Introduction

The project files are located in the `7-kubernetes` folder.

Kubernetes is probably the most frequently used orchestrator platform. It is popular for a reason. First of all, it has autoscaling. It can scale not just containers but also nodes. You can define rules, such as: "I want the API to run at least 4 replicas but if the traffic is high let's scale up to a maximum of 8" and you can do the same with your actual servers. So it's quite powerful.

It has every feature we discussed in the Docker Swarm chapter. It can apply resource limits and requests to containers as well which is a pretty nice optimization feature. It has great auto-healing properties.

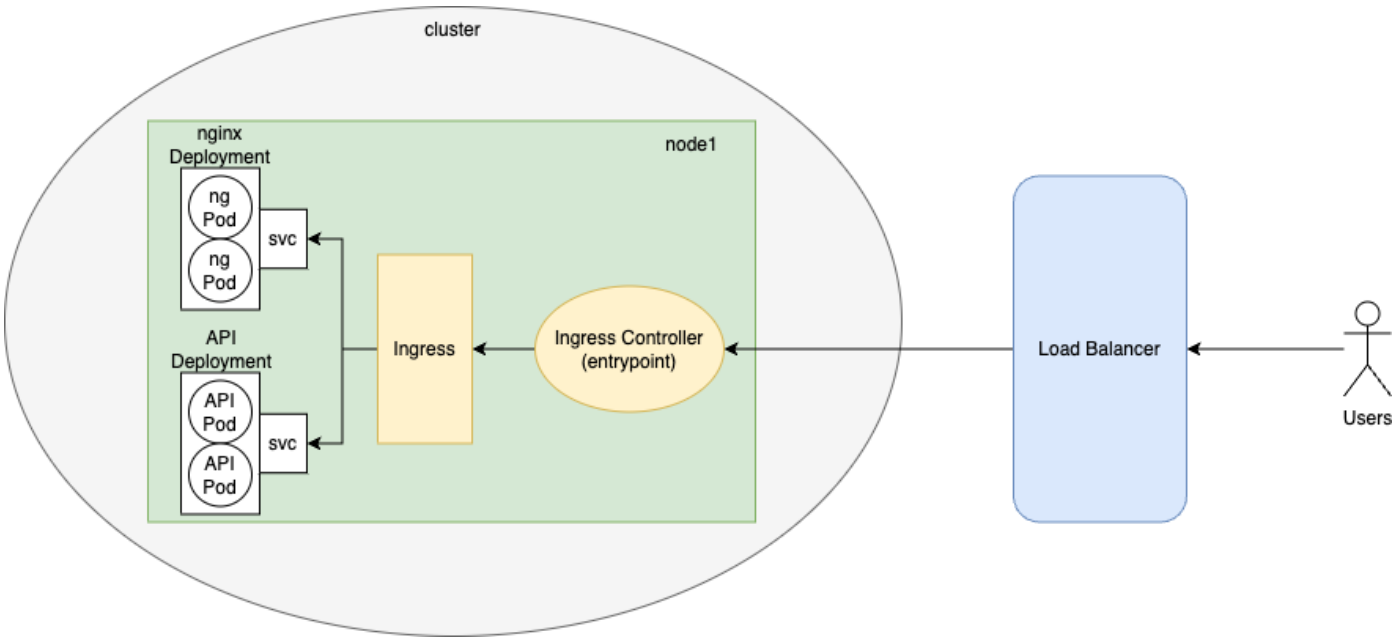
So when you use Kubernetes it almost feels like you have a new Ops guy on your project. And in fact, it's not that hard to learn.

Basic concepts

These are the basic terms:

- **Node** is a server with Kubernetes installed on it.
- **Cluster** is a set of nodes. Usually, one project runs on one cluster with multiple nodes.
- **Node pool** is a set of nodes with similar properties. Each cluster can have multiple node pools. For example, your cluster can have a "standard" pool and a "gpu" pool where in the gpu pool you have servers with powerful GPUs. Pools are mainly used for scalability.
- **Pod** is the smallest and simplest unit in Kubernetes. It's basically a container that runs on a given node. Technically, a pod can run multiple containers. We'll see some practical examples of that, but 99% of the time a pod === 1 container. In the sample application, each component (such API, frontend) will have a dedicated pod.
- **ReplicaSet** ensures that a specified number of replicas of a Pod are always running. In Docker Swarm, it was just a configuration in the `docker-compose.yml` file. In Kubernetes, it's a separate object.
- **Deployment:** everyone wants to run pods. And everyone wants to scale them. So Kubernetes provides us with a deployment object. It merges together pods and replica sets. In practice, we're not going to write pods and replica sets but deployments. They define how applications are managed and scaled in the cluster. A deployment is a higher-level abstraction that manages Pods and provides features like scaling, rolling updates, and lifecycle management. It ensures that the desired number of Pods are always running and can be easily scaled up or down.
- **Services** provide network access to a set of Pods. They expose a given port of a set of pods and load balance the incoming traffic among them. Unfortunately, exposing a container (pod) to other containers is a bit more difficult than in docker-compose for example.
- **Ingress** and **ingress controller** are components that help us expose the application to the world and load balance the incoming traffic among the nodes.
- **Namespaces**: Kubernetes supports multiple virtual clusters within a physical cluster using namespaces. Namespaces provide isolation and allow different teams or applications to run independently without interference. Resources like Pods, Deployments, and Services can be grouped and managed within specific namespaces.

This is what it looks like:



This figure shows only one node, but of course, in reality, we have many nodes in a cluster. Right now, you don't have to worry about the ingress components or the load balancer. The important thing is that there are deployments for each of our services, they control the pods, and there are services (svc on the image) to expose ports.

We'll have a load balancer (a dedicated server with the only responsibility of distributing traffic across the nodes), and the entry point of the cluster is this ingress controller thing. It forwards the request to a component called ingress, which acts like a reverse proxy and calls the appropriate service. Each service acts like a load balancer and they distribute the incoming requests among pods.

Pods can communicate with each other, just like containers in a docker-compose config. For example, nginx will forward the requests to the API pods (they run php-fpm).

I know it sounds crazy, so let's demystify it! First, a little bit of "theory" and then we start building stuff.

Pod

The smallest (and probably the most important) unit in Kubernetes is a Pod. It usually runs a single container and contains one component (such as the API) of your application. When we talk about autoscaling we think about pods. Pods can be scaled up and down based on some criteria.

A pod is an object. There are other objects such as services, deployments, replica sets, etc. In Kubernetes, we don't have a single configuration file such as `docker-compose.yml` but many small(er) config files. Each object is defined in a separate file (usually, but they can be combined).

This is the configuration of a pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: api
spec:
  containers:
    - name: api
      image: martinjoo/posts-api:latest
      ports:
        - containerPort: 9000
```

As you can see, the `kind` attribute is set to `Pod`. Every configuration file has a `kind` key that defines the object we're about to create.

In `metadata` you can name your object with a value you like. It is going to be used in CLI commands, for example, if you list your pods, you're going to see `api` in the name column.

The `spec` key defines the pod itself. As I said, a pod can run multiple containers this is why the key is called `containers` not `container`. We can define the containers in an array. In this case, I want to run the `martinjoo/posts-api:latest` image and name the container `api`. `containerPort` is used to specify the port number on which the container listens for incoming traffic. We're going to talk about ports later. Right now, you don't have to fully understand it, it's just an example.

So this is a pod configuration. The smallest unit in k8s and it's not that complicated, actually.

ReplicaSet

A ReplicaSet is an object that manages pods. It defines how many replicas should be running from a given pod.

This is what the configuration looks like:

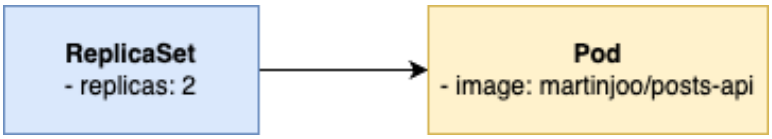
```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
```

It's not the whole configuration yet, but it shows you the most important things. It defines two things.

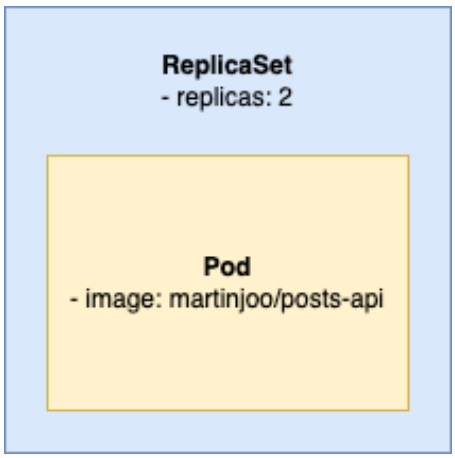
What do we want to run? It's defined in the `selector` key. This ReplicaSet is responsible for pods labeled as `api`. This is described in the `matchLabels` key.

How many replicas do we want to run? It's defined in the `replicas` key.

So this ReplicaSet runs two replicas of the `api` pod. But a replica set is an abstraction above pods. So it doesn't work this way:



So we don't need to define two separate objects. We can merge them together in one ReplicaSet config so the result is more like this:



This means we need to define the pod inside the ReplicaSet config:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: martinjoo/posts-api:latest
          ports:
            - containerPort: 9000
```

And this points out one of the most annoying things about Kubernetes config files: all of these labels and selectors.

This part:

```yaml
selector:
  matchLabels:
    app: api
```

defines that the **ReplicaSet** is responsible for pods labeled as `api`.

And this part:

```yaml
metadata:
  labels:
    app: api
```

labels the pod as `api`.

I know it looks weird in one small YAML config, but when we apply (run) these configs the ReplicaSet doesn't know much about YAML files. The Pod doesn't know much about them either. The ReplicaSet needs to take care of the `api` pods on hundreds of servers. The Pod needs to label itself as `api` so the ReplicaSet can find it.

Every config file contains this label and selector mechanism so you better get used to it.

Finally, here's a picture that might help you to understand the relationship between a Pod and a ReplicaSet:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: martinjoo/posts-api:latest
          ports:
            - containerPort: 9000
```

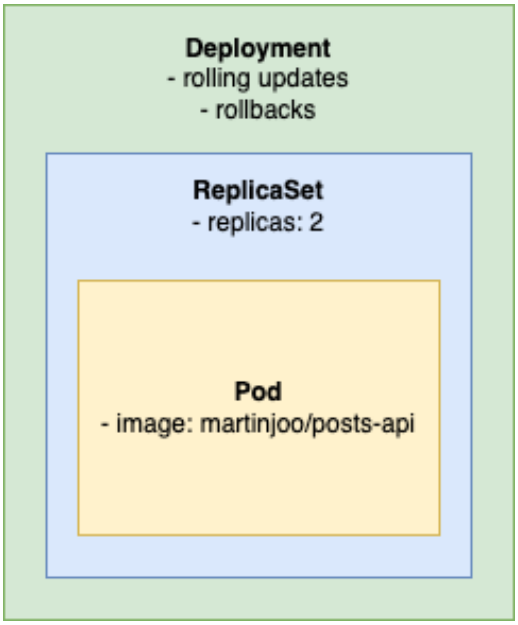
The blue part is the definition of the ReplicaSet (what pod to run and in how many replicas). And the green part is the pod's definition (which image you want to run). And the black part is basically just gibberish for k8s.

Deployment

Now that you have read a few pages about Pods and ReplicaSets I have good and bad news:

- The bad news is that we won't use them. Not a single time.
- The good news is that everything you learned applies to deployments.

It looks like this:



Deployment is another layer of abstraction on top of a ReplicaSet. It can define things such as rolling updates and rollback configs. This means:

- We won't write Pods and ReplicaSets but only Deployments.
- When deploying or rolling back we don't interact directly with Pods or ReplicaSets but only Deployments.

And fortunately, the YAML of a Deployment looks exactly the same as a ReplicaSet:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  template:
    metadata:
```

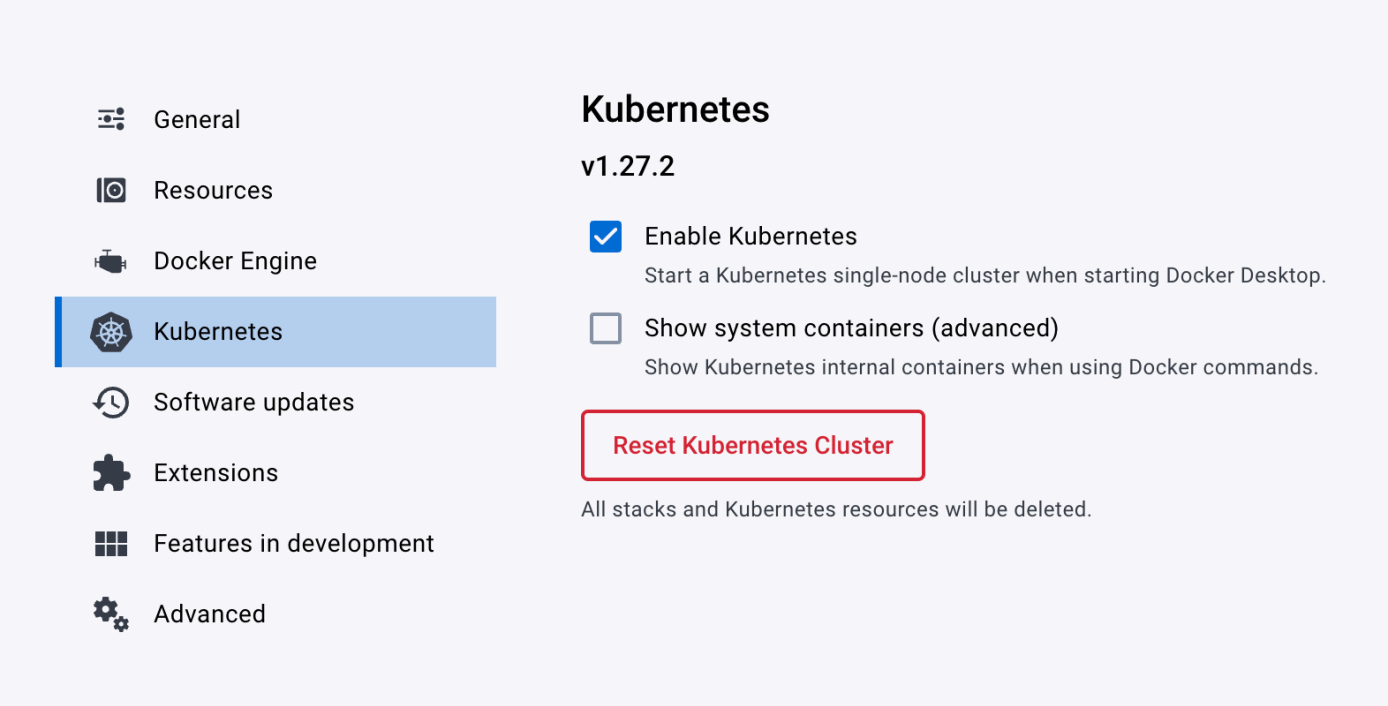
```
    labels:
      app: api
  spec:
    containers:
      - name: api
        image: martinjoo/posts-api:latest
        ports:
          - containerPort: 9000
```

I just modified `kind` to `Deployment`. Of course, a deployment can define extra keys (such as a rolling update strategy) but it's not important right now.

Creating a cluster

I don't advise you to create a self-hosted Kubernetes cluster at all. The setup is quite complicated and you need to maintain it as well. The good news is that every cloud provider offers k8s solutions and most of them are free. It means that you don't have to pay extra money because you're using a k8s cluster. You just need to pay for the actual servers you're using. Just as if you rented a standard VPS, I'm going to use DigitalOcean.

The command line tool to interact with a cluster is `kubectl`. If you are using Docker Desktop you can install it with one click. Just turn on the `Enable Kubernetes` option in the settings:



This will install `kubectl` on your machine and it also starts a single-node cluster on startup. We're not going to use this cluster in this book. I think it's much better to just create a cluster in the cloud with the cheapest nodes and delete it after you don't need it. If you create a 2-node cluster the monthly cost is going to be $24. If you only use it for 5 day (for testing purposes) your cost will be around $4.

After you enabled k8s run this to verify it's working:

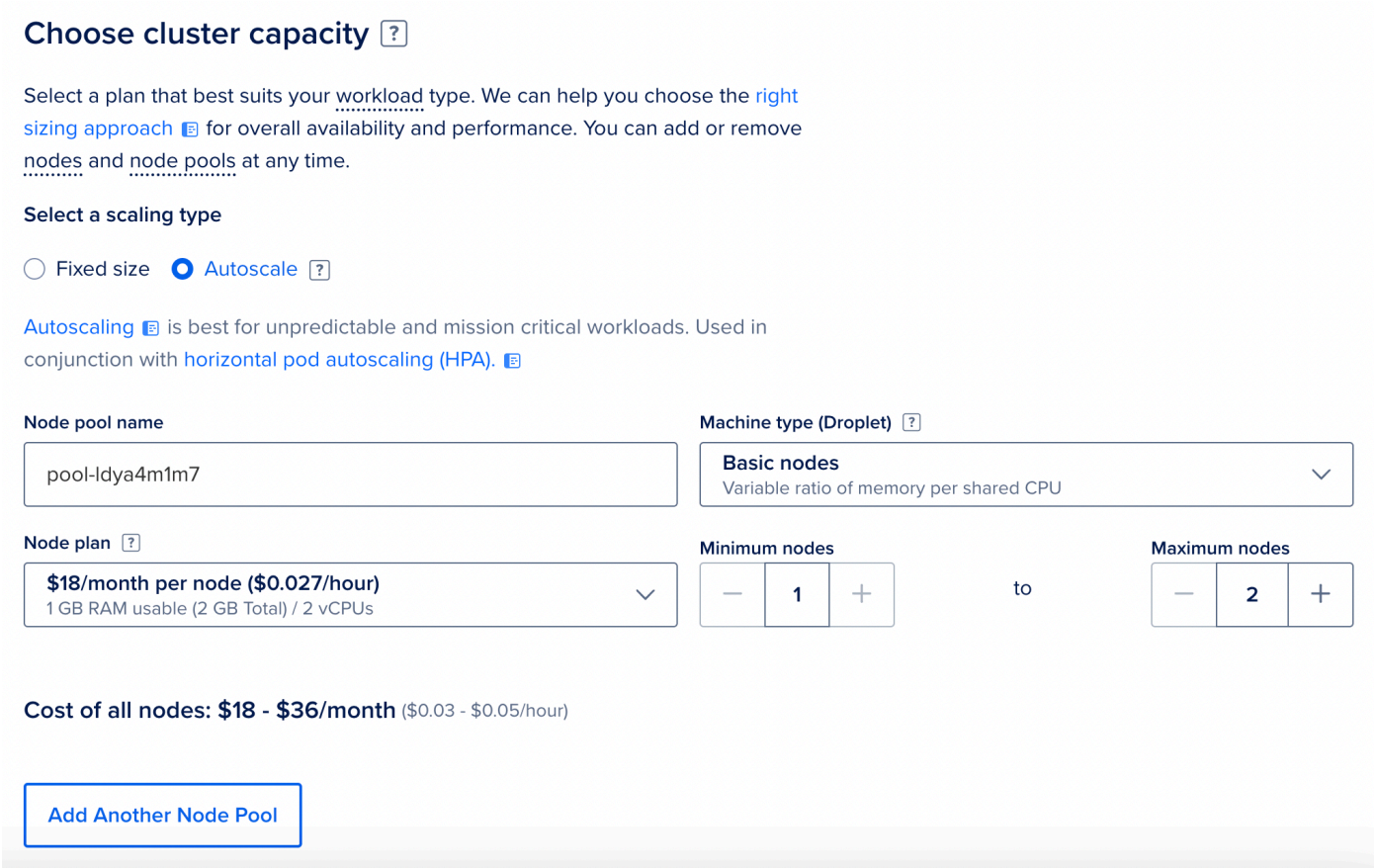
```
kubectl version --output=yaml
```

If you're not using Docker Desktop you can check out the official guide.

Next, go to DigitalOcean and create a new k8s cluster. In the pool configuration, there are two important things:

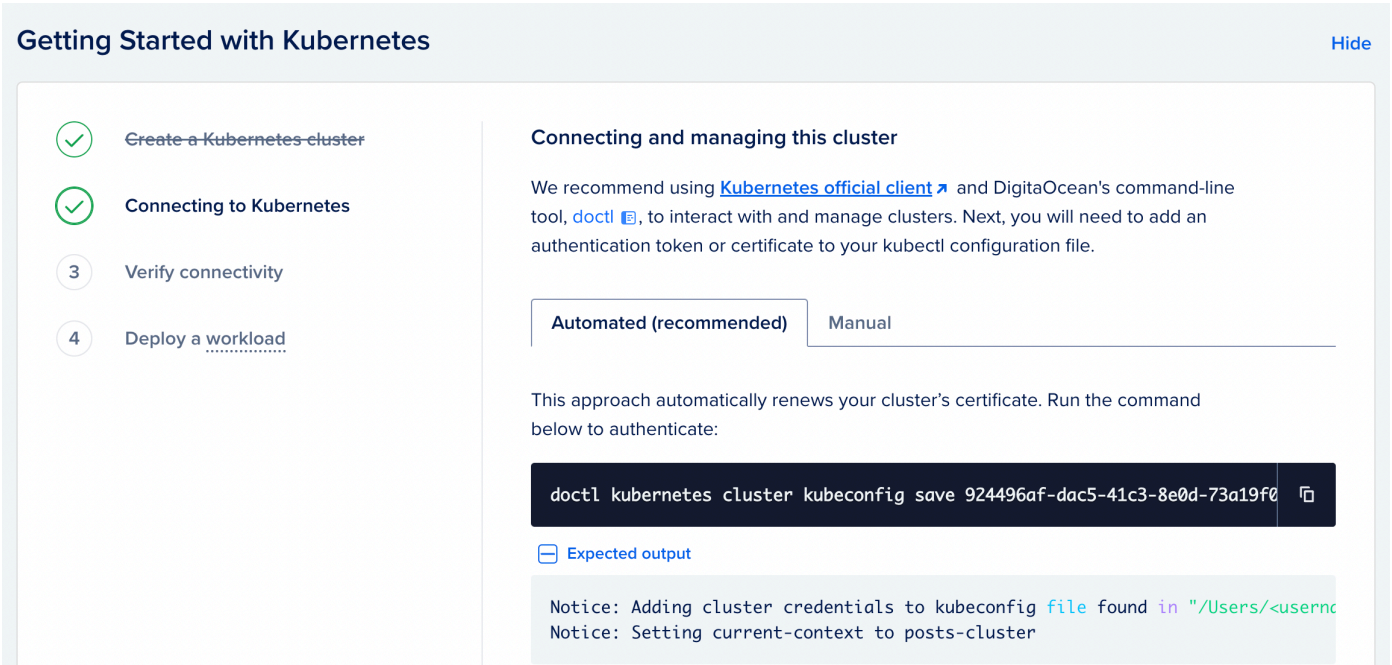
- Fixed-size or autoscale. If you choose a fixed size you'll have a cluster of X nodes. If you choose autoscale you can define the minimum and the maximum number of nodes. Your cluster will start with the minimum number and will scale up if needed.
- The number of nodes. For this demo project, I'm going to choose autoscale with 1-2 nodes.

This is my node pool config:



I choose the $18 nodes because they have 2 CPUs and they are a bit better (in a tiny 2-node cluster it's a huge difference, actually).

After the cluster is created you should see a page like this:



Run the command shown on the page:

```
doctl kubernetes cluster kubeconfig save <your-cluster-id>
```

It creates a file in `$HOME/.kube/config` and configures `kubectl` to connect to your new DigitalOcean k8s cluster.

That's it! Now you can verify if it works:

```
kubectl get pods -A
```

This should return a table with pods in the `kube-system` namespace.

Managed databases

I you haven't read the Docker Swarm chapter, please check out the first part of it (the one that talks about the state and also applies to Kubernetes).

Here's the TL;DR:

- A database has a state (the files that represent the tables and your data) and it's hard to manage this state in a replicated environment.
- k8s can't just place replicas of `mysql` container to a random node because it needs the files.

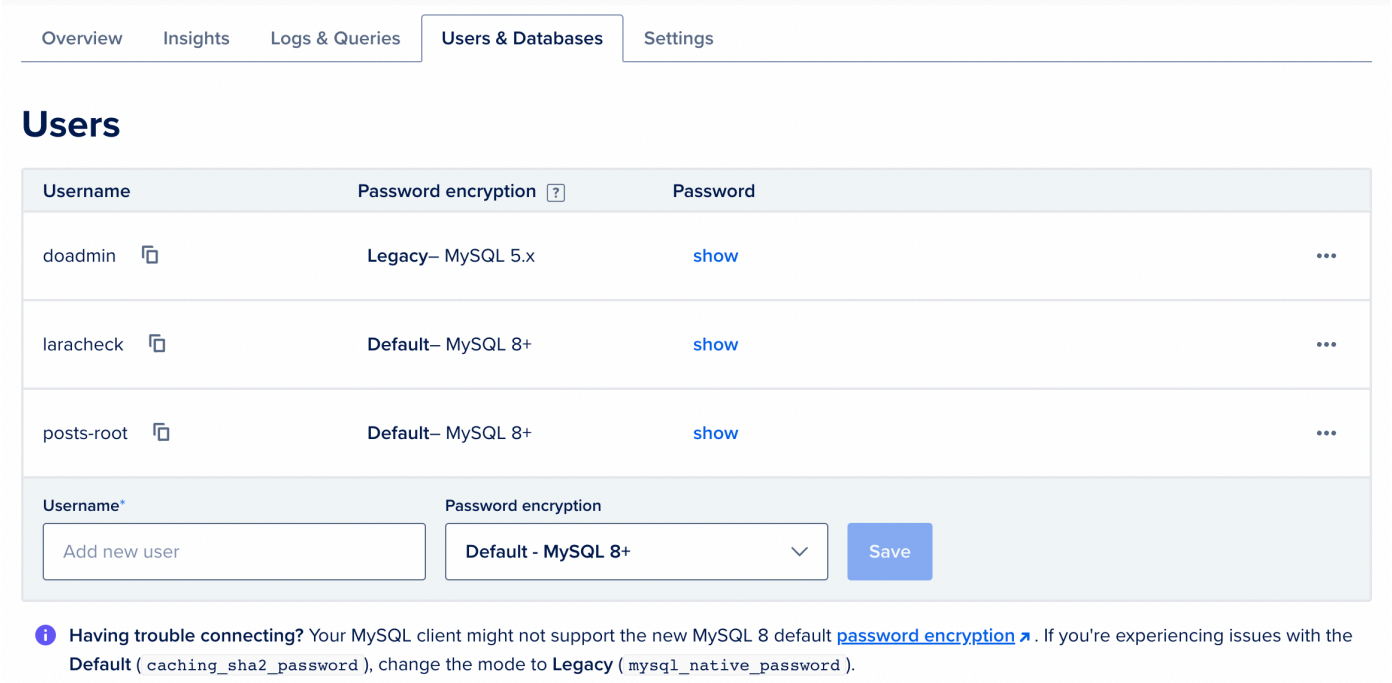
The way we solved this issue was placement constraints in Swarm. k8s offers something called `PersistentVolumeClaim` and `StatefulSet`. The `PersistentVolumeClaim` create a volume that stores the data, and the `StatefulSet` handles the pod placement in a way that each has a persistent identifier that k8s maintains across any rescheduling.

But in this chapter, we're not going to host the database for ourselves. We're going to use managed databases. A managed database is a simple database server provided and maintained by a cloud provider. They have a number of advantages:

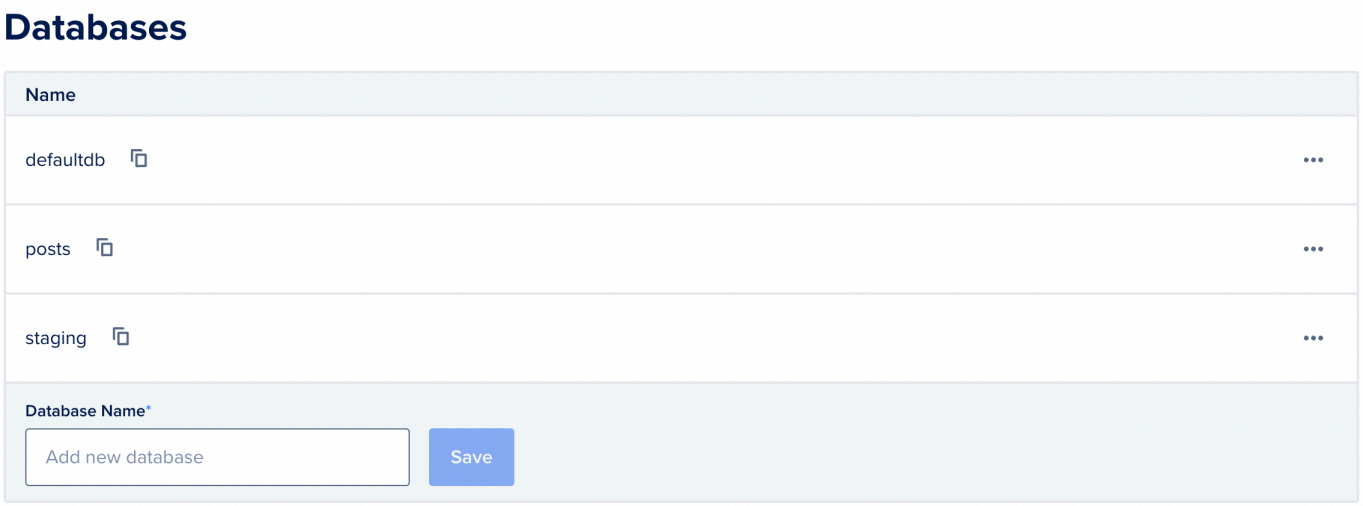
- You don't have to solve the state problem when deploying in a cluster
- You have automatic backups and upgrades
- The uptime is pretty good
- It's easy to increase the server size if your application grows
- It's easy to create a replicated database cluster (a master node and a read-only node, for example). Which is not that straightforward if you try to do it on your own.
- You have great dashboards and metrics by default

And of course, the biggest disadvantage is that you have to pay extra money. At DigitalOcean, managed MySQL databases range from $15 per month to $3830 per month. Similar pricing applies to Redis as well.

Just go to DigitalOcean and create a new MySQL server. After it's created you can manage the databases and users:

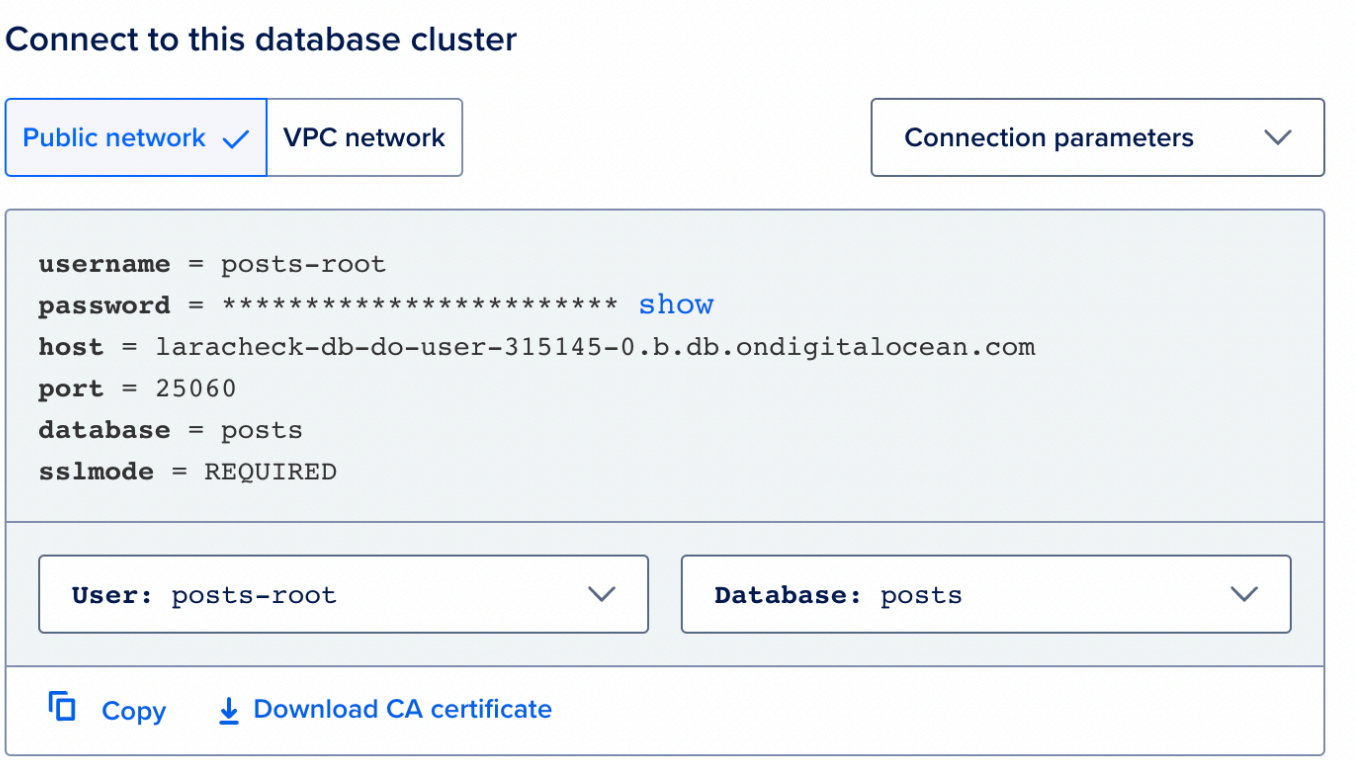


You have a default `doadmin` user. You can use that, or create a new one. I created a `posts-root` for only this project.



The server also comes with a default database called `defaultdb`. You can use that or create a new one. I created a new one called `posts`.

On the `Overview` tab they provide you with the connection details:



These are the environment variables we need to use later.

That's it. If you now want to try it, just set up these values in a random Laravel project and run your migrations. It should work.

One small thing. In the `Connection parameters` dropdown there's a `Connection string` option that gives a connection string similar to this:

```
mysql://<your-user>:<your-password>@laracheck-db-do-user-315145-0.b.db.ondigitalocean.com:25060/posts?ssl-mode=REQUIRED
```

It can be set in the `DATABASE_URL` environment variable. I'm going to use that since it only requires one environment variable so it's a bit more convenient.

I also created a **Redis** database server. It's the same process with the same steps. After it's created you'll have a connection string as well:

```
rediss://<your-user>:<your-password>@posts-redis-do-user-315145-0.b.db.ondigitalocean.com:25061
```

Now that the databases are ready, let's create our deployments!

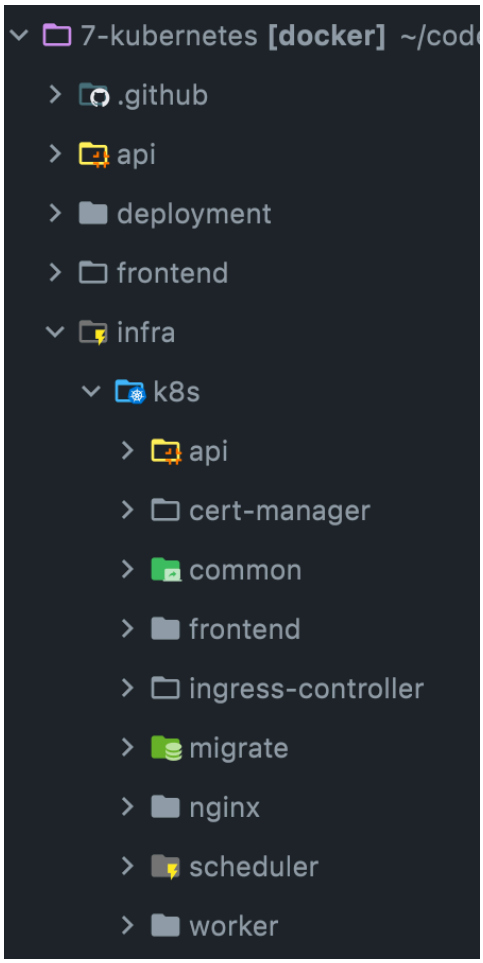
Deploying a Laravel API

Configuring the deployment

Let's start with the API. I'm going to create k8s-related files inside the

```
/infra/k8s
```

folder. Each component (such as API or frontend) gets a separate directory so the project looks like this:



In the `api` folder create a file called `deployment.yml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 4
  selector:
    matchLabels:
```

```
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: martinjoo/posts-api:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9000
```

You've already seen a deployment similar to this one.

The image I'm using is `martinjoo/posts-api:latest`. In production, I never use the `latest` tag. It is considered a bad practice. It's a bad practice because you don't know exactly which version you're running and it's harder to roll back to a previous version if something goes wrong since the previous version was also `latest`... And also, `latest` if you're using some 3rd party images, `latest` is probably not the most stable version of the image. As the name suggests, it's the latest meaning it has the most bugs. Always use exact versions of Docker images. Later, I'm going to remove the `latest` tag and use commit SHAs as image tags.

The other thing that is new is the `imagePullPolicy`. It's a configuration setting that determines how a container image is pulled by k8s. It specifies the behavior for image retrieval when running or restarting a container within a pod. There are three values:

- `Always`: The container image is always pulled, even if it exists locally on the node. This ensures that the latest version of the image is used, but it means increased network and registry usage.
- `IfNotPresent`: The container image is only pulled if it is not already present on the node. If the image already exists locally, it will not be pulled again. This is the default behavior.
- `Never`: The container image is never pulled. It relies on the assumption that the image is already present locally on the node. If the image is not available, the container runtime will fail to start the container.

`IfNotPresent` is a pretty reasonable default value. However, if you use a tag such as `latest` or `stable` you need to use `Always`.

A note about `containerPort`. As I said in the introduction, exposing ports in k8s is a bit more tricky than defining two values in a YAML file. This `containerPort` config basically **does nothing**. It won't expose port 9000 to the outside world or not even to other containers in the cluster. It's just information that is useful for developers. Nothing more. Later, we're going to expose ports and make communication possible between components.

Configs and secrets

Before we can run the deployment we need to add environment variables to the deployment. In Kubernetes, we have two objects to handle environment variables: `ConfigMap` and `Secret`.

Create a new directory called `infra/k8s/common` and a new file called `app-config.yml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: posts
data:
  APP_NAME: "posts"
  APP_ENV: "production"
  APP_DEBUG: "false"

  LOG_CHANNEL: "stack"
  LOG_LEVEL: "error"

  QUEUE_CONNECTION: "redis"

  MAIL_MAILER: "log"

  AWS_BUCKET: "devops-with-laravel-storage"
  AWS_DEFAULT_REGION: "us-east-1"
  AWS_USE_PATH_STYLE_ENDPOINT: "false"
  AWS_URL: "https://devops-with-laravel-storage.s3.us-east-1.amazonaws.com/"

  FILESYSTEM_DISK: "s3"

  CACHE_DRIVER: "redis"
```

The `kind` is set to `ConfigMap` and `data` basically contains **the non-secret** environment variables from your `.env` file. Don't add passwords or tokens to this file!

And now we have to attach this config to the container:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 4
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: martinjoo/posts-api:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9000
          envFrom:
            - configMapRef:
                name: posts
```

`envFrom` and `configMapRef` are used to inject environment variables into containers from a `ConfigMap` resource we just created. `envFrom` is a field that allows you to specify multiple sources from which you can load environment variables. So we load the `configMap` resource named `posts`. If you remember we specified a name for the `ConfigMap`:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: posts
```

That's the name we are referencing in `configMapRef`. With this setting, all the values defined in the `data` key of the `ConfigMap` can be accessed by the container as environment variables.

Now we need to handle secrets as well. Create a new file called `app-secret.yml` in the `infra/k8s/common` folder:

```
apiVersion: v1
kind: Secret
metadata:
  name: posts
type: Opaque
 ...
```

The `kind` is `Secret` and the `type` is `Opaque`.

When you create a Secret, you can store different types of data such as strings, TLS certificates, or SSH keys. The Opaque type does not enforce any special encoding or structure on the stored data. It is typically used when you want to store arbitrary key-value pairs. So it's a good type for storing string-like secret values such as passwords or tokens.

And here's the content of the secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: posts
type: Opaque
stringData:
  APP_KEY: "..."

  DATABASE_URL: "..."

  REDIS_URL: "..."

  AWS_ACCESS_KEY_ID: "..."
  AWS_SECRET_ACCESS_KEY: "..."

  ROLLBAR_TOKEN: "..."

  HEALTH_CHECK_EMAIL: "..."
```

Do this for only development purposes

Yes, right now we're storing database passwords and AWS access keys in raw format. Later, we're going to solve that, of course, but for learning purposes it's perfect.

We can use the `Secret` the same way we used the `ConfigMap`:

```
...
envFrom:
  - configMapRef:
      name: posts
  - secretRef:
      name: posts
```

Applying the deployment

And now you can apply the deployment and run the pods.

Every resource we configure in these yaml files needs to be applied with

```
kubectl apply
```

It will create an actual object from the configuration and run the pods if it's a deployment.

So let's apply the changes:

```
kubectl apply -f infra/k8s/common/app-config.yml
kubectl apply -f infra/k8s/common/app-secret.yml
kubectl apply -f infra/k8s/api/deployment.yml
```

If everything went well, you should see message such as this:

```
deployment.apps/api created
```

And now you can see the running pods in your cluster:

```
kubectl get pods
```

```
→  7-kubernetes git:(main) k get pods
NAME                   READY   STATUS    RESTARTS   AGE
api-595b8846f-9j7kl    1/1     Running   0          19h
api-595b8846f-j6fqp    1/1     Running   0          19h
api-595b8846f-rv54t    1/1     Running   0          19h
api-595b8846f-rxjdt    1/1     Running   0          19h
```

As we expected, the deployment created 4 pods because of the `replicas: 4` setting. Each pod has the name `api` with a random identifier.

You can check out the logs of a pod by running:

```
kubectl logs <pod-name>
```

```
→  7-kubernetes git:(main) k logs api-595b8846f-9j7kl
Defaulted container "api" out of: api, optimize (init)
[29-Jul-2023 16:14:14] NOTICE: [pool www] 'user' directive is ignored when FPM is not running as root
[29-Jul-2023 16:14:14] NOTICE: [pool www] 'user' directive is ignored when FPM is not running as root
[29-Jul-2023 16:14:14] NOTICE: [pool www] 'group' directive is ignored when FPM is not running as root
[29-Jul-2023 16:14:14] NOTICE: [pool www] 'group' directive is ignored when FPM is not running as root
[29-Jul-2023 16:14:14] NOTICE: fpm is running, pid 1
[29-Jul-2023 16:14:14] NOTICE: ready to handle connections
```

(Ignore the first line for now)

If you need more information about a running pod you can run

```
kubectl describe pod <pod-name>
```

It gives you information such as the image name, the config maps or secrets used by the container, and lots of other things.

You can also list the secrets or config maps:

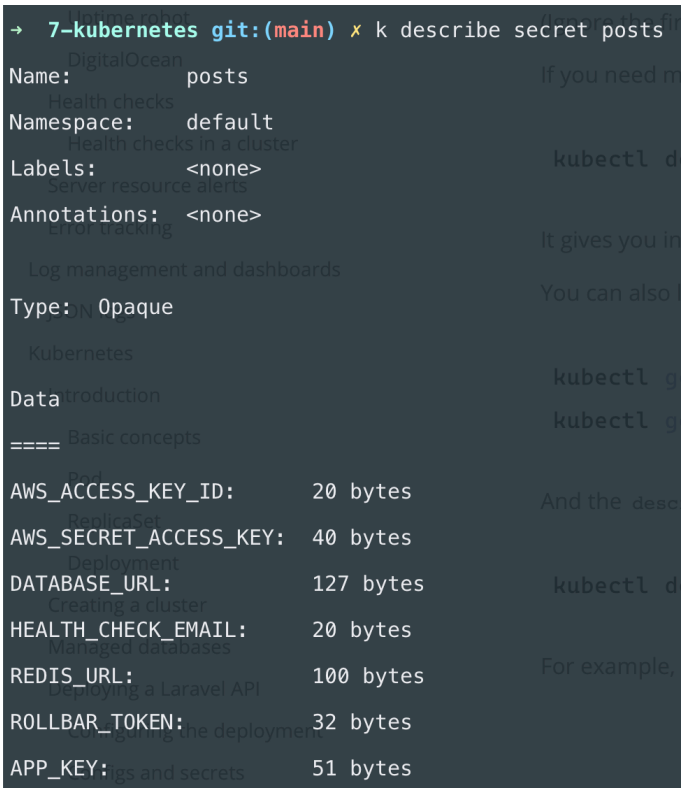
```
kubectl get secrets
kubectl get configmaps
```

And the `describe` command can be used with a number of different resources in the format of

```
kubectl describe <resource-type> <resource-name>
```

For example, here's how you check the secrets available to the application:

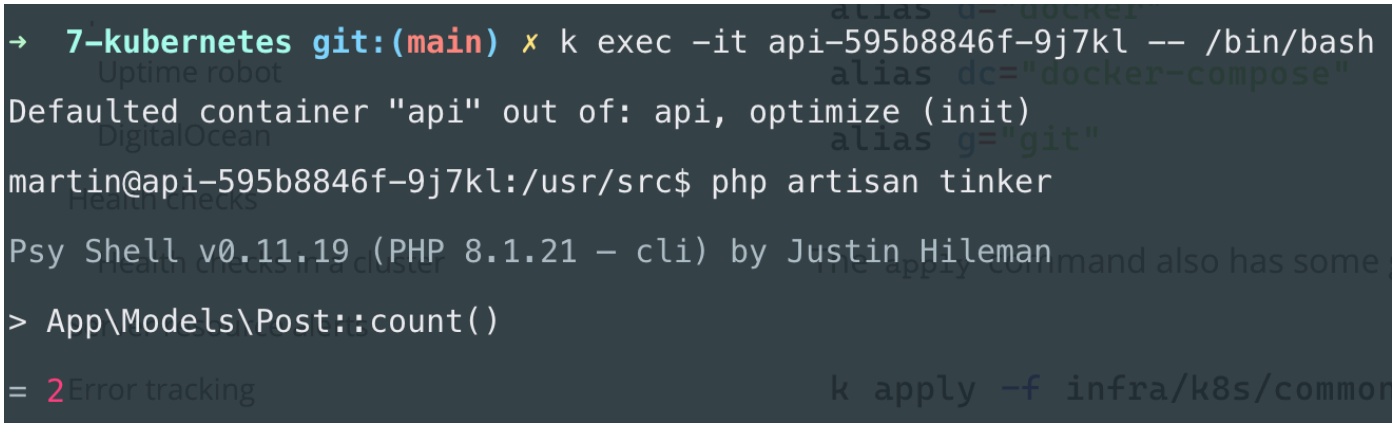
```
kubectl describe secret posts
```



If you want to validate that the database works you can run artisan commands inside the pods by running this command:

```
kubectl exec -it <pod-name> -- /bin/bash
```

`-it` opens an interactive terminal session (just as `docker exec -it`) and `/bin/bash` runs the bash:



Shortcuts

First of all, you're going to type `kubectl` a lot. Make your life easier and add this alias to your `.bash_aliases` or `.zsh_aliases` file located in your home directory:

```
alias k="kubectl"
alias d="docker"
alias dc="docker-compose"
alias g="git"
```

The `apply` command also has some great features. For example, it can read a whole directory, so instead of these commands:

```
k apply -f infra/k8s/common/app-config.yml
k apply -f infra/k8s/common/app-secret.yml
```

We can just run this one:

```
k apply -f infra/k8s/common
```

It will read the YAML located in the `common` directory. It's not recursive so it won't fetch subdirectories.

However, there's a `-R` flag, so we can do just this:

```
k apply -R -f infra/k8s
```

This command applies everything located inside the `k8s` folder and its subfolders.

kubectl apply

`apply` is the most frequently used command so let's talk about it a little bit.

It creates and updates resources based on your current configuration files. When you run the command k8s checks the cluster and evaluates the difference between the current state and the desired state. The current state is, well, the current state of your cluster. For example, before you run the `apply` commands the current state was nothing. There were no deployments, pods, or config maps. When you first ran the command:

```
kubectl apply -f infra/k8s/common/app-config.yml
```

The desired state was to create a `ConfigMap` resource with the environment variables in your YAML file. So k8s created the resource. The same happened with the deployment and pods.

If you now change your deployment config, for example changing the `image`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
   ...
     containers:
       - name: api
         image: martinjoo/posts-api:1.0.0
```

And run apply again, you'll get a message such as this:

```
deployment.apps/api configured
```

`configured` mean the desired state was different from the current state so k8s configured your cluster and made the necessary changes.

`apply` is the recommended way of deploying applications to production so we're going to use it a lot.

Deploying nginx

The API cannot do too much on its own. It needs nginx so next we're deploying that component.

The deployment file is located in `infra/k8s/nginx/deployment.yml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: martinjoo/posts-nginx:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 80
```

That's it. It doesn't need environment variables or secrets. And once again, `containerPort` is only for informational purposes as I explained in the API chapter.

We can apply nginx by running:

```
kubectl apply -f infra/k8s/nginx
```

Communication between nginx and FPM

The next step is to make communication possible between nginx and API. To do that we need a service.

Service is another `kind` of resource, and there are different service types but the one we need is `ClusterIP`.

`ClusterIP` is the default type for services and this is what we need right now. It is a service type that provides connectivity to **internal** cluster IP addresses. So it makes communication possible among pods. Internally, in the cluster. This is exactly what we need since we want to connect nginx and FPM using port 9000 exposed by the `api` container.

There are a few other services such as `NodePort` or `LoadBalancer`. We'll talk about `LoadBalancer` later. You can check out the other types in the official documentation.

Create a new file called `service.yml` in the `infra/k8s/nginx` folder:

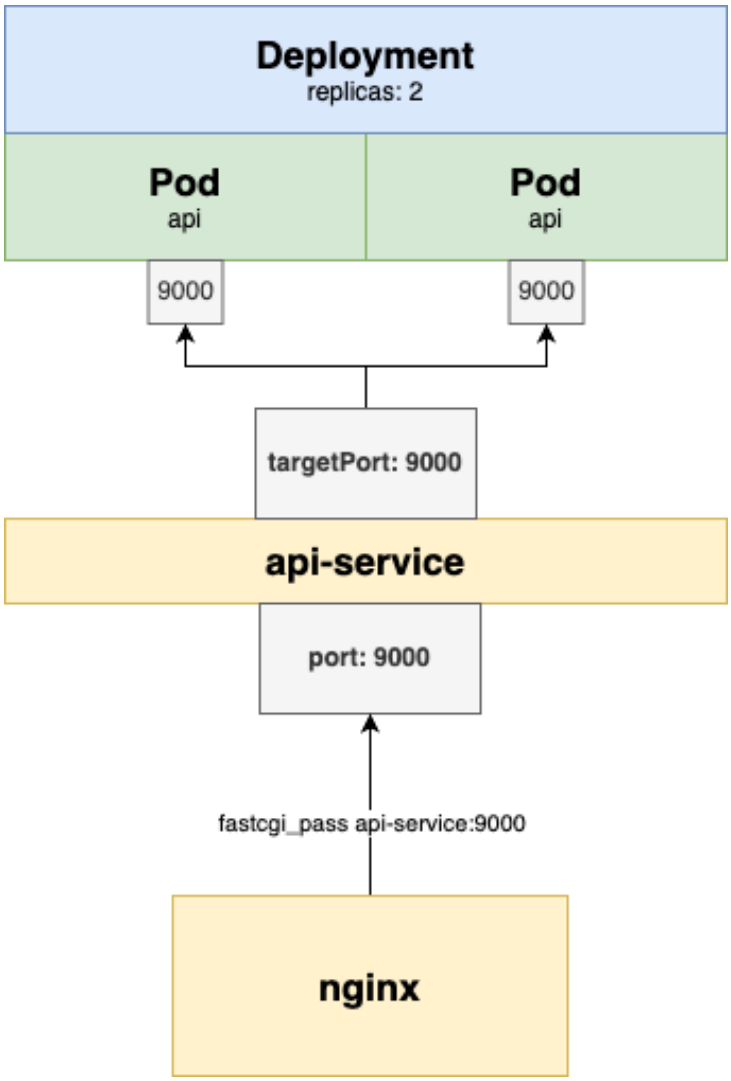
```
apiVersion: v1
kind: Service
metadata:
  name: api-service
spec:
  selector:
    app: api
  ports:
    - protocol: TCP
      port: 9000
      targetPort: 9000
```

The `kind` is set to `Service` and the `type` is `ClusterIP`. Since `ClusterIP` is the default value I'm not going to explicitly write it down in the future.

The service target ports labeled as `nginx` are defined in the `selector` object. And then the important part:

```
ports:
  - protocol: TCP
    port: 9000
    targetPort: 9000
```

`targetPort` is the port listening in the container. And `port` is the one that is going to be exposed to other pods in the cluster.



As you can see, the `api-service` exposes port 9000 to other components in the cluster. It also acts as a load balancer because it balances the traffic among the API pods.

As you might expect, these names are going to be domain names inside the cluster. Just as in docker-compose, you can access the API container from nginx as `api:9000`. The same thing can be done with Kubernetes as well. The only difference is that now in nginx we cannot reference `api` since it's the pod's name. We need to use `api-service` since the service exposes the port.

Change `api:9000` to `api-service:9000` in the nginx config file:

```
location ~\.php {
  try_files $uri =404;
  include /etc/nginx/fastcgi_params;
  # api-service:9000 instead of api:9000
  fastcgi_pass api-service:9000;
  fastcgi_index index.php;
  fastcgi_param PATH_INFO $fastcgi_path_info;
  fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
}
```

The next is to add an `nginx-service`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

It's the same thing it exposes port 80.

Apply the changes to the cluster:

```
kubectl apply -R -f infra/k8s
```

If you now run `kubectl get services` you should see `api-service` and `nginx-service` running:

```
→  7-kubernetes git:(main) ✗ k get services
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
api-service       ClusterIP   10.245.137.220   <none>        9000/TCP   25h
frontend-service  ClusterIP   10.245.28.127    <none>        80/TCP     25h
kubernetes        ClusterIP   10.245.0.1       <none>        443/TCP    25h
nginx-service     ClusterIP   10.245.236.203   <none>        80/TCP     25h
```

Right now, there's no way to access the cluster because it's not exposed to the outside world yet. But we can still test the configuration.

You can forward traffic from the cluster to your local machine by running this command:

```
kubectl port-forward svc/nginx-service 8080:80
```

You can test if it works by hitting the `health-check` API:

```
curl -w "%{http_code}\n" localhost:8080/api/health-check
```

It should write `200` to your terminal.

You can also check the logs of the nginx pods with `kubectl log <nginx-pod-name>` and you should see access log entries such as these:


```
{"time_local":"30/Jul/2023:13:32:24 +0000","remote_addr":"127.0.0.1","request_method":"GET","request_uri":"/api/health-check","status":"200","body_bytes_sent"
:"5","http_referer":"-","http_user_agent":"curl/7.88.1"}
{"time_local":"30/Jul/2023:13:32:25 +0000","remote_addr":"127.0.0.1","request_method":"GET","request_uri":"/api/health-check","status":"200","body_bytes_sent"
:"5","http_referer":"-","http_user_agent":"curl/7.88.1"}
{"time_local":"30/Jul/2023:13:32:27 +0000","remote_addr":"127.0.0.1","request_method":"GET","request_uri":"/api/health-check","status":"200","body_bytes_sent"
:"5","http_referer":"-","http_user_agent":"curl/7.88.1"}
{"time_local":"30/Jul/2023:13:32:29 +0000","remote_addr":"127.0.0.1","request_method":"GET","request_uri":"/api/health-check","status":"200","body_bytes_sent"
:"5","http_referer":"-","http_user_agent":"curl/7.88.1"}
```


If you don't see the logs it's likely that you run multiple replicas and you access the logs from the wrong replica.

Deploying a worker

The worker's deployment file is very similar to the API. The file is located in `infra/k8s/worker/deployment.yml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker
spec:
  replicas: 2
  selector:
    matchLabels:
      app: worker
  template:
    metadata:
      labels:
        app: worker
    spec:
      containers:
        - name: worker
          image: martinjoo/posts-worker:latest
          imagePullPolicy: Always
          envFrom:
            - configMapRef:
                name: posts
            - secretRef:
                name: posts
```

It runs the `posts-worker` image with the same `ConfigMap` and `Secret` we created earlier.

If you remember from earlier, the entry point of the worker image is this script:

```
nice -10 php /usr/src/artisan queue:work --queue=default,notification --
tries=3 --verbose --timeout=30 --sleep=3 --max-jobs=1000 --max-time=3600
```

It simply runs the `queue:work` command.

That's it. Now we can apply it:

```
kubectl apply -R -f infra/k8s
```

Deploying a scheduler

The next component is the scheduler. If you remember from the earlier chapters of the book it always needed some special care.

With docker-compose, we ran this script in the Dockerfile: `CMD ["/bin/sh", "-c", "nice -n 10 sleep 60 && php /usr/src/artisan schedule:run --verbose --no-interaction"]`. It sleeps for 60 seconds runs the `scheduler:run` command and then it exists. compose will restart it and the process starts over.

Swarm eliminated the need for `sleep 60` because it can restart containers with a delay. So the container config was this:

```
image: martinjoo/posts-scheduler:${IMAGE_TAG}
command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/scheduler.sh"
restart_policy:
  condition: any
  delay: 60s
  window: 30s
```

And `scheduler.sh` was dead simple: `nice -n 10 php /usr/src/artisan schedule:run --verbose --no-interaction`

Fortunately, Kubernetes offers a solution and we can eliminate these "hacks" entirely. There is a resource type called `CronJob` and it's pretty simple and great:

- We need to define the container config and the image
- Then we specify the schedule and k8s takes care of the rest. It runs and stops the container according to the schedule we defined.

Create a new file in `infra/k8s/scheduler/cronjob.yml`:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: scheduler
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
```

```
        containers:
          - name: scheduler
            image: martinjoo/posts-scheduler:latest
            imagePullPolicy: Always
            envFrom:
              - configMapRef:
                  name: posts
              - secretRef:
                  name: posts
```

The `kind` is set to `CronJob` and the `schedule` defines how frequently k8s needs to run the container. In this case, it's every minute. You can use the usual Linux crontab syntax. In the `spec` section, however, you need to use `jobTemplate` instead of `template`. Other than that, everything is the same as with any other deployment so far.

Let's apply the changes:

```
kubectl apply -R -f infra/k8s
```

In the case of a `CronJob` you can always check out the last 3 runs of it using `kubectl get pods`:

```
scheduler-28178766-zrqwm    0/1     Completed   0          2m57s
scheduler-28178767-scwft    0/1     Completed   0          117s
scheduler-28178768-5dvgs    0/1     Completed   0          57s
```

Running `kubectl logs <pod-name>` you van get the log messages:

```
→  7-kubernetes git:(main) ✗ k logs scheduler-28178770-mgspr

  2023-07-30 14:10:01 Running [App\Jobs\PublishPostsJob] ..... 533ms DONE
```

And now you can also get some logs from the worker pods since the scheduler dispatches jobs.

Deploying a frontend

Finally, we need to deploy the frontend as well.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: martinjoo/posts-frontend:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 80
```

Since the frontend also exposes a port we need a service as well:

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

There's nothing new we haven't seen so far. The files are located in the `infra/k8s/frontend` folder.

Don't forget to apply them:

```
kubectl apply -R -f infra/k8s
```

Running migrations in a cluster

Just as we discussed in the Docker Swarm chapter, running migrations when you have a cluster of services can be tricky.

If you make the API pod responsible for running migrations (for example, defining a command that runs `php artisan migrate`) it won't work very well. These pods run in multiple replicas so each of them tries to run the command. What happens when two migrations run concurrently? I don't know exactly, but it does not sound good at all.

With Docker Swarm, we solved this problem by defining a `replicated_job` that runs exactly once on every deployment. Fortunately, k8s also provides a solution like this. This resource is called `Job`.

Create a new file called `infra/k8s/migrate/job.yml`:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: migrate
spec:
  template:
    spec:
      containers:
        - name: migrate
          image: martinjoo/posts-api:latest
          command: ["sh", "-c", "php artisan migrate --force"]
          envFrom:
            - configMapRef:
                name: posts
            - secretRef:
                name: posts
```

The `kind` is set to `Job` which means this container will run only once when deploying or updating the app. As you can see, it runs the `posts-api` image but the command is overwritten to `php artisan migrate --force`. So the Job starts, it runs `artisan migrate` then it exists.

Jobs can also define a `restartPolicy` and a `backoffLimit`:

- `restartPolicy` tells Kubernetes when to restart the job. Usually, we don't want to restart these jobs, but there's a `restartPolicy: OnFailure` option that can be useful.
- `backoffLimit` defines how many times Kubernetes tries to restart the job if it fails.

For example, a `restartPolicy: OnFailure` with a `backoffLimit: 2` option means that k8s restarts the job two times if something goes wrong.

However, if migrations fail, I don't think it's a good idea to run them again before you investigate what happened. The only situation I can think of is when the database was unavailable for a short period of time. But hopefully, that happens pretty rarely.

You can apply the changes:

```
kubectl apply -R -f infra/k8s
```

If you now change the image version (for example) and try to run `apply` again you'll get an error:

```
The Job "migrate" is invalid: spec.template: Invalid value:
core.PodTemplateSpec{ObjectMeta:v1.ObjectMeta{:} field is immutable
```

The thing is that a job is immutable. Meaning, once applied it cannot be re-applied. Before you apply an existing job, you need to delete it by running:

```
kubectl delete -f infra/k8s/migrate/job.yml
```

This is going to be an important command when deploying the cluster from a pipeline.

Caching configs

As I said in the introduction, a pod can run multiple containers. But why is this useful? I show you an interesting situation.

It's highly recommended to run `php artisan optimize` after deploying an application (it caches configs and routes). But where to run it?

A logical answer should be to run it after the migrations. Let's see:

```
- name: migrate
  image: martinjoo/posts-api:latest
  command: ["sh", "-c", "php artisan migrate --force && php artisan
optimize"]
```

But now all we did is this:

- k8s runs the API image in a container
- Migrations run successfully
- `artisan optimize` collects the configs and the routes and caches them into one single file
- k8s **removes** the container

So all we did was cache routes and configs in a container, which was then removed after 3 seconds.

We need to run `artisan optimize` as the `CMD` of the Dockerfile. This is the end of the `api` stage:

```
FROM php:8.1-fpm as api
 ...
CMD ["/bin/sh", "-c", "php artisan optimize && php-fpm"]

FROM api AS worker
CMD ["/bin/sh", "/usr/src/worker.sh"]
```

And I also added this to the `worker.sh` script:

```
#!/bin/bash

nice -10 php /usr/src/artisan optimize && php /usr/src/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3 --max-
jobs=1000 --max-time=3600
```

Liveness and readiness probes

The next chapter is **crucial**. As you may know, Kubernetes has self-healing properties. Meaning it kills, starts, and restarts pods if they are unhealthy. But how does it know when a pod is unhealthy? There are obvious situations, for example, when your worker process exits with code 1. But we can also configure more advanced checks, for example, restart the API if it takes more than 1s to respond.

These are called "probes" in k8s. They are basically the same as health checks in docker-compose or Swarm but much easier to write down.

It's pretty easy to confuse them so here's a quick definition:

- `readinessProbe` asks the container: "Are you ready to work?"
- On the other hand, `livenessProbe` asks the container: "Are you still alive?"

So `readinessProbe` plays a role when the container is being started, while `livenessProbe` is important when the container is already running.

`readinessProbe` indicates whether a container is ready to serve requests. It ensures that it is fully initialized before starting to receive incoming traffic. If a container fails the `readinessProbe`, it is temporarily removed from load balancing until it passes the probe. This allows k8s to avoid sending traffic to containers that are not yet ready or are experiencing problems.

`livenessProbe` is used to determine whether a container is running as expected, and if it's not, Kubernetes takes action based on the probe's configuration. It helps in detecting and recovering from failures automatically. If a container fails the livenessProbe, Kubernetes will restart the container.

These probes together help ensure the high availability of your app by making sure containers are running correctly (`livenessProbe`) and ready to serve requests (`readinessProbe`).

Let's configure liveness and readiness probes for every container!

API probes

```
template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: martinjoo/posts-api:latest
          imagePullPolicy: Always
          livenessProbe:
            tcpSocket:
              port: 9000
            initialDelaySeconds: 20
            periodSeconds: 30
            failureThreshold: 2
          readinessProbe:
            tcpSocket:
              port: 9000
            initialDelaySeconds: 10
            periodSeconds: 30
            failureThreshold: 1
```

There are 3 different types of probes:

- tcpSocket
- httpGet
- exec

The API exposes port 9000 over TCP so in this case `tcpSocket` is the one we need. Both `livenessProbe` and `readinessProbe` has the same properties:

- `initialDelaySeconds` tells k8s to wait 20 seconds before sending the request.
- `periodSeconds` tells k8s to run the probe every 30 seconds.
- `failureThreshold` instructs k8s to tolerate only 2 failed probes.

The following sentence comes from the Kubernetes documentation:

> *A common pattern for liveness probes is to use the same low-cost HTTP endpoint as for readiness probes, but with a higher failureThreshold. This ensures that the pod is observed as not-ready for some period of time before it is hard killed.*

This is why I use the same config but `failureThreshold` is set to 2 in the `livenessProbe`. The `initialDelaySeconds` is also a bit higher.

nginx probes

```
livenessProbe:
  httpGet:
    path: /api/health-check
    port: 80
  initialDelaySeconds: 30
  periodSeconds: 30
  failureThreshold: 2
readinessProbe:
  httpGet:
    path: /api/health-check
    port: 80
  initialDelaySeconds: 20
  periodSeconds: 30
  failureThreshold: 1
```

Since nginx exposes an HTTP port we can use the `httpGet` type which is pretty straightforward to use. I'm using a bit higher `initialDelaySeconds` because nginx needs the API.

An `httpProbe` fails if the response is not `2xx` or `3xx`.

worker probes

```
livenessProbe:
  exec:
    command:
      - sh
      - -c
      - php artisan queue:monitor default
  initialDelaySeconds: 20
  periodSeconds: 30
  failureThreshold: 2
readinessProbe:
  exec:
    command:
      - sh
      - -c
      - artisan queue:monitor default
  initialDelaySeconds: 10
  periodSeconds: 30
  failureThreshold: 1
```

Workers don't expose any port, so I just run `queue:monitor` command with the `exec` probe type. You can also use an array-type notation:

```
command: ["sh", "-c", "php artisan queue:monitor default"]
```

frontend probes

```
livenessProbe:
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 40
  periodSeconds: 30
  failureThreshold: 2
readinessProbe:
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 30
  periodSeconds: 30
  failureThreshold: 1
```

timeoutSeconds

If something is wrong with some of your pods based on the `livenessProbe` you can see it in the `events` section in the output of `kubectl describe pod <pod-name>`

For example:

```
  Normal   Started            21m (x3 over 24m)     kubelet            Started container nginx
  Warning  Unhealthy          20m (x5 over 23m)     kubelet            Liveness probe failed: Get "http://10.244.0.69:80/api/health-check": context deadline
exceeded (Client.Timeout exceeded while awaiting headers)
  Warning  BackOff            9m46s (x7 over 10m)   kubelet            Back-off restarting failed container nginx in pod nginx-5d655cc6cc-p96vt_default(8b2ab
ede-3f37-4c1b-93a5-0d7abcc90074)
  Warning  Unhealthy          4m59s (x29 over 24m)  kubelet            Readiness probe failed: Get "http://10.244.0.69:80/api/health-check": context deadline
 exceeded (Client.Timeout exceeded while awaiting headers)
```

You can see that it started 21 minutes ago, but then the liveness probe failed 20 minutes ago so Kubernetes restarted the container but then the readiness probe failed.

We can check the logs of the nginx container:

```
→  7-kubernetes git:(main) k logs nginx-5d655cc6cc-p96vt
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
{"time_local":"30/Jul/2023:16:46:36 +0000","remote_addr":"10.244.0.106","request_method":"GET","request_uri":"/api/health-check","status":"499","body_bytes_se
nt":"0","http_referer":"-","http_user_agent":"kube-probe/1.27"}
```

The status code is `499` which is an nginx-specific code that means the client closed the connection before the server could send the response. This is because there's another option for probes called `timeoutSeconds` which defaults to 1. So if your container doesn't respond in 1 second the probe fails.

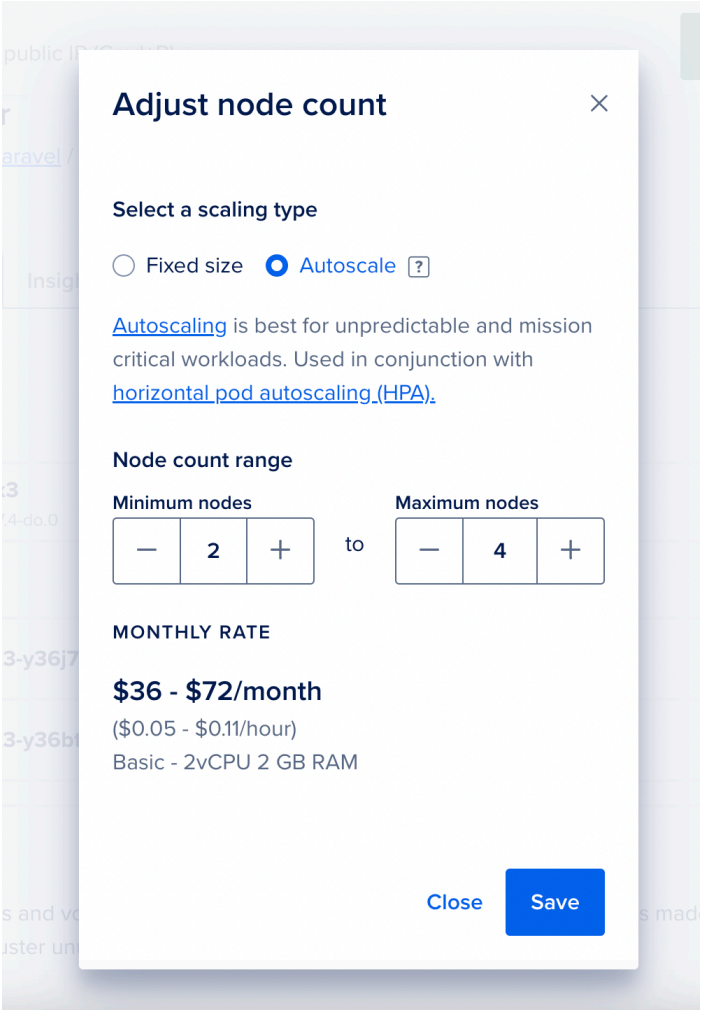
It's a good thing because your probe endpoint should be pretty low-cost and very fast. So if 1 second is not enough for your container to respond there's certainly a problem with it. In my case, I made this situation happen on purpose. I decreased the cluster size, then the server size, then I configured too high request limits (later) to the containers so the whole cluster is dying right now.

If something like that happens you can see it in `kubectl get pods`:

```
→  7-kubernetes git:(main) k get pods
NAME                      READY   STATUS      RESTARTS      AGE
api-7db7644c75-27t8v      1/1     Running     0             38m
api-7db7644c75-klhwp      1/1     Running     0             39m
api-7db7644c75-s9g4l      1/1     Running     0             38m
api-7db7644c75-vfn9k      1/1     Running     0             39m
frontend-886966ccb-tqnjz  1/1     Running     0             39m
frontend-886966ccb-zdksb  1/1     Running     0             38m
migrate-zdrph             0/1     Completed   0             39m
nginx-5d655cc6cc-p96vt    0/1     Running     12 (60s ago)  39m
nginx-5d655cc6cc-wksvm    1/1     Running     9 (9m17s ago) 36m
scheduler-28178933-959bx  0/1     Completed   0             2m7s
scheduler-28178934-rmks4  0/1     Completed   0             67s
scheduler-28178935-m9jbr  0/1     Completed   0             7s
worker-57d95b4988-fzhnb   0/1     Running     29 (38m ago)  24h
worker-6cf9b8bf79-vb4bk   0/1     Running     0             39m
worker-7d4ccb4487-wxbcx   0/1     Running     1 (37m ago)   50m
```

There is a high number of restarts everywhere. And `READY 0/1` means that the container is not ready so the readiness probe failed. You can see that all of my workers failing the readiness probe.

Of course, on the DigitalOcean dashboard, you can always adjust the autoscaling settings:



Autoscaling pods

The cluster is already autoscaled in terms of nodes. However, we still have a fixed number of replicas. For example, this is the API deployment:

```
spec:
  replicas: 2
  containers:
    - name: api
      image: martinjoo/posts-api:latest
```

If you want your pods to run a dynamic number of replicas you should delete these `replicas` settings.

We need a new resource type called `HorizontalPodAutoscaler`. Each deployment you want to scale will have a dedicated HPA. This is what it looks like:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 4
  maxReplicas: 8
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
```

`scaleTargetRef` specifies which object (usually a deployment) you want to autoscale. `minReplicas` and `maxReplicas` define how many containers the given deployment should run at any given moment. By default, this will run 4 replicas and as the traffic grows the number of replicas will also increase to 8.

Finally, `metrics` define which metric it should autoscale on. In this case, it's the CPU utilization. So if the CPU is utilized 75% or more the HPA will spin up more pods. This sounds like a bad thing but remember that the cluster will also grow in the number of servers. HPA and auto-scaling cluster work together.

This comes directly from the DigitalOcean documentation:

> Cluster Autoscaling (CA) manages the number of nodes in a cluster. It monitors the number of idle pods, or unscheduled pods sitting in the `pending` state, and uses that information to determine the appropriate cluster size.
>
> Horizontal Pod Autoscaling (HPA) adds more pods and replicas based on events like sustained CPU spikes. HPA uses the spare capacity of the existing nodes and does not change the cluster's size.
>
> CA and HPA can work in conjunction: if the HPA attempts to schedule more pods than the current cluster size can support, then the CA responds by increasing the cluster size to add capacity. These tools can take the guesswork out of estimating the needed capacity for workloads while controlling costs and managing cluster performance.

You can also use memory as the base metric:

```
metrics:
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 70
```

However, in most cases, CPU is the better target to aim for.

Metrics server

Unfortunately, these settings won't work properly. If you run the

```
kubectl get hpa
```

command you will see something similar to this:

```
→  7-kubernetes git:(main) k get hpa
NAME             REFERENCE               TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
api-hpa          Deployment/api          <unknown>/75%   2         8         4          29h
frontend-hpa     Deployment/frontend     <unknown>/75%   2         4         2          29h
nginx-hpa        Deployment/nginx        <unknown>/75%   2         4         2          29h
worker-hpa       Deployment/worker       <unknown>/75%   2         4         2          29h
```

Kubernetes is unable to get the current CPU utilization so it doesn't know when to scale up or down. It defaulted to 4 replicas.

If I run the `kubectl describe hpa api-hpa` command I see the following error message:

```
Conditions:
  Type            Status  Reason                   Message
  ----            ------  ------                   -------
  AbleToScale     True    SucceededGetScale        the HPA controller was able to get the target's current scale
  ScalingActive   False   FailedGetResourceMetric  the HPA was unable to compute the replica count: failed to get cpu utilization: unable to get metrics for re
source cpu: unable to fetch metrics from resource metrics API: the server could not find the requested resource (get pods.metrics.k8s.io)
```

`unable to get metrics for resource cpu` indicates something is missing. We actually need to install the `metrics-server` component which can communicate with Kubernetes and it can gather resource information from the nodes.

Download this yaml file https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml and save it to `infra/k8s/metrics-server/components.yml`

Then apply it:

```
kubectl apply -f infra/k8s/metrics-server
```

To verify the installation run:

```
kubectl top nodes
```

It should give you metrics about your nodes:

```
→  7-kubernetes git:(main) ✗ k top nodes
NAME                      CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
pool-ftz1m06x3-y36bt      110m         5%     1158Mi          73%
pool-ftz1m06x3-y36j7      118m         6%     1530Mi          97%
```

If I now run the `kubectl get hpa api-hpa` command again I see this:

```
→  7-kubernetes git:(main) ✗ k get hpa api-hpa
NAME      REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
api-hpa   Deployment/api    10%/75%   2         8         2          30h
```

Now it can detect the current CPU utilization and it can scale down to 2 replicas.

Rolling update config

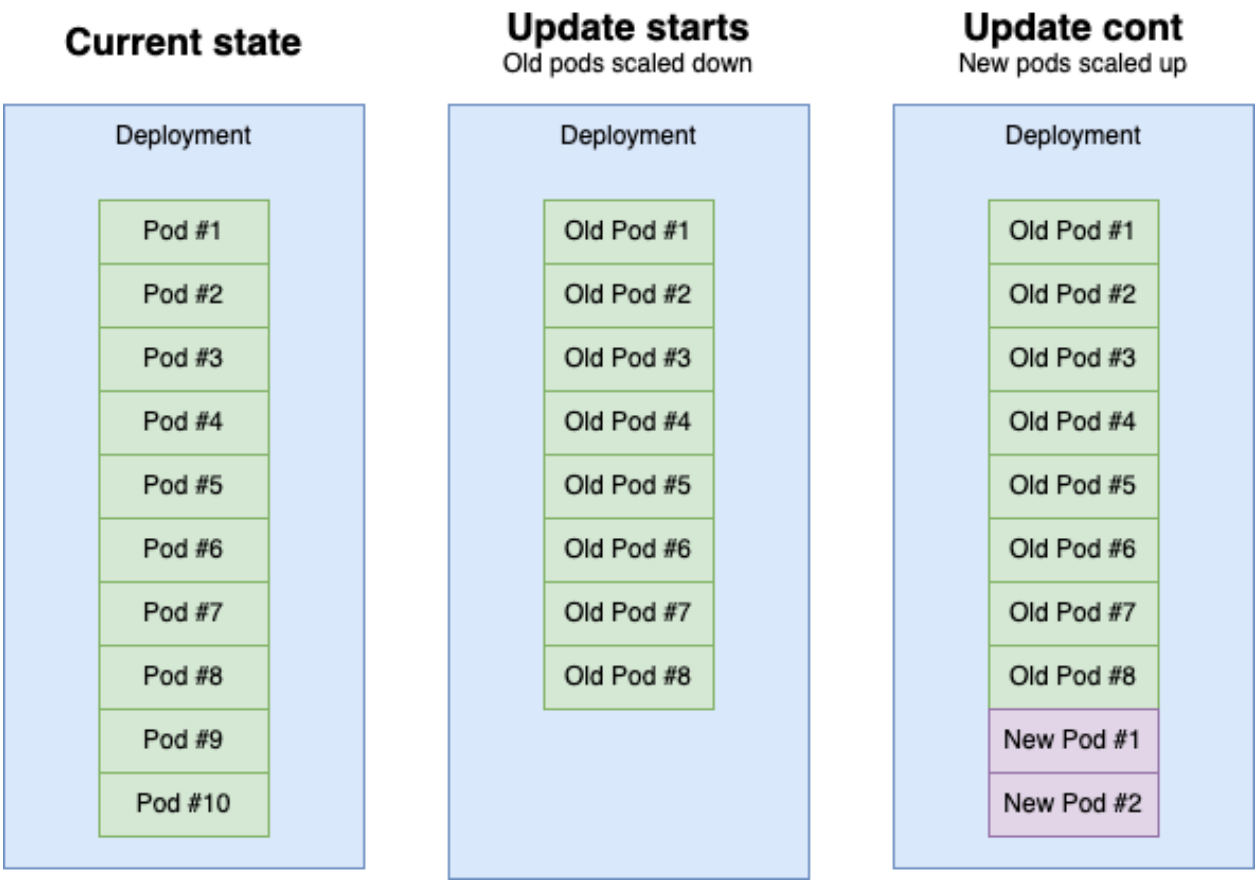
maxUnavailable

Now that we have proper autoscaling it's time to configure how our pods should be updated. It's pretty similar to Docker Swarm.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 2
```

There's a `strategy` section inside `spec`.

`maxUnavailable` specifies the maximum number of pods that can be unavailable during the update process. This is what it looks like:



This is just a logical representation of an update. It's more complicated in the real world

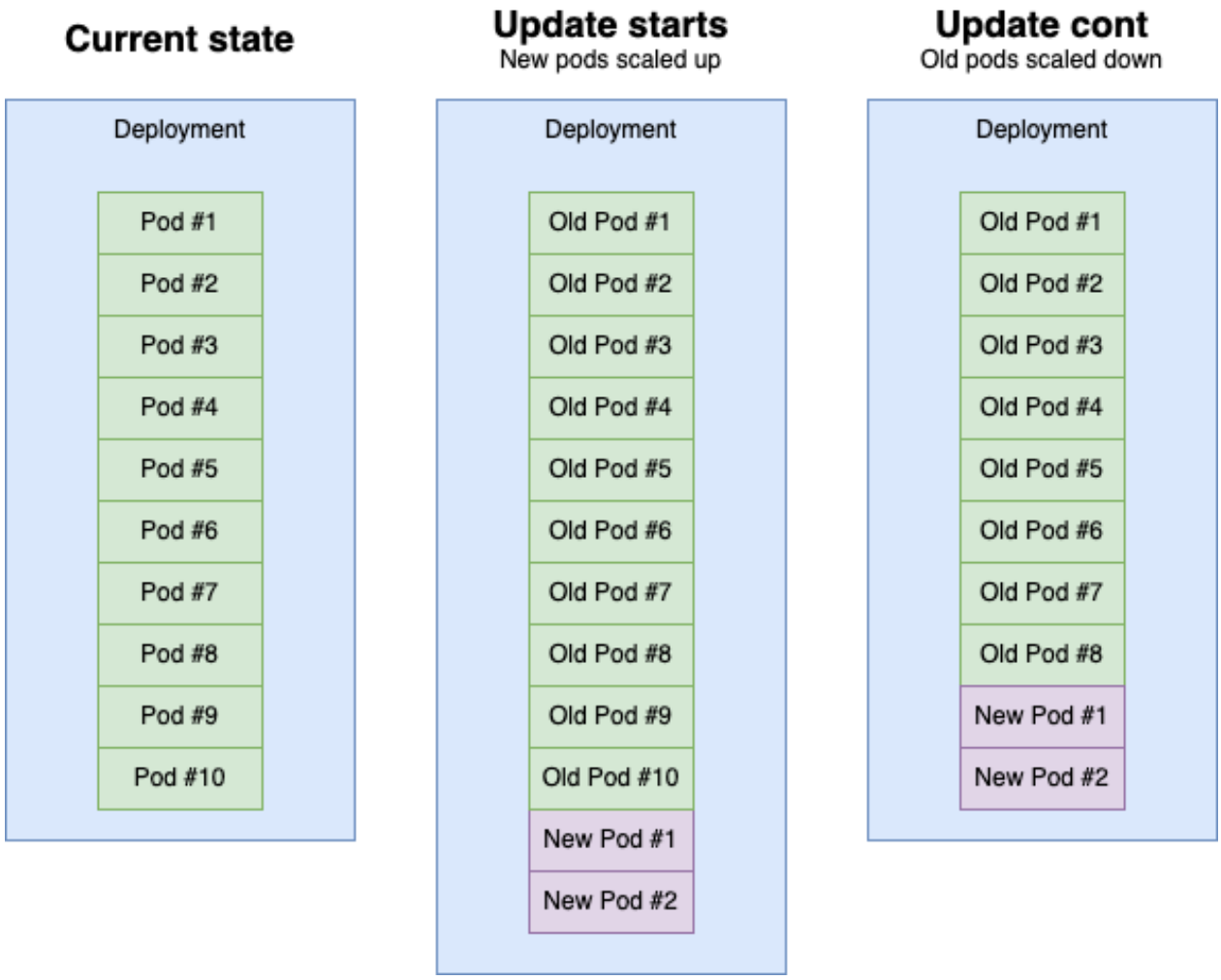
If you have 10 replicas and you set `maxUnavailable` to 2 it means that at least 8 replicas have to run during the whole update. Of course, in a production system, it's hard to define how many replicas you want. Fortunately, it doesn't need to be an absolute number. It can be a percentage as well:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 20%
```

In this example, with 10 pods, 20% means the same thing as 2.

maxSurge

`maxSurge` on the other hand specifies the maximum number of pods that can be created over the desired number of pods.



This is just a logical representation of an update. It's more complicated in the real world

If you have 10 replicas and you set `maxSurge` to 2 it means that a maximum of 12 replicas can run during the update. It also supports percentage values:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 20%
```

It's hard to say what numbers are great because it varies from project to project.

- `maxSurge` means your servers might experience an increased number of pods during updates. Which may or may not be a problem for you. It probably won't cause any problem if you run the "usual" Laravel-related containers. Having this number around 20--30% sounds like a safe bet. The default value is 25% which is a reasonable default.
- `maxUnavailable` means there can be a decrease in pod replicas during updates. Unless you run some pretty mission-critical software I think it's not a real problem. Once again 20--30% sounds like a safe bet. The default value is 25% as well.

Resource requests and limits

In Kubernetes, we have the ability to set the minimum and maximum amount of resources that a pod can use on the host machine. Here's the configuration:

```
spec:
  containers:
    - name: api
      image: martinjoo/posts-api:latest
      imagePullPolicy: Always
      resources:
        requests:
          cpu: "100m"
          memory: "64Mi"
        limits:
          cpu: "250m"
          memory: "256Mi"
```

The `memory` is quite simple: the API needs at least 64MB at any given moment but it cannot use more than 256MB.

The CPU is a bit trickier. `100m` reads as "100 milli cores" or "100 milli CPUs". And it really is just a fancy way of saying: "0.1 CPU cores". One core is 1000m

The important thing is that it always refers to an **absolute** number. So 100m means the same computing power on a 1-core machine, a 2-core machine, or a 16-core machine. It's always 0.1 CPU cores.

A resource **request** means that the container is always guaranteed to have at least the requested amount of resources. Of course, it can use more than the request.

A resource **limit** prevents a container to use more resources than the limit itself. What happens when a container exceeds the limit?

- If the container exceeds the **memory** limit it is killed with an out-of-memory error and then it is restarted.
- If the **cpu** limit is exceeded the container won't be terminated but it's now limited. It's throttled. When CPU throttling is enforced, the CPU usage of the container is limited to a certain percentage of available CPU time. In practice all it means is that the container is going to be super slow.

Limiting resources can be appealing since they prevent containers from using all the resources and potentially bringing the whole server down. Yes, it sounds great in theory but it can cause some problems as well. Let's discuss all the possible variations.

No limits, no requests

Assumption: none of the pods has any limits or requests.

Imagine that a worker starts some pretty heavy process. For example, it transcodes videos with ffmpeg. There are no CPU limits so it uses 100% of it. Since there are no CPOU requests either, other containers are not guaranteed to get any CPU at all. So if there's an nginx container on the same node and requests come in, they will be served pretty slowly since the worker uses 100% of the CPU.

The result is CPU starvation. Containers didn't get CPU time because of a very hungry process.

There are limits

Assumption: some (or all) containers have limits

So the server went down because of the worker and you set some CPU limit for the container. Great. It starts another video transcoding process. 4 in the morning on Sunday. The nginx container has 1 request/hour traffic but your worker is only able to use 50% of the CPU because you limited it. The other 50% is completely idle.

The result is an idle CPU. Containers don't use the available resources.

No limits, only requests

Assumption: none of the pods has any limits, but each of them has requests

One way to avoid idle CPU time is to add requests to every pod. Since requests are **guaranteed** we don't actually need limits to avoid CPU starvation.

Your worker starts transcoding videos. It uses 100% of the CPU. And now traffic starts to come into your nginx container. No problem. Since nginx has a CPU request the worker gets less and less CPU while nginx gets more and more of it. But we won't throttle the worker either since it also has a request.

The result is: everyone gets their CPU time. No resources are wasted, there's no idle CPU time.

So my advice is to avoid limits and use requests. If you'd like to learn more about the topic an amazing article with top-notch analogies.

When defining requests there are two important things to keep in mind:

- A request is applied to each replica. If you define 256MB memory and you want to run 8 replicas that's already 2GB of memory requested.
- If you request too much of a resource your pod won't be scheduled at all. Here's a screenshot where my worker requested 8GB of RAM on a 4GB machine:

```
Events:
  Type     Reason             Age                From               Message
  ----     ------             ----               ----               -------
  Warning  FailedScheduling   80s                default-scheduler  0/3 nodes are available: 3 Insufficient memory. preemption: 0/3 nodes are available: 3 No
preemption victims found for incoming pod..
  Warning  FailedScheduling   49s (x2 over 51s)  default-scheduler  0/3 nodes are available: 3 Insufficient memory. preemption: 0/3 nodes are available: 3 No
preemption victims found for incoming pod..
  Normal   NotTriggerScaleUp  61s                cluster-autoscaler  pod didn't trigger scale-up: 1 Insufficient memory
```

Here are the limits I'm using for this project.

API:

```
resources:
  requests:
    cpu: "200m"
    memory: "64Mi"
```

The `memory_limit` in `php.ini` is set to 128MB so 64MB as a minimum should be more than enough. And as we discussed in the PHP-FPM optimization chapter, PHP scripts are usually not CPU-heavy at all so 200m is a great start. You can always change and fine-tune these numbers to your own need.

nginx:

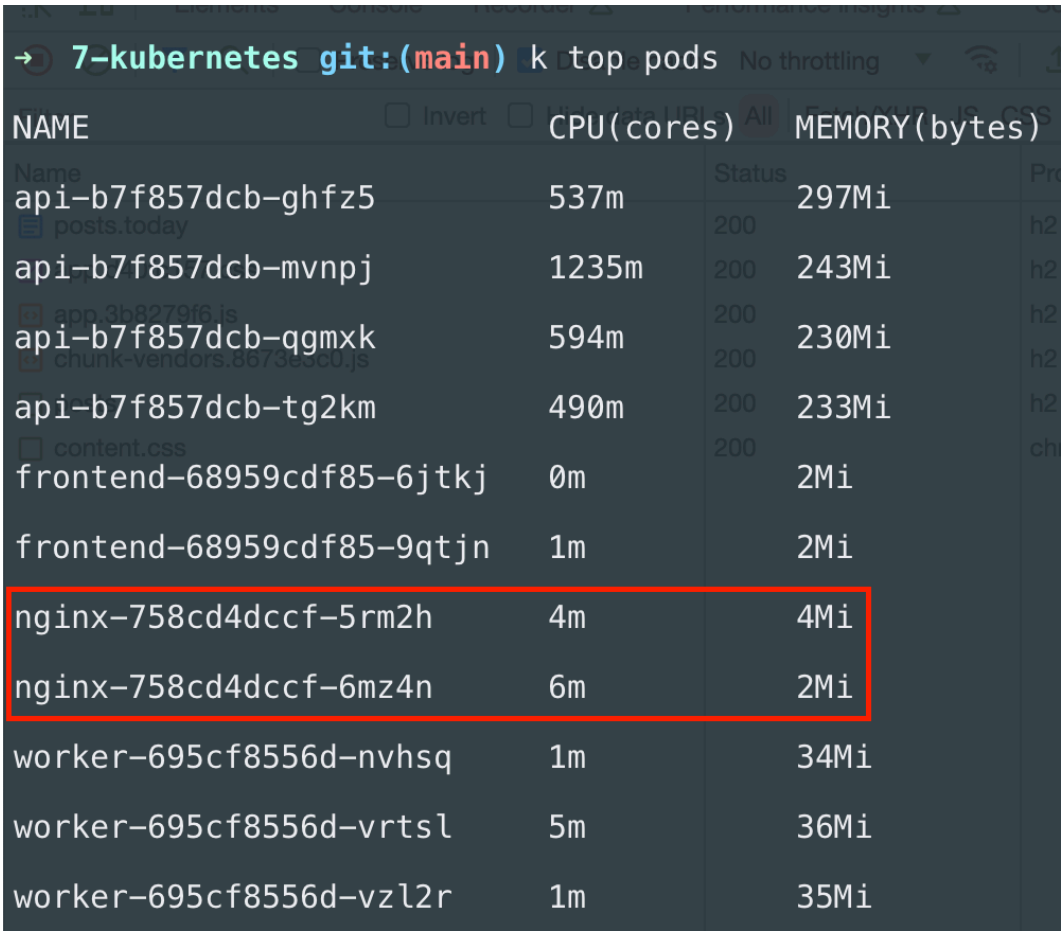
```
resources:
  requests:
    cpu: "50m"
    memory: "16Mi"
```

nginx is one of the most performant software I've ever used. Maybe `ls -la` is the only one that uses fewer resources. The resource consumption of nginx won't be a problem. It only needs a minimum amount of CPU and memory to do its job.

I ran a quick `ab` benchmark that sends 250 requests to the API. 50 of those are concurrent.

```
ab -c 50 -n 250 https://posts.today/api/load-test/
```

This was the memory/CPU consumption of nginx:

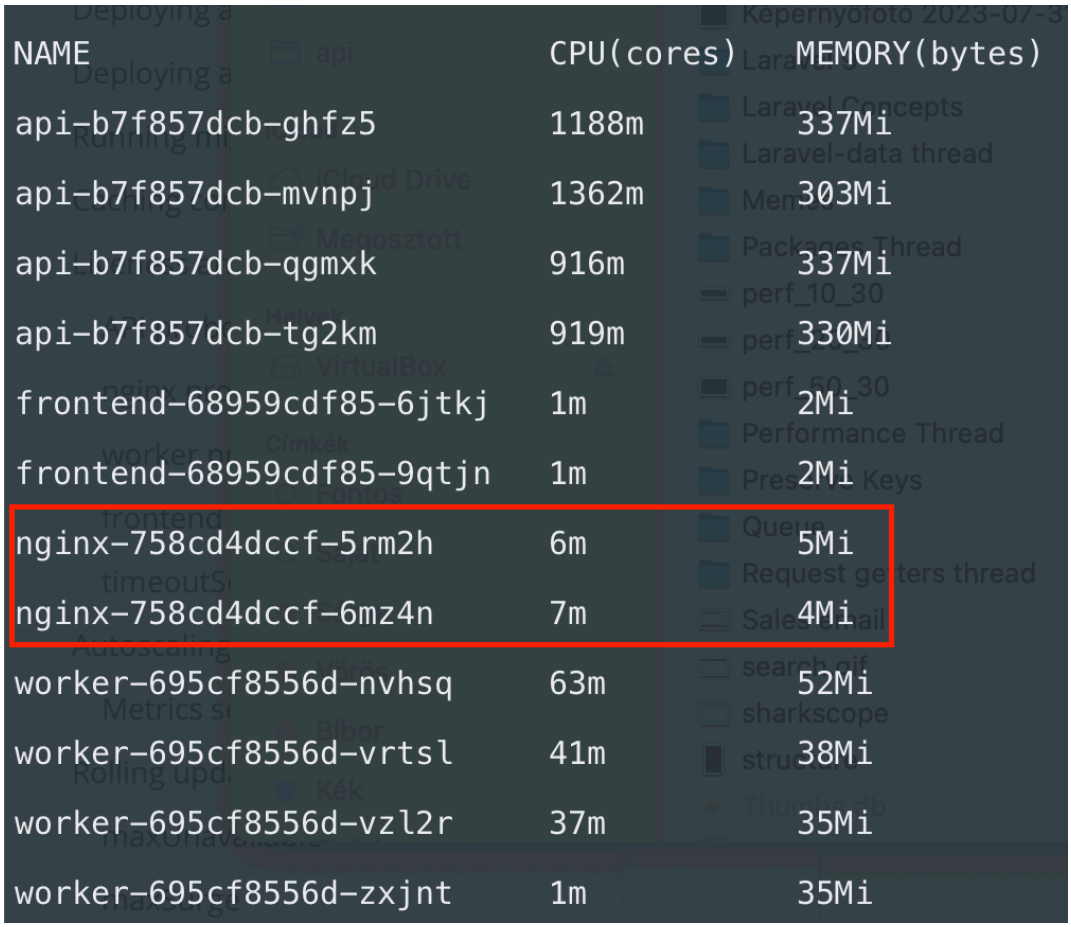


The two containers consumed 6MB of RAM and they used 10m CPU. just to be clear, 10m means 0.01 cores or 1% of the CPU.

Let's increase the numbers:

```
ab -c 250 -n 2000 https://posts.today/api/load-test/
```

nginx just doesn't care about my numbers:



```
NAME                        CPU(cores)   MEMORY(bytes)
api-b7f857dcb-ghfz5         1188m        337Mi
api-b7f857dcb-mvnpj         1362m        303Mi
api-b7f857dcb-qgmxk         916m         337Mi
api-b7f857dcb-tg2km         919m         330Mi
frontend-68959cdf85-6jtkj   1m           2Mi
frontend-68959cdf85-9qtjn   1m           2Mi
nginx-758cd4dccf-5rm2h      6m           5Mi
nginx-758cd4dccf-6mz4n      7m           4Mi
worker-695cf8556d-nvhsq     63m          52Mi
worker-695cf8556d-vrtsl     41m          38Mi
worker-695cf8556d-vzl2r     37m          35Mi
worker-695cf8556d-zxjnt     1m           35Mi
```

This time it increased to 13m CPU and 9MB of RAM. I literally have no idea how they do this...

How can an API container use 337MB of RAM if the `memory_limit` is 128MB? There are 15 FPM processes running inside each container. They process requests concurrently. So 337/15 = 22MB of RAM for each PHP request. The 4 PHP containers processed 15*4 = 60 concurrent requests at the same time.

frontend:

```yaml
resources:
  requests:
    cpu: "100m"
    memory: "32Mi"
```

Frontend is also an nginx container but I gave it a bit more CPU and memory since it can serve larger files (which requires memory) and it uses GZIP which can use some CPU as well.

Worker

```yaml
resources:
  requests:
    cpu: "100m"
    memory: "64Mi"
```

Worker is similar to the API but usually it has less load and it's not that "important" so I gave it less than the API. "Not that important" means it's okay to give it fewer resources because the only consequence is slower background jobs. Meanwhile, insufficient allocation of resources to the API results in slower requests. Requests are sync and user-facing so they usually have priority over queue jobs.

Health check pods

If you haven't read the health check monitors chapter, please do. As a reminder, I'm going to use the `spatie/laravel-health` package to monitor the health of the application. The configuration looks like this:

```
<?php

namespace App\Providers;

class HealthCheckServiceProvider extends ServiceProvider
{
  public function boot(): void
  {
    Health::checks([
        UsedDiskSpaceCheck::new(),
        DatabaseCheck::new(),
        CpuLoadCheck::new()
          ->failWhenLoadIsHigherInTheLast5Minutes(2)
          ->failWhenLoadIsHigherInTheLast15Minutes(1.75),
        DatabaseConnectionCountCheck::new()
          ->warnWhenMoreConnectionsThan(50)
          ->failWhenMoreConnectionsThan(100),
        DebugModeCheck::new(),
        EnvironmentCheck::new(),
        RedisCheck::new(),
        RedisMemoryUsageCheck::new(),
        QuerySpeedCheck::new(),
    ]);
  }
}
```

Everything is explained in the linked chapter. Now we'll only talk about Kubernetes-specific things.

The goal is to run the `posts-health-check` container on every node. It runs the `health:check` command every minute and reports any issues it finds. With swarm (and compose) we solved this by configuring the restart policy of the container:

```
health-check:
  image: martinjoo/posts-health-check:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/health-check.sh"
  deploy:
    mode: global
    restart_policy:
      condition: any
      delay: 60s
      window: 30s
```

`global` mode means it runs in one replica on every node. The script runs, the container exits, and then it is restarted by Swarm after 60 seconds. So it runs every minute.

It's an easy solution, however, in Kubernetes, we cannot configure a restart delay. It has a default behavior:

> *After containers in a Pod exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, ...), that is capped at five minutes. Once a container has executed for 10 minutes without any problems, the kubelet resets the restart backoff timer for that container. - Docs*

So we need another solution.

`CronJob` can run a container every minute just as we've seen with the scheduler. But unfortunately, they can be placed on every node in the cluster. They just run on a random node and that's it. So it's not quite what we want.

First, we need something that can run as a `global` container in Swarm. Fortunately, k8s has a similar resource, it's called `DaemonSet`. Its purpose is to run a container on every node.

It's very easy to configure:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: health-check
spec:
  selector:
    matchLabels:
      name: health-check
  template:
    metadata:
      labels:
```

```
        name: health-check
    spec:
      containers:
        - name: health-check
          image: martinjoo/posts-health-check:latest
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          envFrom:
            - configMapRef:
                name: posts
            - secretRef:
                name: posts
```

It's like a standard deployment but with `kind: DaemonSet`. For this container, I set a limit, because I don't want a health check container to bring down my servers if something is terribly wrong.

The spec has one more thing I haven't shown you:

```
spec:
  tolerations:
    # these tolerations are to have the daemonset runnable on control plane
nodes
    # remove them if your control plane nodes should not run pods
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
    - key: node-role.kubernetes.io/master
      operator: Exists
      effect: NoSchedule
  containers:
```

It's required to run the pod on control plane nodes as well. In k8s the control plane node is the "leader" or "master" node. It stores state information about the whole cluster, schedules pods, etc.

Now we configured a pod that runs on every node. But we still have a problem. By default, a `ReplicaSet` (just like a `Deployment`) expects a long-running process in the container. The `health:check` command is a short lived one. It runs, prints out some informations and then it exits. Because of that k8s thinks that something is wrong inside the pod and restarts it. Then the container runs and it exits. k8s thinks that something is wrong inside the pod and restarts it.Then the container... You got the point. This is the famous `CrashLoopBackOff` state. When a container starts, exits, starts, exits.

Since k8s does not provide restart delays, we need to change the `health-check.sh` script a little bit:

```
#!/bin/bash

while true; do
    nice -n 10 php /usr/src/artisan health:check
    sleep 60
done
```

It should work now because the infinite loop creates a long-running process so k8s won't restart the container all the time. Now we have the health check pods up and running.

Exposing the application

Now that we have:

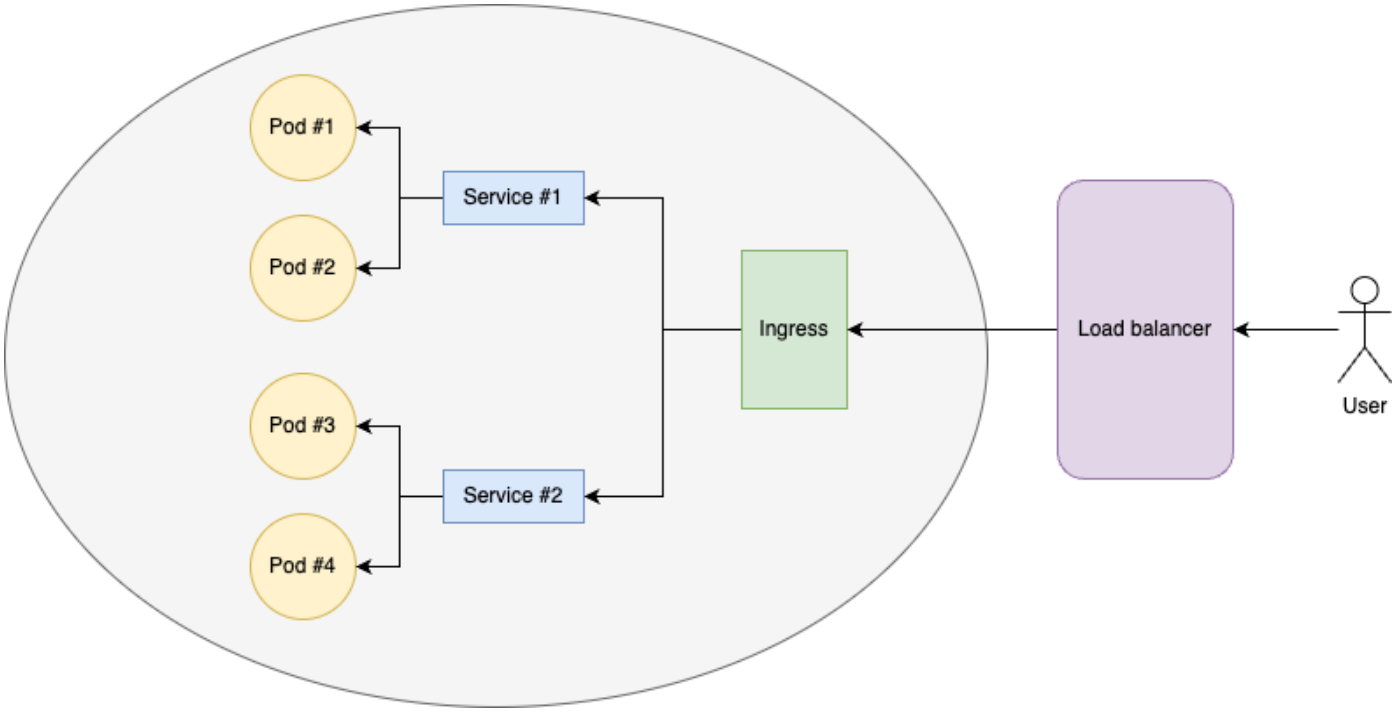
- Autoscaling pools
- Autoscaling pods
- Liveness probes
- Readiness probes
- Health check containers
- Highly available databases

so a very stable, production-ready cluster overall. Let's ship it to the world!

Unfortunately, it's going to be one of the most confusing chapters of this book, I believe.

Ingress

The first component we need is an ingress:



Try to not worry about the load balancer for now. Let's just say there's incoming traffic to the cluster.

Ingress is a resource that allows you to manage external access to services within your cluster. It acts as a reverse proxy, routing incoming HTTP and HTTPS traffic to the appropriate services based on the rules defined.

So every request first comes into the ingress. It looks at it and decides which service it should route the request to. It's a reverse proxy.

For example, look at these two requests:

- `GET https://posts.today/index.html`
- `GET https://posts.today/api/posts`

The first one should clearly go to the frontend service but the second one should go to the nginx service (which provides access to the Laravel API). The ingress will route these requests to the right services.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: posts-ingress
  annotations:
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  ingressClassName: nginx
```

```
  rules:
    - http:
        paths:
          - path: /api(/|$)(.*)
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
```

If you ever wrote an nginx reverse proxy it looks very familiar. It defines two paths or routes:

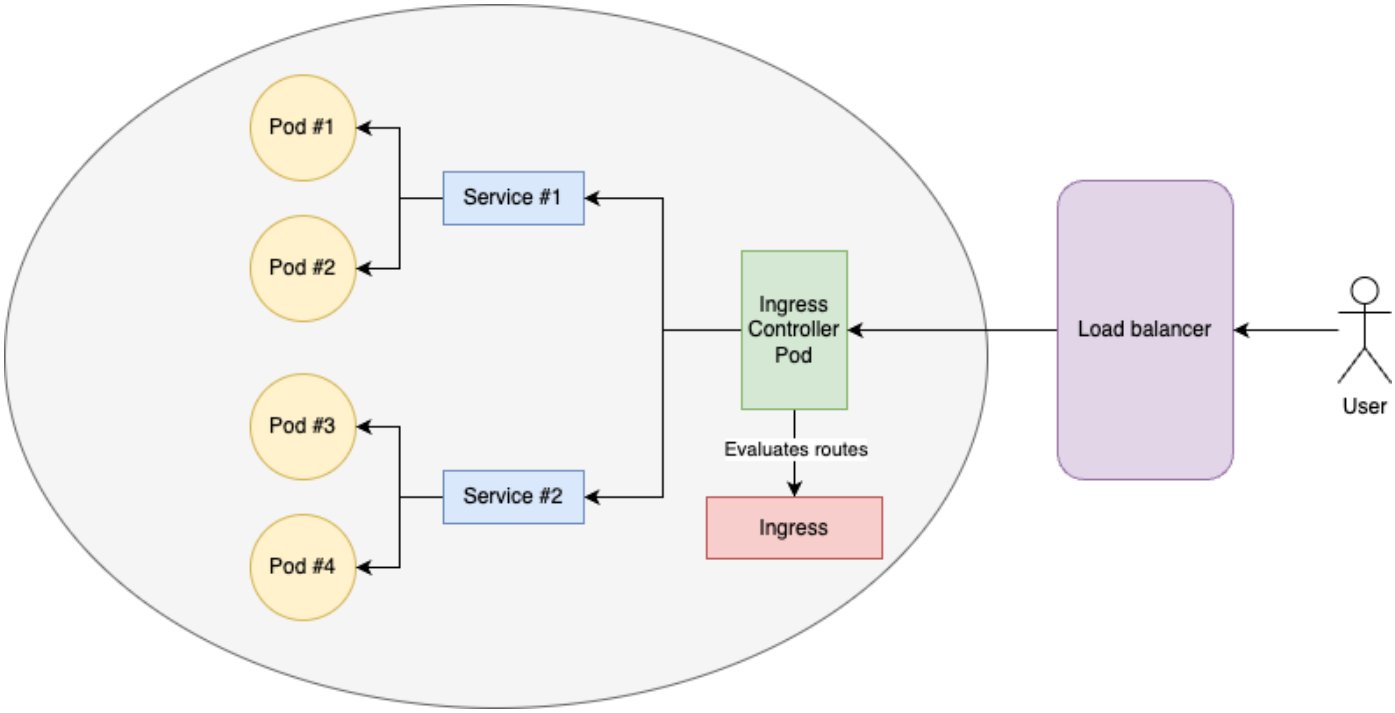
- `/api(/|$)(.*)` matches requests such as `/api/posts`
- `/` matches everything else

This way if you request the home page you are proxied to the `frontend-service`, but if the frontend sends an API request it gets proxied to the `nginx-service`.

Ingress controller & load balancer

Now we have routing rules to act as an entry point to the application. However, an Ingress is not like a deployment. If you apply the config it won't run pods that somehow can accept traffic and apply the routing rules. But ultimately, this is what we want. We want a running container that evaluates the ingress rules and distributes the traffic among our services.

An ingress controller is a component that does exactly that. So this is a more accurate representation:



It is called a controller because it reads the routes and calls the appropriate services. Just like a Controller in Laravel.

What's inside the ingress controller? It's actually a proxy so it can use nginx, Traefik, HAProxy, etc. Fortunately, there are lots of pre-built ingress controller implementations from these vendors. Here's the full list of available vendors. We're going to use nginx.

```
mkdir ingress-controller
wget https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.1/deploy/static/provider/do/deploy.yaml -O ingress-controller/controller.yml
```

If you look inside the new file you'll find a `Deployment` with the of `ingress-nginx-controller`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ingress-nginx-controller
  namespace: ingress-nginx
```

In the `spec` you can see it runs the following image:

```
spec:
  dnsPolicy: ClusterFirst
  containers:
    - name: controller
      image: k8s.gcr.io/ingress-nginx/controller:v1.8.1
      imagePullPolicy: IfNotPresent
```

This will run the actual ingress controller pod that has nginx in it.

If you search for the term `type: LoadBalancer` you'll find this:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.beta.kubernetes.io/do-loadbalancer-enable-proxy-protocol: 'true'
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: http
      appProtocol: http
    - name: https
      port: 443
```

```
    protocol: TCP
    targetPort: https
    appProtocol: https
```

Hmm. It seems like a real load balancer that accepts traffic on ports 80 and 443. But what kind of load balancer? Do I need to add another server or something like that? These are the questions I asked first. And then I looked into the Kubernetes docs:

> *On cloud providers which support external load balancers, setting the `type` field to `LoadBalancer` **provisions a load balancer for your Service**. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's `.status.loadBalancer` field. Docs*

Please notice the sentence **provisions a load balancer for your Service**.

But how? Check out the link again where you downloaded the YAML:

raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.1/deploy/static/**provider/do**/deploy.yaml

"do" stands for DigitalOcean. If you check out the load balancer service one more time you'll notice this:

```
metadata:
  annotations:
    service.beta.kubernetes.io/do-loadbalancer-enable-proxy-protocol: 'true'
```

It's just an annotation, a metadata that doesn't look too important, but the `service.beta.kubernetes.io/do-loadbalancer-enable-proxy-protocol` is so specific it must mean something. Yes, it does.

The managed cluster on DigitalOcean looks for annotations like this one and we can configure our balancer with these. For example, if you want your load balancer to use keep-alive connections you turn it on with:

```
metadata:
  annotations:
    service.beta.kubernetes.io/do-loadbalancer-enable-proxy-protocol: 'true'
    service.beta.kubernetes.io/do-loadbalancer-enable-backend-keepalive:
'true'
```

You can check out the full list of annotations here.

So let's summarize what is happening:

- We downloaded a YAML file which was a DigitalOcean-specific ingress controller definition

- If you run your cluster on a cloud provider, the `LoadBalancer` service instructs the provider to create a new load balancer for your cluster
- When you apply the file **a load balancer will be created** on your account
- It'll be connected to the cluster
- It accepts traffic and forwards it to the ingress controller which reads the ingress configuration (routing) and sends the traffic to our services

I know it's hard to process all these at first. Give it a few days, and you'll understand it better. For example, here's pretty good video on the topic.

And now, let's apply everything:

```
kubectl apply -R -f infra/k8s
```

In the downloaded YAML file we've seen a namespace called `ingress-nginx`. This is how you can get pods from a namespace other than the default:

```
kubectl get pods -n ingress-nginx
```

It returns the pod that runs the controller:

```
→  7-kubernetes git:(main) k get pods -n ingress-nginx
NAME                                       READY   STATUS    RESTARTS   AGE
ingress-nginx-controller-65bc6f599f-4sxvp   1/1     Running   0          20h
→  7-kubernetes git:(main)
```

We can also get the logs:

```
kubectl logs ingress-nginx-controller-65bc6f599f-4sxvp -n ingress-nginx
```

And you can see the access logs from my previous load test:

```
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.148 [default-nginx-service-80] [] 10.244.0.183:8
0 949 2.148 200 d2a70907ee393755187767bd6592e9ac
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.149 [default-nginx-service-80] [] 10.244.0.183:8
0 949 2.152 200 c93fc5688a3e654be844fa404fae2991
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.157 [default-nginx-service-80] [] 10.244.1.25:80
 949 2.156 200 68e14e3f199694d29e79628bf566e677
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.348 [default-nginx-service-80] [] 10.244.0.183:8
0 949 2.348 200 66e7c5d246fe6c3e4c2627511deb3322
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.380 [default-nginx-service-80] [] 10.244.0.183:8
0 949 2.380 200 77bb1bab55d7edb807dedf34d0e7106a
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.719 [default-nginx-service-80] [] 10.244.0.183:8
0 949 2.716 200 40d2cee39df874dc0f00e7f9fc3ce2cc
84.0.18.23 - - [30/Jul/2023:23:40:24 +0000] "GET /api/load-test/ HTTP/1.0" 200 937 "-" "ApacheBench/2.3" 93 2.801 [default-nginx-service-80] [] 10.244.0.183:8
0 949 2.800 200 34f0e605ea6ad55026ff3c5c25b6c656
```

We just confirmed that every request goes through the ingress controller. You can also see the `default-nginx-service-80` which is the target service based on the route `/api/load-test`

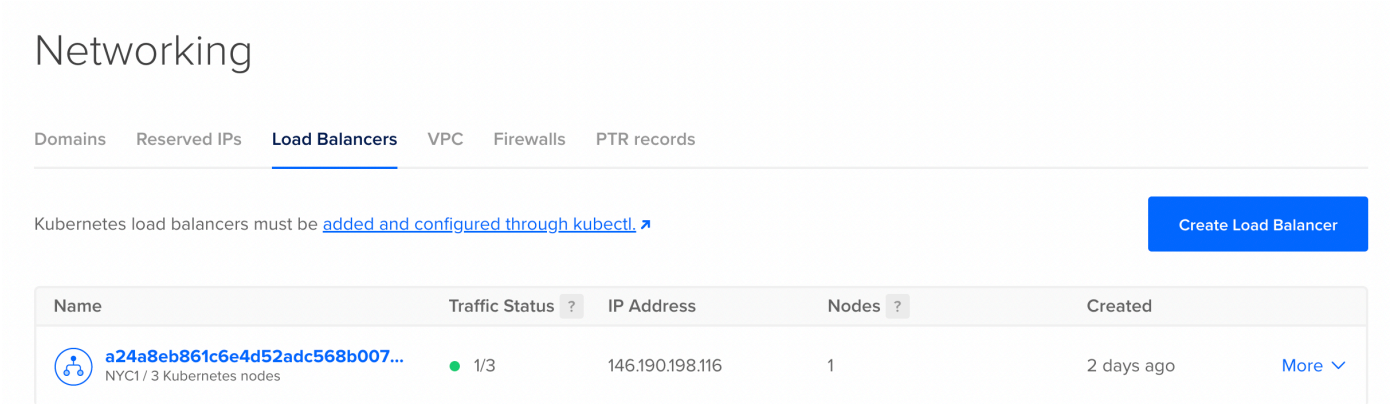
But more important we can get the load balancer service:

```
kubectl get service -n ingress-nginx
```

```
→  7-kubernetes git:(main) k get service -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller             LoadBalancer   10.245.36.113    posts.today   80:30858/TCP,443:32011/TCP   2d5h
ingress-nginx-controller-admission   ClusterIP      10.245.218.32    <none>        443/TCP                      2d5h
```

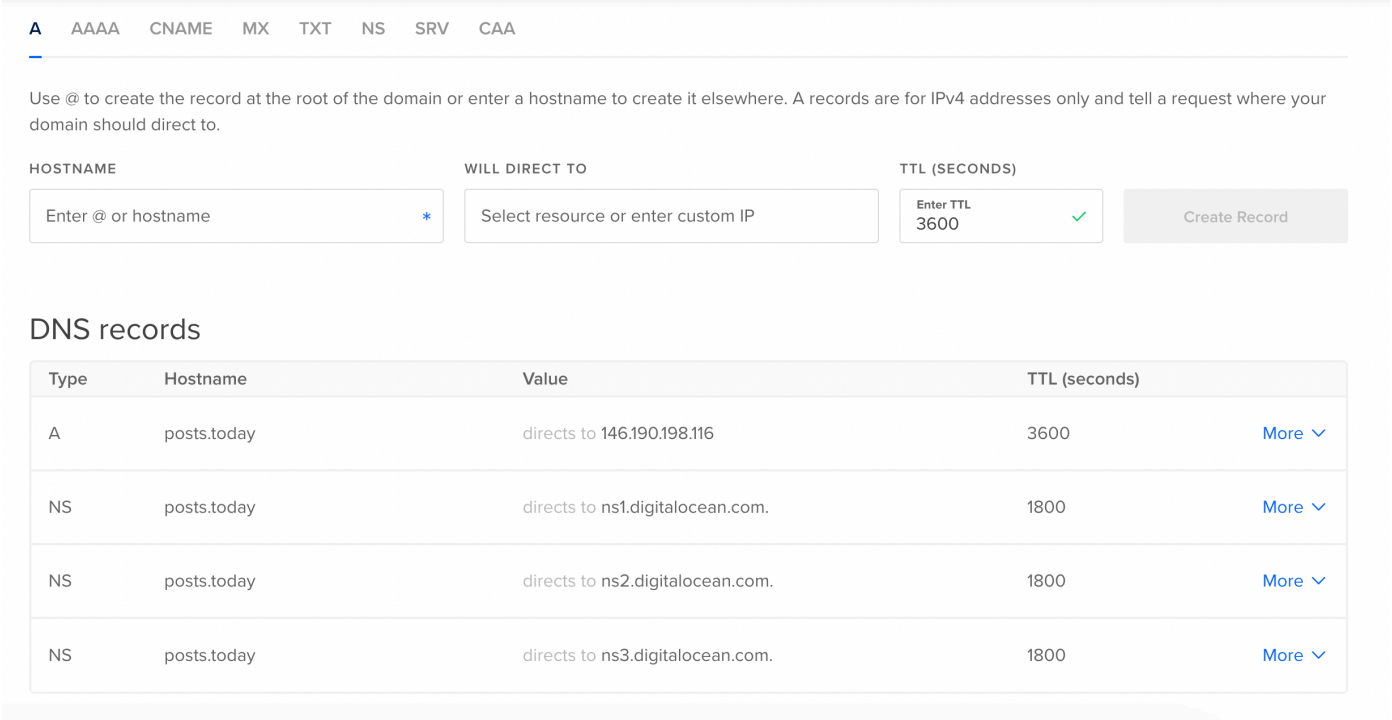
And now you finally have **an external IP**. It can take a few minutes but should see a valid IP address in the `EXTERNAL-IP` column. And you should be able to access the application using this IP.

Of course, this IP address belongs to your shiny new load balancer on DigitalOcean:



Domain & HTTPS

The last thing we need to take care of is a domain name and HTTPS. You can buy a domain from Namecheap (or any other provider you like) and then set up an `A` record that points to the load balancer and the necessary NS record provided by DigitalOcean



If you're not familiar with these DNS records check out the "Domains and HTTPS with nginx" chapter in the first part of the book. I already described them in great detail.

The next step is to add a new annotation to the ingress controller (in the `infra/k8s/ingress-controller/controller` file) that sets the hostname:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.beta.kubernetes.io/do-loadbalancer-enable-proxy-protocol: 'true'
    service.beta.kubernetes.io/do-loadbalancer-hostname: 'posts.today'
```

You must do this step otherwise HTTPS won't work properly. It's a well-known DigitalOcean issue. You can read more about it here.

Now you should be able to access your cluster via the domain. Of course, it's not safe because it doesn't have HTTPS yet.

Fortunately, there's a 3rd party component for Kubernetes called cert-manager. You can check out the docs here. It can create certificates and issuers as resources. For example, certificates become `Secrets` that can be used by an issuer. And of course, it also helps us renew these certs.

Here's how you can install it:

```
mkdir infra/k8s/cert-manager
wget https://github.com/jetstack/cert-manager/releases/download/v1.5.3/cert-manager.yaml -O infra/k8s/cert-manager/manager.yml
```

You can apply the changes with `kubectl apply`. After that you should see some new pods in the `cert-manager` namespace:

```
→ 7-kubernetes git:(main) k get pods -n cert-manager
NAME                                       READY   STATUS    RESTARTS        AGE
cert-manager-8457df4d9c-zjz26              1/1     Running   1 (3h40m ago)   22h
cert-manager-cainjector-7d7c8b7fbb-nz6lj   1/1     Running   0               22h
cert-manager-webhook-698b6fbdc7-qf5wc      1/1     Running   0               22h
```

Next, we need to create an issuer for the certificate. An issuer represents a Certificate Authority (CA) which I also described in the mentioned chapter. We're going to use Let's Encrypt which is a free CA.

We're going to create two different issuers. One for testing purposes and for production. Let's Encrypt provides a staging endpoint that has no rate limits and can be used during development. The production endpoint has a rate so don't use it while setting up your cluster.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: posts-staging
spec:
  acme:
    email: <your-email>
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: posts-staging-key
    solvers:
      - http01:
          ingress:
            class: nginx
---
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
```

```
  name: posts-production
spec:
  acme:
    email: <your-email>
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: posts-production-key
    solvers:
      - http01:
          ingress:
            class: nginx
```

They specify the ACME server with the production and staging endpoints. cert-manager will automatically create ACME account private keys. In the `privateKeySecretRef` property we can define the `Secret` it'll create. You can read more about the ACME issuer type here.

By the way, `ClusterIssuer` is not a native resource type provided by Kubernetes. It's a Custom Resource Definition provided by cert-manager and is defined in `manager.yml` we just downloaded. This is why we need to define `apiVersion` for every resource we create. Custom resources come from custom API versions such as `cert-manager.io/v1` which is also defined in `manager-yml`. You can run `kubectl get crd` to get custom resource definitions.

If you apply the issuers you should see the keys as secrets:

```
kubectl get secrets -n cert-manager
```

```
→  7-kubernetes git:(main) k get secrets -n cert-manager
NAME                          TYPE     DATA   AGE
cert-manager-webhook-ca       Opaque   3      2d6h
posts-production-key          Opaque   1      2d5h
posts-staging-key             Opaque   1      2d5h
```

If you run a `describe` command you can see it's the same `Opaque` type we used earlier:

```
kubectl describe secrets posts-production-key -n cert-manager
```

```
→  7-kubernetes git:(main) k describe secrets posts-production-key -n cert-manager
Name:         posts-production-key
Namespace:    cert-manager
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
tls.key:  1679 bytes
```

And the last thing to do is adding the domain and issuer to the ingress (router):

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: posts-ingress
  annotations:
    nginx.ingress.kubernetes.io/use-regex: "true"
    cert-manager.io/cluster-issuer: "posts-production"
spec:
  tls:
    - hosts:
        - posts.today
      secretName: posts-tls
  ingressClassName: nginx
  rules:
    - host: posts.today
      http:
        paths:
          - path: /api(/|$)(.*)
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80
          - path: /
```

```
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
```

First, there's a new annotation for cert-manager:

```
cert-manager.io/cluster-issuer: "posts-production"
```

It tells to cert-manager that ingress uses the `posts-production` issuer.

If you're testing your setup use the staging issuer.

Next, the `spec` section contains a `tls` with the domain and a secret name:

```
tls:
  - hosts:
      - posts.today
    secretName: posts-tls
```

This tells the Ingress controller to secure the channel from the client to the load balancer using TLS. The TLS will be created by cert-manager.

And I also changed the `rules` section to include the domain:

```
rules:
  - host: posts.today
    http:  ...
```

If you apply the changes you can see there's a new certificate resource:

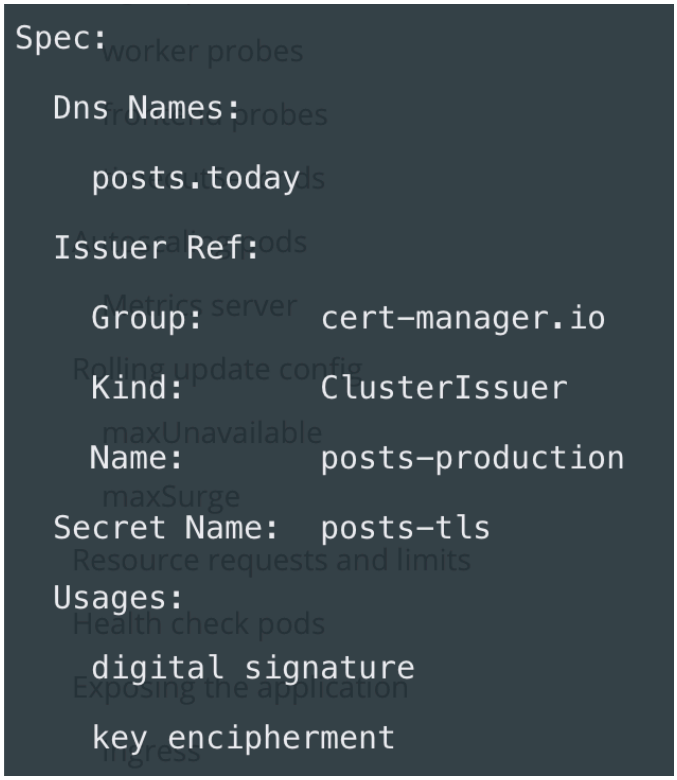
```
kubectl get certificateS
```

```
→  7-kubernetes git:(main) k get certificateS
NAME        READY   SECRET      AGE
posts-tls   True    posts-tls   2d6h
```

If you describe the certificate running:

```
kubectl describe certificate posts-tls
```

In the `spec` section you'll see it's created by cert-manager using the `posts-production` issuer:



Now you should be able to access your cluster via HTTPS.

Deploying the cluster from a pipeline

Now that we are production ready let's ship the app using a pipeline.

To apply the cluster from a pipeline we need to do the same thing that we did locally:

- Install kubectl
- Install doctl and configure it
- Run `kubectl apply`

But first, we need to solve two problems:

- The passwords are still stored in plain text in the `app-secret.yml` file
- Every deployment uses the `latest` tag

Secrets

First, let's solve the secret problem. Here's the plan:

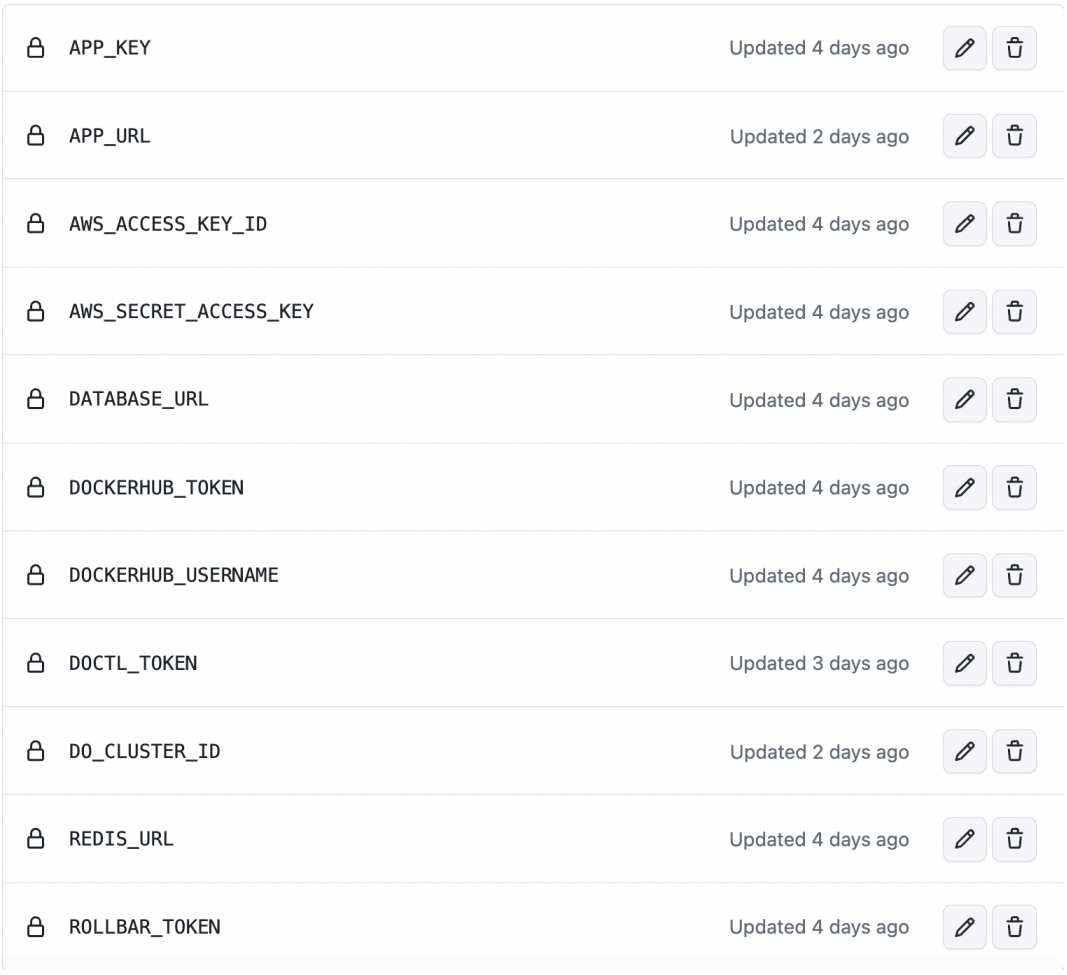
- Change the secret file to have template variables, for example: `APP_KEY: "$APP_KEY"`
- Store passwords and tokens in GitHub secrets (just as earlier)
- In the pipeline create a `.env` file from the `.env.prod.template` file and replace the passwords from the GitHub secret store (just as earlier)
- Export the variables from the file to the current process using the `export` command (as we did with Swarm)
- Substitute the template variables in the `app-secret.yml` file with the real values stored in the process environment

Now we have a valid `app-secret.yml` file containing all the secrets and we're ready to run `kubectl apply`. All of this happens on a runner server which gets destroyed after the pipeline finishes.

After the change, `app-secret.yml` looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: posts
type: Opaque
stringData:
  APP_KEY: "$APP_KEY"

  DATABASE_URL: "$DATABASE_URL"

  REDIS_URL: "$REDIS_URL"

  AWS_ACCESS_KEY_ID: "$AWS_ACCESS_KEY_ID"
  AWS_SECRET_ACCESS_KEY: "$AWS_SECRET_ACCESS_KEY"

  ROLLBAR_TOKEN: "$ROLLBAR_TOKEN"

  HEALTH_CHECK_EMAIL: "$HEALTH_CHECK_EMAIL"
```

I added all of these values to the GitHub secret store:



There's no `DB_USERNAME` or `DB_PASSWORD` only a `DATABASE_URL`.

The next step is to create a `.env` file in the pipeline and substitute `app-secret.yml`:

```
env:
  IMAGE_TAG: ${{ github.sha }}

- name: Prepare secrets
  run: |
    cp .env.prod.template .env

    sed -i "/IMAGE_TAG/c\IMAGE_TAG=$IMAGE_TAG" .env

    sed -i "/DATABASE_URL/c\DATABASE_URL=${{ secrets.DATABASE_URL }}" .env
    sed -i "/REDIS_URL/c\REDIS_URL=${{ secrets.REDIS_URL }}" .env

    # Other env values

    export $(cat .env)

    envsubst < infra/k8s/common/app-secret.yml > tmp_secret
    mv tmp_secret infra/k8s/common/app-secret.yml

    envsubst < infra/k8s/common/app-config.yml > tmp_config
    mv tmp_config infra/k8s/common/app-config.yml
```

We've already seen `sed` commands like these and we've also used the `export $(cat .env)` command with Swarm. It reads the env file and then exports each variable to the current process environment.

`envsubst` is a program used to substitute environment variables in a file. It replaces placeholder values with the current process' environment variables:

```
APP_KEY: "$APP_KEY"

becomes

APP_KEY: "key-1234"
```

Where `key-1234` is the value of `echo $APP_KEY`

This is the generic form of the command:

```
envsubst < input_file > output_file
```

It reads the `app-secret.yml` file, substitutes the values, then it prints the output to `tmp_secret`.

Then the

```
mv tmp_secret infra/k8s/common/app-secret.yml
```

moves the substituted file back to its original path. It effectively overwrites the original files.

I repeat the same thing with `app-config` as well. Not because it contains passwords, but I added a new line:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: posts
data:
  IMAGE_TAG: "$IMAGE_TAG"
   ...
```

Earlier the pipeline write the current commit's SHA to the env file:

```
sed -i "/IMAGE_TAG/c\IMAGE_TAG=$IMAGE_TAG" .env
```

So `envsubst` replaces `$IMAGE_TAG` with the commit SHA. As you might guessed we're going to use this value to run a specific version of the images instead of `latest`. By the way, we did the same thing with Swarm and compose as well.

Image versions

In every deployment file I changed the `latest` tag to `$IMAGE_TAG`:

```
spec:
  containers:
    - name: api
      image: martinjoo/posts-api:$IMAGE_TAG
```

In the pipeline, there's a new step that recursively substitutes these values with the environment variable:

```
- name: Prepare deployment files
  run: |
    for subdir in "infra/k8s"/*; do
      for file in "$subdir"/*.yml; do
        envsubst < $file > tmp
        mv tmp $file
      done
    done
```

It goes through the directories found in `infra/k8s` then it iterates over every `*.yml` file and finally runs `envsubst`. As easy as that. Now on the runner server, every YAML file looks like this:

```
spec:
  containers:
    - name: api
      image: martinjoo/posts-api:c1f3076381606530ba15a78f2abf389384052b90
```

Each image has the current commit SHA as the tag number.

Ship it

And the pipeline's last step is most simple one:

```
- name: Update cluster
  run: |
    kubectl delete -f infra/k8s/migrate
    kubectl apply -R -f infra/k8s
```

As I said earlier, jobs are immutable, they cannot be re-applied. They need to be deleted first. After that, we only need to run the `apply` command.

That's it! You just deployed a Kubernetes cluster into production.

kubectl & doctl

I skipped the first two steps of the pipeline. They are quite boring but pretty important because we need to install `kubectl` `doctl` and we also need to configure the cluster ID. Just as we did on our local machine. Otherwise `kubectl` wouldn't know what cluster it needs to communicate with.

```
- name: Install doctl
  run: |
    wget
https://github.com/digitalocean/doctl/releases/download/v1.94.0/doctl-1.94.0-
linux-amd64.tar.gz
    tar xf ./doctl-1.94.0-linux-amd64.tar.gz
    mv ./doctl /usr/local/bin
    doctl version
    doctl auth init --access-token ${{ secrets.DOCTL_TOKEN }}
    doctl k8s cluster kubeconfig save ${{ secrets.DO_CLUSTER_ID }}
- name: Install kubectl
  run: |
    curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
    curl -LO "https://dl.k8s.io/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
    echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
    chmod +x kubectl
    mv ./kubectl /usr/local/bin
```

```
    kubectl version --output=yaml
```

`DOCTL_TOKEN` is a DigitalOcean API token which you can get from their UI. `DO_CLUSTER_ID` is the cluster's ID which you can get from the URL or their UI.

Every other step is unchanged in the pipeline. They look the exact same as without Kubernetes.

This is the whole `deploy-prod` step:

```
deploy-prod:
  runs-on: ubuntu-latest
  needs: [ "build-frontend", "build-nginx" ]
  steps:
    - uses: actions/checkout@v3
    - name: Install doctl
      run: |
        wget
https://github.com/digitalocean/doctl/releases/download/v1.94.0/doctl-1.94.0-
linux-amd64.tar.gz
        tar xf ./doctl-1.94.0-linux-amd64.tar.gz
        mv ./doctl /usr/local/bin
        doctl version
        doctl auth init --access-token ${{ secrets.DOCTL_TOKEN }}
        doctl k8s cluster kubeconfig save ${{ secrets.DO_CLUSTER_ID }}
    - name: Install kubectl
      run: |
        curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
        curl -LO "https://dl.k8s.io/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
        echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
        chmod +x kubectl
        mv ./kubectl /usr/local/bin
        kubectl version --output=yaml
    - name: Prepare secrets
      run: |
        cp .env.prod.template .env

        sed -i "/IMAGE_TAG/c\IMAGE_TAG=$IMAGE_TAG" .env
```

```
        sed -i "/DATABASE_URL/c\DATABASE_URL=${{ secrets.DATABASE_URL }}"
.env
        sed -i "/REDIS_URL/c\REDIS_URL=${{ secrets.REDIS_URL }}" .env

        sed -i "/AWS_ACCESS_KEY_ID/c\AWS_ACCESS_KEY_ID=${{
secrets.AWS_ACCESS_KEY_ID }}" .env
        sed -i "/AWS_SECRET_ACCESS_KEY/c\AWS_SECRET_ACCESS_KEY=${{
secrets.AWS_SECRET_ACCESS_KEY }}" .env

        sed -i "/ROLLBAR_TOKEN/c\ROLLBAR_TOKEN=${{ secrets.ROLLBAR_TOKEN }}"
.env

        sed -i "/APP_KEY/c\APP_KEY=${{ secrets.APP_KEY }}" .env

        export $(cat .env)

        envsubst < infra/k8s/common/app-secret.yml > tmp_secret
        mv tmp_secret infra/k8s/common/app-secret.yml

        envsubst < infra/k8s/common/app-config.yml > tmp_config
        mv tmp_config infra/k8s/common/app-config.yml
    - name: Prepare deployment files
      run: |
        for subdir in "infra/k8s"/*; do
          for file in "$subdir"/*.yml; do
            envsubst < $file > tmp
            mv tmp $file
          done
        done
    - name: Update cluster
      run: |
        kubectl delete -f infra/k8s/migrate
        kubectl apply -R -f infra/k8s
```

Using `$ENV_VAR` templates in configs and `envsubst` in the pipeline is not the best option we have. We can make the process seamless with Helm templates. In a future edition of this book I'm going to add a new chapter dedicated to Helm.. I'll notify you when it comes out (before the end of 2023).

Monitoring the cluster

Monitoring a Kubernetes cluster is a pretty big topic. Unfortunately, I'm not an expert but I can show you some options. If this is your first time learning Kubernetes and you want to use it effectively my advice is not to go down on the monitoring rabbit whole for a while. Just choose a SaaS product that does everything for you.

middleware.io is one of these tools. It offers the following solutions:

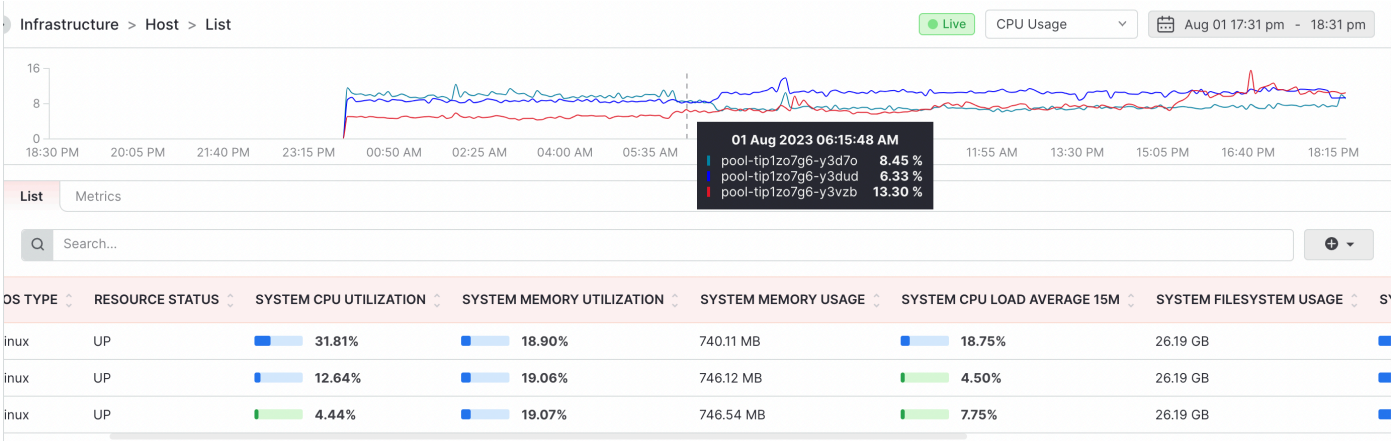
- Infrastructure monitoring
- Log monitoring
- Application performance monitoring (APM)
- Database monitoring
- etc

Their pricing is quite good and they offer a limited free forever plan.

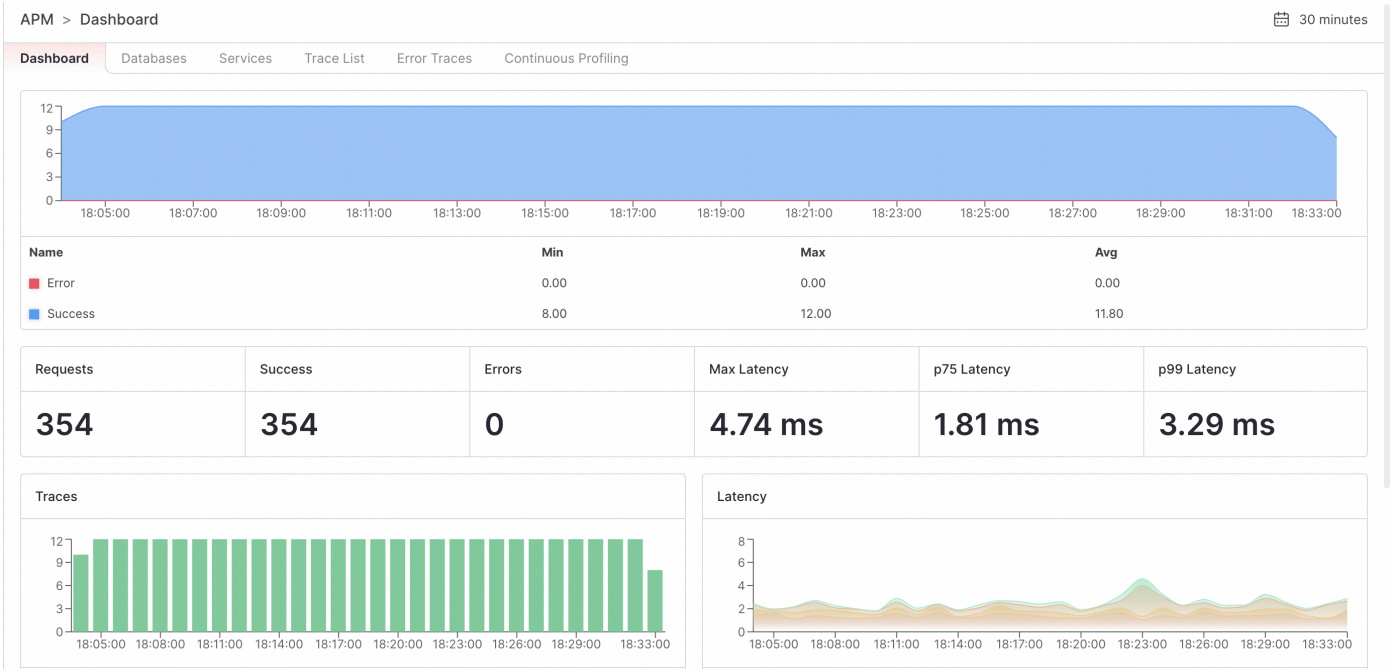
The installation is pretty simple:

```
MW_API_KEY=<your-key> TARGET=https://wwqwg.middleware.io:443 bash -c "$(curl -L https://install.middleware.io/scripts/mw-vision.sh)"

# configure it on your localhost:
kubectl port-forward svc/vision-ui 3000:3000 -n mw-vision
```

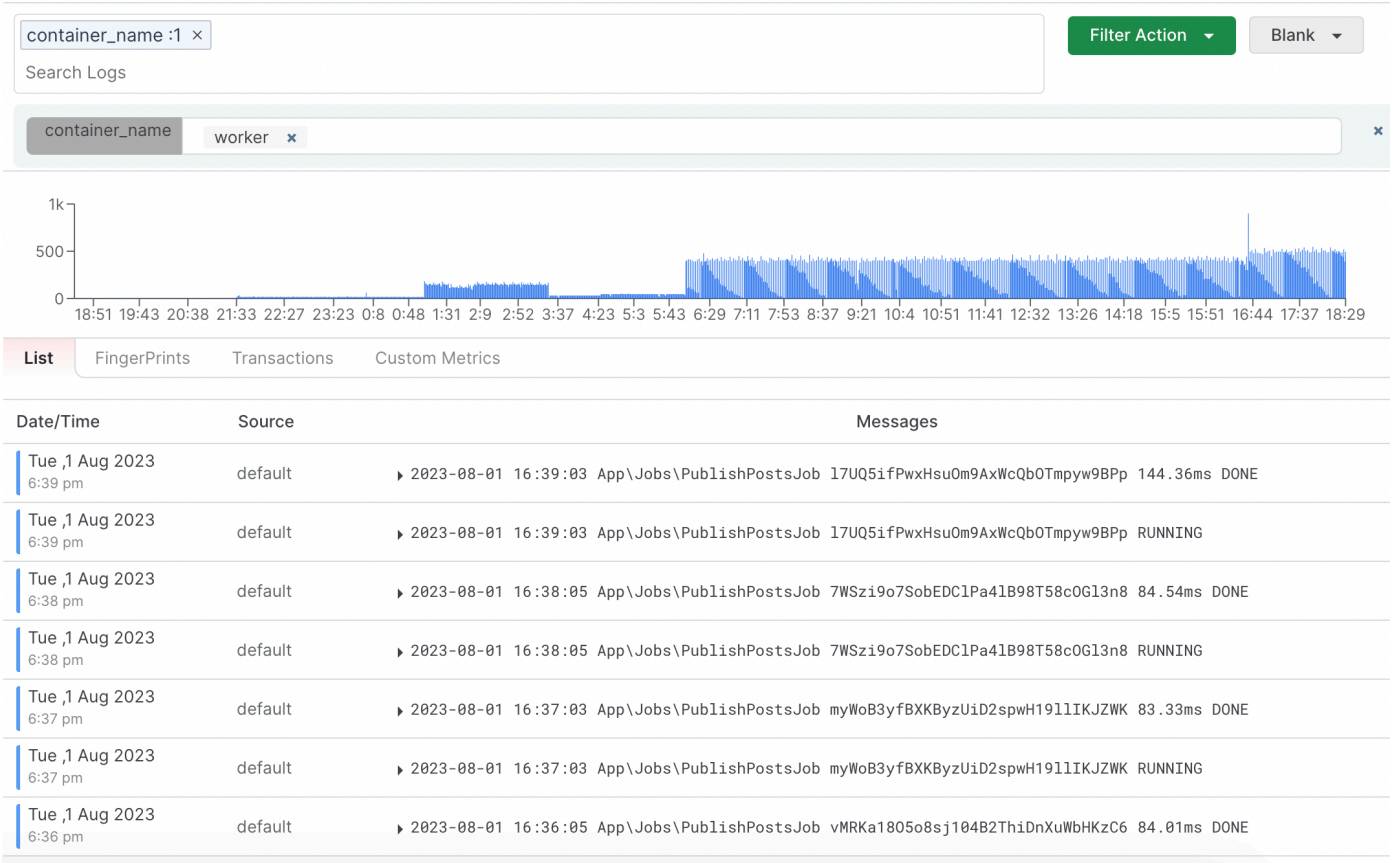
After a 15-minute set up you'll get a nice overview of your infrastructure:



It offers some basic APM metrics out-of-the-box:



And of course you'll have your logs in one central place:

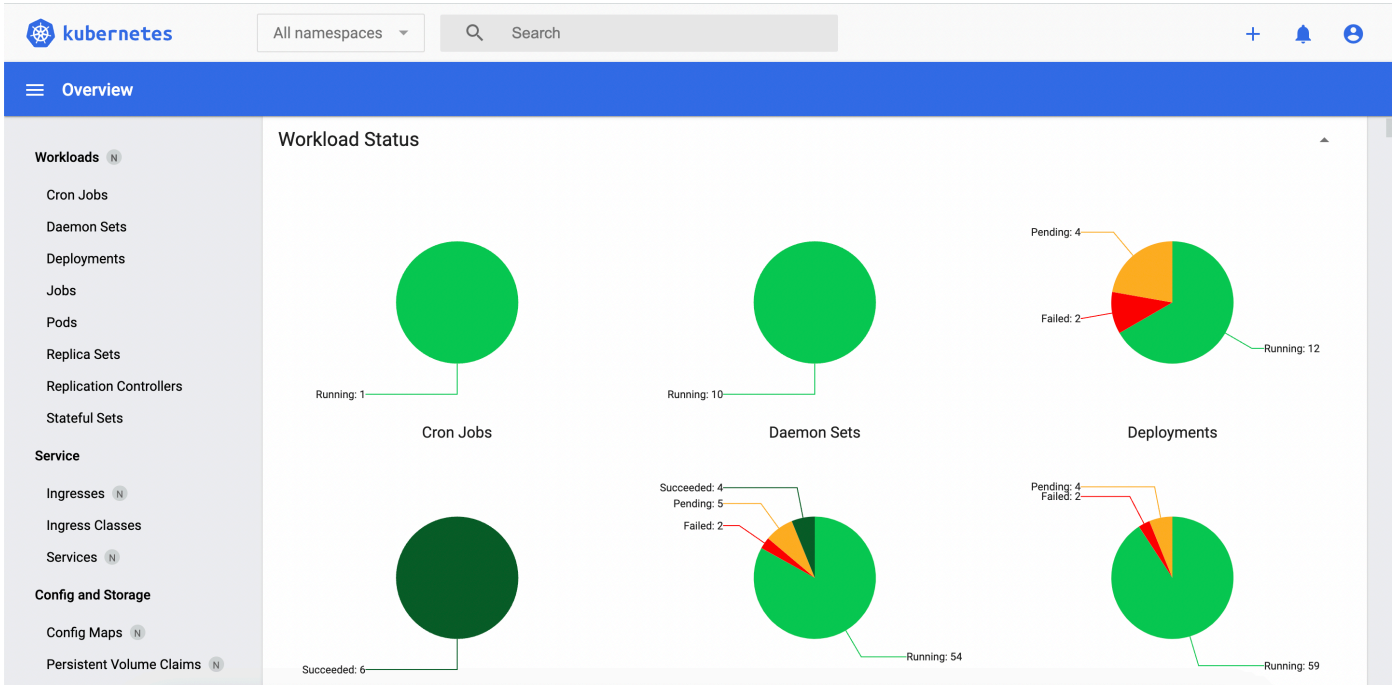


The page above is filtered to show only logs from worker pods.

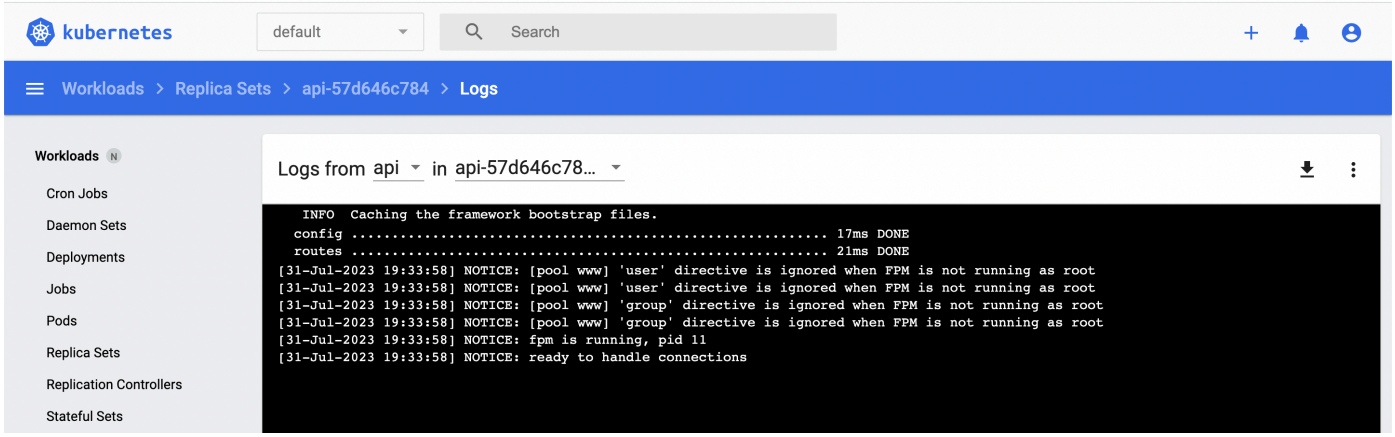
So they offer great solutions and the setup is very easy. Of course, there are other services you can use, for example:

- New relic
- Dynatrace
- Datadog

Another monitoring tool that has out-of-the-box if you use a DigitalOcean (or other cloud provider) cluster is the Kubernetes dashboard. It's a web-based UI add-on for Kubernetes clusters. There's a big "Kubernetes dashboard" button on DO UI. If you click on it you'll see a dashboard such as this one:

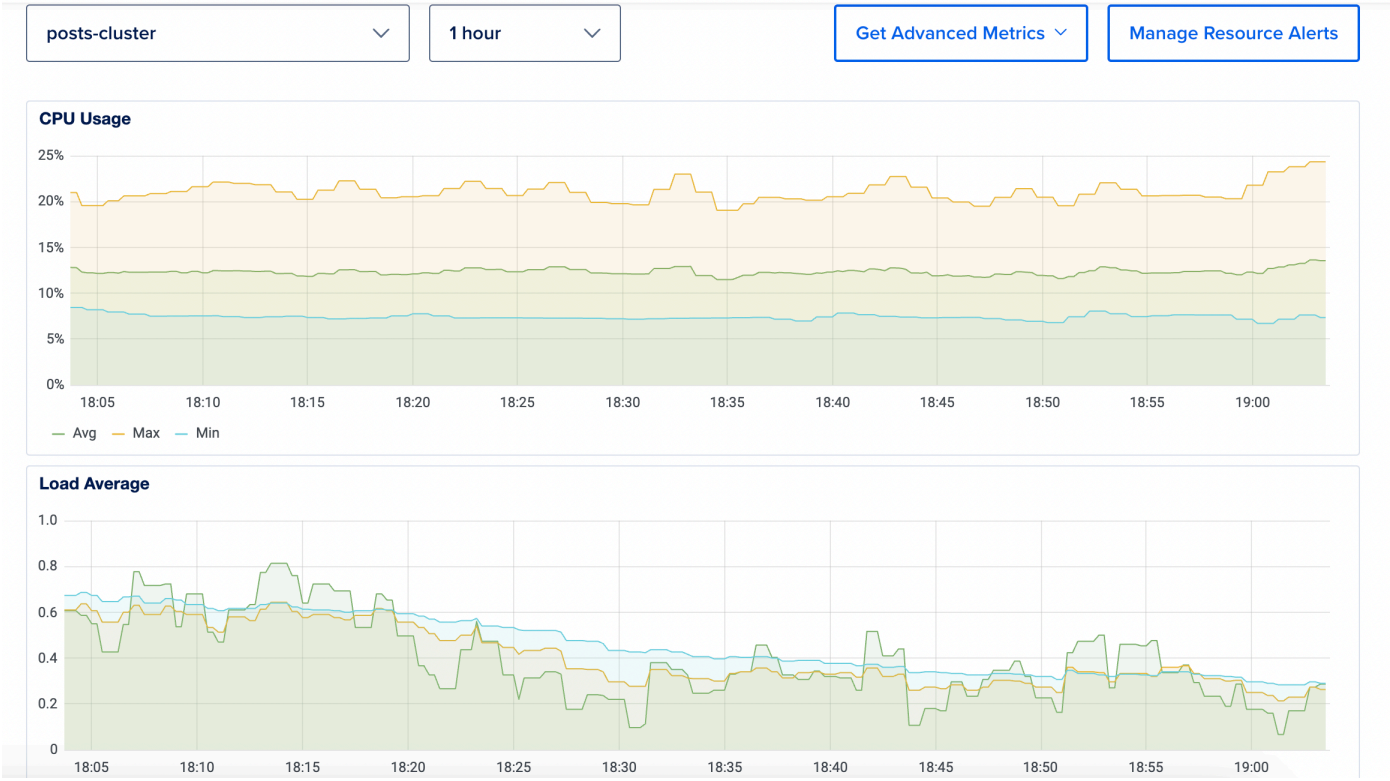


You can instantly see every problematic workflow on one page. You can check out all the running pods, deployment, etc. You can also check out the log of a given pod:



It's a great, free dashboard you have out-of-the-box.

There are lots of other solutions as well. For example, you can install the same stack we used with Docker Swarm (fluentbit, Loki, Grafana). And of course, you have the nice DigitalOcean dashboard as well:


posts-cluster
1 hour
Get Advanced Metrics
Manage Resource Alerts
CPU Usage
25%
20%
15%
10%
5%
0%
18:05
18:10
18:15
18:20
18:25
18:30
18:35
18:40
18:45
18:50
18:55
19:00
Avg
Max
Min
Load Average
1.0
0.8
0.6
0.4
0.2
0
18:05
18:10
18:15
18:20
18:25
18:30
18:35
18:40
18:45
18:50
18:55
19:00

Conclusions

I think Kubernetes it's not that hard. When I first tried it, it was easier than I thought because everyone talked about it as some kind of "magic tool" that solves all my problems. I think it's a bit "over-mystified."

The question is: when should you use it? In my opinion, you need solid knowledge of it before you start running your apps in k8s clusters. Please don't move your company's infrastructure into k8s clusters just because you read this book. Start with something really small and get some real-world experience. Better if you have reliable DevOps guys with experience. If you don't have a dedicated DevOps team, I think Docker Swarm can be a better start.

Thank you very much for reading this book! If you liked it, don't forget to Tweet about it. If you have questions you can reach out to me here.