Deploying Applications

In this lesson, we will deploy an application to our cluster hosted on AWS.

WE'LL COVER THE FOLLOWING

Remoteness

Deploying resources to a Kubernetes cluster running in AWS is no different from deployments anywhere else, including Minikube. That's one of the big advantages of Kubernetes, or of any other container scheduler. We have a layer of abstraction between hosting providers and our applications. As a result, we can deploy (almost) any YAML definition to any Kubernetes cluster, no matter where it is.

It gives us a very high level of freedom and allows us to avoid vendor locking. Sure, we cannot effortlessly switch from one scheduler to another, meaning that we are "locked" into the scheduler we chose. Still, it's better to depend on an open source project than on a commercial hosting vendor like AWS, GCE, or Azure.

> ℹ We need to spend time setting up a Kubernetes cluster, and the steps will differ from one hosting provider to another. However, once a cluster is up-and-running, we can create any Kubernetes resource (almost) entirely ignoring what's underneath it. The result is the same no matter whether our cluster is AWS, GCE, Azure, on-prem, or anywhere else.

Deploying Resources

Let's get back to the task at hand and create `go-demo-2` resources.

```
cd ~/k8s-specs

kubectl create \
    -f aws/go-demo-2.yml \
    --record --save-config
```

We moved back to the repository's root directory, and created the resources defined in `aws/go-demo-2.yml`.

The **output** is as follows.

```
ingress "go-demo-2" created
deployment "go-demo-2-db" created
service "go-demo-2-db" created
deployment "go-demo-2-api" created
service "go-demo-2-api" created
```

Next, we should wait until `go-demo-2-api` Deployment is rolled out.

```
kubectl rollout status \
    deployment go-demo-2-api
```

The **output** is as follows.

```
deployment "go-demo-2-api" successfully rolled out
```

Finally, we can validate that the application is running and is accessible through the DNS provided by the AWS Elastic Load Balancer (ELB).

```
curl -i "http://$CLUSTER_DNS/demo/hello"
```

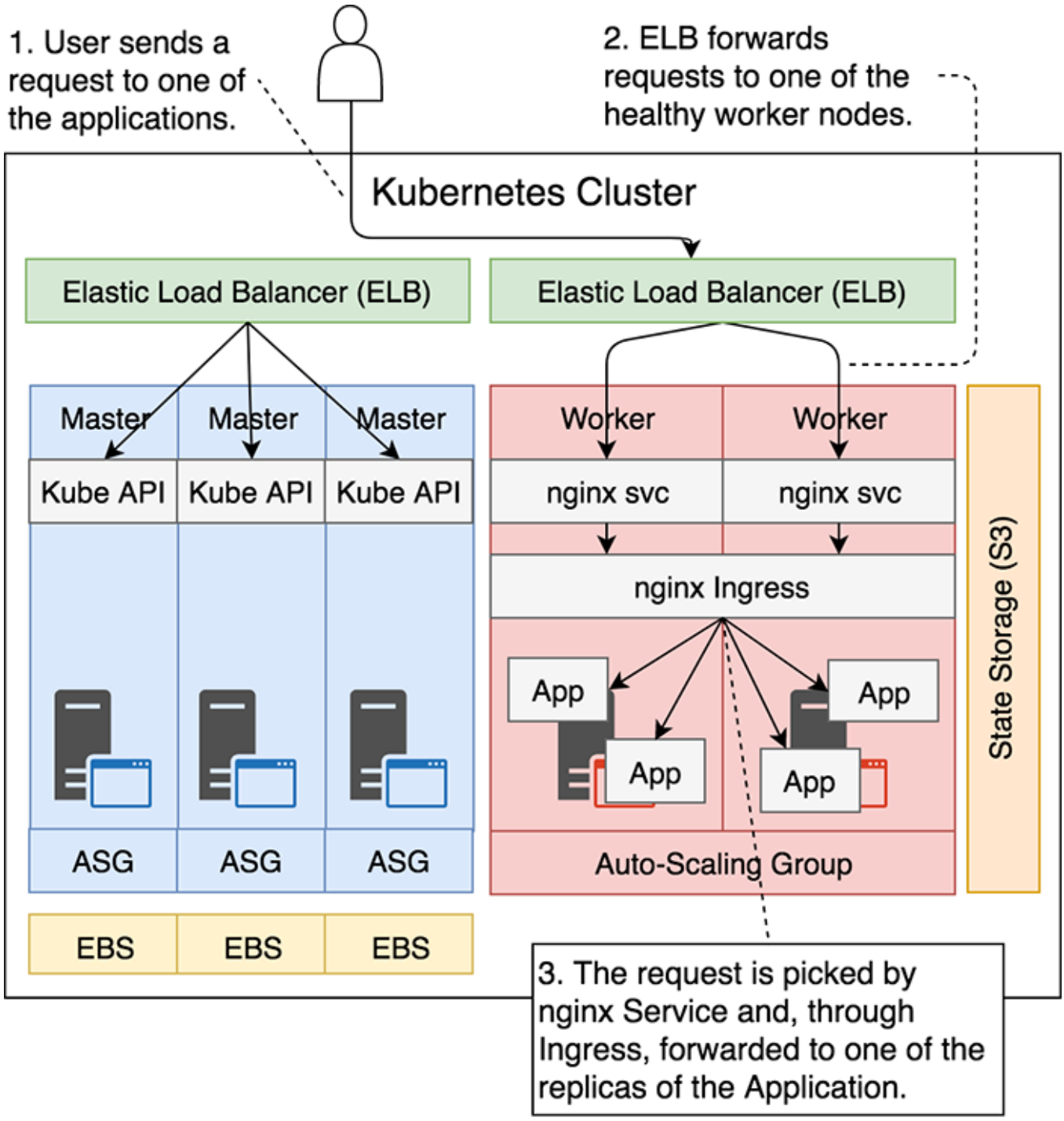
We got response code `200` and the message `hello, world!`. The Kubernetes cluster we set up in AWS works!

Sequential Breakdown of the Process

- When we sent the request to the ELB dedicated to workers, it performed round-robin and forwarded it to one of the healthy nodes.

- Once inside the worker, the request was picked by the nginx Service

- Once inside the worker, the request was picked by the nginx Service, forwarded to Ingress, and, from there, to one of the containers that form the replicas of the `go-demo-2-api` ReplicaSet.



Load balancer behind Kuberentes worker nodes

It might be worth pointing out that containers that form our applications are always running in worker nodes. Master servers, on the other hand, are entirely dedicated to running Kubernetes system.

It's possible to create a cluster in the way that masters and workers are combined into the same servers, just as we did with Minikube. However, that is risky, and we're better off separating the two types of nodes. Masters are more reliable when they are running on dedicated servers. Kops knows that.

and it does not even allow us to mix the two.

In the next lesson, we will explore the high-availability and fault-tolerance of our cluster.