it becomes a hassle to handle deployment, scaling, networking, and everything else container related. Luckily, there are a handful of container orchestration tools available to manage the container lifecycle so that you can spend time of other things.

In general, orchestration tools work by specifying where to get your container images and how you want your containers to act and interact. The most widely used such tool is Google's Kubernetes, aka k8s. Based on Borg, Google's in-house orchestration tool, 15 years in the making, Kubernetes offers a mature set of features including but not limited to auto-scaling, self-healing, load-balancing and automated rollouts and rollbacks.

How does Kubernetes work?

Kubernetes consists of three primary units:

- **Pods** – the smallest unit. In general a pod can be viewed as the equivalent of a container. It's possible for a pod to host more than one container if they share some resources. Within the cluster each pod has its own IP address. The IP can change and is thus not reliable.
- **Node** – A node in Kubernetes is a worker machine, either a virtual or physical machine. A node is in charge of running pods. A master node controls the worker nodes. Each node has at the least a Kuberlet which is a process that handles the communication between the master node and itself, as well as a container runtime that pulls container images, for example from the Docker Hub, and runs the application.
- **API Server** – the API server is our way of communicating with Kubernetes, for example by asking for more containers of an image. It is housed in the master node.

When Kubernetes receives a request the master server will delegate the task to one of its node with a low load. Kubernetes can create new nodes if there's a need and issues new containers in either of them. For the record, Kubernetes runs on all of the big three of cloud computing – Amazon Web Services, Microsoft Azure and Google Cloud Platform, as well on traditional servers, preventing provider lock in.

To run our apps locally we will use Minikube. Minikube runs a single node cluster inside a local virtual machine, allowing us to easily test our setup. What we setup on the Minikube will also work in a production environment. Minikube is however not indented to be used in a production environment.

After installing Minikube start it with `minikube start`. If you wish you can now check out your cluster by running `kubectl get nodes`.

Configuring the frontend

We'll define our setup starting with the configuration files for the frontend. A configuration file in Kubernetes has several property options. The properties we will use are `apiVersion`, `kind`, `metadata` and `spec`, as well as some properties within those. `apiVersion` refers to the version of Kubernetes API. "v1" is the first stable release. There are other versions, for

example a beta version with more experimental features. `Kind` refers to what type of object we are dealing with, for example Pod, Service, Event etc. `Metadata` specifies details about the object itself. `Spec` defines what state we want our object to be in. It is within spec where we define what containers the object should host.

Pods

We will use the smallest Kubernetes component to house our frontend containers. Our pod file will look like this:

```
apiVersion: v1
kind: Pod
metadata:
     name: frontend
spec:
     containers:
          - image: $DOCKER_USER_ID/guide-frontend
          name: frontend
          ports:
               - containerPort: 80
```

Here we can see the container within spec, and that it is pointing to the image we created earlier. We have also given it a name and declared what port the container is listening to. Replace `$DOCKER_USER_ID` with your Docker user ID so that you fetch the correct container image.

Create the pod by running:
`kubectl create -f frontend-pod.yaml`
Verify that its running:
`kubectl get pods`

Our pod should be up and running, however, it is not yet obvious how we will connect the pod with the rest of our app. This is because we are missing a **Service**, an entry point to the pods.

Service

As we previously mentioned, pods have unreliable IP addresses and because of that we cannot communicate with the pods directly. In contrast a service has a static IP which we can target to in turn communicate with the pods. A service is able to discover pods by looking at their labels.

The service will delegate the requests it receives to the pods in has been assigned. A service in Kubernetes is defined in much the same way as a pod is. Many of the properties you will recognize from the pod definition. What is new for the service is that the `spec` property does not contain container images. Instead it declares a `type` (in our case a load balancer), what protocols to use in the port and which selector to target. We will use this definition:

```
apiVersion: v1
kind: Service
metadata:
      name: frontend-service
spec:
      selector:
            app: frontend
      ports:
          - protocol: TCP
            targetPort: 80
            port: 80
            name: http
      type: LoadBalancer
```

The perhaps most important part of the definition is the `selector` property. This defines what pods the Service should target. The service will look for pods that have a label matching its selector. Currently though, our pod does not have any labels. Let's fix that.

Add this to the pod definition, nested in the metadata tag.

```
labels:
      app: frontend
```

Re-create the pods:

`kubectl create -f frontend-pod.yaml`

Now create the service:

`kubectl create -f frontend-service.yaml`

Verify that its up:

`kubectl get svc`

Now the service can find the pods, and delegate tasks to them.

Scaling up

Let's imagine for a moment that our app becomes a huge success and has millions of visitors every day. Currently, however, we only have one pod. No way that will be enough to handle all those requests! To solve this problem, we could add more pod definitions and label them with `app: frontend`, but this would mean manually duplicating our first pod definition to scale our app, and that would be bad for everyone. As you might expect Kubernetes has a built in solution for this, so called **Deployments**. So let's create a Deployment definition!

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
      name: frontend-deployment
spec:
      replicas: 2
      strategy:
            type: RollingUpdate
            rollingUpdate:
                  maxUnavailable: 1
                  maxSurge: 1
      template:
            metadata:
                  labels:
                        app: frontend
            spec:
                  containers:
                        - image: $DOCKER_USER_ID/guide-frontend
                        imagePullPolicy: Always
                        name: frontend-container
                        ports:
                              - containerPort: 80
```

```
name: http
```

With the deployment object we can delegate pod creation to avoid having to manually add pods to meet our needs. As you can see, we have moved almost everything from our pod definition to the template property of the deployment definition. Besides that, we have also added the `replicas` field and the the `strategy` field. We will still probably not be able to handle millions of requests, but for our expected popularity it will probably suffice.

The `replicas` field defines how many pods from the `template` should be created upon deployment. The `strategy` field defines the strategy to choose when deploying changes. We have chosen the type to be `RollingUpdate`, which means that the app will still be reachable during an update. The alternative, `Recreate`, kills all existing nodes before deploying meaning the app would be down until the new pods are created. Along with our update strategy type goes some rollingUpdate properties: `maxUnavailable` and `maxSurge`. `maxUnavailable` is how many nodes can be down during an update, in our case 1, so the app will continue to be reachable during the update. With `maxSurge` we tell Kubernetes that during an update one extra node can be created if necessary to complete the update. *Also note that we changed the API version to the beta version.*

The deployment definition actually replaces the pod definition we created in the beginning. Let's create our new deployment object and stop our currently running pods.

Instead of `create` we will use the `apply` command on deployments:
`kubectl apply -f frontend-deployment.yaml`

Verify that the pods were created:
`kubectl get pods`

Stop the previously created pods:
`kubectl delete pod [pod-name]`

Now we have our more flexible pods running and our service to manage them.

Configuring the backend

We still have the backend left, but that process is pretty much the same as for the frontend. We want a load balancer and we want to enable rolling updates. Of course we can skip the step of first creating pod files. The only difference is that we will use our backend containers!

Backend service definition:

```
apiVersion: v1
kind: Service
metadata:
      name: backend-service
spec:
      selector:
            app: backend
      ports:
          - protocol: TCP
            targetPort: 80
            port: 80
            name: http
      type: LoadBalancer
```

Backend Deployment defintion

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
      name: backend-deployment
spec:
      replicas: 2
      strategy:
            type: RollingUpdate
            rollingUpdate:
                  maxUnavailable: 1
                  maxSurge: 1
      template:
            metadata:
                  labels:
                        app: backend
            spec:
                  containers:
```

```
- image: $DOCKER_USER_ID/guide-backend
imagePullPolicy: Always
name: backend-container
ports:
      - containerPort: 80
        name: http
```

Create the service

`kubectl create -f backend-service.yaml`

Apply the deployment

`kubectl apply -f backend-deployment.yaml`

To verify that our app is working, we can run `minikube service frontend-service`. Try clicking the “Tell me a joke” button and a joke should appear!

That’s it! You got your app running like a pro. So what have we learned?

- We’ve learned adapt the classic workflow of how to run webapps into Dockerfiles
- We’ve learned to create Kubernetes pods to manage our containers, and Kubernetes services to manage our pods locally using Minikube
- We’ve learned how to automatically scale Kubernetes pods by using Kubernetes Deployments

I hope you enjoyed this guide and learned something from it! If you want to continue from here, you could try adding probes to your pods. Probes are used to determine when pods are ready to receive traffic or when to restart a pod. For more advanced topics I suggest you check out the Kubernetes Documentation.

Sources

https://kubernetes.io/docs/home/

https://docs.docker.com

https://hub.docker.com/

https://blog.newrelic.com/engineering/container-orchestration-explained/

https://medium.freecodecamp.org/learn-kubernetes-in-under-3-hours-a-detailed-guide-to-orchestrating-containers-114ff420e882